Colimit Library for Graph Transformations and Algebraic Development Techniques

vorgelegt von
Diplom-Informatiker
Dietmar Wolz

Vom Fachbereich 13 – Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor-Ingenieur
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: Prof. Dr. Stefan Jähnichen
Berichter: Prof. Dr. Hartmut Ehrig
Berichter: Prof. Dr. Martin Gogolla

Tag der wissenschaftlichen Aussprache: 18 Februar 1998

Berlin 1998

D 83

Contents

1 Introduction

1.1 Motivation

Software design always involves structuring of a system into subsystems, objects and the relations between them. Usually this structure is visualised by some kind of graphical representation, for example by class diagrams in an object oriented design method [RBP+91a, Boo94a, BRJ+97] or by entity-relationship diagrams in the area of semantic data modeling [HK87]. Additional requirements, detection of errors or migration to a different environment forces changes on the software system. To preserve consistency between design and implementation the diagrams should be adapted to reflect the new design. The embedding of each diagram into its context imposes constraints for the applicability of diagram transformations. A graphical design tool could support the software engineer in following these constraints by providing a predefined set of transformation rules which it can perform automatically, thereby taking account of the context relations.

Tool support presupposes some kind of formal description which clarifies the notions diagram, embedding and transformation. The concept of correctness makes only sense in the context of a formal semantical basis. The literature provides us with several different approaches [CL95, HK87], which all have their specific advantages. In this thesis we will focus on methods based on category theory, which are distinguished by their elegance and high level of abstractness. Category theory is a mathematical theory developed to generalise mathematical concepts from different areas. Similar to object oriented design methods it uses diagrams to visualise the relations between (mathematical) objects. Category theory can be regarded as a theory of object relations called morphisms. Diagrams have a formal mathematical definition in category theory. To distinguish them from class and entity-relationship diagrams we will use the notion scheme for the latter ones. From the viewpoint of software engineering there is a strong relation to the concept of information hiding, since category theory abstracts from the inner structure of the objects. A high degree of abstraction usually results in a shift of workload from the user, in this case the software designer, to the tool developer. The question is, if and how the theoretical constructions can be mapped on an implementation.

In their pioneering work on computational category theory Rydeheard and Burstall [RB88] showed, that such a mapping indeed exists. In exploiting the enormous expressional power of the functional language ML, they provide an elegant implementation of many constructions from category theory. This book is of great value not only for a tool builder, but for all computer scientists, who want to learn the mechanisms and ideas behind category theory. They can investigate and execute the code to see how the constructions work. A nice idea from [RB88] is, that the implementations are derived directly from constructive proofs. That means, the proofs were designed with the implementation in mind. That way additional correctness proofs become superfluous. But as stated in their remarks about possible applications, because of its inefficiency, this implementation is not intended for practical usage. A tool builder can neither directly use the code nor the algorithms. The inefficiency is not only caused by the chosen functional programming language, but mainly because the algorithms are designed to be generally applicable for

all categories.

Practical experience with the implementation of tools based on category theory [TB94, EC96] has shown, that efficiency is achievable if only special constructions in set-based categories are treated. But for each tool and algorithm a separate correctness proof is needed. And there are some constructions demanded from applications like scheme or graph transformations, where an efficient implementation is hard to find. This imposes restrictions for the applicability of these tools. These problems are caused by the fact, that there is still a gap between the theory from [RB88] and the practical demands of the tool builder. Obviously tool development would be considerably simplified if this gap could be closed.

What we need is a specialised theory restricted to categories which are relevant for the applications we have in mind. But this theory should be general enough to cover also the cases which are hard to implement. Constructive proofs should guide the implementation of a reusable library of data structures and algorithms operating on diagrams. This library should, whenever possible, avoid programming language specific features so that tool developers are not forced to use a specific implementation language. Finally, the algorithms should not only have linear complexity both in time and space, their efficiency should be comparable to optimised algorithms for special constructions. This means full scalability in all dimensions: The number of relations and the size of diagrams and objects.

For the treatment of structuring mechanisms there is one most important concept in category theory: The colimit construction. In applications related to computer science, the colimit separates elements in different components and identifies elements which are connected via their relations. The colimit construction is not only relevant for scheme transformations. Recently current object oriented modeling methods like the Unified Modeling Language [BRJ+97], which is accepted as international standard by the OMG, integrate structural and behavioural aspects with extensions of state machine formalisms. Category theory, and specially the colimit construction, is suitable for both aspects providing an unifying formal framework for object oriented design. Other application areas are:

- Structuring and refinement of formal specifications [SJ95, Cla93, BG80, EM85, EM90] based on colimits in the category of signatures.
- Alternative definition of the operational semantics of functional logic programming languages [CW94] based on the category of jungles (special hypergraphs representing terms).
- Algebraic development techniques, which aim at an extension of structuring and development techniques from algebraic specifications to other specification techniques. For a survey see [EGW97]. Examples are:
  - Horizontal and vertical structuring of statecharts [EGKP97], a visual specification technique for concurrent and reactive systems introduced by D. Harel in [Har87], see also [HG95]. A variation of the statechart formalism is part of the Unified Modeling Language (UML) [BRJ+97].

- Description of dynamic state transitions of parallel and distributed systems using graph grammars [EHK+96, CMR+96, Tae96a] or algebraic high level nets [PER95, Pad96] based on the category of graph structures.
- Action Nets, which combine elements from statecharts, Petri Nets and graphs [EGP97]
- Dynamic abstract data types [EO94, AZ95], a general integration framework including data type, process and time oriented systems.

It could even serve as the basis for the formalisation of parameterisation in programming languages like C++, Eiffel or Ada. For all these methods to be applicable, tool support is strongly required. Existing tools [SJ95, TB94, EC96, CEW93] do not share a common implementation of the colimit construction, they even do not share common algorithms. But there is not only a problem at the implementation level. The lack of an efficient reusable colimit algorithm in the past lead to alternative semantic definitions of the specification languages CLEAR and ACT ONE [San84, Cla88] to enable tool development for them. This caused a significant overhead, because additionally the equivalence of both kinds of semantics had to be proven.

1.2 Aims of this Thesis

As we will see later, all the applications mentioned above can be based on colimit computations in only two categories, namely signatures and graph structures [Löw93a], that is, algebras with unary operations and partial morphisms generalising comma categories over sets. Specialisation of the cocompleteness proof to these categories leads to an efficient, modular and incremental algorithm. The aim of this thesis are the following three main results:

1. A new constructive proof of cocompleteness for sets, signatures and graph structures with partial morphisms (theorems 3.1.5, 3.2.2 and 4.2.8).

2. Proofs establishing (almost) linear complexity of the colimit constructions in all these categories (theorems 3.1.6, 3.2.4 and 4.2.9).

3. A library for colimit computations in these categories implemented in Eiffel, JAVA and C++. The algorithms are directly derived from the constructive proofs mentioned above (see chapter 5).

Additionally the application of the colimit library in different areas both in the context of specification languages and graph transformations is outlined (section 3.3 and 4.4). Performance tests and comparisons on several different development platforms establish the practical applicability of the colimit library (see section 5.10). The long-term goal of this research is to simplify the development of tools related to category theory and colimit constructions. The colimit library is already used in the ALPHA library [EC96] supporting scheme transformations and in the new version of the graph manipulation system AGG [LB93].

1.3 Related Developments

We already mentioned the book on computational category theory from Rydeheard and Burstall [RB88], the pioneering work in the field. There colimit constructions are implemented using an inefficient modularisation approach, following the structure of the usual cocompleteness proof (see next section). This is one of the reasons, this approach failed to satisfy the performance needs of tool builders. But this was never the intention of this book. It succeeds very well in illustrating the concepts of category theory to computer scientists and shows that they are implementable. We adopted the idea to provide a constructive cocompleteness proof guiding the implementation. But our proof is specialised for the requirements of tool development. It works only with set-based categories, but reaches a level of efficiency comparable to algorithms optimised for special constructions like coequalizers and pushouts. [RB88] covers also limits, the dual construction to colimits. In this thesis we focus on colimits, because we are mainly interested in the support of structuring mechanisms and transformations rules. For this application areas limits play a secondary role.

Another important contribution is the work from Y. V. Srinivas, J. L. McDonald and R. Jüllig on the SPECWARE system [SJ95, SM96]. This system demonstrates the practical applicability of category theory based structuring mechanisms for formal specifications in a commercial application. SPECWARE supports the modular construction of formal specifications and their stepwise and component wise refinement into executable code. It provides a visual interface to a suite of composition and transformation operators for building specifications, refinements and code components based on category theory, sheaf theory, algebraic specification and logics. As noted in section 5 from [SM96], category theory was of great help in structuring the implementation. Enhanced reusability enabled the realisation of important parts within a few days. The operators are implemented in FRANZ-LISP, a dialect of the functional programming language LISP. Since SPECWARE is a commercial tool, its implementation is not publicly available. The implementation of SPECWARE uses an algorithm for the colimit of signatures very similar to the one from the colimit library (chapter 5). This algorithm, already described by the author in [Wol96], was independently discovered by the SPECWARE team. This thesis extends the algorithm (and the corresponding cocompleteness proof) such that it is applicable to scheme and graph transformations.

M. Löwe established the notion of algebraic graph structures [Löw93b] and initiated the development of several related tools, for instance the ALPHA library [EC96] and the AGG system [LB93] mentioned before. These tools contained in their initial versions implementations of special colimits, namely coequalizers, coproducts, initial objects and pushouts. Currently our colimit library replaces these implementations which results in an improved flexibility, because now all colimits are supported and several restrictions for the applicability of the old algorithms were released. [Löw93b] already presents a description of an algorithm for pushouts in graph structures. But this algorithm is very complex and hard to implement.

The first ideas how colimit computations in graph structures may be modularised were shown by M. Korff in [Kor96]. They had some influence on the theory presented here,

but this work is at a much higher level of abstraction and has no relation to a concrete implementation.

In contrast to the related developments we present an object oriented library for arbitrary colimits for signatures and graph structures in three of the most popular object oriented programming languages (Java, C++ and Eiffel). We combine the idea of an implementation guided by a cocompleteness proof from [RB88] with the efficiency of algorithms in existing tools optimised for special colimits like pushouts or coproducts.

1.4 The Colimit Library

In this section we present an overview on different aspects of the colimit library.

1.4.1 Suitable Programming Languages

Instead of forcing the user of the colimit library to write his application in a specific programming language we chose a more general approach. Suitability of a programming language is related to several application dependent requirements like efficiency, portability, availability of programming environments, specific libraries and programming skills. We designed our implementation in such a way, that it is easily portable to many programming languages by restricting the use of specific libraries and language features.

Since we use techniques from generic programming [DRM89] and object oriented design [Boo94b] we require support of classes and inheritance. Generics (templates) and static type checking are desirable, but not essential. We provide example implementations in C++, Eiffel and JAVA, discuss their differences and compare their performance. Ports to ADA 9x, Sather, Smalltalk and Objective C can easily be done, to name some alternative languages. One of the most important advantages of C++, Eiffel and JAVA is, that there are free compilers available for them on a huge number of platforms.

Tests and comparisons have shown huge performance differences between the different languages and development platforms (see section 5.10), specially JAVA has here some problems. But it is very likely that these will decrease in the future, since there are many ongoing projects improving the efficiency of JAVA implementations.

1.4.2 Modularisation of the Colimit Algorithm

How should the colimit computation be structured? There are two different ideas:

- Following the usual constructive proof of *finite cocompleteness* [1] the computation may be based on some elementary colimit constructions, namely *pushouts* and *coproducts* [2] (see for instance [RB88]). Arbitrary colimits may be constructed solely via

[1] Finite cocompleteness of a category means that the colimit of an arbitrary finite diagram in that category exist.

[2] A pushout is an elementary colimit construction suitable for the description of simple graph transformations and the actualisation of parameterised data types. It is the colimit of a diagram of shape $A \leftarrow B \rightarrow C$. In our applications the coproduct represents the concept of a disjoint union. It corresponds to a diagram of shape A, B without any arrows.

these easy to compute constructions. As will be discussed later, unfortunately this modularisation leads to an unnecessary high complexity of the whole computation.

- The alternative approach modularises the computation according to the structure of the objects in the category. *Comma categories* are a very general concept for building categories with structured objects from simpler ones. The constructive cocompleteness proof for comma categories can guide the implementation of a general colimit algorithm.

Because it can be realized without any performance penalty we have chosen the second approach. Since comma categories are not general enough to describe the categories needed for our applications, we generalise them thereby preserving the modular structure of the colimit computation.

1.4.3 Signatures

Colimit computations in the category of signatures are relevant for structuring mechanisms for algebraic specification languages. Algebraic specifications are signatures augmented with axioms and structuring mechanisms like enrichment, parametrisation and modularisation [EM85, EW86, EM90]. A *signature* $SIG = (S, OP)$ introduces the sorts S and operations OP used in the axioms of the specification together with an arity function mapping operation symbols to sort lists denoting their functionality.

Since the category of signatures $\mathcal{SIG}$ may be expressed as comma category over $\mathcal{SET}$, colimit computations for $\mathcal{SIG}$-diagrams can be built on the computations for $\mathcal{SET}$-diagrams. The category of algebraic specifications will not directly be supported by our library, since there is no unique kind of axioms in different specification languages. If only a syntactical transformation of the axioms according to the corresponding signature morphisms is needed, it can easily be built using the computed $\mathcal{SIG}$-colimit. If specification morphisms in addition require a semantical transformation of the axioms, theorem proving techniques, as realized for instance in the SPECWARE system [SJ95], have to be used. We do not cover that issue in this thesis. But semantical transformations are based on the syntactical colimit construction on signatures as provided by our library.

1.4.4 Partiality

The introduction of partial mappings, called partial morphisms in category theory, is motivated by several applications like scheme transformations, algebraic specification languages and graph grammars. Partiality expresses the concept of deletion during a transformation step.

Partial morphisms may be represented as spans of total morphisms $L \leftarrow K \rightarrow R$ as in the *double pushout approach* for graph transformations [CMR+96]. Here a basic derivation step is modelled by two gluing diagrams (pushouts) with total morphisms. Though advantageous in some applications the direct treatment of partial graph morphisms as done in the *single pushout approach* [Löw93b], from an implementational view, simplifies

the representation and colimit computation of general diagrams in the category of graphs with partial morphisms. In [EHK+96] a comparison of both approaches can be found.

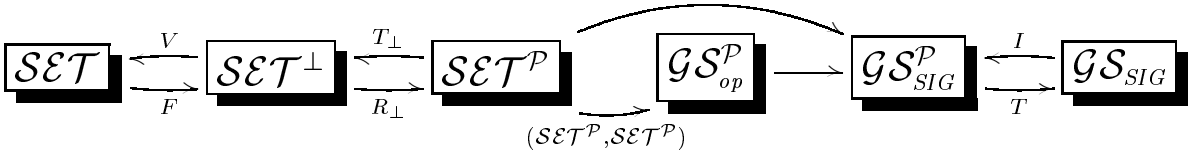
Support of partiality adds a level of abstraction to the diagrams since it hides the "inner structure" of partial morphisms. If we try to hide structure through abstraction, usually the complexity is shifted somewhere else. Indeed partiality adds some complexity to colimit computations which can be seen by the very complex and hard to implement description of pushouts of graph structures in [Löw93b]. Often the rule representing morphisms are required to be injective to simplify their realization. This thesis presents a solution to this problem by providing a library of efficient general colimit algorithms, such that a tool designer can take colimits as given. Since colimits have an intuitive understandable semantics neither the tool designer nor the user of the tool really needs to fully understand the theory behind their realization as presented in this thesis.

1.4.5 Graph Structures

We can view ordinary graphs as algebras to a signature with two sorts *node* and *edge* and two unary operations *source* and *target*. We know that the category of graphs with partial graph morphisms is cocomplete, but not the category of algebras with partial morphisms to a signature $SIG = (S, OP)$ containing constants or some operations taking more than one argument. But a restriction to signatures only with unary operations (like the *source* and *target* operations in graphs) enables a general proof of the cocompleteness of the corresponding category [Löw93a]. These special algebras are called *graph structures* since they form a generalisation of ordinary graphs.

Hyper graphs, attributed graphs, terms, even entity-relationship diagrams [CL95] can be represented as graph structures. Since they are the most general algebraic structures with partial morphisms where the corresponding category is cocomplete, and because they are relevant in most of our application fields, we will focus on them in our theoretical considerations. That means, we will provide a new cocompleteness proof, constructive enough not only to guide the implementation, but also enabling a proof of the complexity of the algorithm. The category of graph structures to a signature $SIG = (S, OP)$ with partial morphisms is denoted as $\mathcal{GS}_{SIG}$. Note that in this category the operations of a graph structure are total as the *source* and *target* operations in a graph.

The problem for the cocompleteness proof is, that graph structures are not only a generalisation of comma categories over sets, but also a specialisation, because we have partial morphisms together with total operations. We solve it by a generalisation of graph structures thus allowing also the operations to be partial. The corresponding category of graph structures with partial operations is denoted as $\mathcal{GS}^{\mathcal{P}}_{SIG}$.



The picture above presents an overview of the structure of both the cocompleteness proof and the implementation of the colimit algorithm for graph structures. Sets with partial

morphisms $\mathcal{SET}^{\mathcal{P}}$ are represented by pointed sets $\mathcal{SET}^{\perp}$ and form the basis of the colimit computation in the category of simple graph structures with partial morphisms and one single partial operation *op* called $\mathcal{GS}^{\mathcal{P}}_{op}$. This category is similar to the *arrow category* [3] of morphisms $(\mathcal{SET}^{\mathcal{P}}, \mathcal{SET}^{\mathcal{P}})$, so the colimit can be computed independently for the carrier sets and then for the operation. Next we regard a diagram $\mathcal{GS}^{\mathcal{P}}_{SIG}$ with $SIG = (S, OP)$ as a set of $\mathcal{SET}^{\mathcal{P}}$–diagrams, one for each sort $s \in S$, together with a set of $\mathcal{GS}^{\mathcal{P}}_{op}$–diagrams, one for each operation $op \in OP$. In this view the carrier sets are redundantly represented, but we will show, that this redundancy doesn't lead to an inconsistency with respect to colimits and their universal morphisms. Thus we can simply put together the separately computed colimits. Note that in $\mathcal{GS}_{SIG}$, this technique is not applicable, because there the operations determine the colimits computed for the carrier sets.

Finally we apply a colimit preserving functor which translates the partial graph structure into a total one, obtaining the deserved $\mathcal{GS}_{SIG}$–colimit.

Instead of directly supporting general graph structures we decided to implement the colimit algorithm for ALPHA algebras, special graph structures well suited for implementation purposes. General graph structure diagrams can be expressed as diagrams of typed ALPHA algebras. The typing is represented via typing morphisms and the typing of the resulting colimit may be obtained via unique extension of the untyped colimit easily constructed with the help of the colimit library. For ALPHA algebras there exists a powerful general class library [EC96]. The colimit library was integrated into this ALPHA library extending its functionality by general colimit computations.

1.4.6 Double and Single Pushout Approach

As mentioned above, in the *double pushout* approach to graph transformations [CMR+96] partial morphisms are represented as spans of total morphisms $L \leftarrow K \rightarrow R$ where K contains the non–deleted part of the left side L of the rule, the "gluing items" preserved by the rule. Viewed from their applications, the main differences between both approaches is related to their treatment of ambiguous deletions and dangling edges. In the double pushout approach such derivations are not allowed, in the *single pushout* approach additional deletions are performed "repairing" the resulting colimit such that there are no ambiguities and dangling edges. In different applications one or the other behaviour may be desirable, so we decided to support both of them.

The conditions related to ambiguous deletions rsp. dangling edges are called *identification* rsp. *dangling* condition. The colimit library provides routines to check those conditions enabling the applications to decide whether a rule is applicable or not. So we can support both kinds of behaviour by only implementing the single pushout approach. This strategy simplifies the implementation but also has some disadvantages:

- To support the double pushout approach a transformation of the span representations of the rules into partial morphisms is required.

[3] Arrow categories are special comma categories. Their objects are the morphisms of the category they are based on.

- The identification rsp. dangling conditions are checked by computing the $\mathcal{GS}^{\mathcal{P}}_{SIG}$–colimit and then inspecting the result. So internally the rule is partly applied even if the result of the test is negative. This may cause some overhead. But keep in mind that both the colimit computation in $\mathcal{GS}^{\mathcal{P}}_{SIG}$ and the following test is performed very efficiently in linear time. The development of an alternative algorithm for checking these conditions in linear time is not trivial.

- The search of applicable rules may be optimised such that it is performed simultaneously for different rules and occurrences. Here our implementations of the identification rsp. dangling conditions may not be suitable.

- Non–injective $L \leftarrow K$ morphisms supporting the concept of duplication cannot be expressed via equivalent partial morphisms. Fortunately up to now most theoretical concepts developed for the double pushout approach require the injectivity of $L \leftarrow K$ morphisms.

On the other hand, our checks of the identification rsp. dangling conditions are not restricted to pushouts but are realized for arbitrary colimits, enabling direct support of parallel or distributed graph transformation approaches as described for instance in [Tae96a].

Alternatively, the colimit library could be used only to compute the right hand side pushout of the transformation rules. Then the transformation of the rules into a representation using partial morphisms is avoided, but to apply a rule in addition we have to compute the pushout complement of the left hand side pushout and to check the application conditions.

1.5 Organisation of the following Chapters

The following chapters are structured as follows:

- *Chapter 2* presents the theoretical foundations of this thesis. It introduces the main concepts of category theory and provides basic notations, definitions and proofs. A reader familiar with category theory may skip parts of this chapter.

- *Chapter 3* analyses the category of sets $\mathcal{SET}$ and comma categories over $\mathcal{SET}$. We show a new constructive cocompleteness proof for this categories and prove the complexity of the corresponding colimit algorithm. Then its application in the area of algebraic specifications is outlined.

- *Chapter 4* extends the theory to graph structures with partial morphisms. Again we prove cocompleteness and the complexity of the algorithm and show possible applications for data base scheme transformations, high level petri nets and the simulation of an abstract machine executing functional logic programs.

- *Chapter 5* covers the relation of our theoretical investigations with their practical realization. The structure of the colimit library and its interface with possible applications is presented. We motivate our choice of concrete data types to represent

the diagrams in the different categories. We compare the suitability of possible implementation languages and present benchmark results which not only illustrate the efficiency of our colimit computations but also allow to compare several languages (C++, Java and Eiffel), compilers, and libraries.

- *Chapter 6* summarises the results and outlines possible extensions.

We presume basic knowledge about category theory [AHS90, BW90, AL91, RB88], algebraic specifications [EM85, EM90, CEW93] and graph transformations [Tae96a, CEL+96, CMR+96, EHK+96]. After completing this thesis, the reader will know about colimits in set-based categories, some of their applications, how they may be efficiently implemented and how the implementation is related to the theory. If he is involved in a software project in the area of algebraic specifications or graph (structure) transformations the reader should be able to use the colimit computations provided by the colimit library. For this purpose also the HTML-documentation should be consulted. The comparison of programming languages directly supported by the colimit library could support the decision for a suitable implementation language.

For readers only interested in theoretical results we should indicate that there exist already alternative cocompleteness proofs for most of the categories covered in this thesis, despite one exception. [4] But although some of these proofs are constructive, they are not of much help for the development of efficient algorithms. The main idea of this thesis is to adapt the theory for the needs of the tool designer. There is more or less a one to one correspondence between our theoretical cocompleteness proofs and the data structures and algorithms used in the implementation.

1.6 Acknowledgements

I like to thank my supervisor Hartmut Ehrig for his constructive support concerning the writing of this thesis. Furthermore, I thank my co-supervisor Martin Gogolla, who agreed to report on this thesis although the time was very short.

Special thanks to Ingo Classen, Uwe Wolter and Reiko Heckel providing me with a lot of valuable hints, comments and corrections. Thanks to Sebastian Erdmann, Michael Rudolf and Claudia Ermel for their cooperation integrating the colimit library into their software projects. Finally I thank Martin Grosse Rhode for proof reading parts of this thesis and his numerous suggestions.

[4] [Löw93a, Kor96] restrict their cocompletess proofs to hierarchical graph structures. Since this constraint has negative consequences for practical applications, we generalise the proof for arbitrary graph structures.

2 Theoretical Foundations

After a brief introduction in category theory we will present the basic definitions and notations used in this thesis together with some fundamental propositions related to the decomposition of morphisms in diagrams and the totalisation of graph structures.

2.1 Category Theory

In [EM45] category theory was introduced to establish an uniform framework for different areas of mathematics, in particular topology and algebra, avoiding redundancy in their theory and proofs. In opposition to set theory, where mathematical objects are characterised by their inner structure, category theory concentrates on their relations. These are structure preserving mappings between objects called morphisms. Objects are characterised by their specific role within the net of their relationships. The idea to describe properties of objects by their relations to other objects leads to an elegant graphical notation of concepts and proofs, which makes category theory not only interesting to mathematicians, but also to computer scientists.

Since category theory abstracts from the inner structure of objects, they can be characterised only up to isomorphism, that means, independently from a particular representation. Of course, at the end, if we want to present the computed results, we have to choose an appropriate representation. But internally, during the computations, it is of no relevance. This property is very useful for optimisation purposes and guided the design of the data structures of the colimit library.

A *category* $\mathcal{C}$ consists of a class of *objects* $Obj(\mathcal{C})$ and a class of *morphisms* between objects $Mor(\mathcal{C})$. A morphism f between objects A and B is denoted as $f : A \to B$. For each object A there is an *identity* morphism $1_A : A \to A$ and for any two morphisms $f : A \to B$ and $g : B \to C$ there exists their *composition* $g \circ f : A \to C$. The composition is associative and the identities are called so because $f = f \circ 1_A = 1_B \circ f$ for all morphisms $f : A \to B$. The set of morphisms $f : A \to B$ between objects A and B is called *homset* $hom(A, B)$.

As example consider the category of sets called $\mathcal{SET}$. Their objects are the class of all sets, their morphisms are the set of total functions. Another example is $\mathcal{SET}^{\mathcal{P}}$, the category of sets with partial functions. In this thesis we treat only categories derivable from $\mathcal{SET}$ or $\mathcal{SET}^{\mathcal{P}}$ which augment these categories with some structure. These categories cover all applications we have in mind. The relation of these categories with $\mathcal{SET}$ is of crucial importance for the design of the colimit library, since colimit computations for them are based on colimit computations in $\mathcal{SET}$.

Since morphisms play a fundamental role in category theory, it is quite natural to require that mappings between categories should preserve their structure. These structure preserving mappings are called functors. They are given by mappings between the objects and morphisms of two categories. For a functor $F : \mathcal{C} \to \mathcal{D}$ between the categories $\mathcal{C}$ and $\mathcal{D}$ holds $F(f) : F(A) \to F(B)$ for all morphisms $f : A \to B$ in $\mathcal{C}$. Functors preserve identities and composition, that is, $F(1_A) = 1_{F(A)}$ and $F(g \circ f) = F(g) \circ F(f)$ for all objects A and morphisms g, f in $\mathcal{C}$.

Another important concept is that of a canonical construction $F : \mathcal{C} \to \mathcal{D}$ converting objects A canonically from one category $\mathcal{C}$ to another one $\mathcal{D}$. Canonically means, that the construction should be compatible with the morphisms, that is, extendable to a functor, and that the relationships between objects in $\mathcal{C}$ should be preserved. To express this requirement we need an adjoint construction $V : \mathcal{D} \to \mathcal{C}$ in the opposite direction. Preservation of relationships means, that there is a bijection between the homsets $hom(A, V(B))$ and $hom(F(A), B)$ for all objects A in $\mathcal{C}$ and B in $\mathcal{D}$. This requirement determines the canonical construction F up to isomorphism. If we have such an *adjoint* situation we call F *left adjoint* or *free* and V *right adjoint* functor.

The relation between the category of partial graph structures $\mathcal{GS}^{\mathcal{P}}_{SIG}$ and the category of total graph structures $\mathcal{GS}_{SIG}$ (both with partial morphisms) illustrates the idea. We want to convert the computed partial graph representing the $\mathcal{GS}^{\mathcal{P}}_{SIG}$-colimit canonically into a total one representing the corresponding $\mathcal{GS}_{SIG}$-colimit. Left adjoint functors preserve colimits, so we only have to show the left adjointness of our construction to be sure, that the resulting structure is indeed a $\mathcal{GS}_{SIG}$-colimit. Since the text books on category theory provide us with several equivalent characterisations (not only the one given above), we can choose the one which is easiest to prove, following the motto: "Let the mathematicians do the hard work for us". Of course we must not forget to prove the functor properties, which indeed is the most difficult task in this particular case.

2.2 Diagrams and Colimits

Since the fundamental notions of category theory have a well-established meaning in the literature [AHS90, BW90, AL91] only the definitions relevant for the implementation of the colimit library are given explicitly.

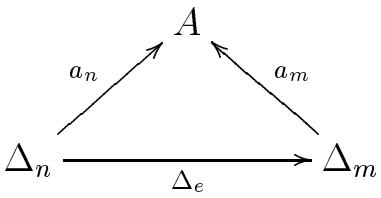
DEFINITION 2.2.1 (GRAPH)
A graph (N, E, s, t) consists of a set of nodes N, a set of edges E and two mappings $s, t : E \to N$ called source rsp. target. An edge e with $s(e) = n$ and $t(e) = m$ is denoted as $n \xrightarrow{e} m$.

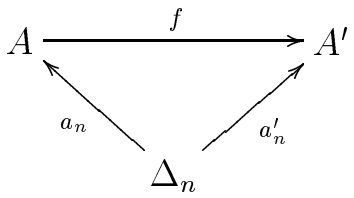
DEFINITION 2.2.2 (DIAGRAM)
A diagram Δ in a category $\mathcal{A}$ is a graph (N, E, s, t) (its shape) and two functions $f : N \to Obj(\mathcal{A})$ and $g : E \to Mor(\mathcal{A})$, where for each edge $e \in E$, $f(s(e)) = s_{\mathcal{A}}(g(e))$ and $f(t(e)) = t_{\mathcal{A}}(g(e))$, where $s_{\mathcal{A}}, t_{\mathcal{A}}$ are source and target of morphisms in $\mathcal{A}$. We denote $f(n)$, the object at node n by Δ_n and $g(e)$, the morphism at edge e by Δ_e. [5]

DEFINITION 2.2.3 (SINK,COCONE)
A sink in a category on a diagram Δ (the base) is an object A (the apex) together with a morphism $a_n : \Delta_n \to A$ for each node n in Δ. It is called cocone if for all edges $n \xrightarrow{e} m$ in Δ the first diagram below commutes. We abbreviate "$a_n : \Delta_n \to A$ for each $n \in N$" by "$a_n : \Delta_n \to A$" when the meaning is clear from the context.

