Transitive Term Graph Rewriting

R. Banach
Computer Science Dept., Manchester University,
Manchester, M13 9PL, UK.
(banach@cs.man.ac.uk)

Abstract: A version of generalised term graph rewriting is presented in which redirections are interpreted transitively. It is shown that the new semantics enjoy enhanced properties regarding ease of implementation on distributed machines compared with conventional semantics because of the Church-Rosser properties of transitive redirections. Furthermore, the good properties of orthogonal systems are largely retained.

Key Words: Distributed Computing, Programming Languages, Term Graph Rewriting, Transitive Redirections, Packet Store Rewriting, DACTL, MONSTR.

1.0 Introduction

In term graph rewriting [4,7], the basic mechanism of update is redirection; all in-arcs of a node x are made to point to some other node y. If x and y happen to be the same, the model specifies a null action. This imposes the burden of an identity test on implementations, which is mild enough for a single processor serial implementation, but becomes burdensome in a distributed environment. Furthermore, when there are multiple redirections, all redirections are performed simultaneously; for example if x is to be redirected to y, and y to z, the original in-arcs of x end up at y, and the original in-arcs of y end up at z. The in-arcs of z also remain at z assuming that z was not itself redirected.

In this paper we describe a different operational semantics for redirections which obviates the problems that distributed implementations have, and which moreover preserves most of the nice properties of the original semantics. Redirections are interpreted *transitively*, i.e. if x is to be redirected to y, then the in-arcs of x end up wherever the in-arcs of y end up. In the preceding example, all the original in-arcs of x, y, z, end up at z. This raises the question of what to do about cycles of redirections. To cope with these we introduce special purpose $\perp$-nodes, with suitable properties.

2.0 Transitive Term Graph Rewriting

We assume given an alphabet of symbols $\mathbf{S}$, and two further distinguished symbols, Any and $\perp$.

Definition 2.1 A graph G is a triple $(N_G, \sigma_G, \alpha_G)$ where

(1) N_G is a set of nodes.
(2) $\sigma_G : N_G \to \mathbf{S} \cup \{\perp\}$, is the labelling function.
(3) $\alpha_G : N_G \to N_G{}^*$ sends each node to its sequence of children.

And such that $\perp$-labelled nodes have no children (i.e. $\sigma_G(x) = \perp \Rightarrow \alpha_G(x) = []$).

Definition 2.2 A pattern P is a triple $(N_P, \sigma_P, \alpha_P)$ where

(1) N_P is a set of nodes.
(2) $\sigma_P : N_P \to \mathbf{S} \cup \{\mathsf{Any}\}$, is the labelling function.
(3) $\alpha_P : N_P \to N_P{}^*$ sends each node to its sequence of children.

And such that Any-labelled nodes have no children (i.e. $\sigma_G(x) = \mathsf{Any} \Rightarrow \alpha_G(x) = []$).

Definition 2.3 A matching $h : P \rightarrow G$ of a pattern P to a graph G is a node map $h : N_P \rightarrow N_G$ such that for all $x \in N_P$ such that $\sigma_P(x) \neq \mathsf{Any}$ and $\sigma_G(h(x)) \neq \bot$

(1) $\sigma_P(h(x)) = \sigma_G(x)$.
(2) $\mathrm{length}(\alpha_G(h(x))) = \mathrm{length}(\alpha_P(x))$ and for all $k \in \mathrm{indices}(\alpha_P(x))$, $h(\alpha_P(x)[k]) = \alpha_G(h(x))[k]$.

A matching is *proper* iff $h^{-1}\{x \in N_G \mid \sigma(x) = \bot\}$ consists only of Any-labelled nodes of P.

