Graph Rewrite Rules with Structural Recursion

Berthold Hoffmann[1], Edgar Jakumeit[2], and Rubino Geiß[2 3]

[1] Universität Bremen and DFKI-Lab Bremen, Germany
[2] Universität Karlsruhe (TH), Germany
[3] LPA GmbH Frankfurt/Main, Germany

Abstract. Graph rewrite rules, programmed by sequencing and iteration, suffice to define the computable functions on graphs—in theory. In practice however, the control program may become hard to formulate, hard to understand, and even harder to verify. Therefore, we have extended graph rewrite rules by variables that are instantiated by a kind of hyperedge replacement, before the so instantiated rules are applied to a graph. This way, rules can be defined recursively over the structure of the graphs where they apply, in a fully declarative way. Generic rules with variables and recursive rule instantiation have been implemented in the graph rewrite tool GrGen.

1 Introduction

Graph rewriting is a basis for rule-based ("declarative") programming with graphs, in the same way as term rewriting is a basis of functional programming—another rule-based paradigm. The following example from biology illustrates essential concepts of functional programming. We take this as a starting point for discussing concepts that would be useful for rule-based programming with graphs, and shall come back to it later.

Example 1 (Transcribing DNA to RNA). The genetic information of DNA is coded in four nucleic bases guanine (G), cytosine (C), adenine (A), and thymine (T), where uracil (U) replaces thymine in RNA. These bases form pairs G–C and A–T/U. A transcription of DNA to RNA starts after a sequence "TATAAA" on the DNA strand, and builds an RNA strand with complementary bases, until the termination sequence "CCCACT…AGTGGGAAAAAA" is found (where "…" stands for six arbitrary bases).

The following Haskell function defines transcription on strings representing the base sequences.

```
transcription  ds
  | length  ds  < 30 = []
  | isTATAAA ds = d2rna ((drop 6) ds)
  | otherwise  =  transcription  ( tail  ds) where
                  d2rna ds  |  length  ds  < 24 = error "unterminated␣gene"
                            |  isCCCACTuvwxyzAGTGGGAAAAAA ds = []
                            |  otherwise  = d2r (head ds)  :  d2rna  ( tail  ds)
                  d2r  'A' = 'U';  d2r  'C' = 'G';  d2r  'T' = 'A';  d2r  'G' = 'C';
```

(The omitted functions "isTATAAA" and "isCCC. . . AAA" test whether their arguments begin with the corresponding bases, and (drop *i*) removes *i* leading elements from a list.)

A rule-based functional language offers the following concepts:

1. A function may be defined with several rules that use *pattern matching* and *application conditions* for case distinction.
2. Patterns may contain *variables* like *ds*, which are placeholders for values with a specific, possibly recursive *structure*—character lists in this case.
3. Functions are defined by *recursion over the structure* of values. In our example, d2rna calls d2r on the head, and itself recursively on the tail of its argument.

Graph rewrite rules do certainly provide pattern matching, and may also support application conditions. However, in contrast to term rewriting, graph rewrite rules do not support variables that are placeholders for graphs of a specific structure. In most cases, structural recursion is not supported either. Instead, several graph rewrite tools feature constructs for choosing a rule from a set, sequential composition, and iteration. This suffices to define all computable functions on graphs [12]. However, are these concepts adequate from a programmer's point of view? They do suggest a style of programming that is imperative rather than declarative. More importantly, they certainly allow to control *which* rule shall be applied next, but provide only little help to control the places *where* it shall be applied.

Considering these deficiencies, we have extended the graph-rewrite tool GrGen [2] by generic rules with variables, where structure rules define the graphs that may be substituted for variables. Several structure rules may define alternative substitutions of a variable, and the substitutions may contain variables again, also recursively. So, a generic rule is instantiated recursively over a graph structure before it is applied. Variables may be placeholders for *sub-patterns* of a generic rule, like *ds* is a placeholder for string values. However, they may as well denote a *sub-rule*, like d2rna and d2r denote auxiliary functions in the example above. This concept shall improve the support for a declarative style of programming with graphs.

