Total-Single-Pushout Derivations for Internal Graph Attribution

(Extended Abstract)

WOLFRAM KAHL

Institut für Softwaretechnologie, Fakultät für Informatik
Universität der Bundeswehr München, D-85577 Neubiberg
E-Mail: kahl@informatik.unibw-muenchen.de

Abstract: We present a simple, abstract approach that allows to integrate graph attribution into the graph structure itself. We use a declarative setting where the calculation of the attributions relies on single-pushout derivations in a category of *total* homomorphisms.

We discuss applications to term graph typing, including multi-level typing and type classes, and visualisation of compilation.

1. Introduction

Attribute grammars have been developed by Knuth for specifying and implementing the (static) semantic aspects of programming languages [10,11]. Since then, attribute grammars have grown into a recognised field of study with numerous applications; for one of many surveys see [13].

Attribute grammars are an extension of context-free grammars which consists of *semantic rules* added to the syntactic rules of the context-free grammar. An abstract view of this extension sees attribute grammars as defining *attributions* of *syntax trees*. Syntax trees can be considered as a special kind of graphs. An attribution of a syntax tree is then a (partial) function which expects a node of the graph and an attribute name as arguments and which may deliver as its result an attribute value from some attribute value set.

This view of attributions carries over to general graphs without any problems, and this has already been persued in a number of ways. However, all approaches in the literature essentially consider attributes as atomic items, usually from some algebra, that are attached to nodes essentially just like node labels. An overview over different graph attribution approaches is given in [15]. There, intrinsic and derived attributes are distinguished, where intrinsic attributes are amenable to being changed in the course of rewriting, and derived attributes depend on attribute values at neighbouring nodes.

Work on attributed graphs in the single-pushout approach, e.g. [12], all concentrates on modifying intrinsic attributes taken from an algebra.

Other approaches, like [16] and the project PAGG [4], attach directed attribute equations to structure replacement rules. Somewhat similar to that approach we have in [9] shown a way to consider well-typed term graphs as the language created of a certain kind of graph grammar where metavariables are stepwise instantiated with typing elements — corresponding to the attribution rules of this paper. This way to consider typing as an attribution will be further investigated in this paper in Sect. 6.

Since we do not consider rewriting (except for the purpose attribution) in this paper, we shall only concentrate on derived attributes (if seen as depending on the graph structure itself, too). We also want to be independent of the way the language of the graphs to be attributed is defined, so we do not want to attach directed attribute equations to graph rewriting rules. However, our approach still might be described as attaching attribute definitions to graph patterns.

Our motivation started from applications to term graphs, which are very close to term trees. There we noticed that attribute values, which usually can be represented as terms in some appropriate language, can, of course, be represented by term graphs, too. Therefore we started with an internal term graph attribution scheme, and in this paper we abstract away from the term graph setting and present a generalised graph attribution mechanism where we fully integrate the attributes into the graph structure.

At first it might seem more natural to let the graph representations of the attribute values live in a separate graph, but full integration on one hand permits simpler formalisation, and on the other hand allows attribution of attribute nodes without extending the formalism, which has interesting applications in advanced typing systems for term graphs.

The main novelty of our approach therefore is that we move the attributes *into* the graph structure, thus achieving two interesting effects:

- the internal structure of the attribute values can be fully exposed to the attribution mechanism, and
- attribute nodes themselves may carry attributes again.

After fixing some notation, we first define the way in which we abstractly add internal attribution to existing graph structure in Sect. 3. In Sect. 4 we then define attribution rules and their application via a single pushout with total homomorphisms. Finally we discuss examples from the fields of compilation and typing.

The graph attribution mechanism presented in this paper has been designed and (partially) implemented for the term-graph programming and program transformation system HOPS [5,8,17], which also has been used to produce all the drawings.

2. Notation

In our formalisation, we frequently use relational operations since this allows very clear and concise formalisations in the context of graphs, see e.g. [2,14].

The powerset of a set A is written $\mathbb{P}A$. The set of relations between two sets A and B is written $A \leftrightarrow B$ and is equal to the power set of the cartesian product: $(A \leftrightarrow B) := \mathbb{P}.(A \times B)$. The set of univalent relations or partial functions from A to B is written $A \nrightarrow B$, and that of total functions or mappings is written $A \rightarrow B$. Application of a function $f : A \nrightarrow B$ to an argument $x : A$ is written "$f.x$", has higher precedence than all other operators, and is only used if the argument is known to be in the domain of the function.

