# 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].

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

## Abstract

Graph rewriting (also called reduction) as defined in Wadsworth [1971] was introduced in order to be able to give a n efficient implementation of functional programming languages in the form of lambda calculus or term rewrite syste 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 gi mathematical descriptions of graph rewriting, usually employing concepts from category theory. These papers prove am other things the correctness of graph rewriting in the form of the Church-Rosser property for "well-behaved" (i.e. regu rewrite systems. However, only Staples has formally studied the soundness and completeness of graph rewriting with respe term rewriting.

In this paper we give a direct operational description of graph rewriting that avoids the category theoretic notions. 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 reduc the original term t (*soundness*). For weakly regular term rewrite systems, there is also a *completeness* result: every nor form of a term t can be obtained from the graphical implementation. We also show completeness for all term rewrite syst which possess a so called hypernormalising strategy, and in that case the strategy also gives a normalising strategy for graphical implementation.

Besides having nice theoretical properties, weakly regular systems offer opportunities for parallelism, since redexe different places can be executed independently or in parallel, without affecting the final result.

## 0. Introduction and background.

Graph rewriting is a well-known and standard technique for implementing functional languages ba on term rewriting (e.g. Turner [1979a]), but the correctness of this method has received little attenti being simply accepted folklore. For both theory and practice, this makes a poor foundation, especially in presence of parallelism. Staples [1980a,b,c] provides the only published results we are aware of. digested summary of these papers is in Kennaway [1984].) Wadsworth [1971] proves similar results the related subject of pure lambda calculus.

Our principal result is that the notion of graph rewriting provides a sound and complete representat (in a sense precisely defined below) of *weakly regular* TRSs. A counterexample is given to show that non-weakly regular TRSs completeness may fail: some term rewriting computations cannot be expresse the corresponding graph rewrite system. A second result concerns the mapping of evaluation strateg between the term and the graph worlds. A counterexample is exhibited to show that an evaluation strat which is normalising (i.e. computes normal forms) in the term world may fail to do so when it is transfer to the graph world. We prove that any strategy which satisfies a stronger condition of be *hypernormalising* in the term world is normalising (and indeed hypernormalising) in the graph world. briefly consider the problem of defining a graph rewriting implementation of non-left linear term rew rules.

The general plan of the paper is as follows: Section 1 presents basic definitions, and introduces a lin syntax for terms represented as graphs. Section 2 introduces a category of term graphs. Section 3 defi

the notion of graph rewriting, and section 4 introduces the notion of *tree rewriting* as a prelude to sectio which develops our theory of how to relate the worlds of term and graph rewriting. Section 6 considers 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 by:

$$A \in \mathbf{F},\ t_1,...,t_n \in T \ \Rightarrow\ A(t_1,...,t_n) \in T \qquad (n \geq 0)$$

A() is written as just A. n

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

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

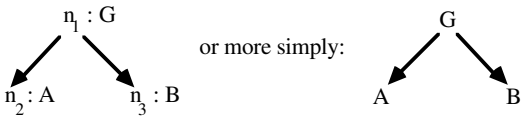
When we draw pictures of graphs, a directed edge will go from each node n to each node in succ with the left-to-right ordering of the sources of the edges corresponding to the ordering of the compone of succ(n). The identity of nodes is usually unimportant, and we may omit this information from picture

1.4 EXAMPLE. Let N={$n_1,n_2,n_3$} and define lab and succ on N as follows.

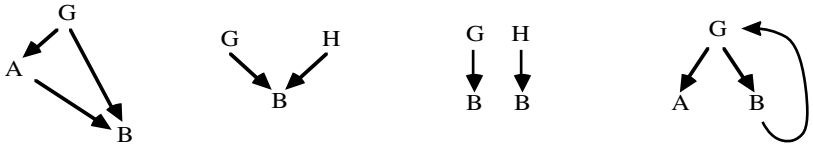
$lab(n_1)$=G, $lab(n_2)$=A, $lab(n_3)$=B,

$succ(n_1)=(n_2,n_3)$, $succ(n_2)$=(), $succ(n_3)$=().

This defines a labelled graph that can be drawn as:




Using this notation, four more examples of graphs are the following.




1.5 DEFINITION. (i) A *path* in a labelled graph (N,lab,succ) is a list ($n_0,i_0,n_1,i_1,...,n_{m-1}$, $_1,n_m$) where m≥0, $n_0,...,n_m \in N$, $i_0,...,i_{m-1} \in \mathbb{N}$ (the natural numbers) and $n_{k+1}$ is the $i_k$-th successor $n_k$. This path is said to be *from* $n_0$ *to* $n_m$ and m is the *length* of the path.

(ii) A *cycle* is a path of length greater than 0 from a node n to itself. n is called a *cyclic* node.

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

1.6 DEFINITION. (i) A *term graph* (often, within this paper, simply a *graph*) is a quadru (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 *r* of the graph. (We do not require that every node of a term graph is reachable by a path from the root.) 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 ther a cycle containing the root.