The paper is structured as follows. In the next section, the concepts of single-pushout (SPO) rewriting with negative application conditions are recalled. In Section 3, controlled graph rewriting is discussed. The limitations of these control programs have motivated our extension of rules by variables that are substituted according to recursive structure rules, which is described in Section 4. Finally, some related and future work is outlined in Section 5.

2 Graph Rewriting

In this section, we review the major notions of graphs and rules implemented in the graph rewrite generator GrGen which is fully documented in [2] and [7].

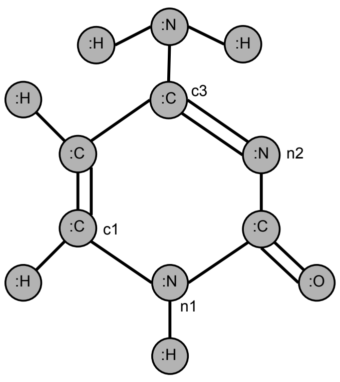
Graphs. The graphs used in GrGen are directed and allow loops and multiple edges from one node to another one. Their nodes and edges are labeled (typed). Undirected edges are supported too, but for conciseness we want to view them as a shorthand notation for pairs of undirected counter-parallel edges in this paper. A fixed pair $T = (\dot{T}, \bar{T})$ of disjoint finite sets provides *types* for nodes and edges. A (typed) graph $G = (\dot{G}, \bar{G}, \mathit{src}_G, \mathit{tgt}_G, \dot{\tau}_G, \bar{\tau}_G)$ consists of disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *edges*, with mappings $\mathit{src}_G, \mathit{tgt}_G \colon \bar{G} \to \dot{G}$ that associate a *source* and a *target* node to its edges, and *type mappings* $\dot{\tau}_G \colon \dot{G} \to \dot{T}$ and $\bar{\tau}_G \colon \bar{G} \to \bar{T}$. We often write "$x \in G$" instead of "$x \in \dot{G}$ or $x \in \bar{G}$" and call x an *item* of G.

Let G and H be graphs. A pair $m = (\dot{m}, \bar{m})$ of functions $\dot{m} \colon \dot{G} \to \dot{H}$ and $\bar{m} \colon \bar{G} \to \bar{H}$ is a *graph morphism* (or just *morphism*, for short) if it preserves sources, targets, and types, i.e., if $\mathit{src}_H \circ \bar{m} = \dot{m} \circ \mathit{src}_G$, $\mathit{tgt}_H \circ \bar{m} = \dot{m} \circ \mathit{tgt}_G$, $\dot{\tau}_G = \dot{\tau}_H \circ \dot{m}$, and $\bar{\tau}_G = \bar{\tau}_H \circ \bar{m}$. Then m is denoted as $m \colon G \to H$, and called *injective* (*surjective* resp.) if its component mappings have this property. If m is injective and surjective, G and H are *isomorphic*, denoted as $G \cong H$.

We say that a graph G is a *subgraph* of a graph H, and write $G \subseteq H$, if the nodes and edges of G are subsets of those of H, and the mappings of G are restrictions of the respective mappings of H to $\bar{G}$ and $\dot{G}$.

Let G be a graph with a subgraph $D \subseteq G$. A morphism $m \colon D \to H$ is called a *partial morphism* from G to H, written $m \colon G \dashrightarrow H$, and D is called the *domain* of m, denoted by $\mathit{Dom}(m)$. The partial morphism m is *total* if $\mathit{Dom}(m) = G$.

Example 2 (Graphs). In Figure 1, the molecular structure of the nucleic base cytosine is specified in GrGen (on the right-hand side), and as a diagram (on the left-hand side). The GrGen program model on top declares four node types representing atoms, which are extensions of the predefined node type `Node`. Undirected edges of the predefined type `UEdge` represent chemical bonds. In the graph



```
node class C; // Program model
node class H;
node class N;
node class O;
```

```
test cytosine {
  n1:N -- c1:C -- c2:C;
  c2 -- c3:C -- n2:N -- c4:C -- n1;
  c2 -- c1; n2 -- c3;
  n1 -- :H; c1 -- :H; c2 -- :H;
  c3 -- n3:N -- :H; n3 -- :H;
  c4 -- o:O -- c4;
}
```

Fig. 1. The molecular structure of cytosine

specification[4] below, items are introduced with $x : t$, where x is an optional item identifier, and t its type; items may be reused with their name. An undirected edge e with source x and target y is introduced by "x `-` e `-` y", and "`--`" introduces an anonymous edge of type `UEdge`.