For two relations $R, S : A \leftrightarrow B$, their union is $R \cup S$, and inclusion is written $R \subseteq S$. The converse of R is the relation $R^{\smile} : B \leftrightarrow A$, with $(x, y) \in R^{\smile} \Leftrightarrow (y, x) \in R$. The reflexive transitive closure of a relation $R : A \leftrightarrow A$ is written R^*. For two relations $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$, their composition is $R;S : A \leftrightarrow C$ with $(x, y) \in (R;S) \Leftrightarrow \exists z \bullet (x, z) \in R \wedge (z, y) \in S$.

When a relation $R : A \leftrightarrow A$ is considered as a *graph*, a node $y : A$ is *reachable* from another node $x : A$ if and only if $(x, y) \in R^*$.

In categories, too, we use ";"as composition of homomorphisms and "." for application of functors.

3. Enriching Graphs with Internal Attributions

In this paper, we want to extend exisiting graph structures of the most diverse kinds with an internal attribution structure. We therefore first define the interface between those arbitrary graph structures and our attributions:

Definition 3.1: A category $\mathcal{S}$ of **superficial graph structures** is a category that has

- pairs $(\mathcal{N}, D)$ consisting of a *node set* $\mathcal{N}$ and an *associated relation* $D : \mathcal{N} \leftrightarrow \mathcal{N}$ as its objects, and
- for every two objects $(\mathcal{N}_1, D_1)$ and $(\mathcal{N}_2, D_2)$ some or all partial or total functions $F : \mathcal{N}_1 \nrightarrow \mathcal{N}_2$ between the respective node sets as homomorphisms, where composition is defined as the usual composition of functions. $\square$

Typically there will be some relation between homomorphisms and associated relations, but there is no need to assume some fixed relation such as $F^{\smile};D_1 \subseteq D_2;F^{\smile}$, which is reasonable in many contexts, but fails for example in term graphs with second-order variables [6,9].

Definition 3.2: A **setting** for our approach to internal graph attribution consists of the following:

- a category $(\mathcal{G}, \mathcal{M})$ of **underlying graphs** with objects (graphs) $G \in \mathcal{G}$ and homomorphisms $F \in \mathcal{M}$,
- a category $\mathcal{S}$ of superficial graph structures, and
- a functor S from $(\mathcal{G}, \mathcal{M})$ into $\mathcal{S}$; for every graph $G \in \mathcal{G}$ we denote the node set of $\mathsf{S}.G$ by $\mathcal{N}_G$ and call it the node set of G, and the same with the associated relation $D_G : \mathcal{N}_G \leftrightarrow \mathcal{N}_G$; we sometimes do not distinguish notationally between a homomorphism $F \in \mathcal{M}$ from G_1 to G_2 and its image under S, which is then a (partial) function $F : \mathcal{N}_1 \nrightarrow \mathcal{N}_2$.

There has to be a *subgraph restriction* in $(\mathcal{G}, \mathcal{M})$ along S, such that for every total injective homomorphism[1] ι from $(\mathcal{N}_1, D_1)$ to $(\mathcal{N}_2, D_2)$ and for every graph $G_2 \in \mathcal{G}$ with $\mathsf{S}.G_2 = (\mathcal{N}_2, D_2)$ there are a graph $G_1 \in \mathcal{G}$ and a monomorphism $\iota' \in \mathcal{M}$ from G_1 to G_2 such that $\mathsf{S}.\iota' = \iota$. $\square$

There may be other components to the underlying graphs such as node and edge labels, and restrictions of different kinds on the graphs themselves, such as restrictions on the (co-) occurrences of edges with certain labels at nodes with certain labels etc. There also may be restrictions on the possible homomorphisms, such as commutativity with other components, but all these possibilities are essentially irrelevant to the discussion in this paper.

