internal report nr. 87

Term Graph Rewriting

H.P. Barendregt[1], M.C.J.D. van Eekelen[1], J.R.W. Glauert[2],
J.R. Kennaway[2], M.J. Plasmeijer[1] and M.R.Sleep[2].

october 1986

[1] University of Nijmegen, Nijmegen, The Netherlands,
and partially supported by the Dutch Parallel Reduction Machine Project.
[2] School of Information Systems, University of East Anglia, Norwich, U.K.,
and partially supported by the U.K. ALVEY Project.

Term Graph Rewriting

H.P. Barendregt[1], M.C.J.D. van Eekelen[1], J.R.W. Glauert[2],
J.R. Kennaway[2], M.J. Plasmeijer[1] and M.R.Sleep[2].

Abstract

Graph reduction as defined in Wadsworth [1971] was introduced in order to be able to give a more efficient implementation of functional programming languages in the form of lambda calculus or term rewrite systems: identical subterms are shared using pointers.

Several other authors, e.g. Ehrig [1979], Staples [1980a,b,c], Raoult [1984] and van den Broek et al. [1986] have given mathematical descriptions of graph reduction, usually employing some concepts from category theory, in particular the notion of a pushout. These papers prove among other things the correctness of graph reduction in the form of the Church-Rosser property. However, no results were available as to the soundness and completeness of graph reduction with respect to term reduction.

In this paper we give a direct operational description of graph reduction that avoids the category theoretic notions. We show that if a term t is interpreted as a graph g(t) and is reduced in the graph world, then the result represents an actual reduct of the original term t (*soundness*). For weakly regular term rewrite systems, there is also a *completeness* result: every normal form of a term t can be obtained from the graphical implementation. We also show completeness for all term rewrite systems which possess a so called hypernormalising strategy, and in that case the strategy also gives a normalising strategy for the graphical implementation.

Graph reduction is not only of importance as an efficient implementation technique of term rewriting, but is also interesting by itself. Among other things side effects can be described in general graph reduction. The language LEAN is being developped by the same authors for this purpose. The corresponding extension of the theory of graph reduction will be developped in a future paper.

[1] University of Nijmegen, Nijmegen, The Netherlands,
and partially supported by the Dutch Parallel Reduction Machine Project.

[2] School of Information Systems, University of East Anglia, Norwich, U.K.,
and partially supported by the U.K. ALVEY Project.

CONTENTS.

0. Introduction.
1. Terms as trees and graphs.
2. Term graphs.
3. Graph reduction.
4. Tree reduction.
5. Relations between graph and tree reduction.
6. Normalising strategies.
7. Conclusions.

0. Introduction.

The work described here is the outcome of collaboration between two research groups, based at the University of Nijmegen (The Netherlands) and the University of East Anglia (Norwich, England). The collaboration started in late summer 1985. Both groups had, as central goals, the development of efficient implementations of Functional and other languages and an interest in exploiting the opportunities for parallelism offered by VLSI (Very Large Scale Integration) techniques. A common belief was that *graph reduction*, in the sense of Wadsworth [1971] and e.g. implemented by Turner [1979a,1979b], offered a promising basis for our joint work.

After some discussion, it was agreed that the appropriate focus for collaboration was the development of a common computational model, based on the notion of graph rewriting, and the development of an associated intermediate language called LEAN (Language of East Anglia and Nijmegen) - this is the subject of a separate paper, Barendregt et al.[1986].

Our principal result is that the notion of graph reduction provides a sound and complete representation (in a sense precisely defined below) of *weakly regular* TRSs. A second result concerns the mapping of strategies between the term and the graph worlds. Any strategy that is *hypernormalising* in the term world is normalising (indeed hypernormalising) in the graph world.

The general plan of the paper is as follows: Section 1 presents basic definitions, and introduces a linear syntax for terms represented as graphs. Section 2 introduces the *category of term graphs*. Section 3 defines the notion of graph reduction, and section 4 introduces the notion of *tree rewriting* as a prelude to section 5, which develops our theory of how to relate the worlds of term and graph reduction. Section 6 considers the problem of mapping strategies between the two worlds. Finally, section 7 gives a summary of the work.

1. Terms as trees and graphs.

1.1 DEFINITION. (i) Let F be a (finite or infinite) set of objects called *function symbols*. A,B,... range over F.

(ii) The set T of *terms* over F is defined inductively as follows:

$$A \in F,\ t_1,...,t_n \in T \ \Rightarrow\ A(t_1,...,t_n) \in T$$

A() is written as just A.

1.2 EXAMPLE. Let F={0,S}. Then T ={0,S(0),0(S,S(0,0)), S(S,S,S)...}. Note that we do not assume that function symbols have fixed arities. This might appear inconvenient if one wished to represent, for example, the Peano integers, with a constant 0 and a successor operator S, since one also obtains extra "unintended" terms such as some of those listed above. When we define rewrite systems in section 3, we will see that this does not cause any problems.

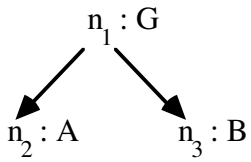
1.3 DEFINITION. A *labelled graph* (*over* F) is a triple (N,lab,succ) involving a (finite or infinite) set N of *nodes*, a function lab: N→F, and a function succ:N→N* such that for all n∈N, if succ(n) = $(n_1,...,n_k)$, then k is the arity of lab(n). In this case we say that the $n_1,...,n_k$ are the *successors* of n. The ith component of succ(n) is denoted by $succ(n)_i$.

When we draw pictures of graphs, a directed edge will go from each node n to each node in succ(n), with the left-to-right ordering of the sources of the edges corresponding to the ordering of the components of succ(n).

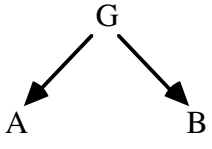
1.4 EXAMPLES. (i) Let N={n_1,n_2,n_3} and define lab and succ on N as follows.

$lab(n_1)=G$, $lab(n_2)=A$, $lab(n_3)=B$,
$succ(n_1)=(n_2,n_3)$, $succ(n_2)=()$, $succ(n_3)=()$.

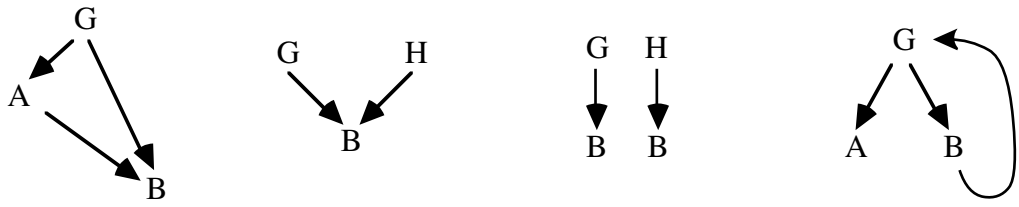
This defines a labelled graph that can be drawn as:



or more simply:



where the second diagram is an abstraction of the first with respect to node assignments. Using this notation, four more examples of graphs are the following.



Note that our labelled graphs are directed graphs: the notion of successor is not symmetric. The successors of a node are ordered. In the pictures we draw, this ordering will always be the left-to-right ordering of the roots of the arrows.

1.5 DEFINITION. (i) A *path* in a labelled graph (N,lab,succ) is a list $(n_1,i_1,n_2,i_2,\ldots,n_{m-1},i_{m-1},n_m)$ where $m\geq 1$, $n_1,\ldots,n_m\in N$, $i_1,\ldots,i_{m-1}\in \mathbb{N}$ and n_{k+1} is the i_k-th successor of n_k. This path is said to be *from* n_1 *to* n_m and m is the *length* of the path.

(ii) A *cycle* is a path of length greater than 1 from a node n to itself. In this case n is called a *cyclic* node.

(iii) A graph is *cyclic* if it contains a cyclic node, otherwise it is *acyclic*.

1.6 DEFINITION. (i) A *term graph* (often, within this paper, simply a *graph*) is a quadruple (N,lab,succ,r) where (N,lab,succ) is a labelled graph and r is a member of N. The node r is called the *root* of the graph. (We do not require that every node of a term graph is reachable by a path from the root.) For a graph g, the components are often denoted by N_g, lab_g, $succ_g$, and r_g.

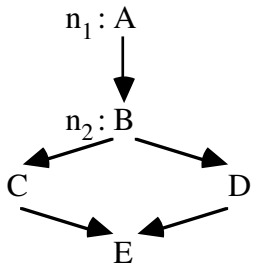
(ii) A path in a graph is *rooted* if it begins with the root of the graph. The graph is *root-cyclic* if there is a cycle containing the root.

When we draw pictures of term graphs, the topmost node is usually the root.

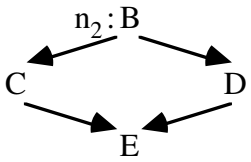
1.7 DEFINITION. Let g = (N,lab,succ) be a labelled graph and let $n\in N$. The *subgraph of* g *rooted at* n is the term graph (N',lab',succ',n) where $N'=\{n'\in N \mid$ there is a path from n to n'$\}$ and lab' and succ' are the restrictions of lab and succ to N'. We denote this graph by g|n. The definition also applies when g is a term graph.

1.8 EXAMPLES.

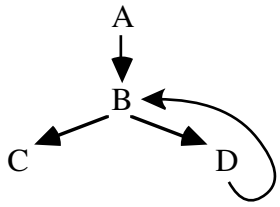
(i)



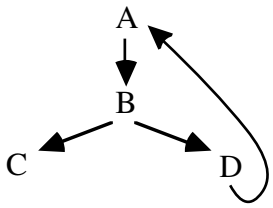
The subgraph rooted at n_2 is:



(ii) A cyclic graph.



(iii) A root-cyclic graph.



A formal description of a graph requires a complete specification of the quadruple (N,lab,succ,r). When writing down examples of finite graphs, it is convenient to adopt a more concise notation, which abstracts away from details such as the precise choice of the elements of N. We will use a notation based on the definition of terms in definition 1.1, but with the addition of node-names, which can express the sharing of common subexpressions. The notation is defined by the following context-free grammar, with the restrictions following it.