In diagrams of graphs, nodes are depicted as circles, and edges are drawn as arrows from their source to their target nodes, undirected edges without tips. The type will be inscribed to the circle of a node, and ascribed to the arrow of an edge. Sometimes, node identifiers are ascribed to their circles.

Rewriting. GrGen is based on rewrite rules according to the single-pushout approach (SPO for short, see [15] for details) that may have negative application conditions as defined in [10].

A *graph rewrite rule* (*rule*, for short) is an injective partial morphism $r\colon P \dashrightarrow R$. A *conditional rule* is a pair (C, r) with r as above, and a set $C = \{c_1, \ldots, c_k\}$ of injective morphisms $c_i\colon P \to \tilde{P}_i$ (with $1 \leqslant i \leqslant k$). The graphs $\tilde{P}_i$ are *negative patterns*, P is the *pattern*, and R is the *replacement* of (C, r).

An injective total morphism $m\colon P \to G$ is a *match* of a conditional rule (C, r) as above if for all $c\colon P \to \tilde{P}$ in C there is no total injective morphism $\tilde{m}\colon \tilde{P} \to G$ so that $\tilde{m} \circ c = m$. A *rewrite step* of G using (C, r) via a match m yields a graph G' that is defined as a pushout, and can be constructed from the disjoint union of G and R by *(i) identifying*, for all $x \in \mathit{Dom}(r)$, the items $r(x)$ and $m(x)$, and *(ii) deleting*, for every $x \in P \setminus \mathit{Dom}(r)$, the item $m(x)$, including all edges of G that are incident with $m(x)$ if x is a node.

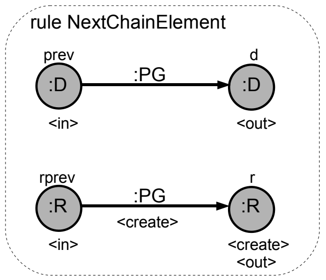
Such a step is denoted as $G \Rightarrow_{m,C,r} G'$. For a finite or infinite set $\mathcal{R}$ of conditional rules, we write $G \Rightarrow_{\mathcal{R}} G'$ if $G \Rightarrow_{m,C,r} G'$ for some match m and some $(C, r) \in \mathcal{R}$. As usual, $\Rightarrow^*_{\mathcal{R}}$ shall denote the reflexive-transitive closure of $\Rightarrow_{\mathcal{R}}$.

The default way of rewriting in GrGen is via injective matches, but a specification `hom(`x`,`y`)` allows that the items x and y in P are identified by a match. This can be modeled by extending the rule set $\mathcal{R}$ by a variant of the rule wherein x and y are identical. However, a potential match $\tilde{m}(\tilde{P})$ of a negative pattern may always overlap with the match $m(P)$ of the pattern in an arbitrary way.

Rules as Graphs. Since rules shall be instantiated by applying other rules to them (in the next section), it is important to note that a conditional rule can be represented as a single graph. The *rule graph* $\langle C, r\rangle$ of a conditional rule (C, r) is obtained from the disjoint union of its graph components $P \uplus R \uplus \biguplus_{(c\colon P \to \tilde{P}) \in C} \tilde{P}$ by identifying, for every $x \in \mathit{Dom}(r)$, x with $r(x)$, and for every $c\colon P \to \tilde{P} \in C$ and every $y \in \mathit{Dom}(c)$, y with $c(y)$.