Definition 2.4 A rule $D = (L \subseteq P, \mathit{Red})$ is a triple where

(1) P is a pattern.
(2) L is a subpattern of P containing at least all the Any-labelled nodes of P *linearly*, (i.e. each Any-labelled node has exactly one parent in L).
(3) $\mathit{Red} \subseteq L \times P$ is the (set theoretic) graph of a partial function such that $(a, b), (c, d) \in \mathit{Red} \Rightarrow [\ \sigma(a) = \sigma(c) \Rightarrow a = c$ and $\sigma(a) \neq \mathsf{Any}\ ]$.

Thus a rule D is given by the inclusion of the left subpattern L into the full pattern of D, and by the set Red, which specifies the redirections of which we spoke before.

A rule $D = (L \subseteq P, \mathit{Red})$ is applied to a graph G, by first finding a proper matching $g : L \rightarrow G$ (called the redex). Then in our version of rewriting, there are three phases: contractum building which produces graph G', $\bot$-analysis, and finally redirection which produces the result of the rewrite H.

Contractum building glues a copy of $P - L$ into G at $g(L)$, in such a way that there is a proper matching $g' : P \rightarrow G'$ that extends g in the obvious way. Formally (writing $\uplus$ for disjoint union), we have:

Definition 2.5 G' is given by

(1) $N_{G'} = N_G \uplus (N_P - N_L)$.
(2) $\sigma_{G'}(x) = \sigma_G(x)$ if $x \in N_G$,
$\sigma_P(x)$ otherwise.
(3) $\alpha_{G'}(x)[k] = \alpha_G(x)[k]$ if $\{x, \alpha_G(x)[k]\} \subseteq N_G$,
$\alpha_P(x)[k]$ if $\{x, \alpha_P(x)[k]\} \subseteq (N_P - N_L)$,
y if $x \in N_P$ and $g(\alpha_P(x)[k]) = y$.

$\bot$-analysis consists of the following. Let

$\mathit{Red}' = \{(x, y) \mid$ for some $(a, b) \in \mathit{Red}$, $g'(a) = x$, $g'(b) = y\}$

View Red' as a relation on $N_{G'}$, writing $\mathit{Red}'+$, $\mathit{Red}'*$ for its transitive, reflexive transitive closures.

Let $x \sim y$ iff $\exists\, z \in N_{G'} \bullet x\ \mathit{Red}'*\ z$ and $y\ \mathit{Red}'*\ z$. Then $\sim$ is clearly an equivalence relation because Red is a partial function on P (hence so is Red' on G' by 2.4.(3)). We write $[x]$ to represent the equivalence class containing x as usual. Further, we will write $[x]^\circ$ iff $\exists\, y \in [x] \bullet y\ \mathit{Red}'+\ y$ (i.e. we write $[x]^\circ$ to indicate that $[x]$ contains a non-trivial Red'-cycle). We write $[x]^{-}$ otherwise.

Assuming rules are finite, it is an easy lemma to show that

$\forall\, [x]^{-} \bullet \exists!\, y^{-} \in [x]^{-} \bullet \forall\, x \in [x]^{-} \bullet x\ \mathit{Red}'*\ y^{-}$

(i.e. each $[x]^{-}$ equivalence class is a tree in $N_{G'}$ and has a unique root y^{-}). When the context makes the class $[z]^{-}$ clear, we will use the $^{-}$ notation to refer to this root element without further comment.

Redirection now performs the redirections specified in Red' transitively, building new $\bot$-nodes to catch the Red' cycles.

Definition 2.6 H is given by

(1) $N_H = N_{G'} \uplus B$ where $B = \{[x]^\circ \mid x \in N_{G'}\}$.
(2) $\sigma_H(x) = \sigma_{G'}(x)$ if $x \in N_{G'}$,
$\bot$ if $x \in B$.
(3) $\alpha_H(x)[k] = y^{-} \in N_{G'}$ if $\alpha_{G'}(x)[k] = y$ and $y \in [y^{-}]^{-}$,
$[y]^\circ \in B$ if $\alpha_{G'}(x)[k] = y$ and $y \in [y]^\circ$,
$\alpha_{G'}(x)[k]$ otherwise.

