A Heterogeneous Pushout Approach to Term-Graph Transformation

Dominique Duval\textsuperscript{1}, Rachid Echahed\textsuperscript{2}, and Frédéric Prost\textsuperscript{2}

\textsuperscript{1} LJK, Université de Grenoble, France
Dominique.Duval@imag.fr
\textsuperscript{2} LIG, CNRS, Université de Grenoble
B.P.53, F-38041 Grenoble, France
Rachid.Echahed@imag.fr, Frederic.Prost@imag.fr

Abstract. We address the problem of cyclic termgraph rewriting. We propose a new framework where rewrite rules are tuples of the form $(L, R, \tau, \sigma)$ such that $L$ and $R$ are termgraphs representing the left-hand and the right-hand sides of the rule, $\tau$ is a mapping from the nodes of $L$ to those of $R$ and $\sigma$ is a partial function from nodes of $R$ to nodes of $L$. The mapping $\tau$ describes how incident edges of the nodes in $L$ are connected in $R$, it is not required to be a graph morphism as in classical algebraic approaches of graph transformation. The role of $\sigma$ is to indicate the parts of $L$ to be cloned (copied). Furthermore, we introduce a notion of heterogeneous pushout and define rewrite steps as heterogeneous pushouts in a given category. Among the features of the proposed rewrite systems, we quote the ability to perform local and global redirection of pointers, addition and deletion of nodes as well as cloning and collapsing substructures.

1 Introduction

Complex data-structures built by means of records and pointers, can formally be represented by termgraphs \cite{21518}. Roughly speaking, a termgraph is a first-order term with possible sharing and cycles. The unravelling of a termgraph is a rational term. Termgraph rewrite systems constitute a high-level framework which allows one to describe, at a very abstract level, algorithms over data-structures with pointers. Thus avoiding, on the one hand, the cumbersome encodings which are needed to translate graphs (data-structures) into trees in the case of programming with first-order term rewrite systems and, on the other hand, the many classical errors which may occur in imperative languages when programming with pointers.

Transforming a termgraph is not an easy task in general. Many different approaches have been proposed in the literature which tackle the problem of termgraph transformation. The algorithmic approach such as \cite{218} defines in detail every step involved in the transformation of a termgraph by providing the

\footnote{This work has been partly funded by the project ARROWS of the French Agence Nationale de la Recherche.}
corresponding algorithm; for our purpose, this approach is too close to implementation techniques. In [1], equational definition of termgraphs is exploited to define termgraph transformation. These transformations are obtained up to bisimilar structures (two termgraphs are bisimilar if they represent the same rational term). Unfortunately, bisimilarity is not a congruence in general, e.g., the lengths of two bisimilar but different circular lists are not bisimilar.

A more abstract approach to graph transformation is the algebraic one, first proposed in the seminal paper [10]. It defines a rewrite step using the notion of pushouts. The algebraic approach is quite declarative. The details of graph transformations are hidden thanks to pushout constructs. There are mainly two different algebraic approaches, namely the double pushout (DPO) and the single pushout (SPO) approaches, which can be illustrated as follows:

\[ \begin{array}{ccc}
L & \leftarrow & K \\
\downarrow m & & \downarrow d \\
G & \leftarrow & D \\
\end{array} \quad \begin{array}{ccc}
\begin{array}{c}
L \\
\downarrow m \\
G \\
\end{array} & \leftarrow & \begin{array}{c}
K \\
\downarrow d \\
D \\
\end{array} & \leftarrow & \begin{array}{c}
R \\
\downarrow m' \\
H \\
\end{array}
\end{array} \]

Double pushout: a rewrite step

Single pushout: a rewrite step

In the DPO approach [10,5], a rule is defined as a span, i.e., as a pair of graph morphisms \( L \leftarrow K \rightarrow R \). A graph \( G \) rewrites into a graph \( H \) if and only if there exists a morphism (a matching) \( m : L \rightarrow G \), a graph \( D \) and graph morphisms \( d, m', l', r' \) such that the left and the right squares in the diagram above for a DPO step are pushouts. In general, \( D \) is not unique, and sufficient conditions may be given in order to ensure its existence, such as dangling and identification conditions. Since graph morphisms are completely defined, the DPO approach is easy to grasp, but in general this approach fails to specify rules with deletion of nodes, as witnessed by the following example. Let us consider the reduction of the term \( f(a) \) by means of the rule \( f(x) \rightarrow f(b) \). This rule can be translated into a span \( f(x) \leftarrow K \rightarrow f(b) \) for some graph \( K \). When applied to \( f(a) \), because of the pushout properties, the constant \( a \) must appear in \( D \), hence in \( H \), although \( f(b) \) is the only desired result for \( H \), in the context of term rewriting.

In the SPO approach [17,12,13,9], a rule is a partial graph morphism \( L \rightarrow R \). When a (total) graph morphism \( m : L \rightarrow G \) exists, \( G \) rewrites to \( H \) if and only if the square in the diagram above for a SPO step is a pushout. This approach is appropriate to specify deletion of nodes thanks to partial morphisms. However, in the case of termgraphs, some care should be taken when a node is deleted. Indeed, deletion of a node causes automatically the deletion of its incident edges. This is not sound in the case of termgraphs since each function symbol should have as many successors as its arity.

In this paper, we investigate a new approach to the definition of rewrite steps for cyclic termgraphs. We are interested in rewrite steps such that \( H \) is obtained from \( G \) by performing one of the six following kinds of actions: (i) addition of new nodes, (ii) redirection of particular edges, (iii) redirection of all incoming edges of a particular node, (iv) deletion of nodes, (v) cloning of nodes, and (vi) collapsing of nodes. In order to deal with these features in a single framework,