1.9 DEFINITION (linear notation for graphs).

graph	::=	node | node + graph	
node	::=	A(node,...,node) | identifier | identifier	**:** A(node,...,node)

A ranges over F. Every identifier x which occurs in a graph must occur exactly once in the context

x **:** A(node,...,node).

Again A() is abbreviated to A. Identifiers are represented by tokens beginning with a lower-case letter. Function symbols will be non-alphabetic, or begin with an upper-case letter.

The five graphs of the examples 1.4 are in this notation as follows.

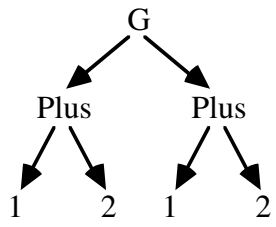
G(A,B);
G(A(x:B),x);
G(x:B) + H(x);
G(B) + H(B);
x:G(A,B(x)).

As indicated by the introductory examples of graphs, we are concerned with abstract graph structure, not with the concrete identities of the nodes. The shorthand reflects this by introducing the syntactic category identifier, which abstracts from particular choices of nodes. An identifier acts as a name for a node, so that repeated uses of the same identifier express multiple references to the same node.

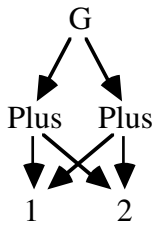
The definition of terms in 1.1 corresponds to a sublanguage of our shorthand notation, consisting of those graphs obtained from the first production for graph and the first production for node. *So terms have a natural representation as graphs.*

1.10 EXAMPLES.

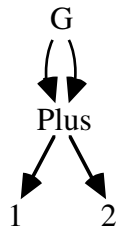
(i) G(Plus(1,2),Plus(1,2))



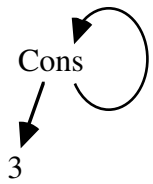
(ii) G(Plus(n_1:1,n_2:2),Plus(n_1,n_2))



(iii) G(n:Plus(1,2),n)



(iv) n1:Cons(3,n1)



1.10 DEFINITION. A *tree* is a graph (N,lab,succ,r) such that there is exactly one path from r to each node in N.

Thus examples (i) and (ii) above are trees, and the others are not. Trees are always acyclic. Notice that a graph g is a finite tree iff g can be written by the grammar of 1.8 without using any identifiers.

The natural mapping of terms to graphs represents each term as a finite tree. However, some terms can also be represented as proper graphs, by sharing of repeated subterms. For example, the term G(Plus(1,2),Plus(1,2)) can be represented by any of the graphs pictured in example 1.9 (i), (ii), or (iii), as well as by the graphs G(Plus(x:1,2),Plus(x,2)) or G(Plus(1,x:2),Plus(1,x)).

2. Term graphs.

2.1 DEFINITION. Given two graphs $g_1 = (N_1,lab_1,succ_1,r_1)$ and $g_2 = (N_2,lab_2,succ_2,r_2)$, a *homomorphism* from g_1 to g_2 is a map $f:N_1 \to N_2$ such that for all $n \in N_1$,

$$lab_2(f(n)) = lab_1(n)$$
$$succ_2(f(n)) = \mathbf{f}(succ_1(n))$$

where **f** is defined by $\mathbf{f}(n_1,...,n_k) = (f(n_1),...,f(n_k))$. That is, homomorphisms preserve labels, successors, and their order.

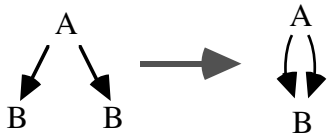
2.2 DEFINITION. Graph(F) is the category whose objects are graphs over F and whose morphisms are homomorphisms. Tree(F) is the full subcategory of Graph(F) whose objects are the trees over F. It is easy to verify that these are categories.

2.3 EXAMPLES. We shall write

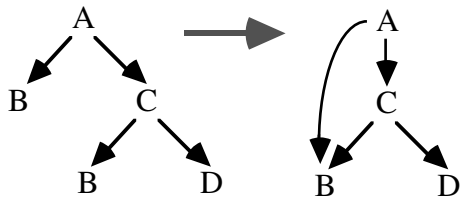
$g_1 \longrightarrow g_2$

when there is a homomorphism from g_1 to g_2. We have the following pictures.

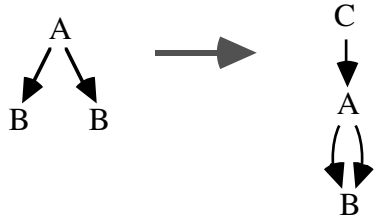
(i)



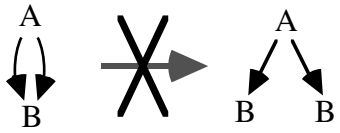
(ii)



(iii)



(iv)



2.4 DEFINITION. (i) A homomorphism $f:g_1 \to g_2$ is *rooted* if $f(r_1)=r_2$.

(ii) An *isomorphism* is a homomorphism which has an inverse. We write $g \sim g'$ when g and g' are isomorphic.

(iii) Two graphs are *equivalent* when they are isomorphic by a rooted isomorphism. We write $g \approx g'$ when g and g' are equivalent.

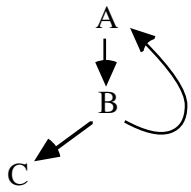
The following proposition collects some simple facts about homomorphisms.

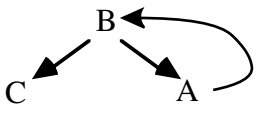
2.5 PROPOSITION. *(i) For any graphs g_1 and g_2 we have*

$g_1 \approx g_2 \Rightarrow g_1 \sim g_2$.

(ii) Every rooted homomorphism from one tree to another is an isomorphism.

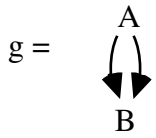
2.6 EXAMPLE. These two graphs are isomorphic but not equivalent (recall that in graphical representations the root node is the topmost):

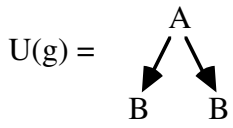




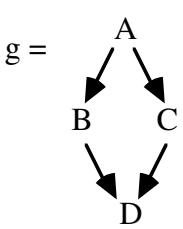
2.7 DEFINITION. Given any graph g=(N,lab,succ,r) we can define a tree U(g) which results from "unravelling" g from the root. We start with some examples.

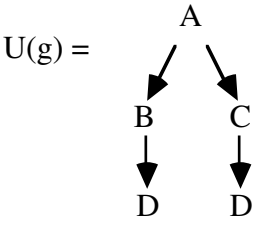
(i)

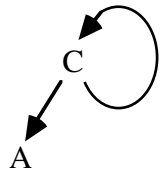


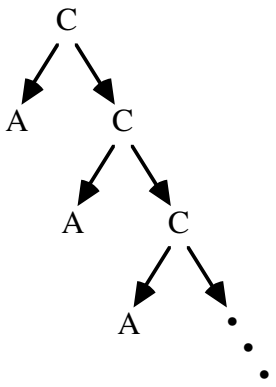


(ii)





(iii) g = 

U(g) = 

Now we give the formal definition. U(g) has as nodes the rooted paths of g. The root of U(g) is the path (r). For a path $p=(n_1,i_1,...,n_{m-1},i_{m-1},n_m)$, $lab_{U(g)}(p) = lab_g(n_m)$ and $succ_{U(g)}(p) = (p_1,...,p_k)$ where p_i is the result of appending $(i,succ_g(n_m)_i)$ to p. Clearly this is a tree.

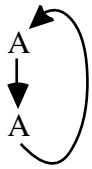
2.8 PROPOSITION. *For every graph g there is a rooted homomorphism u_g: U(g)→g defined by:*

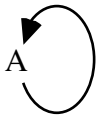
$$u_g(n_1,i_1,...,n_m) = n_m.$$

2.9 PROPOSITION. *A graph g is a tree iff g ≈ U(g).*

2.10 DEFINITION. Two graphs g and g' are *tree-equivalent*, notation $g \approx_t g'$, if U(g) ≈ U(g').

For example, the graphs of example 1.9 (ii), (iii), and (iv) are all tree-equivalent. So are these two graphs:





3. Graph reduction.

We now turn to rewriting. First we recall the familiar definitions of terms with free variables. We then explain informally how we represent terms as graphs, and then define our notion of graph rewriting. Our definition is quite similar to the one in Staples [1980].

3.1 DEFINITION (term rewriting). (i) Let V be a fixed set of function symbols, the members of which are called *variables*, and denoted by lower-case letters. An *open term* over a set of function symbols F is a term over F∪V in which every occurrence of a variable is applied to zero arguments. An open term in which no variables occur (that is, what we have been calling simply a term) is a *closed* term.

(ii) A *term rewrite rule* is a pair of terms t_L and t_R (written $t_L \rightarrow t_R$) such that every variable occurring in t_R occurs in t_L. t_L and t_R are called, respectively, the *left-hand* and *right-hand* sides of the *term rewrite rule* $t_L \rightarrow t_R$.

(iii) A term rewrite rule is *left-linear* if no variable occurs more than once in its left-hand side.

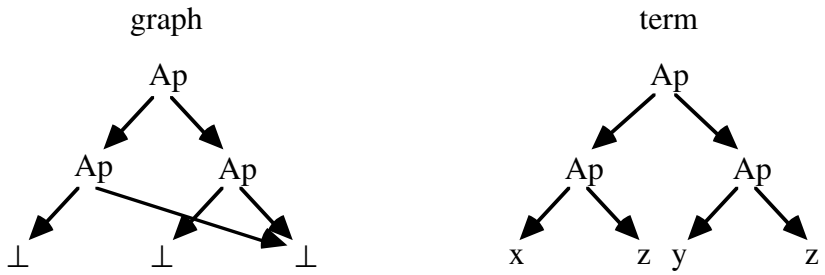
The usual definition of a term rewrite rule requires that t_L be not just a variable. However, our results are not affected by the presence of such rules, so we do not bother to exclude them.

In order to introduce graph reduction, first we need some preparatory definitions.

3.2 DEFINITION. (i) An *open labelled graph* is an object (N,lab,succ) like a labelled graph, except that lab and succ are only required to be partial functions on N, with the same domain. A node on which lab and succ are undefined is said to be *empty*. The definition of an *open (term) graph* bears the same relation to that of a (term) graph. When we write open graphs, we will use the symbol ⊥ to denote empty nodes. As with terms, we talk of closed (labelled or term) graphs and closed trees as being those containing no empty nodes.