[5] There is an alternative representation of diagrams as functors minimising the number of needed concepts. We prefer the given one because it has a stronger correspondence to its implementation.

A cocone morphism on Δ from $a_n : \Delta_n \to A$ to $a'_n : \Delta_n \to A'$ is a morphism $f : A \to A'$ such that for all nodes n in Δ the second diagram below commutes.





Definition 2.2.4 (Colimit)
A colimit of a diagram Δ is a cocone C such that for any cocone C' there is an unique cocone morphism $u : C \to C'$. The apex of C is called colimit object. A category having colimits of all finite diagrams is said to be finitely cocomplete. Functors which preserve colimits are called cocontinous.

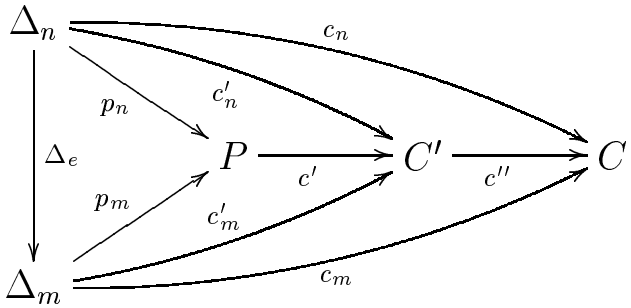
Colimits have the universal property, i.e. they are initial objects in the category of cocones on diagram Δ. They are the basic syntactical constructions to compose larger systems from several related subparts or as Goguen puts it in his categorical manifesto [Gog89]:

> *Given a category of widgets, the operation of putting a system of widgets together to form some super-widget corresponds to taking the colimit of the diagram of widgets that shows how to interconnect them.*

Definition 2.2.5 (Initial Object, Coproduct, Coequalizer)
An initial object in a category $\mathcal{C}$ is the colimit on the empty diagram. A coproduct in $\mathcal{C}$ of a diagram Δ is the colimit of the discrete diagram Δ' obtained from Δ by removing all morphisms. A coequalizer is the colimit on a diagram Δ with shape $n \underset{e'}{\overset{e}{\rightrightarrows}} m$. We say, the morphism from Δ_m into the colimit object is coequalizer of Δ_e and $\Delta_{e'}$.

The next lemma shows us, how colimit computation can be decomposed into the computation of coequalizers and binary coproducts.

Lemma 2.2.6
Diagram Δ' is obtained from Δ by removing morphism $\Delta_e : \Delta_n \to \Delta_m$. Let $p_n : \Delta_n \to P_\Delta$ be coproduct of Δ (and Δ'). Let $c'_n : \Delta_n \to C_{\Delta'}$ be colimit of Δ' with $c'_n = p_n \circ c'$. Then $c_n : \Delta_n \to C_\Delta$ with $c_n = c'' \circ c' \circ p_n$ is colimit of Δ, iff c'' is coequalizer of c'_n and $c'_m \circ \Delta_e$.



Proof: See [RB88, Pad91]. □

THEOREM 2.2.7 (DECOMPOSITION OF COLIMIT COMPUTATION)
A category having an initial object, binary coproducts and coequalizers has all finite colimits.

Constructive proofs are given in [RB88, Pad91]. They suggest an algorithm for computing colimits of a diagram Δ:

1. Compute recursively the coproduct $p_n : \Delta_n \to P_\Delta$ of Δ using the initial object and the binary coproduct. $p_n : \Delta_n \to P$ is colimit of the discrete diagram obtained from Δ by removing all morphisms.

2. Compute the colimit of Δ by recursively extending the coproduct using coequalizers. In every step an additional morphism Δ_e is added to the diagram. Here we have the situation described in lemma 2.2.6. $c'_n : \Delta_n \to C_{\Delta'}$ rsp. $c_n : \Delta_n \to C_\Delta$ are colimits of the diagrams before rsp. after insertion of Δ_e. By induction follows, that after insertion of all morphisms of Δ, the finally computed cocone is colimit of Δ.

Obviously this algorithm has worse then linear complexity related to the size of the diagram. Experiments with several implementations of this algorithm showed, that it is not applicable to larger diagrams.

Ordered categories [Jay90] play an important role for the treatment of categories with partial morphisms like graph structures. The concept of an ordered category is very general, we will restrict its application to express partial morphisms between set-based categories.

DEFINITION 2.2.8 (ORDERED CATEGORY)
An ordered category $\mathcal{O}$ is a category whose homsets are ordered, with the order preserved by composition, that is, $f \leq f' : A \to B$ and $g \leq g' : B \to C$ implies $g \circ f \leq g' \circ f'$.

The following well known definitions about properties of morphisms in arbitrary rsp. ordered categories are fundamental for later proofs.

DEFINITION 2.2.9 (PROPERTIES OF MORPHISMS)
A morphism $f : A \to B$ in a category $\mathcal{C}$ is called

- epi, if for all morphisms $g, h : B \to C$ in $\mathcal{C}$, $g \circ f = h \circ f$ implies $g = h$;
- mono, if for all morphisms $g, h : C \to A$ in $\mathcal{C}$, $f \circ g = f \circ h$ implies $g = h$;
- left inverse of $g : B \to A$, if $f \circ g = 1_B$;
- right inverse of $g : B \to A$, if $g \circ f = 1_A$;
- inverse of $g : B \to A$, if $f \circ g = 1_B$ and $g \circ f = 1_A$;
- split epi, if it has a right inverse $g : B \to A$;

- split mono, if it has a left inverse $g : B \to A$;
- iso, if it has an inverse $g : B \to A$;

The definitions for epi and mono can be generalized for sets of morphisms. A set of morphisms $F : A \to B$ is called jointly epi, if for all morphisms $g, h : B \to C$ in $\mathcal{C}$, $g \circ f = h \circ f$ for all $f \in F$ implies $g = h$. The notion jointly mono is defined analogously.

A morphism $f : A \to B$ in an ordered category $\mathcal{O}$ is called

- embedding with projection $f^\bullet : B \to A$, if $f^\bullet \circ f = 1_A$ and $f \circ f^\bullet \leq 1_B$
- deflation, if $A = B$ and $f \circ f = f \leq 1_A$.
- total, if for all morphisms $g : C \to A$ and deflations $d : C \to C$ holds $f \circ g \circ d = f \circ g \Rightarrow \ g \circ d = g$

These properties are related as shown in the next proposition.

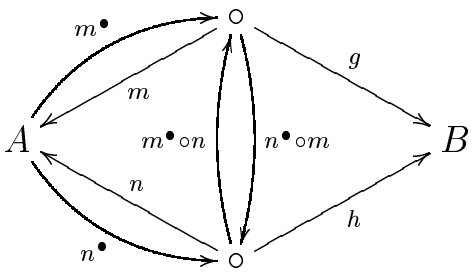
PROPOSITION 2.2.10
For all morphisms $f : A \to B$ in a category $\mathcal{C}$ holds:

1. *f is split mono $\Rightarrow$ f is mono*
2. *f is split epi $\Rightarrow$ f is epi*
3. *f is split mono and epi $\Leftrightarrow$ f is split epi and mono $\Leftrightarrow$ f is iso*

Let $c_n : \Delta_n \to C$ be colimit of Δ in $\mathcal{C}$ then the morphisms c_n are jointly epi.

For all morphisms $f : A \to B$ in an ordered category $\mathcal{O}$ holds:

1. *f is embedding $\Rightarrow$ f is split mono*
2. *f is projection $\Rightarrow$ f is split epi*
3. *f is mono $\Rightarrow$ f is total*
4. *f is embedding with projection $f^\bullet : B \to A \Rightarrow f \circ f^\bullet$ is deflation*
5. *f has at most one factorization (up to isomorphism) as a projection $m^\bullet$, followed by a total morphism g, i.e. , if m, n are embeddings, g, h are total and $g \circ m^\bullet = h \circ n^\bullet$, then $m^\bullet \circ n$ and $n^\bullet \circ m$ are inverse.*



Proof: See [BW90, Jay90]. □

Properties of total morphisms are:

PROPOSITION 2.2.11
For all morphisms $f : A \to B$ and $g : B \to C$ in an ordered category $\mathcal{O}$ holds:

1. *f and g are total $\Rightarrow$ $g \circ f$ is total*
2. *$g \circ f$ is total $\Rightarrow$ f is total*

Proof: Let $h : D \to A$ be a morphism and $d : D \to D$ be a deflation, then

1. $g \circ f \circ h \circ d = g \circ f \circ h \;\Rightarrow\; f \circ h \circ d = f \circ h$, since g is total, and
   $f \circ h \circ d = f \circ h \;\Rightarrow\; h \circ d = h$, since f is total, so $g \circ f$ is total.
2. $g \circ f \circ h \circ d = g \circ f \circ h \;\Rightarrow\; h \circ d = h$, since $g \circ f$ is total, and
   $f \circ h \circ d = f \circ h \;\Rightarrow\; g \circ f \circ h \circ d = g \circ f \circ h \;\Rightarrow\; h \circ d = h$, thus f is total.

□

Now we will show, how the morphisms in a diagram may be decomposed preserving the cocone and colimit property of a sink. This decomposition forms the basis of the cocompleteness proof for the category of graph structures.

LEMMA 2.2.12 (MORPHISM DECOMPOSITION)
Suppose $d_n : \Delta_n \to D$ is a sink of a $\mathcal{C}$–diagram Δ and $\Delta_e : \Delta_n \to \Delta_m$ is a morphism in Δ, as depicted in the left picture below.

Diagram Δ' is obtained from Δ by substituting Δ_e as shown in the right picture above, where $\Delta_e = f \circ m^\bullet$ and $m^\bullet$ is left inverse of m, i.e., $m^\bullet \circ m = 1_S$. Then $d_n : \Delta_n \to D$ is cocone rsp. colimit of Δ, iff $d_n : \Delta'_n \to D$ is cocone rsp. colimit of Δ'. [6]

Proof: First we show the preservation of the cocone property.

1. Suppose $d_n : \Delta_n \to D$ is cocone of Δ. Then follows for Δ' that $d_m \circ f \circ m^\bullet = d_m \circ \Delta_e = d_n$ and $d_m \circ f = d_m \circ f \circ 1_S = d_m \circ f \circ m^\bullet \circ m = d_n \circ m$, i.e., $d_n : \Delta'_n \to D$ is cocone of Δ'.
2. Suppose $d_n : \Delta'_n \to D$ is cocone of Δ'. Then follows for Δ that $d_m \circ \Delta_e = d_m \circ f \circ m^\bullet = d_n$, i.e. , $d_n : \Delta_n \to D$ is cocone of Δ.

[6] The morphism from S to D is given by $d_m \circ f$.

This means, that the category of Δ–cocones is isomorphic to the category of Δ'–cocones. Since colimits are initial (unique) in cocone categories, also the colimit property is preserved in both directions. □

REMARK 2.2.13
By induction follows easily that we can replace all morphisms as shown in lemma 2.2.12 thereby still preserving the colimit. The idea is to represent partial morphisms as spans of an embedding m (with projection $m^\bullet$) and a total morphism f. Lemma 2.2.12 shows, that if we replace Δ_e by its representation, the cocone rsp. colimit properties of a sink of Δ is not influenced. Later we will define unique span representations of partial graph structure morphisms, which form the basis of colimit computations in this category.

2.3 Signatures, Algebras and Homomorphisms

The idea of signatures is to provide an elementary syntactic description of data structures and operations on such structures.

DEFINITION 2.3.1 (SIGNATURE, GRAPH SIGNATURE)
A *signature* $SIG = (S, OP)$ consists of a set S of sorts and a set OP of operation symbols. The set OP is the union of pairwise disjoint subsets $OP_{w,s}$, operation symbols with argument sorts $w \in S^*$, i.e., $w = s_1 \dots s_n$, and range sort $s \in S$. We call operation symbols from $OP_{\lambda,s}$ constants of sort s. We declare operations $op \in OP_{w,s}$ by $op : w \to s$ and constants by $op :\to s$. We call w the *arity* and s the sort (or *range*) of op. A signature $SIG = (S, OP)$ is called graph signature if it contains only unary operations, that is, OP consists of subsets $OP_{s,t}$ with $s, t \in S$.

Algebras are the elementary semantic level associated with signatures. They provide a set of data structures for each sort and an operation, i.e., a function on data structures, for each operation symbol. Homomorphisms are structure preserving functions between algebras of the same signature.

DEFINITION 2.3.2 (PARTIAL ALGEBRA)
Given a signature $SIG = (S, OP)$, a partial SIG–algebra $A = (A(S), A(OP))$ consists of a S–sorted set $A(S) = (A_s)_{s \in S}$ of carrier sets and an OP–indexed family of operations $A(OP) = (op^A)_{op \in OP}$ such that each operation symbol $op : w \to s \in OP$ is realized as a partial function $op^A : A_w \to A_s$, where in the case $w = \lambda$ the operation op^A is called constant and for $w = s_1 \dots s_n$ we have $A_w = A_{s_1} \times \dots \times A_{s_n}$. We call A_w domain, A_s codomain and the subset of A_w, on which op^A is defined, scope of op^A denoted by $A_{w|op^A}$. We call a partial SIG–algebra total if all functions in $(op^A : A_w \to A_s)_{s \in S}$ are total, i.e. , their scopes are equal to their domains. [7]

The notion of subalgebra is an essential for this thesis. Not only the definition of partial homomorphisms depends on it, but also colimit computation in total graph structures is completed by the computation of a special subalgebra.

[7] Constants $op : \lambda \to s$ are defined, iff their scope is $\{A_\lambda\}$, i.e , $op^A :\to A_s \in A_s$, otherwise their scope is empty.

DEFINITION 2.3.3 (SUBALGEBRA)
Given SIG–algebras A and B then B is subalgebra of A, denoted $B \subseteq A$, iff all carrier sets of B are subsets of the carrier sets of A and for all operations $op : w \to s$ and arguments $b \in B_w$:

- b is in the scope of op^B iff b is in the scope of op^A;
- if b is in the scope of op^B, then $op^B(b) = op^A(b) \in B_s$.

Homomorphisms are structure preserving morphisms in the category of (partial) algebras.

DEFINITION 2.3.4 (HOMOMORPHISM)
Given a signature SIG and two SIG–algebras A and B, a total SIG–homomorphism $f : A \to B$ is a family of total functions $f = (f_s : A_s \to B_s)_{s \in S}$ such that for all operations $op : w \to s$ and arguments $a \in A_w$:

- $f_w(a)$ is in the scope of op^B iff a is in the scope of op^A;
- if $f_w(a)$ is in the scope of op^B then $op^B(f_w(a)) = f_s(op^A(a)) \in B_s$. [8]

A partial SIG–homomorphism $f : A \to B$ is represented by a total SIG–homomorphism $f_{|A_f} : A_f \to B$ from a subalgebra A_f of A, called scope of f, to B. $f_{|A_f}$ is the domain restriction of f to its scope. Let $C \subseteq A$ and $D \subseteq B$, then $f(C) = \{f(x) \mid x \in C\} \subseteq B$ is the image of C under f and $f^{\perp 1}(D) = \{x \mid f(x) \in D\} \subseteq A$ is the preimage of D under f.

The composition of two partial homomorphisms $f : A \to B$ and $g : B \to C$ is given by the componentwise composition of the underlying partial mapping. The scope of $g \circ f$ is $A_{g \circ f} = f^{\perp 1}(B_g \cap f(A_f))$.

REMARK 2.3.5
The scopes of the operations in the preimage are the inverse images of the scopes of the corresponding operations in the codomain. In the literature there are different definitions of (total) homomorphisms between partial algebras. Our definition corresponds to the notion of closed homomorphisms rsp. closed subalgebras from [Bur86]. Note that for each morphism $f : A \to B$ we have

- If A is total and C is subalgebra of A, then C is total.
- If A is total, then $f(A) \subseteq B$ is total.
- If B is total, then $f^{\perp 1}(B) \subseteq A$ is total.

Isomorphisms describe some kind of structural equivalence between algebras. Most concepts in category theory, for example colimits, describe objects only up to isomorphism. From this view we identify isomorphic objects. Note that in concrete tools based on category theoretical concepts we need a layer on top of them, managing the use of identifiers. For instance, if we apply colimits for parameterisation of abstract data types, we would expect identifiers from the actual parameter in the actualised data type, although these are identified by the colimit construction with the identifiers from the formal parameter.

[8] For constants this means: $op^A :\to A_s \in A_s$, iff $op^B :\to B_s \in B_s$, and then $op^A = op^B$.

Definition 2.3.6 (Isomorphism)
A SIG–homomorphism $f : A \to B$ is called isomorphism if all functions in $(f_s : A_s \to B_s)_{s \in S}$ are bijective, written $f : A \xrightarrow{\sim} B$, which allows to construct an inverse isomorphism $f^{\perp 1} : B \xrightarrow{\sim} A$ with $f^{\perp 1} \circ f = 1_A$ (identity on A) and $f \circ f^{\perp 1} = 1_B$ (identity on B). In this case the algebras A and B are called isomorphic, written $A \cong B$.

2.4 Graph Structures

Graph structures can be viewed as a very powerful generalisation of graphs and attributed graphs. They have been successfully applied for (data base) scheme transformations [CL95], the description of dynamic state transitions of parallel and distributed systems [Tae96b] and for the definition of the operational semantics of functional logic programming languages [CW94], to mention only some applications. Their definition enables an uniform treatment of all their concrete instances. For tool development especially the transformation between different graph structures is interesting. Later we will see, how complex graph structures will be transformed into Alpha algebras [EC96], which are much easier to implement.

Definition 2.4.1 (Graph Structure)
A partial SIG–algebra G is called partial graph structure if SIG is a graph signature, i.e., contains only unary operations. If G is a total algebra, then G is called total graph structure. G is called hierarchical, if $s \neq t$ for all compositions of operations $G^{op_1} \circ \cdots \circ G^{op_n} : G_s \to G_t$. Given a graph signature SIG, then

- The category of total graph structures over SIG with total homomorphisms is denoted by $\mathcal{GS}^{\mathcal{T}}_{SIG}$.
- The ordered category of partial graph structures over SIG with partial homomorphisms is denoted by $\mathcal{GS}^{\mathcal{P}}_{SIG}$, where the order relation $\leq$ on $\mathcal{GS}^{\mathcal{P}}_{SIG}$–morphisms is defined as $f \leq f' : A \to B$, if $A_f \subseteq A_{f'}$ and $f_{|A_f} : A_f \to B = f'_{|A_f} : A_f \to B$.
- The ordered category of total graph structures over SIG with partial homomorphisms is denoted by $\mathcal{GS}_{SIG}$. The order is defined as for $\mathcal{GS}^{\mathcal{P}}_{SIG}$.

Remark 2.4.2
It can easily be shown that:

1. The definition of a total morphism in definition 2.3.4 is consistent with definition 2.2.9.
2. Each deflation $d : A \to A$ in $\mathcal{GS}^{\mathcal{P}}_{SIG}$ rsp. $\mathcal{GS}_{SIG}$ is of the form $m^\bullet \circ m$, where m is embedding of projection $m^\bullet$, and $d(A)$, the image of A under d, is subalgebra of A.
3. The ordered category $\mathcal{SET}^{\mathcal{P}}$ of sets with partial morphisms can be regarded as single sorted graph structure without operations. According to this view we apply the notions scope, partial morphism, domain restriction and the definition of the order relation $\leq$ from $\mathcal{GS}^{\mathcal{P}}_{SIG}$ to $\mathcal{SET}^{\mathcal{P}}$.

Definition 2.4.3 (Strong Projection)
A projection $p : A \to B$ in a category of partial or total SIG–algebras is called strong projection if its right inverse $m : B \to A$ is an inclusion.

Remark 2.4.4
In $\mathcal{GS}^{\mathcal{P}}_{SIG}$ and $\mathcal{GS}_{SIG}$ a partial homomorphism $f : A \to B$ is defined as total homomorphism $f_{|A_f} : A_f \to B$ from a subalgebra A_f of A. Thus we can represent the unique isomorphism class of factorisations of $f = g \circ m^\bullet$ into a projection $m^\bullet$ followed by a total morphism g by choosing as corresponding embedding m the inclusion from A_f to A and as total morphism $f_{|A_f} : A_f \to B$. The left inverse of the inclusion is a strong projection.

Colimit computations are not of the same level of complexity in all graph structures. In a modular colimit algorithm the computation for complex structures should use the implementation of simpler ones. We derive the following order:

1. The category $\mathcal{SET}$ of sets with total morphisms, i.e., one sorted graph structures from $\mathcal{GS}^{\mathcal{T}}_{SIG}$ without operations.
2. The category $\mathcal{SET}^{\mathcal{P}}$ of sets with partial morphisms.
3. The category of simple partial graph structures with one operation op, denoted $\mathcal{GS}^{\mathcal{P}}_{op}$.
4. The category $\mathcal{GS}^{\mathcal{P}}_{SIG}$ of graph structures with partial operations.
5. The category $\mathcal{GS}_{SIG}$ of graph structures with total operations.

We will describe the implementation (and the constructive proof of cocompleteness) for each of these categories separately, and each is based on the previous one. In addition we will investigate comma categories over $\mathcal{SET}$ to handle signature and specification morphisms for algebraic specifications. Additionally to this vertical decomposition a structure on the colimit itself is required if we want to make our algorithm incremental. Extension of the diagram by a new object or morphism should not require a full recomputation, only the changes should by considered by the algorithm.

We could think of a decomposition of the whole computation into special colimits like coequalizers and coproducts (or pushouts) and initial objects using theorem 2.2.7. We have to be very careful here, because a naive implementation of this idea leads to a slow and unusable algorithm. But we can show that using our approach

- there is no overhead for incrementality;
- there is no need for separate implementations of pushouts and coproducts, because there is no overhead for general colimits;
- unfortunately deletion of objects and morphisms cannot be done incrementally.

The next definition suggests how we should represent partiality in our implementation.

DEFINITION 2.4.5 (POINTED SETS)
The category of pointed sets $\mathcal{SET}^{\perp}$ is defined as category of total algebras with total morphisms over an one sorted signature with a single constant. The value of the unique constant is denoted as $-$.

REMARK 2.4.6
Since pointed sets are total algebras, the constant is always defined, i.e., pointed sets contain a value $-$, which is always mapped on $-$ by a $\mathcal{SET}^{\perp}$–homomorphism.

The next theorem states the (trivial) fact, that the categories $\mathcal{SET}^{\perp}$ and $\mathcal{SET}^{\mathcal{P}}$ are equivalent. Therefore given an implementation for $\mathcal{SET}$, we get one for $\mathcal{SET}^{\mathcal{P}}$ almost for free.

THEOREM 2.4.7 (EQUIVALENCE OF $\mathcal{SET}^{\perp}$ AND $\mathcal{SET}^{\mathcal{P}}$)
The categories $\mathcal{SET}^{\perp}$ and $\mathcal{SET}^{\mathcal{P}}$ are equivalent, i.e, there is a completion functor $T_{\perp} : \mathcal{SET}^{\mathcal{P}} \to \mathcal{SET}^{\perp}$ and a restriction functor $R_{\perp} : \mathcal{SET}^{\perp} \to \mathcal{SET}^{\mathcal{P}}$, such that

$$R_{\perp} \circ T_{\perp} = 1_{\mathcal{SET}^{\mathcal{P}}} \quad \textit{and} \quad T_{\perp} \circ R_{\perp} \cong 1_{\mathcal{SET}^{\perp}}$$

Proof: The completion functor $T_{\perp} : \mathcal{SET}^{\mathcal{P}} \to \mathcal{SET}^{\perp}$ and the restriction functor $R_{\perp} : \mathcal{SET}^{\perp} \to \mathcal{SET}^{\mathcal{P}}$ are defined as follows:

- For all sets S we define $T_{\perp}(S) = S^{\perp}$, where $S^{\perp} = S \uplus \{-\}$.
- For all pointed sets $S^{\perp}$ we define $R_{\perp}(S^{\perp}) = S$, where $S = S^{\perp} - \{-\}$. [9]
- For all morphisms $f : A \to B$ we define $T_{\perp}(f) = f^{\perp} : A^{\perp} \to B^{\perp}$, where

$$f^{\perp}(x) = \begin{cases} f(x) & \text{if } x \in A_f \\ - & \text{otherwise} \end{cases}$$

- For all morphisms $f^{\perp} : A^{\perp} \to B^{\perp}$ we define $R_{\perp}(f^{\perp}) = f : A \to B$, where f is given by its scope $A_f = \{a \in A^{\perp} \mid f^{\perp}(a) \neq -\}$ and its total domain restriction $f_{|A_f}$, where $f_{|A_f}(x) = f^{\perp}(x)$.

These definitions immediately provide, that identities and compositions of morphisms are preserved by $T_{\perp}$ and $R_{\perp}$, so they are indeed functors, and that $R_{\perp} \circ T_{\perp} = 1_{\mathcal{SET}^{\mathcal{P}}}$ and $T_{\perp} \circ R_{\perp} \cong 1_{\mathcal{SET}^{\perp}}$. □

DEFINITION 2.4.8 ($\mathcal{SET}^{\perp}$ AS ORDERED CATEGORY)
Since $\mathcal{SET}^{\perp}$ and $\mathcal{SET}^{\mathcal{P}}$ are equivalent we define the order on $\mathcal{SET}^{\perp}$–morphisms as

$$f \leq f' \Leftrightarrow R_{\perp}(f) \leq R_{\perp}(f')$$

REMARK 2.4.9 Note that both $T_{\perp}$ and $R_{\perp}$ preserve embeddings, projections, inclusions and strong projections.

[9] $A - B$ is defined as $\{a \in A \mid a \notin B\}$.

But how are $\mathcal{SET}^{\perp}$ and $\mathcal{SET}$ related ?

THEOREM 2.4.10 (RELATION OF $SET^{\perp}$ AND $\mathcal{SET}$)
The free functor $F : \mathcal{SET} \to \mathcal{SET}^{\perp}$ is left adjoint to the corresponding forgetful functor $V : \mathcal{SET}^{\perp} \to \mathcal{SET}$. Thus F preserves colimits from $\mathcal{SET}$ to $\mathcal{SET}^{\perp}$.

Proof: See [EM85]. □

REMARK 2.4.11 Note that the forgetful functor $V : \mathcal{SET}^{\perp} \to \mathcal{SET}$ preserves limits, but not colimits. But we will see later, that colimits can be computed in $\mathcal{SET}^{\perp}$ as easy as in $\mathcal{SET}$.

The following proposition characterises the largest total subalgebra of a given partial graph structure. This construction not only important from a theoretical point of view. We will provide an efficient implementation computing $\mathcal{GS}_{SIG}$–colimits from $\mathcal{GS}^{\mathcal{P}}_{SIG}$–colimits.

PROPOSITION 2.4.12 (TOTALISATION)
Let $T(A) \subseteq A$ denote the largest total subalgebra of A with respect to $\subseteq$. Let $C \subseteq A$, $D \subseteq B$ and E, E' be graph structures and $f : A \to B$ be a partial homomorphism in $\mathcal{GS}^{\mathcal{P}}_{SIG}$. Then we have:

1. *C is total $\Leftrightarrow$ $f(C)$ is total*
2. *D is total $\Leftrightarrow$ $f^{\perp 1}(D)$ is total*
3. *$T(f(C)) = f(T(C))$*
4. *$T(f^{\perp 1}(D)) = f^{\perp 1}(T(D))$*
5. *$T(E \cap E') = T(E) \cap T(E')$*
6. *T can be extended to a functor $T : \mathcal{GS}^{\mathcal{P}}_{SIG} \to \mathcal{GS}_{SIG}$ (called totalisation), where the scope of $T(f) : T(A) \to T(B)$ is $T(A_f)$ and its total domain restriction is defined as $f_{|T(A_f)}$.*

Proof:

1. and 2. follow directly from the definition of *SIG*–homomorphisms (definition 2.3.4), since the scopes of the algebra operations are preserved by f from C to its image $f(C)$ and from D to its preimage $f^{\perp 1}(D)$.

3. $T(C)$ is total, with 1. follows $f(T(C)) \subseteq f(C)$ is total, since $T(f(C)$ is the largest total subalgebra of $f(C)$, we have $f(T(C)) \subseteq T(f(C))$.
With 2. we have, $f^{\perp 1}(T(f(C))) \subseteq C$ is total, hence subalgebra of $T(C) = f^{\perp 1}(f(T(C)))$, we derive $f^{\perp 1}(T(f(C))) \subseteq f^{\perp 1}(f(T(C))) \Leftrightarrow T(f(C)) \subseteq f(T(C))$.

4. Follows analogously to 3.

5. Since $T(E) \cap T(E') \subseteq E \cap E'$ is total we have $T(E) \cap T(E') \subseteq T(E \cap E')$. And since $T(E \cap E') \subseteq T(E)$ and $T(E \cap E') \subseteq T(E')$ we also have $T(E \cap E') \subseteq T(E) \cap T(E')$.

6. We show that $T : \mathcal{GS}^{\mathcal{P}}_{SIG} \to \mathcal{GS}_{SIG}$ is a functor:

- $T(f) : T(A) \to T(B)$ is $\mathcal{GS}_{SIG}$–morphism since

  $$T(f)(T(A)) = f_{|T(A_f)}(T(A)) \subseteq f(T(A)) \subseteq T(B)$$

  because $f(T(A)) \subseteq B$ is total, and $T(B)$ is the largest total subalgebra of B.
- $T(1_A) = 1_{A|T(A)} = 1_{T(A)}$
- For all $\mathcal{GS}^{\mathcal{P}}_{SIG}$–morphisms $f : A \to B$ and $g : B \to C$ holds

  $$T(g \circ f) = T(g) \circ T(f)$$

  The scope of $T(g) \circ T(f)$ is defined as

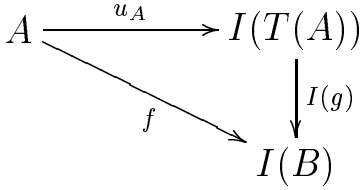
  $$T(A)_{T(g)\circ T(f)} = T(f)^{\perp 1}(T(B)_{T(g)} \cap T(f)(A_{T(f)}))$$

  From 3., 4. and 5. we can derive that it is equal to the scope of $T(g \circ f)$, i.e., $T(A_{g\circ f}) = T(f^{\perp 1}(B_g \cap f(A_f)))$. Since both $T(g \circ f)$ and $T(g) \circ T(f)$ are domain restrictions of $g \circ f$ to this scope, the equality is shown.