We now enrich the underlying graphs with attributes that we attach to their superficial structure made accessible by the functor S:

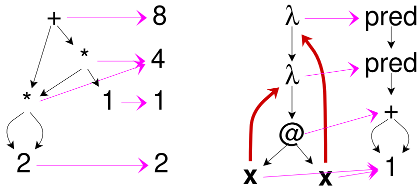
Definition 3.3: The **attribute extension** of a setting includes

- a set $\mathcal{A}$ of **attribute identifiers**,
- a category where objects are **internally attributed graphs**, i.e., pairs (G, A_G) consisting of graphs $G \in \mathcal{G}$ with in addition an *attribution function* $A_G : \mathcal{A} \to \mathcal{N}_G \nrightarrow \mathcal{N}_G$, and where a homomorphism F from (G_1, A_1) to (G_2, A_2) is a homomorphism $F \in \mathcal{M}$ from G_1 to G_2 that in addition preserves attribution: $\forall \alpha : \mathcal{A} \bullet F^{\smile};A_1.\alpha \subseteq A_2.\alpha;F^{\smile}$.

A homomorphism F is called **strict** if F also *preserves attributelessness:* $F;A_2.\alpha \subseteq A_1.\alpha;F$. $\square$

[1] This condition is rather strong, but an appropriate weaker condition would be technically very involved.

Consider the following examples of attributed term graphs.



The left part of the first graph corresponds to the term "$2 \cdot 2 + 2 \cdot 2 \cdot 1$" from arithmetic, and the violet resp medium grey, wide tipped arrows represent the attribution function for some fixed attribute identifier; here the attributes of the nodes are the values of the corresponding subexpressions.

In the second graph, the attributes are expressions denoting the numbers of free variables in the respective subexpressions, and the left part of the graph corresponds to the λ-expression "$\lambda x .\lambda f. f\,x$", where variable bindings are drawn with thick, red resp. dark grey, usually curved arrows — these can in the current setting be considered as edges with just a special kind of edge labels; for more information about this kind of second-order term graphs see [7,9].

We still have to show consistency of the above definition:

Theorem 3.1: The category of internally attributed graphs defined in Def. 3.3 is well-defined.

Proof: For two homomorphisms F_1 from (G_1, A_1) to (G_2, A_2) and F_2 from (G_2, A_2) to (G_3, A_3) and for any attribute identifier α we have:

$$(F_1^{\smile};F_2);A_1.\alpha = F_2^{\smile};F_1^{\smile};A_1.\alpha \subseteq F_2^{\smile};A_2.\alpha;F_1^{\smile} \subseteq A_3.\alpha;F_2^{\smile};F_1^{\smile} = A_3.\alpha;(F_1^{\smile};F_2)\ ,$$

and together with the homomorphism property in the underlying category we obtain that $F_1;F_2$ is again a homomorphism. All the other conditions are trivial. □

Let us now consider pushouts. Let two homomorphisms F_1 from (G_1, A_1) to (G_2, A_2) and F_2 from (G_1, A_1) to (G_3, A_3) on internally attributed graphs be given. If F_1 from G_1 to G_2 and F_2 from G_1 to G_3 have a pushout in $(\mathcal{G}, \mathcal{M})$, consisting of F_3 from G_2 to G_4 and F_4 from G_3 to G_4, then G_4 still needs not be the graph part of a possible pushout object in the category of internally attributed graphs, since $R := F_4^{\smile};A_3^{\smile};F_4;F_3^{\smile};A_2^{\smile};F_3$ need not be contained in the identity on $\mathcal{N}_4$. For making a univalent attribution function possible, we have to find a homomorphism that at least identifies nodes in the equivalence closure of R. This could also be defined as a kind of equaliser for $F_1;A_2;F_3$ and $F_2;A_3;F_4$.

In short, even if there is a pushout in the underlying category, there need not exist a pushout of internally attributed graphs, and even if that exists, it not necessarily coincides with the pushout in the underlying category — the forgetful functor from the category of internally attributed graphs to the underlying category does not preserve pushouts (nor does S).

4. Attribution Rules and Their Application

Since we intend to define term graph attributions in a declarative manner, we now introduce *attribution rules* that correspond to the attribute definitions of attribute grammars. We choose a rather wide definition that allows very different settings with respect to the attributes expected before application.

For simplicity, we let an attribution rule be a single internally attributed graph with an additional attribution function, and from this single graph we then construct the two rule sides of a single-pushout rule:

Definition 4.1: An **attribution rule** is an internally attributed graph (G,A) together with a function

$A' : \mathcal{A} \to \mathcal{N} \nrightarrow \mathcal{N}$ containing *new attributions*, such that G with A' merged into A is again an attributed graph, called the *right-hand side* of the rule.

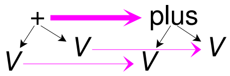
The *left-hand side* of such an attribution rule is the subgraph induced by the set containing exactly those nodes that are reachable via $(D \cup A)$ from the connected component wrt. D containing the sources of the new attribution arrows.