(ii) A *homomorphism* from one open graph g_1 to another g_2 is defined as for graphs, except that the "structure preserving" conditions are only required to hold at nonempty nodes of g_1.

Open term graphs are intended to represent terms with variables. Instead of using the set V of variables, we find it more convenient, for technical reasons, to follow Staples[1980] by using empty nodes. The precise translation is as follows. Given an open graph over F, we interpret it as an open term by first replacing each empty node in it by a different variable symbol from V, and then unravelling the resulting closed graph over F ∪ V to obtain an open term over F. Thus where a graph has multiple edges pointing to the same empty node, the term will have repeated occurrences of the same identifier. Thus the graph Ap(Ap(⊥,w:⊥),Ap(⊥,w)) translates to the term Ap(Ap(x,z),Ap(y,z)). Below we show the graph, and the syntax tree of the term corresponding to it.



We could obtain any term differing from this one only by changes of variables. Formally speaking, we actually obtain from an open graph a class of terms, which are distinguished from each other only by the names of their variables. We shall treat such terms as the same.

We now turn to the graph representation of term rewrite rules. We only deal with left-linear rules in this paper. In 5.13 we discuss briefly the problems involved in graphically describing non-left-linear rules.

3.3 DEFINITION. (i) A *graph rewrite rule* is a triple (g,n,n'), where g is an open labelled graph and n and n' are nodes of g, called respectively the *left root* and the *right root* of the rule.

(ii) A *redex* in a graph g_0 is a pair $\Delta = (R,f)$, where R is a graph rewrite rule (g,n,n') and f is a homomorphism from g|n to g_0. The homomorphism f is called an *occurrence* of R.

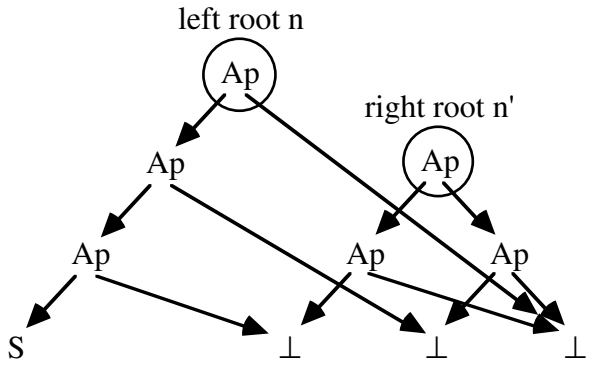
Rather than introduce our formal definition of graph rewriting immediately, we begin with some examples. The formal definition is given in section 3.6.

3.4 TRANSLATION OF TERM RULES TO GRAPH RULES.

Let $t_L \rightarrow t_R$ be a left-linear term rewrite rule. We construct a corresponding graph rewrite rule (g,n,n'), where g is a labelled graph and n and n' are nodes of g. First take the graphs representing t_L and t_R. Form the union of these, sharing those empty nodes which represent the same variables in t_L and t_R. This graph is g. Take n and n' to be the respective roots of t_L and t_R. Here are two examples which should make the correspondence between term and graph rewrite rules clear.

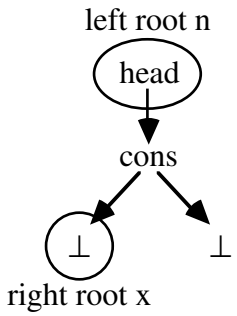
(i) Term rule: $Ap(Ap(Ap(S,x),y),z) \rightarrow Ap(Ap(x,z),Ap(y,z))$

Graph rule: $(n:Ap(Ap(Ap(S,x:\bot),y:\bot),z:\bot) + n':Ap(Ap(x,z),Ap(y,z)), n, n')$



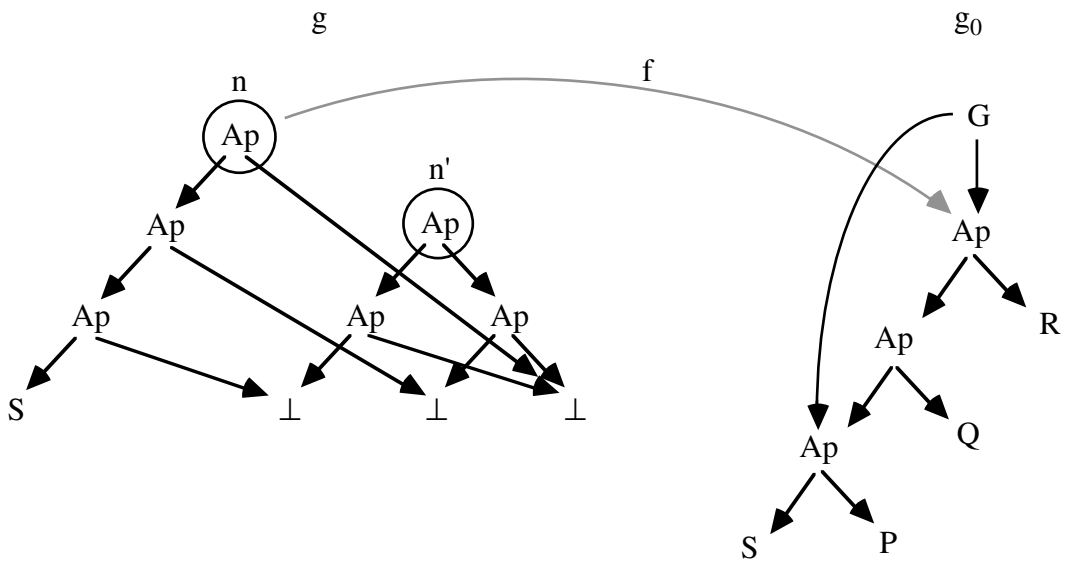
(ii) Term rule: head(cons(x,y)) → x

Graph rule: (n:head(cons(x:⊥,⊥)), n, x)

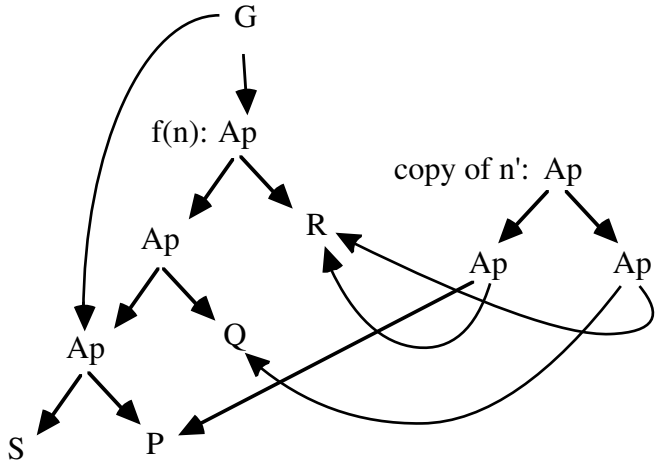


3.5 INFORMAL DEFINITION OF GRAPH REWRITING.

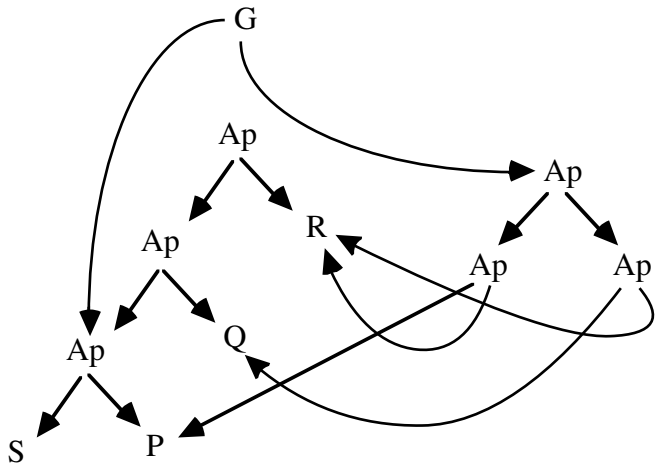
A redex ((g,n,n'), f: g|n → g_0) in a graph g_0 is reduced in three steps. We shall use the following redex as an example: (g,n,n') is the S-rule above, g_0 = G(a,Ap(Ap(a:Ap(S,P),Q),R)) and f operates on n as indicated in the picture (which completely determines how f behaves on the rest of g|n).



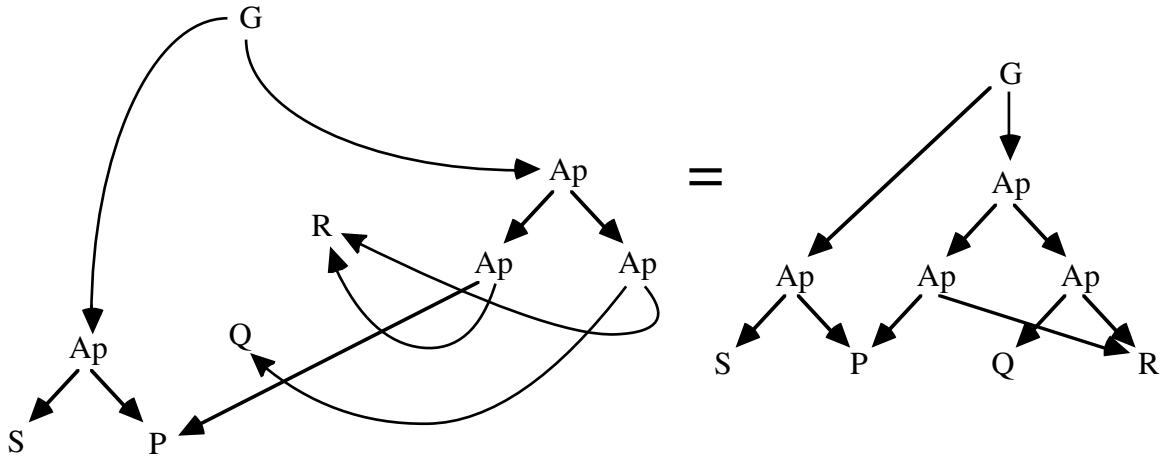
First (the *build* phase) an isomorphic copy of that part of g|n' not contained in g|n is added to g_0, with lab, succ, and root defined in the natural way. Call this graph g_1:



Then (the *redirection* phase) all edges of g_1 pointing to f(n) are replaced by edges pointing to the copy of n', giving a graph g_2. The root of g_2 is the root of g_1, if that node is not equal to f(n). Otherwise, the root of g_2 is the copy of n'.



