Meta Typing is Compatible to the Typed SPO Approach

Stefan Gruner
Fachbereich Informatik
Technische Universität Berlin
http://tfs.cs.tu-berlin.de/~stefan/

Abstract. Meta-typing, as for example employed in the PROGRES environment, is syntactic sugar making the specification of typed graph grammar systems more convenient. This paper presents an approach of meta-typing that fits in the well-elaborated framework of the typed single pushout (SPO) approach to graph transformation. In a first step the node meta-typing system of PROGRES is generalized such that edges may be subject to meta-typing, too. In a second step it is shown how any meta-typed graph grammar can be transformed into an equivalent ordinary typed SPO graph grammar.
The main result of this contribution is that the presented concept of meta-typing is compatible to the SPO theory and can be implemented, therefore, into the SPO-based graph grammar engineering environment AGG without spoiling its formally sound basis. As a more practical result it can be claimed that the well-known application gap between PROGRES and AGG will certainly become smaller in the immediate future.

1 Introduction

A remarkable advantage of the well-known PROGRES system [10] is the possibility of using meta-typed nodes in the graphs of the left hand sides of the graph replacement rules. For example: If there is some node (vertex) v of type x occuring in some host graph G, and if there is some node (vertex) $\hat{v}$ of type X occuring in some left hand graph L of some graph replacement rule r which shall be applied to G, then $\hat{v}$ may not be mapped onto v in case that X is not a meta-type of x.[‡] Continuing the example, the meta-type X is an abbreviation subsuming every of its possible *ground-types* $\{x_1, \ldots, x_n\}$ as explained in more detail in [8]. Thus: the advantage of meta-typing turns out as a convenient reduction of alternatives in a graph grammar specification.

[‡] In the terminology of PROGRES (belonging to the algorithmic approaches of graph transformation rather than to the algebraic ones), the nodes and edges occuring in a replacement rule are called "node *variables*" and "edge *variables*". The concept of that terminology is to instantiate those variables with actual host graph nodes and edges at rule application time. Finding such an instantiation, however, is equivalent to finding homomorphic mappings from the nodes and edges of a rule graph onto some nodes and edges of a host graph (expressed in the terminology of the algebraic graph transformation approach). For a comparative discussion of algorithmic versus algebraic graph transformation approaches the reader is referred to [12].

While the PROGRES system allows meta-typing only on nodes, the AGG system [1] allows no meta-typing at all, because this system is immediately derived from the typed single pushout approach (SPO) as described in [4]. Therefore, the goal and the contribution of this paper is to show that meta-typing is not incompatible with the SPO approach, such that the concept of meta-typing is implementable in AGG, too. Unlike PROGRES, the here presented approach allows meta-typing also on edges (arcs). Finally, the typing systems of both PROGRES and AGG are expressed and compared in terms of a more general formalism.

1.1 Genealogy of the Meta-Typing Approach

The meta-typing approach presented in this paper combines mainly two ideas that can be found in the recent literature on graph transformation systems. In [6], you can find a partially-ordered labeling system wherein *variables* play a similar role as the meta-types which belong to the topics of this contribution. In [5] you can find a concept of a *type-graph* restricting the structure (shape, form, *gestalt*) of graphs that shall be graph-grammatically generated. Type schemas, as introduced in the following section, may be viewed as a generalized cross-over of partially-ordered labeling systems and non-hierarchically labeled type graphs.

2 Type Schema

2.1 Motivation

The lattice-shaped *type schema* of a graph transformation system, as introduced below, plays the role of a simple terminological system (*begriffssystem*) [7] in a certain universe of discourse wherein the graph transformation system shall operate. Of course, the type schema shall reflect the usual concepts of nodes (vertices) and edges (arcs) as well as the possible adjacency relations between them. Nodes and edges are disjoint concepts here, in contrast to the more general ALR graph transformation approach of [2].