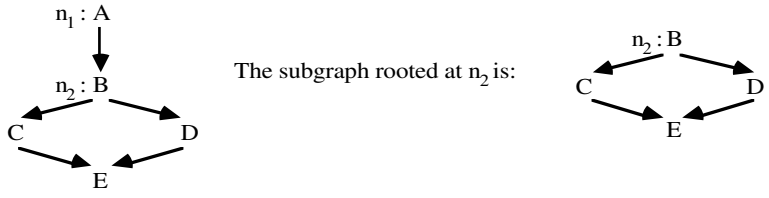
When we draw pictures of term graphs, the topmost node is the root. n

Term graphs are exactly the graphs discussed in the paper Barendregt et al.[1987], which define language of generalised graph rewriting of which the rewriting treated in this paper is a special case.

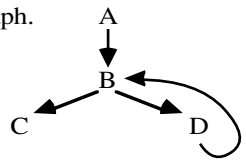
1.7 DEFINITION. Let g = (N,lab,succ) be a labelled graph and let n∈ N. The *subgraph o rooted at* n is the term graph (N',lab',succ',n) where N'={n'∈ N | there is a path from n to n'} and lab' a succ' are the restrictions of lab and succ to N'. We denote this graph by g|n. The definition also app when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

1.8 EXAMPLES.

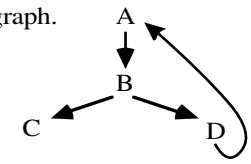
(i)




(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). Wh writing down examples of finite graphs, it is convenient to adopt a more concise notation, which abstra away from details such as the precise choice of the elements of N. We will use a notation based on definition of terms in definition 1.1, but with the addition of node-names, which can express the sharing common subexpressions. The notation is defined by the following context-free grammar, with restrictions following it.

1.9 DEFINITION (linear notation for graphs).

graph ::= node | node + graph

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

A ranges over **F**. x ranges over a set, disjoint from **F**, of *nodeids* ('node identifiers'). Any nodeid x wh occurs in a graph must occur exactly once in the context x **:** A(node,...,node). Nodeids are represented tokens beginning with a lower-case letter. Function symbols will be non-alphabetic, or begin with upper-case letter. We again abbreviate A() to A. n

This syntax is, with minor differences, the same as the syntax for graphs in the language LE (Barendregt et al.[1987]). The five graphs of the examples 1.4 are in this notation: G(A,B), G(A(x:B) G(x:B) + H(x), G(B) + H(B) and x:G(A,B(x)). Note that multiple uses of the same nodeid expr multiple references to the same node.

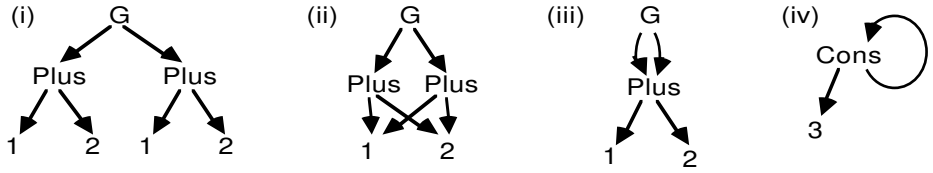
The definition of terms in 1.1 corresponds to a sublanguage of our shorthand notation, consisting those graphs obtained by using only the first production for graph and the first production for node. *terms have a natural representation as graphs.*

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) $n_1$: Cons( 3, $n_1$ )




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

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

The natural mapping of terms to graphs represents each term as a finite tree. However, some terms also be represented as proper graphs, by sharing of repeated subterms. For example, the te G(Plus(1,2),Plus(1,2)) can be represented by any of the graphs pictured in example 1.10 (i), (ii), or (iii) well as by the graphs G(Plus(x:1,2),Plus(x,2)) or G(Plus(1,x:2),Plus(1,x)).

## 2. Homomorphisms of graphs and trees.

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

$$lab_2(f(n)) = lab_1(n)$$

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

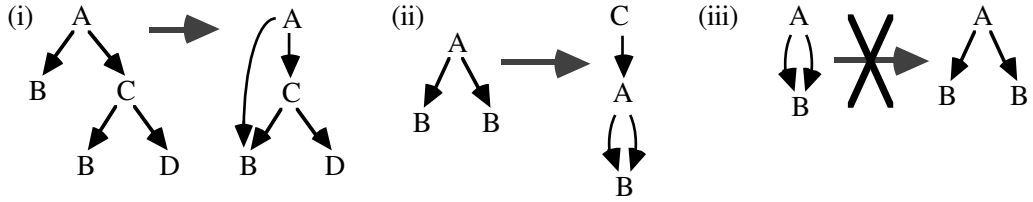
where **f** is defined by $\mathbf{f}(n_1,...,n_k) = (f(n_1),...,f(n_k))$. That is, homomorphisms preserve labels, successo and their order. n

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

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.




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 ~ g' when g and g' isomorphic.

