Electronic Communications of the EASST
Volume 18 (2009)

Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)

A Generic Graph Transformation, Visualisation, and Editing Framework in Haskell

Scott West, Wolfram Kahl

18 pages

Guest Editors: Artur Boronat, Reiko Heckel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Generic Graph Transformation, Visualisation, and Editing Framework in Haskell

Scott West[1], **Wolfram Kahl**[2]

[1]scott.west@inf.ethz.ch, http://se.inf.ethz.ch/people/west/
Chair of Software Engineering, Department of Computer Science, ETH Zürich, Switzerland

[2]kahl@cas.mcmaster.ca, http://sqrl.mcmaster.ca/~kahl/
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Abstract: Graph transformation, visualisation, and editing are useful in many contexts, and require domain-specific customisation. However, many general-purpose graph solutions lack customisability in at least one area.

We present a framework that aims to allow polished customisation in all three areas, using the powerful abstraction capabilities of the pure functional programming language Haskell. The design of our framework integrates and adapts time-tested object-oriented designs into a purely functional framework, and uses current user-interface libraries (GTK+ and Cairo) to achieve polished presentation.

Our framework provides both a low-level programmed approach to graph transformation, and, on top of this, high-level approaches including SPO and DPO, which are implemented using categorical abstractions in an intuitive and flexible way.[1]

Keywords: Programmed graph transformation, Algebraic graph transformation, Pure functional programming, Generic graph editor

1 Introduction

Although graphs are a mathematically simple concept, the pragmatics of providing tool support for graph manipulation is surprisingly complex. The lack of general frameworks for graph manipulation tools has as a consequence that even successful tools for special-purpose graph manipulation get away with surprisingly poor user interfaces.

The more widely known graph transformation tools, such as AGG, DIAGEN, Progress, and DAVINCI (now uDraw) tend to concentrate on the particular aspects that are related to the research direction they have grown out of. This means that these tools frequently introduce limitations in other aspects that cannot be overcome by a reasonable effort on the side of the framework user, i.e., of the developer who implements a more customised application.

In our framework, we strive to implement the following requirements:

(1) Visual presentation of and interaction with graphs must be customisable to an extremely large degree, and in a "natural" way.

(2) Arbitrary graph transformation needs to be programmable.

[1] This research is supported by NSERC Canada, and Scott West worked on this while at McMaster University.

(3) High-level graph transformation approaches need to be expressible in a natural way, and in a way that results in reasonably efficient implementations.

(4) A programmer willing to invest more effort into optimisation should have the tools available to implement more efficient transformations by reverting to lower-level primitives.

The problem here is that requirement (3) implies, in particular, availability of categorical abstractions, which tend to be blissfully unaware of efficiency concerns, and is also normally understood to contradict (2), or tends to accommodate (2) in an only partial way that, however, still compromises the principled and declarative nature of the high-level approaches. That apparent conflict is best resolved by embedding the transformation capabilities into a language that provides powerful abstraction mechanisms, and allows abstraction barriers to be enforced. The latter strongly suggests the use of a pure programming language that controls side effects via its type system; together with the former, and with general availability and library support requirements, essentially only the pure functional programming language Haskell fits this bill.

However, for visual presentation of and interaction with graphs (requirement 1) there is no good functional paradigm available yet, while it is essentially *the* case study of object-oriented design [Joh92]. Therefore it is natural to satisfy requirement (1) by exposing an object-oriented interface for visual interaction purposes. This does not even contradict the choice of Haskell — Kiselyov and Lämmel have recently catalogued [KL05] a number of ways to satisfactorily implement object-oriented abstractions in Haskell. Using this, we created a framework that, in its kernel (Sections 3–5),

- encapsulates the presentational and interactive aspects of graphs and graph items (i.e., nodes and edges) in an object-oriented class hierarchy closely emulating that of [Joh92],
- but still provides a purely functional interface to item-level read-only graph access,
- and also provides a safe monadic interface to item-level graph manipulation, enabling a programmed graph transformation approach not too different from that of PROGRESS [SWZ99].

On top of this kernel, we use the abstraction mechanisms of Haskell to provide a high-level interface that includes

- an abstract datatype of partial graph homomorphisms with both algebraic and item-level access functions (Sect. 6),
- an item-level implementation of DPO transformation of node- and edge-labelled graphs using these homomorphisms, and
- basic category-theoretic constructions, and standard algebraic approaches to graph transformation (DPO and SPO) implemented on top of those (Sect. 8).

In Sect. 10, we discuss some related work.

2 A Few Quick Haskell Notes

The pure functional programming language Haskell [P+03] features a relatively lean, mathematical syntax; it uses indention to indicate components which are part of the same term, **case**

alternatives, and utilises monadic **do** statements to program "with an imperative flavour".

Function application is normally denoted by juxtaposition, as in "f x", has highest precedence, and associates to the left. To avoid parentheses, there is also a low-priority infix operator $ for function application; this allows to write e.g. "f $ g x" instead of "f (g x)". Another infix operator stands for function composition: $(f \circ g)\ x = f\ (g\ x)$.

Binary functions in Haskell can be converted to infix operators using a pair of back-ticks; for example, "x `someFunction` y" is just another notation for "someFunction x y".

Haskell features of *type classes*, which are similar to Java interfaces. In the Eq (equality) type class, instances must supply at least one of $\equiv$ and $\not\equiv$ to be override the default implementation.

$$\begin{aligned}&\textbf{class}\ \mathsf{Eq}\ a\ \textbf{where}\ (\equiv), (\not\equiv)\ ::\ a \to a \to \mathsf{Bool}\\ &\qquad\qquad a \equiv b \quad = \neg\ (a \not\equiv b)\\ &\qquad\qquad a \not\equiv b \quad = \neg\ (a \equiv b)\end{aligned}$$

Type classes may have class constraints, such as **class** Eq a $\Rightarrow$ Ord a **where**..., which specifies that instances of Ord must necessarily also be instances of Eq. The typing annotation

$$\mathsf{lookup} :: \mathsf{Eq}\ a \Rightarrow a \to [(a, b)] \to \mathsf{Maybe}\ b$$

specifies that the function lookup is to be applied to a value of some type, a, for which equality must be implemented (type class constraints appearing to the left of the $\Rightarrow$), and to a list of pairs as second argument; results are then of type Maybe b, which contains the value Nothing and the values Just y for each value $y :: b$.