Example 3 (A Rule Graph). The rule in Figure 2 extends a ribose chain in correspondence to a deoxyribose chain. Here and in the following examples, DNA and RNA are represented by chains, with nodes (of type `D` for deoxyribose and `R` for

[4]Actually, this is a test for the existence of a cytosine molecule in a graph.



```
*rule* NextChainElem(prev:D, rprev:R):(D,R)
{
  prev -:PG-> d:D;

  *modify* {
    rprev -:PG-> r:R;
    *return*(d,r);
  }
}
```

Fig. 2. Rule `NextChainElem` extending a ribose chain

ribose) representing the sugars, and edges of type `PG` representing the phosphate groups linking them. Nodes labeled `A`, `C`, `G`, `T`, and `U` that are connected to the sugars represent the nucleic bases.

The textual notation of the rule in the specification language of GrGen is shown on the right-hand side. The rule has a name (`NextChainElement`), two node parameters (`prev`, `rprev`) and two result nodes (`d`, `r`). Parameters may be used in the body, and results are indicated by ***return***. The outer block defines the pattern of the rule (in line 3); it contains a ***modify***-block that specifies how items shall be added to the pattern (in line 6). A ***delete***-list could indicate nodes and items to be removed from the pattern; this is not used in our example. One or more ***negative***-blocks could define negative application conditions (as in rule `DNAchain` of Example 5).

The rule graph of `NextChainElement` is shown on the left-hand side of the figure. The rule name appears at the top of the graph, its parameters are annotated with $\langle in \rangle$, and its results with $\langle out \rangle$. Items in R that are not in $r(P)$ are annotated with $\langle create \rangle$, whereas items in P that are not in $Dom(r)$ would be annotated with $\langle delete \rangle$, and items of the negative application condition would be crossed out.

3 Controlled Graph Rewriting

Controlled rewriting is typically expressed by operations that combine single rule applications. As an example, we summarize the (*graph*) *rewrite sequences* offered by GrGen.

– A rule application $(y_1, \dots y_k) = r(x_1, \dots, x_m)$ attempts to extend the matches of its parameters $x_1, \dots, x_m$ to an arbitrary match of its pattern so that the rule can be applied. If this is possible, the application *succeeds*, and defines the variables $y_1, \dots y_k$; otherwise it *fails*. A test is handled the same way, but does not modify the graph.

- For rewrite sequences S_1, S_2, the logical operations conjunction $S_1 \,\&\&\, S_2$ and disjunction $S_1 \,||\, S_2$ are evaluated lazily from left to right: S_2 is not evaluated if the success or failure of S_1 does already determine the result of the operation. Their strict counterparts &, |, and the negation ! exist as well.
- Iteration is supported by the constructs S^* and S^+ which evaluate a rewrite sequence S until it fails. S^* never fails, and S^+ is equivalent to $S \,\&\&\, S^*$.
- Transactional brackets $\langle S \rangle$ undo all effects of intermediate evaluation steps in a rewrite sequence S if the evaluation of S as a whole fails. Backtracking however – in the sense of exploring all possible rewrite sequence applications automatically – is not supported; this yields high efficiency in many cases, but complicates handling of recursive structures, as it is not possible to simply restart a stuck sequence at the last decision point.

A. Habel and D. Plump have shown in [12] that rewrite programs supporting *(i)* choice of one rule from a set of (DPO) graph rewrite rules, *(ii)* sequential composition, and *(iii)* exhaustive application suffice to define every computable function on graphs. However, this does not mean that this kind of control supports practical programming in an optimal way.

Example 4 (A Graph Rewrite Sequence for DNA Transcription). The following graph rewrite sequence performs DNA-to-RNA transcription like the HASKELL function in Example 1.

```
< (prev,rprev) =  findTATAAA
 && ( !isCCCACTuvwxyzAGTGGGAAAAA(prev)
     && (prev,rprev)=NextChainElement(prev,rprev)
     && (A(prev,rprev) || C(prev,rprev) || G(prev,rprev) || T(prev,rprev))
    )*
 &&  isCCCACTuvwxyzAGTGGGAAAAA(prev) >
```

The rule `findTATAAA` searches for the transcription starting sequence, the rule `NextChainElement` known from Example 3 extends the ribose chain to the rear, and the rules `A`, `C`, `G`, and `T` construct the nucleic base pair for the RNA chain that is complementary to the nucleic base in the DNA chain. (Their rules are similar to the alternatives of the pattern `DNANucleotide` shown in Example 5 further below.)

