Dag Representation and Optimization of Rewriting *

Ke Li
Courant Institute
New York University
251 Mercer Street
New York, NY 10012
U.S.A.
E-mail: like@cs.nyu.edu

Abstract

In all applications of term rewriting systems, computing the normal forms of terms is the fundamental step. In this paper, we propose a directed acyclic graph (dag) representation for term and term rewriting. The rewriting on dags is much more efficient than the rewriting on trees. We design several efficient strategies for rewriting on dags. By dag representation, we can even obtain efficient rewriting strategies for non-left-linear rewriting systems.

*This research was supported in part by the National Science Foundation under grant number CCR-89-6949.

1 Introduction

In any application of term rewriting systems (TRSs for short), computing normal form of terms is a basic procedure. In language application, the semantics of a term is its normal form [Der85]. In solving word problem [KB70], to test equivalence of two terms t and s, the normal forms of t and s are computed. To unify terms in an equation theory, the equations are transformed to a terminating and confluent TRS and the normal forms of the terms are computed [Hu80]. To combine logic and functional programming languages, TRSs are attached to Horn clauses and in unification of clauses, normal form are computed [SY86][GM86]. Therefore, efficiency of normal form computation is important in all applications of TRSs.

The research for efficient normal form computation is done in many ways. Huet and Levy [HL79] have given a theoretic analysis for derivations and proposed "strong sequential" TRSs which can be transformed to an automata computing normal form fast. In [CKS87], Choppy et al. have given techniques for estimating the expected number of steps in the derivations with the goal of constructing efficient "regular" TRSs. In [Li90], NP-completeness results are obtained for efficient normal form computation and several optimal strategies are proposed for subclasses of TRSs.

Conventionally, also in all discussions above for efficient normal form computation, terms are assumed to be represented by trees. But tree representation is inefficient in space. If we allow common subterms to be shared, a term can be represented by a directed acyclic graph (dag for short). It is well-known that there is some term that can be represented by a dag with n nodes, but it has to be represented by a tree with exponential number of nodes. Based on dag representation of terms, Paterson and Wegman [PW78] have designed an efficient unification algorithm. In reverse, the cause of unefficiency of Robinson's unification algorithm [Rob71] is due to the tree representation. Besides the advantage of space-saving, dag representation of terms can save computing time, which is rewriting steps in term rewriting. For example, given a TRS:

$$\begin{aligned} f_1(x) &\rightarrow f(f_2(x), f_2(x)) \\ f_2(x) &\rightarrow f(f_3(x), f_3(x)) \\ &\cdots \\ f_{n-1}(x) &\rightarrow f(f_n(x), f_n(x)) \end{aligned}$$

and given a term $f_1(a)$, in tree representation, each rewriting will introduce two subterms that are actually the same. Eventually, we need exponential of n rewritings to rewrite $f_1(a)$ to its normal form. But if terms are represented by dags, only linear number of rewritings are needed.

Another good example is computation of Fibonacci numbers, which is shown in [HP88a].

Barendregt et al. [BEG87] have proposed "term graph rewriting", and Hoffman and Plump [HP88a] have proposed "jungle evaluation". In the both research, terms are represented by mechanisms like dags, the goal being that common subterms are shared and normal form computation will be more efficient. But in the both representations for terms, some common subterms may not be shared. In the rewriting, to maintain the structure of dag representation, extra work is needed. The both papers have not analyzed the trade-off between the extra work and the efficiency of rewriting.

In this paper, we propose a dag representation for terms, which shares the most number of common subterms. One of the advantages of this representation is that each term is mapped to a unique dag. This uniqueness will provides us a clear understanding for the relationship between the rewriting on trees and the rewriting on dags. Properties of term rewriting, such as termination, confluence, non-ambiguity, etc., will be compared for two representations. We will see that given a canonical (terminating and confluent) TRS represented by trees, the corresponding TRS represented by dags is also canonical. Furthermore, we will show that the unique normal form of a term for a canonical TRS can be efficiently computed by the corresponding TRS in dag representation.

For rewriting on dag representation, we will give detailed analysis on how to find a redex, how to rewrite it, and how to maintain the dag structure.

Another interesting point in this paper is to give optimal or "good" strategies for rewriting on dags. This has not been done for the existing systems for rewriting on dags [BEG87][HP88a]. We will see that the strategies for efficient rewriting on trees, which are proposed in [Li90], can be adapted for efficient rewriting on dags. With dag representation, the applications of these strategies are expanded to much larger subclasses of TRSs.