Monads, which are often used to model sequential instructions in the otherwise functional language, are a very versatile concept. The related idea of monad transformers is used in this work to provide a clear definition of the sequential context in which we are operating in.

If we operate in the Maybe monad, we know that the results of the computation will either be success (Just) or failure (Nothing). To add some integer-variable lookup environment such as a Map String Integer to the Maybe monad, we could use the ReaderT monad transformer to construct a new monad instance. This new instance would have an underlying read-only state which can be accessed by using the ask function anywhere within the monad. When the computation is run, the results are still of type Maybe. Similar write-only and read-write transformers are defined as WriterT and StateT transformers — explicitly composing complex monads in this way has the advantage that imperative effects are precisely constrained by the type system.

3 Object-Oriented Abstractions for Interaction and Visualisation

The framework presented in this paper strives to make generating fully-featured graph editors very simple. The built-in features include the ability to select and drag nodes around, as well as resize the extents of the nodes. Edges are represented as Bezier curves, allowing for smooth transitions between nodes. Additionally, the view can be modified by using linear transformations such as zooming and panning.

At a low-level, these abilities are implemented in an object-oriented manner, as there are known, working solutions in the object-oriented world [Joh92]. We realist this in Haskell using a method of translating OO designs into a functional setting which is largely inspired by [KL05].

In Haskell, programming to a concept of updatable state is one instance of the use of *monads*, which allows an imperative programming style embedded into pure functional programs. Since

object-oriented designs naturally employ also the concepts of object creation, we essentially have the choice between the two predefined monads providing dynamic creation of references to updatable memory, namely ST, in which reference allocation and updates are the only possible side effects, and IO, which also allows input and output as side-effects. We implement our object-oriented data structures parametrised over such monads, will always use the type variable m for this monad parameter. As far as objects are used to support a GUI, we will be compelled to use IO for m, but the possibility of using an ST monad instead makes it feasible to create and transform graphs within pure functional computations.

A *canvas* for us is an area for graphical representation that is not unlike a magnet-board: Objects can be placed, moved, and modified in an interactive way. This is unlike a drawing-surface which allows write-only operation, i.e., where marks, once made, cannot be easily changed or deleted. The classes supporting our canvas abstraction include Figure for elements placed on the canvas, ConnectionFigures that form connections between other figures, aggregations of figures called Drawings, and DrawingView, the visualisation of the drawing as it appears on-screen.

For deep changes to the way the framework operates interactively and visually by default, appropriate subclasses to these classes can be defined. For more superficial changes, much can be accomplished through the use of node and edge labels, see Sect. 4.

Since the framework's visual and interactive components aim to be cross-platform and appealing to the eye, they use a Haskell binding to the well-known and portable GTK+ library. Graph display uses a vector graphics library associated with GTK+, the Cairo rendering engine (http://cairographics.org), which can draw on various kinds of "surfaces". Aside from the "screen" surface, one can also draw on more abstract surfaces, such as PDF and PostScript surfaces. The result is that any graph drawings produced in the editing framework can be easily reproduced in a high-quality format for use in printable documents.

4 Graph Models

In the object-oriented HotDraw-inspired design, the model component of the current state consists of the Figures contained in a Drawing, and their interconnections involving Connectors and ConnectionFigures. These can be accessed through the object-oriented method interface, which is monadic, because of the state-centered organisation of the object abstraction. Furthermore, modification through this interface can be delicate where the user is responsible for preserving invariants, and inflexible where invariants are guaranteed by encapsulation.

Due to the differences between the Figure object web structure and the conceptual graph structure it is intended to represent, programming graph transformations is therefore not well-served by the Figure-based model.

To provide more natural interfaces, we currently support generic extraction of node- and edge-labelled graphs, and re-integration of graph changes (represented as a partial graph homomorphism) into the live Figure web. Nodes are extracted from "normal" figures, and edges from connection figures.

Since this is programmed against the public interface of the object-oriented abstractions, and actually does not require very much code, more complex graph models could be realised simi-

larly, giving access to additional aspects of the HotDraw-inspired drawing abstractions, for example the presence of connectors, which could translate, for example, into "ports" of hyperedges.

To make programming the GUI aspects of node- and edge-labelled graphs easier, we provide a special LabelledFigure sub-class in the Figure hierarchy. Also, implementations both of simple labelled figures and labelled connection figures (drawn by default with the edge label close to the middle of the edge line) are provided. Labels are defined via a simple FigureLabel type class interface (type reïfication via Typeable is necessary for the class casting mechanisms):

```
class (Show label, Typeable label) ⇒ FigureLabel label where
  draw       :: label → Rect → Render ()
  size       :: label → Maybe Point
  parseLabel :: Parser label
  interfaceIO :: LabelledFigure label IO → DView IO → IO ()
```

The members of this type-class should provide the following functionality:

- draw – The visual representation of the label adapted to its physical size, specified as a rectangle. The resulting drawing operations are encapsulated in the Render monad.
- size – Specify that the label should be drawn either with variable size (Nothing), or with fixed size by producing Just pt, where pt :: Point will be interpreted as a size.
- parseLabel – The framework also handles the saving and loading of graphs in files, and uses the Parsec [LM01] parser combinator library. To allow the labels to be restored from a string, a parsing function needs to be implemented that acts as inverse to the Show instance required by the superclass constraint.
- interfaceIO – We can additionally allow interactive actions to be performed when the node carrying the label is activated (currently by double-clicking) through a GUI view. These actions could be anything including popping up a dialog-box to modify the label, or performing a graph transformation.

For example, a Petri net can be considered as a graph with trivial edge labels, and with node labels that indicate whether a node is a place (with some number of tokens) or a transition.

```
data Petri = Place Integer | Transition deriving (Show, Typeable)
```

Using about two pages of literate Haskell code, we defined a FigureLabel instance for the Petri label type introduced above, and with two additional pages defining the GUI wrapper around the drawing canvas, we created the editor and animator shown in Figure 1, where interfaceIO will fire the clicked transition if possible — this is implemented using the programmed approach described in Sect. 5.