It is remarkable that the rewrite rules themselves perform rather trivial tasks (finding a subsequence, duplicating a chain element, attaching a node), whereas the controlling rewrite sequence that combines them is rather complex, even for such a small example. For efficient rewriting it is important to pass nodes matched in one rule to another one. Then we cannot only control *which* rule is to be applied next, but may also indicate *where* it shall be applied. Rewrite sequences can achieve this only for linear structures like lists, but for non-linear recursive structures like trees, parameter passing cannot be handled without general recursion in rewrite programs. The concepts devised for overcoming the limitations of rewrite programs are described in the next section.

4 Generic Rules

In his master thesis [14], E. Jakumeit has designed and implemented rules with structural recursion. Extending the rules strengthens the rule-based kernel of GrGen, rather than the rewrite sequences defined on top of it. The basic idea is that a generic rule contains variables, nonterminal nodes which are attached to a fixed number of terminal nodes. A set of structure rules describes how variables can be substituted. A variable may have several substitutions, which can be used alternatively. These substitutions may again contain variables, even in a recursive way. If the variable occurs in a pattern (positive or negative) of the generic rule, its instantiation yields a sub-pattern. However, since generic rules are represented as rule graphs, a variable may be attached to its (positive) pattern and replacement at the same time. Then its instantiation yields a sub-rule. Both sub-patterns and sub-rules are defined by structural recursion.

Formally, the semantics of generic rules is defined by a two-level graph rewrite process. First, all variables in a generic rule are instantiated according to the structure rules, by a context-free way of graph rewriting similar to hyperedge replacement [9]. This process yields a language of simple rules that may be infinite. Then, the host graph is rewritten with the resulting simple rules.

Assumption (Nonterminal Types). We assume that the type alphabets $T = (\dot{T}, \bar{T})$ contain a subset $N \subseteq \dot{T}$ of *nonterminals*, which are equipped with an *arity function* $\mathcal{A}\colon N \to \wp(\bar{T} \times (\dot{T} \setminus N))$.

For all graphs G occurring in the following, we assume that nonterminals are used according to their arity: Whenever G contains a node x with $\dot{\tau}_G(x) = n \in N$, G shall contain, for every $(\bar{t}, \dot{t}) \in \mathcal{A}(n)$, exactly one edge e and node y with $src_G(e) = x$ and $tgt_G(e) = y$ so that $\bar{\tau}_G(e) = \bar{t}$ and $\dot{t} = \dot{\tau}_G(y)$.

Nonterminals will occur only during instantiation, as types of variables in generic rules or in structure rules, but neither in the host graphs, nor in the simple rules applied to them. The remaining types, $(\dot{T} \setminus N) \cup \bar{T}$, are called *terminal*, as well as rules and graphs over these types.

Variables. A node x with type $n \in N$ is called a *variable*. A variable x is called *straight* if it has as many incident edges as adjacent nodes. A subgraph S induced by the incident edges of a variable is called a *star*, and its edges are called rays, and drawn like that (see Figure 3).

Structure Rules. A rule $s = S \dashrightarrow \langle r \rangle$ is a *structure rule* if its pattern S is a straight star, $\langle r \rangle$ is the graph of an unconditional rule $r\colon P \dashrightarrow R$, and $Dom(s)$ is the discrete subgraph that contains all terminal nodes of S. With S_p we denote the maximal subgraph of $Dom(s)$ so that its image $s(S_p)$ is in the pattern P of the rule graph $\langle r \rangle$. A structure rule $s = S \dashrightarrow \langle r \rangle$ is a *sub-pattern structure rule* if the morphism $r\colon P \dashrightarrow R$ is total and surjective, otherwise, it is a *sub-rule structure rule.*

Instantiation of Generic Rules. A conditional rule (C, r) with $r\colon P \dashrightarrow R$ is called *generic* if every variable y in R has a variable x in P with $r(x) = y$.