The most of previous researches on efficient rewriting have been done only for left-linear TRSs. In this paper, with dag representation, we obtain several results on non-left-linear TRSs.

In the following, we will introduce our dag representation for terms and TRSs in section 2, discuss relationship between rewriting on trees and rewriting on dags in section 3, and address efficient rewriting strategies for our dag representation of TRSs in section 4. Section 5 concludes the paper and suggests future work.

2 Dag Representation of Terms and Dag Rewriting

Recursively, a term is defined to be a constant or a variable or $f(t_1, ..., t_n)$ where f is a function of arity n and $t_1, ..., t_n$ are terms. In convention, a term is depicted as a tree: constants and variables are the bottom nodes and each function corresponds to an distinct inner node. For example, term $f(g(h(a,a),h(a,a)),g(h(a,a),h(a,a)),h(a,a))$ is a tree shown in figure $1(a)$ If we allow subterms to be shared, a term can be represented by a directed acyclic graph (short as dag). For example, the term in figure $1(a)$ is represented by a dag in figure $1(b)$ in which an arc represents a directed up-down edge.

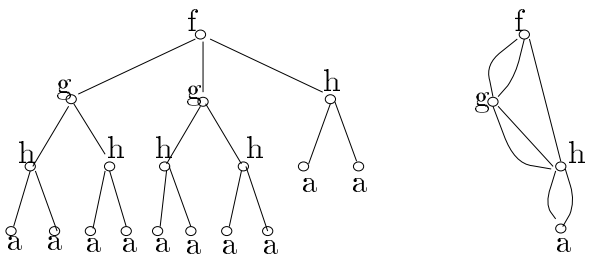


Figure 1

The figures apparently show that the dag representation of a term save space than the tree representation. Besides advantage of space-saving, dag representation of terms can save a lot of computing time — rewriting steps in words of rewriting systems. In the early stage, dag representation is used to efficiently compute recursive programs [PMT74][BL79]. Recently, dag representation is proposed to generate efficient reductions for general rewriting systems [BEG87][HKP88][HP88a]. Due to the advantage of saving space and time, dag representation is preferred in many applications of rewriting systems.

Our topic is how to take the greatest advantage of time-saving by dag representation in term rewriting. For a term, there may exist several corresponding dags. See figure 2. The term in figure $2(a)$ can be represented by each of the 5 dags in figure 2 from (b) to (f). The dag in figure $2(f)$ shares the most number of subterms.

The reason why dag representation saves reduction time for rewriting systems is that many rewritings are performed on the shared subterms so that many duplicated rewritings are avoided. For example, given rewriting system $\{a \rightarrow b\}$ and term $t = f(g(a,a),g(a,a))$, 4 rewritings have to be done to t. If t is represented by the dags in figure 2, 3 rewritings are needed for (b) and (c), 2 needed for (d) and (e), 1 needed for (f). It is easy to conclude that the more subterms are shared, the less

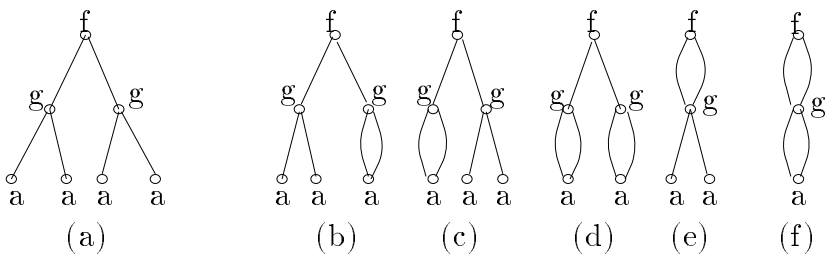


Figure 2

number of rewritings are needed, that is, the more reduction time is saved.

The existing dag representation for rewriting systems (expressed by " graph representation" in [BEG87] and "jungle" in [HP88]) does not specificly use the dag which shares the most number of subterms and it permits some common subterms not to be shared. As an example, one of the 5 dags in figure 2 is allowed to represent the term in figure 2(a). If dag representation is used for rewriting systems, the extra work has to be done to maintain the dag structure. The maintaining work is called "garbage collection" in [BEG87] and "folding" in [HP88]. We immediately think of the trade-off between the extra maintaining work and the saved rewriting steps for dag representation. From the conclusion mentioned above, we prefer the dag representation that shares more subterms and does not need much maintaining time.