Below we give an example of this semantics on the infamous circular-I rewrite. Using DACTL-like syntax [4], the rule is I[a:Any] => a, which means that an I-labelled node is to be redirected to its only child. Applied to the circular graph x:I[x], conventional redirection semantics yields an unchanged result, while ours gives the rewrite in Fig. 1, where bot:$\bot$ is a $\bot$-node introduced because x Red' x.

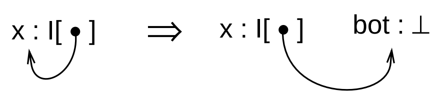

Fig. 1.

As a further example consider the rule r:F[a:S[b:Any]] –> r:=a, a:=b (where $P - L$ is empty and the redirections are (r, a), (a, b)), on the cyclic graph x:F[y:S[x]]. Conventional semantics yields self loops on F and S, while ours gives Fig. 2.

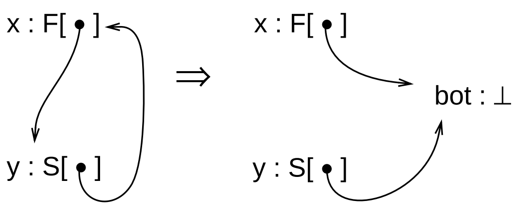

Fig. 2.

Note that we have said nothing about garbage collection, nor about whether left hand sides of rules (the L subpatterns) need be rooted; these are somewhat orthogonal issues, to which we will allude below as required.

Here is a larger example.

f:F[s:S[a:Any t:T[b:Any c:Any]] u:U[t v:V]]
–> x:X[a y:Y], f:=u, u:=v, t:=s, s:=a

(Here $P - L$ is the nodes x and y, and the redirections are written using assignment notation.) Applied to the graph in Fig. 3, we get the result illustrated.

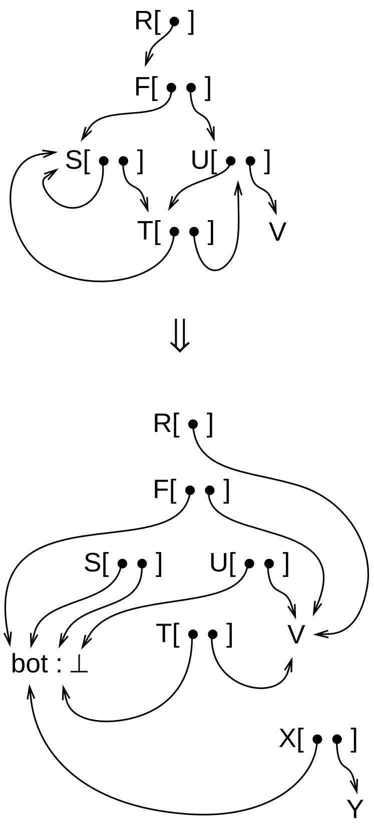

Fig. 3.

3.0 Implementation

As mentioned above, adherence to non-transitive redirection semantics demands determining precisely the left and right nodes of each redirection in order to ensure that when they coincide, nothing (untoward) is done by the implementation. Things get worse when there are many agents rewriting the graph concurrently, as the left and right nodes (x, y) of a redirection need to be locked by the agent doing the rewrite so that another agent does not inadvertently cause x and y to be identified, while the first is treating them as distinct.

Things get even worse still in a distributed implementation with the graph scattered over several processors, as an agent may have only imperfect information about the identity of a node, having access to perhaps only an indirection to the node it is really interested in; necessitating distributed locking with all its attendant overheads and difficulties, to ensure a correct implementation.

With transitive redirection semantics though, the situation regarding all of the above becomes much happier. Let us fix on a specific implementation model in which nodes are represented by *packets*, residing at *locations* in a data store, where each packet contains the node symbol and a sequence of pointers leading (perhaps via a sequence of indirections) to the locations of packets representing the chldren of the node in question.

Let us further assume that we are working under the umbrella of a notion of garbage in which the left hand node x of any redirection (x, y) can be garbaged unless x is itself the right hand side of some other redirection (w, x); such notions of garbage arise quite naturally in many applications.