5 Programmed Transformations

Graph transformation steps typically divide into two parts: A matching phase, which does not need more than read-only access to the graph that is to be transformed, and a modification phase. Haskell programmers will naturally expect that the type system can prevent modification steps

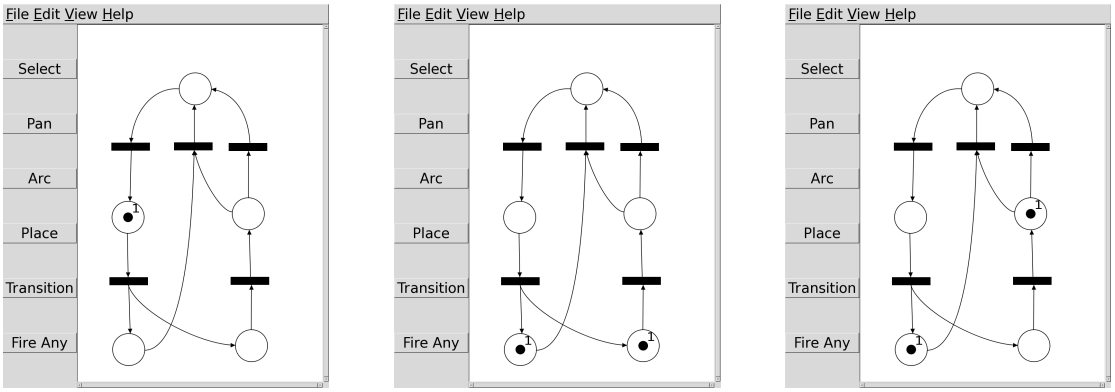


Figure 1: Petri net editor and animator, defined using labels of type Petri.

to be used during the matching phase — this makes reasoning about graph transformation programs, and activities like code refactoring much easier. Since matching is conceptually non-side-effecting, it is only natural to expect that matching can be programmed without performing monadic computations. However, one of the problems of programming to a graph view built on the object-oriented programming principles outlined above is that all graph access has to be performed as part of a monadic computation.

Therefore we start graph transformations by adding a "pure view" to the object-oriented view of a drawing representing a graph. This "pure view" allows pure functions to access the graph for matching and analysis purposes, but also connects to the object-web in a way that monadic updates can be performed on both together. For both purposes we provide low-level and higher-level interfaces, described in the remainder of this section.

5.1 Low-Level Interfaces

All the graph transformation machinery in our framework is built on top of two interfaces.

A *graph inspection* interface ReadLGraph provides *pure* (i.e., non-monadic) functions for read-only access to the graph structure, including node and edge sets, edge incidence, node and edge labels. This interface not only includes basic functions, but also more advanced facilities, as for example a function calculating strongly connected components — although such functions could be implemented on top of the basic functions, in general only custom implementations by the interface instances can achieve satisfactory performance.

A *monadic graph modification* interface MonadLGraph, containing item-level graph update functions, like addition and deletion of labelled nodes and edges, label updates, updates of source or target of edges, etc. The unusual aspect of the modification interface is the typing of the exposed functions: A pure node addition function would typically have the type[2] $\mathsf{a} \to \mathsf{g} \to (\mathsf{n}, \mathsf{g})$, meaning that given a node label and a graph, a new graph is produced that differs from the argument graph only by including a new, appropriately labelled node, which is returned together

[2] In the Haskell code fragments in the remainder of this paper, we use the type variables g for graphs, n for nodes, e for edges, a for node labels, and b for edge labels.

with the new graph. (Our inspection interface contains *no* functions with graph results.)

A typical monadic interface would instead use the type $a \to g \to m\ n$, for a monad type constructor m, i.e., the function would produce a computation of type $m\ n$, that is, a computation returning the new node, with the understanding that the graph reference passed in as argument now refers to the updated graph when it is be used in subsequent computations.

For our graph modification interface, we intentionally allow "the worst of both worlds" in order to give the implementation more liberties:

$$\mathsf{grNewLNode} :: a \to g \to m\ (n, g)$$

At first sight, this appears to imply a very awkward style of programming to this interface. However, the ordering of arguments and results has been chosen so that this function can be embedded directly into a standard-library state monad transformer [Jon95], so that we obtain $\mathsf{StateT} \circ \mathsf{grNewLNode} :: a \to \mathsf{StateT}\ g\ m\ n$. From this typing, we see that this is now a function that, given a node label, returns a computation in the "state-enriched" monad $\mathsf{StateT}\ g\ m$, which keeps a current graph (of type g) as its state while performing computations in monad m. This StateT view of the modification interface then allows a normal imperative programming style, and also forms the basis for the primitive operations of the GraTra monad, see Sect. 5.3 below.

5.2 The Select Monad

Matching, i.e., identifying redexes for graph transformation rules, has two aspects that invite application of "pre-fabricated abstractions": All matching activity happens in the context of a fixed, "current" graph, and matching steps are frequently non-deterministic, and failure requires backtracking. The first can be implemented using a Reader monad, while the second is most easily handled by a list monad; using monad transformers [Jon95, LHJ95], the two can be combined:

$$\textbf{type}\ \mathsf{Select}\ g = \mathsf{ListT}\ (\mathsf{Reader}\ g)$$

This is automatically a monad, and allows an intuitive programming style for matching purposes, as the following utility operator demonstrates: starting from a node selector $\mathsf{sel} :: \mathsf{Select}\ g\ n$, the selector $\mathsf{sel} \rightsquigarrow p$ finds nodes that sel generates whose outgoing edges target nodes which have a label satisfying the node label predicate $p :: a \to \mathsf{Bool}$.

```
(~>) :: (ReadLGraph g n e a b) => Select g n -> (a -> Bool) -> Select g (e,(n,n))
sel ~> p = do
  n1 <- sel                          -- select a first node
  e  <- sel_out1 n1                  -- select an outgoing edge
  n2 <- sel_trg e                    -- obtain target node of that edge
  sel_node_label n2 >>= guard p      -- backtrack if target node label does not satisfy p
  return (e,(n1,n2))
```

In the Petri net editor, the identification of transitions that can fire together with the preparation for firing is handled by a Select computation that, in case of success, returns the list of all predecessor places and the list of all successor places of its argument.

```
firingSel :: (ReadLGraph g n e Petri b) => n -> Select g ([n],[n])
firingSel n = do
  guard . isTrans =<< selNodeLabel n                    -- n has a transition label
```

```
srcs ← selectFromReadF (‘grPredNodes‘n)          -- srcs are the predecessors
guard ∘ all nonemptyPlace =≪ mapM selNodeLabel srcs   -- all srcs are non-empty places
trgs ← selectFromReadF (‘grSuccNodes‘n)          -- trgs are the successors
guard ∘ all isPlace =≪ mapM selNodeLabel srcs    -- all trgs are places
return (srcs, trgs)
```

5.3 The GraTra Monad

For higher-level graph modification, we have to start from the monad m required by the low-level modification interface, and we add backtracking as for Select, and an updatable state containing the current graph together with its "history" represented as a chain of graph homomorphisms. The history is necessary for cases where, for example, a lot of information is collected during matching, and then applied in separate modification steps: Nodes used in later steps may have been identified with other nodes, and taken on their identity; the history is used to implement the necessary indirection in a modular way that mixes well with backtracking. The history also allows to form an overall partial graph morphism as a result of a series of operations.