Let $T = \langle C, r\rangle$ be the graph of a generic rule and consider a structure rule $s = S \dashrightarrow \langle \hat{r}\rangle$. A total morphism $m\colon S \to T$ is a *rule match* if $m(S_p)$ is a subgraph of the pattern of the rule graph $\langle C, r\rangle$, or, if s is actually a sub-pattern structure rule (and $S_p = S$), if $m(S)$ either lies completely in a negative pattern $\tilde{P}$ (where $c\colon P \to \tilde{P} \in C$), or in the pattern of $\langle C, r\rangle$. Then $T \Rightarrow_{m,\emptyset,s} T'$ is an *instantiation step*, where the transformed graph is a rule graph $T' = \langle C', r'\rangle$ again, whose negative patterns, pattern and replacement can be distinguished by considering the pushouts for S_p and $S \setminus S_p$ separately.

Let $\mathcal{S}$ be a finite set of structure rules, and define $\Rightarrow_{\mathcal{S}}$ to be its instantiation relation. Then $\mathcal{S}$ derives, for some set $\mathcal{R}$ of generic conditional rules, the set of simple conditional rules

$$\mathcal{S}(\mathcal{R}) = \{\langle C', r'\rangle \mid \langle C, r\rangle \in \mathcal{R}, \langle C, r\rangle \Rightarrow^*_{\mathcal{S}} \langle C', r'\rangle,\ \text{where } \langle C', r'\rangle \text{ is terminal}\}$$

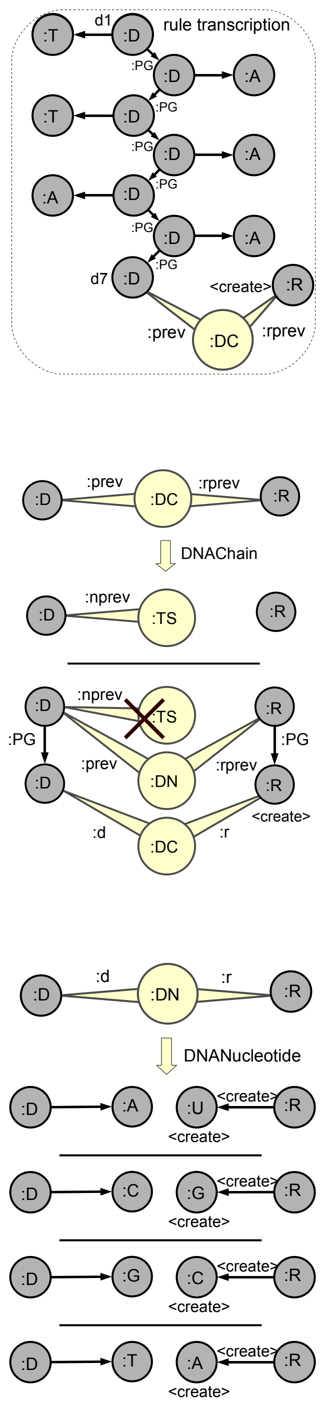
The rewrite relation of generic rules $\mathcal{R}$ over structure rules $\mathcal{S}$ is given as $\Rightarrow_{\mathcal{S}(\mathcal{R})}$.

Example 5 (Transcription of DNA to RNA). Coming back to Examples 1 and 4, we show a generic rule transcribing DNA to RNA in Figure 3. Now the transcription can be specified by a single rule, with three nonterminals `DNAChain`, `DNANucleotide`, and `isCCCACTuvwxyzAGTGGGAAAAAA`. We first discuss the textual notation of GrGen on the right-hand side of the figure. Six structure rules define the sub-rules `End`, `Chain` and `A`, `C`, `G`, `T` for the first two nonterminals; `Chain` uses `DNAChain` recursively. The rays and adjacent nodes of these nonterminals are given by the names and types of the formal parameters that follow their name, plus those that follow the ***modify*** blocks in their rules. The structure rule for the nonterminal `isCCC...AAA` defines a sub-pattern; note that it is used for two (anonymous) variables: as a negative application pattern in the structure rule `End`, and as a positive pattern in `Chain`. Using two (or more) variables of the same structure within one rule is also important for expressing structural recursion over non-linear structures like trees.