2.2 Definition (Syntax)

Let $\Sigma := S \uplus \{\bot, \top\}$ a finite supply of symbols such that $S \neq \emptyset$. A quadruple $\mathfrak{S} := (\Sigma, \prec, src, tgt)$ is called *type schema* if all the following conditions hold:

- The reflexive, transitive, and antisymmetric partial order $\prec :\subseteq (\Sigma \times \Sigma)$ defines a lattice [3] such that $\forall \sigma \in \Sigma : (\bot \prec \sigma \prec \top)$.
- $S = A \uplus V$ disjoint such that $V \neq \emptyset$ and $\forall \alpha \in A \ \forall \nu \in V \ \forall \sigma \in \Sigma \ \forall \tau \in \Sigma :$ $((\sigma \prec \alpha \prec \tau) \wedge (\sigma \prec \nu \prec \tau)) \Longrightarrow ((\sigma = \bot) \wedge (\tau = \top))$.

- $src : A \rightharpoonup V$ and $tgt : A \rightharpoonup V$ partial but equally defined such that $def(src) = def(tgt) \subseteq dom(src) = dom(tgt) = A$ and $\forall \alpha \in A \; \forall \beta \in A : (\alpha \prec \beta) \Longrightarrow ((src(\alpha) \prec src(\beta)) \wedge (tgt(\alpha) \prec tgt(\beta))$ wherever src and tgt are defined.†

2.3 Comment

Now that the syntax definition of a type schema is given, it is necessary to provide some comments on the meaning as well as on the intended application of these concepts:

- In the terminology of lattice theory, the set $T := \{\tau \in S \mid \nexists \sigma \in S : \sigma \prec \tau \wedge \sigma \neq \perp \wedge \sigma \neq \tau\}$ contains the *atoms* of the lattice structure. In order to clarify the intention, the atoms are called *ground types* here. $\mathfrak{T} := S \backslash T$ is the set of *meta types*.¶ Any host graph G must be ground-typed in T. If $r : L \rightharpoonup R$ is a graph replacement rule, the objects L and $r(L)$ may be ground-typed or meta-typed in $S = T \uplus \mathfrak{T}$. The generative part $R \backslash r(L)$, however, must be ground-typed in T because of the ground-typed host graphs. A rule object x may be mapped onto a host object y, if $typ(y) \prec typ(x)$; please note that $\forall \tau \in \Sigma : \tau \prec \tau$ because of the reflexivity condition.

- The subset A contains the arc-related types and meta types, whereas V contains the vertex-related types and meta types.

- The adjacency functions src and tgt properly assign certain source node types, target node types, and edge types to each other. In contrast to the typing system of the SPO approach, it is not necessary that every occurring arc type may be combined with every occurring vertex type here.

- In [8] it is shown that the lattice-structured meta-typing in Σ is a sufficient reason for src and tgt to be functions: They are not required to be relations, as those adjacency relations can be simulated by the adjacency functions of above, together with the property *inheritance* represented by the lattice order $prec$. For the same reason, the adjacency functions src and tgt need not to be total functions in A (thus: $def(src) \neq dom(src) = A$ — the same with tgt), because the property of totality can be simulated by partiality with help of the semantic inheritance relation $\prec$ as well.

---

† Please note that $dom(f)$ is the "land" in which the individuals x_i subject to some partial function f are "living", whilst $def(f)$ is the set of individuals x_j for which $f(x_j)$ is not undefined. Obviously we have $dom(f) = def(f)$ if f is a total function.

¶ When operating with *attributed* graph grammars, the attribute set of a meta type τ must be obviously a subset of the attribute set of each of its ground types $t \prec \tau$, but that subject is out of the scope of this paper.