Due to the necessity of history concatenation, the standard StateT monad transformer [Jon95] is not sufficient here, so the GraTra monad is implemented directly, and exported abstractly, so that its state invariants can be guaranteed.

The GraTra monad provides primitive operations such as adding and removing graph elements, as well as updating the node labels. In addition, inside GraTra computations, Select computations can be performed by embedding them using the following function:

```
select :: (MonadLGraph m g n e a b) ⇒ Select g x → GraTra g n e m x
```

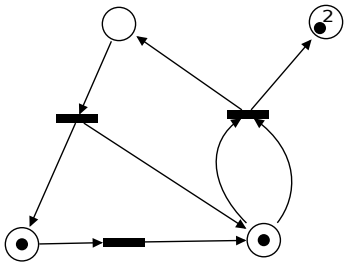
All these, together with the abstraction capabilities of Haskell, constitute a powerful toolkit to express graph transformations in a programmatic way.

In the Petri net editor, firing a transition uses the result node lists produced by firingSel above:

```
tokenTrans :: (MonadLGraph m g n e Petri b) ⇒ n → GraTra g n e m ()
tokenTrans n = do
  (srcs, trgs) ← select (firingSel n)
  mapM_ (updNodeLabel decToken) srcs
  mapM_ (updNodeLabel incToken) trgs
```

The functions used for updating the node labels can fail; which would make the GraTra computation backtrack. In the context here, since they will only be called on Places, such failure can only occur for decToken, namely if a place occurs more than once among the predecessors of the selected transition.

```
decToken, incToken :: Petri → Maybe Petri
decToken (Place 0) = Nothing
decToken (Place i) = Just $ Place (i − 1)
decToken Trans = Nothing
incToken (Place i) = Just $ Place (i + 1)
incToken Trans = Nothing
```



Therefore, although firingSel as programmed above will succeed on the rightmost transition in the net drawn to the right, tokenTrans will backtrack over that succees during the second updNodeLabel decToken on its input place. (So only the bottom transition can fire.)

6 Abstractions for Categorical Rewriting Approaches

Low-level graph transformations in a high-level programming language are of course unsatisfactory, so we provide a number of abstractions to enable programming of and with high-level graph transformations approaches.

For any graph type g implementing the ReadLGraph g n e a b interface, the data type SubGraph g n e encapsulates a graph of type g together with the node and edge sets of a subgraph (so the node set includes all nodes incident to edges in the edge set). On such SubGraphs, both item-level operations, like insertion, deletion, membership test, and lattice operations, like join (union), meet (intersection), and pseudo-complement, (the subgraph lattice of a non-discrete graph is not complemented) are available.

Similarly, a GraphMor g n e encapsulates two graphs and two finite maps representing a partial graph homomorphism between these graphs. Again, item-level operations are provided, including membership tests, and insertion and deletion of item pairs from the maps. More importantly, a categorical interface is provided, including morphism source, target, composition "⁂", identities, and also operations returning domain and range of a morphism as SubGraphs, and restricting a morphism on either side with a (compatible) SubGraph.

7 Item-level Implementation of DPO Graph Transformation

The *double-pushout approach* is the "classical" variant of the "algebraic approach" to graph rewriting, going back to [EPS73]. In this approach, a rewriting rule is a *span* $\mathscr{L} \xleftarrow{\phi_L} \mathscr{G} \xrightarrow{\phi_R} \mathscr{R}$ of morphisms, which we represent using a record datatype:

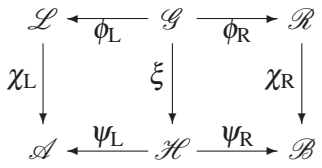
data Span g n e = Span {spanLMor, spanRMor :: GraphMor g n e}

Performing a DPO rewrite step on an application graph $\mathscr{A}$ involves

1. identifying a redex, i.e., a suitable "matching" morphism χ_L from the left-hand side $\mathscr{L}$ into the source graph,

2. completing the left square via the construction of a pushout complement, including the host morphism ξ from the gluing graph $\mathscr{G}$ to the host graph $\mathscr{H}$, and

3. completing the right square by constructing a pushout.



One of the two parts of the "gluing condition" necessary for the existence of a pushout complement is the "dangling condition", which is conventionally given as follows:

Definition 7.1 *For two graph morphisms $\phi : \mathscr{G} \to \mathscr{L}$ and $\chi : \mathscr{L} \to \mathscr{A}$, the* dangling condition *holds iff whenever an edge connects a node outside the image of χ with a node x inside the image of χ, then x has in its pre-image via χ only nodes in the image of ϕ.* □

For implementing this version of the dangling condition directly, one may pre-calculate the pre-images via χ, by calculating the converse of χ considered as a relation; with that, it should be straight-forward to see how the following code implements the above definition (a more concise danglingCond is defined below):

```
danglingCond' :: forall g n e ∘ ReadGraph g n e ⇒ GraphMor g n e → SubGraph g n e → Bool
danglingCond' chi ranPhi = let
  g          = Mor.trg chi                    -- the target of the matching
  ranChi     = SubGraph.nodeSet $ Mor.ran chi -- the node range of chi
  chiConverse :: MRel n n   -- a relation represented as set-valued map
  chiConverse = converseToMRel $ Mor.nodeMap chi
  safeNode :: n → Bool   -- holds iff x has in its χ-pre-image only nodes in the image of φ.
  safeNode x = lookupMRel x chiConverse `Set.isSubsetOf` SubGraph.nodeSet ranPhi
    -- Remember that the ordering ⩽ on Bool is implication:
  safeEdge (s,t) = if    s `Set.member`    ranChi
                   then t `Set.notMember` ranChi ⩽ safeNode s
                   else t `Set.member`    ranChi ⩽ safeNode t
  in all safeEdge $ mapMaybe (grIncidence g) (grEdges g)
```

This serves to show how categorical abstractions can lead to more concise code, but also how concepts that break the categorical abstraction (like Def. 7.1_{10}) can still be implemented in a mathematically accessible way.