Note that the abstract DNA model employed here, where nodes and edges represent sub-molecules, can easily be defined on the underlying chemical structure that is composed of atoms. To do that, every nucleic base node (of type `A`, `C`, `G`, `T`, or `U`), every phosphate group edge (of type `PG`), and every sugar node (of type `D` or `R`) has to be turned into a nonterminal, whose structure rules specify the corresponding sub-molecules, and have one, two, and three attachment points respectively, which have to be joined according to the chemical bonds between these sub-molecules. See [14] for details.

Rule Application. Instantiating generic rules first, and matching them afterwards is only possible in theory —in practice, we have to interleave instantiation with matching, as sketched in the following *operational semantics* of the recursive rules, which has been implemented in the extension of GrGen [14]:

1. The terminal items in the generic rule's pattern are matched.



```
rule transcription() {
  d1:D -:PG-> d2:D -:PG-> d3:D -:PG-> d4;
  d4:D -:PG-> d5:D -:PG-> d6:D -:PG-> d7;
  d7:D; d1 --> :T; d2 --> :A; d3 --> :T;
  d4 --> :A; d5 --> :A; d6 --> :A;
  d2r:DNAChain(d7);
  modify {
    r:R; // new starting point for RNA
    d2r(r);
  }
}
pattern DNAChain(prev:D) {
  alternative {
    End {
      :isCCCACTuvwxyzAGTGGGAAAAAA(prev);
      modify(rprev:R) {
      }
    }
    Chain {
      negative {
        :isCCCACTuvwxyzAGTGGGAAAAAA(prev);
      }
      prev -:PG-> next:D;
      head:DNANucleotide(prev);
      tail:DNAChain(next);
      modify(rprev:R) {
        rprev -:PG-> rnext:R;
        head(rprev);
        tail(rnext);
      }
    }
  }
  modify(rprev:R) { }
}
pattern DNANucleotide(d:D) {
  alternative {
    A { // analogously for C, G. T
      d --> a:A;
      modify(r:R) {
        r --> u:U;
      }
    }
    ...
}
pattern isCCCACTuvwxyzAGTGGGAAAAAA(d:D) {
   ...
}
```

Fig. 3. DNA-to-RNA Transcription defined with a generic rule

2. It is checked whether the terminal items of a negative pattern may be matched.
3. If this is the case, a variable attached to this negative pattern is substituted according to a structure rule, and matching continues in step 2. If no variable is left in the negative pattern, a match is found, and application of the rule fails.
4. Otherwise, a variable attached to the pattern is substituted with one of its structure rules, and matching continues with step 4. If there is no variable anymore in the pattern, application of the rule succeeds.
5. The replacement of the generic rule—wherein variables are now instantiated—replaces the match of the rule.

The structure rules $\mathcal{S}$ correspond to hyperedge replacement graph grammars. Thus non-productive nonterminals, unused nonterminals, and chain rules can be detected and removed [9]. When we assume $\mathcal{S}$ to be free of such nonterminals and rules, the operational semantics is effective, since the substitution process is bound to terminate. If the rules are defined with care, it can also be efficient.

5 Conclusions

In this paper we have described a concept by which graph rewrite rules can be refined recursively so that advanced transformation tasks can be specified by a single rule, without using imperative control structures. The declarative rule gets applied in one single step to the host graph, in contrast to the sequence of host graph states occuring during programmed rewriting. The concept has been implemented in GrGen.NET 2.0, which is available at www.grgen.net. Due to lack of space, we have simplified the full concepts of GrGen in several respects: Nodes and edges of graphs may carry attribute values, their typing may use inheritance, and the structure rules used for refining generic rules may themselves be conditional.