Lastly (the *garbage collection* phase), all nodes not accessible from the root of g_2 are removed, giving g_3, which is the result of the reduction.



Note that the bottommost Ap node of the redex graph and the S node remain after garbage collection, since they are still accessible from the root of the graph. The other two Ap nodes of the redex vanish.

3.6 FORMAL DEFINITION OF GRAPH REWRITING.

We now give a formal definition of the general construction. Let $((g,n,n'), f: g|n \rightarrow g_0)$ be a redex in a graph g_0. The graphs g_1 (the build phase), g_2 (the redirection phase) and g_3 (the garbage collection phase) are defined as follows.

(i) The node-set N of g_1 is the disjoint union of N_{g0} and $N_{g|n'} - N_{g|n}$. The root is r_{g0}. The functions lab_{g1} and $succ_{g1}$ are given by:

$lab_{g1}(m)$	=	$lab_{g0}(m)$	(m in N_{g0})
	=	$lab_g(m)$	(otherwise - i.e. when m is in $N_{g\|n'} - N_{g\|n}$)
$succ_{g1}(m)_i$	=	$succ_{g0}(m)_i$	(m in N_{g0})
	=	$succ_g(m)_i$	(both m and $succ_g(m)_i$ in $N_{g\|n'} - N_{g\|n}$)
	=	$f(succ_g(m)_i)$	(m in $N_{g\|n'} - N_{g\|n}$, $succ_g(m)_i$ in $N_{g\|n}$)

We write $g_1 = g_0 +_f (g,n,n')$.

(ii) The next step is to replace in g_1 all references to f(n) by references to n'. We can define a substitution operation in general for any term graph h and any two nodes a and b of h.

h[a:=b] is a term graph $(N_h,lab,succ,r)$, where lab, succ, and r are given as follows.

$lab(c) = lab_h(c)$	for each node c of N_h
$succ(c)_i = b$	if $succ_h(c)_i = a$
$succ(c)_i = succ_h(c)_i$	otherwise
$r = b$	if $r_h = a$
$r = r_h$	otherwise

With this definition, g_2 is $g_1[f(n):=n']$.

(iii) Finally, we take the part of g_2 which is accessible from its root, by defining $g_3 = g_2|r_{g2}$. We give this operation a name: for any term graph h, $GC(h) = h|r_h$.

We denote the result of reducing a redex Δ in a graph g by $RED(\Delta,g)$. Collecting the notations we have introduced, we have

$$RED(((g,n,n'),f),g_0) = GC((g_0 +_f (g,n,n'))[f(n):=n']).$$

Our definition of graph rewriting is a special case of a more general notion, defined in [UScat] by a category-theoretic construction. Those familiar with category theory may recognise the build phase of a reduction as a pushout, and redirection and garbage collection can be given definitions in the same style (though the categories involved are not those defined in this paper). For the purpose of this paper - describing graph rewritings which correspond to conventional term rewritings - the direct "operational" definition we have given is simpler.

3.7 DEFINITION. (i) If g reduces to g' by reduction of a redex Δ, we write $g \rightarrow^{\Delta} g'$, or $g \rightarrow g'$ if we do not wish to indicate the identity of the redex. The reflexive and transitive closure of the relation $\rightarrow$ is $\rightarrow^*$.

(ii) A *graph reduction system* (GRS) over F consists of a pair (G,**R**) where **R** is a set of rewrite rules and G is a set of graphs over F closed under reduction by the members of **R**.

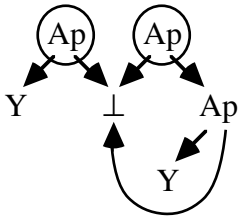
(iii) We write $g \rightarrow_{\mathbf{R}} g'$ if $g \rightarrow g'$ by reduction of a redex using one of the rules in **R**. The reflexive and transitive closure of $\rightarrow_{\mathbf{R}}$ is $\rightarrow^*_{\mathbf{R}}$. If clear from the context, we omit the subscript **R**.

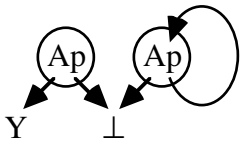
(iv) A graph g such that for no g' does one have $g \rightarrow_{\mathbf{R}} g'$ is said to be an **R**-normal form (or to be in R-normal form). If $g \rightarrow^*_{\mathbf{R}} g'$ and g' is in **R**-normal form, we say that g' is an **R**-normal form of g, and that g has an **R**-normal form. Again, we often omit the **R**.

Note that a GRS is not required to include all the graphs which can be formed from the given set of function symbols F. Any subset closed under reduction will do. This allows our definition to automatically handle such things as, for example, sorted rewrite systems, where there are constraints over what function symbols can be applied to what arguments, or arities, where each function symbol may only be applied to a specified number of arguments. From our point of view, this amounts to simply restricting the set of graphs to those satisfying these constraints. So long as reduction always yields allowed graphs from allowed graphs, we do not need to develop any special formalism for handling restricted rewrite systems, nor do we need to prove new versions of our results.

Our definition of a graph rewrite rule allows any conventional term rewrite rule to be interpreted as a graph rewrite rule, provided that the term rewrite rule is left-linear, that is, if no variable occurs twice or more on its left-hand side. However, some new phenomena arise with graph rewrite rules.

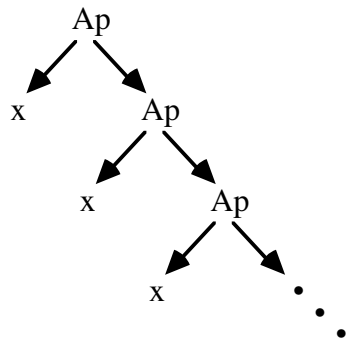
3.8 EXAMPLES. (i) The fixed point combinator Y can be given the term rewrite rule $Y\ x \rightarrow x\ (Y\ x)$. This can be transformed into the graph rewrite rule (n:Ap(Y,x:⊥) + n':Ap(x,Ap(Y,x)),n,n'). However, it can also be given the graph rewrite rule: (n:Ap(Y,x:⊥) + n':Ap(x,n'),n,n'):





(n:Ap(Y,x:⊥) + n':Ap(x,Ap(Y,x)),n,n') (n:Ap(Y,x:⊥) + n':Ap(x,n'),n,n')

This captures the fact that the Böhm tree of the term Ap(Y,x) is as follows.



The graph rule can do all the "unravelling" in one step, which in the term rewrite world requires an infinite sequence of reductions.

(ii) A more subtle example is the following. Consider the term rewrite rule

$$F(Cons(x,y)) \rightarrow G(Cons(x,y))$$

Our standard representation of this as a graph rewrite rule is:

$$(n{:}F(Cons(x{:}\perp,y{:}\perp)) + n'{:}G(Cons(x,y)), n, n')$$

Note that each application of this rule will create a new node of the form Cons(...,...), which will have the same successors as an existing node Cons (...,...). In a practical implementation, there is no need to do this. One might as well use that existing node, instead of making a copy of it. The following graph rewrite rule does this:

$$(n{:}F(z{:}Cons(\perp,\perp)) + n'{:}G(z), n, n')$$

Both the languages Standard ML and Hope, which are languages of term rewriting, allow an enhanced form of term rewrite rules such as:

$$F(z{:}Cons(x,y)) \rightarrow G(z)$$

with precisely this effect. Of course, given referential transparency (which ML lacks) there is no reason for an implementation not to make this optimisation wherever there is an opportunity, even if the programmer does not. But providing this feature to the programmer may make his programs more readable.

(iii) Term rule: $A(x) \rightarrow B(x)$

Graph rule: $(n{:}A(x{:}\perp) + n'{:}B(x), n, n')$

Graph: x:A(x)

Result of reduction: x:B(x)

(iv) Term rule: $I(x) \rightarrow x$

Graph rule:$(n{:}I(n'{:}\perp), n, n')$

Graph: I(3)

Result of reduction: 3

(v) Term rule: $I(x) \rightarrow x$

Graph rule:(n:I(n':⊥), n, n')

Graph: x:I(x)

Result of reduction: x:I(x)

The last of these rules is the rule for the I combinator. Consider the GRS for combinatory logic, whose rules are those for the S, K, and I combinators. The graph of example (v) can be interpreted as "the least fixed point of I" (cf. the example of the Y combinator above), and in the usual denotational semantics in terms of reflexive domains should have the bottom, "undefined" value. As g reduces to itself (and to nothing else), g has no normal form. Thus our operational semantics of graph rewriting agrees with the denotational semantics. This is not true for some other attempts we have seen at describing formally the graphical implementation of term rewriting, which either exclude cyclic graphs altogether as in Raoult [1984], exclude rules for which the right-hand side is a variable (again in Raoult [1984]), or accept these but fail to reduce example 3.8 (v).

We now study some properties of graph rewrite systems. We establish a version of the theorem of finite developments for term rewriting, and the confluence of weakly regular systems. We begin with a technical lemma relating the various graphical constructions which make up the rewriting operation.

3.9 LEMMA. *(i) Let (R_i,f_i) $(i = 1,2)$ be two redexes in a graph g.Then*

$$g +_{f1} R_1 +_{f2} R_2 = g +_{f2} R_2 +_{f1} R_1.$$

(ii) Let n_1, n_2, n'_1, and n'_2 be nodes of g such that $n_1 \neq n_2$, $n_1 \neq n'_2$, and $n_2 \neq n'_1$. Then

$$g[n_1:=n'_1][n_2:=n'_2] = g[n_2:=n'_2][n_1:=n'_1].$$

(iii) $GC(GC(g)) = GC(g)$.

(iv) For any node n of g we have $GC(g|n) = g|n$.

(v) If n is a node of g not in GC(g), then $GC(g[n:=n']) = GC(g)$.