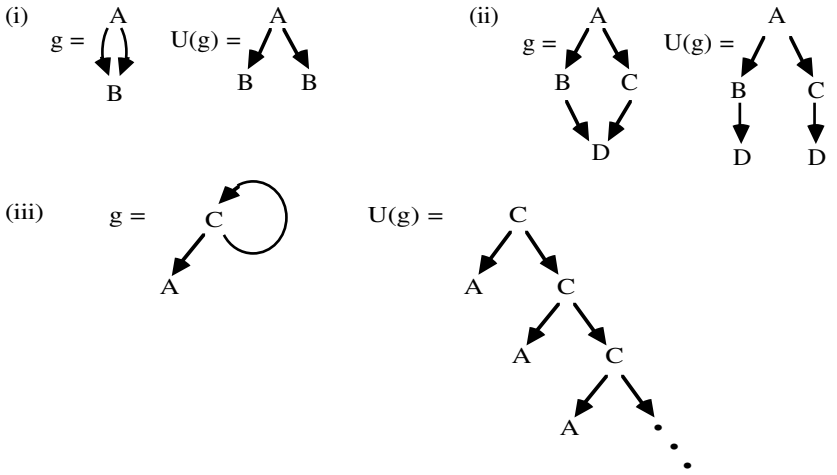
(iii) Two graphs are *equivalent* when they are isomorphic by a rooted isomorphism. We write g ≈ when g and g' are equivalent. n

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.* n

2.6 EXAMPLE. These two graphs are isomorphic but not equivalent (recall that in diagramma 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 resu from "unravelling" g from the root. We start with some examples.




Now we give the formal definition. U(g) has as nodes the rooted paths of g. The root of U(g) is path (r). For a path $p=(n_0,i_0,...,n_{m-1},i_{m-1},n_m)$, $lab_{U(g)}(p) = lab_g(n_m)$ and $succ_{U(g)}(p) = (p_1,...,p_k)$ where is the result of appending $(i,succ_g(n_m)_i)$ to p. Clearly this is a tree. n

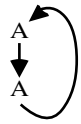
2.8 PROPOSITION. *For every graph g there is a rooted homomorphism $u_g$: U(g)→g defined by:*

$$u_g(n_0,i_0,...,n_m) = n_m.$$ n

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

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

For example, the graphs of example 1.10 (i), (ii), and (iii) are all tree-equivalent. So are these t
graphs:



## 3. Graph rewriting.

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

3.1 DEFINITION (term rewriting). (i) Let **V** be a fixed set of function symbols, disjoint from **F**. T
members of **V** are called *variables*, and are denoted by lower-case letters. An *open term* over a se
function symbols **F** is a term over **F**∪**V** in which every node labelled with a variable has no successo
An open term containing no variables (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
$t_R$ occurs in $t_L$. $t_L$ and $t_R$ are, respectively, the *left-* 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. n

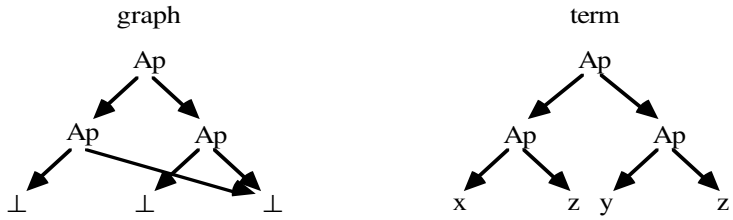
The usual definition of a term rewrite rule requires that $t_L$ be not just a variable. However, our resu
are not affected by the presence of such rules, so we do not bother to exclude them.

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

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

Open term graphs are intended to represent terms with variables. Instead of using the set **V**
variables, we find it more convenient, for technical reasons, to follow Staples[1980a] by using em
nodes. The precise translation from open graphs to open terms is as follows. Given an open graph ove
we first replace each empty node in it by a different variable symbol from **V**, and then unravel the result
closed graph over **F**∪**V**, obtaining an open term over **F**. Thus where a graph has multiple edges point
to the same empty node, the term will have multiple occurrences of the same nodeid. For example,
graph Ap(Ap(⊥,w:⊥),Ap(⊥,w)) translates to the term Ap(Ap(x,z),Ap(y,z)):




We could obtain any term which only differs from this one by changes of variables. We shall treat s terms as the same.

We now turn to the graph representation of term rewrite rules. We only deal with left-linear rule this paper. In 5.13 we discuss briefly the problems 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 gr 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 i homomorphism from g|n to $g_0$. The homomorphism f is called an *occurrence* of R. n

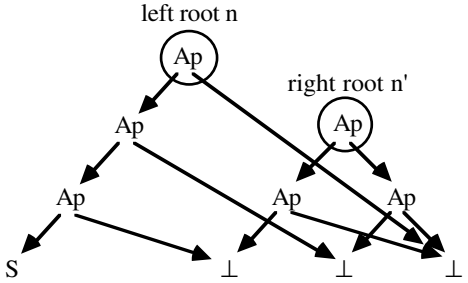
Rather than introduce our formal definition of graph rewriting immediately, we begin with so examples. The formal definition is given in section 3.6.

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, where g is a labelled graph and n and n' are nodes of g. First take the graphs representing $t_L$ and $t_R$. Fo the union of these, sharing those empty nodes which represent the same variables in $t_L$ and $t_R$. This gr is g. Take n and n' to be the respective roots of $t_L$ and $t_R$. Here are two examples which should make correspondence between term and graph rewrite rules clear.