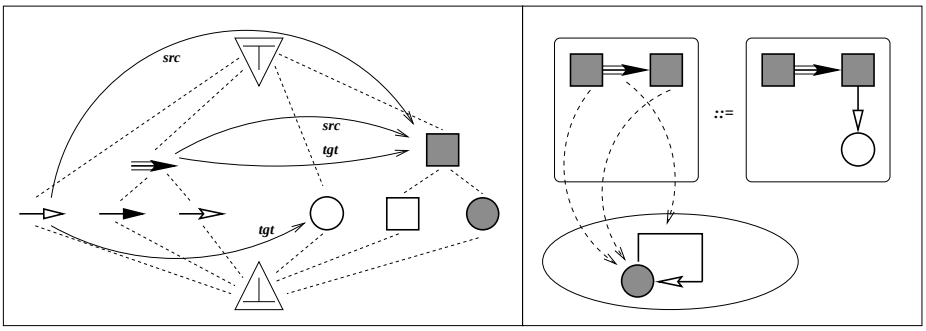

Fig. 1 *type schema and meta-typed rule application*

3 Example (Part I)

Fig.1 shows how a meta-typed left hand side of a graph replacement rule is mapped by a conjective (i.e.: not injective) homomorphism onto a ground-typed host graph according to a given type schema. In the schema part of Fig.1, the lattice relation $\prec$ is symbolized by dotted lines.

4 Meta Typed Graph Grammars

4.1 Definition (Syntax)

Let $\mathfrak{S}$ be a type schema as characterized by *Def.2.1* and $\mathcal{G} = \{g_0, r_1, \ldots, r_n\}$ a graph grammar whereby $r_i := L_i \rightharpoonup R_i$ for all $i = 1, \ldots, n < \infty$. Further it supposed that all r_i are finite structures themselves. $\mathcal{G}$ is called an $\mathfrak{S}$-*meta-typed graph grammar*, if the following conditions are satisfied:

- Every node and every edge of the start graph g_0 is ground-typed with an atom of $\mathfrak{S}$.
- In every left-hand-side rule graph L_i, every node or edge x is typed in $\mathfrak{S}$ such that $\bot \neq typ(x) \neq \top$. (The total function $typ(\cdot)$ provides the graph elements with types.)
- $typ(x) = typ(r_i(x)) \in \mathfrak{S}$ for all nodes or edges $x \in def(r_i) \subseteq L_i$.
- all generative nodes or edges $y \in R_i$ (for which $\nexists x \in def(r_i) : y = r_i(x)$) are ground-typed with an atom of $\mathfrak{S}$.

4.2 Typing Lemma

Let $\mathfrak{S}$ be a type schema (as characterized by *Def.2.1*) and $\mathcal{G} = \{g_0, r_1, \ldots, r_n\}$ a $\mathfrak{S}$-meta-typed graph grammar (as characterized by *Def.4.1*). Then $\mathcal{G}$ can be

losslessly translated into an SPO graph grammar $\mathfrak{G} = \{\gamma_0, \varrho_1, \ldots, \varrho_m\}$ such that $m \geq n$ and $\forall G : G \in \mathcal{L}(\mathcal{G}) \iff G \in \mathcal{L}(\mathfrak{G})$ ground-typed in $T \subset \mathfrak{S}$.

4.3 Proof Sketch

According to *Motivation 2.1* and *Comment 2.3*, the lemma is shown by a nondeterministic fixpoint construction relying on the supposition that type schemas, graph grammars, and graph replacement rules are finite structures.

- Let *get* be an operation returning two graph objects x and y (nodes or edges) from a given graph replacement rule $r : L \rightharpoonup R$ such that $y = r(x)$, whereby y may be undefined because of r being partially defined.
- Let $proj_1$ and $proj_2$ be projections from tuples as usual.
- Let $retype_{(\tau)}$ be an operation on the typing of a graph object x such that $typ(retype_{(\tau)}(x)) := \tau$ whereby $typ(x) = \sigma$ with $\tau \prec \sigma$, $\tau \neq \perp$, and $\tau \neq \sigma$ in the given type schema $\mathfrak{S}$. (Otherwise, $retype_{(\tau)}$ is the void operation which does nothing.)
- Let *create* be an operation constructing a new graph replacement rule q from a given graph replacement rule p by applying $retype_{(t)}(proj_1(get(p)))$ at the same time with $retype_{(t)}(proj_2(get(p)))$, where $t \neq \perp$ is arbitrarily taken from $\mathfrak{S}$.
- Let *augment* be an operation performing $\mathcal{G}_{new} := \mathcal{G}_{old} \cup create(r)$, where $r \in \mathcal{G}_{old}$ ad libitum.
- $\hat{\mathfrak{G}} := FIX(augment^*(\mathcal{G}))$ is the maximum of the type replacement procedure, and of course $\hat{\mathfrak{G}}$ exists since $\mathfrak{S}$ is finite.
- Let *flush* be an operation performing $\mathfrak{G}_{new} := \mathfrak{G}_{old} \setminus \varrho$, where $\varrho \in \mathfrak{G}_{old}$ such that $typ(proj_1(get(\varrho))) \in \mathfrak{T}$ or $typ(proj_2(get(\varrho))) \in \mathfrak{T}$.
- $\mathfrak{G} := FIX(flush^*(\hat{\mathfrak{G}}))$ is the desired graph grammar containing only ordinary typed graph replacement rules according to the definitions of the SPO transformation approach.