(vi) If n and n' are nodes of $GC(g) +_f R$, then

$$GC((GC(g) +_f R)[n:=n']) = GC((g +_f R)[n:=n']).$$

(vii) Let (R,f), where R = (g',m,m'), be a redex in a graph g. Let n and n' be nodes of g such that n is not equal to f(n") for any nonempty node n" of g'. Let f[n:=n'] be the composition of f with the function which takes n to n' and is the identity elsewhere. Then f[n:=n'] is a homomorphism and

$$g[n:=n'] +_{f[n:=n']} R = (g +_f R)[n:=n']$$

(viii) Let (R,f) be a redex in a graph g. $GC(g +_f R) = GC(g)$.

PROOF. (i) Obvious.

(ii) Obvious. The stipulated inequalities prevent clashes between the two substitutions.

(iii) Every node of GC(g) is by definition accessible from the root. Hence a second garbage collection can have no effect.

(iv) Every node of g|n is accessible from the root n, so garbage collection has no effect.

(v) g[n:=n'] differs from g only in the value of succ for nodes which in g have n as a successor. If n is garbage, all nodes pointing to n are garbage. Therefore the graph which remains after garbage collection on g[n:=n'] is just GC(g).

(vi) $(GC(g) +_f R)[n:=n']$ is clearly a subgraph of $(g +_f R)[n:=n']$. To establish their equality, it is sufficient to prove that every path of $(g +_f R)[n:=n']$ beginning at the root is also a path of $(GC(g) +_f R)[n:=n']$ beginning at the root. First note that if there is an edge from a node m to a node m' in $(g +_f R)[n:=n']$ which is not an edge of $(GC(g) +_f R)[n:=n']$, then m cannot be a node of $(GC(g) +_f R)[n:=n']$. Now consider the shortest path from the root of $(g +_f R)[n:=n']$ which is not a path of $(GC(g) +_f R)[n:=n']$. There must be such a path, since the path which consists of just the root of $(g +_f R)[n:=n']$ is a path of $(GC(g) +_f R)[n:=n']$, which has the same root. Let the last two nodes in this path be m and m', m' being a successor of m. The edge from m to m' is not in $(GC(g) +_f R)[n:=n']$. Therefore m is not in $(GC(g) +_f R)[n:=n']$, contradiction.

(vii) f[n:=n'](n") can differ from f(n") only if n" is an empty node of g'. Thus if f is a homomorphism, then so is f[n:=n']. (R,f[n:=n']) is therefore a redex of g[n:=n'], and the equality is obvious.

(viii) Since the successor function of $g +_f R$, restricted to nodes of g, agrees with $succ_g$, none of the nodes added to g to make $g +_f R$ can be accessible from the root of g. Therefore
$GC(g +_f R) = GC(g)$. ⊠

3.10 PROPOSITION. *Garbage collection can be postponed. That is, given* $g \rightarrow^{\Delta 1} g_1 \rightarrow^{\Delta 2} g_2$, $\Delta_i = (R_i,f_i)$, $R_i = (g_i,n_i,n'_i)$ $(i = 1,2)$ *and* $g'_1 = (g +_{f1} R_1)[f(n) = n']$, *then* Δ_2 *is also a redex of* g'_1, *and* $g'_1 \rightarrow^{\Delta 2} g_2$.

PROOF. It is clear that Δ_2 is a redex of g'_1.

$$\begin{aligned} RED(g'_1,\Delta_2) &= GC((g'_1 +_{f2} R_2)[f_2(n_2):=n'_2]) \\ &= GC((GC(g'_1) +_{f2} R_2)[f_2(n_2):=n'_2]), \qquad \text{by lemma 3.7(vi)} \\ &= RED(g_1,\Delta_2). \end{aligned}$$ ⊠

3.11 DEFINITION. Two redexes $\Delta_1 = ((g_1,n_1,n'_1),f_1)$ and $\Delta_2 = ((g_2,n_2,n'_2),f_2)$ of a graph g are *disjoint* if:

(i) $f_2(n_2)$ is not equal to $f_1(n)$ for any nonempty node n of $g_1|n_1$, and

(ii) $f_1(n_1)$ is not equal to $f_2(n)$ for any nonempty node n of $g_2|n_2$.

Δ_1 and Δ_2 are *weakly disjoint* if either they are disjoint, or the only violation of conditions (i) and (ii) is that $f_1(n_1) = f_2(n_2)$, and the results of reducing Δ_1 or Δ_2 are identical.

A GRS is *regular* (resp. *weakly regular*) if for every graph g of the GRS, every two distinct redexes in g are disjoint (resp. weakly disjoint).

3.12 PROPOSITION. *Let $\Delta_1 = ((g_1,n_1,n'_1),f_1)$ and $\Delta_2 = ((g_2,n_2,n'_2),f_2)$ be two disjoint redexes of a graph g. Let $g \rightarrow^{\Delta 1} g'$. Then either $f_2(n_2)$ is not a node of g', or there is a redex $((g_2,n_2,n'_2),f)$ of g' such that $f(n_2) = f_2(n_2)$.*

PROOF. The build phase of the reduction of Δ_1 cannot affect the existence of the redex Δ_2. The redirection phase replaces references to $f_1(n_1)$ by references to another node. This can only destroy Δ_2 if $f_1(n_1)$ is equal to $f_2(n)$ for some non-empty node n of $g_2|n_2$. Since Δ_1 and Δ_2 are disjoint, this cannot happen. After the redirection, there will be a redex $((g_2,n_2,n'_2),f)$, where $f = f_2[f_1(n_1):=n']$. If the garbage collection phase does not remove $f_2(n_2)$, then all the nodes of $f(g_2)$ will still be present, and $((g_2,n_2,n'_2),f)$ will be a redex of g'. ◘

3.13 DEFINITION. (i) With the notations of the preceding proposition, if $f_2(n_2)$ is not a node of g' then Δ_2/Δ_1 is the empty reduction sequence from g' to g'; otherwise, Δ_2/Δ_1 is the one-step reduction sequence consisting of the reduction of $((g_2,n_2,n'_2),f)$. This redex is the *residual* of Δ_2 by Δ_1 and is denoted by $\Delta_2//\Delta_1$. Δ_2/Δ_1 and $\Delta_2//\Delta_1$ are not defined when Δ_1 and Δ_2 are not disjoint.

(ii) Given a reduction sequence $g \rightarrow^{\Delta 1} \rightarrow^{\Delta 2} ... \rightarrow^{\Delta i} g'$ and a redex Δ of g, the *residual* of Δ by the sequence $\Delta_1...\Delta_i$, denoted $\Delta//(\Delta_1...\Delta_i)$ is $(\Delta//(\Delta_1...\Delta_{i-1}))//\Delta_i$ (provided that $(\Delta//(\Delta_1...\Delta_{i-1}))$ exists and is disjoint from Δ_i).

For weakly disjoint but not disjoint redexes Δ_1 and Δ_2, Δ_1/Δ_2 and Δ_2/Δ_1 both denote the empty reduction sequence from g' to g'.

3.14 PROPOSITION. *Let Δ_1 and Δ_2 be weakly disjoint redexes of g, and let $g \rightarrow^{\Delta i} g_i$ (i = 1,2). Then there is a graph h such that $g_1 \rightarrow^{\Delta 2/\Delta 1} h$ and $g_2 \rightarrow^{\Delta 1/\Delta 2} h$. That is, weakly disjoint redexes are subcommutative.*

PROOF. For the case where Δ_1 and Δ_2 are weakly disjoint but not disjoint, this is trivial ($h = g_1 = g_2$). Otherwise, define

$h = GC((g +_{f1} R_1 +_{f2} R_2) [f_1(n_1) := n'_1] [f_2(n_2) := n'_2])$.

Let $f'_2 = f_2[f_1(n_1) := n'_1]$. Then

$h \approx GC((((g +_{f1} R_1) [f_1(n_1) := n'_1]) +_{f'2} R_2) [f'_2(n_2) := n'_2])$, by lemma 3.9(vii)).

Δ_2/Δ_1 is nonempty whenever $f'_2(n_2)$ is a node of $g_1 = GC((g +_{f1} R_1) [f_1(n_1) := n'_1])$. By lemma 3.9(vi) we have $h \approx GC((g_1 +_{f'2} R_2) [f'_2(n_2) := n'_2]) = RED(g_1,\Delta_2//\Delta_1)$. When $\Delta_2//\Delta_1$ is empty, h is by lemma 3.9(v,viii) equal to $GC((g +_{f1} R_1) [f_1(n_1) := n'_1])$, which is just g_1. So in both cases, $g_1 \rightarrow^{\Delta 2/\Delta 1} h$.

Since the definition of h is symmetric between 1 and 2 (using lemma 3.9(i,ii)), we also have $g_2 \rightarrow^{\Delta 1/\Delta 2} h$. ⊠

3.15 COROLLARY. *Every weakly regular GRS is confluent. That is, if $g \rightarrow^* g_i$ (i = 1,2), then there is an h such that $g_i \rightarrow^* h$ (i = 1,2).*
PROOF. By induction on the lengths of the given sequences, using the preceding proposition. ⊠

3.16 DEFINITION. Let g be a graph and F be a set of disjoint redexes of g. A *development* of F is a reduction sequence in which the redex reduced at each step is a residual, by the preceding steps of the sequence, of a member of F. A *complete development* of F is a development of F which is not an initial segment of a longer development of F.

3.17 PROPOSITION. *Every complete development of a finite set of pairwise disjoint redexes F is finite. In fact, its length is bounded by the number of redexes in F.*
PROOF. By induction from the subcommutativity of disjoint redexes. ⊠

3.18 PROPOSITION. *Let F be a set of redexes of a graph g. Every finite complete development of F ends with the same graph (up to isomorphism). This graph is:*

$$GC((g +_{f1} R_1 +_{f2} \ldots +_{fi} R_i)[f_1(n_1) := n'_1]\ldots[f_i(n_i):=n'_i])$$

where the redexes whose residuals are reduced in the complete development are $\Delta_1 = (f_1,R_1),\ldots,\Delta_i = (f_i,R_i)$.
PROOF. By induction from the subcommutativity of disjoint redexes. ⊠

REMARK. Note that since we allow infinite graphs, a set of redexes F as in the last two propositions may be infinite. Nevertheless, it may have a finite complete development, if reduction of some members of F causes all but finitely many members of F to be erased.

4. Tree reduction.

In order to study the relationship between term rewriting and graph rewriting, we define the notion of *tree rewriting*. This is a formalisation of conventional term rewriting within the framework of our definitions of graph rewriting.