□

THEOREM 2.4.13 (RELATION OF $\mathcal{GS}^{\mathcal{P}}_{SIG}$ AND $\mathcal{GS}_{SIG}$)
The totalisation functor $T : \mathcal{GS}^{\mathcal{P}}_{SIG} \to \mathcal{GS}_{SIG}$ is a left adjoint functor, thus preserves colimits from $\mathcal{GS}^{\mathcal{P}}_{SIG}$ to $\mathcal{GS}_{SIG}$. Its right adjoint is the inclusion from $\mathcal{GS}_{SIG}$ into $\mathcal{GS}^{\mathcal{P}}_{SIG}$.

Proof: We denote the inclusion functor by $I : \mathcal{GS}_{SIG} \to \mathcal{GS}^{\mathcal{P}}_{SIG}$. For all $\mathcal{GS}^{\mathcal{P}}_{SIG}$–objects A the unit of the adjunction is defined as $u_A : A \to I(T(A))$, where the scope of u_A is $T(A)$, its total domain restriction is $1_{A|T(A)}$.



Now we have to show that for all $\mathcal{GS}^{\mathcal{P}}_{SIG}$–objects A, $\mathcal{GS}_{SIG}$–objects B and $\mathcal{GS}^{\mathcal{P}}_{SIG}$–morphisms $f : A \to I(B)$ there is an unique $\mathcal{GS}_{SIG}$–morphism $g : T(A) \to B$ with $I(g) \circ u_A = f$. Obviously u_A is a strong projection, i.e., has a right inverse inclusion i with $u_A \circ i = 1_{T(A)}$. Since $I(g) = g$ we derive:

$$g \circ u_A = f \quad \Leftrightarrow \quad g \circ u_A \circ i = f \circ i \quad \Leftrightarrow \quad g \circ 1_{T(A)} = f \circ i \quad \Leftrightarrow \quad g = f \circ i$$

Thus g is uniquely determined by $g \circ u_A = f$. [10] □

[10] An outline of this proof was developed during private communication with Ingo Claßen.

REMARK 2.4.14 This theorem enables a very efficient and modular colimit computation for $\mathcal{GS}_{SIG}$. For finite carrier sets there is an algorithm with linear complexity computing the largest total subalgebra of an algebra.

THEOREM 2.4.15 (COCOMPLETENESS OF HIERARCHICAL TOTAL GRAPH STRUCTURES)

The category of hierarchical total graph structures with partial morphisms has all finite colimits.

Proof: A (non-constructive) proof can be found in [Löw93a]. □

REMARK 2.4.16 Unfortunately the proof given in [Löw93a] isn't of much help in the design of an efficient colimit algorithm. Many important applications (ALPHA algebras [EC96], AGG-graphs [TB94]) violate the hierarchy constraint. A constructive proof that the category of general total graph structures $\mathcal{GS}_{SIG}$ is cocomplete is essential part of this thesis.

REMARK 2.4.17 Note that in [Löw93a] counterexamples for the cocompleteness of general total algebras with partial morphisms are given, which is the main motivation for the restriction to graph structures.

Now we have gathered all the theoretical background, to give an outline how colimits in $\mathcal{GS}_{SIG}$ can be computed in a modular and efficient manner:

1. First we investigate colimits in the category of sets $\mathcal{SET}$. The colimit is computed as disjoint union followed by a factorisation, which is efficiently implementable using the UNION-FIND algorithm [Tar83]. Note, that in [BW90] application of the Warshall's Algorithm [Sed83] for the factorisation is proposed. This leads to an inefficient colimit computation, since we have to compute a factorised set and not the factorising equivalence.

2. For the category of sets with partial morphisms $\mathcal{SET}^{\mathcal{P}}$, we switch to the equivalent category $\mathcal{SET}^{\perp}$ by adding a $-\!\!-$element to the sets in the diagram. After application of the algorithm for $\mathcal{SET}$, we have to apply the restriction functor $R_{\perp}$, i.e., the (only) equivalence class containing $-\!\!-$elements has to be deleted. It is a common mistake (see for example [Löw93a]) to mix factorisation and deletion, which causes significant unnecessary overhead.

3. Now we investigate the category of simple partial graph structures with only one operation *op* called $\mathcal{GS}^{\mathcal{P}}_{op}$. This category is similar to the arrow category $(\mathcal{SET}^{\mathcal{P}}, \mathcal{SET}^{\mathcal{P}})$, which means, that the colimit can be computed independently for the carrier sets and then for the operation. But we need to apply lemma 2.2.12 to split the morphisms of the first component, to express exactly the homomorphism properties of $\mathcal{GS}^{\mathcal{P}}_{SIG}$. This fact was already noticed in [Kor96].

4. Next we regard a diagram $\mathcal{GS}^{\mathcal{P}}_{SIG}$ with $SIG = (S, OP)$ as a set of $\mathcal{SET}^{\mathcal{P}}$–diagrams, one for each sort $s \in S$, together with a set of $\mathcal{GS}^{\mathcal{P}}_{op}$–diagrams, one for each operation $op \in OP$. In this view the carrier sets are redundantly represented, but we will show, that this redundancy does not lead to an inconsistency with respect to colimits and their universal morphisms. Thus we can simply put together the separately computed colimits. Note that in $\mathcal{GS}_{SIG}$, this technique is not applicable, because there the operations determine the colimits computed for the carrier sets. This is our main motivation to investigate $\mathcal{GS}^{\mathcal{P}}_{SIG}$, since we do not consider direct applications for partial graph structures.

5. Finally we apply theorem 2.4.13 and perform a totalisation of the result, obtaining the desired $\mathcal{GS}_{SIG}$–colimit.

3 Colimits in the Category of Sets and in Comma Categories over Sets

This chapter provides the theory for colimit computations in $\mathcal{SET}$ and in categories constructed as comma categories over $\mathcal{SET}$. We chose as examples signatures, specifications and attributed graphs. The constructive cocompleteness proofs presented here have a direct correspondence to the code of the colimit library, therefore serving also as a correctness proof of the implementation. In addition we can derive an estimation of the expected performance by a proof of the complexity of the computations. But the theory does not tell us, how the data structures have to be designed to optimise the real performance. As will be explained in chapter 4, new techniques from generic programming helped us to try different data structures without modifying the code of the algorithms to determine the best solution.

In addition we will give a brief overview over an example application of the colimit library, namely structuring operations for algebraic specification languages.

3.1 The Category of Sets

Colimit computation in $\mathcal{SET}$ is fundamental for the implementation of colimits in signatures, specifications, attributed graphs and graph structures in general. Design and performance of these algorithms depend heavily on the design of the data structures and algorithms in $\mathcal{SET}$. Similar to [RB88] we derive the algorithms from a proof of cocompleteness of $\mathcal{SET}$ (theorem 3.1.5), but because we have different requirements (genericity, reusability and efficiency) we choose a different proof leading to an alternative design.

First we define the equivalences induced by a $\mathcal{SET}$–morphism, both on its domain and codomain.

Definition 3.1.1 (Equivalences Generated by Morphisms)

- The equivalence $\sim_h$ induced on A by a $\mathcal{SET}$–morphism $h : A \to B$ is defined as

$$\sim_h = \{(x, y) \mid h(x) = h(y)\}$$

- For an arbitrary $\mathcal{SET}$–diagram Δ with coproduct $p_n : \Delta_n \to P$, the equivalence $\sim_\Delta^*$ on P generated by its edges is the equivalence closure of $\sim_\Delta$, defined as union of all

$$\sim_e = \{(p_n(x), p_m(\Delta_e(x))) \mid x \in \Delta_n\}$$

  for all edges e with source n and target m in Δ.

In $\mathcal{SET}$, computation of coproducts and coequalizers is well known, we present the facts in the next lemma.

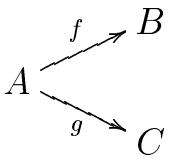
LEMMA 3.1.2 (COPRODUCT AND COEQUALIZER IN $\mathcal{SET}$)
In $\mathcal{SET}$ the coproduct is the disjoint union. Equivalence relations are represented by coequalizers, that is , given the diagram $A \overset{f}{\underset{g}{\rightrightarrows}} B \xrightarrow{h} C$ in $\mathcal{SET}$, $\sim_f^g$ denotes the equivalence closure over $\{(f(x), g(x)) \mid x \in A\}$ on B. Then h is coequalizer of f and g iff h is onto and $\sim_f^g = \sim_h$. The quotient function corresponding to $\sim_f^g$ is coequalizer of f and g. A map h is onto iff h is epi iff h is coequalizer (of some f and g).

Proof: See [BW90]. □

Together with lemma 2.2.6 this provides already a recursive algorithm for colimit computation using coproducts and coequalizers. But this solution is not optimal. We will first look at the whole colimit, choose an efficient implementation, and then check how we can achieve incrementality of the algorithm. We will loose the ability of an easy incremental deletion, but will gain a large performance boost.

The next lemma is needed for the construction of the universal morphism from the colimit to other cocones, which is used in theorem 3.1.5 and for the colimit algorithm for comma categories.

LEMMA 3.1.3
Given the $\mathcal{SET}$–diagram



where f is an onto map and $\sim_f \subseteq \sim_g$. Then there is an unique morphism $h : B \to C$ such that the diagram commutes.

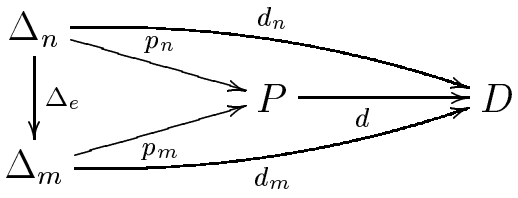
Proof: Define h by $h(f(x)) = g(x)$. The definition is complete because f is onto. It is sound, because

$$f(x) = f(y) \Rightarrow x \sim_f y \Rightarrow x \sim_g y \Rightarrow g(x) = g(y)$$

Obviously $h \circ f = g$ holds. h is unique since f is epi. □

The following lemma characterises cocones by the factorisation and deletion, they induce on the coproduct. It is necessary for proving applicability of lemma 3.1.3.

LEMMA 3.1.4
Let $p_n : \Delta_n \to P$ be coproduct of Δ in $\mathcal{SET}$. Let $d_n : \Delta_n \to P$ be cocone of Δ so that the diagram below commutes.



$d : P \to D$ is defined by the universal property of the coproduct P in the diagram obtained from Δ by removing all edges. Then holds

$$\sim_\Delta^* \subseteq \sim_d$$

Proof: By induction over the number of morphisms in Δ. If Δ has no morphisms, $\sim_\Delta$ is the empty relation, $\sim_\Delta^* \subseteq \sim_d$ trivially holds.

For the induction step we assume that

$$\sim_{\Delta'}^* \quad \subseteq \quad \sim_d \tag{1}$$

where Δ' is obtained from Δ by removing morphism $\Delta_e : \Delta_n \to \Delta_m$. Since $d \circ p_n : \Delta_n \to C$ is cocone after insertion of Δ_e, we have, applying definition 3.1.1,

$$d \circ p_n = d \circ p_m \circ \Delta_e \Rightarrow \sim_e \subseteq \sim_d \tag{2}$$

and together with assumption (1) follows $\sim_e \cup \sim_{\Delta'} \subseteq \sim_d$, hence

$$(\sim_e \cup \sim_{\Delta'})^* \quad = \quad \sim_\Delta^* \subseteq \sim_d \tag{3}$$

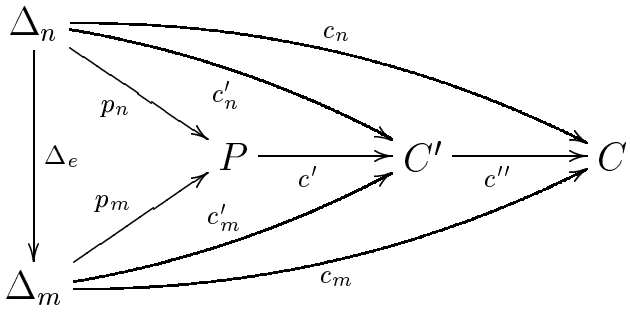
because $\sim_d$ is an equivalence. [11] □

Now we are ready to show, how colimits in $\mathcal{SET}$ can be computed.

THEOREM 3.1.5 (COCOMPLETENESS OF $\mathcal{SET}$)
Let Δ be a $\mathcal{SET}$–Diagram, $P = \biguplus_{n \in N} \Delta_n$ denotes the disjoint union of all objects Δ_n, $p_n : \Delta_n \to P$ are the corresponding injections, Let $C = P_{/\sim_\Delta^*}$ and $c : P \to C$ be the corresponding quotient function, that is, $\sim_c = \sim_\Delta^*$. Then $c \circ p_n : \Delta_n \to C$ is colimit of Δ.

Proof: We show the cocone property by induction over the number of morphisms in Δ. If Δ has no morphisms, $\sim_\Delta$ is the empty relation, $C = P$ and $c \circ p_n = p_n$ because c is the identity on P.

For the induction step we have the situation of lemma 2.2.6.



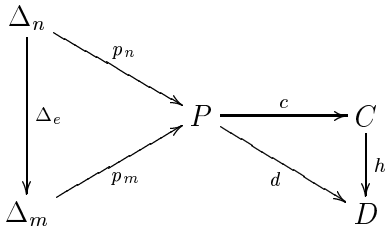
[11] Informally, d must identify everything in P, which is identified by Δ_e.

To prove that $c \circ p_n : \Delta_n \to C$ is colimit we assume that $c' \circ p_n : \Delta_n \to C'$ is colimit in Δ' where Δ' is obtained from Δ by removing morphism $\Delta_e : \Delta_n \to \Delta_m$ and $C' = P_{/\sim^*_{\Delta'}}$.

From the definition of c and c' follows $\sim_c = \sim^*_\Delta$ and $\sim_{c'} = \sim^*_{\Delta'}$, since $\sim^*_{\Delta'} \subseteq \sim^*_\Delta$, we have $\sim_{c'} \subseteq \sim_c$. So we can apply lemma 3.1.3, that is, there is an unique morphism $c'' : C' \to C$ such that $c'' \circ c' = c$.

Because $\sim_e \subseteq \sim_c$ we have $c \circ p_n = c \circ p_m \circ \Delta_e$, and since $c' \circ p_n : \Delta_n \to C'$ is cocone in Δ', $c \circ p_n : \Delta_n \to C$ is cocone in Δ (and Δ').

Suppose there is another cocone $d_n : \Delta_n \to D$ in Δ with $d : P \to D$ defined by the universal property of the coproduct P in the diagram obtained from Δ by removing all edges.



Following lemma 3.1.4, $\sim^*_\Delta \subseteq \sim_d$ holds. Hence $\sim_c \subseteq \sim_d$ and following lemma 3.1.3 there is an unique morphism $h : C \to D$ with $h \circ c = d$ commuting the whole diagram. □

That means, in the second step of the algorithm the sets are factorised by the equivalence relation induced by the morphisms, i.e., colimit computation in $\mathcal{SET}$ consists of computing the disjoint union of sets followed by a factorisation. The construction of the universal morphism to another cocone is based on the proof of lemma 3.1.3. We will need this construction later for colimit computation for comma categories.

Since we know from theorem 3.1.5, that colimit computation means mainly to separate a set (the coproduct) into disjoint subsets, we derive for the complexity of colimit computations in $\mathcal{SET}$:

THEOREM 3.1.6 (COMPLEXITY OF $\mathcal{SET}$–COLIMIT COMPUTATION) *Suppose application of a morphism to an element takes constant time. Then there is an algorithm for the computation of the colimit of a $\mathcal{SET}$–diagram Δ which has worst case running time $O(n + m \cdot \alpha(m, n))$, where $m = |P|$ is the number of elements in the disjoint union, $n = |\sim_\Delta|$ is the size of the relation induced by the morphisms and α denotes the inverse of the Ackermann's function.*

Proof: Obviously the disjoint union can be computed in linear ($O(m)$) time, for instance by tagging the elements. For the factorisation choose the UNION-FIND algorithm with weighted union and path compression, which has worst case running time $O(n + m \cdot \alpha(m, n))$ [Tar83] for n union and m find operations. The factorisation needs $n = |\sim_\Delta|$ union operations. The colimit morphisms c_n can be computed by $m = |P|$ find operations. □

The inverse of the Ackermann's function is *very* slowly growing. In any conceivable application, $\alpha(m, n) <= 4$; thus, we can view the running time as linear in all practical situations. Later we will see, how this result should be realized in a concrete implementation.

3.2 Comma Categories over Sets

Signatures, specifications and attributed graphs can be viewed as comma categories over $\mathcal{SET}$, which means that colimit computation in these categories can be modularised as shown below.

DEFINITION 3.2.1 (COMMA CATEGORY)
Let $L : \mathcal{A} \to \mathcal{C}$ and $R : \mathcal{B} \to \mathcal{C}$ be functors. Then triples $(A, f : L(A) \to R(B), B)$ are objects of the *comma category* (L, R) where A rsp. B are objects in $\mathcal{A}$ rsp. $\mathcal{B}$. Morphisms in (L, R) are pairs $(s : A \to A', t : B \to B')$ with $f' \circ L(s) = R(t) \circ f$ where composition is defined component-wise. The associated projection functors $\mathsf{left} : (\mathsf{L}, \mathsf{R}) \to \mathcal{A}$ and $\mathsf{right} : (\mathsf{L}, \mathsf{R}) \to \mathcal{B}$ are defined by $\mathsf{left}(A, f, B) = A$ and $\mathsf{right}(A, f, B) = B$. If $\mathcal{A} = \mathcal{C}$ and $L = 1_{\mathcal{C}}$ we denote (L, R) by $(\mathcal{C}, R)$, if in addition $\mathcal{B} = \mathcal{C}$ and $R = 1_{\mathcal{C}}$ we denote (L, R) by $(\mathcal{C}, \mathcal{C})$ and call it *arrow category* over $\mathcal{C}$.

THEOREM 3.2.2 (COCOMPLETENESS OF COMMA CATEGORIES)
The comma category (L, R) is finitely cocomplete if L and R are finitely cocomplete and the functor L preserves colimits.

Proof: Let Δ be a diagram in (L, R) and $\Delta^{\mathcal{A}}, \Delta^{\mathcal{B}}$ be the diagrams obtained from Δ by application of the projection functors left and right. Note that $\Delta^{\mathcal{A}}$ and $\Delta^{\mathcal{B}}$ share the same shape with Δ. Objects in Δ at node n are denoted as (A_n, f_n, B_n). Suppose that $a_n : A_n \to A$ and $b_n : B_n \to B$ are colimits of $\Delta^{\mathcal{A}}$ rsp. $\Delta^{\mathcal{B}}$.

$$\begin{array}{ccc} L(A_n) & \xrightarrow{f_n} & R(B_n) \\ {\scriptstyle L(a_n)}\downarrow & & \downarrow{\scriptstyle R(b_n)} \\ L(A) & \xrightarrow[f]{} & R(B) \end{array}$$

Then $R(b_n) \circ f_n : L(A_n) \to R(B)$ is cocone on $L(\Delta^{\mathcal{A}})$. In this diagram
$f_n : L(A_n) \to L(A)$ is colimit by the co-continuity of L. Thus by its universal property there is an unique f such that the diagram above commutes. The universal property of cocone

$$(a_n, b_n) : (a_n, f_n, b_n) \to (A, f, B)$$

in Δ can be derived from the universal properties of $a_n : A_n \to A$ and $b_n : B_n \to B$, see [Pad91]. □

This proof suggests an algorithm for computing colimits in comma categories (L, R) where $L : \mathcal{A} \to \mathcal{C}$ and $R : \mathcal{B} \to \mathcal{C}$:

1. Compute the colimits $a_n : A_n \to A$ of $\Delta^{\mathcal{A}}$ and $b_n : B_n \to B$ of $\Delta^{\mathcal{B}}$.
2. Construct the $\mathcal{C}$–morphism f using the universal property of $a_n : A_n \to A$ such that the diagram above commutes, that is, for all $x \in A_n$ and $n \in N$

$$f(L(a_n)(x)) = R(b_n)(f_n(x))$$

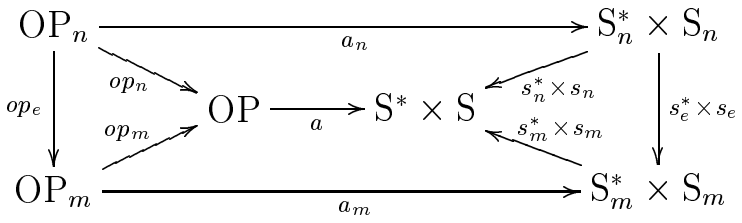
This constructive definition of f is complete, because every element in colimit object A is image of at least one $x \in A_n$. If there are more than one, we have the choice to avoid recomputation or to check consistency of the comma category morphisms.

3.2.1 Signatures

The usual definition as pairs $(\mathrm{S}, \mathrm{OP})$ of sets of sorts S and operations OP is equivalent to the following definition as comma category.

Definition 3.2.3 (Signatures as Comma Category)
The category of signatures $\mathcal{SIG}$ is defined as the comma category $(\mathcal{SET}, M)$ where $M : \mathcal{SET} \to \mathcal{SET}$ is defined by $M(\mathrm{S}) = \mathrm{S}^* \times \mathrm{S}$, $M(s : \mathrm{S} \to \mathrm{S}') = s^* \times s : \mathrm{S}^* \times \mathrm{S} \to \mathrm{S}'^* \times \mathrm{S}'$. Objects $(\mathrm{OP}, a, \mathrm{S})$ consist of a set of operations OP, an arity function $a : \mathrm{OP} \to \mathrm{S}^* \times \mathrm{S}$ and a set of sorts S. We denote an operation $\rho \in \mathrm{OP}$ with arity $a(\rho) = (s_1, \ldots, s_n, s)$ by $\rho : s_1, \ldots, s_n \to s$.

For the colimit computation of a $\mathcal{SIG}$–diagram Δ we have the following situation:



Let Δ^{OP} rsp. Δ^{S} the corresponding $\mathcal{SET}$–diagrams. (op_e, s_e) is the $\mathcal{SIG}$–morphism for edge $e : n \to m$. The $\mathcal{SIG}$–colimit $(op_n, s_n) : (\mathrm{OP}_n, a_n, \mathrm{S}_n) \to (\mathrm{OP}, a, \mathrm{S})$ for all nodes n in Δ can be computed using the two $\mathcal{SET}$–colimits $op_n : \mathrm{OP}_n \to OP$ and $s_n : \mathrm{S}_n \to \mathrm{S}$ together with the arity functions $a_n : \mathrm{OP}_n \to \mathrm{S}^* \times \mathrm{S}$ on the operation sets as described above for general comma categories.

Theorem 3.2.4 (Complexity of $\mathcal{SIG}$–Colimit Computation)
Suppose application of a morphism to an element and adding a single relation to an arity function takes constant time. Then there is an algorithm computing the colimit of a $\mathcal{SIG}$–diagram which has worst case running time $O(n + m \cdot \alpha(m, n))$, where $m = |P_{\mathrm{OP}}| + |P_{\mathrm{S}}|$ is the number of all operations plus the number of all sorts, $n = |\sim_{\Delta^{\mathrm{OP}}}| + |\sim_{\Delta^{\mathrm{S}}}|$ is the sum of the sizes of the relations induced by the morphisms.

Proof: Follows from theorem 3.1.6 and our remarks on the proof of theorem 3.2.2. Because the length of the sort sequences assigned by the arity functions to the operations $|a_n(\rho : s_1, \ldots, s_k \to s)| = k + 1$ is finite, its sum $\sum_{\rho \in OP_n, n \in N} |a_n(\rho)|$ is lower then $c \cdot |P_{OP}|$ for some constant c. Thus we need at most $|\sim_{\Delta^{\mathrm{OP}}}| + |\sim_{\Delta^{\mathrm{S}}}|$ **union** and $c \cdot |P_{\mathrm{OP}}| + |\sim_{\Delta^{\mathrm{S}}}|$ **find** operations. □

3.2.2 Specifications

Flat algebraic specifications are defined as pairs consisting of a signature and a set of axioms (equations, conditional equations or Horn clauses). Specification morphisms are defined as signature morphisms, such that the translated axioms are deducible. Deducability can be investigated by a theorem prover, using the colimit on the corresponding $\mathcal{SIG}$–diagram $(op_n, s_n) : (\mathrm{OP}_n, a_n, \mathrm{S}_n) \to (\mathrm{OP}, a, \mathrm{S})$. Translation of the axioms using the computed morphism (op_n, s_n) into the colimit $\mathcal{SIG}$–object is straightforward and has (in practice) linear complexity. Structuring operations on specifications as described in [Cla93] are expressed as diagrams over flat specifications. Flattening of structured specifications is done by an algorithm based on colimit computation on the corresponding $\mathcal{SIG}$–diagram. A more detailed discussion on algebraic specifications can be found in section 3.3.

3.2.3 Attributed Graphs

Another useful comma category is the category of attributed graphs as defined in [Sch91a]. We will not further investigate graph transformations in this category, because graph structures (see chapter 3) can be viewed as a generalisation of attributed graphs and are more flexibly applicable.

Definition 3.2.5 (Attributed Graphs)
The category of graphs (directed multigraphs) $\mathcal{GRAPH}$ is defined as the comma category $(\mathcal{SET}, P)$ where $P : \mathcal{SET} \to \mathcal{SET}$ is defined by $P(N) = N \times N$, $P(f : N \to N') = f \times f : N \times N \to N' \times N'$. Objects are triples (E, st, N), where $st : E \to N \times N$ defines source rsp. target node of an edge in E. [12] The category of attributed graphs $\mathcal{AGRAPH}$ is defined as the comma category (W, F) where $W : \mathcal{GRAPH} \to \mathcal{SET}$ extracts the nodes and edges: $W((E, st, N)) = E + N$, $W((f_E, f_N)) = f_E + f_N$. The forgetful functor $F : \mathcal{A} \to \mathcal{SET}$ from an arbitrary attribute category $\mathcal{A}$ maps each $\mathcal{A}$–object and $\mathcal{A}$–morphism to a representation in $\mathcal{SET}$ by "forgetting" the structure of $\mathcal{A}$. Objects are triples (G, a, A) where G is a graph (E, st, N), a an attribute function assigning an attribute from the $\mathcal{A}$–object A to each node and edge.

Corollary 3.2.6 (Cocompleteness of $\mathcal{AGRAPH}$)
Category $\mathcal{AGRAPH}$ is cocomplete if the attribute category $\mathcal{A}$ is cocomplete.

Proof: Follows from theorem 3.2.2. See also [Sch91a]. The category of F-graphs defined there is a generalisation of attributed graphs. □

An important example are algebra attributed graphs where the attribute category is the category of SIG–algebras over some signature SIG. The category of algebra attributed graphs is cocomplete because the category of SIG–algebras is cocomplete. Note that

- The definition suggests an algorithm for computing colimits of attributed graph diagrams similar to that for signature diagrams.

[12] This definition is equivalent to definition 2.2.1.

- From the implementation point of view the image of the forgetful functor F can be regarded as typed pointers uniquely identifying the attributes.
- Colimit computation on attributed graphs can be performed independent from the attributes. It is based on the colimit computation of the corresponding unattributed graphs which itself uses the computation of the colimits on the node– and edge–sets. As for signatures the worst case running time is almost linear.

3.3 Example Application: Parameterised Algebraic Specifications

In this section we present a brief overview of the theory of parameterised algebraic specifications to show where colimit computations in signature categories may be applied. We concentrate on syntactical aspects, for the treatment of semantical aspects we refer mainly to [Cla93, BG80], from where we use the basic definitions and constructions. Another very promising approach is used in the SPECWARE system [SJ95] where colimits are used directly as operations combining specifications. In [Smi93] techniques for automatic and semi-automatic construction of specification morphisms are presented.

3.3.1 Institutions and Specification Frames

Institutions [GB84] have been introduced by Goguen and Burstall to provide a structural theory of algebraic specifications that is independent of a concrete specification concept like, e.g., horn clause specifications. They have a very close relationship to specification frames (also called specification logic) with a logic of constraints presented by Ehrig et. al. in [EG91, EBCO91, EBO91, EJO93]. Specification frames together with constraints lead to a powerful abstraction of existing specification concepts. They are used in [Cla93] to provide structuring operations that are independent of concrete specification concepts.

Definition 3.3.1 (Specification Frame)
A specification frame $SF = (\mathrm{SPEC}, MOD\colon \mathrm{SPEC} \to \mathrm{CAT}^{\mathrm{OP}})$ consists of a category SPEC of (abstract) specifications and specification morphisms and a contravariant functor $MOD\colon \mathrm{SPEC} \to \mathrm{CAT}^{\mathrm{OP}}$ (where CAT is the quasi category of all categories and $\mathrm{CAT}^{\mathrm{OP}}$ the dual category of CAT), called the model functor, assigning to every specification $\mathrm{spec} \in \mathrm{SPEC}$ the category of models $MOD(\mathrm{spec})$ and to every specification morphism $f\colon \mathrm{spec} \to \mathrm{spec}'$ the forgetful functor $V_f =_{def} MOD(f)\colon MOD(\mathrm{spec}') \to MOD(\mathrm{spec})$.

For the semantical theory of structuring operations we require compositionality of specification frames as defined below.

Definition 3.3.2 (Compositional Specification Frame)
A specification frame $SF = (\mathrm{SPEC}, MOD)$ is called compositional if SPEC is finitely cocomplete and MOD preserves finite colimits.

Examples of compositional specification frames are conditional equational specifications based on partial algebras [Rei87, Wol90, CGW92] and specifications with constraints [EM90]. Constraints are used to achieve standard interpretations for certain specification parts which cannot be expressed by the specification frame. In [Cla93] the logic of constraints is extended for technical reasons to cover also morphisms.

Definition 3.3.3 (Extended Logic of Constraints)
An extended logic of constraints $ELC = (Constr, \models_{Obj}, \models_{Mor})$ on a specification frame

SF = (SPEC, MOD) is given by a functor $Constr$: SPEC $\rightarrow$ CLASSES and for every specification spec $\in$ SPEC two relations

$$\models_{Obj} \subseteq Obj(MOD(\text{spec})) \times Constr(\text{spec}) \tag{4}$$
$$\models_{Mor} \subseteq Mor(MOD(\text{spec})) \times Constr(\text{spec}) \tag{5}$$

called satisfaction relations, such that for all specification morphisms h: spec $\rightarrow$ spec$'$, all $A', f' \in MOD(spec')$ and $c \in Constr(\text{spec})$ the following satisfaction conditions are satisfied:

$$A' \models_{Obj} h^{\#}(c) \iff V_h(A') \models_{Obj} c \tag{6}$$
$$f' \models_{Mor} h^{\#}(c) \iff V_h(f') \models_{Mor} c \tag{7}$$

with $h^{\#} = Constr(h)$.