4.4 Comment

Resulting from this construction one can show that in any case a left-hand-side graph L of a meta-typed rule r *matches* to a ground-typed host graph G via the type inheritance order $\prec$ of $\mathfrak{S}$, there is exactly one ground-typed rule ϱ derived (deduced) from r such that ϱ *is applicable* to G and, moreover, $\varrho(G) = r(G)$ is true for the application results. Thus: the applicability of ground-typed rules is a consequence of the match-ability of meta-typed rules in this approach.

5 Example (Part II)

Fig.2 shows the result of applying the operations of *Proof Sketch 4.3* to the meta-typed graph replacement rule taken from Fig.1: eight ground-typed rules have come into existence, *exactly one of them is applicable* to the given hostgraph (taken from Fig.1, too) — which intuitively holds true also in the general case.

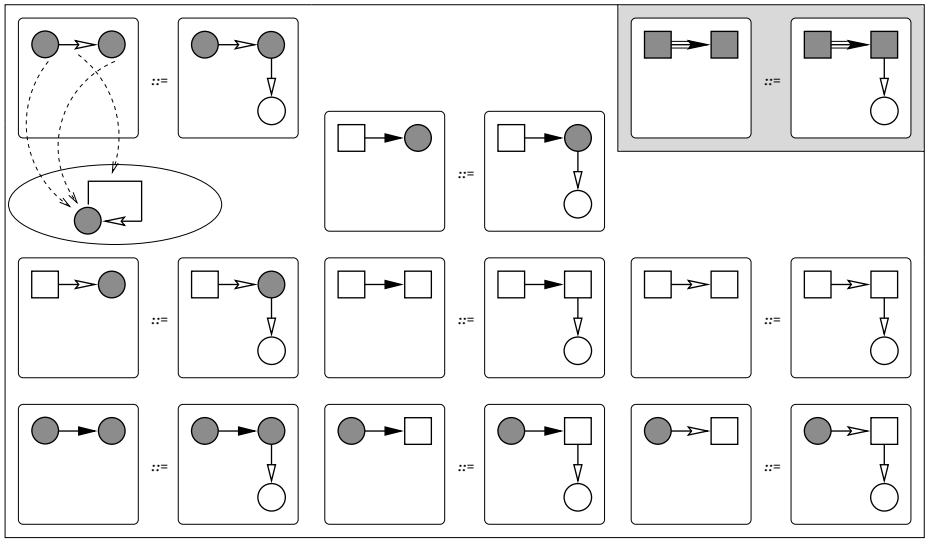

Fig. 2 *ground-typed rules after expansive meta-type reduction: one is matching*

6 The Type Systems of PROGRES and AGG until Now

Now that a notion of type schemas is supplied by *Definition 2.1*, the typing mechanisms of PROGRES and AGG can be compared quite easily. (A similar comparison can be found on top of Table 7.1, page 531 of [12], however, table representations are considered as less intuitive in general.)

6.1 PROGRES

The type schema of the PROGRES language is called *graph schema* in the PROGRES terminology. Except of the possibility of special edge-type cardinality-constraints (the operational semantics of which is yet rather vague), a PROGRES graph schema can be characterized as follows (which is illustrated by Fig.3):

- $\forall \alpha \in A\ \forall \sigma \in \Sigma\ \forall \tau \in \Sigma : (\alpha \prec \tau \Longrightarrow \tau = \top) \wedge (\sigma \prec \alpha \Longrightarrow \sigma = \bot)$, in words: all edge types in the lattice are atoms which means that there is no meta-typing on edges at all.