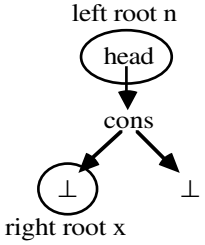
(i) Term rule: Ap(Ap(Ap(S,x),y),z) → Ap(Ap(x,z),Ap(y,z))

Graph rule: ( n:Ap(Ap(Ap(S,x:⊥),y:⊥),z:⊥) + n':Ap(Ap(x,z),Ap(y,z)), n, n' )



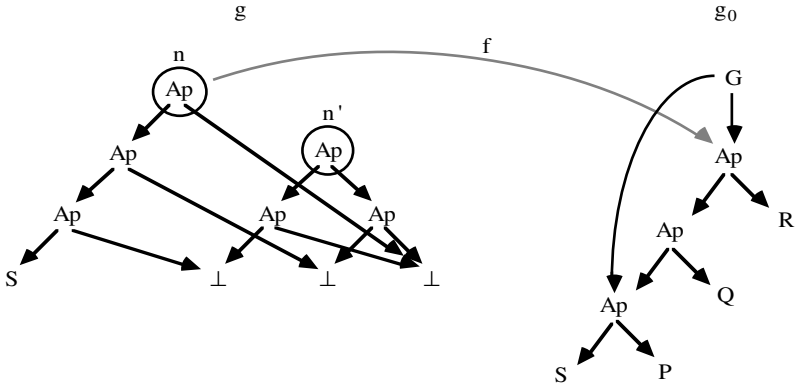

(ii) Term rule: head(cons(x,y)) → x

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

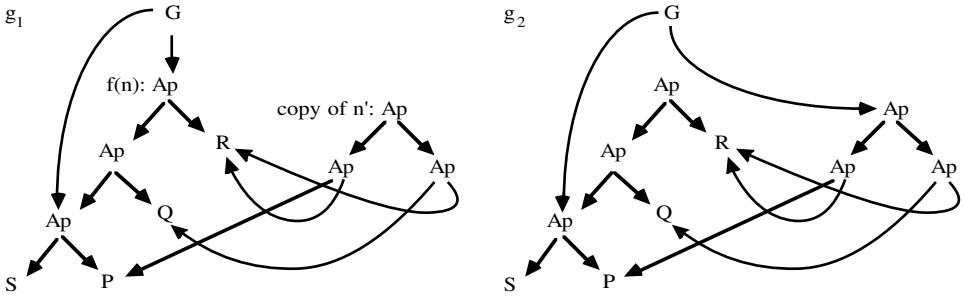



3.5 INFORMAL DEFINITION OF GRAPH REWRITING.

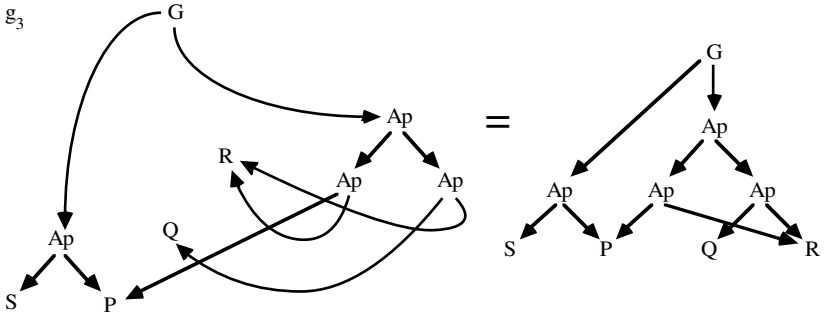
A redex ((g,n,n'), f: g|n → $g_0$) in a graph $g_0$ is reduced in three steps. We shall use the follow redex as an example: (g,n,n') is the S-rule above, $g_0$ = G(a,Ap(Ap(a:Ap(S,P),Q),R)) and f operates o as indicated in the picture (which completely determines how f behaves on the rest of g|n).




First (the *build* phase) an isomorphic copy of that part of g|n' not contained in g|n is added to $g_0$, with l succ, and root defined in the natural way. Call this graph $g_1$. Then (the *redirection* phase) all edges of 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 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 which is the result of the rewrite.




Note that the bottommost Ap node of the redex graph and the S node remain after garbage collecti 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 → $g_0$) be a redex i graph $g_0$. The graphs $g_1$ (the build phase), $g_2$ (the redirection phase) and $g_3$ (the garbage collection pha 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 functi $lab_{g1}$ and $succ_{g1}$ are given by:

$$lab_{g1}(m) = lab_{g0}(m) \qquad (m \in N_{g0})$$

$$= \mathrm{lab}_g(m) \qquad (m \in N_{g|n'} - N_{g|n})$$
$$\mathrm{succ}_{g1}(m)_i = \mathrm{succ}_{g0}(m)_i \quad (m \in N_{g0})$$
$$= \mathrm{succ}_g(m)_i \quad (m, \mathrm{succ}_g(m)_i \in N_{g|n'} - N_{g|n})$$
$$= f(\mathrm{succ}_g(m)_i) \quad (m \in N_{g|n'} - N_{g|n}, \mathrm{succ}_g(m)_i \in N_{g|n})$$

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 defin 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 $(N_h,\mathrm{lab},\mathrm{succ},r)$, where lab, succ, and r are given as follows.

$\mathrm{lab}(c) = \mathrm{lab}_h(c)$ for each node c of $N_h$

if $\mathrm{succ}_h(c)_i = a$ then $\mathrm{succ}(c)_i = b$, otherwise $\mathrm{succ}(c)_i = \mathrm{succ}_h(c)_i$

if $r_h = a$ then $r = b$, otherwise $r = r_h$

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 g this operation a name: for any term graph h, we denote $h|r_h$ by GC(h) (*G*arbage *C*ollection).