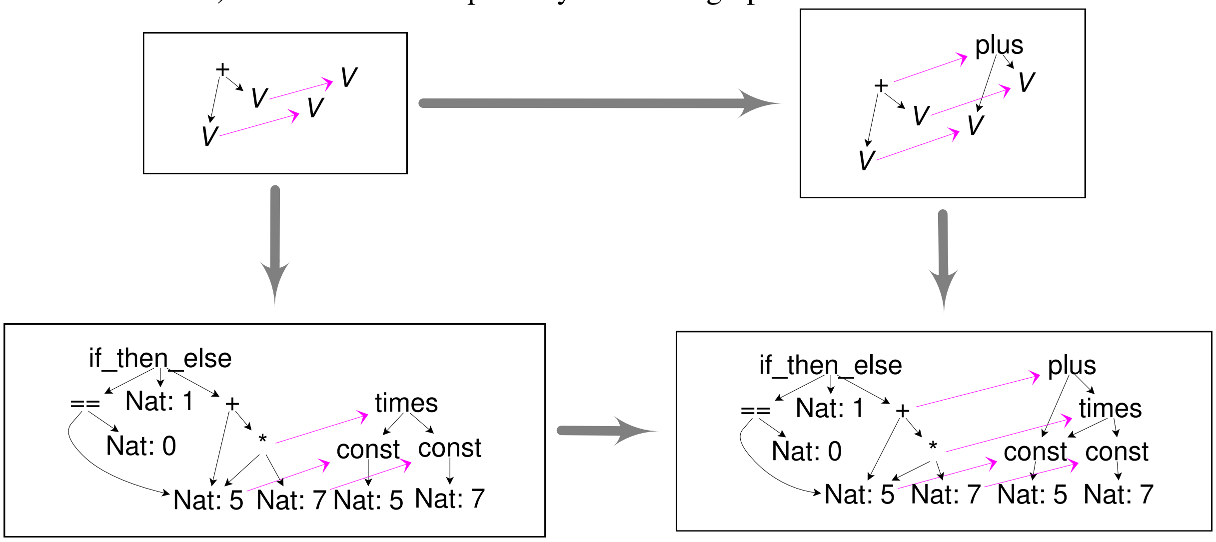
A *matching* of an attribution rule is a homomorphism from its left-hand side to some application graph.

Application of an attribution rule is defined as the pushout of the matching on one hand and the injection of the left-hand side into the right-hand side on the other hand, and application is restricted to those cases where the application graph is not isomorphic to the pushout object. □

In most settings this pushout always exists if the matching is a strict homomorphism. The last condition ensures that there are no "invisible" attribution rule applications.

As an example attribution rule application we apply the simple attribution rule drawn to the right (with the new attributions drawn as thicker arrows) to a redex inside a partially attributed graph.





Definition 4.2: An attributed graph is **consistent** wrt. a set of attribution rules, if the application pushout exists for all matchings.

An attributed graph is **final** wrt. a set of attribution rules, if no rule can be applied.

An attributed graph G is **well-attributed** wrt. a set $\mathcal{R}$ of attribution rules, if G is both consistent and final wrt. $\mathcal{R}$. □

From the compositionality of homomorphisms we obtain the following trivial fact:

Lemma 4.1: If there is a matching for an attribution rule r in the graph G, and if the application pushout exists, then there is a matching into the pushout object, too. Also for this matching the application pushout exists, but is trivial, so repeated application is impossible. □

This ensures that the application of a rule preserves consistency wrt. that rule, but it still could clash with other rules, so we define:

Definition 4.3: A set $\mathcal{R}$ of attribution rules is **consistent** if for every application of one of the rules to some graph that is consistent wrt. $\mathcal{R}$, the result graph is also consistent wrt. $\mathcal{R}$ and still allows all the same matchings. $\square$

Therefore, if two different applications of rules from a consistent set do not commute, then one is subsumed in the other and we obtain the following result:

Theorem 4.1: The attribution rule application relation wrt. a consistent set of attribution rules is **confluent.** $\square$

Termination of attribute calculation for finite attribute identifier sets on finite graphs is obvious if only strict matchings can be used, because then every rule application has to add at least one attribution arrow. Without strict matchings we need Lemma 4.1 to obtain termination at least if in addition the attribution rule set is finite.