Recently, there has been a tendency to consider only DPO rules where both rule morphisms are injective [CMR+97]; since this restriction simplifies the implementation considerably, we adopt it for the following example implementation.

The function dpoTrafo takes as first argument a DPO rule consisting of two total and injective graph homomorphisms, and uses this argument to pre-calculate most aspects of the reduction step. It then takes the matching $\chi_L : \mathscr{L} \to \mathscr{A}$ as second argument, and constructs the result graph $\mathscr{B}$, returning the partial homomorphism $\psi_L^\vee ; \psi_R$ and the embedding or the right-hand side $\chi_R : \mathscr{A} \to \mathscr{B}$ and $\mathscr{R} \to \mathscr{B}$:

```
dpoTrafo :: (MonadLGraph m g n e a b)
    ⇒ Span g n e → GraphMor g n e → m (GraphMor g n e, GraphMor g n e)
dpoTrafo (Span phiL phiR) chiL = let
  gR = Mor.trg phiR
  phiLRan = Mor.ran phiL
  phiRRan = Mor.ran phiL
  newNodesR = mapM (StateT ∘ Mor.copyNodeToTrg) ∘ Set.toList
              $ grNodeSet gR `Set.difference` SubGraph.nodeSet phiRRan
```

```
newEdgesR = mapM (StateT ∘ Mor.copyEdgeToTrg) ∘ Set.toList
              $ grEdgeSet gR `Set.difference` SubGraph.edgeSet phiRRan
ruleC = fromJust (Mor.converse phiR) *** phiL
deleteMatchedN (n, n') = if n `SubGraph.memberN` phiLRan
  then return ()
  else StateT (Mor.delTrgOfNode n') >> return ()
deleteMatchedE (e, e') = if e `SubGraph.memberE` phiLRan
  then return ()
  else StateT (Mor.delTrgOfEdge e') >> return ()
in do    -- now use matching χL : 𝒟 → 𝒜
  psiLC ← flip execStateT (Mor.idMor $ Mor.trg chiL) $ do
    mapM deleteMatchedN $ Mor.toListN chiL
    mapM deleteMatchedE $ Mor.toListE chiL
  let fromR = ruleC *** chiL *** psiLC    -- ℛ → ℋ
  chiR ← execStateT (newNodesR >> newEdgesR) fromR
  let gB = Mor.trg chiR    -- only added items to ℋ
  return (Mor.setTrg gB psiLC, chiR)
```

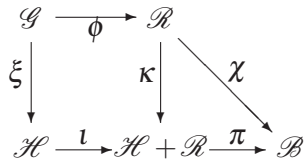
8 Categorical Implementation of DPO and SPO Graph Transformation

On top of the SubGraph and GraphMor abstractions, we also provide abstract categorical constructions. Since these involve creation of new graphs, they require access to the monadic modification interface MonadGraph. We provide, for example, production of isomorphic copies via copyGraph, disjoint union of graphs via directSum, and division modulo graph congruences via quotientProj, which is used to implement co-equalisers of total graph homomorphisms — we just list the type signatures for these:

```
copyGraph    ::                    g                    → m (GraphMor g n e)
directSum    :: g →                g                    → m (GraphMor g n e, GraphMor g n e)
quotientProj :: [[n]] → [[e]] → g                       → m (GraphMor g n e)
coEqualiser  :: GraphMor g n e → GraphMor g n e → m (GraphMor g n e)
```

It is well-known that pushouts of total graph homomorphisms can be calculated from direct sums and co-equalisers where those exist; in Haskell, the category-theoretic diagram can be almost directly transliterated for this purpose:

```
pushout :: GraphMor g n e → GraphMor g n e → m (GraphMor g n e, GraphMor g n e)
pushout xi phi = do
  (iota, kappa) ← Mor.directSum (Mor.trg xi) (Mor.trg phi)
  proj ← coEqualiser (xi *** iota) (phi *** kappa)
  return (iota *** proj, kappa *** proj)
```



We now show how to use these abstractions to program DPO and SPO graph rewriting at the level of categorical descriptions.

Kawahara has shown that the gluing condition can be formulated directly in the language of relations in the topos of graphs [Kaw90]; the following equivalent definitions [Kah01, Def. 5.4.1], all for morphisms $\phi : \mathscr{G} \leftrightarrow \mathscr{L}$ and $\chi : \mathscr{L} \leftrightarrow \mathscr{A}$, directly use the setting of relational graph homomorphisms, and can easily be implemented using our Morphism and SubGraph libraries:

- The *identification condition* holds for ϕ and χ iff χ is almost-injective besides $\mathsf{ran}\,\phi$:

  $$\chi ; \chi^{\smile} \subseteq \mathbb{I} \cup (\mathsf{ran}\,\phi) ; \chi ; \chi^{\smile} ; \mathsf{ran}\,\phi \quad .$$

  For almost-injectivity, we have a morphism predicate:

  ```
  identCond ranPhi chi = Mor.almostInjectiveBesides chi ranPhi
  ```

- The *dangling condition* holds for ϕ and χ iff $\chi ; \mathsf{ran}\,\chi^{\sim} \subseteq (\mathsf{ran}\,\phi) ; \chi$.

  The dangling condition uses semicomplement $S^{\sim}$ of a subgraph S of G. This is the least subgraph C such that $S \cup C = G$. The subgraph $S^{\sim}$ therefore overlaps with S exactly in the nodes inside S that are connected to nodes outside S.

  ```
  danglingCond ranPhi chi =
                (chi `Mor.ranRestr` SubGraph.semiComplement (Mor.ran chi))
    `Mor.leq` (ranPhi `Mor.domRestr` chi)
  gluingCond phi chi = danglingCond ranPhi chi ∧ identCond ranPhi chi
    where ranPhi = Mor.ran phi
  ```

- χ is called *conflict-free for* ϕ iff $\mathsf{ran}\,(\phi ; \chi ; \chi^{\smile}) \subseteq \mathsf{ran}\,\phi$.