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

$$\mathrm{RED}(((g,n,n'),f),g_0) = \mathrm{GC}((g_0 +_f (g,n,n'))[f(n):=n']). \quad \text{n}$$

Our definition of graph rewriting is a special case of a more general notion, defined in Glauer al.[1987] by a category-theoretic construction. Those familiar with category theory may recognise the bu phase of a rewrite as a pushout, and redirection and garbage collection can be given definitions in the sa style (though the categories involved are not those defined in this paper). For the purpose of this pap describing graph rewritings which correspond to conventional term rewritings - the direct "operation definition we have given is simpler.

3.7 DEFINITION. (i) If g reduces to g' by reduction of a redex $\Delta$, we write $g \to^{\Delta} g'$, or $g \to g'$ if we do wish to indicate the identity of the redex. The reflexive and transitive closure of the relation $\to$ is $\to^*$.

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

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

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

Note that a GRS is not required to include all the graphs which can be formed from the given se function symbols **F**. Any subset closed under rewriting will do. This allows our definition to automatica handle such things as, for example, sorted rewrite systems, where there are constraints over what funct symbols can be applied to what arguments, or arities, where each function symbol may only be applied t specified number of arguments. From our point of view, this amounts to simply restricting the se graphs to those satisfying these constraints. So long as rewriting always yields allowed graphs fr

allowed graphs, we do not need to develop any special formalism for handling restricted rewrite syste
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 a
graph rewrite rule, provided that the term rewrite rule is left-linear, that is, if no variable occurs twice
more on its left-hand side. As some of the following examples show, however, some new phenom
arise with graph rewrite rules.

3.8 EXAMPLES.

(i) Term rule: $A(x) \rightarrow B(x)$; Graph rule: ( n:A(x:⊥) + n':B(x), n, n' )

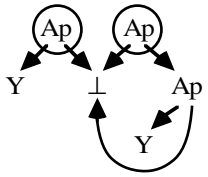
Graph: x:A(x);

Result of rewriting: x:B(x)

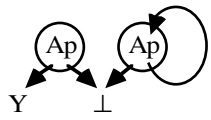
(ii) Term rule: $I(x) \rightarrow x$; Graph rule:( n:I(n':⊥), n, n' )

Graph: I(3); Result of rewriting: 3

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

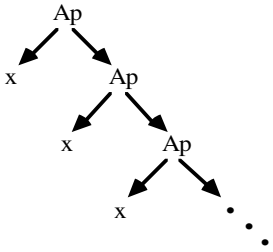



(n:Ap(Y,x:⊥) + n':Ap(x,Ap(Y,x)),n,n')




(n:Ap(Y,x:⊥ ) + n':Ap(x,n'),n,n')

This captures the fact that the Böhm tree (Barendregt [1984]) of the term Ap(Y,x) is:




The graph rule can do all the 'unravelling' in one step, which in the term rewrite world requires an infi
sequence of rewritings.

(iv) Here is a more subtle example of the same phenomenon illustrated by (iii). Consider the te
rewrite rule

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

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

$$( n:F(Cons(x:\bot,y:\bot)) + n':G(Cons(x,y)), n, n' )$$

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

$$( n:F(z:Cons(\bot,\bot)) + n':G(z), n, n' )$$

Both the languages Standard ML and Hope, which are languages of term rewriting, allow an enhanced fo
of term rewrite rules such as (using our syntax):

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

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

(v) Term rule: $I(x) \rightarrow x$; Graph rule: ( n:I(n':⊥), n, n' )

Graph: x:I(x); Result of rewriting: x:I(x)

Example 3.8(v) is deliberately pathological. Consider the GRS for combinatory logic, whose rules
those for the S, K, and I combinators. The graph can be interpreted as "the least fixed point of I" (cf.
example of the Y combinator above), and in the usual denotational semantics in terms of reflexive doma
should have the bottom, "undefined" value. As the graph reduces to itself (and to nothing else), it has
normal form. Thus our operational semantics of graph rewriting agrees with the denotational semant
This is not true for some other attempts we have seen at formalising graphical term rewriting.

We now study some properties of graph rewrite systems. We establish a version of the theorem
finite developments for term rewriting, and the confluence of weakly regular systems. For reasons
space, the longer proofs are omitted from this paper. They appear in Barendregt et al. [1986].

3.9 PROPOSITION. *Garbage collection can be postponed. That is, given* $g \rightarrow^{\Delta 1} g_1 \rightarrow^{\Delta 2} g_2$, $\Delta_i = (R_i,f_i)$ $= (g_i,n_i,n'_i)$ $(i = 1,2)$ *and* $g'_1 = (g +_{f1} R_1)[f(n) = n']$, *then* $\Delta_2$ *is also a redex of* $g'_1$, *and* $g'_1 \rightarrow^{\Delta 2} g_2$. n

3.10 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* i

(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 $g_2|n_2$.

$\Delta_1$ and $\Delta_2$ are *weakly disjoint* if either they are disjoint, or the only violation of conditions (i) and (ii) is t
$f_1(n_1) = f_2(n_2)$, and the results of reducing $\Delta_1$ or $\Delta_2$ are identical.