4.1 DEFINITION. (i) A *tree rewrite rule* is a graph rewrite rule (g,n,n') such that g|n is a tree.

For a set of tree rewrite rules **R**, the relation $\rightarrow_{t\mathbf{R}}$ of *tree reduction* with respect to **R** is defined by:

$$t_1 \rightarrow_{t\mathbf{R}} t_2 \iff \text{for some graph g, } t_1 \rightarrow_{\mathbf{R}} g \text{ and } U(g) = t_2$$

(ii) A *tree rewrite system* (TreeRS) over F is a pair (T,R) where R is a set of tree rewrite rules and T is a set of trees over F closed under $\rightarrow_{t\mathbf{R}}$. A *term rewrite system* (TRS) is a TreeRS, all of whose trees are finite.

When t_1 reduces to t_2 by tree reduction of a redex Δ, we write $t_1 \rightarrow_t^{\Delta} t_2$, or $t_1 \rightarrow_t t_2$ when we do not wish to indicate the identity of the redex.

Tree rewrite systems differ from conventional term rewrite systems in two ways. Firstly, infinite trees are allowed. We need to handle infinite trees, since they are produced by the unravelling of cyclic graphs. We need to handle cyclic graphs because some implementors of graph reduction use them, and we do not want to limit the scope of this paper unnecessarily. Secondly, the set of trees of a TreeRS may be any set of trees over the given function sumbols which is closed under tree reduction. This is for the same reason as was explained above for GRSs.

If for each rule (g,n,n') in the rule-set, g is finite and acyclic, the set of all finite trees generated by the function symbols will be closed under tree reduction. This is true for those rules resulting from term rewrite rules by our standard representation. Thus the conventional notion of a TRS is included in ours.

4.2 DEFINITION. Let t, t_1,... t_i be trees, and n_1,...,n_i be distinct nodes of t. We define $t[n_1:=t_1,...,n_i:=t_i]$ to be the tree whose nodes satisfy

(1) all paths of t not including any of n_1,...,n_i, and

(2) every $(m_0,i_1,m_1,..,i_{j-1},m'_0,i'_1,m'_1,...,i'_{j'-1},m'_j)$, where $(m_0,i_1,m_1,..,i_{j-1},n_k)$ is a path of t not including any of n_1,...,n_i, except for the explicit occurrence of n_k, and $(m'_0,i'_1,m'_1,...,i'_{j'-1},m'_j)$ is a path of t_k.

For any of these paths p, the label of p is the label of the last node in p, in whichever of t, t_1,... t_i it came from. The successors function is defined similarly.

4.3 PROPOSITION. *Let n and n' be nodes of g, $u_g^{-1}(n) = \{n_1,...,n_i\}$, and $t = U(g|n')$. Then*

$U(g[n:=n']) \approx U(g)[n_1:=t,...,n_i:=t]$.

PROOF. Immediate from the definition of U. ⊠

The results concerning disjointness, regularity, and confluence which we proved for graph rewriting all have versions for tree rewriting. We shall not give the proofs, as they are much the same as for the

earlier version. The following technical lemma is useful, showing how unravelling relates to the other constructions.

4.4 LEMMA.*(i) For any tree t, U(t) ≈ t.*

(ii) U(g[n:=n']) ≈ U(U(g)[n_1:=t,...,n_i:=t]) where $u_g^{-1}(n) = \{n_1,...,n_i\}$ and t = U(g|n').

(iii) U(GC(g)) = U(g) = GC(U(g)).

(iv) U(g +$_f$ R) = U(g)

(v) *Let $t \to^{\Delta}_t t'$ where $\Delta = (R,f)$, $R = (g,n,n')$. Then $t' = t[f(n):=U(t +_f R \mid n')]$. If Δ_1, ..., Δ_i are disjoint redexes of t, with $\Delta_j = (R_j,f_j)$ (j:1..i), and $t \to^*_t t'$ by their complete development, then $t' = t[f_1(n):=U(t +_{f1} R_1 \mid n'_1),...,f_i(n):=U(t +_{fi} R_i \mid n'_i)]$.*

PROOF. (i), (ii), (iii), (iv) Obvious. (v) A simple computation shows

t'	= U(GC((t +$_f$ R)[f(n):=n'])),	by definition
	= U((t +$_f$ R)[f(n):=n']),	by (iii)
	= U(t +$_f$ R)[f(n):=U(t +$_f$ R | n')],	by (ii)
	= U(t)[f(n):=U(t +$_f$ R | n')],	by (iv)
	= t[f(n):=U(t +$_f$ R | n')],	by (i).

The case of i disjoint redexes is similar. ⊠

4.5 PROPOSITION. (Unravelling can be postponed). *If $t_1 \to_t t_2 \to_t t_3$, then there are graphs g and g' such that*

(1) t_2 = U(g) and $t_1 \to g$ (by graph reduction)

(2) $g \to g'$ and $t_3 \to^*_t U(g')$. ⊠

5. Relations between tree and graph reduction.

In this section we prove our principal result: for weakly regular rule-systems, graph reduction is a sound and complete implementation of term reduction.

5.1 DEFINITION. Let (T,**R**) be a TreeRS.

(i) L(T,**R**), the *lifting* of this system, is the GRS whose set of graphs is L(T) = {g | U(g)∈ T}, and whose rule set is **R** (but now interpreted as graph reduction rules). It is trivial to verify that L(T) is closed under $\to_{\mathbf{R}}$.

(ii) A *graphical term rewrite system* (GTRS) is a GRS of the form L(T,**R**), where (T,**R**) is a term rewrite system.

(iii) A GRS (G,**R**) is *acyclic* if every member of G is acyclic.

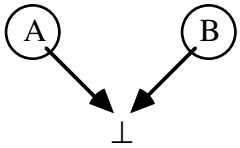
When (T,**R**) is a term rewrite system, L(T,**R**) represents its graphical implementation. There are two fundamental properties it must have to be a correct implementation, which we now define.

5.2 DEFINITION. (i) A TreeRS (T,**R**) is called *graph-reducible* if for every graph g in L(T), if t is a normal form of U(g) in (T,**R**), then there is a normal form g' of g in L(T,**R**) such that U(g') = t, and if U(g) has no normal form in (T,**R**), then g has no normal form in L(T,**R**).

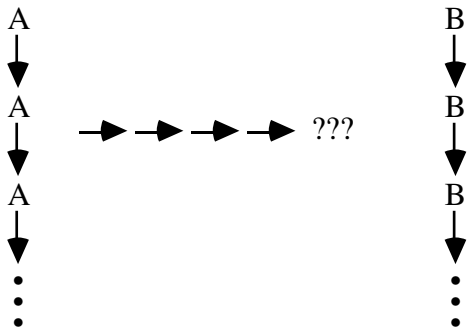
(ii) A GRS (G,**R**) is *tree-reducible* if there is a TreeRS (T,**R**) such that (G,**R**) = L(T,**R**), and such that if g' is a normal form of g in (G,**R**), then U(g') is a normal form of U(g) in (T,R), and if g has no normal form in (G,**R**), then U(g) has no normal form in (T,**R**).

L(T,**R**) is the graphical implementation of (T,**R**). Tree-reducibility of L(T,**R**) expresses soundness: every result which is obtainable by graph rewriting in L(T,**R**) is also obtainable by tree rewriting in (T,**R**). Graph-reducibility of (T,**R**) expresses completeness: every result which is obtainable by tree rewriting is also obtainable by graph rewriting. We shall see that every GTRS is tree-reducible, and every weakly regular TRS is graph-reducible. Not all GRSs, even those of the form L(T,**R**), are tree-reducible, nor is every TreeRS graph-reducible, as the following examples show.

5.3 EXAMPLE. Tree reducibility can fail when there are cyclic graphs. Consider the term rewrite rule $A(x) \to B(x)$, represented graphically by:



A cyclic graph may contain a single redex with respect to this rule, while its unravelling contains infinitely many:

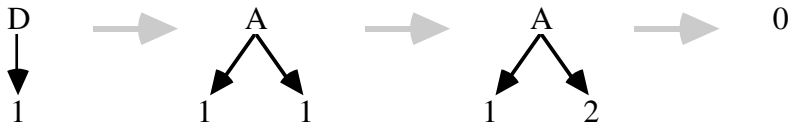


5.4 EXAMPLE. The following TreeRS is not graph-reducible:

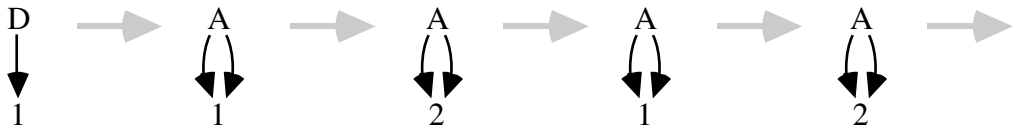
T: trees over {A,D,0,1,2}, with the following arities: A is binary, D is unary, and 0, 1, and 2 are nullary.

R: $A(1,2) \to 0$; $1 \to 2$; $2 \to 1$; $D(x) \to A(x,x)$.

For a counterexample, consider the following tree reduction sequence:



In the graph reduction system we have:



In this example, the sharing of (tree) subterms in the graph world has excluded certain reduction sequences in the tree world. Distinct subterms of A(1,1) correspond to the same subgraph of A(x:1,x), forcing "synchronized reduction of siblings", which makes the normal form inaccessible.

5.5 DEFINITION. (i) Redexes $\Delta = ((g,n,n'),f)$ and $\Delta' = ((g',m,m'),f')$ in a graph h are *siblings* if $h|f(n) \approx_t h|f'(m)$.

(ii) For a redex $\Delta = ((U(g),n,n'),f)$ of a tree U(g) we define $u_g(\Delta)$ to be the redex $((g,n,n'),u_g \cdot f)$ of g.

(iii) For a redex Δ of a graph g, the set of redexes Δ' of U(g) such that $u_g(\Delta') = \Delta$ is denoted by $u_g^{-1}(\Delta)$. For a set of redexes F of a graph g, $u_g^{-1}(\mathsf{F})$ denotes $\bigcup\{ u_g^{-1}(\Delta') \mid \Delta' \in \mathsf{F} \}$.