Here, we propose a dag representation for rewriting systems that shares the most of subterms and the maintaining work is not much. Furthermore, we will propose strategies for this dag representation to speed up the rewriting process.

First, rewriting on terms should be understood clearly. Let t be a term and $l \to r$ a rule. We say $l \to r$ can be applied to t if there is a substitution σ and a subterm t' in t such that $l\sigma = t'$. This process is called a matching, i.e., l matches t'. Term t with a subterm t' is denoted by $t[t']$. A rewriting of $l \to r$ to t is a replacement of t' by $r\sigma$ in t, denoted by $t[t'] \to t[r\sigma]$. The reflexive and transitive closure of rewriting $\to$ is denoted by $\stackrel{*}{\to}$.

In our dag representation, every term is represented by a dag that shares all common subterms. As an example, for the term in figure 2(a), our corresponding dag is the dag in figure 2(f). Formally, a term t is mapped to a dag t_{dag} in which there is only one occurrence for each constant and variable, and if we have two terms $p = f_1(t_1, ..., t_k)$ and $q = f_2(s_1, ..., s_k)$ where $f_1 = f_2$, and t_i and s_i $(1 \leq i \leq k)$ share one node in the dag, then p and q share one node in the dag. We have

Proposition 1 *Any term is mapped to a unique dag.*

Proof: Given a term t, the corresponding dag representation t_{dag} can be constructed from bottom to up. The uniqueness can be easily shown by induction on the level of the nodes from bottom to up. □

Let R be a term rewriting system consisting of m rules, i.e., $R = \{l_i \rightarrow r_i \,|1 \leq i \leq m\}$. l_i and r_i are terms. In our system, terms in rules are also represented by dags. Specifically, l_i and r_i in R are dags. Now, we consider how rewriting is performed for our dag representation.

Each rewriting consists of 4 steps:
Step 1. Redex finding;
Step 2. Dag splitting;
Step 3. Rewriting;
Step 4. Dag merging.
We will describe each step in the detail. After that, for clear understanding, an example will be given.

Suppose $l \rightarrow r$ is applied to t.

In step 1, redex finding is processed almost conventionally. But since terms are represented by dags, the matching of a term by a rule can be done by Paterson and Wegman's efficient unification algorithm [PW78] (or any other efficient unification algorithm based on dags), because matching is a special case of unification, that is, the variables in the term being matched are considered as constants. Therefore, redex finding is much efficient than the conventional one in which terms are represented by trees.

In step 2 (dag splitting), make a copy of the redex obtained in step 1 and delete the nodes in t which are in the redex but not shared by any nodes besides the redex. We will let the root of the redex stay and call it dangling root. This can be done in linear time, for example, by width first algorithm. Suppose after this step, t becomes t_{med}.

In step 3, rewriting is done to the copy obtained in step 2. This is the same as the conventional one, but the result is a dag. Denote the result by t_{re}.

In step 4, merge t_{re} with t_{med}. In the following we will give a linear merging algorithm for two dags. After merging, make the dangling root obtained in step 2 point to the root of t_{re}. See the following example for clear understanding.

Example. Given rewriting system $\{g(x, y) \rightarrow h(x)\}$ and term $f(g(a, b), g(b, c))$. The four steps to rewrite the term are shown in figure 3.

Here is the algorithm for dag merging.

Dag Merging Algorithm.
Input: dag t_{med} and dag t_{re}.
Output: one dag.
Process:

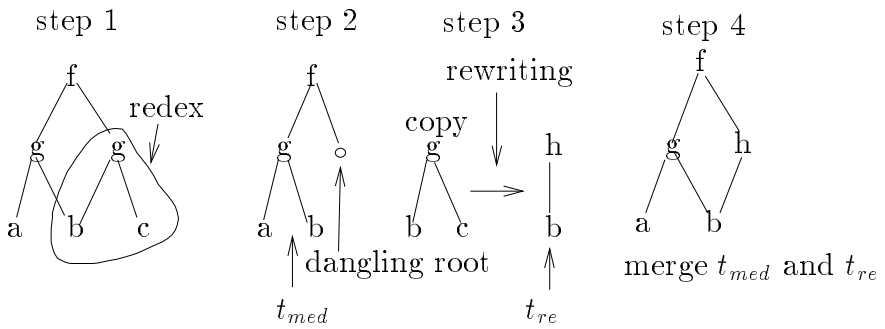


Figure 3

a) Merge the bottom nodes of t_{med} and t_{re}, that means, if there are two common bottom nodes in t_{med} and t_{re}, merge the duplicated node in t_{re} to the one in t_{med}.
b) In t_{re}, from bottom to up, for each inner node c do: if all children of c have been merged into t_{med} and the children have a father c in t_{med}, merge c in t_{re} to c in t_{med}; otherwise, keep c in t_{re}.