Examples are initial, (free) generating, first order logical constraints [EM90] and model constraints [Cla93]. For tool development we identify two tasks with respect to constraints:

- Checking whether a constraint holds for a given specification.
- Translation of constraints by specification morphisms.

We don't cover the first issue in this thesis, the second one is easy implementable for a given specification morphism. In [Cla93] it is shown, that a specification frame together with an extended logic of constraints induces a specification frame of abstract specifications with constraints. It is compositional if the underlying specification frame is compositional.

Definition 3.3.4 (Specifications with Extended Constraints)
For every specification frame SF = (SPEC, MOD) and every extended logic of constraints ELC = ($Constr, \models_{Obj}, \models_{Mor}$) the category SPECC has

1. specifications with constraints specc = (spec, C) as objects, with spec $\in$ SPEC and $C \subseteq Constr(\text{spec})$, and
2. consistent specification morphisms as morphisms, where a consistent specification morphism h: (spec$_1$, C_1) $\rightarrow$ (spec$_2$, C_2) is a specification morphism h: spec$_1$ $\rightarrow$ spec$_2$ that satisfies $C_2 \Rightarrow h^{\#}(C_1)$.

3.3.2 Unstructured (flat) specifications

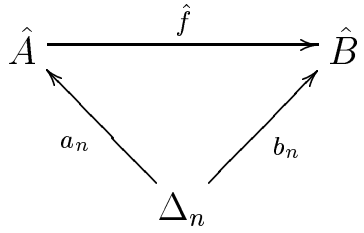
The category SPECC of specifications with constraints forms the basis of several structuring operations. An algebraic specification in SPECC consists of a signature, a set of axioms and a set of constraints. Specification morphisms are defined as signature morphisms, such that the translated axioms are deducible. They are consistent, if the translated constraints hold in the target specification.

Structuring operations on specifications involve only syntactical translations of the signatures and axioms may be implemented using colimit computations for signatures. They form the basis for semantical transformations [Smi93] which need support from a theorem prover.

3.3.3 Based Objects

Based objects were introduced by Burstall and Goguen [BG80] to provide a category theory based description of sharing issues in specification languages. The idea behind based objects is to regard a structured unit composed of several related subparts as a holistic entity. Note that this theory and the syntactical part of the structuring operations based on it [Cla93] is applicable also in other areas, like data base integration or parametrisation of programming languages.

DEFINITION 3.3.5 (BASED OBJECT, BASED MORPHISM)
Given a category $\mathcal{C}$ and a diagram Δ in $\mathcal{C}$ called environment. A based object $A = a_n : A_n \to \hat{A}$ for Δ is a cocone on a sub-diagram $\underline{A}$ of Δ, called base of A. Its apex is denoted by $\hat{A}$ and the morphisms a_n are called base morphisms. Given two based objects $A = a_n : A_n \to \hat{A}$ and $B = b_n : B_n \to \hat{B}$ such that $\underline{A}$ is a sub-diagram of $\underline{B}$, then a based morphism $f : A \to B$ [13] is induced by a morphism $\hat{f} : \hat{A} \to \hat{B}$ such that



commutes for all nodes n in $\underline{A} \subseteq \Delta$. The category of based objects over $\mathcal{C}$ for environment Δ is denoted by $\mathcal{C}_\Delta$. The identity based morphism $1_A : A \to A$ is induced by $1_{\hat{A}} : \hat{A} \to \hat{A}$, the composition $g \circ f : A \to C$ of two $\mathcal{C}_\Delta$–morphisms $f : A \to B$ and $g : B \to C$ is induced by $\hat{g} \circ \hat{f} : \hat{A} \to \hat{C}$

Now we present some auxiliary operations on based objects and environments. The composition of a based object A in $\mathcal{C}_\Delta$ with a $\mathcal{C}$–morphism f creates a new based object A' with the same base as A by applying f to the apex $\hat{A}$ of A. The base reduction cuts away part of the base of a based object.

DEFINITION 3.3.6 (COMPOSITION, BASE REDUCTION)
Let $A = a_n : A_n \to \hat{A}$ with A_n in $\underline{A}$ be based object in $\mathcal{C}_\Delta$.

- Let $f : \hat{A} \to \hat{A}'$ be morphism in $\mathcal{C}$. Then $A \star f = A' = f \circ a_n : A_n \to \hat{A}'$ with $\underline{A} = \underline{A'}$ denotes the composition of A with f.
- Let $\underline{B}$ be sub-diagram of the environment Δ, then $A{\downarrow_{\underline{B}}}$, the base reduction of A to $\underline{B}$ is given by $A{\downarrow_{\underline{B}}} = a_n : A_n \to \hat{A}$ with A_n in $\underline{A{\downarrow_{\underline{B}}}} = \underline{A} \cap \underline{B}$.

The induced based object for a diagram $\underline{A}$ represents the colimit of $\underline{A}$, the induced based morphism represents the universal morphism from the colimit to another cocone.

[13] Note that a base morphism is a morphism from a base object to the apex of a based object, where a based morphism is a morphism between two based objects.

Definition 3.3.7 (Induced Based Object and Morphism)
Let $\underline{A}$ be sub-diagram of the environment Δ. Then $\mathsf{Obj}(\underline{A}) = A = a_n : A_n \to \hat{A}$, the based object in $\mathcal{C}_\Delta$ induced by $\underline{A}$, is defined such that $a_n : A_n \to \hat{A}$ is colimit of $\underline{A}$ in $\mathcal{C}$.
Let $B = b_n : B_n \to \hat{B}$ be based object in $\mathcal{C}_\Delta$ where $\underline{A}$ is a sub-diagram of $\underline{B}$. Then $\mathsf{Mor}(B, \underline{A}) : \mathsf{Obj}(\underline{A}) \to B$, the based morphism induced by B and $\underline{A}$ is induced by the unique universal morphism $u : \hat{A} \to \hat{B}$ from the colimit object $\hat{A}$ to the cocone object $\hat{B}$, since $\mathsf{Obj}(\underline{A})$ is colimit and B is cocone of $\underline{A}$ in $\mathcal{C}$.

The $\mathcal{C}$–diagram Δ induced by a $\mathcal{C}_\Delta$–diagram ? is the "flattened" version of ?, i.e., we forget about the structure at the level of based objects.

Definition 3.3.8 (Induced Diagram)
Let $A = a_n : A_n \to \hat{A}$ with A_n in $\underline{A}$ be based object in $\mathcal{C}_\Delta$, then $\mathsf{Dia}(A)$, the $\mathcal{C}$–diagram induced by A, is the smallest diagram containing $\underline{A}$ with

- $\mathsf{Dia}(A)_i = \hat{A}$
- $\mathsf{Dia}(A)_{e_n} = a_n$ for all nodes n with A_n in $\underline{A}$.

where i is a fresh node and the e_n are fresh edges not in $\underline{A}$.
Let ? be a diagram in $\mathcal{C}_\Delta$, then $\mathsf{Flat}(?)$, the $\mathcal{C}$–diagram induced by ? is the smallest diagram containing

- $\mathsf{Dia}(?_n)$ for all nodes n in ?, where the fresh edges are chosen such that all edges in ? are not in $\mathsf{Dia}(?_n)$.
- $\mathsf{Flat}(?)_e = \hat{?_e} : \hat{?_n} \to \hat{?_m}$ for all edges $n \xrightarrow{e} m$ in ?, where the $\mathcal{C}$–morphism $\hat{?_e}$ induces $?_e : ?_n \to ?_m$ in $\mathcal{C}_\Delta$.

The next operation extends the environment Δ by a new based object.

Definition 3.3.9 (Extension of the Environment)
Let environment Δ be diagram in $\mathcal{C}$ and $A = a_n : A_n \to \hat{A}$ with A_n in $\underline{A}$ a based object in $\mathcal{C}_\Delta$. Then $\mathsf{bind}(i, A, \Delta) = \Delta'$ is the smallest diagram containing Δ with

- $\Delta'_i = \hat{A}$
- $\Delta'_{e_n} = a_n$ for all nodes n with A_n in $\underline{A}$.

where i is a fresh node and the e_n are fresh edges not in Δ.
$\mathsf{bind}(\langle i_1, \ldots, i_m \rangle, \langle A_1, \ldots, A_m \rangle, \Delta)$ denotes the straightforward extension of $\mathsf{bind}(i, A, \Delta)$ to tuples of nodes and based objects.

Colimits of diagrams ? of based specifications are implemented using the "flattened" diagram $\mathsf{Flat}(?)$.

Lemma 3.3.10 (Cocompleteness of $\mathcal{C}_\Delta$)
The category $\mathcal{C}_\Delta$ of based objects over $\mathcal{C}$ has all finite colimits if $\mathcal{C}$ has all finite colimits.

Proof: A proof can be found in [BG80]. The idea is as follows: Let ? be a diagram in $\mathcal{C}_\Delta$ then the colimit of ? is $C = c_n : C_n \to \hat{C}$ with C_n in $\underline{C}$ where

1. Apex $\hat{C}$ is colimit of $\mathsf{Flat}(?)$ (see definition 3.3.8) in $\mathcal{C}$.
2. Base $\underline{C}$ is the union of the bases of all based objects in ?.
3. The colimit morphisms c_n are given by the colimit construction in step 1. above. For all $\mathcal{C}_\Delta$–objects C_n in $\underline{C}$, c_n is induced by the colimit injection from $\hat{C}_n$ to the colimit object of $\mathsf{Flat}(?)$.

□

Definition 3.3.11 (Based Specifications)
Given a category of specifications with constraints SPECC, the category of based specifications with environment Δ is the category SPECC_Δ of based objects over SPECC.

Corollary 3.3.12
The category SPECC_Δ *of based specifications has all finite colimits.*

Proof: Follows directly from lemma 3.3.10. □

3.3.4 Parameterised Specifications

A parameterised specification has a formal parameter part that is to be instantiated by actual parameters. It can be realized by defining a parameterised specification to be a pair (P, B) of specifications [EM85] consisting of a parameter part P and a body part B such that there is a specification morphism from P to B, which normally represents an inclusion relation between parameter and body. That means, parameterised specifications for a given category SPECC of specifications with constraints can be defined as morphisms in SPECC.

Definition 3.3.13 (Parameterised Specifications)
Given a category of specifications with constraints SPECC, the category of parameterised specifications is the arrow category $\mathrm{PSPEC} = (\mathrm{SPECC}, \mathrm{SPECC})$ over SPECC.

Parameterised specifications with parameter P and body B are denoted by $P \xrightarrow{p} B$. Unparameterised specifications are regarded as morphisms with empty domain and denoted by $\emptyset \xrightarrow{\Box_B} B$.

Since arrow categories are special comma categories, we know from theorem 3.2.2 and section 3.2.2 how to compute colimits in PSPEC, if we restrict us to a syntactical translation of the axioms.

Proposition 3.3.14
The category PSPEC *of parameterised specifications w.r.t. a compositional specification frame* SF *and an extended logic of constraints* ELC *has all finite colimits.*

Proof: Follows directly from theorem 3.2.2, since SPECC has all finite colimits. □

3.3.5 Based Parameterised Specifications

We represent the structure of a specification in a diagram Δ which serves as an environment. The idea is that we are able to retrieve the places of origin for all sorts, operations and axioms in the specification. On a higher level of abstraction we view a cocone A, where the base of A is a subdiagram of Δ, as object in the category of based specifications. A based specification represents a specification, the apex, together with a set of imported specifications, the base of the cocone. Colimits at the level of based specifications are computed via colimits on the corresponding diagram of (flat) specifications.

Definition 3.3.15 (Category of Based Parameterised Specifications)
The category PSPEC_Δ of based parameterised specifications w.r.t. PSPEC and a PSPEC–diagram Δ (the environment) is the category of based objects in PSPEC over Δ.

Based parameterised specifications from PSPEC_Δ are denoted by $P \xrightarrow{p} B$ where $\hat{P} \xrightarrow{\hat{p}} \hat{B}$ denotes its apex.

Proposition 3.3.16
The category PSPEC_Δ *of based parameterised specifications w.r.t.* PSPEC *and an environment* Δ *is finitely cocomplete.*

Proof: Follows directly from proposition 3.3.14 and lemma 3.3.10. □

To provide tool support for based parameterised specifications, we need general colimit computations for signatures and specifications since their structuring mechanisms express the sharing of subparts by additional morphisms.

3.3.6 Algebraic Concepts for Modularity

The development of large software systems naturally requires mechanisms to divide systems into reasonable pieces and modularisation is regarded as a major technique to achieve this goal. No matter what specific model of software development and what modularisation technique are adopted some general modularisation principles can be identified.

- splitting of the workload and system into pieces
- hiding of irrelevant information in system pieces
- compositionality of system correctness
- compositionality of system formalisation

The algebraic concepts of modularity were introduced by Weber and Ehrig in [WE86] and further established in [EM90] and [CEW93]. These concepts are based on the pioneering work of Parnas on modules [Par72] and of Liskov and Zilles on data abstraction [LZ75].

In contrast to many other data abstraction approaches modules have a precise algebraically defined semantics that can be considered the prerequisite for the formal specification of modular systems and for the verification of their correctness. An important

property of the approach from [EM90] is the formal nature of imported parts, i.e., an import interface only states what kinds of services are needed but not which module provides them.

Modules in this approach consist of four constituent parts: an export interface, an import interface, a body, and a parameter part.

module
parameter
import interface
export interface
body
endmodule

Modular systems are interconnections of modules with matching import and export interfaces. The entire modular system is a hierarchy the interconnected modules.

The import interface contains a number of formal parameters that may be matched by the export of one or a number of other modules. These formal parameter parts are combined into one specification. The parameter represents the part of the import, which is also part of the export interface, and hence visible to the user. The theory from [EM90] may be combined with the theory of based specifications to support also the sharing of subparts as introduced in [BG80, Cla93]. Then we would need also for modular specifications general colimit computations.

4 Colimits in the Category of Graph Structures

In this chapter we extend the results for $\mathcal{SET}$ to $\mathcal{SET}^{\mathcal{P}}$, the category of sets with partial functions, and from comma categories over $\mathcal{SET}$ to partial and total graph structures.

There is a broad literature on categories with partial maps, for instance [RR88, Jay90, SP82], which helps to understand the topic at a higher level of abstraction. Our definitions are influenced by them, especially by [Jay90], but for $\mathcal{SET}^{\mathcal{P}}$ and partial graph structures this high level of abstraction is not necessary. Keep in mind that most of our proofs describe concrete algorithms. For this reason and for the sake of brevity we choose more concrete definitions.

We will present a brief overview over several applications of the colimit computations for graph structures, namely

- tool design for entity relationship scheme transformations,
- re-implementation of the AGG-system, a tool for the manipulation of graphs supporting different layers of abstraction with a modern mouse driven user interface,
- tool development for algebraic high level nets, which can be viewed as petri nets augmented with a powerful data type specification formalism.
- simulation of an abstract machine for the execution of functional logic languages at different levels of abstraction.

4.1 The Category of Sets with Partial Morphisms

We will carry over theorem 3.1.5 about cocompleteness of $\mathcal{SET}$ to $\mathcal{SET}^{\mathcal{P}}$ via the equivalence between $\mathcal{SET}^{\perp}$ and $\mathcal{SET}^{\mathcal{P}}$.

Note, that partial $\mathcal{SET}^{\mathcal{P}}$–morphism $f : A \to B$ can be interpreted as a total morphism from its scope $A_f \subseteq A$ to B, the domain restriction of f to A_f, denoted by $f_{|A_f} : A_f \to B$.

LEMMA 4.1.1
For all morphisms $f : A \to B$ in $\mathcal{SET}^{\mathcal{P}}$ holds:

1. *f is mono $\Leftrightarrow$ f is total and injective*
2. *f is epi $\Leftrightarrow$ f is onto*
3. *f is embedding with corresponding projection $f^{\bullet} : B \to A \Leftrightarrow$ f is total and injective, $f^{\bullet}$ is onto and injective*

For all morphisms $f : A \to B$ in $\mathcal{SET}^{\perp}$ holds:

1. *f is total $\Leftrightarrow$ $x \neq -$ implies $f(x) \neq -$*
2. *f is embedding with corresponding projection $f^{\bullet} : B \to A \Leftrightarrow$ f is total and injective, $f^{\bullet}$ is onto, $f^{\bullet}_{|(B \perp \perp)}$ is injective and $x \neq f(f^{\bullet}(x))$ implies $x = -$*

Proof: trivial. □

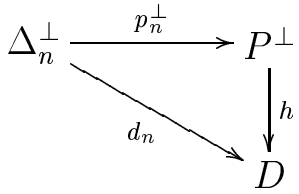
REMARK 4.1.2 Note that we only use inclusions (and corresponding strong projections) in span representations of partial $\mathcal{SET}^{\mathcal{P}}$–morphisms.

First we will investigate pointed sets. They represent our "implementation" of partiality. We use the equivalence between $\mathcal{SET}^{\perp}$ and $\mathcal{SET}^{\mathcal{P}}$ to verify the cocompleteness of $\mathcal{SET}^{\mathcal{P}}$. We could have done it directly, but in our implementation, partial morphisms are realized by a $--$–element exactly as in $\mathcal{SET}^{\perp}$. We aim at a close correspondence between the theory and our implementation. In addition this representation clarifies the relation between factorisation and deletion, which was often misunderstood in the past, leading to an unnecessary complex description of colimit rsp. pushout constructions in $\mathcal{GS}_{SIG}$ [Löw93a, Kor96]. Later we will hide that "implementation", for the description of $\mathcal{GS}_{SIG}$–colimits we will use again $\mathcal{SET}^{\mathcal{P}}$.

The coproduct in $\mathcal{SET}^{\perp}$ differs slightly from $\mathcal{SET}$, since every $\mathcal{SET}^{\perp}$–homomorphism preserves the constant $-$. Thus we have only one copy of $-$ in the coproduct.

LEMMA 4.1.3 (COPRODUCTS IN $\mathcal{SET}^{\perp}$)
Let $\Delta^{\perp}$ be a discrete $\mathcal{SET}^{\perp}$–diagram, $P^{\perp} = \biguplus_{n\in N}(\Delta_n^{\perp} - -) \uplus -$, and $p_n^{\perp} : \Delta_n^{\perp} \to P^{\perp}$ are the corresponding injections. Then $p_n^{\perp} : \Delta_n^{\perp} \to P^{\perp}$ is coproduct of $\Delta^{\perp}$.

Proof: Obviously $p_n^{\perp} : \Delta_n^{\perp} \to P^{\perp}$ is cocone of $\Delta^{\perp}$, since $\Delta^{\perp}$ is discrete.



Suppose there is another cocone $d_n : \Delta_n^{\perp} \to D$ in $\Delta^{\perp}$. Then morphism $h : P^{\perp} \to D$ is given by: If $x = p_n^{\perp}(y)$ for some node n and element $y \in \Delta_n^{\perp}$ then $h(y) = d_n(x)$. The definition is complete, since all $p_n^{\perp}$ are jointly onto. It is sound since all p_n are injective and their image overlaps only in $-$, hence for all nodes n and m and $x \neq -$ or $y \neq -$ we have

$$p_n^{\perp}(x) = p_m^{\perp}(y) \Rightarrow x = y \Rightarrow d_n(x) = d_m(y)$$

Obviously $h \circ p_n = d_n$ and h is unique since all $p_n^{\perp}$ are jointly epi. □

Since the factorisation is identical in $\mathcal{SET}$ and $\mathcal{SET}^{\perp}$ we derive from lemma 4.1.3:

LEMMA 4.1.4 (COCOMPLETENESS OF $\mathcal{SET}^{\perp}$)
Let $\Delta^{\perp}$ be a $\mathcal{SET}^{\perp}$–diagram, $P^{\perp} = \biguplus_{n\in N}(\Delta_n^{\perp} - -) \uplus -$ and $p_n^{\perp} : \Delta_n^{\perp} \to P^{\perp}$ are the corresponding injections. Let $C^{\perp} = P^{\perp}_{/\sim^*_{\Delta^{\perp}}}$ and $c^{\perp} : P^{\perp} \to C^{\perp}$ be the corresponding quotient function, that is, $\sim_{c^{\perp}} = \sim^*_{\Delta^{\perp}}$. Then $c^{\perp} \circ p_n^{\perp} : \Delta_n^{\perp} \to C^{\perp}$ is colimit of $\Delta^{\perp}$.

Proof: Lemma 3.1.3 and lemma 3.1.4 hold also in $\mathcal{SET}^{\perp}$ (the proofs are identical). Together with lemma 4.1.3 we can lift the proof of theorem 3.1.5 from $\mathcal{SET}$ to $\mathcal{SET}^{\perp}$. □

Now we come to the fundamental theorem on colimits in $\mathcal{SET}^{\mathcal{P}}$, showing how they have to be computed.

THEOREM 4.1.5 (COCOMPLETENESS OF $\mathcal{SET}^{\mathcal{P}}$) *Let Δ be $\mathcal{SET}^{\mathcal{P}}$–Diagram and $\Delta^{\perp}$ the diagram obtained from Δ by applying the completion functor $T_{\perp}$ to all its objects and morphisms. Let $P^{\perp}$ and $c^{\perp}$ be defined as in lemma 4.1.4. Let $P = \biguplus_{n\in N} \Delta_n$ and $p_n : \Delta_n \to P$ are the corresponding injections, $C = P^{\perp}_{/\sim^*_{\Delta^{\perp}}} - -$ and $c = R_{\perp}(c^{\perp})$. Then*

1. *$p_n : \Delta_n \to P$ is coproduct of Δ in $\mathcal{SET}^{\mathcal{P}}$.*
2. *$c \circ p_n : \Delta_n \to C$ is colimit of Δ in $\mathcal{SET}^{\mathcal{P}}$.*

Proof: Follows immediately from the equivalence between $\mathcal{SET}^{\perp}$ and $\mathcal{SET}^{\mathcal{P}}$ and the definition of the completion functor $T_{\perp} : \mathcal{SET}^{\mathcal{P}} \to \mathcal{SET}^{\perp}$ and the restriction functor $R_{\perp} : \mathcal{SET}^{\perp} \to \mathcal{SET}^{\mathcal{P}}$ in the proof of theorem 2.4.7. We have $P = R_{\perp}(P^{\perp})$ and $C = R_{\perp}(C^{\perp})$. Hence we can apply lemma 4.1.3 and lemma 4.1.4. Since $\mathcal{SET}^{\mathcal{P}}$ and $\mathcal{SET}^{\perp}$ are equivalent we derive both 1. and 2. □

REMARK 4.1.6 (RELATION OF FACTORISATION AND DELETION)
Thus we have a colimit in $\mathcal{SET}^{\mathcal{P}}$ given by $C = P^{\perp}_{/\sim^*_{\Delta^{\perp}}} - -$. This clarifies, how deletion and factorisation are related to each other. The question is, how we should interpret this result. In $C^{\perp}$ element $-$ represents the equivalence class of deleted elements. The whole deletion process is performed by the restriction functor $R_{\perp}$, since $C = R_{\perp}(C^{\perp})$, thus simply by removing $-$ from $C^{\perp}$. Obviously this is both simpler and more efficient than to perform the deletion partly in advance (as proposed by [Löw93a] and [Kor96]), since the factorisation has linear complexity and we avoid the danger of a mutual dependency between factorisation and deletion.

We derive the following algorithm computing the colimit of a $\mathcal{SET}^{\mathcal{P}}$–diagram.

1. Represent partiality by a $--$–element.
2. Compute the disjoint union of all sets in the diagram.
3. Factorise according to the morphisms in the diagram.
4. Remove the equivalence class corresponding to $-$ representing deleted elements in the colimit.

That means, colimit computation in $\mathcal{SET}^{\mathcal{P}}$ has the same complexity as in $\mathcal{SET}$.

COROLLARY 4.1.7 (COMPLEXITY OF $\mathcal{SET}^{\mathcal{P}}$–COLIMIT COMPUTATION)
Let Δ be $\mathcal{SET}^{\mathcal{P}}$–Diagram and $\Delta^{\perp}$ the diagram obtained from Δ by applying the completion functor $T_{\perp}$ to all its objects and morphisms. Suppose application of a morphism to an element takes constant time. Then there is an algorithm for the computation of the colimit of Δ, which has worst case running time $O(n + m \cdot \alpha(m, n))$ where m is the sum of the number of elements in all nodes of $\Delta^{\perp}$, $n = \mid \sim_{\Delta^{\perp}} \mid$ is the size of the relation induced by the morphisms and α denotes the inverse of the Ackermann's function.

Proof: Follows immediately from theorem 3.1.6, theorem 4.1.5 and our remark on the computation of colimits in $\mathcal{SET}^{\mathcal{P}}$. □

REMARK 4.1.8
Note that in an extreme situation where all morphisms are (almost) everywhere undefined, the size of a completed morphism is still proportional to the size of its domain (and not to the size of its scope). If in addition the number of morphisms in the diagram is very high compared to the number of nodes, it may be more efficient to apply a different strategy and to perform the deletion partly in advance. But this algorithm is much harder to prove and to implement. A performance improvement can only be achieved, if we choose another representation of morphisms (by associative containers), where undefinedness is no longer represented explicitly by a $-\!-$–element.

Since the situation described above occurs very rarely (if at all) in our applications, we will always defer the deletion after the factorisation is completed. But our implementation will be designed in a way, that we could easily replace the representations of objects and morphisms, if there is a demand by a special new application.

4.2 The Category of Graph Structures

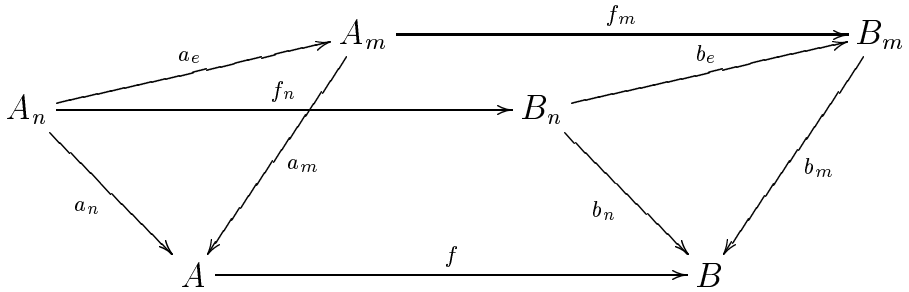
In this section we will investigate colimits in the category $\mathcal{GS}^{\mathcal{P}}_{SIG}$ of partial graph structures (with partial morphisms). First we will analyse graph structures with only one operation.

DEFINITION 4.2.1 (SIMPLE GRAPH STRUCTURE)
A graph signature is called simple, if it contains exactly one operation symbol $op : s \to t$ together with the (not necessarily different) sort symbols s and t. Algebras to these signatures are called simple graph structures. The category of partial simple graph structures with operation $op : s \to t$ is denoted by $\mathcal{GS}^{\mathcal{P}}_{op}$. We denote analogously to arrow categories [14] simple graph structures G as $(G_s \xrightarrow{op^G} G_t)$ and morphisms $c : G \to G'$ as

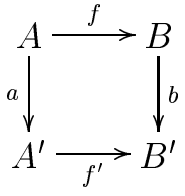
$$(c_s, c_t) : (G_s \xrightarrow{op^G} G_t) \to (G'_s \xrightarrow{op^{G'}} G'_t)$$

REMARK 4.2.2
To avoid confusion by using too many subscripts, and since we have at most two sort symbols in a simple graph signature, we simplify the notation by denoting objects as $(A \xrightarrow{f} B)$ and morphisms as $(a, b) : (A \xrightarrow{f} B) \to (A' \xrightarrow{f'} B')$, where A, B are sets and a, b are $\mathcal{SET}^{\mathcal{P}}$–morphisms. Accordingly we decompose diagrams Δ in $\mathcal{GS}^{\mathcal{P}}_{op}$ into two subdiagrams Δ^A and Δ^B sharing the same shape with Δ, which correspond to the domain rsp. codomain sets of the operation. Sinks in Δ are denoted as

$$(a_n, b_n) : (A_n \xrightarrow{f_n} B_n) \to (A \xrightarrow{f} B)$$



The main idea is to realize the similarities and differences of $\mathcal{GS}^{\mathcal{P}}_{op}$ with the arrow category $(\mathcal{SET}^{\mathcal{P}}, \mathcal{SET}^{\mathcal{P}})$, that is, the category where the objects are $\mathcal{SET}^{\mathcal{P}}$–morphisms and the morphisms are pairs of $\mathcal{SET}^{\mathcal{P}}$–morphisms, such that for all objects $(A \xrightarrow{f} B), (A' \xrightarrow{f'} B')$ and morphisms $(a, b) : (A \xrightarrow{f} B) \to (A' \xrightarrow{f'} B')$ the following diagram commutes.



[14] Arrow categories are simple comma categories, where both functors are the identity.

Then objects $(A \xrightarrow{f} B)$ correspond to simple partial graph structures with carrier sets A and B and operation $f : A \to B$.

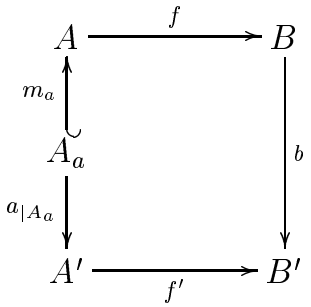
Since arrow categories are special comma categories we could easily compute colimits in this category as shown in section 3.2. But unfortunately, as already noticed in [Kor96], $\mathcal{SET}^{\mathcal{P}}$–arrow morphisms as defined above don't correspond exactly to the definition of $\mathcal{GS}^{\mathcal{P}}_{op}$–homomorphisms (compare definition 2.3.4).

Imagine, one sort in our simple graph structure represents items, the other one attributes, and the operation a reference from items to attributes. Then every graph transformation rule [15] deleting an item would also delete all referenced attributes, caused by the commutativity of the diagram above. This is obviously a severe restriction of the expressiveness of transformation rules, since the attributes may be shared with other items.

Replacement of the commutativity by weak commutativity, that is, $f' \circ a \leq b \circ f$ is also not satisfactory for our purposes. For our example this would allow the deletion of the reference to an undeleted attribute. Keep in mind that our main concern are total graph structures, we use $\mathcal{GS}^{\mathcal{P}}_{SIG}$ mainly as an intermediate step in the colimit computation for $\mathcal{GS}_{SIG}$. This was the reason for requiring the preservation of scopes by $\mathcal{GS}^{\mathcal{P}}_{SIG}$–morphisms [16].