Under these asumptions the non-transitive redirection model has an implementation that can be discerned from section 6 of [2]. The transitive model of rewriting however enjoys a particularly simple implementation as follows.

Packet_Rewrite : *Packet_store* → *Packet_Store* ;

Pattern_Match:
Find a suitable redex to rewrite. (Note that because the matching for a rewrite is required to be proper, no ⊥-node ever needs to have its symbol pattern-matched for a successful rewrite, and attempts by the implementation to match ⊥-nodes may be allowed to fail quietly. This is not to say that ⊥-nodes cannot appear matched to Any-nodes in the redex. See below.)

Contractum_Building:
Allocate fresh packets for all nodes in $P - L$, giving each the symbol of the node it represents and a sequence of pointers to the representatives of its children.

Redirection:
For each redirection $(x, y) \in Red'$, overwrite the packet representing node x, by an indirection packet (Ind packet) pointing to the packet representing node y.

End(*Packet_Rewrite*).

For the second example above we get the packet store transformation in Fig. 4.

Proposition 3.1 *Packet_Rewrite* provides a correct implementation of the transitive term graph rewriting model (up to garbage).

The proof rests on the facts that: (a), since LHS's of all redirections are garbage in the transitive model, it is safe to overwrite their packet representatives; (b), the semantics of an indirection are to redirect all incoming pointers to the target of the indirection, or if the target is itself an indirection, to *its* target, and so on. If the rewrite specifies a cycle of redirections, a cycle of Inds is created, corre-

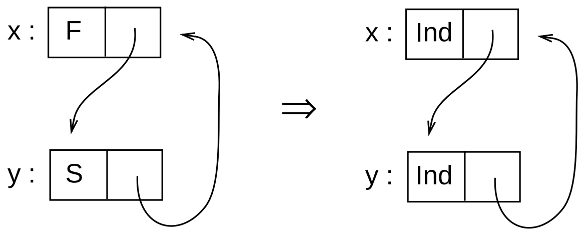

Fig. 4.

sponding to a ⊥-node $[-]^{\circ}$ in B in the formal model; otherwise the chain of Inds ends at the representative of the unique y^{-} element of the appropriate class $[-]^{-}$ of the model.

We further note that with non-transitive redirection semantics, given a set of redirection pairs Red', the order in which they are done is critical. Eg. if $Red' = \{(x, y), (y, z)\}$, (and all of x, y, z have in-arcs), then we get different results according to whether we do the redirections singly (either sequential order giving a result in agreement with the transitive model[1]), or simultaneously (giving the conventional result for a pair of redirections done non-transitively). Larger aggregates of redirections give a greater variety of possible answers.

Transitive redirection semantics suffers no such problems, as in the example above, all in-arcs of x, y, z end up at z, no matter what the order or granularity of performing the redirections. This Church-Rosser property of transitive redirections can be incisively exploited by concurrent implementations in reducing the redex-locking burden of packet rewriting.

Specifically, for each redirection (x, y), a packet rewrite needs to access x's packet, but only needs a pointer to (perhaps only an indirection chain leading to) y's packet. If y is matched to an Any node of the rule, such a pointer is present in some non-Any matched packet of the redex (by the linearity requirement), and so the packet rewrite need

1. Note that if we do (y, z) first, the redirection (x, y) becomes (x, z) since all references to y become references to z.

not ascertain the precise location or nature of y at all to correctly perform the redirection (x, y).

Indeed y may be a $\perp$-node, or even the left node of a redirection (y, z) of some other packet rewrite that matches y to a non-Any node; and provided the non-Any matched nodes of the two rewrites are distinct, they may be done in any order, or simultaneously, getting in every case the same result. We may even have that $z = x$ if z is matched to an Any node of the second rewrite. Since all left nodes of redirections are non-Any matched we can easily prove:

Proposition 3.2 *Packet_Rewrite* provides a correct *concurrent* implementation of the transitive term graph rewriting model (up to garbage), if each *Packet_Rewrite* action locks only the non-Any matched packets of the redex.