Proposition 2 *The dag merging algorithm generates a dag and the running time is linear to* $|t_{med}| + |t_{re}|$. *Actually, in the most of cases, the time is almost linear to* $|t_{re}|$.

Proof: At step a) in the algorithm, all the bottom nodes are merged so that there is only a single node for a variable or a constant. By induction on the levels of the nodes in t_{re} from bottom to up, it is easy to verify that there is no redundant copys of any subterm. That is, a dag is produced.

Merging bottom nodes of t_{med} and t_{re} can be done in time linear to the number of the bottom nodes. For each inner node in t_{re}, we check its children to see if it can be merged into t_{med}. Notice that the arities of function symbols are fixed, so the time for step b) is $O(|t_{re}|)$. The total time is $O(|t_{med}| + |t_{re}|)$. Actually, for t_{med}, we only consider the bottom nodes of t_{med}, which are constants and variables. Generally, in a term, most of symbols are function symbols while there are only a few *distinct* constants and variables. Therefore, in most of cases, the algorithm is almost linear to $|t_{re}|$. □

If terms are represented by dags and rewritings on dags are performed by the method described above, we call this rewriting a *dag rewriting*. The conventional rewriting on trees is called a *tree rewriting*. In the same way, we call a TRS represented by trees a *tree TRS* and a TRS represented by dags a *dag TRS*. Henceforth, we assume that dag TRSs are applied only to terms in dag representation and tree TRSs are applied only to terms in tree representation.

Now, let's analyze efficiency of dag rewriting.

To rewrite a term in dag rewriting, step 1 (described above) saves time while step 2 and step 4 are extra work. Both step 2 and step 4 run in time almost linear to

the size of the redex found in step 1. Step 1 saves time in two hands: (1) Because of dag representation, linear matching algorithms (or any efficient algorithm on dags) can be used. (2) To find a redex from term t, we have to try matching algorithm at several positions in t. Dag representation reduces the search space, that means, less matching would be tried. We already know that matching algorithm is at best linear to the sizes of the two terms being matched. Therefore, we can conclude that the extra work in step 2 and in step 4 is balanced by the time-saving in step 1.

3 Characteristics of Tree Rewriting and Dag Rewriting

Traditionally, many properties of term rewriting systems are discussed on tree representation, such as termination, confluence, ambiguity, etc.. Here, we will consider these properties on dag rewriting systems. These properties are discussed in [HP88b] for "jungle evaluation", but our system is simpler and our discussion is much simpler. In this section, rewriting system R and term t without any subscripts will be assumed to be represented by trees and their corresponding dag representations are denoted by R_{dag} and t_{dag} respectively.

Rewriting Derivations

Intuitively, given a tree TRS R and the corresponding dag TRS R_{dag}, a rewriting derivation D_{dag} using R_{dag} is a shorter version of D using R. Formally, we have

Proposition 3 *Given R and the corresponding R_{dag}, let D_{dag} be a derivation using R_{dag} which rewrites term t to term t'. Then, there is a derivation D using R such that D rewrites t to t' and $|D| \geq |D_{dag}|$.*

Proof: Consider one rewriting step $D_{dag}[i]$ in D_{dag} which rewrites s_{dag} to s'_{dag} by rule $l_{dag} \to r_{dag}$ in R_{dag}. In $D_{dag}[i]$, one of redexes of s_{dag} is rewritten, say it is u. There must be many copies of u in s all of which can be rewritten by the rule $l \to r$ in R. We arrange these rewritings by $l \to r$ in a sequence, obtaining a derivation using R which rewrites s to s'.

We expand each rewriting step in D_{dag} in this way, obtaining a rewriting derivation D using R which rewrites t to t'. Because each rewriting step in D_{dag} is expanded to at least one rewriting step in D, we have $|D| \geq |D_{dag}|$. □

Termination and Confluence

We say a TRS R is *terminating* if for any term t, there is no infinite rewriting chain using R for t.