(iv) A redex Δ of a graph G is *acyclic* if $u_g^{-1}(\Delta)$ is finite.

5.6 PROPOSITION. *Let $g \to g'$ by reduction of an acyclic redex Δ. Then $U(g) \to^*_t U(g')$ by complete development of $u_g^{-1}(\Delta)$. For any redex Δ' of g, weakly disjoint from Δ, $u_g^{-1}(\Delta'//\Delta) = u_g^{-1}(\Delta')//u_g^{-1}(\Delta)$.*
PROOF. Δ being acyclic, $u_g^{-1}(\Delta)$ is a finite set $\{\Delta_1,...,\Delta_i\}$. Let $U(g) \to^*_t t$ by complete development of this set. Let $\Delta = (R,f)$, $R = (h,n,n')$, $u_g^{-1}(f(n)) = \{n_1,...,n_i\}$, $t' = U((g +_f R)|n')$.

$$\begin{aligned} U(g') &= U(GC((g +_f R)[h(n):=n'])) \\ &= U((g +_f R)[f(n):=n']) \\ &= U(g +_f R)[n_1:=t',...,n_i:=t'] \\ &= U(g)[n_1:=t',...,n_i:=t'] \end{aligned}$$

This is the result of a complete development of the redexes $u_g^{-1}(\Delta)$. The statement about $u_g^{-1}(\Delta'//\Delta)$ is obvious. ⊠

5.7 PROPOSITION. *Let $g \to^* g'$ by a complete development of a set F of disjoint acyclic redexes of g whose associated rewrite rules are acyclic. Then $U(g) \to^*_t U(g')$ by a complete development of $u_g^{-1}(\mathsf{F})$.*
PROOF. Let $\Delta_1,...,\Delta_i$ be a listing of the members of F whose residuals are reduced in the sequence $g \to^* g'$, in the order in which they are reduced. We proceed by induction on i. For i = 0 it is trivial, and for i = 1 this is the preceding proposition. Assume i > 1. Let

$$g = g_0 \to^{\Delta 1} g_1 \to^{\Delta' 2} ... \to^{\Delta' i} g_i = g'$$

where $\Delta'_i = \Delta_i//(\Delta_1\Delta'_2...\Delta'_{i-1})$. By the preceding proposition, $U(g) \to^*_t U(g_1)$ by a complete development σ_1 of $u_g^{-1}(\Delta_1)$. $g_1 \to^* g'$ by complete development of $\{\Delta_2//\Delta_1,...,\Delta_{i-1}//\Delta_1\}$, so by induction, $U(g_1) \to^*_t U(g')$ by a complete development τ of $u_{g1}^{-1}(\{\Delta_2//\Delta_1,...,\Delta_{i-1}//\Delta_1\})$. But

$$u_{g1}^{-1}(\Delta_2//\Delta_1,...,\Delta_{i-1}//\Delta_1\}) = \bigcup\{ u_g^{-1}(\{\Delta_2,...,\Delta_i\})/\sigma_1 \mid k:2..i \}$$

Thus $\sigma_1 \cdot \tau$ is a complete development of $u_g^{-1}(\mathsf{F})$.

The converse follows from the fact that all finite complete developments of a set of disjoint redexes in a GRS or TreeRS end with the same graph or tree. ⊠

5.8 DEFINITION. (i) In a weakly regular GRS, the relation of *Gross-Knuth* reduction, notation $\to^{GK}$, is defined as follows

$g \rightarrow^{GK} g' \Leftrightarrow g \rightarrow^{*} g'$ by complete development of the set of all redexes of g.

(ii) In a weakly regular TreeRS we define *Gross-Knuth* reduction by

$t \rightarrow^{GK}_{t} t' \Leftrightarrow t \rightarrow^{*}_{t} t'$ by complete development of the set of all redexes of t.

5.9 PROPOSITION. *Let (T,**R**) be a weakly regular TRS. Then L(T,**R**) is weakly regular. Let g and g' be graphs in L(T) such that* $g \rightarrow^{GK} g'$. *Then* $U(g) \rightarrow^{GK}_{t} U(g')$.

PROOF. Immediate from the preceding proposition. ⊠

5.10 PROPOSITION. *If every graph in L(T) is acyclic, then L(T,**R**) is tree-reducible. In particular, a graphical term rewrite system is tree-reducible.*

5.11 PROPOSITION. *For any TreeRS (T,**R**) and any graph g in L(T), g is a normal form of L(T,**R**) iff U(g) is a normal form of (T,**R**).*

PROOF. Clearly, for any redex Δ of U(g), $u_g(\Delta)$ is a redex of g. Conversely, for any redex Δ of g, $u_g^{-1}(R)$ is nonempty. ⊠

Thus in a graphical term rewrite system L(T,**R**), everything which can happen can also happen in the term rewrite system, and all the normal forms are the same. Graph-reducibility may fail, however, since it may be that for some graph g, U(g) has a normal form but g does not.

5.12 THEOREM. *Every weakly regular TRS is graph-reducible.*

PROOF. Let (T,**R**) be a weakly regular TRS. Let g be a graph of L(T,**R**) such that U(g) has a normal form. Proposition 5.7 relates the Gross-Knuth reduction sequences for g and U(g) in the following way.

$$g \xrightarrow{GK} g_1 \xrightarrow{GK} g_2 \xrightarrow{GK} \cdots$$

$$U(g) \xrightarrow[t]{GK} U(g_1) \xrightarrow[t]{GK} U(g_2) \xrightarrow[t]{GK} \cdots$$

It is a standard result that for regular TRSs, Gross-Knuth reduction is normalising [Klop], and the proof carries over immediately to weakly regular TreeRSs. Therefore the bottom line of the diagram terminates with some tree U(g') in normal form such that g reduces to g' in L(T,**R**). Therefore g' is a normal form of g, and (T,**R**) is graph-reducible. ⊠

5.13 NON-LEFT-LINEARITY.

We shall now discuss non-left-linearity, and indicate why we excluded non-left-linear TRSs from consideration. In term rewriting theory, for a term to match a non-linear left-hand side, the subterms corresponding to all the occurrences of a repeated variable must be identical. This test of textual equality in itself appears to be too impractical to implement, and not theoretically useful.

Our method of using empty nodes to represent the variables of term rewrite rules suggests a very different semantics for non-left-linear rules. Our representation of a term rule $A(x,x) \rightarrow B$ would be (n:A(x:⊥,x), n, x). This will only match a subgraph of the form a:A(b:...,b). That is, the subgraphs matched by the repeated variable must be not merely textually equal, but identical - the very same nodes. If one is implementing graph rewriting as a computational mechanism in its own right, rather than considering it merely as an optimisation of term rewriting, then this form of non-left-linearity may be useful, However, it is not the same as non-left-linearity for term rules.

To introduce a concept more akin to the non-left-linearity of term rules, we could use variables in graphs, just as for terms, instead of empty nodes. A meaning must then be chosen for the matching of a graph A(Var1,Var1) where Var1 is a variable symbol, occurring at two different nodes. Two possibilities naturally suggest themselves. The graphs depending from nodes matched by the same variable may be required to be equivalent, or they may only be required to be tree-equivalent. The latter definition is closer to the term rewriting concept.

When a variable occurs more than once on the left-hand side of a rule, there is also the difficulty of deciding which of the subgraphs matched by it is referred to by its occurrences on the right-hand side. One method would be to cause those subgraphs to be first coalesced, replacing the isomorphism detected by the matching by pointer equality. This is similar to an implementation technique for logic programming. Further investigation of the matter lies outside the scope of the present paper.

Lastly, we note that although some term rewriting languages like SASL (Turner [1979]) and Miranda (Turner [1986]) allow non-left-linear rules, they interpret the implied equality test neither as textual equality, nor as pointer equality, but as the equality operator of the language. In these languages, any program containing non-left-linear rules is equivaent to one which does not.

6. Normalising Strategies.

In this section we define the notion of an evaluation strategy in a general setting which includes term and graph rewrite systems. We then study the relationships between strategies for term rewrite systems and for the corresponding graph systems.

6.1 DEFINITION. (i) An *abstract reduction system* (*ARS*) is a pair $(O,\rightarrow)$, where O is a set of *objects* and $\rightarrow$ is a binary relation on O. This notion abstracts from term and graph reduction systems. The transitive reflexive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

(ii) An element x of an ARS is a *normal form* (nf) if for no y does one have $x\rightarrow y$.

(iii) An element x *has a normal form* if $x\rightarrow^* y$ and y is a normal form.

(iv) A *reduction sequence* of an ARS is a sequence $x_0\rightarrow x_1\rightarrow ...\rightarrow x_n$. The *length* of this sequence is n. A sequence of length 0 is *empty*.

6.2 DEFINITION. (i) Given an ARS $(O,\rightarrow)$, a *strategy* for this system is a function S which takes each $x\in O$ to a set S(x) of nonempty finite reduction sequences, each beginning with x. Note that S can be empty.

(ii) S is *deterministic* if, for all x, S(x) contains at most one element.

(iii) S is a *one step strategy* (or *1-strategy*) if for every x in O, every member of S(x) has length 1.

(iv) Write $x\rightarrow_S y$ if S(x) contains a reduction sequence from x to y. By abuse of notation, we may write $x\rightarrow_S y$ to denote some particular but unspecified member of S(x).

(v) An S-*sequence* is a reduction sequence of the form $x_0\rightarrow_S x_1 \rightarrow_S x_2 \rightarrow_S .....$

(vi) S is *normalising* if for all x having a normal form any sequence

$x_0\rightarrow_S x_1 \rightarrow_S x_2 \rightarrow_S .....$

must eventually terminate with a normal form.

6.3 DEFINITION. (i) Let S be a strategy of an ARS $(O,\rightarrow)$. *quasi*-S is the strategy defined by:

quasi-S(x) = $\{x \rightarrow^* x' \rightarrow_S y \mid x' \text{ in } O\}$.

Thus a quasi-S path is an S-path diluted with arbitrary reduction steps.

(ii) A strategy S is *hypernormalising* if quasi-S is normalising.

A 1-strategy for a TreeRS or GRS can be specified as a function which takes the objects of the system to some subset of its redexes. This will be done from now on.