The idea of using rules to refine rules has been first used in two-level (van-Wijngaarden) grammars [3]. Early adaptations of this idea to graph grammars [13, 8] were oriented towards defining graph languages, and not intended for defining computations on graphs. The graph variables of shaped generic graph rewrite rules [5] resemble the variables introduced here; they are refined by adaptive star replacement [4]. This is more general than the star replacement used here, but more difficult (and less efficient) to implement. Graph variables as such were first proposed in [16], but without the capability to constrain the shape of the graph to be matched. The path expressions and multi-nodes of Progres [17] and Fujaba [6] allow matching of a subset of the structures which can be handled by recursive sub-patterns. (In contrast to the instantiations defined here, the match of a path expression may overlap with the rest of a match.) Fujaba [6] as well as earlier versions of GrGen furthermore support recursion on the right hand side of a rule, i.e., it is possible to call a rule during a rewrite step, after the match is done. This is purely imperative, because the calling rule will make its changes to the graph anyway—regardless whether the called rule is applicable or not. VIATRA [1] was the first graph rewrite system to support recursive sub-*patterns*, sub-*rules* however are not supported (they are only vaguely sketched in the given reference). As of now it still is the only other system offering sub-patterns, but about two orders of magnitude slower than GRGEN [14]. In any of the mentioned cases, variables are placeholders for sub-patterns only, so that recursive patterns can get matched, but not rewritten (besides deleting the entire sub-pattern).

An interesting question for the future is: *Can rules and patterns be merged to a single concept?* Then, generic rules could refer to other rules like to variables, and the application of a rule could "call" other rules, also recursively. With an additional concept for the sequential composition of rules, this could set up a fully declarative way of programming with graph rewrite rules that is computationally complete in the sense of [12]. For such a declarative framework, it would also be promising to analyze properties of generic rules, such as the existence of critical pairs, or to try to transfer first results concerning overlapping rules with graph variables [11] to it.

References

1. A. Balogh and D. Varró. Pattern composition in graph transformation rules. In *European Workshop on Composition of Model Transformations*, Bilbao, Spain, July 2006. See also http://viatra.inf.mit.bme.hu/update/R2.
2. J. Blomer and R. Geiß. GRGEN.NET: A generative system for graph-rewriting, user manual. www.grgen.net, 2007.
3. C. Cleaveland and R. Uzgalis. *Grammars for Programming Languages.* Elsevier, New York, 1977.
4. F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. Van Eetvelde. Adaptive star grammars. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *3rd Int'l Conference on Graph Transformation (ICGT'06)*, number 4178 in Lecture Notes in Computer Science, pages 77–91. Springer, 2006.
5. F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. Van Eetvelde. Shaped generic graph transformation. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, Lecture Notes in Computer Science. Springer, 2008. to appear.
6. T. Fischer, J. Niere, L. Turunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in Lecture Notes in Computer Science, pages 296–309. Springer, 2000. http://www.fujaba.de/.
7. R. Geiß. *Graphersetzung mit Anwendungen im Übersetzerbau (in German).* Dissertation, Universität Karlsruhe, 2007.
8. H. Göttler. Semantical descriptions by two-level gaph-grammars for quasi-hierarchical graphs. In M. Nagl and H.-J. Schneider, editors, *Graphs, Data Structures, Algorithms (WG'79)*, number 13 in Applied Computer Science, pages 207–225, München-Wien, 1979. Carl-Hanser Verlag.
9. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.

10. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.
11. A. Habel and B. Hoffmann. Parallel independence in hierarchical graph transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *2nd Int'l Conference on Graph Transformation (ICGT'04)*, number 3256 in Lecture Notes in Computer Science, pages 178–193. Springer, 2004.
12. A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.
13. W. Hesse. Two-level graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 255–269. Springer, 1979.
14. E. Jakumeit. *Mit* GrGen.NET *zu den Sternen*. Diplomarbeit (in German), Universität Karlsruhe (TH), 2008. http://www.info.uni-karlsruhe.de/papers/da_jakumeit.pdf.
15. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.
16. D. Plump and A. Habel. Graph unification and matching. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
17. A. Schürr, A. Winter, and A. Zündorf. The Progres approach: Language and environment. In G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, chapter 13, pages 487–550. World Scientific, Singapore, 1999.