  This property is important in the single-pushout approach; we can replace the occurrence of a conversion with a pre-image operator, since $\mathsf{ran}\,(\phi ; \chi ; \chi^{\smile}) = \mathsf{ran}\,(\mathsf{ran}\,(\phi ; \chi) ; \chi^{\smile})$.

  ```
  conflictFree phi chi = Mor.preImg chi (Mor.ran (phi ⁂ chi)) `SubGraph.leq` Mor.ran phi
  ```

The host graph $\mathscr{H}$ in a DPO rewriting step is always a subgraph of the application graph $\mathscr{A}$; Kawahara's construction for the DPO approach is also useful for the SPO approach. We add a variant that preserves nodes incident to dangling edges and can be used for Parisi-Presicce's "restricting derivations" [PP93]; both variants directly transliterate [Kah01, Def. 5.4.6]:

```
straightHostSG phi chi = Mor.ran chi `SubGraph.implication` Mor.ran (phi ⁂ chi)
sloppyHostSG phi chi =
  SubGraph.semiComplement (Mor.ran chi) `SubGraph.join` Mor.ran (phi ⁂ chi)
```

While SubGraph denotes a subgraph via subsets of the carrier sets of the underlying graph, the function Mor.subGraph produces an independent graph resulting from restriction to these subsets, and returns a pair consisting of a total injection morphism from the newly constructed subgraph to the original graph, and its converse, which is a partial (univalent) homomorphism, and which can be used to calculate the host morphism ξ.

```
constructHost chiL sp = do
  psi@(_, psiLC) ← Mor.subGraph $ straightHostSG phiL chiL
  return (psi, phiL ⁂ chiL ⁂ psiLC)
  where phiL = spanLMor sp
```

For reducing a DPO redex, we just need to put this together with the pushout construction for the right-hand side. We choose to return, together with the embedding of the right-hand side, the partial morphism $(\psi_L^\smile ; \psi_R) : \mathscr{A} \to \mathscr{B}$ along the bottom of the DPO diagram; one could of course also choose to return the constituant morphisms, or even the whole diagram.

```
reduceDPOredex rule chiL = do
  ((_psiL, psiLconv), xi) ← constructHost chiL rule
  (psiR, chiR) ← pushout xi $ spanRMor rule
  return (psiLconv ⁂ psiR, chiR)
```

This redex reduction is an essentially deterministic computation (in the monad m for which no backtracking capabilities are assumed). A DPO rule induces a non-deterministic reduction relation by permitting arbitrary redexes; we implement this as a backtracking computation into which we lift the redex reduction:

```
applyDPO rule = do
  let phiL = spanLMor rule
  chiL ← select ∘ matchSel $ Mor.trg phiL      -- backtrack over possible matchings
  guard $ gluingCond phiL chiL                 -- prunes illegal redexes
  (m, chiR) ← lift $ reduceDPOredex rule chiL  -- DPO construction
  doMorphism m                                 -- updates "current" graph
  return chiR                                  -- return RHS embedding
```

For many applications, matchSel will not be an appropriate choice for determining redexes, either for efficiency reasons, or because of the presence of some strategy. The function applyDPO is therefore only an example how our building blocks can be assembled at a high level to easily produce reasonable implementations of high-level graph transformation approaches.

For a further example, we re-use some of the material above to implement the single-pushout approach. A partial morphism is easily converted into a span of total morphisms:

```
spanFromPMor mor = do
  inj ← Mor.subGraphInj $ Mor.dom mor
  return $ Span inj (inj ⁂ mor)
```

With that, we can use [Kah01, Thm. 5.4.11] (which is a generalised version of the result by Löwe [Löw90, Cor. 3.18.5] that single-pushout squares for conflict-free matchings have total embeddings of the right-hand side) to employ reduceDPOredex to implement single-pushout rewriting for conflict-free matchings.

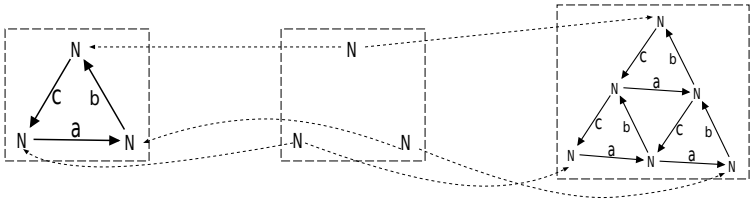
```
applySPO rule = do
  chiL ← select ∘ matchSel $ Mor.src rule
  guard $ conflictFree rule chiL
  (m, xi) ← lift $ spanFromPMor rule >>= flip reduceDPOredex chiL
  doMorphism m
  return xi
```

It is important to note that these functions implement DPO and SPO graph transformation via the purely categorical (and relation-categorical) parts of the interface of Mor and SubGraph. Therefore, these formulations are completely independent of the underlying graph category, as long as it allows implementations of the interfaces used here. Importing corresponding different

modules under the names Mor and SubGraph then makes the DPO and SPO implementations provided in this section available for that new category without requiring further changes, unlike the implementation in Sect. 7 which is glued to the category of node- and edge-labelled graphs.

9 Performance Comparison

Although performance has not been a first-class citizen in the design of the framework, we provide a small case-study based on the Sierpinski problem from the 2007 AGTIVE tool contest [TBB+08], which requires that the tool generate iterations of approximations to Sierpinski triangles. The complexity of this task generally grows exponentially as the iterations progress. We measured the transformation time required by our tool for three different implementations of double-pushout rewriting, applied in parallel applications of the following rule:



Since the paper [TBB+08] only contains a performance graph, we compare our implementations with numbers found in on-line documents for the participating tools MOMENT2-GT [BH07], Tiger EMF [TB07], and GrGen.NET [KM07]. For our own measurements we used a 2.5GHz PowerMac G5 with 5GB RAM running Linux, which is at least not more recent than the different machines used in the cited documents.