6.4 DEFINITION. Let S be a 1-strategy for a TreeRS $(T,\mathbf{R})$. The strategy S_L for the lifted graph reduction system $L(T,\mathbf{R})$ is defined by

$S_L(g) = u_g(S(U(g)))$.

Although a 1-strategy for a TreeRS may be normalising, its lifting may not be. This may be because the lifting of the TreeRS does not preserve normal forms (e.g. as in example 5.4), or for more subtle reasons, such as in the following example.

6.5 EXAMPLE. Introduce the following TreeRS

Function symbols: A (binary), B, 1, 2 (nullary).

Rules: $1 \to 2$, $2 \to 1$, $A(x,y) \to B$.

S is defined as follows (with the chosen redexes boldfaced):

$A(\mathbf{1},1) \to A(2,1)$

$A(2,\mathbf{2}) \to A(2,1)$

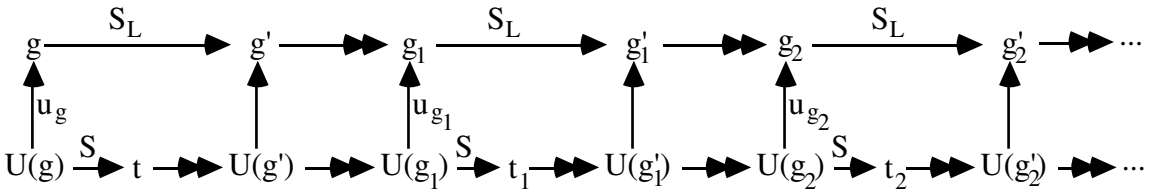
$\mathbf{A(x,y)} \to B$, if neither of the preceding cases applies

S takes the tree A(1,1) to normal form B in two steps. S_L takes the graph A(x:1,x) to A(x:2,x) and back again in an infinite loop.

The next theorem shows that if a strategy S for a TreeRS is hypernormalising, S_L is hypernormalising for the corresponding GRS.

6.6 THEOREM. *Let (T,**R**) be a TreeRS and let S be a 1-strategy for it. Let (G,**R'**) be the lifting of (T,**R**). If S is hypernormalising then S_L is hypernormalising.*

PROOF. Assume S is hypernormalising. Let g be a graph in G having a normal form, and consider a quasi-S_L reduction sequence starting from g. By proposition 5.7 and the definition of S_L, we can construct the following diagram, where the top line is the quasi-S_L reduction sequence:



Since g has a normal form, so does U(g), so since quasi-S is normalising, the bottom line must stop at some point, with a normal form of U(g). Therefore the top line also stops, and must do so with a graph which unravels to the normal form in the bottom line. ⚄

6.7 EXAMPLE. The converse does not hold. If S_L is hypernormalising, S need not be. Consider the following TRS.

Function symbols: A (binary), B (nullary)

Rules: $A(x,y) \to B$ $A(x,y) \to A(x,x)$

Every non-normal form of this system has the form $A(\alpha,\beta)$ for some terms α and β. Let S be the strategy:

$A(\alpha,\beta) \to_S A(\alpha,\alpha)$ (if $\alpha \neq \beta$)

$A(\alpha,\alpha) \to_S B$

The first S_L-step in any quasi-S_L-sequence will produce either a graph of the form $A(x:\alpha,x)$ or the normal form B. In the former case, whatever extra steps are then inserted, the result can only be either another term of the same form or B. In the former case, the next S_L-step will reach B. Therefore S_L is hypernormalising. However, S is not hypernormalising. A counterexample is provided by the term A(A(B,B),B). An infinite quasi-S sequence beginning with this term is:

$$A(A(B,B),B) \to_S A(A(B,B),A(B,B)) \to A(A(B,B),B) \to_S A(A(B,B),A(B,B)) \to \ldots$$

6.8 COROLLARY. *If a TreeRS (T,**R**) has a hypernormalising 1-strategy then it is graph reducible.*
PROOF. By theorem 6.5 the lifting (G,**R**) of the TreeRS has a normalising strategy. Now assume U(g) = t. Suppose g has no nf. Then the S_L path of g is infinite. This gives, by the construction of 6.6, an infinite quasi-S-path of t, hence t has no normal form. ⊡

An application of this result that strongly sequential TRSs in the sense of Huet and Lévy [1979] are graph reducible. Indeed they have shown that the strategy consisting of the contraction of needed redexes is hypernormalising.

The condition that a strategy be hypernormalising is unnecessarily strong. Inspection of the proofs of the preceding theorem and corollary shows that the following weaker concept suffices. 6.9 DEFINITION. (i) Let S be a 1-strategy of a TreeRS (T,**R**). Then *sib*-S is the strategy defined by:

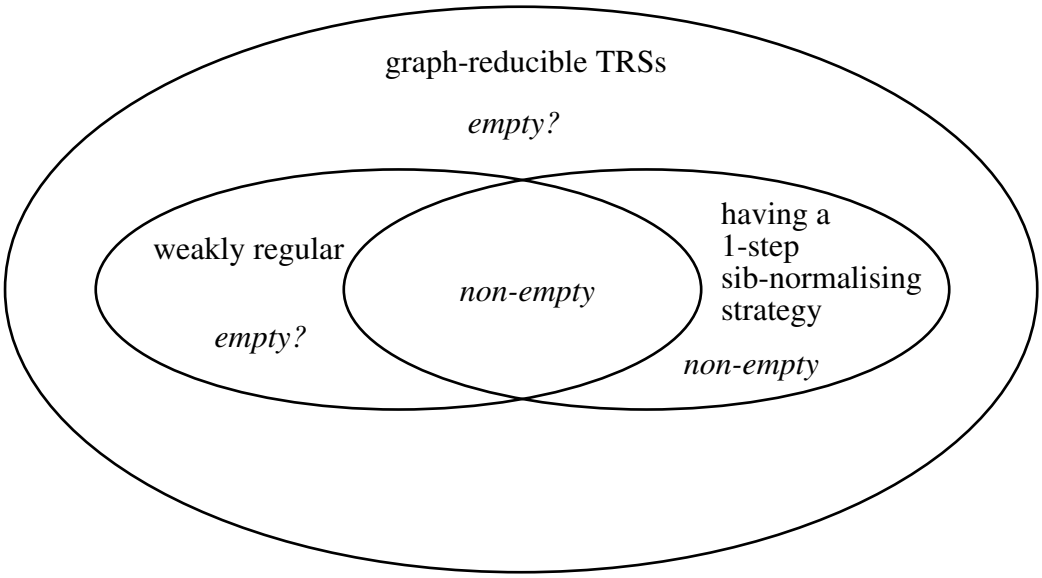
$$\text{sib-S}(x) = \{\, x \to_S y \to^* z \mid \text{the sequence } y \to^* z \text{ consists of siblings of } S(x)\}.$$

That is, a sib-S path is an S-path diluted with arbitrary sib-steps from the reduction relation.

(ii) A strategy S is *sib-normalising* if sib-S is normalising.

6.10 THEOREM. *Let (T,**R**) be a TreeRS and let S be a 1-strategy for it. Let (G,**R'**) be the lifting of (T,**R**). If S is sib-normalising then S_L is sib-normalising and (T,**R**) is graph-reducible.*
PROOF. Immediate from the proofs of theorem 6.6 and corollary 6.8. ⊡

Our results can be summarised in the following picture:



6.11 OPEN PROBLEM. We do not know if every weakly regular TRS has a sib-normalising 1-step strategy, nor, if so, whether such a strategy can always be found which is computable.

7. Conclusion.

Graph reduction is an efficient way to perform term reduction and is known to satisfy the Church-Rosser theorem. We have shown that

1. graph reduction does not give wrong results.
2. Graph reduction gives all correct results.
3. Many normalising strategies (the hypernormalising ones) on terms can be lifted to graphs to yield normalising strategies there. In particular for strongly sequential term rewrite systems the strategy consisting of contracting needed redexes, see Huet and Lévy [1979], can be lifted to graphs.

References.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep

[198-] LEAN, in preparation.

[198-] Category theoretic concepts of graph rewriting and garbage collection, in preparation.

van den Broek, P.M. and G.F. van der Hoeven

[1986] Combinatorgraph reduction and the Church-Rosser theorem, preprint INF-86-15, Department of Informatics, Twente University of Technology, P.O. Box 217, 7500AE Enschede, The Netherlands.

Ehrig, H.

[1979] Introduction to the algebraic theory of graph grammars, in: *Graph grammars and their Applications in Computer Science and Biology*, ed. V. Claus, H. Ehrig, and G.Rozenberg. Lecture notes in Computer Science 73, Springer, Berlin, 1-69.

Huet, G. and Lévy, J.J.

[1979] Call-by-need computations in non-ambiguous term rewriting systems, Report 359, IRIA-Laboria, B.P. 105, 78150 Le Chesney, France.

Kennaway, J.R.

[1984] An outline of some results of Staples on optimal reduction orders in replacement systems, Report CSA/19/1984, School of Information Systems, University of East Anglia, Norwich, England.

[198-] On "On graph rewritings", in preparation.

Klop, J.W.

[1980] *Combinatory Reduction Systems*, Mathematical Centre Tracts n.127, Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam.

Raoult, J.C.

[1984] On graph rewritings, *Theor. Comput. Sci.* 32, 1-24.

Peyton Jones, S.L.

[1987] *The Implementation of Functional Languages*, Prentice-Hall, London, to appear.

Staples, J.

[1980a] Computation on graph-like expressions, *Theor. Comput. Sci.* 10, 171-185.

[1980b] Optimal evaluations of graph-like expressions, *Theor. Comput. Sci.* 10, 297-316.

[1980c] Speeding up subtree replacement systems, *Theor. Comput. Sci.* 11, 39-47.

Turner, D.A.

[1979a] A new implementation technique for applicative languages, in: *Software: Practice and Experience* 9, 31-49.

[1979b] Another algorithm for bracket abstraction, J. Symbolic Logic 44(2), 267-270.

[1979c] SASL Language Manual, "combinators" version, University of St. Andrews, U.K.

[1986] Miranda System Manual, Research Software Ltd., 1986.

Wadsworth, C.P.

[1971] Semantics and Pragmatics of the Lambda Calculus, D.Phil. thesis, Programming Research Group, Oxford University.