Remark 4.2.3
If we really want to allow the deletion of the reference to an undeleted attribute, we can simply choose another graph structure signature with an extra sort for the references and an additional source and target operation representing source rsp. target of a reference. If we conversely would weaken the requirements on graph structure morphisms, we would loose the ability to express the (possibly recursive) deletion of dangling references by a single transformation rule.

Therefore we are forced by our applications, to implement graph structure homomorphisms exactly as they are defined. To model exactly the definition of an $\mathcal{GS}^{\mathcal{P}}_{SIG}$–morphism, we have to require that $f' \circ a$ is maximal with respect to $\leq$, so that $f' \circ a \leq b \circ f$. But this can easier be expressed by the commutativity of the diagram

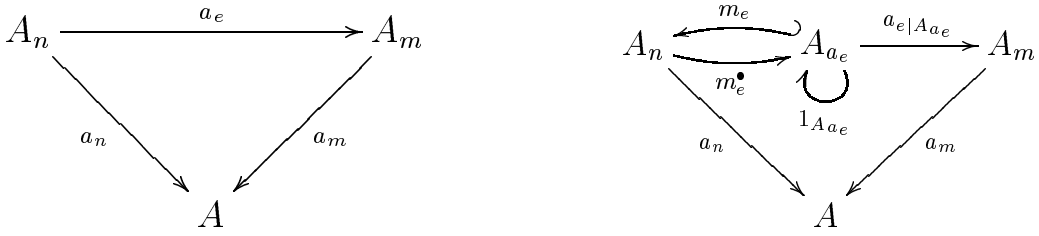


where morphism a is replaced by m_a, the inclusion of the scope of a into A, and $a_{|A_a}$, the total domain restriction of a.

[15] Graph transformation rules are represented by partial morphisms.
[16] Otherwise there would be no colimit preserving left adjoint between $\mathcal{GS}^{\mathcal{P}}_{SIG}$ and $\mathcal{GS}_{SIG}$.

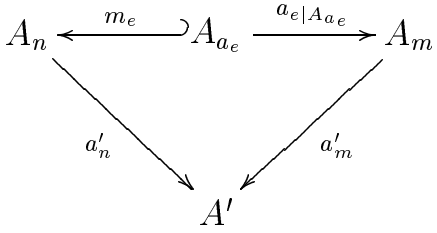
The question is now, how we can derive from our proof (and corresponding algorithm) for comma categories in 3.2 a proof for simple graph structures. The main difference is, that the commutativity requirement for morphisms is slightly weakened.

From lemma 2.2.12 we learned, that if we decompose the morphisms in a diagram Δ^A by replacing all morphisms by their span representations, sinks over Δ^A preserve their cocone rsp. colimit property:



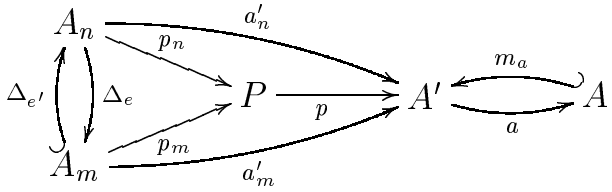
All morphisms a_e corresponding to an edge e between nodes n and m in the diagram are replaced by the inclusion m_e from its scope A_{a_e} into its domain A_n, its total domain restriction $a_{e|A_{a_e}}$ and the left inverse of m_e, the strong projection $m_e^\bullet$. Then $a_n : A_n \to A$ is colimit in the left diagram iff it is colimit in the right one.

What happens, if we remove all projections $m_e^\bullet$ from the diagram ?



How is the new colimit $a'_n : A_n \to A'$ related to A ? The next lemma will provide an answer.

LEMMA 4.2.4
Given a diagram Δ in $\mathcal{SET}^{\mathcal{P}}$ or $\mathcal{SET}^{\perp}$, diagram Δ' is obtained from Δ by removing any number of morphisms $\Delta_e : A_n \to A_m$ from Δ, such that Δ_e is a strong projection and its right inverse inclusion $\Delta_{e'} : A_m \to A_n$ is in Δ, that is, $\Delta_e \circ \Delta_{e'} = 1_{A_m}$ and $\Delta_{e'} \circ \Delta_e \leq 1_{A_n}$.



Let $p_n : A_n \to P$ be coproduct of Δ (and Δ'), let $a'_n : A_n \to A'$ be colimit of Δ' and $a \circ a'_n : A_n \to A$ be colimit of Δ, where p and a are given by the universal properties of $p_n : A_n \to P$ and $a'_n : A_n \to A'$. Then

1. *a is a projection, that is, has a right inverse embedding m_a with $a \circ m_a = 1_A$ and $m_a \circ a \leq 1_{A'}$.*

2. *If in addition both colimits are constructed as described in theorem 4.1.5 rsp. lemma 4.1.4, then m_a is an inclusion, that is, $A \subseteq A'$.*

Proof:

1. Follows immediately from 2., because colimits are unique up to isomorphism.

2. Let's choose $\mathcal{SET}^{\perp}$ first. Proof by induction over the number of deleted morphisms. If no morphisms were deleted we have $A' = A$ and $a = m_a = 1_A = 1_{A'}$.

   For the induction step we have to prove the case of one deleted morphism Δ_e. From lemma 4.1.4 follows $A = P_{/\sim^*_\Delta}$ and $A' = P_{/\sim^*_{\Delta'}}$. Since $\Delta_{e'} : A_m \to A_n$ is an inclusion we have $A_m \subseteq A_n$. From the definition of $\sim_e$ and lemma 4.1.1 follows for all $v, w \in P$, that

   $$v \sim_e w \quad \text{and} \quad v \neq w \quad \Rightarrow \quad w = - \tag{1}$$
   $$w \not\sim_{e'} v \quad \Rightarrow \quad v \neq w \tag{2}$$

   resulting in

   $$v \sim_e w \quad \text{and} \quad w \not\sim_{e'} v \quad \Rightarrow \quad w = - \tag{3}$$

   Suppose there are two elements $x, y \in P$ with $x \sim^*_\Delta y$ but $x \not\sim^*_{\Delta'} y$. Then there must exist two elements $v, w \in P$ with

   $$x \sim^*_\Delta v \sim_e w \sim^*_\Delta y \qquad \text{and} \qquad w \not\sim_{e'} v \tag{4}$$

   Together with (3) we have $w = -$, hence $x \sim^*_\Delta -$ and $y \sim^*_\Delta -$. That means for the equivalence classes in A and A' :

   $$x \not\sim^*_\Delta - \;\Leftrightarrow\; [x]_{\sim^*_\Delta} \neq - \quad \Rightarrow \quad [x]_{\sim^*_{\Delta'}} = [x]_{\sim^*_\Delta} \tag{5}$$

   that is, $A \subseteq A'$.

   For $\mathcal{SET}^{\mathcal{P}}$ we use the fact that both the completion functor $T_\perp : \mathcal{SET}^{\mathcal{P}} \to \mathcal{SET}^{\perp}$ and the restriction functor $R_\perp : \mathcal{SET}^{\perp} \to \mathcal{SET}^{\mathcal{P}}$ preserve strong projections and inclusions. In theorem 4.1.5 colimits in $\mathcal{SET}^{\mathcal{P}}$ are defined over colimits in $\mathcal{SET}^{\perp}$ by application of $R_\perp$. So $A \subseteq A'$ holds also in $\mathcal{SET}^{\mathcal{P}}$.

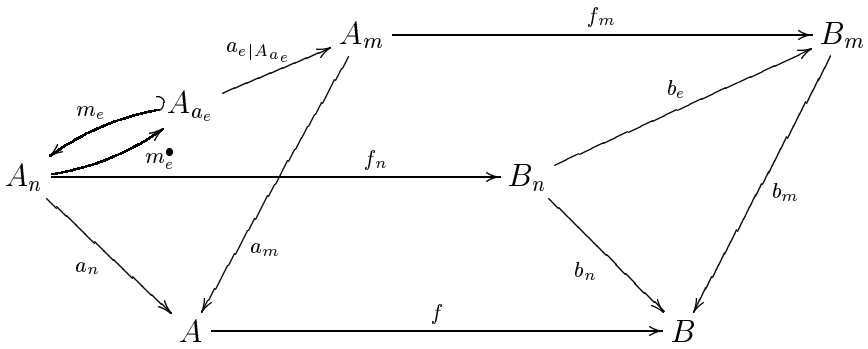
□

Now we show, how colimits in $\mathcal{GS}^{\mathcal{P}}_{op}$ can be computed:

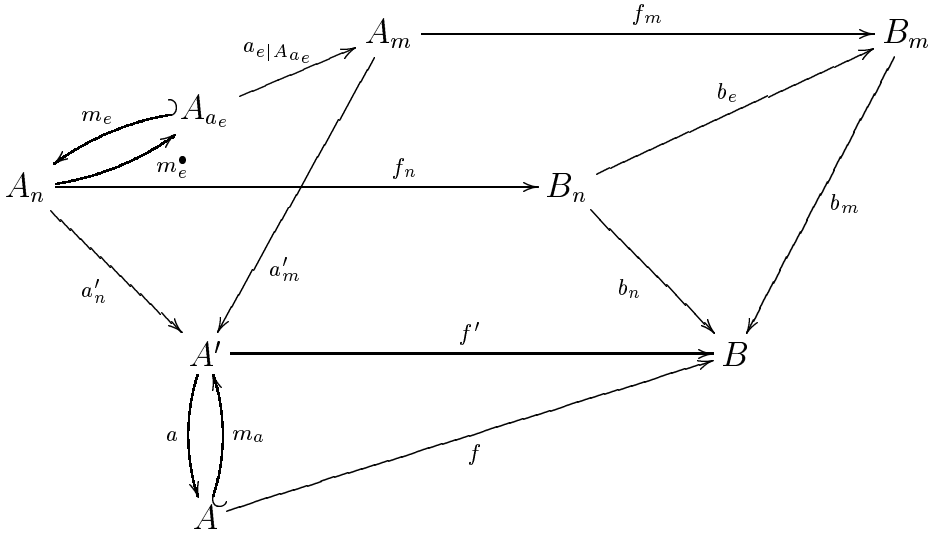
THEOREM 4.2.5 (COCOMPLETENESS OF $\mathcal{GS}^{\mathcal{P}}_{op}$)
The category $\mathcal{GS}^{\mathcal{P}}_{op}$ has all finite colimits.

Proof: Given a $\mathcal{GS}^{\mathcal{P}}_{op}$ diagram Δ, decomposed into two $\mathcal{SET}^{\mathcal{P}}$–diagrams Δ^A and Δ^B (one for each carrier set) with colimits $a_n : A_n \to A$ and $b_n : B_n \to B$, given by theorem 4.1.5.

To express the homomorphism conditions in $\mathcal{GS}^{\mathcal{P}}_{op}$, we decompose the morphisms a_e in Δ^A corresponding to an edge e between nodes n and m into its span representation,



that is, $a_e = a_{e|A_{a_e}} \circ m_e^\bullet$ and the inclusion m_e is right inverse to $m_e^\bullet$. Diagram $\Delta_\bullet^A$ is obtained from Δ^A by removing the left inverses $m_e^\bullet$ of the inclusions m_e for all edges $n \xrightarrow{e} m$ in Δ.



Let $a'_n : A_n \to A'$ be colimit in $\Delta_\bullet^A$ and $a_n = a \circ a'_n$. From lemma 4.2.4 we derive, that a has a right inverse inclusion m_a, that is, $a \circ m_a = 1_A$.

Since $(a_e, b_e) : (A_n \xrightarrow{f_n} B_n) \to (A_m \xrightarrow{f_m} B_m)$ represents a $\mathcal{GS}_{op}^{\mathcal{P}}$–homomorphism, we have for all edges $n \xrightarrow{e} m$ in Δ :

$$f_m \circ a_{e|A_{a_e}} \quad = \quad b_e \circ f_n \circ m_e \tag{1}$$

and since $b_n : B_n \to B$ is cocone in Δ^B :

$$b_m \circ b_e \quad = \quad b_n \tag{2}$$

which together results in

$$b_m \circ f_m \circ a_{e|A_{a_e}} \quad = \quad b_n \circ f_n \circ m_e \tag{3}$$

i.e, $b_n \circ f_n : A_n \to B$ is cocone in $\Delta_\bullet^A$. Then, from the universal property of the $\Delta_\bullet^A$–colimit $a'_n : A_n \to A'$ we have an unique morphism $(A' \xrightarrow{f'} B)$ with

$$f' \circ a'_n \quad = \quad b_n \circ f_n \tag{4}$$

By $f \circ a = f'$ we uniquely determine the morphism $(A \xrightarrow{f} B)$, since

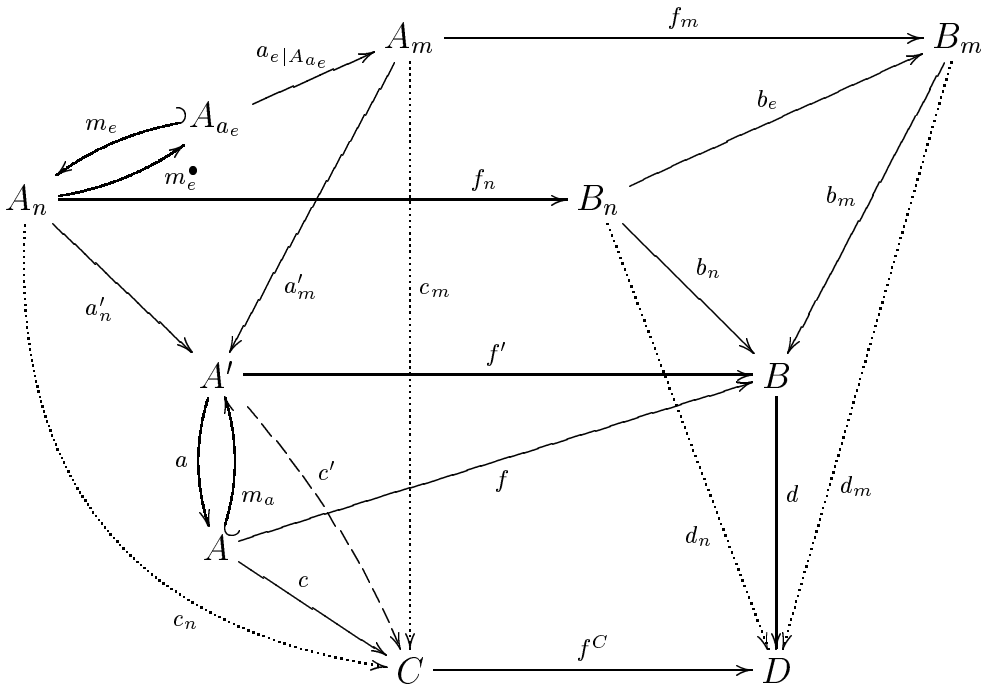
$$f \circ a = f' \;\Rightarrow\; f \circ a \circ m_a \;=\; f \circ 1_A = f = f' \circ m_a \tag{5}$$

Together with (4) we get

$$f \circ a \circ a'_n \;=\; f \circ a_n = b_n \circ f_n \tag{6}$$

which means, that $(a_n, b_n) : (A_n \xrightarrow{f_n} B_n) \to (A \xrightarrow{f} B)$ is cocone in Δ.

Suppose there is another cocone $(c_n, d_n) : (A_n \xrightarrow{f_n} B_n) \to (C \xrightarrow{f^C} D)$ in Δ.



From the universal properties of the $\Delta^A_\bullet$–colimit $a'_n : A_n \to A'$, the Δ^A–colimit $a_n : A_n \to A$ and the Δ^B–colimit $b_n : B_n \to B$ we derive unique morphisms c', c and d with $c' \circ a'_n = c_n$, $c \circ a_n = c_n$ and $d \circ b_n = d_n$.

Since $d_n : B_n \to D$ is cocone in Δ^B :

$$d_m \circ b_e \;=\; d_n \tag{7}$$

Together with (1), we get

$$d_m \circ f_m \circ a_{e|A_{a_e}} \;=\; d_n \circ f_n \circ m_e \tag{8}$$

i.e, $d_n \circ f_n : A_n \to D$ is cocone in $\Delta^A_\bullet$. Thus there is an unique morphism $h : A' \to D$ and we have

$$h = d \circ f' \;=\; f^C \circ c \circ a \tag{9}$$

together with (5) we derive

$$d \circ f = d \circ f' \circ m_a \quad = \quad f^C \circ c \circ a \circ m_a = f^C \circ c \circ 1_A = f^C \circ c \tag{10}$$

which means, that the unique Δ–morphism $(c, d) : (A \xrightarrow{f} B) \to (C \xrightarrow{f^C} D)$ is indeed a $\mathcal{GS}^{\mathcal{P}}_{op}$–homomorphism. □

REMARK 4.2.6

1. In the literature on graph structures [Löw93a, Kor96] we often find the characterisation of *hierarchical* graph structures, where for all compositions of operations $G^{op_1} \circ \cdots \circ G^{op_n} : G_s \to G_t$ holds $s \neq t$.

   Specially the simple graph structure $(A \xrightarrow{f} A)$ is not hierarchical. The main motivation for this characterisation is, that cocompleteness proofs are easier for hierarchical graph structures. In $\mathcal{GS}_{SIG}$ the totality requirement on the operations causes additional deletions, which were easier to handle for hierarchical graph structures. This problem doesn't occur with partial graph structures. From the proof of theorem 4.2.5 we learned, that colimit computation in $\mathcal{GS}^{\mathcal{P}}_{op}$ is modular. The $\mathcal{SET}^{\mathcal{P}}$–components of the colimit do not depend on the operation. For this reason we separated the totalisation phase in the colimit computation for $\mathcal{GS}_{SIG}$. We will see later, that non hierarchical finite graph structures can also easily be totali-zed.

2. In the proof above we only use the property of m_a and a, that they are an embedding/projection pair, that is, that $a \circ m_a = 1_A$ and $m_a \circ a \leq 1_{A'}$, and not, that m_a is an inclusion. This is not surprising, since our arguments are about morphisms, and not at the level of $\mathcal{SET}^{\mathcal{P}}$–elements. The reason, we have restricted us to inclusions, is that it exactly represents the situation in our implementation, it supports our intuition and it doesn't cause any theoretical overhead.

3. Since m_a is an inclusion, $f' = f_{A'}$ is the domain restriction of f to A'. Note that the a_n are total (since all m_e and $a_{e|A_{a_e}}$ are total), but normally not f', that is $A_f \subseteq A'$.

4. To prove the homomorphism property of the universal morphism $(c, d) : (A \xrightarrow{f} B) \to (C \xrightarrow{f^C} D)$ and of $(a_n, b_n) : (A_n \xrightarrow{f_n} B_n) \to (A \xrightarrow{f} B)$ we have shown that $d \circ f = f^C \circ c$ and $f \circ a = b_n \circ f_n$. It was not necessary to decompose c rsp. a_n into their span representations, since these squares directly commute. That means for our implementation, that colimit computation in $\mathcal{GS}^{\mathcal{P}}_{op}$ is very similar to colimit computation in the arrow category $(\mathcal{SET}^{\mathcal{P}}, \mathcal{SET}^{\mathcal{P}})$.

THEOREM 4.2.7 (COMPLEXITY OF $\mathcal{GS}^{\mathcal{P}}_{op}$–COLIMIT COMPUTATION)
Suppose application of a morphism to an element and adding a single relation to an operation takes constant time. Then there is an algorithm computing the colimit of a $\mathcal{GS}^{\mathcal{P}}_{op}$–diagram Δ, which has worst case running time $O(n + m \cdot \alpha(m, n))$, where m is the sum of the number of elements in all nodes of Δ, and n is the sum of the sizes of the relations induced by the morphisms in Δ.

Proof: Follows immediately from theorem 3.1.6 and from the construction of the colimit in the proof of theorem 4.2.5. □

THEOREM 4.2.8 (COCOMPLETENESS OF $\mathcal{GS}^{\mathcal{P}}_{SIG}$)
The category $\mathcal{GS}^{\mathcal{P}}_{SIG}$ has all finite colimits.

Proof: Let Δ be a diagram in $\mathcal{GS}^{\mathcal{P}}_{SIG}$ with $SIG = (S, OP)$. We decompose Δ into a set of sub-diagrams Δ^s in $\mathcal{SET}^{\mathcal{P}}$, one for each sort $s \in S$, and a set of sub-diagrams Δ^{op} in $\mathcal{GS}^{\mathcal{P}}_{op}$, one for each operation $op \in OP$. Objects in Δ^{op} at node n are denoted as $(\Delta^s_n \xrightarrow{op^{\Delta n}} \Delta^t_n)$ where $op \in OP_{s,t}$. Let $c^s_n : \Delta^s_n \to C_s$ for all $s \in S$ be colimit in Δ^s and $(c^s_n, c^t_n) : (\Delta^s_n \xrightarrow{op^{\Delta n}} \Delta^t_n) \to (C_s \xrightarrow{op^C} C_t)$ for all $op \in OP_{s,t}$ be colimit in Δ^{op} given by theorem 4.1.5 and theorem 4.2.5.

Then $c_n : \Delta_n \to C$ is colimit of Δ. Note that in Δ^{op} the $\mathcal{SET}^{\mathcal{P}}$–components of the colimit are $\mathcal{SET}^{\mathcal{P}}$–colimits, [17] thus the definition of the colimit of Δ is consistent.

$c_n : \Delta_n \to C$ is cocone of Δ, since each component $c^s_n : \Delta^s_n \to C_s$ is colimit and the c_n are $\mathcal{GS}^{\mathcal{P}}_{SIG}$–homomorphisms by application of theorem 4.2.5 for each operation $op \in OP$.

Suppose there is another cocone D of Δ. Then we construct the universal morphism $h : C \to D$ component-wise for each sort using the construction in the proof of theorem 4.1.5. Its uniqueness follows from the uniqueness of the h_s for all $s \in S$. Application of theorem 4.2.5 for each operation $op \in OP$ ensures, that h is indeed a $\mathcal{GS}^{\mathcal{P}}_{SIG}$–homomorphism. □

THEOREM 4.2.9 (COMPLEXITY OF $\mathcal{GS}^{\mathcal{P}}_{SIG}$–COLIMIT COMPUTATION)
Suppose application of a morphism to an element and adding a single relation to an operation takes constant time. Then there is an algorithm computing the colimit of a $\mathcal{GS}^{\mathcal{P}}_{SIG}$–diagram Δ, which has worst case running time $O(n + m \cdot \alpha(m, n))$, where m is the sum of the number of elements in all nodes of the diagrams Δ^s with $s \in S$, and m is the sum of the sizes of the relations induced by the morphisms in Δ.

Proof: Follows immediately from theorem 4.2.7 and from the construction of the colimit in the proof of theorem 4.2.8. □

COROLLARY 4.2.10 (COCOMPLETENESS OF $\mathcal{GS}_{SIG}$)
The category $\mathcal{GS}_{SIG}$ has all finite colimits.

Proof: Follows immediately from theorem 4.2.8 and theorem 2.4.13. □

Theorem 2.4.13 doesn't tell us, how we can compute the largest total subalgebra $G^T = T(G)$ of a partial graph structure G.

REMARK 4.2.11 (COMPLEXITY OF $\mathcal{GS}_{SIG}$–COLIMIT COMPUTATION)
Complexity of colimit computation in $\mathcal{GS}_{SIG}$ depends on theorem 4.2.9 and on the complexity of the computation the largest total subalgebra of the computed $\mathcal{GS}^{\mathcal{P}}_{SIG}$–colimit.

[17] This fact doesn't hold for total graph structures.

The following naive algorithm computes G^T:

G^T := G
repeat
  bool total := true
  for all $s \in S$ for all $x \in G_s$
    for all $t \in S$ for all $op \in OP_{s,t}$
      if ($op^G(x)$ undefined or $op^G(x) \notin G_t^T$) then
        $G_s^T := G_s^T - x$
        total := false
until total = true

In this pseudo-code the scopes of the *repeat until*, *for all* and *if then* statements are determined by indentation, which is sufficient for our small examples.

Since we don't have a hierarchy constraint on graph structures as defined in [Löw93a, Kor96], there may be recursive dependencies. The deletion of an element may cause an operation to become undefined for another element, which then also has to be deleted. The computation must continue, until there is no new element deleted during an iteration. Obviously this algorithm is extremely time consuming.

REMARK 4.2.12 (HIERARCHICAL GRAPH STRUCTURES)
Note that we don't have a hierarchy constraint, that is, allow compositions of operations $G^{op_1} \circ \cdots \circ G^{op_n} : G_s \to G_t$ with $s = t$. From the theoretical point of view everything is fine, but for the implementation a significant optimisation of the naive algorithm described above is desperately needed.

In the following definition we characterise G^T in a way, which enables an efficient implementation.

DEFINITION 4.2.13 (THE RELATION $\xrightarrow{OP^G}{}^*$ AND THE S–SORTED SET D^G)
Given a partial graph structure G in $\mathcal{GS}_{SIG}^{\mathcal{P}}$ with $SIG = (S, OP)$. Then

$$U^G = \{(x, s) \mid x \in G_s \text{ for some } s \in S\}$$

denotes the disjoint union of all elements in $(G_s)_{s \in S}$. The relation $\xrightarrow{OP^G}$ on U^G is defined as

$$(x, s) \xrightarrow{OP^G} (op^G(x), t) \quad \Leftrightarrow \quad x \in G_s \text{ and } op \in OP_{s,t}$$

Let $\xrightarrow{OP^G}{}^*$ denote the transitive reflexive closure of $\xrightarrow{OP^G}$. The S–sorted set $D^G = (D_s^G)_{s \in S}$ is defined as

$$D_s^G = \{x \in G_s \mid \quad \exists y \in G_{s'}, \exists op \in OP_{s',t} \text{ with } (x, s) \xrightarrow{OP^G}{}^* (y, s') \text{ and } op^G(y) \text{ undefined}\}$$

LEMMA 4.2.14 (LARGEST TOTAL SUBALGEBRA)
The subalgebra $G^T \subseteq G$ is defined as $G^T_s = G_s - D^G_s$. Then G^T is the largest total subalgebra of G.

Proof:

1. First we show, that G^T is total. Suppose $G^T = G - D^G$ is not total, then there must exist a $x \in G^T_s$ and an $op \in OP_{s,t}$ with $op^{G^T}(x)$ undefined.

   - Then, if also $op^G(x)$ is undefined, we derive $x \in D^G_s \Rightarrow x \notin G^T_s$.
   - Otherwise we have $op^G(x) \notin G^T_t$, then follows from the definition of D^G, that $op^G(x) \in D^G_t \Rightarrow x \in D^G_s \Rightarrow x \notin G^T_s$.

   Both cases contradict our assumption $x \in G^T_s$, thus G^T must be total.

2. Suppose there is a subalgebra $G' \subseteq G$ with $G' \cap D^G \neq \emptyset$. Then we have an $x \in G'_s$ with $x \in D^G_s$. From the definition of D^G follows, that then there must exist a $y \in G'_{s'}$ with $(x,s) \xrightarrow{OP^G}{}^*(y,s')$ and an $op \in OP_{s';t}$ with $op^{G'}(y)$ undefined, since the scopes of the operations in G' are subsets of the scopes of the operations in G. That means that G' is not total, that is, that G^T is the largest total subalgebra of G.

□

REMARK 4.2.15 That means, that following theorem 2.4.13 we can compute the colimit of a $\mathcal{GS}_{SIG}$–diagram with nodes G_n by computing the colimit $g_n : G_n \to G$ in $\mathcal{GS}^{\mathcal{P}}_{SIG}$ and then deleting all elements in D^G from G, since the nodes G_n are objects also in $\mathcal{GS}^{\mathcal{P}}_{SIG}$.

The main idea, to make an algorithm computing G^T efficient, is to separate the computation of the relation $\xrightarrow{OP^G}$ from the computation of D^G.

We represent $\xrightarrow{OP^G}$ by a function $d^{OP} : U^G \to \mathcal{P}(U^G)$ where $\mathcal{P}(G)$ denotes the powerset of U^G. We define d^{OP} by $d^{OP}(x,s) = \{(y,t) \in U^G \mid (y,t) \xrightarrow{OP^G} (x,s)\}$. An algorithm computing d^{OP} is given by

```
for all s ∈ S   for all x ∈ G_s
    d^OP(x,s) := ∅
for all s ∈ S   for all x ∈ G_s
    for all t ∈ S   for all op ∈ OP_{s,t}
        d^OP(op^G(x),t) := d^OP(op^G(x),t) ∪ (x,s)
```

We represent G^T by a function $t^G : U^G \to \mathit{bool}$ where

$$t^G(x,s) = \begin{cases} \mathit{true} & \text{if } x \notin D^G_s \\ \mathit{false} & \text{if } x \in D^G_s \end{cases}$$

which is computed by the following algorithm:

```
for all s ∈ S   for all x ∈ G_s
    t^G(x, s) := true
for all s ∈ S   for all x ∈ G_s
    for all t ∈ S   for all op ∈ OP_{s,t}
        if (op^G(x) undefined) then
            delete(x, s)
```

where the recursive procedure delete, which realizes the transitive reflexive closure $\xrightarrow{OP^G}{}^*$ of $\xrightarrow{OP^G}$, is defined as

```
delete(x, s) :
    if (t^G(x, s) = true) then
        t^G(x, s) := false
        for all (y, t) ∈ d^OP(x, s)
            delete(y, t)
```

Finally we obtain $G^T = G - D^G$ from

```
G^T := G
for all s ∈ S   for all x ∈ G_s
    if (t^G(x, s) = false) then
        G^T_s := G^T_s − x
```

Correctness of this algorithm follows immediately from lemma 4.2.14 and from the definitions of d^{OP} and d^G. Its performance depends on the chosen representation for d^{OP} and for sets and operations in general. We will show later, that our implementation has in practice linear complexity related to the number of elements in G. The main reasons of the performance gain compared with the naive algorithm are:

1. The function $d^{OP} : U^G \to \mathcal{P}(U^G)$ allows fast access to elements which depend on deleted ones.

2. In the procedure delete we avoid unnecessary recursion by testing in advance, whether an element was already visited by another call of delete. In our implementation we have further optimised delete by avoiding recursive calls also in situations, where we know, there will be a delete–call for the same element later. For this optimisation an order relation on G_s is needed, corresponding to the sequence, the iteration is performed by the notion *for all* $x \in G_s$.