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

3.11 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 gr*
*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 t*
$f(n_2) = f_2(n_2)$. n

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

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

3.13 PROPOSITION. *Let $\Delta_1$ and $\Delta_2$ be weakly disjoint redexes of g, and let $g \rightarrow^{\Delta i} g_i$ (i = 1,2). Then ther 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.*

3.14 COROLLARY. *Every weakly regular GRS is confluent. That is, if $g \rightarrow^* g_i$ (i = 1,2), then there is a such that $g_i \rightarrow^* h$ (i = 1,2).* n

3.15 DEFINITION. Let g be a graph and $\mathsf{F}$ be a set of disjoint redexes of g. A *development* of $\mathsf{F}$ i reduction sequence in which the redex reduced at each step is a residual, by the preceding steps of sequence, of a member of $\mathsf{F}$. A *complete development* of $\mathsf{F}$ is a development of $\mathsf{F}$ , at the end of wh there remain no residuals of members of $\mathsf{F}$. n

3.16 PROPOSITION. *Every complete development of a finite set of pairwise disjoint redexes $\mathsf{F}$ is finite. fact, its length is bounded by the number of redexes in $\mathsf{F}$.* n

3.17 PROPOSITION. *Let $\mathsf{F}$ be a set of redexes of a graph g. Every finite complete development of $\mathsf{F}$ e 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 $\Delta_1 = (f_1,R_1),\ldots,\Delta$ $(f_i,R_i)$.* n

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

## 4. Tree rewriting.

In order to study the relationship between term rewriting and graph rewriting, we define the notion *tree rewriting*. This is a formalisation of conventional term rewriting within the framework of 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 rewriting* 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 a set of trees over **F** closed under $\rightarrow_{t\mathbf{R}}$. A *term rewrite system* (TRS) is a TreeRS, all of whose trees finite.

When $t_1$ reduces to $t_2$ by tree rewriting of a redex $\Delta$, we write $t_1 \rightarrow_t^{\Delta} t_2$, or $t_1 \rightarrow_t t_2$ when we do wish to indicate the identity of the redex. n

Tree rewrite systems differ from conventional term rewrite systems in two ways. Firstly, infinite tr are allowed. We need to handle infinite trees, since they are produced by the unravelling of cyclic grap

We need to handle cyclic graphs because some implementors of graph rewriting use them, and we do want to limit the scope of this paper unnecessarily. Secondly, the set of trees of a TreeRS may be any se trees over the given function symbols which is closed under tree rewriting. This is for the same reasor 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 function symbols will be closed under tree rewriting. This is true for those rules resulting from term rew 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 $t[n_1:=t_1,...,n_i:=t_i$ be the tree whose nodes are

(i) all paths of t which do not include any of $n_1$,...,$n_i$, and

(ii) every path obtained by taking a path p of t, which ends at $n_j$ ($1 \le j \le i$) and contains no ot occurrence of $n_1...n_i$, and replacing the last node of p by any path of $t_j$.

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

The results concerning disjointness, regularity, and confluence which we proved for graph rewriting have versions for tree rewriting. Again we omit proofs. We also have the following:

4.3 PROPOSITION. *Unravelling can be postponed. That is, if $t_1 \to_t t_2 \to_t t_3$, then there are graphs g a g' such that*

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

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

## 5. Relations between tree and graph rewriting.

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

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 rewrite rules). It is trivia 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 te rewrite system.

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

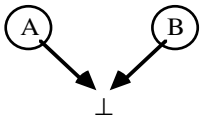
When (T,**R**) is a term rewrite system, L(T,**R**) represents its graphical implementation. There are t fundamental properties it must have to be a correct implementation, which we now define.

5.2 DEFINITION. (i) A TreeRS (T,**R**) is called *graph-reducible* if for every graph g in L(T), if t is a norı 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 normal form in (T,**R**), then g has no normal form in L(T,**R**).

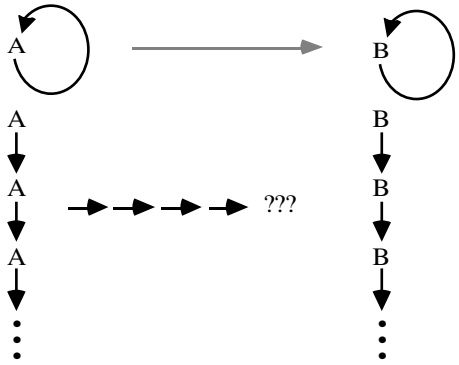
(ii) A GRS (G,**R**) is *tree-reducible* if there is a TreeRS (T,**R**) such that (G,**R**) = L(T,**R**), and such t if g' is a normal form of g in (G,**R**), then U(g') is a normal form of U(g) in (T,R), and if g has no norı form in (G,**R**), then U(g) has no normal form in (T,**R**). n

L(T,**R**) is the graphical implementation of (T,**R**). Tree-reducibility of L(T,**R**) expresses soundne every result which is obtainable by graph rewriting in L(T,**R**) is also obtainable by tree rewriting in (T, Graph-reducibility of (T,**R**) expresses completeness: every result which is obtainable by tree rewritin also obtainable by graph rewriting. We shall see that every GTRS is tree-reducible, and every wea regular TRS is graph-reducible. Not all GRSs, even those of the form L(T,**R**), are tree-reducible, no every TreeRS graph-reducible, as the following examples show.