5. Applications to Compilation

In the context of inspection and verification of compilation results, see e.g. [3], it can be useful to see the source and target code directly side by side with explicit correspondences indicated between the two versions.

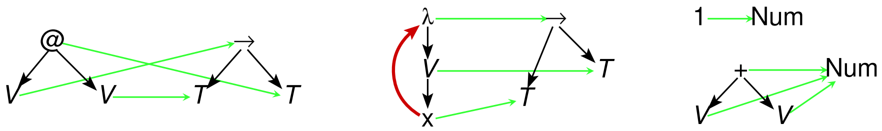
If the compiled code is the result of some attribute evaluation (as in the original motivation of attribute grammars), then our approach can provide some visualisation of the compilation relation. In fact, the example attribution rule application of the last section is part of a simple translation example where the target subgraph of the attribute arrows is the result of translating the source subgraph into a different language.

However, our approach has difficulties to deal with inherited attributes in shared nodes: since only one attribution rule application with a strict homomorphism may initialise an attribute of a shared node, only certain kinds of interaction between attributes inherited via different edges can be formulated as attribution rules. This suggests that at least for parts of the translation a tree attribution will be more natural — tree attributions can of course also be defined and performed in our approach.

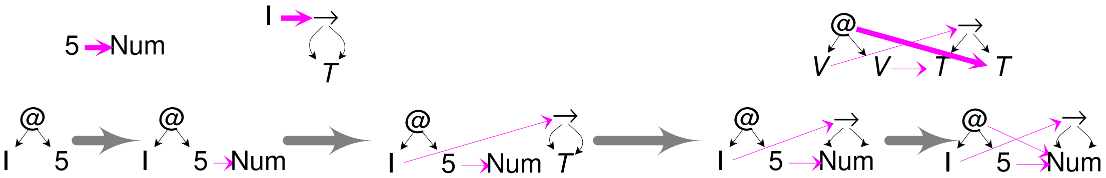
6. Typing

Typing of term graphs as presented in [9] can also be seen as an attribution, where typing is one attribute identifier and, as in pure type systems [1], the structure of types is essentially the same as that of the typed language, so that a homogeneous, internal treatment in the context of term graphs is only natural.

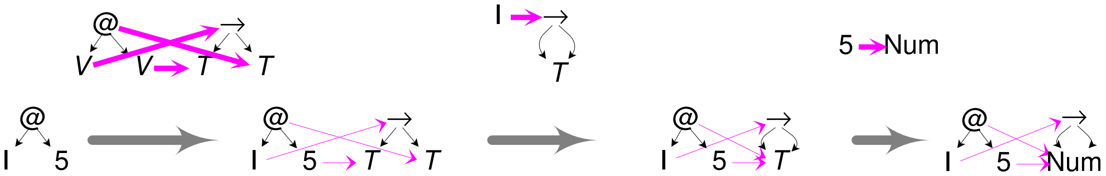
In [9], well-typed term graphs are characterised by the simultaneous existence of homomorphisms from *typing elements* into the typed graph. A few example typing elements are the following — the typing function is denoted by green resp. thin, light grey arrows:



Typing elements essentially correspond to typing rules in pure type systems. Our attribution rule application mechanism can even be used to mimick conventional type derivations in pure type systems. There, all rules besides the axioms start from precedents that have to be established before applying the rule — this corresponds to "old attributions" in attribution rules. Letting the root attribution of a typing element be the only new attribution when that typing element is used as an attribution rule can therefore be used to enforce the corresponding attribution derivations (which will proceed only along strict matchings) to be constrained in the same way as the conventional typing derivations — however, this will only work if it is possible to assign all the leaves there final types, or, if instantiation of type variables is allowed between attribution rule applications. Consider the following example where such an instantiation is necessary before the typing element for "@" can be applied:



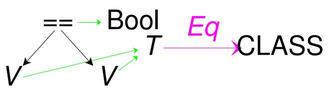
However, the simplest way to turn typing elements into attribution rules is to consider all typing arrows as new attributions — a typing derivation will then have attribution rule applications with non-strict matchings in particular in all those cases where unification is necessary. Typing deriviations defined this way have arbitrary freedom how to traverse the graph to be typed, and allow propagation of typing information in arbitrary directions:



7. Outlook and Conclusion

We have seen in the last section how simple typing can be handled by attributions. A more advanced application in this direction is to use *attributed types* to mimick Haskell type classes.