4.3 ALPHA Algebras

Graph transformation systems provide an intuitive description for the manipulation of graphical structures as they occur in different areas of computer science like distributed systems, programming language semantics and information systems. The algebraic approach has been successfully applied not only to formalise the transformations, but also their structuring both in horizontal and vertical direction [CMR+96, EHK+96]. Vertical structuring describes the relations between different levels of abstraction or between a specification and its implementation. Horizontal structuring concepts formalise the relations of different parts of a large system. Other approaches to graph transformations are based on set theory [Roz87, Nag87]. They are used for instance in the PROGRES system [Sch91b]. These approaches will not further be investigated here because the colimit library cannot be applied to them. See for example [Tae96a] for a comparison with the algebraic approaches.

In typed graph transformation systems [CEL+96, CL95] the usual labelling of nodes and edges is replaced by typing morphisms. The typing significantly extends the expressive power of the transformation formalism allowing them to model various kinds of specification techniques like entity- relationship schemes or petri nets. This makes it possible to carry over compositionality results related to horizontal and vertical refinement steps to the different kinds of specifications they can model.

Information systems may be described by semantic data models [HK87], object-oriented models [Boo94b, RBP+91b, JCJÖ93] or Petri nets [Rei85]. All these techniques support graphical visualisation where different levels of abstraction (for example models, schemes and instances) help to structure the modelled systems. There are different approaches formalising these models (see e. g. [GH91]), where the algebraic approach [CLWW94, CL95] has the advantage, that it integrates both the static and dynamic aspects. Here we want to demonstrate, that it not only provides a general frame for the formalisation of graphical modelling techniques supporting scheme evolution simultaneously on different abstraction levels, but also can guide the design of restructuring and modelling tools.

The ALPHA library [EC96] was designed for that purpose separating functionality already existing in graph manipulation systems like PROGRES [Sch94], GraphED [Him88], and AGG [LB93].

Advanced software engineering techniques, for instance design patterns [GHJV95], together with object oriented modelling techniques were used to make the ALPHA library as flexible and extendible as possible. ALPHA algebras are general enough to support different graphical modelling techniques. They will be used as basis of the implementation of a general scheme transformation tool, in a new version of the AGG-system and in a tool for high level replacement systems.

In its first version the applicability of the ALPHA library was restricted. Several features like the support of non-injective graph transformation rules [18] or general colimits were not implemented. These problems were fixed owing to the integration of the colimit library into the ALPHA library.

In this section we want to present the theoretical concept of an ALPHA algebra and

[18] Non-injective rules realize the concept of identification of different nodes or edges

its relation to general graph structures. In chapter 4 also the implementation of colimits for ALPHA algebra diagrams will be discussed.

4.3.1 Attributed Graph Signatures

The concept of ALPHA algebras is based on attributed graph signatures in the sense of [LKW93]. An *attributed graph signature* consists of three parts

- a graph signature SIG_{ITEM} representing the graph part,
- an unrestricted algebraic signature SIG_{DATA} representing the data part,
- a $SIG_{\text{ITEM}} \times SIG_{\text{DATA}}$ indexed family of attribute operations OP_{ATTR}, unary operations from graph sorts into data sorts representing the attribute part.

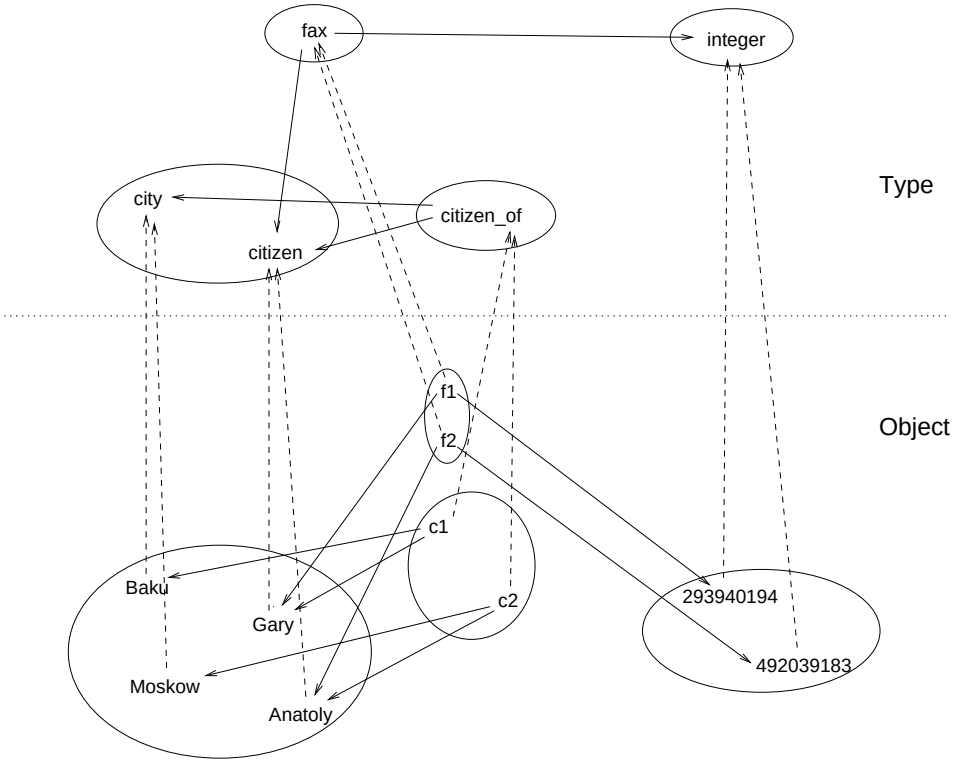
Algebras to attributed graph signatures are called *AGS-algebras* . Their operations are total for the graph and attribute part and partial for the data part. Their homomorphisms, called *AGS-homomorphisms* , are partial for the graph part and total for the data part and must be compatible with the attribute operations.

Intuitively spoken, the category of *AGS-algebras* is cocomplete because the colimit for the graph part and the data part may be constructed separately [19], the AGS-homomorphisms are compatible with the attribute operations and the attribute operations map graph sorts on data sorts, that is, all terms built from attribute operations are finite.

An interesting observation from [LKW93] is, that we gain a lot of expressive power considering total AGS-homomorphisms instead of AGS-algebras as objects of the category used for modelling scheme transformations. The resulting category of *typed AGS-algebras* (also called object structures) is constructed similar to a comma category. The type part is considered to be fixed during the transformations and due to the partiality of the morphisms for the graph part the corresponding compatibility requirement is somewhat weakened.

Total AGS-homomorphisms represent typed objects where the source corresponds to the untyped object, the target to the type and the morphism itself to the typing. Consider for example a little data base storing citizens, cities and fax numbers which contains the citizens $\{Gary, Anatoly\}$, the cities $\{Baku, Moskow\}$, a *citizen_of* relation between these sets and some fax numbers as attributes. This data base may be represented as typed AGS-algebra as follows:

[19] Their corresponding categories, namely graph structures with partial morphisms and algebras with total morphisms, are cocomplete



The type AGS-algebra contains $\{city, citizen, citizen\_of\}$ in the graph part, $\{integer\}$ in the data and $\{fax\}$ in the attribute part, where the object AGS-algebra consists of concrete values according to these types. Note that this way many sorted algebras may be "internalised" by the typing morphism.

Transformation rules for typed AGS-algebras modify the represented model simultaneously at different levels of abstraction such that compatibility with the typing is preserved. We don't want to repeat here their formal definition. Important for us is, that they rely on colimit computations. Pushouts are sufficient for simple rules, but if parallel transformations or a shared context have to be treated, we need arbitrary colimits. Similar to the construction in comma categories, we can compute the colimit separately for the objects and their types and then construct the unique extension induced by the object colimit to the type colimit which represents the typing of the colimit for the object structures.

4.3.2 The Category of ALPHA Algebras

We made the observation, that for most practical applications of this theory the data part of an attributed graph signature represents the abstract interface to a library with predefined datatypes. Therefore it is sufficient to consider the data part as fixed. That means, that AGS-homomorphisms leave the data part unchanged. This assumption simplifies both the theoretical concepts and the tool development. Now the data part of the morphisms and the data operations are no longer relevant for the transformations and for the colimit computations. Regarding tool design questions we can simply omit them.

Next we can ask, whether we really need more than one sort in the graph part, since the (hierarchical) typing concept subsumes their purpose. Last but not least we could simplify

the naming of the operations in the graph and attribute part. All these considerations together lead directly to the definition of an ALPHA *algebra* as a graph structure to the signature $SIG = (S, OP)$ with

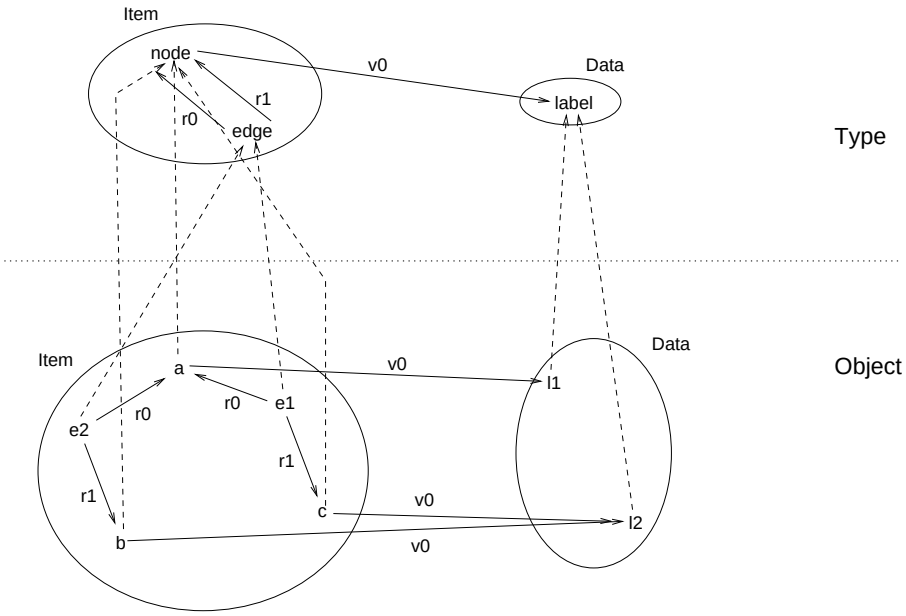
$$S = \{ITEM, DATA\} \text{ and}$$

$$OP \subseteq \bigcup_{i \in \mathbb{N}} r_i : ITEM \to ITEM \quad \cup \quad \bigcup_{i \in \mathbb{N}} v_i : ITEM \to DATA$$

We call the operations $r_i : ITEM \to ITEM$ references and the $v_i : ITEM \to DATA$ attribute assignments. References are required to be total (to avoid dangling edges), attribute assignments may be partial. The carrier set for *ITEM* together with the references r_i represents the graph part, the carrier set for *DATA* the data part and the attribute assignments v_i the attribute part of the ALPHA algebra. ALPHA algebra morphisms (short ALPHA morphisms) are partial graph structure morphisms with the restriction, that the data part is fixed, that is, the carrier sets for sort *DATA* are mapped identically on their target (see also [EC96]).

Note, that in opposition to AGS-algebras ALPHA attribute assignments may be partial. Since we have only one sort in the graph part, there is no possibility to separate the items equipped with attributes. Since ALPHA algebras are special graph structures we can apply our theory for the computation of colimits on them. In the theoretical part of this thesis we considered only graph structures where all operations were either total or partial. But it is obvious, how graph structure categories with "mixed" operations have to be treated: The totalisation has to be applied only to the operations which are required to be total. Applied to ALPHA algebras, that means, that the references are totalised, but not the attribute assignments. We can even imagine applications were the totalisation of the references is not appropriate. Therefore the colimit library provides also a colimit algorithm for ALPHA algebras without totalisation. The indexed notation for references and attribute assignments suggests their realization via some array data type as indeed it is done in the colimit library.

Again the morphisms representing typed ALPHA algebras are the objects we are interested in. Note that even simple graphs cannot be modelled without typing since we have only a single sort *ITEM* in the graph part. The following picture shows a typed ALPHA algebra representing a small graph with labelled nodes.



The references r_0 and r_1 represent the source and target functions, v_0 the node labelling. The attentive reader may have noticed, that the references r_0 and r_1 are required to be total, but seem to be shown as partial operations in the picture above. As default references map items identically on them self, the corresponding "loops" were omitted in the picture to enhance its readability. So references are indeed total to enable the removal of dangling edges during transformation steps.

4.4 Example Applications

The colimit library currently doesn't provide direct support of general graph structures, since typed ALPHA algebras are easier to implement and general enough to enable the applications we are interested in. In this section we present a brief overview about the ongoing developments based on the colimit library.

4.4.1 Entity-Relationship Scheme Transformations

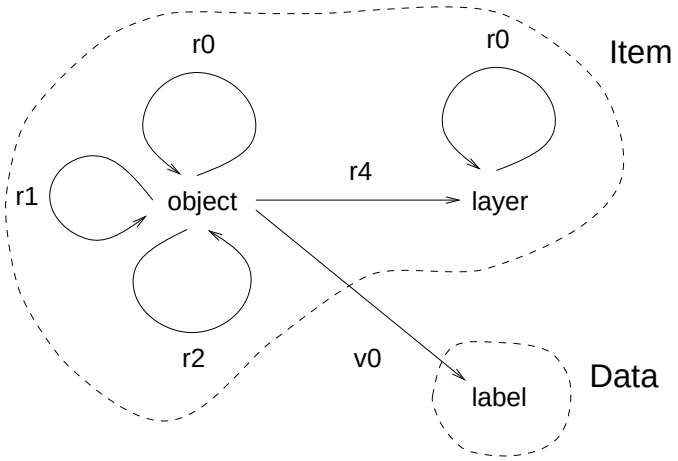
In [CL95] the modelling of scheme evolution generalised using algebraic graph transformation techniques is described for typed AGS-algebras, [CEL+96] presents a slightly different formalisation using the double pushout approach. In a software project at the Technical University of Berlin students applied this formalism to a small data base application. In a first step ER-diagrams were transformed into equivalent typed AGS-algebras which can further be transformed into typed ALPHA algebras. It is planned to build graphical tools based on the ALPHA library (which includes the colimit library) supporting data base scheme transformations on different levels of abstraction.

4.4.2 The AGG-System

The AGG-system provides comprehensive functionality for the generation and manipulation of graphs with a modern mouse driven interface. It supports hierarchical graphs, that means, graphs describing the same thing at different levels of abstraction, and higher order edges (edges between edges). These graphs are called AGG-graphs. In this model graphs, transformations and occurrences can be described in a uniform way.

In AGG-graphs we have, in addition to the source and target functions defined on edges, an abstraction and a labelling function defined both for nodes and edges. Levels of abstraction are represented by a set of layers. Since there are edges between edges, the distinction between nodes and edges is weakened. The set consisting both of nodes and edges is called set of objects. A mapping assigns a specific layer to each object. Abstractions are defined both for objects and layers. There are several compatibility requirements for the definition of these functions. The set of objects contains a specific —-element which allows the source and target functions to be total on the set of objects.

Up to now there exists no implementation of general colimits in the AGG-system. This problem is currently fixed by the integration of the colimit library. Again we can use the colimit computation for ALPHA algebras. For this purpose we have to find an ALPHA representation of AGG-graphs. Here we will outline the idea.



The picture above presents a possible ALPHA type algebra for AGG-graphs. r_0, r_1 and r_2 correspond to the abstraction, source and target functions, r_4 represents the assignment of layers to objects and v_0 is the labelling function. Note that although not shown in the picture, since all references are total, r_1, r_2 and r_3 are defined also for $layer$. This shows that the expressive power of the typing mechanism of ALPHA algebras is limited, if we have to deal with self-referencing graph operations. At the type level we cannot distinguish between defined and undefined references. At the level of the typed ALPHA algebra representing the modelled AGG-graph this is no problem, since here self-referencing graph operations are not allowed, for instance, an edge must not be its own abstraction.

With this typing AGG-graphs can be represented as typed ALPHA algebras and the colimit library can compute their colimit. The resulting typed ALPHA algebra then has to be converted back into an AGG-graph. It has to be proven that the result of the

conversion fulfils all compatibility requirements and is indeed colimit in the category of AGG-graphs. Up to now there exists no formal cocompleteness proof for AGG-graphs.

4.4.3 Algebraic High Level Nets

Algebraic high level nets combine specification techniques from petri nets, algebraic specifications and high level replacement systems. They are special high level petri nets, like coloured petri nets [Jen92, Jen95] and predicate/transition nets [GL81, Gen91], used for the specification of distributed and concurrent systems. In algebraic high level nets every token may contain a structured set of data, describing for instance the complete state of a process or data base. An algebraic high level net consists of two main components,

- the static part describing the needed data types via an algebraic specification,
- the dynamic part describing the possible transitions.

The net structure of an algebraic high level net consists of places, representing the possible states of a system and the corresponding (state) transitions connected to the places via directed edges. This is consistent with the structure of classical petri nets. But here the edges are labelled with terms specifying the data structures which are moved around the net, and the transitions are equipped with a set of axioms specifying the condition under which they are activated.

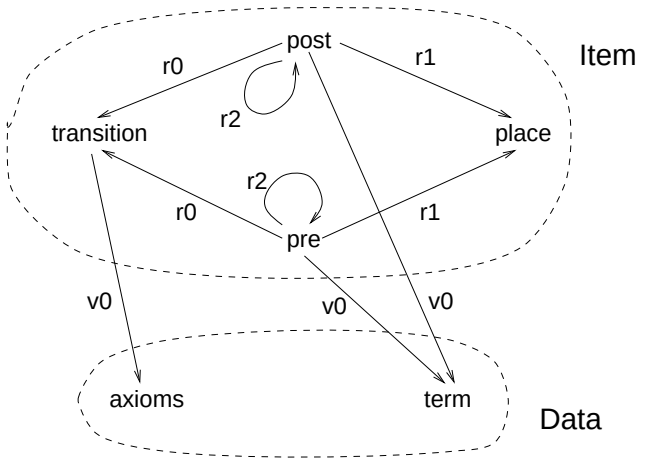
As for algebraic graph transformations there exist horizontal and vertical structuring mechanisms together with the corresponding compatibility results. These mechanisms and high level net transformations rely on colimit computations. The cocompleteness of the category of algebraic high level nets is proved by a combination of the cocompleteness results for the dynamic and the static part of the net. Again the structure of the cocompleteness proof may guide the implementation of net transformations which are performed by colimit computations. In a first approach we chose to implement the transformations of the dynamic part (the places and transitions) which can easily be represented as an ALPHA algebra. Later the tool may be extended to cover also transformations of the static part. This is complicated by the fact, that here we have to deal with morphisms between algebras to different specifications.

The theory of algebraic high level nets is formulated using the "double pushout approach" for algebraic transformation rules [CMR+96]. That means, that we have two choices,

- either to convert the span representations of the transformation rules into equivalent partial morphisms supported by the colimit library,
- or to provide an implementation of the pushout complement and the check of the application conditions and use the colimit library only to compute the second pushout.

The first choice has the advantage, that the colimit library itself can check the "dangling" and "identification" conditions, which test whether a rule application may lead to dangling references on deleted places or to ambiguous deletions.

Now we want to show, how algebraic high level nets may be represented as typed ALPHA algebras.



The picture above illustrates the type algebra. The transitions and places are in the graph part, the axioms and terms in the data part. The directed edges for the pre and post ranges of the transitions are represented by the elements *pre* and *post*. v_0 points to the set of axioms assigned to a transition as application condition and to the terms labelling the edges. r_0 and r_1 denote the source transition rsp. target place of the edges. Since the edges are ordered, we need a reference pointing to the next edge which is represented by r_2. Not all alpha algebras of this type represent algebraic high level nets since we could for instance misuse reference r_2. But we can prove, that if we have only ALPHA algebras representing high level nets in a diagram, its colimit is again the representation of a high level net.

With this approach only the dynamic part of the algebraic high level net colimit is computed since ALPHA algebra morphisms leave the data part unchanged. The question is now, how the colimit library should represent the computed colimit, such that later an extension for the data part may be added. The colimit computation identifies some places and transition by building equivalence classes. The computed colimit algebra (net) contains only one representative element for each equivalence class. For the attribute assignments one representative data element per class would not be sufficient. Therefore the library provides the full set of attributes assigned to all members of an equivalence class. These may for instance later be used to compute compatibility requirements [20] for the applicability of transformation rules.

4.4.4 Simulation of Narrowing

Another application of the colimit algorithm is the simulation of an abstract machine for the execution of functional logic programming languages. Based on previous work with Montanari, Ehrig and Löwe [CMR+91], Corradini and Rossi proposed in [CR93] *hyperedge replacement jungle rewriting* as a formalism well suited for the modelling of logic

[20] For the terms assigned to the edges from transitions to places an unification procedure may be applied.

programming [Llo87]. Pushouts in the category of jungles correspond to term unification. Hence definite clauses may be represented as graph transformation rules and it was proved that jungle derivations based on the double pushout approach faithfully correspond to the resolution calculus. In [CW94] Corradini and the author of this thesis extended this work by modelling the behaviour of an abstract machine for lazy narrowing [Wol91] by hyperedge replacement jungle rewriting at three different levels of abstraction.

Narrowing subsumes rewriting and unification, and can be efficiently implemented by combining compilation techniques from logic and functional programming [Pad88, War83, Joh87, Loc92]. Its purpose is to find solutions for sets of equations (*goals*), i.e., substitutions that unify the left- and right-hand sides of all the equations in the set, or to detect that there is no solution. Narrowing can be seen as a combination of reduction and unification: non-variable prefixes of reduction redices are completed to full redices by substituting into variables. Its main difference to term rewriting is that narrowing makes full use of unification while term rewriting uses only pattern matching. But unification becomes pattern matching if the rewritten terms are ground, as it is usually the case in term rewriting.

Jungles are special hypergraphs that are suited to represent (collections of) terms, making explicit the sharing of common sub-terms. They are essentially equivalent to *directed acyclic graphs*, as discussed in [CMR+91]. In this section we introduce the category of jungles (see also [HP91]) and the basic definitions of jungle rewriting, which is a specific graph rewriting formalism defined along the lines of the "double-pushout" approach ([EPS73, CMR+96]).

Now we present the formal definitions for hypergraphs and jungles:

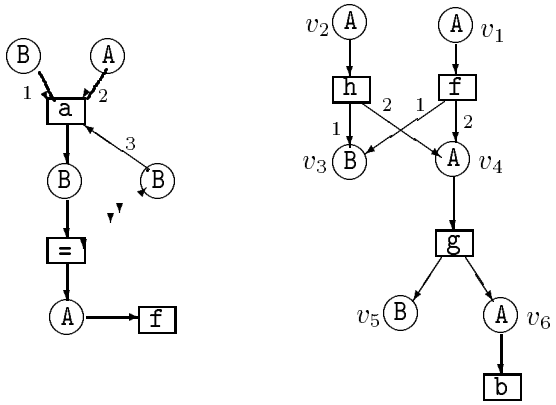
DEFINITION 4.4.1 (HYPERGRAPHS)

A *hypergraph* H *over* SIG is a tuple $H = (V, E, src, trg, m, l)$, where V is a set of *nodes*, E is a set of *hyperedges*, src and $trg : E \to V^*$ are the *source* and *target* functions, $l : V \to S$ maps each node to a sort of SIG, and $m : E \to OP$ maps each edge to an operator of SIG. For a hypergraph H, $indegree_H(v)$ (resp. $outdegree_H(v)$) denotes the number of occurrences of a node v in the target strings (resp. source strings) of all edges of H.

DEFINITION 4.4.2 (JUNGLES)

A *jungle over* SIG is an acyclic hypergraph $J = (V, E, src, trg, m, l)$ over SIG such that 1) for each node $v \in V$, $outdegree_J(v) \leq 1$, and 2) the labelling of the edges is consistent with both the number and the labelling of the connected nodes. The *roots* of a jungle are defined as $ROOTS(J) = \{v \in V_J \mid indegree_J(v) = 0\}$.

As an example consider the following signature $SIG = (S = \{\mathtt{A}, \mathtt{B}, \mathtt{bool}\}, OP = \{OP_{\mathtt{B}\cdot\mathtt{A},\mathtt{A}} = \{\mathtt{g}, \mathtt{h}, \mathtt{f}\}, OP_{\mathtt{A}\cdot\mathtt{A},\mathtt{bool}} = \{=\}, OP_{\mathtt{B}\cdot\mathtt{B},\mathtt{bool}} = \{=\}\})$. In the following figure two hypergraphs over SIG are shown. Edges and nodes are depicted as box and circles, respectively, with the label drawn inside and sometimes (for the need of reference) with a unique name depicted outside.



The right hypergraph is also a jungle while the left one is not. The categories of hypergraphs and jungles are defined as follows:

DEFINITION 4.4.3 (CATEGORIES $HGraph_{\mathrm{SIG}}$ AND $Jungle_{\mathrm{SIG}}$)

A *morphism of hypergraphs (over* SIG*)* $f : H_1 \to H_2$ consists of a pair of functions $f = (f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ between edges and nodes respectively, which are compatible with the source and target functions and which are label preserving.

Morphisms of jungles are defined exactly in the same way. Because of the structure of jungles, a morphism is uniquely determined by its behaviour on roots. That is, if $f, g : J \to J'$, $f = g \Leftrightarrow f_V(v) = g_V(v)\ \forall v \in ROOTS(J)$ [HP91].

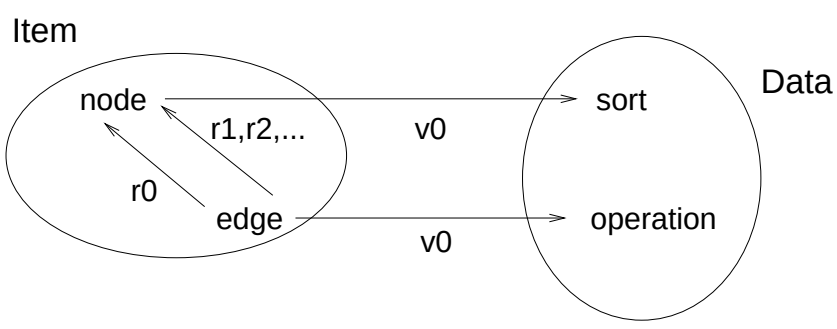
The category whose objects are hypergraphs over SIG and whose arrows are hypergraph morphisms will be denoted by $HGraph_{\mathrm{SIG}}$. The full sub-category of $HGraph_{\mathrm{SIG}}$ including all jungles will be called $Jungle_{\mathrm{SIG}}$.

In [CR93] it has been shown that unification of terms may be simulated by jungle pushout computations. In [CW94] this property of the category of jungles in used to model the behaviour of an abstract machine for lazy narrowing [Wol91] on different levels of abstraction.

Here we are mainly interested to show, that the colimit library may be used to compute pushouts in the category of jungles. Then a simulation tool for testing the behaviour of the abstract machine may be supported by the colimit library.

Because the theory in [CW94] is formulated using the double pushout approach, the span representations of the hyperedge replacement rules defined there has to be converted into equivalent partial morphisms, if we want to apply the colimit library.

A suitable ALPHA type algebra is defined as:



Here the reference r_0 represents the (single) source, references $r_1, r_2, \ldots$ the target nodes and v_0 assigns sorts to nodes and operations to edges. In practice only a finite subalgebra of this type algebra is needed, since jungle edges have only a finite number of target nodes. Because the sorts and operations are left unchanged by jungle morphisms, they are modelled exactly by the corresponding typed ALPHA algebra morphisms. Unfortunately the characterisation of jungles with $outdegree_J(v) \leq 1$ cannot be modelled by the ALPHA algebra representation. That means, that the computed typed ALPHA algebra colimit represents the colimit in the category of hypergraphs, not in the category of jungles.

The category of jungles is not cocomplete which expresses in our application that two terms have no common unificator. The computed hypergraph colimit can be used to check whether the corresponding jungle colimit exists. For all nodes v with $outdegree_J(v) > 1$ the corresponding edges have to be compared. If they are not equal, the colimit jungle does not exist. Otherwise the comparison proceeds recursively with the targets w of these edges for which $outdegree_J(w) > 0$. If in this way no differences were found, i.e., the represented terms at position v are equal, the redundant edges have to be removed to convert the hypergraph colimit into the corresponding jungle colimit.

5 Implementation of the Colimit Library

This section presents the ideas behind the implementation of the colimit library. After introducing the main design criteria we reveal its structure and its interface. Then we explain, how the library may be embedded in an example application. Different object oriented programming languages are analysed and compared with respect to their suitability for our purposes motivating our choice. We present a short introduction into the concepts of component programming showing their relevance for our implementation. Then we show, how the mathematical concepts are mapped to concrete data types by correlating the requirements of the colimit computation with specific properties of different container classes. We explain the representations of sets, morphisms, attributed graphs, signatures, ALPHA algebras and of the diagrams of these structures. Next some excerpts of the code are presented. We restrict us to the most fundamental function definitions. For a more complete overview we refer to the HTML-documentation of the code, available via ftp from ftp.cs.tu-berlin.de. Implementation aspects specific to our chosen programming languages are discussed. Finally we analyse benchmark results, compare different compilers, languages, libraries and garbage collection methods.

5.1 Design Criteria

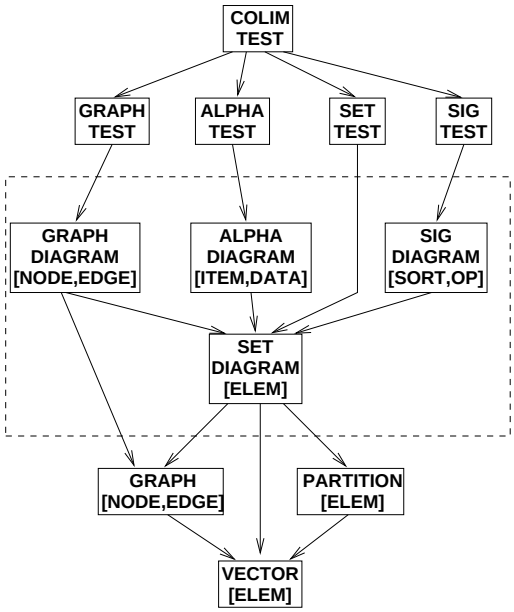
The main criteria which guided the design of the colimit library in addition to basic requirements like correctness and expandability were:

- The library should support different applications in the context of scheme, graph or petri net transformations and template mechanisms for parameterised programs or specifications.
- The interface to the client applications should meet its needs and should be well documented and easy to use.
- Language independence. We don't want to force the user of the colimit library to choose a specific programming language. Suitability of a programming language is related to several application dependent requirements like efficiency, portability, availability of programming environments, specific libraries and programming skills. The other design criteria restrict the variety, but still we have a rich choice: C++, Eiffel, Java, Ada 9x, Sather, Smalltalk and Objective C, to name some of the possible implementation languages. For reasons we will discuss later, we concentrated our efforts on the first three of them, but designed the library such that a port to another object oriented language can easily be done. There is no essential language feature missing in one of our chosen implementation languages.
- Performance of the colimit computation. For our application performance is essential because we are faced with preconceptions against colimits as a theoretically founded, but slow technique. We proved linear complexity and will support this

result by practical benchmarks. We will see that colimit computations are competitive compared to alternative approaches for the implementation of structuring mechanisms and scheme transformations.