Iter.	**Categoric**	MOMENT2	**Full Item**	Tiger EMF	**Fast Item**	GrGen
1	3.6	42.0	2.2	16	1.3	12
2	6.8	144.9	1.3	0	1.2	14
3	50.6	453.3	3.9	0	3.0	13
4	341.0	1,475.7	20.4	16	7.1	14
5	4,018.1	4,928.7	178.6	62	18.6	14
6	47,987.4	22,284.0	2,030.7	250	59.5	18
7	644,468.3	172,868.0	39,324.6	2,000	199.3	27
8		1,511,668.0	603,506.2	16,500	658.4	77
9				227,250	2,092.4	206
10				3,604,984	6,988.7	749
11					25,232.7	1,930
12					104,382.9	5,876
13					596,509.1	20,872
14						49,919

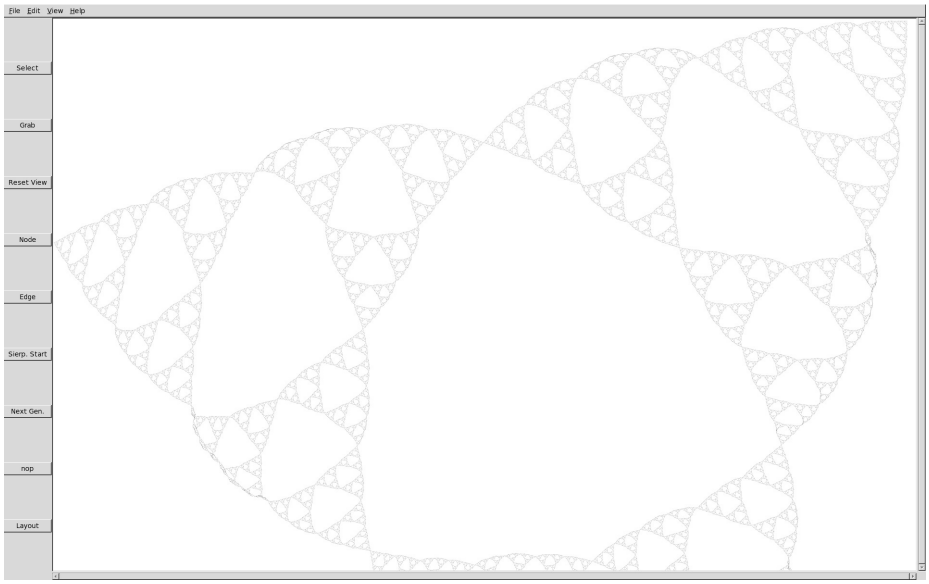
Table 1: Comparison of iteration timings (in ms)

Our “categoric” imlementation using reduceDPOredex from Sect. 8 is spectacularly slow due to the fully general pushout construction implemented by a quotient (co-equaliser) of a direct sum,

which is applied for every individual rule application, so that most of the time is spent in the quotient construction.

Our "full item-level implementation" dpoTrans of DPO graph transformation from Sect. 7 performs comparable to the single-pushout implementation of MOMENT2-GT. Profiling shows that much effort is expanded in the generation of the transformation morphisms $\widetilde{\psi_L}; \psi_R : \mathscr{A} \to \mathscr{B}$ and their accumulating composition.

A variant dpoTrans′ that does not generate and return that transformation morphism underlies our "fast item-level implementation", which performs between the Tiger EMF tool, which offers a high-level modeling environment with a focus on making graph tools, and the GrGen.NET implementation, which is focused on purely rewriting, and less on visualization and easy tool creation. GrGen.NET achieves its performance by compiling the rules into .NET assemblies.



10 Related Work

The DiaGen diagram editor generation tool [MK00, Min02] aims in a similar direction as our framework. The similarities include the idea of combining graph transformation and visualisation into a single framework, to generate interactive editors. However, the DiaGen system uses a different fundamental transformation system, namely hypergraph grammars (for syntax definition) and hypergraph transformation rules for the realization of editing operations. Our approach is not limited in this way, offering a programmed approach that can be expanded upon to utilise DPO and SPO if required. Also, our approach does not require the code generation step that DiaGen does, which the developer then is expected to integrate with their own project.

Another influential graph rewriting system is the PROgrammed Graph REwriting SyStem (PROGRESS) [SWZ99]. PROGRESS combines textual and graphical representations to specify graph productions, tests and paths. Simple productions can be used to construct more complicated programmed transformations by using the PROGRESS control structures. The organisation of PROGRESS is similar to our approach, including the ability to backtrack programmed

transformations in the case of failure. However, the inclusion of a graphical language to specify graph transformations is one large difference. Also, PROGRESS seems to be less concerned with user-interaction, customisability, and algebraic approaches.

In the same spirit as our framework, the AToM3 tool [LVM03] offers an environment to create interactive editors for graphs. AToM3 is able to generate Python scripts which can then be loaded back into AToM3 to offer customised model creation. For example, the base-tool offers a Petri net model that can be loaded at runtime to start editing Petri nets. The AToM3 approach differs from ours in a few fundamental ways. Firstly, the visualisations are markedly more primitive, not offering any compositing of graphical primitives, nor any high-quality rendering options like anti-aliasing and PDF and PostScript export. We do not yet match the export facilities of AToM3 to GXL and GML, but these would be easy to add even for a user of our framework.

Tiger [EEHT04] is an ambitious project offering full graphical descriptions of editors. It is built on top of the Eclipse IDE framework, extended by Eclipse GEF (Graphical Editing Framework) as well as the EMF (Eclipse Modeling Framework); graph transformation facilities are handled by the AGG-engine. The heavy usage of the Eclipse framework leads to some UI confusion, as the custom editors still sport many of the default Eclipse menus and interface elements, leading to an arguably less focused user experience. Transformation rules and the appearance of nodes and edges are customisable visually.

On the Haskell side, the most notable graph abstractions are the inductive graph interfaces of Erwig's Functional Graph Library FGL [Erw97]. These interfaces allow the decomposition of non-empty graphs by separating them from the remaining graph the context (label, and incoming and outgoing edges) for either a given or an arbitrary node. Since edges are simply triples of source node, target node, and edge label, these interfaces do not directly support the concept of edge identity essential for most categorical approaches. Our implementation actually uses FGL for the "pure view" explained in Sect. 5, and therefore has to embed edge identity information in the FGL edge labels.

The depth-first search tree abstraction of King and Launchbury [KL95] is entirely geared towards graph analysis, where it provides a collection of useful tools. It provides an array-based representation for unlabelled graphs, and includes no facilities for graph modification.