The above can be exploited by implementations of MONSTR [2], an extended term graph rewriting formalism intended for execution on parallel machines using a packet rewriting strategy similar to that which we have described. MONSTR rules feature small, rooted left hand sides, with a function symbol labelling the root, and with the function having at most a single child labelled with a stateholder symbol; other children of the root may be labelled with constructors. (Unlike functions and stateholders, constructors may not occur as left nodes of redirections.) And all these nodes mentioned, may further have additional Any-labelled children.

Since constructors may not be redirected (and thus may be concurrently inspected by other rewrites without harm), a concurrent implementation of MONSTR needs to lock only the function and its stateholder child during a rewrite in order to ensure correctness, if the rules are interpreted according to transitive semantics.

In fact implementations of MONSTR use such a locking strategy, but in attempting to adhere to non-transitive semantics, inevitably fail on certain rewrites (both in constructing Ind loops, and in misinterpreting the right nodes of redirections in certain cases). The interesting thing is that such troublesome rewrites are never encountered in systems of practical interest. In fact one can redefine the operational semantics of MONSTR to include transitive redirections (and other features) without harming the behaviour of most useful programs, in such a manner that a very reasonable Church-Rosser theorem holds for systems featuring at worst deterministic synchronisations [3].

4.0 Graph and Term Rewriting

Of particular interest among graph rewriting systems, are ones that look like implementations of term rewrite systems. Such systems feature rules such that: (a), P has at least one and at most two root nodes (and one of the roots is the (unique) root of L); (b), there is exactly one redirection, of the root node of L to the other root (if P has two roots), or to some subroot node of L (if not); (c), the notion of liveness/garbage that is used, pronounces live all nodes accessible from a distinguished root node of the graph (modulo redirections of this root), and pronounces all other nodes garbage. We will call such systems term emulating graph rewriting systems (TEGRS) because their correspondence with term rewrite systems has been widely studied. See [6,8] and further references therein.

Since a TEGRS rewrite features only one redirection, the only way that transitive semantics can give a result different from non-transitive semantics is if for a redex $g : L \to G$, we have $g(root) = g(b)$ where $\{(root, b)\} = Red$, i.e. the cyclic redex case where transitive semantics creates a $\perp$-node, while non-transitive semantics gives an unchanged graph.

We formally define a graph rewrite system as a triple $(\mathcal{G}, \mathcal{R}, \mathcal{S})$, where $\mathcal{G}$ is a set of graphs, $\mathcal{R}$ is a set of rules (with $\mathcal{G}$ closed under rewriting by $\mathcal{R}$), and $\mathcal{S}$ is a set of rewrite sequences of members of $\mathcal{G}$ by $\mathcal{R}$. We write $\mathcal{S}^-$ to denote the the set $\mathcal{S}$ with all rewrites of circular redexes taken out of each element of $\mathcal{S}$ (under either semantics). The following is then obvious.

Proposition 4.1 Let $(\mathcal{G}, \mathcal{R}, \mathcal{S})$ be a TEGR system under non-transitive semantics. Then $(\mathcal{G}, \mathcal{R}, \mathcal{S}^-)$ is a TEGR system under transitive semantics. Conversely, let $(\mathcal{G}, \mathcal{R}, \mathcal{S})$ be a TEGR system under transitive semantics with $\mathcal{S} = \mathcal{S}^-$. Then $(\mathcal{G}, \mathcal{R}, \mathcal{S})$ is a TEGR system under non-transitive semantics.

This shows that everything that can be achieved by non-transitive semantics, can also be achieved by transitive semantics, if the latter pursues a $\perp$-avoiding strategy, so to speak; and vice versa.