5.2 Structure of the Colimit Library

The following class graph illustrates the structure of the colimit library.



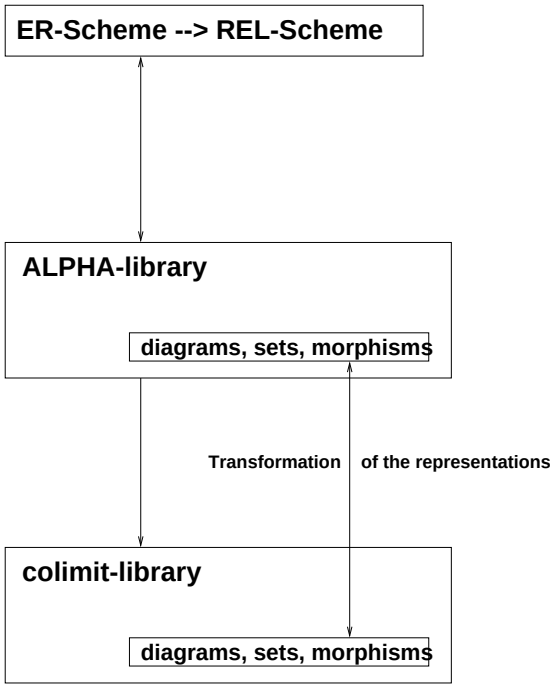
On top there is the class COLIM_TEST, which is used to test the correctness and efficiency of the colimit computations. It uses the test classes for graph, ALPHA, set and signature diagrams below. These classes also provide example code using the interface of the colimit library, which forms the next layer in the diagram above.

These interface classes implement the colimit computations for graph, ALPHA and signature diagrams. They are based on a class for set diagrams. All these classes are parameterised over the type of set elements they use. Since the colimit computation is independent from this type, there are no special requirements for it. Even no equality operation is needed because we identify the elements by their position in the data structure representing the coproduct.

Set diagrams themselves are based on a class for attributed graphs with diagram nodes and edges as attributes, and a class implementing the UNION-FIND set partition algorithm for the colimit computation. All these classes use a class implementing dynamic arrays called VECTOR[ELEM]. The restriction to a single base data structure has many advantages as will be explained in the following sections.

5.3 Interface and Embedding

Now we show, how the library can be embedded in an application for scheme transformations.



We take as example the transformation of a data base specification given as ER-diagram into a relational data base scheme. On top there is a class realizing this transformation. It is based on the ALPHA library [EC96] for general hierarchical algebraic graph structures. Colimit computations are the essential operations in the ALPHA library, they are implemented by the colimit library. The general principles illustrated by this example are:

- There should be an intermediate library layer providing the application specific representation of the diagrams and their transformation into the representation for the colimit library.
- The application specific library and the colimit library should use different data representations for the diagrams.
- The additional overhead imposed by the change of representation is outweighed by the advantages (simplicity, efficiency and portability) for the colimit library.

Before the colimit library was integrated with the ALPHA library it provided separate specialised routines for the different colimit constructions like pushouts, coproducts and coequalizers. These routines are very useful for the comparison of different possible realizations of a general colimit procedure.

Our approach structures the colimit procedure such that it relies on general set colimits computed by the UNION-FIND set partition algorithm. Alternatively it could recursively call specialised functions for coproducts and coequalizers (or pushouts and initial objects) following a different proof of cocompleteness [RB88]. Obviously for general colimit computations this approach is less efficient and the integration of partiality is more laborious.

But despite this fact there could be some doubts whether this is also true if our application mostly requires elementary colimit computations like pushouts and coproducts. A specialised pushout routine can be realized without changing the representation and we have more opportunities for optimisations.

Our experiences with the ALPHA library have shown that even under these circumstances the speed advantage of the specialised routines is negligible in practice.

5.4 Comparison of the Chosen Implementation Languages

5.4.1 C++

Currently C++ is very close to a standard among object oriented programming languages. It has the greatest variety of compilers and development environments and is considered the standard development language for OS/2 (IBM), Microsoft Windows and DOE from Sunsoft. Companies like IBM, Apple, Sun, Hewlett Packard and Microsoft produce most of their software with C++. In short: C++ is the most common, most supported and most portable object oriented programming language with the richest choice of available class libraries.

C++ provides a smooth transition path from error-prone C-like techniques to an object oriented programming style. It supports low-level programming in a way that leaves no room for a lower-level language except assembler. In other languages like Java and Eiffel sometimes you are forced by performance requirements to link your program with routines written in a special low-level language. This aggravates maintenance and produces interface problems. Progress in compiler technology will probably reduce this advantage for C++ in the near future.

The major risk not using C++ are:

- Poor integration into the target environment and with other tools or applications.
- Lack of portability.
- Lack of available class libraries.
- Lack of availability and quality of tools.
- Lack of runtime performance.

We choose Java and Eiffel as alternative languages, because they avoid mostly the first four of these risks. In addition we will show by performance measurements, that Eiffel has competitive runtime performance for our application. The runtime performance problems with Java will probably disappear when better compilers or specialised Java processors become available.

The main reason for choosing C++ as our first implementation language is its enormous flexibility and the general availability of the Standard Template Library (STL) [AS94]. The STL provides a higher level of abstraction by separating containers, iterators and algorithms, a concept called generic programming in [Jaz95]. Iterators provide access

to the containers, and algorithms are formulated using the iterators. This means, that we can easily exchange a container by another one for efficiency reasons. Because every container class has some advantages and drawbacks, the best choice is dependent on the concrete application. The STL enabled us to experiment with different representations of objects and morphisms for the colimit algorithm.

But as always there is a backside of the medal. Following [RK94], C++ is hard to learn and few programmers get really good at it. These are the words from the chairman of a major vendor of C++ compilers (Borland International). [ML94] states that the person-days per class value is higher for C++ than for other languages because *C++ is more cryptic and difficult to reuse, resulting in more reinvention versus reuse.* Behaviour of C++ programs can be subtle and hard-to-predict. C++ may be regarded as a significant barrier to learning and applying object oriented methodologies.

5.4.2 Java

Java, although like C++ an object oriented C-extension, avoids many of the problems with C++ mainly by the restriction on essential language features. There are no *goto* statements, header files, structs and unions, operator overloading, multiple inheritance and templates (generic data types). But the most important simplification is that Java doesn't use pointers and provides automatic memory management (garbage collection). Pointers and hand-coded memory management are one of the most bug-prone aspects of C++ programming. Despite its syntax, Java has more similarity with Eiffel than with C++. It is questionable whether it was a good idea to remove multiple inheritance and genericity (the major remaining differences to Eiffel).

Both Eiffel and Java are strongly object oriented in the sense that every data type is an object. In Eiffel this holds for all data types, in Java some built-ins like integers form an exception. This forces the programmer to use object oriented design methodologies and enables the definition of general features shared by all classes and objects. In the colimit library for instance it simplifies the implementation of the test output for diagrams and colimits.

The main advantages Java currently has both over C++ and Eiffel are:

- Platform independence supported by the Java Virtual Machine (JVM), an abstract machine interpreting Java byte code on many different machines and operating systems. The performance penalty for byte code interpretation is partly balanced out by new compilation techniques like *just in time* compilation. There are plans to build compilers into the JVM also for Eiffel, which is near at hand because of the strong similarities between both languages.

- General availability of powerful libraries (together with their source code). But since the language is relatively new, they are currently not very stable. At least once a year new versions with major extensions are published.

- Support of distributed applications and threads built into the language (keyword synchronised). For Eiffel new concepts for synchronisation are developed but not yet integrated in the language and the tools.

- Run time support from internet browsers, which recently became one of the most important applications.

5.4.3 Eiffel

Beside C++ and Java, which are mainstream languages, Eiffel currently hasn't achieved such a wide spread acceptance for several reasons. Most programmers today have experience in C programming, so they favour languages with a syntax similar to that of C. Additionally Eiffel doesn't come with a powerful public domain library of predefined classes as Java. But C++ and Java inherit from C together with the syntax several language design flaws. In C++ it is even possible to code in a low level C-style, without using any object oriented methodology.

Eiffel is a simple, easy to learn purely object oriented language, with built in safety features like assertions. Its main advantages related to C++ and Java are

- A powerful constraint genericity mechanism which combines the safety of static type checking with the flexibility of dynamic binding.
- A true and safe multiple inheritance mechanism. Often it is claimed that multiple inheritance is tricky (specially with C++), but Eiffel has a safe mechanism to handle the issue. Even if Java's interfaces are a nice feature to support multiple specification inheritance, there's no way to inherit more than one implementation.
- The "Design By Contract" methodology (preconditions, postconditions and class invariants) is directly supported by the language. On the one hand this leads to a great improvement of software quality. On the other hand, applied to class libraries, it enhances productivity by shortening and simplifying the debugging phase. Java's boundary checked arrays aim in the same direction.
- The covariance principle, which is superior to Java's (and C++'s) invariance principle. Features (attributes and routines) in a child class can be redefined with children types. Covariance is supported by the "like" keyword, a simple and powerful way to make routines covariant.

As mentioned in the section before, in addition Eiffel shares most of the advantages related to C++ with Java.

Summarising we can say, that there are many arguments for and against all possible implementation languages for the colimit library so that choosing only one would be too restrictive. The given arguments motivate why we have decided to support C++, Java and Eiffel.

5.5 Component Programming

A few years ago software engineering was clearly dominated by object oriented design and programming. Recently another more data type oriented programming paradigm, called generic programming [DRM89] gained importance. Component programming [Jaz95]

combines paradigms from object oriented, generic and functional programming. It depends on recent advances in programming language technology, structuring of algorithms and data structures and programming methodology. The language C++ plays an important role, because of its enormous flexibility in supporting different design paradigms. The C++ Standard Template Library (STL) [AS94] illustrates the promise of component programming.

There is a strong relation between genericity and parameterised data types. Stepanov states in [DRM89]: "The nature of the problem of verifying generic algorithms should be attractive to researchers in computer science and mathematics". The theory of structured algebraic specifications semantically described by colimit constructions is a step in this direction. But on the other side, the research on component programming should influence the tool design for structured specification languages.

We will motivate the importance of generic programming by a discussion on the efficiency of colimit computations. The STL provides a new higher level of abstraction by separating containers, iterators and algorithms. Iterators provide access to the containers, and algorithms are formulated using the iterators. This means, that we can easily exchange a container by another one for efficiency reasons. Because every container class has some advantages and drawbacks, the best choice is dependent on the concrete application. For instance, hash tables and search trees both can implement associative containers supporting the same iterator class. Hash tables provide constant average access time but may be linear in the worst case while search trees have logarithmic worst case access and the additional advantage, that the elements are sorted. See [Mus95] for a comparative discussion. Using the STL switching from hash tables to search trees means to change one line of code (the declaration of the container type), all algorithms work as before, only the efficiency has changed.

Therefore experimenting with different representations of objects and morphisms for the colimit algorithm was easy. Linear complexity is a theoretical result which doesn't tell us how to reduce the constant factor. We tried hash tables, search trees, dynamic arrays and linked lists both for the set objects and the morphisms. From our first try to the final implementation performance increased by a factor of 8-10, both in time and space.

Beside using the STL alternatively we made experiments with the LEDA class library [KM95], which directly supports graphs and set partitions. But choosing a library with such a rich functionality conflicts with some of our main design criteria, namely independence from the implementation language and efficiency.

5.6 Mapping of the Mathematical Concepts to Data Types

How should the mathematical concepts described in the theoretical part of this thesis be mapped to concrete data types? Coincide the requirements for their design from the applications with that for the colimit computation? Can we use the same representation or is it necessary to introduce a transformation? We will see, that optimal performance of the crucial operations of the colimit computation conflicts with performance requirements of operations needed by their applications. Hence we propose not to share the representa-

tions but to perform an extra transformation step to a representation optimised specifically for the colimit algorithm. This has the additional advantage, that this representation is completely independent from the application. If additional requirements occur and the application data types are extended only the transformation has to be adapted.

The colimit library handles diagrams of categories of structured objects like signatures, graphs, graph structures, terms, etc. . In the chapters 3 and 4 we have theoretically shown, how these diagrams can be represented in a modular manner by simple set diagrams. So we mainly have to find an optimal data type representation for set objects and morphisms, and for the relations derived from their structure, for example the source and target mappings for graphs or the arity function for signatures.

For the colimit computation the most important set operations are application and composition of morphisms, coproduct, factorisation and computation of the universal morphism. To achieve linear complexity of the colimit algorithm we need linear complexity of all these operations which requires constant time complexity for the following two container class operations:

- Random access to an element. This enables the application of a morphism to a set element in constant time and is needed for the factorisation , i.e. , the applicability of the UNION-FIND set partition algorithm.

- Insertion of an element at some position into the container. This represents the coproduct of an arbitrary set with a set containing one element. If it has constant time complexity the coproduct of two arbitrary sets has linear complexity.

In terms of category theory: These two requirements enable the computation of dual coequalizers and coproducts in linear time. Obviously these are necessary criteria for the linear complexity of the algorithm for arbitrary colimits. Theorems 3.1.6, 3.2.4, 4.2.7 and 4.2.9 show, that they are also sufficient. [21]

Less relevant for the colimit computation are the deletion of an element, the insertion at a certain position inside the container and the access of an element corresponding to a key (for instance a string identifier). At least the key access is of crucial importance at the application level which motivates our decision to use different diagram representations.

Our requirements exclude for instance linked lists as container class implementing set morphisms. Linked lists provide deletion and insertion at an arbitrary position in constant time (which is not needed), but random access has linear complexity. This means, that application of a morphism to a set element wouldn't be possible in constant time.

The identification of elements is needed only for the coproduct and the colimit, but not for the input sets. Suppose two elements from two different sets are equal (have the same key). If we would represent the distinction of these elements in the coproduct via some tag, their comparison would require to compare both the key and the tag. Obviously there are more efficient solutions. The container object representing the coproduct contains references to all elements.

[21] In the practical part of this thesis we forget about the (minimal) non-linearity of the complexity of the UNION-FIND algorithm because it is irrelevant for practical applications. It is even not measurable.

- We could use the memory locations of these references as key for the coproduct. Since today machines use a flat memory model, this solution is feasible in general. But there might occur problems when the memory location of an element changes, caused for instance by garbage collection. Another argument against this solution is that some programming languages like Java and Eiffel don't support pointers. There memory locations are not comparable.

- We could decide to use (dynamic) arrays as container class. Then we simply could use the array index as key, enabling very efficient coproduct element comparisons in constant time.

After analysis and evaluation of many experiments with different representations we finally have chosen the second solution with dynamic arrays as the basis of our data structures. Beside meeting all our requirements this representation has many advantages:

- The coproduct of a set diagram can simply be computed as array concatenation in linear time.

- Index based set mappings can easily be lifted to the coproduct by index computations in linear time.

- Dynamic arrays are supported by the standard libraries of many programming languages, especially by C++, Java and Eiffel.

- Garbage collection is much easier, because there is only one continuous memory area occupied by the container.

- Caching and paging reduce the performance of containers which are widely distributed over the physical memory area. Performance of secondary and tertiary memory currently can not keep up with the breakneck speed of processor development, which makes this problem even more relevant.

Memory management is crucial for the performance of an implementation of dynamic arrays. Extension of a dynamic array by a single element can only be done in constant time (on average) if the allocated memory is increased exponentially. That means, that after the allocated memory is exhausted, its contents are copied in a new area which is by a certain factor larger than the old one. Most C++ STL implementations, the Java standard library and most Eiffel libraries support this behaviour. If not, the dynamic array class from the library has to be adapted (using inheritance), to avoid a significant loss in performance. This was done for some Eiffel ports of the library. In one case (SmallEiffel) the library itself was modified by its author (D. Colnet, CRIN-INRIA).

The price for the enhanced efficiency is that some memory is wasted. For instance, if the chosen factor is 2, on average only 75% of the memory is occupied by (references to) array elements. But compared with alternative container data types this overhead seems acceptable. Linked lists for instance waste 50% of their memory for the pointers representing the links between elements. Trees and hash tables waste even more.

5.6.1 Sets and Set Morphisms

As motivated in the last section both sets and set morphisms are realized as dynamic arrays. Set elements are distinguished by their array indices. This representation is optimal for the needs of the colimit computation. Exactly the operations needed for the colimit construction, namely element comparison, application and composition of morphisms, coproduct computation, computation of the universal morphism from the coproduct (and colimit) and factorisation are supported with the highest possible efficiency. It is not suitable for other operations like access of a set element corresponding to some key (identifier), set union, difference and intersection, insertion and deletion of an element and set equality. Here for instance a search tree representation as used for the STL *set* container class is more appropriate. Hence we use such a representation for the applications of the colimit algorithm and add to the algorithm the transformation between the two representations. This approach is realized for instance for the implementation of the ALPHA library for scheme transformations [EC96].

The application needs access to the representation of the colimit. Therefore the transformation has to be performed in both directions. We need to establish a bidirectional mapping between the representations. We have to map keys (identifiers) on the corresponding array indices. This mapping can be realized by a hash table or a search tree. Both hash tables and search trees don't have a guaranteed constant access time, which means that we loose linear complexity of the overall colimit computation including the transformations. But the complexity depends only on the size of the involved set objects, not on their number. Under normal conditions hash tables provide constant average access time, so it is not surprising, that this theoretical problem doesn't occur in practical benchmarks.

We abbreviate the contents of a dynamic array $[0 \mapsto a, 1 \mapsto b, \ldots]$ with indices starting at 0 by $[a, b, \ldots]$. The concrete representations are:

- Sets are represented as dynamic arrays of object references.
- Morphisms are represented as dynamic arrays of integers, where the integers are indices into the target set array indicating the target elements corresponding to a source element. For instance, let $A = \{a, b, d\}, B = \{a, b, c\}$ and the partial morphism $f : A \to B = \{a \mapsto a, b \mapsto c, d \mapsto -\}$, then A is represented by $[a, b, d]$, B by $[a, b, c]$ and f by $[0, 2, -1]$ (the first index is 0, an undefined position is characterised by -1).

5.6.2 Attributed Graphs

Although we provide an implementation of graph diagrams, the most important role of graphs is their usage as basic data structure for the implementation of set diagrams. They are realized as follows:

- A graph is represented by four arrays, a node attribute array, an edge attribute array and two integer arrays representing the source and target mappings. The integers

in these arrays are indices into the node attribute array indicating the source rsp. target node of an edge. For instance, given the graph G with nodes $\{a, b, c\}$ and edges $\{e : a \to b, \quad f : a \to c\}$, its nodes are represented by $[a, b, c]$, its edges by $[e, f]$, its source mapping by $[0, 0]$, its target mapping by $[1, 2]$ (0 is the index of the first node a).

- Graph morphisms are given as tuples of set morphism arrays, one for the nodes and one for the edges.

This graph representation not only shares the general advantages of dynamic arrays mentioned above, it is also much closer to their mathematical definition (see 2.2.1) as the usual realization as a pointer structure. Its main disadvantage is that the deletion of a node or edge is relatively expensive, i.e. needs linear instead of constant time, and that it is not appropriate for graph traversal algorithms. But it serves our purposes in an optimal way in providing optimal performance for the colimit computation.

The separation of the node and edge part of a graph morphism is essential for the "comma-categorical" modular representation of graph diagrams constructed from two separate set diagrams.

5.6.3 Set Diagrams

Set diagrams are implemented as attributed graphs with integer tuples and dynamic integer arrays as node rsp. edge attributes where:

- The node attributes contain object delimiting indices into the coproduct (which itself is a set object, hence represented as an array).
- The edge attributes store representations of the morphisms.

The set coproduct is the disjoint union of all set objects in the diagram. It is constructed by array concatenation. The coproduct is simply the concatenation of all arrays representing set objects in the whole diagram. There is no need to store these set arrays, instead their delimiting indices into the coproduct array serve as representation of set objects. These indices are necessary to transform the index of the element of a set object into its related index into the coproduct rsp. colimit array.

5.6.4 Attributed Graph Diagrams

A graph diagram is represented by a tuple of set diagrams and a tuple of integer vectors representing the source and target mappings of the coproduct graph. This coincides exactly with the modular structure of graph diagrams introduced in chapter 3.

5.6.5 Signatures

A signature is represented as a tuple of arrays containing sorts and operations and an array of integer arrays representing the arity. The integers in these arity arrays are indices into

the sort array indicating the argument sorts and the result sort (in this order). For instance, the sorts are given as SORTS $= \{a, b, c\}$, the operations are
OPS $= \{f : a, b \to c;\ g : c \to a;\ h :\to b\}$ then SORTS is represented by $[a, b, c]$, OPS by $[f, g, h]$ and the arity by $[[0, 1, 2],\ [2, 0],\ [1]]$.
Signature morphisms are given as tuples of set morphism arrays, one for the sorts and one for the operations. A signature diagram is represented by a tuple of set diagrams and an array storing the coproduct arity.

The attentive reader may have noticed the strong similarities between graph and signature diagram representations. This is caused by the "comma-categorical" modular representation of these diagrams. The theory from chapter 2 is realized as close as possible leading to an uniform treatment of these categories. But we go not as far as [RB88] where a general algorithm for comma categories is given, because efficiency is one of our main design criteria.

5.6.6 Specifications

In the current version of the colimit library specifications are not yet directly supported. But the class for signatures can quite easily be extended via inheritance to compute the axioms of the colimit object, if we take specification morphisms as pure syntactical transformations. If we need in addition a semantical transformation of the axioms, theorem proving techniques, as for instance realized in the SPECWARE system [SJ95], are required. Despite being out of scope of this thesis a semantical treatment of specification morphisms is very useful in practical applications. But its basis is the syntactical colimit construction on signatures as provided by our library.

5.6.7 ALPHA Algebras

An ALPHA algebra is represented as an array of items, an array of integer arrays representing the references and an array of attribute arrays for the attributes. The integers in the reference arrays are indices into the item array indicating the referenced items. For instance, given the ALPHA algebra

```
ALGEBRA A
   a1()
   a2(1:a1 2:v1)
   v1(1:a1 ATTR 1:val1 2:val2)
END
```

the items are represented by $[a1, a2, v1]$, the references by $[[\,],\ [0, 2],\ [0]]$ (0 is the index of the first item $a1$) and the attributes by $[[\,],\ [\,],\ [val1, val2]]$. ALPHA algebra morphisms are given as item set morphism arrays.

5.6.8 Graph Structures

General graph structure diagrams are not directly supported by the colimit library. Instead they have to be transformed into equivalent (typed) ALPHA algebra diagrams. Dif-

ferent item sorts are distinguished by typing morphisms. The graph structure obtained by re-transforming the typed colimit ALPHA algebra is the colimit of the original graph structure diagram. The typing is obtained as a universal morphism from the colimit of the untyped ALPHA algebra diagram into the type algebra.

5.7 Colimit Computation for Set Diagrams

We will present a slightly simplified version of some of the function definitions implementing the incremental colimit computation on sets. It illustrates the compactness and readability of the code. Note that the following code implements both coproduct computation and the factorisation. We use the Eiffel version of the code, because Eiffel has the most readable syntax of our chosen implementation languages. But there is almost a line to line correspondence to the Java and C++ versions since we designed the classes in a portable manner.

The union operations of the UNION-FIND algorithm can be performed directly during the insertion of morphisms into the diagram. Thus the algorithm becomes incremental, in the sense that we can use the computed colimit and later extend the diagram without recomputation. Unfortunately, deletion of a morphism requires recomputation of the colimit due to the internal structure of the UNION-FIND algorithm.

```
insert_object(set: VECTOR[ELEM]; name: STRING): INTEGER is
   local
      set_lower: INTEGER
      element: VECTOR_ITERATOR[ELEM]
      coprod_obj: COPROD_OBJECT
   do
      set_lower := f_coproduct.upper+1
      from !!element.first(set) until not element.avail
      loop
         f_coproduct.push_back(element.item)
         f_colimit.push_back(f_coproduct.upper)
         f_partition.make_block
         element.next
      end -- loop
      f_name.push_back(name)
      !!coprod_obj.make(set_lower, f_coproduct.upper)
      f_colimit_valid := FALSE
      Result := insert_node(coprod_obj)
end -- insert_object
```

insert_object inserts a $\mathcal{SET}$-object into the diagram, thereby incrementally computing the coproduct. The coproduct is represented as a dynamic array (class VECTOR) of set elements (parameter class ELEM). Array elements are appended to an existing array with *push_back*, which automatically increases the size of the array by a certain factor, if necessary. Iterators are created and set to the first element with *first*, stepped through with *next*, and tested for more available elements with *avail*. The element at the position denoted by the iterator is given by *item*. *upper* and *lower* are the delimiting array indices,

infix operator @ delivers the element at a given index. The *f_partition.make_block* operation creates a new equivalence class for the UNION-FIND set partition algorithm. It is called for all elements in the input set.

The set diagram is represented as an attributed graph with $\mathcal{SET}$-objects as node attributes and $\mathcal{SET}$-morphisms as edge attributes. Only the object delimiting indices into the coproduct vector, not the objects itself are stored in the diagram. The most important routines are those for inserting objects and morphisms and for computing and accessing the colimit.

```
insert_morphism(morphism: VECTOR[INTEGER]; v, w: INTEGER): INTEGER is
   require
      morphism_size: set_at_node(v).upper - set_at_node(v).lower + 1
                                                    =  morphism.count
      morphism_lower: morphism.lower = 0
   local
      source_lower, target_lower,
      source_pos, target_pos, partition : INTEGER
      element: VECTOR_ITERATOR[INTEGER]
   do
      source_lower := set_at_node(v).lower
      target_lower := set_at_node(w).lower
      from !!element.first(morphism) until not element.avail
      loop
         source_pos := source_lower + element.index
         target_pos := target_lower + element.item
         partition := f_partition.union(source_pos, target_pos)
         element.next
      end -- loop
      f_colimit_valid := FALSE
      Result := insert_edge(morphism,v,w)
end -- insert_morphism
```

insert_morphism inserts a $\mathcal{SET}$-morphism into the diagram, thereby incrementally performing the factorisation. For all source elements *element* the corresponding target index *target_pos* into the coproduct vector is computed and the *f_partition.union* operation is called with all these pairs. *f_partition.union* is the union operation from the UNION-FIND set partition algorithm. Before *insert_morphism* can be applied, the dynamic array representation of the morphisms has to be computed, e.g by using an associative container. The code presented here is a slightly simplified version which doesn't include the treatment of partial mappings.

```
get_colimit_index_at_node(element, node: INTEGER): INTEGER is
   do
      Result := get_colimit_index(element + set_at_node(node).lower)
end -- get_colimit_element_at_node
```

```
get_colimit_index (element: INTEGER): INTEGER is
   do
      if f_colimit_valid then
         Result := f_colimit @ element
      else
         Result := f_colimit @ (f_partition.find(element))
      end -- if
end -- get_colimit_index
```

get_colimit_index_at_node returns the index of the colimit element representing a given *element* in a $\mathcal{SET}$-object at a diagram *node*. Note that the input *element* is specified by its position in the $\mathcal{SET}$-object. Computation of this position can be performed using an associative container. First the index of the input *element* into the coproduct vector is computed. Then the index of the corresponding colimit element is obtained from the find operation from the UNION-FIND set partition algorithm (*f_partition.find*). The array *f_colimit* is used for two purposes:

- To select a specific element representing the colimit equivalence class, which may be different from the result of the find operation.
- To reduce the time needed for the function call by storing the results of the find operation precomputed for all coproduct elements.

The colimit element may be obtained from its index by accessing the *f_coproduct* array. The whole computation is done in constant time because of the (almost) constant complexity of the find operation.

```
compute_colimit is
   local
      colim_index: INTEGER
      element: VECTOR_ITERATOR[ELEM]
      index: VECTOR_ITERATOR[INTEGER]
   do
      from !!element.first(f_coproduct)
      until not element.avail
      loop
         colim_index := f_colimit @ (f_partition.find(element.index))
         f_colimit.put(colim_index,element.index)
         element.next
      end -- loop
      f_colimit_valid := TRUE
end -- compute_colimit
```

compute_colimit calls the find operation for all coproduct elements and stores the result in the array *f_colimit*. It serves as an optimisation and is called automatically from interface routines. *compute_colimit* itself is not part of the interface.