As an example, the typing element for equality would be as drawn to the right, with an attribution for the attribute identifier "Eq" at the node standing for the type of the compared objects.



In this case, the target of the attribution is rather irrelevant, so we designate a unique "CLASS" node to be the target of all such type class attributions. Standard types will therefore have attribute arrows for many standard classes pointing to this "CLASS" node.

However, some details of this approach still need to be worked out, in particular whether it extends neatly to so-called constructor classes and multi-parameter classes.

So far, we have only seen applications to term graphs, which are of course also most natural given the close relation to the original realm of attribute grammars.

Nevertheless, it should also be possible to find interesting applications in other fields, too, e.g. in networks or in knowledge representation graphs.

References

[1] Hendrik P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, T.S.E. Maibaum, *Handbook of Logic in Computer Science, Vol. 2*, pages 117–309. Oxford Univertity Press, 1992.

[2] Chris Brink, Wolfram Kahl, Gunther Schmidt (eds.). *Relational Methods in Computer Science.* Advances in Computing. Springer-Verlag, Wien, New York, 1997. ISBN 3-211-82971-7.

[3] Th. Gaul, G. Goos, A. Heberle, W. Zimmermann, W. Goerigk. An Architechture for Verified Compiler Construction. In *Joint Modular Languages Conference JMLC'97, Linz, Austria*, March 1997.

[4] H. Göttler. Attribute Graph Grammars for Graphics. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg (eds.), *Graph-Grammars and Their Application to Computer Science, 2nd International Workshop*, pages 130-142. LNCS. Springer-Verlag, 1983.

[5] Wolfram Kahl. Can Functional Programming Be Liberated from the Applicative Style?. In Bjørn Pehrson, Imre Simon (eds.), *Technology and Foundations, Information Processing '94, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August – 2 September 1994, Volume I*, pages 330–335. IFIP Transactions. North-Holland. IFIP, 1994.

[6] Wolfram Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen.* Reihe Informatik. Herbert Utz Verlag, München, 1996. ISBN 3-931327-60-4, zugleich Dissertation an der Fakultät für Informatik, Universität der Bundeswehr München.

[7] Wolfram Kahl. Relational Treatment of Term Graphs With Bound Variables. *Journal of the IGPL* **6** (2), 259–303 (March 1998).

[8] Wolfram Kahl. The **H**igher **O**bject **P**rogramming **S**ystem — User Manual for HOPS, Fakultät für Informatik, Universität der Bundeswehr München, February 1998. URL http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/.

[9] Wolfram Kahl. Internally Typed Second-Order Term Graphs. In *Graph Theoretic Concepts in Computer Science, 24th International Workshop, WG '98, Smolenice Castle, Slovakia, June 1998, Proceedings.* LNCS. Springer-Verlag, 1998. URL http://diogenes.informatik.unibw-muenchen.de:8080/kahl/PAPERS/Kahl-1998d_ttgv.ps.gz. To appear.

[10] Donald E. Knuth. Semantics of Context-Free Languages. In *Mathematical Systems Theory, Vol. 2*, pages 127–145. Springer-Verlag, New York, 1968.

[11] Donald E. Knuth. The Genesis of Attribute Grammars. In Pierre Deransart and Martin Jourdan, *Attribute Grammars and their Applications (WAGA)*, pages 1–12. LNCS 461. Springer-Verlag, New York–Heidelberg–Berlin, 1990.

[12] Michael Löwe, Martin Korff, Annika Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, M.C.J.D. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, 1993.

[13] Jukka Paakki. Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation. *ACM Computing Surveys* **27** (2), 196–255 (1995).

[14] Gunther Schmidt, Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg/New York, 1993.

[15] Andreas Schürr. Programmed Graph Replacement Systems. In Grzegorz Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1*, pages 479–546. World Scientific, Singapore, New Jersey, London, Hong Kong, 1997.

[16] A. Schütte. *Spezifikation und Generierung von Übersetzern für Graphsprachen durch attributierte Graphgrammatiken.* Dissertation, EWH Koblenz. EXpress Edition, 1987.

[17] Hans Zierer, Gunther Schmidt, Rudolf Berghammer. An Interactive Graphical Manipulation System for Higher Objects Based on Relational Algebra. In *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 68–81. LNCS 246. Springer-Verlag, Bernried, Starnberger See, June 1986.