The main application of such graph rewriting is in the emulation of term rewriting via unraveling, which takes a rooted term graph to the term (tree) whose nodes are the paths through the graph from the root, with labels inherited from the last node in the path. Of the unraveling mapping, one can then show the following, which is a rather trivial adaptation of theorem 6.10 of [8]. We quote it without proof, or indeed without explaining the terminiology used more precisely, relying on the reader's intuition to furnish a good feeling of what is being conveyed, and on the cited reference for a complete development of the required theory.

Theorem 4.2 Under any of the following conditions, the unraveling mapping from an orthogonal transitive term graph rewriting system (G, R, S), to a term rewriting system is adequate.

(1) The TTGRS is finitary and acyclic, and the TRS is its finitary unraveling.

(2) The TTGRS is finitary, $S = S^-$, the TRS is its rational unraveling, and the rule system is almost non-collapsing.

(3) The TTGRS is finitary, the TRS is its rational unraveling, hypercollapsing terms are identified, and graphs rooted at a circular redex are identified with the results of rewriting such redexes.

5.0 Conclusions

We have presented a transitive variation of the conventional semantics of multiple redirection, to get a new model of generalised term graph rewriting. We have demonstrated that the new model has dramatically improved implementability properties for distributed environments compared with non-transitive redirection. We have also observed that the correspondence between term and graph rewriting does not lose very much under the new semantics. Finally we mention two related pieces of work which arrive at essentially the same result as we do for the circular-I rewrite albeit from different directions. In [5] Corradini gets the result by considering parallel rewriting under the approximation topology of CT_Σ in which a $\perp$-node is the graph counterpart of a limit of a series of term rewriting computations each of which yield the term $\perp$. And in [1], Ariola and Klop get the result by considering term graph rewriting interpretations of sets of equations, getting $\perp$ when there is a non well founded cycle of equations. The novelty of the present paper is that we obtain this behaviour in a purely combinatorial setting, in which the notion of update is intrinsically asymmetric and imperative rather than equational; there is no recourse to any specific notion of garbage including that appertaining to the replacement of subterms; and no need for any notion of topology, making the theory applicable to rewriting in which continuity notions are difficult to apply because the graphs in question are too cyclic.

6.0 Acknowledgements

The work reported here was done while the author was on leave at the Computer Science Department of the University of Cyprus. The hospitality of that department, and the financial support of the Royal Society are gratefully acknowledged.

7.0 References

[1] Ariola Z.M., Klop J.W., (1995) *Equational Term Graph Rewriting*. CWI Report CS-R9552, July 1995, Acta Informatica, *to appear.*

[2] Banach R., (1996) *MONSTR I — Fundamental Issues and the Design of MONSTR*. Journal of Universal Computer Science, **2**, 164-216. `http://www.springer.de`

[3] Banach R., (1996) *MONSTR V — Transitive Coercing Semantics and the Church-Rosser Property.* Submitted to Information and Computation.

[4] Barendregt H.P., van Eekelen M.C.J.D., Glauert J.R.W., Kennaway J.R., Plasmeijer M.J., Sleep M.R., (1987) *Term Graph Rewriting*. in Proc. PARLE-87, LNCS **259**, Springer, 141-158.

[5] Corradini A., (1993) *Term Rewriting in CT_Σ*. Proc. CAAP-93, LNCS, **668**, Springer, 468-484.

[6] Farmer W.M., Watro R.J., (1990) *Redex Capturing in Term Graph Rewriting*. International Journal of Foundations of Computer Science, **1**, 369-386.

[7] Glauert J.R.W., Kennaway J.R., Sleep M.R., (1990) *DACTL: An Experimental Graph Rewriting Language*. in Graph Grammars and their Applications to Computer Science, Ehrig, Kreowski, Rozenberg (eds.), LNCS **532**, Springer, 378-395.

[8] Kennaway J.R., Klop J.W., Sleep M.R., de Vries F.J., (1994) *On the Adequacy of Graph Rewriting for Simulating Term Rewriting*. ACM Transactions on Programmng Languages and Systems, **16**, 493-523.