- $src : A \to V$ and $tgt : A \to V$ with $def(src) = def(tgt) = dom(src) = dom(tgt) = A$ are total functions ("$\to$") instead of partial ones ("$\rightharpoonup$") which is a logical consequence of the lacking meta-typing on edges. (Please remember that $def(f) \subset dom(f)$ but $def(f) \neq dom(f)$ for any partial function f.)

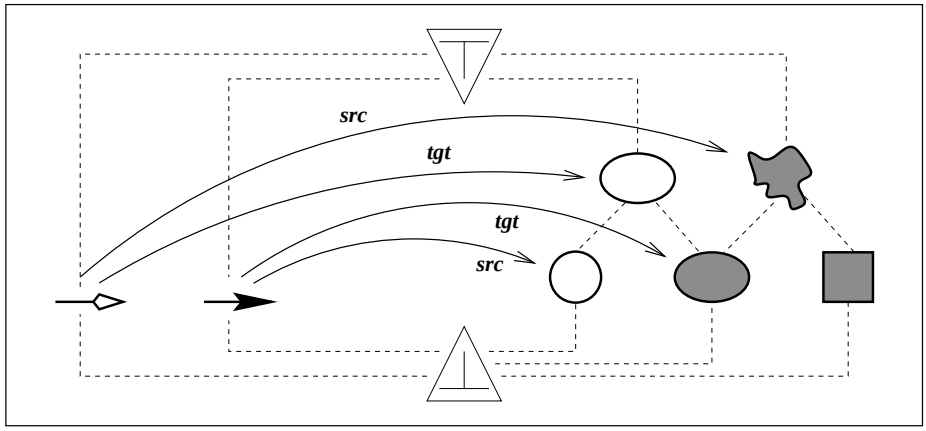

Fig. 3 *example of a* PROGRES *graph schema, expressed as type schema*

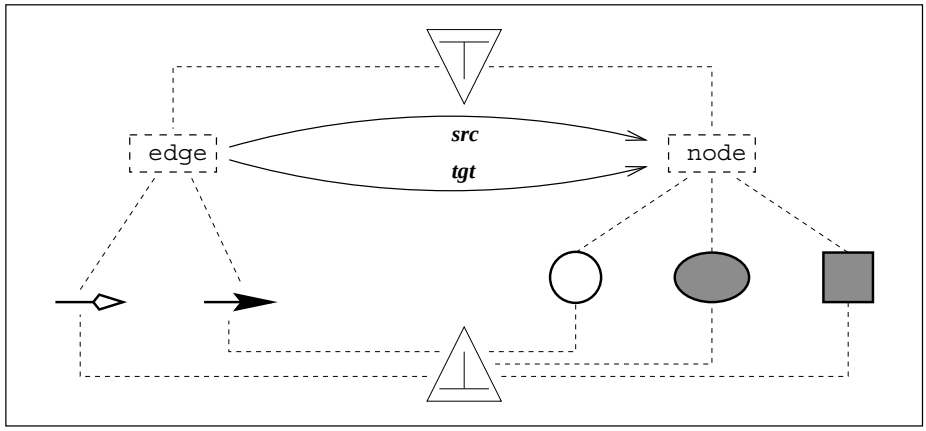

Fig. 4 *example of the poor typing in* AGG*, expressed as type schema*

6.2 AGG

From the view of a graph grammar practioneer (specificator, engineer), the typing system of AGG seems to appear really poor, structure-less and error-prone because of its greatest-possible generality wherein "everything" is allowed.*

- $\forall t \in S\ \forall \sigma \in \Sigma\ \forall \tau \in \Sigma : (t \prec \tau \Longrightarrow \tau = \top) \wedge (\sigma \prec t \Longrightarrow \sigma = \bot)$, in words: there is no meta-typing at all.