5.8 Colimit Computation for Signatures and ALPHA algebras

The colimit computation for signatures is implemented exactly following the cocompleteness proof for comma categories. Signature diagrams are internally represented as tuples of set diagrams together with an array storing the coproduct arity. That means, that the arity is not stored separately for all objects but directly for the coproduct. Since we maintain all object delimiting indices into the coproduct array, we can easily retrieve the arity for any concrete object from the coproduct arity. As for sets, the most important routines are those for inserting objects and morphisms and for computing the colimit.

```
insert_object(sorts: VECTOR[SORT]
              ops: VECTOR[OP]
              arity: VECTOR[VECTOR[INTEGER]]
              name: STRING): INTEGER is
   require
      ops.count = arity.count
   local
      lower, sort_node, op_node: INTEGER
      arity_sorts: VECTOR_ITERATOR[VECTOR[INTEGER]]
      arity_sort: VECTOR_ITERATOR[INTEGER]
      cop_sorts:  VECTOR[INTEGER]
   do
      lower := f_sort_diagram.coproduct_upper + 1
      sort_node := f_sort_diagram.insert_object(sorts,name)
      op_node := f_op_diagram.insert_object(ops,name)
      from !!arity_sorts.first(arity) until not arity_sorts.avail
      loop
         !!cop_sorts.make(arity_sorts.item.lower,arity_sorts.item.upper)
         from !!arity_sort.first(arity_sorts.item)
         until not arity_sort.avail
         loop
            cop_sorts.put(arity_sort.item + lower, arity_sort.index)
            arity_sort.next
         end -- loop
         f_coprod_arity.push_back(cop_sorts)
         arity_sorts.next
      end -- loop
      f_colimit_valid := FALSE
      Result := sort_node
end -- insert_object
```

insert_object inserts a signature into the diagram. First the sorts and operations are inserted in different set diagrams, then the coproduct arity is computed. The sort lists assigned to each operation are stored as index arrays where the indices denote the position in the sort object. Since the sorts of the inserted signature get new positions in the coproduct, some index computations are performed to compute the coproduct arity.

```
insert_morphism(sort_morphism, op_morphism: VECTOR[INTEGER]
                v, w: INTEGER): INTEGER is
   local
      sort_edge, op_edge: INTEGER
   do
      sort_edge := f_sort_diagram.insert_morphism(sort_morphism,v,w)
      op_edge := f_op_diagram.insert_morphism(op_morphism,v,w)
      f_colimit_valid := FALSE
      Result := sort_edge
end -- insert_morphism
```

insert_morphism inserts a signature morphism into the diagram by calling the corresponding function for set diagrams. The signature diagram representation is chosen according to the comma categorical structure of the category, which makes this function almost trivial. The incremental colimit computation for sorts and operations is inherited from the *insert_mophism* operation for sets.

```
compute_colimit is
   local
      colimit_sorts, colimit_ops, colim_arity, arity_sorts : VECTOR[INTEGER]
      op, sort: VECTOR_ITERATOR[INTEGER]
      colim_sort: INTEGER
   do
      colimit_sorts := f_sort_diagram.get_colimit_indices
      colimit_ops := f_op_diagram.get_colimit_indices
      !!f_colimit_arity.make(colimit_ops.lower, colimit_ops.upper)
      from !!op.first(colimit_ops)
      until not op.avail
      loop
         arity_sorts := f_coprod_arity @ op.item
         !!colim_arity.make(arity_sorts.lower,arity_sorts.upper)
         from !!sort.first(arity_sorts)
         until not sort.avail
         loop
            colim_sort := f_sort_diagram.get_colimit_pos(sort.item)
            colim_arity.put(colim_sort, sort.index)
            sort.next
         end -- loop
         f_colimit_arity.put(colim_arity,op.index)
         op.next
      end -- loop
      f_colimit_valid := TRUE
end -- compute_colimit
```

compute_colimit performs the colimit computation calling the corresponding function for set diagrams. That computes separately the colimits for the sorts and the operations. Finally some index conversions are performed to compute the arity of the colimit signature. It is called automatically from interface routines and is itself not part of the interface. *get_colimit_pos* delivers the position of the colimit representation of an element in the colimit object (which is different from its position in the coproduct given by *get_colimit_index* since the colimit contains less elements.

The corresponding functions for ALPHA algebras are very similar so we refer to the documentation of the code itself. Different from signatures mainly is the treatment of partial morphisms. As we know from the theory we have to apply a colimit preserving totalisation functor to the resulting partial ALPHA algebra. In chapter 3 we derived an abstract description of this totalisation algorithm, here we present its concrete code:

```
totalize is
   do
      compute_dependent
      delete_dependent(f_dependent @ bottom)
      f_total_valid := TRUE
end -- totalize

compute_dependent is
   local
      item, item_ref: VECTOR_ITERATOR[INTEGER]
      dependent, colimit_items: VECTOR[INTEGER]
      colimit_item_ref: INTEGER
   do
      if f_dependent = Void or not f_colimit_valid then
         colimit_items := f_diagram.get_colimit_indices
         !!f_dependent.make(f_coprod_refs.lower, f_coprod_refs.upper)
         !!dependent.make_empty
         f_dependent.put(dependent,bottom)
         from !!item.first(colimit_items) until not item.avail
         loop
            !!dependent.make_empty
            f_dependent.put(dependent,item.item)
            item.next
         end -- loop
         from !!item.first(colimit_items) until not item.avail
         loop
            from !!item_ref.first(f_coprod_refs @ item.item)
            until not item_ref.avail
            loop
               colimit_item_ref :=
                  f_diagram.get_colimit_index(item_ref.item)
               dependent := f_dependent @ colimit_item_ref
               dependent.push_back(item.item)
               item_ref.next
            end -- loop
            item.next
         end -- loop
      end -- if
end -- compute_dependent
```

```
delete_dependent(deleted: VECTOR[INTEGER]) is
   local
      item: VECTOR_ITERATOR[INTEGER]
   do
      from !!item.first(deleted) until not item.avail
      loop
         if f_diagram.get_colimit_index(item.item) /= bottom then
            f_diagram.delete_element(item.item)
            delete_dependent(f_dependent @ item.item)
         end -- if
         item.next
      end -- loop
end -- delete_dependent
```

totalize is the main function of this algorithm calling *compute_dependent* and then *delete_dependent*. *compute_dependent* computes recursively a dependency graph. For all items in the partial colimit ALPHA algebra all items pointing via references (directly or indirectly) on them are stored in an array. *delete_dependent* then only has to delete all items dependent on the bottom element. Then there exists not longer any item referencing the bottom element which means that the resulting ALPHA algebra is total.

5.9 Language Specific Implementation Details

Although we designed the ports to our three implementation languages as similar as possible, there are still some language specific implementation details.

5.9.1 The Implementation in C++

C++ was the first implementation language of the colimit library. After the ports to Eiffel and Java were finished we redesigned the C++ version to make the three ports as similar as possible. As in Eiffel we augmented all classes with an *out* function converting the object into a printable string for test outputs. We avoided pointer arithmetic (in Eiffel and Java pointers are not available) or concealed them in iterator classes from the Standard Template Library. We implemented both manual memory management and the integration with an automatic garbage collector to compare their performance. The naming conventions used are somewhat unconventional for C++ code to obtain consistency with the Eiffel port. We denote class methods (features) and local variables with lower case and class names with upper case letters. Data members are preceded by *f_* and they are only visible to the correspondent iterator classes. We tried three versions of the STL on three platforms:

- On Microsoft Windows NT/95 we used the original version from Hewlett Packard together with the IBM Visual Age C++ 3.0 compiler.
- On Linux 2.0 and Solaris 2.3 we tried the GNU version of the STL from libg++2.7.1 and the commercial version from Object Space.

These STL versions differ with respect to their performance and their debugging support (which is better for the Object Space version). The Object Space version needs less memory but is slightly slower than the other versions.

5.9.2 The Implementation in Eiffel

In our comparison of object oriented programming languages above we have shown many advantages of Eiffel over C++. The most important ones were the constraint genericity mechanism, the "Design By Contract" methodology, the safe multiple inheritance mechanism (with possible renamings), the covariance principle, the lack of pointers together with automatic garbage collection and last but not least the much clearer syntax which is very easy to learn. Eiffel can be used both as specification and as programming language which enables a smoother transition from the design specification to the final code.

From our experiences with the first available Eiffel compilers some years ago we expected that these features don't come for free. The Eiffel port of the colimit library at first was started as an experiment to check how much of the efficiency is lost as compensation. As we will see from the benchmark results in the next section, the performance is more dependent on the specific compiler, basic library and operating system than on the decision between Eiffel and C++. This surprisingly positive result motivated us to redesign the colimit library to minimise the differences between the versions for the different languages. From the advantages of Eiffel one would expect a much faster development with a shorter and better supported debugging cycle. Our experiences even surpassed our expectations.

We tested most of the available Eiffel compilers, compensated minor library incompatibilities and finally compared the performance of the generated code.

5.9.3 The Implementation in Java

Since Java shares many advantages with Eiffel and recently becomes more and more important we decided to develop a Java port of the colimit library. It is more or less a pure syntactic transformation of the Eiffel version, since we didn't make heavy usage of the more advanced features of Eiffel like multiple inheritance. At that time the available development environments were quite rudimentary, but since then the situation has improved a lot. The main problem with Java was the missing support of generics (templates). We had to compensate it by adding dynamic type casts to ensure correct typing. Static type checking is hereby replaced by dynamic type checking which independently from the compiler technology reduces the maximal achievable performance. In addition the readability is decreased as the code contains less static type information. We tried Java byte code interpretation, *just in time* compilation and Java to C compilation and compared the efficiency of the generated code. In all cases it is significantly inferior to the C++ and most of the Eiffel results, which is partly caused by the early development stage of the Java compiler technology.

5.10 Benchmark Results

First benchmarks have shown linear space and time complexity of the colimit computation for set, signature, graph and ALPHA algebra diagrams up to diagram sizes which do not cause swapping. Since several hundred MB RAM memory is affordable today, the algorithm can treat diagrams containing several millions of elements in a few seconds. Considering our theoretical complexity proofs this result is not very surprising. More interesting is the comparison of different compilers / programming languages. The differences between the compilers are more or less independent from the size and kind of the computed diagrams. Of course colimit computation cannot be regarded as a general benchmark, with our implementation it mainly checks the performance of dynamic array operations. Since dynamic arrays play such an important role, we optimised their library versions for some of the chosen compilers.

For our comparison we choose the computation of the colimit of a signature diagram containing about 20000 symbols. The computation is repeated 12 times in a loop to test the garbage collection. The measurements are made on a 200Mhz Pentium Linux PC with 192MB memory using the *time* command. We added the user and system times to obtain the results. Note that at least 128Mb RAM is needed to avoid swapping for the test without garbage collection. The test diagram contains several morphisms per node chosen such that the factorization is hard to compute. Although we made a port to the Microsoft Windows NT platform using IBM Visual C++ 3.0, we list only the results for Linux to exclude the influence of different operating systems for our comparison.

We compared the following compilers / programming languages:

- C++: GNU g++ 2.7.2. We tried both the GNU-STL from libg++ 2.7.1 and the commercial STL-implementation from ObjectSpace.
- SmallEiffel: A public domain Eiffel compiler from CRIN-INRIA obtainable via ftp from *ftp.loria.fr/pub/loria/genielog/SmallEiffel*. We used version -0.89 with compile options -boost, -no_split.
- Tower Eiffel: Version 2.0, all optimizations enabled.
- ISE Eiffel: We tested ISE ebench 3.3.7, both the interpreted (melted) and the compiled version (options finalize, precondition checks off, no precompiled libraries, inlining enabled, array optimizations on).
- SIG Eiffel: Because of library incompatibilities we ported only the parts of the colimit library necessary for this test.
- Java: We tested three different Java implementations, the Sun Javac 1.0.1 with bytecode interpreter, the kaffe-0.5p3 just in time compiler and the j2c Java to C compiler from Todo Software (ftp.webcity.co.jp/pub/andoh/Java/j2c-beta4.tar.gz). We compiled the Sun Javac compiler itself with j2c to obtain the compile time results.

All Eiffel implementations and the j2c Java to C compiler use GNU gcc 2.7.2 with option -O2 as back end. As garbage collectors we tested:

- manual: Manual memory management
- BDW: Boehm-Demers-Weiser (ftp.parc.xerox.com/pub/gc/gc4.10.tar.gz)
- ISE, SIG, Tower: Built-in GC from vendor
- Sun, kaffe: Built-in GC, but compiler uses Boehm-Demers-Weiser

We integrated the Boehm-Demers-Weiser garbage collector into the C++, the SmallEiffel and the Java to C compiler to be able to compare all languages / compilers in combination with automatic memory management.

All implementations use iterator classes to traverse the elements of a container. STL-iterators use C++ specific language features to achieve an efficiency which cannot be realized for Eiffel iterator classes. Hence we tested the Eiffel implementations also with a special version of the colimit library where the use of iterator classes is restricted to less critical code areas.

compiler	GC	iter-ators	run time in sec	comp. time in sec	mem. in MB	code size in kB
C++ gnu lib	manual	yes	4.9	12.0	5.2	32
C++ GNU libg++	BDW	yes	9.3	14.5	9.9	76
C++ Obj.Space	BDW	yes	8.6	17.0	6.6	50
SmallEiffel	BDW	yes	6.9	11.0	8.0	58
SmallEiffel	BDW	restr.	5.6	11.0	6.6	57
SmallEiffel	—	yes	7.3	10.8	71.2	31
SmallEiffel	—	restr.	5.8	10.8	60.0	30
Tower Eiffel	Tower	yes	12.7	59.0	9.6	132
Tower Eiffel	Tower	restr.	8.2	59.0	9.5	131
Tower Eiffel	—	yes	8.3	59.0	77.7	132
Tower Eiffel	—	restr.	6.9	59.0	66.5	131
ISE melt	ISE	yes	1404.0	15.0	8.5	–
ISE finalize	ISE	yes	234.0	165.0	6.8	192
ISE finalize	ISE	restr.	219.0	165.0	6.6	192
ISE finalize	—	yes	42.8	165.0	108.0	192
ISE finalize	—	restr.	31.0	165.0	95.0	192
SIG Eiffel	SIG	yes	424.0	12.0	11.2	74
SIG Eiffel	SIG	restr.	39.5	12.0	17.0	74
SIG Eiffel	—	yes	43.2	12.0	214.0	74
SIG Eiffel	—	restr.	36.0	12.0	192.0	74
Java (Sun)	Sun	yes	528.0	0.9	102.0	16
Java (kaffe)	kaffe	yes	4364.0	0.9	43.0	16
Java (j2c)	BDW	yes	53.4	38.0	18.5	243

From the table above we can derive a lot of interesting conclusions:

- Java interpretation results in the fastest compile time but in slow execution time. Only diagrams up to a size of several thousand elements can be computed in an acceptable time. Just in time compilation is even slower (for colimit computations), only Java to C compilation delivers acceptable results. Currently the Java compiler technology is in its early stages, large improvements may be expected in the future. Sun's Java processors will make the performance even more competable. But the missing parameterisation concept in Java restricts not only the readability of the code, the necessary run time type checking causes a principal performance penalty.
- Eiffel interpretation (ISE melt) is by factor three slower than Java interpretation, but garbage collection is more effective.
- C++ with manual memory management gives the best performance results (100 times faster than Java interpretation).
- With automatic garbage collection there is almost no difference between C++ and the best Eiffel implementations (SmallEiffel and Tower).
- For the colimit computation the advantages of automatic garbage collection outweigh its performance penalty. The Boehm-Demers-Weiser garbage collector is a powerful tool applicable to many languages providing them with automatic memory management.
- There are large differences in the performance of the different Eiffel compilers.
- Beside for SIG Eiffel the overhead for separate iterator classes is acceptable.
- Both the Eiffel and the C++ implementation can treat diagrams up to a size of millions of elements in reasonable time and space on a cheap standard PC.

6 Conclusion

After summarising the main results of the thesis we give an outlook over future work.

6.1 Summary

Category theory provides powerful means for the abstract description of the semantics of specification languages and graph transformations. The colimit construction is the fundamental concept on which the definition of language structuring operations resp. graph transformations can be based. As pointed out in [Cla93], the restriction to special colimits (e.g. pushouts) for the definition of the semantics of specification languages leaves important implementation decisions wrt. sharing and structural equivalence open. This leads to the definition of an additional set theoretic semantics which is complex and hard to understand [San84, Cla89]. This seems no longer be necessary, since we have shown by proof and implementation, that there are algorithms computing colimits of specifications with almost linear complexity.

We have pointed out, that the structuring of proofs of cocompleteness of categories by means of comma category constructions leads to a structure of colimit algorithms supporting both reusability and efficiency. In every application the algorithm relies on the colimit computation on $\mathcal{SET}$-diagrams. Here direct implementation of the usual constructive cocompleteness proof as done in [RB88] leads to an inefficient algorithm. We avoided this by providing an alternative proof (theorem 3.2.2) which exploits properties of the category $\mathcal{SET}$.

We extended the theory and its realization for graph structures with partial morphisms and outlined some of its applications for scheme transformations [CL95, EC96], an interactive graph editor and transformation system (AGG-system [LB93]), a tool for algebraic high level (petri) nets and the simulation of an abstract machine for the execution of functional logic programming languages [CW94]. For the implementation special simple graph structures called ALPHA algebras were used.

We presented the design of a library implementing colimits for sets, signatures and graph structures and its integration into a general library for ALPHA algebras. Major design decisions were motivated by their influence on the maintainability, portability and efficiency of the resulting code. Independence from specific implementation languages was one of our goals. We discussed example implementations in C++, Java and Eiffel and showed benchmark results comparing their performance. Recently the language independence of the library has proven its usefulness, since it was decided to change the programming language for the re-implementation of the AGG-system from C++ to Java.

Instead of adapting the tools to the theory, in this thesis we adapted the theory for the needs of the tool design. The cocompleteness proofs were (re-)formulated such that there is a strong correspondence to the implemented code without any compromises related to efficiency.

6.2 Outlook

All applications of the colimit library are currently ongoing developments. We still have to gather experiences to decide on the priorities and directions the work could be improved and extended. From the theoretical point of view, the following extensions may be interesting:

- We need formal correctness proofs of the typed ALPHA algebra representations outlined in this thesis for AGG-graphs, algebraic high level nets and jungles. It has to be shown, that the computed typed ALPHA algebra colimit represents the colimits in these categories.
- For the representation of entity relationship schemes the work from [CL95] has to be extended to cover the transformation into typed ALPHA algebras.
- Some applications demand tool support for other constructions from category theory, for instance limits and pushout complements. We need constructive proofs for them supporting the design of efficient algorithms.
- For graph transformation systems the colimit computation performs a derivation step, but doesn't support the search of matching occurrences of the transformation rules. It would be interesting to investigate how the search for a matching occurrence may be combined with the colimit computation to obtain optimal performance.

The implementation of the library could be enhanced by:

- New ports to alternative platforms and programming languages: Specially the Windows NT platform gains more importance in the future. There is already a port to IBMs VisualAge for C++, but other ports may be required in the future. In most cases only small adaptions to the provided base libraries are needed, since the dependencies to proprietary features are kept as small as possible.
- Direct support of the double pushout approach for graph transformations by providing an efficient implementation of the pushout complement.
- Implementation of algorithms searching matching occurrences of derivation rules.
- Support for changes in the data type part along ALPHA algebra morphisms. This would allow for more flexibility of tools related to algebraic development methods, for example for AHL-nets.

The colimit library could become the kernel of a more general library for tools supporting algebraic development methods [EGW97]. Future experience during the development of such tools will lead to the detection of other reusable components.

References

[AHS90] J. Adádamek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Series in Pure and Applied Mathematics. Wiley, New York, 1990.

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures*. Foundations of Computing Series. MIT Press, 1991.

[AS94] M. Lee A. Stepanov. The standard template library. Technical report, Hewlett-Packard Laboratories, September 20 1994. available via ftp.cs.rpi.edu as pub/stl/doc.ps.Z.

[AZ95] E. Astesiano and E. Zucca. D-oids: A model for dynamic data types. *Math. Struct. in Comp. Sci.*, 5(2):257–282, 1995.

[BG80] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In D. Bjøner, editor, *Abstract Software Specification, Proc. 1979 Copenhagen Winter School, LNCS 86*, pages 292–332. Springer, 1980.

[Boo94a] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[Boo94b] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2 edition, 1994.

[BRJ+97] G. Booch, J. Rumbough, I. Jacobsen, et al. Semantic od the uml 1.1. Technical report, Rational Rose, 1997. available from http://www.rational.com.

[Bur86] P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*, volume 32 of *Mathematical Research — Mathematische Forschung*. Akademie-Verlag, Berlin, 1986.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Series in Computer Science. Prentice Hall International, London, 1990.

[CEL+96] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *LNCS*, 1996. Accepted.

[CEW93] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development — The ACT Approach*, volume 1 of *AMAST Series in Computing*. World Scientific Publishing, 1993.

[CGW92] I. Claßen, M. Große–Rhode, and U. Wolter. Categorical concepts for parameterized partial specifications. Technical Report 92–46, Technische Universität Berlin, 1992.

[CL95] I. Claßen and M. Löwe. Scheme evolution in object-oriented models — a graph transformation approach. In *ICSE '95 – 17th Workshop on Formal Methods Application in Software Engineering Practice*, Seattle, Washington, 1995.

[Cla88] I. Claßen. Semantik der revidierten Version der algebraischen Spezifikationssprache ACT ONE. Technical Report 88/24, Technische Universität Berlin, 1988.

[Cla89] I. Claßen. Revised ACT ONE: Categorical constructions for an algebraic specification language. In *Categorical Methods in Computer Science, LNCS 393*, pages 124–141. Springer, 1989.

[Cla93] I. Claßen. *Compositionality of Application Oriented Structuring Mechanisms for Algebraic Specification Languages with Initial Semantics.* PhD thesis, Technische Universität Berlin, 1993.

[CLWW94] I. Claßen, M. Löwe, S. Waßerroth, and J. Wortmann. Static and dynamic semantics of entity-relationship models based on algebraic methods. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, Informatik aktuell, pages 2–9. Springer-Verlag, 1994.

[CMR+91] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, and M. Löwe. Graph grammars and logic programming. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *LNCS 532*, pages 221–237. Springer, 1991.

[CMR+96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars, Volume 1: Foundations*. World Scientific, 1996. To appear.

[CR93] A. Corradini and F. Rossi. On the power of context-free jungle rewriting for term rewriting systems and logic programming. In *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993.

[CW94] D. Wolz A. Corradini. Jungle rewriting: an abstract description of lazy narrowing. In *LNCS 776*. Springer, 1994. Dagstuhl Seminar 9301 on Graph Transformations in Computer Science.

[DRM89] A. Stepanov D. R. Musser. Generic programming. In *LNCS 358*, pages 13–25. Springer, 1989.

[EBCO91] H. Ehrig, M. Baldamus, F. Cornelius, and F. Orejas. Theory of algebraic module specification including behavioural semantics and constraints. In *Proc. AMAST '91, Iowa City*, 1991.

[EBO91] H. Ehrig, M. Baldamus, and F. Orejas. New concepts for amalgamation and extension in the framework of specification logics. Technical Report 91/05, Technische Universität Berlin, 1991.

[EC96] S. Erdmann and I. Claßen. ALPHA – a class library for a metamodel based on algebraic graph theory. In *Proc. Fifth International Conference on Algebraic Methodology and Software Technology*, LNCS. Springer, 1996.

[EG91] H. Ehrig and M. Große-Rhode. Structural theory of algebraic specifications in a specification logic – Part 1: Functorial parameterized specifications. Technical Report 91/23, Technische Universität Berlin, 1991.

[EGKP97] H. Ehrig, R. Geisler, M. Klar, and J. Padberg. Horizontal and Vertical structuring Techniques for Statecharts. In A. Mazurkiewicz and J. Winkowski, editors, *LNCS Vol. 1243, CONCUR'97: Concurrency Theory, 8^{th} International Conference, Warsaw, Poland*, pages 181 – 195. Springer Verlag, July 1997.

[EGP97] H. Ehrig, M. Gajewsky, and J. Padberg. Action Nets, Hierarchical State Spaces and Abstract Statecharts as High-Level Structures. Technical report, Technical University Berlin, 1997.

[EGW97] H. Ehrig, M. Gajewski, and U. Wolter. From abstract data types to algebraic development techniques: a shift of paradigms. Invited paper for the International Workshop on Algebraic Development Techniques WADT 97 in Tarquinia, 1997.

[EHK+96] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars, Volume 1: Foundations*. World Scientific, 1996. To appear.

[EJO93] H. Ehrig, R. M. Jimenez, and F. Orejas. Compositionality results for different types of parameterization and parameter passing in specification language. In *Proc. TAPSOFT '93, Paris, LNCS to appear*, 1993.

[EM45] S. Eilenberg and S. MacLane. General theory of natural equivalences. *Trans. Am. Math. Soc.*, 58:231–294, 1945.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1990.

[EO94] H. Ehrig and F. Orejas. Dynamic abstract data types: An informal proposal. *Bull. EATCS 53*, 1994.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180, 1973.

[EW86] H. Ehrig and H. Weber. Programming in the large with algebraic module specifications. 86:675–684, 1986. Invited paper, IFIP'86 World Congress.

[GB84] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop, LNCS 164*, pages 221–256. Carnegie–Mellon, Springer, 1984.

[Gen91] H.J. Genrich. Predicate/Transition Nets. In *High-Level Petri Nets: Theory and Application*, pages 3–43. Springer, 1991.

[GH91] M. Gogolla and U. Hohenstein. Towards a semantic view of an extended entity-relationship model. *ACM Transactions on Database Systems*, 16(3):369–416, 1991.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[GL81] H.J. Genrich and K. Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Gog89] J. A. Goguen. A categorical manifesto. Technical Monograph PRG–72, Oxford University Computing Laboratory, 1989.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HG95] D. Harel and E. Gery. Executable object modeling with Statecharts. Technical Report 94-20, Weizman Institute of Science, 1995. http://www.wisdom.weizmann.ac.il/Papers/CSreports/reps94/94-20.ps.Z.

[Him88] M. Himsolt. GraphED: An Interactive Graph Editor. In *Proc. STACS 89*, volume 349 of *Lecture Notes in Computer Science*, pages 532–433. Springer-Verlag, 1988.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[HP91] B. Hoffmann and D. Plump. Implementing term rewriting by jungle evaluation. In *Informatique théorique et Applications/ Theoretical Informatics and Applications*, pages 445–472, Berlin, 1991.

[Jay90] C. B. Jay. Extending properties to categories of partial maps. Technical Report LFCS 90–107, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

[Jaz95] M. Jazajery. Component programming - a fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, September 25-28 1995.

[JCJÖ93] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering — A Use Case Driven Approach.* Addison-Wesley, 1993. Revised Fourth Printing.

[Jen92] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, 1992.

[Jen95] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer, 1995.

[Joh87] T. Johnsson. Compiling lazy functional languages, part 1. Technical report, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[KM95] S. Näher K. Mehlhorn. Leda, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[Kor96] M. Korff. *Generalized Graph Structure Grammars with Applications to Concurrent Object-Oriented Systems.* PhD thesis, TU Berlin, 1996.

[LB93] M. Löwe and M. Beyer. AGG — An implementation of algebraic graph rewriting. In *Proc. 5th Intl. Conf. on Rewriting Techniques and Applications, RTA–93*, volume 690 of *Lecture Notes in Computer Science*, pages 451–456. Springer-Verlag, 1993.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, M. J. Plasmeijer, and M. C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd edition, 1987.

[Loc92] H. Lock. *The Implementation of Functional Logic Programming Languages.* PhD thesis, GMD Universität Karlsruhe, 1992.

[Löw93a] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181 – 224, 1993. Short version of Techn. Rep. 90/05, TU Berlin, Department of Computer Science, 1990.

[Löw93b] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993. Short version of Techn. Rep. 90/05, TU Berlin, Department of Computer Science, 1990.

[LZ75] B. H. Liskov and S. N. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE-1:7–19, 1975.

[ML94] J. Kidd M. Lorenz. *Object Oriented Software Metrics.* Prentice Hall International, N.J., 1994.

[Mus95] D. R. Musser. Rationale for adding hash tables to the c++ standard template library. Technical report, February, 20 1995. available from ftp.cs.rpi.edu as pub/stl/hashrationale.ps.

[Nag87] M. Nagl. A software development environment based on graph technology. In *LNCS 291*, pages 458–478. Springer, 1987.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories.* EATCS Monographs on Theoretical Comomputer Science. Springer, 1988.

[Pad91] J. Padberg. Kolimeskonstruktionen in Algebraischen Spezifikationssprachen. Studienarbeit, Technische Universität Berlin, 1991.

[Pad96] J. Padberg. *Abstract Petri Nets: A Uniform Approach and Rule-Based Refinement.* PhD thesis, Technical University Berlin, 1996.

[Par72] D. C. Parnas. A technique for software module specification with examples. *CACM*, 15(12):1053–1058, 1972.

[PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Math. Struct. in Comp. Science*, 5:217–256, 1995.

[RB88] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory.* International Series in Computer Science. Prentice Hall, 1988.

[RBP+91a] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–oriented Modeling and Design.* Prentice–Hall International Editions. Prentice–Hall, 1991.

[RBP+91b] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall International Editions. Prentice-Hall, 1991.

[Rei85] W. Reisig. *Petri nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer Verlag, 1985.

[Rei87] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, Oxford, 1987.

[RK94] J. Sutherland R.D. Kahn. Let's start under-promising and over-delivering on object technology. *Object Magazine*, 4, 1994.

[Roz87] G. Rozenberg. Behaviour of elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri nets 1986*, volume 254 of *LNCS*, pages 60–94. Springer Verlag Berlin, 1987.

[RR88] G. Rosolini and E. Robinson. Categories of partial maps. *Inf. and Comp.*, 79(2):95–130, 1988.

[San84] D. T. Sannella. A set-theoretic semantics for clear. *Acta Informatica*, 21:443–472, 1984.

[Sch91a] G. Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme.* PhD thesis, Universität Erlangen-Nürnberg, 1991.

[Sch91b] A. Schürr. Progress: A vhl-language based on graph grammars. In *LNCS532*. Springer, 1991.

[Sch94] A. Schürr. PROGRES, A Visual Language and Environment for PROgramming with Graph REwriting Systems. Technical Report AIB 94-11, RWTH Aachen, 1994.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[SJ95] Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. In *Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995. available from www.kestrel.edu (KES.U.94.5).

[SM96] Y. V. Srinivas and J. L. McDonald. The architecture of specware, a formal development system. Technical report, Kestrel Institute, 1996. available from www.kestrel.edu (KES.U.96.7).

[Smi93] D. R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming*, 15:571–606, 1993.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Comp.*, 4, 1982.

[Tae96a] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, FB13, Technical University Berlin, 1996.

[Tae96b] G. Taentzer. Towards synchronous and asynchronous graph transformations. Accepted for special issue of Fundamenta Informatica e, 1996.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms*, volume 44. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.

[TB94] G. Taentzer and M. Beyer. Amalgamated graph transformation systems and their use for specifying AGG – an algebraic graph grammar system. In *LNCS 776*. Springer, 1994.

[War83] D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, SRI International, 1983.

[WE86] H. Weber and H. Ehrig. Specification of modular systems. *IEEE Transactions on Software Engineering*, SE-12(7):784–798, 1986.

[Wol90] U. Wolter. An Algebraic Approach to Deduction in Equational Partial Horn Theories. *J. Inf. Process. EIK*, 27(2):85–128, 1990.

[Wol91] D. Wolz. Design of a compiler for lazy pattern driven narrowing. In *Proc. of the 7th International Workshop on the Specification of Abstract Data Types, LNCS 534*. Springer, 1991.

[Wol96] D. Wolz. Tool design for structuring mechanisms for algebraic specification languages with initial semantics. In *Recent Trends in Data Type Specifications, LNCS 1130*. Springer, 1996. Proc. of the 11th International Workshop on the Specification of Abstract Data Types, Oslo.