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




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

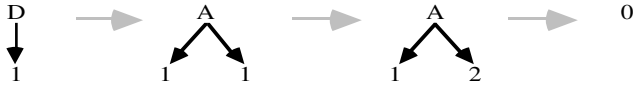



5.4 EXAMPLE. The following TreeRS is not graph-reducible:

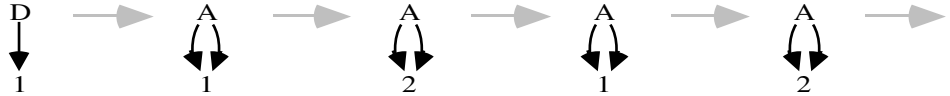
T: trees over {A,D,0,1,2}, with the following arities: A is binary, D is unary, and 0, 1, an are nullary.

**R**: A(1,2) → 0; 1 → 2; 2 → 1; D(x) → A(x,x).

For a counterexample, consider the following tree rewriting sequence:




In the graph rewriting system we have:



In this example, the sharing of (tree) subterms in the graph world has excluded from the graph world cert rewrite sequences of the tree world. Distinct subterms of A(1,1) correspond to the same subgraph A(x:1,x), forcing synchronized rewriting 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

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

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

5.6 PROPOSITION. *Let $g \to g'$ by rewriting of an acyclic redex $\Delta$. Then $U(g) \to^*_t U(g')$ by compl development of $u_g^{-1}(\Delta)$. For any redex $\Delta'$ of g, weakly disjoint from $\Delta$, $u_g^{-1}(\Delta'//\Delta) = u_g^{-1}(\Delta')//u_g^{-1}(\Delta)$.*

5.7 PROPOSITION. *Let $g \to^* g'$ by a complete development of a set $\mathsf{F}$ of disjoint acyclic redexes of g wh associated rewrite rules are acyclic. Then $U(g) \to^*_t U(g')$ by a complete development of $u_g^{-1}(\mathsf{F})$.* n

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

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

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

$t \to^{GK}_t t' \Leftrightarrow t \to^*_t t'$ by complete development of the set of all redexes of t. n

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

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

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 is a normal form of (T,**R**).* n

Thus in a graphical term rewrite system L(T,**R**), everything which can happen can also happen in term rewrite system, and all the normal forms are the same. Graph-reducibility may fail, however, sinc 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 fo Proposition 5.7 relates the Gross-Knuth reduction sequences for g and U(g) in the following way.

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

$$U(g) \xrightarrow{GK}_t U(g_1) \xrightarrow{GK}_t U(g_2) \xrightarrow{GK}_t \cdots$$

It is a standard result that for regular TRSs, Gross-Knuth reduction is normalising (Klop [1980]), and proof carries over immediately to weakly regular TreeRSs. Therefore the bottom line of the diagr

terminates with some tree U(g') in normal form such that g reduces to g' in L(T,**R**). Therefore g' i
normal form of g, and (T,**R**) is graph-reducible. n

5.13 NON-LEFT-LINEARITY.

We shall now discuss non-left-linearity, and indicate why we excluded non-left-linear TRSs fr
consideration. In term rewriting theory, for a term to match a non-linear left-hand side, the subte
corresponding to all the occurrences of a repeated variable must be identical.

Our method of using empty nodes to represent the variables of term rewrite rules suggests a v
different semantics for non-left-linear rules. Our representation of a term rule A(x,x) → B would
(n:A(x:⊥,x), n, x). This will only match a subgraph of the form a:A(b: …, b). That is, the subgra
matched by the repeated variable must be not merely textually equal, but identical - the very same nodes
one is implementing graph rewriting as a computational mechanism in its own right, rather than consider
it merely as an optimisation of term rewriting, then this form of non-left-linearity may be useful. Howe
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 variable
graphs, just as for terms, instead of empty nodes. A meaning must then be chosen for the matching c
graph A(Var1,Var1) where Var1 is a variable symbol, occurring at two different nodes. Two possibili
naturally suggest themselves. The subgraphs rooted at nodes matched by the same variable may be requi
to be equivalent, or they may only be required to be tree-equivalent. The latter definition is closer to
term rewriting concept.

When a variable occurs twice or more on the left-hand side of a rule, there is also a problem of decid
which of the subgraphs matched by it is referred to by its occurrences on the right-hand side. One metl
would be to cause those subgraphs to be first coalesced, replacing the equivalence or tree-equivalence wh
the matching detected by pointer equality. This technique may be useful in implementing lo
programming languages, where non-linearity is much more commonly used than in functional te
rewriting. Further investigation of the matter is outside the scope of the present paper.

Lastly, we note that although some term rewriting languages, such as SASL (Turner [1979b]) a
Miranda (Turner [1986]), allow non-left-linear rules, they generally interpret the implied equality test neit
as textual equality, nor as pointer equality, but as the equality operator of the language (although poi
equality may be used as an optimisation). In these languages, any program containing non-left-linear ru
can be transformed 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 te
and graph rewrite systems. We then study the relationships between strategies for term rewrite systems a
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 is a binary relation on O. This notion abstracts from term and graph rewrite systems. The transit 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 i A sequence of length 0 is *empty*. n

6.2 DEFINITION. (i) Given an ARS $(O,\rightarrow)$, a *strategy* for this system is a function S which takes each x 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 ending with y. By abuse of notation, we n 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. n

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. n

A 1-strategy for a TreeRS or GRS can be specified as a function which takes the objects of the syst 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 rew system $L(T,\mathbf{R})$ is defined by $S_L(g) = u_g(S(U(g)))$. n

For 1-strategies on TreeRSs, this is a natural definition of lifting. For multi-step strategies, it is l clear how to define a lifting, and we do not do so in this paper.