- $src = tgt$ are no functions at all, but left-right-total relations ($A \times V$) instead. For better intuition, these relations are *simulated* in Fig.4 by two *virtual* meta types `node` and `edge` which are, of course, not available in the real AGG environment.

7 Summary and Future Work

Meta-typing is considered as powerful means of abstraction in order to avoid clumsy or error-prone specifications in graph grammar engineering. While the graph grammar engineering environment (GGEE) of PROGRES provides the facility of meta-typing at least partially, namely on nodes, the GGEE of AGG allows no meta-typing until now. This paper has shown in a semi-formal manner that certain forms of meta-typed graph grammars can easily be translated via a lattice-shaped type schema into ordinary typed SPO specifications, as required by the AGG environment. Therefore, it is possible to implement this concept of meta-typing on nodes as well as on edges in the GGEE of AGG, too.

Among other improvements and features, the meta-typing facility is currently being incorporated into the software architecture of the AGG environment. Moreover it is planned to supply the AGG system with a concept of meta-typing that is not only concerned with rules but also with the host-graphs that rules are operating on. Such a concept of host-graph meta-typing should support the already existing object-oriented features of the GGEE of AGG which are due to the underlying JAVA [9] programming language.

Acknowledgments

Good cooperation with Olga Runge, Thorsten Schultzke and Gabi Taentzer from the AGG development group as well as with Manfred Münch from the PROGRES development group is gratefully acknowledged. Uwe Wolter and Hartmut Ehrig as well as the referees of GRATRA 2000 have given valuable remarks and comments on the draft of this paper.

* Please note that arcs and vertices are strictly separated concepts in the GUI-relevant top layer of the AGG software architecture, while they are not strictly separated concepts in the basic ALR layer of the AGG architecture. However, the ALR layer is not visible to the user. For this reason it is legal to express the AGG typing properties by means of a type schema as given above.

References

1. AGG, `http://tfs.cs.tu-berlin.de/agg/`
2. R.Arlt, M.Röder, *Grundlegende Datenstrukturen und Algorithmen zur Implementierung von algebraischen Graphgrammatiken.* Report, Fachbereich Informatik, Technische Universität Berlin 1989
3. G.Birkhoff, *Lattice Theory.* American Math. Soc. 1948
4. A.Corradini, H.Ehrig, R.Heckel, M.Korff, M.Löwe, L.Ribeiro, A.Wagner, *Algebraic Approaches to Graph Transformation: pt.II (Single Pushout Approach and Comparison with Double Pushout Approach).* chpt.4, pp.247-312 in [11]
5. A.Corradini, H.Ehrig, M.Löwe, U.Montanari, J.Padberg, *The Category of Typed Graph Grammars and its Adjunctions with Categories of Derivations.* J.Cuny, H.Ehrig, G.Engels, G.Rozenberg (Eds.), Graph Grammars and their Application to Computer Science: 5^{th} International Workshop. LNCS 1073, Springer-Verlag, Berlin 1995
6. H.Ehrig, U.Montanari, F.Parisi-Presicce, *Graph Rewriting with Unification and Composition.* H.Ehrig, M.Nagl, G.Rozenberg (Eds.), Graph Grammars and their Application to Computer Science: 3^{rd} International Workshop. LNCS 291, Springer-Verlag, Berlin 1987
7. B.Ganter, R.Wille, *Formale Begriffsanalyse: mathematische Grundlagen.* Springer-Verlag, Berlin 1996
8. S.Gruner, *Eine schematische und grammatische Korrespondenzmethode zur Spezifikation konsistent verteilter Datenmodelle.* Shaker-Verlag, Aachen 1999
9. JAVA, `http://www.javasoft.com/`
10. PROGRES, `http://www-i3.informatik.rwth-aachen.de/research/progres/`
11. G.Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation: vol.1 (Foundations).* World Scientific, Singapore 1997
12. A.Schürr, *Programmed Graph Replacement Systems.* chpt.7, pp.479-546 in [11]

This article has been processed with LaTeX in an LNCS-compatible style.