Another specialised approach to declarative high-level specification of graph interaction is embodied by Mazanek and Minas' combinator library for graph parsers (for a simple list-based graph datatype) in the functional-logic programming language Curry (closely related to Haskell) [MM08].

11 Conclusion

We have outlined a framework that allows developers to easily create interactive graph editors that offer polished user interfaces and include powerful graph transformation capabilities.

Graph transformations can be expressed in different ways, with primary support offered for a programmed approach to graph transformation, and derived implementations of the categorical DPO and SPO approaches.

In addition to the graph transformation abilities, the framework has progressed far enough that the direct interaction mechanism works in an reasonably intuitive manner. Several small editors

have been constructed for models including Petri nets and Hasse diagrams besides the string-labelled graphs used also for the Sierpinski example, and some preliminary work has started on a code-graph editor as well. The project is still in an early stage, but already offers interactive resizing and positioning of canvas items, diagrams can be exported to PDF and PostScript, some rudimentary layout facilities are offered by way of GraphViz, with all of these features available to any graphs which the framework produces.

The ease with which the high-level approaches are implemented on top of the low-level programmed approach owes much to the power of the abstraction mechanisms provided by Haskell. The result is a unique environment both for experimentation with novel approaches to graph transformation and interaction, and for easy creation of polished high-quality graph manipulation applications.

Bibliography

[BH07] A. Boronat, R. Heckel. Sierpinski Triangles in MOMENT2-GT. 2007. http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/, last visited 30 April 2009.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. Chapter 3, pp. 163–245. World Scientific, Singapore, 1997.

[EEHT04] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Towards Graph Transformation based Generation of Visual Editors using Eclipse. In *Visual Languages and Formal Methods, Elec. Notes Theor. Comp. Sci. vol. 127*. Pp. 127–143. Elsevier, 2004.

[EPS73] H. Ehrig, M. Pfender, H. J. Schneider. Graph Grammars: An Algebraic Approach. In *Proc. IEEE Conf. on Automata and Switching Theory, SWAT '73*. Pp. 167–180. 1973.

[Erw97] M. Erwig. Functional Programming with Graphs. In *ICFP '97*. Pp. 52–65. ACM, 1997. See also http://web.engr.oregonstate.edu/~erwig/fgl/.

[Joh92] R. E. Johnson. Documenting Frameworks Using Patterns. In *OOPSLA '92*. Pp. 63–76. ACM, 1992.

[Jon95] M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Jeuring and Meijer (eds.), *Advanced Functional Programming*. LNCS 925, pp. 97–136. Springer, 1995.

[Kah01] W. Kahl. A Relation-Algebraic Approach to Graph Structure Transformation. 2001. Habil. Thesis, Fakultät für Informatik, Univ. der Bundeswehr München, Techn. Report 2002-03, http://sqrl.mcmaster.ca/~kahl/Publications/RelRew/.

[Kaw90] Y. Kawahara. Pushout-Complements and Basic Concepts of Grammars in Toposes. *Theoretical Computer Science* 77:267–289, 1990.

[KL95] D. J. King, J. Launchbury. Structuring Depth-First Search Algorithms in Haskell. Pp. 344–354 in [POP95].

[KL05] O. Kiselyov, R. Lämmel. Haskell's overlooked object system. Draft, submitted for journal publication. Online since 30 Sept. 2004; full version released 10 Sept. 2005, http://homepages.cwi.nl/~ralf/OOHaskell/, 2005. (last accessed 19 Dec. 2008).

[KM07] M. Kroll, C. H. Mallon. A finite taste of infinity: A GrGen.NET solution of the Sierpinski triangle case for the AGTIVE 2007 Tool Contest. 2007. http://gtcases.cs.utwente.nl/wiki/uploads/sierpinskigrgennet.pdf, last visited 30 April 2009.

[LHJ95] S. Liang, P. Hudak, M. Jones. Monad Transformers and Modular Interpreters. Pp. 333–343 in [POP95].

[LM01] D. Leijen, E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001. See also: http://www.cs.uu.nl/~daan/parsec.html.

[Löw90] M. Löwe. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. Technical report 90/05, TU Berlin, 1990.

[LVM03] J. de Lara Jaramillo, H. Vangheluwe, M. A. Moreno. Using Meta-Modelling and Graph Grammars to Create Modelling Environments. *Electr. Notes Theor. Comput. Sci.* 72(3), 2003.

[Min02] M. Minas. Specifying graph-like diagrams with DiaGen. In *Proc. International Workshop on Graph-Based Tools (GraBaTs'02)*. ENTCS 72(2), pp. 16–25. 2002.

[MK00] M. Minas, O. Köth. Generating Diagram Editors with DiaGen. In Nagl et al. (eds.), *Applications of Graph Transformations with Industrial Relevance, Proc. AGTIVE'99, Kerkrade, The Netherlands, Spt. 1999*. LNCS 1779, pp. 443–440. Springer, 2000.

[MM08] S. Mazanek, M. Minas. Functional-Logic Graph Parser Combinators. In Voronkov (ed.), *Rewriting Techniques and Applications, RTA 2008*. LNCS 5117, pp. 261–275. Springer, 2008.
http://www.springerlink.com/content/w568v7647066/

[P+03] S. Peyton Jones et al. *The Revised Haskell 98 Report*. Cambridge Univ. Press, 2003. Also on http://haskell.org/.

[PP93] F. Parisi-Presicce. Single vs. double pushout derivation of graphs. In Mayr (ed.), *Graph Theoretic Concepts in Computer Science, WG '92*. LNCS 657, pp. 248–262. Springer, 1993.

[POP95] *POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. acm press, 1995.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. Chapter 13, pp. 487–550. World Scientific, Singapore, 1999.

[TB07] G. Taentzer, E. Biermann. Generating Sierpinski Triangles by the Tiger EMF Transformation Framework. 2007. http://tfs.cs.tu-berlin.de/publikationen/Papers07/TB07.pdf, last visited 30 April 2009.

[TBB+08] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, Á. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, T. Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In Schürr et al. (eds.), *Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007*. LNCS 5088, pp. 514–539. 2008.
doi:10.1007/978-3-540-89020-1
http://www.springerlink.com/content/f65r3041v3015222/