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

6.5 EXAMPLE. Consider the following TreeRS:

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

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

By stipulating that A is binary and B, 1, and 2 are nullary, we mean, as discussed following definitions and 4.1, that trees not conforming to these arities are not included in the system. Define a strategy S follows (where the redexes chosen by S are boldfaced):

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

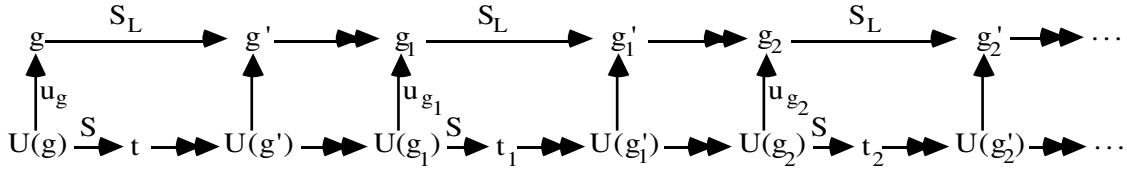
$\mathbf{A(x,y)} \to B$, if neither of the preceding cases applies

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 b again in an infinite loop.

The next theorem shows that if a 1-strategy S for a TreeRS is hypernormalising, then $S_L$ 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**, 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 qu $S_L$ reduction sequence starting from g. By proposition 5.7 and the definition of $S_L$, we can construct 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 some point, with a normal form of U(g). Therefore the top line also stops, and must do so with a gr which unravels to the normal form in the bottom line. n

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

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

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

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

$A(\alpha,\beta) \to_S A(\alpha,\alpha)$ $\quad\quad$ (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 nor form B. In the former case, whatever extra steps are then inserted, the result can only be either another te of the same form or B. In the former case, the next $S_L$-step will reach B. Therefore $S_L$ hypernormalising. However, S is not hypernormalising. A counterexample is provided by the te 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) Suppose g has no nf. Then the $S_L$ path of g is infinite. This gives, by the construction of 6.6, an infi quasi-S-path of t, hence t has no normal form. n

An application of this result is that strongly sequential TRSs (in the sense of Huet and Lévy [1979]) graph reducible. This follows from their theorem that the 1-strategy which chooses any needed rede hypernormalising.

The condition that a strategy be hypernormalising is unnecessarily strong. Inspection of the proofs 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. n

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**,* *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. n

## 7. Conclusion.

Graph rewriting is an efficient way to perform term rewriting. We have shown:

1. Soundness: for all TRSs, graph rewriting cannot give incorrect results.
2. Completeness: for weakly regular TRSs, graph rewriting gives all results.
3. Many normalising strategies (the hypernormalising, or even the sib-normalising ones) on terms can be lifted to graphs to yield normalising strategies there. In particular, for strongly sequential te rewrite systems, the strategy of contracting needed redexes can be lifted to graphs.

We have also given counterexamples illustrating incompleteness for non-weakly regular TRSs and liftings of non-sib-normalising strategies.

## References.


Barendregt, H.P.
[1984] *The Lambda Calculus: its Syntax and Semantics* (revised edition), North-Holland, Amsterdam.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep
[1986] Term graph rewriting, Report 87, Department of Computer Science, University of Nijmegen, and also as Re SYS-C87-01, School of Information Systems, University of East Anglia.
[1987] Towards an intermediate language based on graph rewriting, these proceedings.

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 Informat Twente University of Technology.

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

Glauert, J.R.W., J.R. Kennaway and M.R. Sleep
[1987] Category theoretic concepts of graph rewriting and garbage collection, in preparation, School of Informa Systems, University of East Anglia.

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

Kennaway, J.R.
[1984] An outline of some results of Staples on optimal reduction orders in replacement systems, Report CSA/19/1 School of Information Systems, University of East Anglia, Norwich, England.

Klop, J.W.
[1980] *Combinatory Reduction Systems*, Mathematical Centre Tracts n.127, Mathematical Centre, Kruislaan 413, 1 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] 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, Ox University.

Line 1 This page is to test the page width and length.
2 It should be 18 centimetres wide
3 It should be 5 1/2 inches tall, which at 6 lines per inch is 33 lines.
4 The maximum number of pages in the paper is 36. This page must be no more than page 3
5
6
7
8
9
Line 10
1
2
3
4
5
6
7
8
9
Line 20
1
2
3
4
5
6
7
8
9
30
1
2
Line 33 This should not be the last line on this page.
4 This should not be the first line on a new page.
5
6
7
8
9
Line 40
1
2
3
4
5
6
7
8
9
Line 50
1
2
3
4
5
6
7
8
9
Line 60

1
2 Should be the last line on the page
3 Should be the first line on a page
4 What you see is **never** what you get.
5
6
7
8
9
Line 70
1
2
3
4
5
6
7
8
9