Proposition 4 *If R is terminating, then R_{dag} is terminating.*

Proof: If there is an infinite sequence of rewritings using R_{dag}, by proposition 3, we can construct an infinite sequence of rewritings using R, contrary to the condition that R is terminating. Therefore, R_{dag} is terminating. □

A TRS R is said *confluent* if for any t, two rewriting derivations $t \xrightarrow{*} s_1$ and $t \xrightarrow{*} s_2$ imply that there exists a term t' such that there are two rewriting derivations $s_1 \xrightarrow{*} t'$ and $s_2 \xrightarrow{*} t'$.

Unfortunately, the condition that R is confluent does not guarantee that R_{dag} is confluent. One of courter-examples is given by [HP88b]: $R = \{f(x) \to g(x,x), f(a) \to N, g(a,a') \to N, a \to a', a' \to a\}$. R is confluent for tree representation. For dag representation, term $f(a)$ can be rewritten to N and $g(a,a)$, but the latter can only be rewritten to $g(a',a')$, that means $g(a,a)$ cannot be rewritten to N by rule $g(a,a') \to N$.

In this paper, our major concern is that given a terminating and confluent TRS, how to compute normal forms for terms fast. Fortunately, if R is *both* terminating and confluent, we can guarantee that R_{dag} is terminating and confluent. To prove this, we need the proposition below:

Proposition 5 *t is reducible using R if and only if t_{dag} is reducible using R_{dag}.*

Proof: First, because matching (special case of unification) between dags obtains the same substitution for variables as matching between the corresponding trees (see [PW78]), we have the claim that t can be matched by s if and only if t_{dag} can be matched by s_{dag}.

Therefore, t can be rewritten by a rule r from R if and only if t_{dag} can be rewritten by the corresponding rule of r from R_{dag}. □

Proposition 6 *If R is terminating and confluent, then R_{dag} is terminating and confluent.*

Proof: Termination of R_{dag} is proved above. We prove confluence of R_{dag} as follows.

Given any term t_{dag}, suppose there are two rewriting derivations $t_{dag} \xrightarrow{*} s_{dag}$ and $t_{dag} \xrightarrow{*} s'_{dag}$. Since R_{dag} terminating, we rewrite s_{dag} and s'_{dag} to their normal forms u_{dag} and u'_{dag} respectively. If $u_{dag} = u'_{dag}$, we are done. Suppose $u_{dag} \neq u'_{dag}$. By proposition 3, there are two tree rewriting derivations using R that rewrite t to u and u' respectively. By proposition 1, $u_{dag} \neq u'_{dag}$ implies that $u \neq u'$. Since R is

confluent, one of u and u' must be reducible using R. By proposition 5, one of u_{dag} and u'_{dag} must be reducible using R_{dag}, but this is contrary to the assumption that both u_{dag} and u'_{dag} are normal forms for R_{dag}. Therefore, we must have $u_{dag} = u'_{dag}$. Confluence is proved. □

Non-ambiguity

We say a TRS is *non-ambiguous* if there are no two rules $l_1 \to r_1$ and $l_2 \to r_2$ in R such that a non-variable subterm of l_1 can be unified with l_2, except that $l_1 \to r_1$ and $l_2 \to r_2$ are the same rule and the non-variable subterm is l_1 itself.

Because unification between dags obtains the same substitution for variables as unification between corresponding trees (see [PW78]), for R and corresponding R_{dag}, if there are two rules $l_{dag} \to r_{dag}$ in R_{dag} and $l'_{dag} \to r'_{dag}$ such that a non-variable subterm of l_{dag} can be unified with l'_{dag}, then we have two rules in $l \to r$ and $l' \to r'$ in R such that a non-variable subterm of l can be unified with l'. Therefore, we have

Proposition 7 *If R is non-ambiguous, then R_{dag} is non-ambiguous.*

Normal Form Computation

We are concerned about how to efficiently compute normal forms for terms given terminating and confluent TRSs. For a terminating and confluent TRS, any term has a unique normal form. We have already shown that if R is terminating and confluent, then R_{dag} is terminating and confluent. Naturally, we would ask if any t has the same normal form using R as t_{dag} has using R_{dag}.

Proposition 8 *If R is terminating and confluent, the normal form of any term t using R is the tree representation of the normal form of t_{dag} using R_{dag}.*

Proof: By proposition 6, R_{dag} is terminating and confluent. t_{dag} has a unique normal form t'_{dag}, that means, there is a dag rewriting derivation D_{dag} using R_{dag} that rewrites t_{dag} to t'_{dag}. By proposition 3, there is a tree rewriting derivation D using R that rewrites t to t'. By proposition 5, t' is irreducible. Since t has only one normal form, t' is the unique normal form of t. □

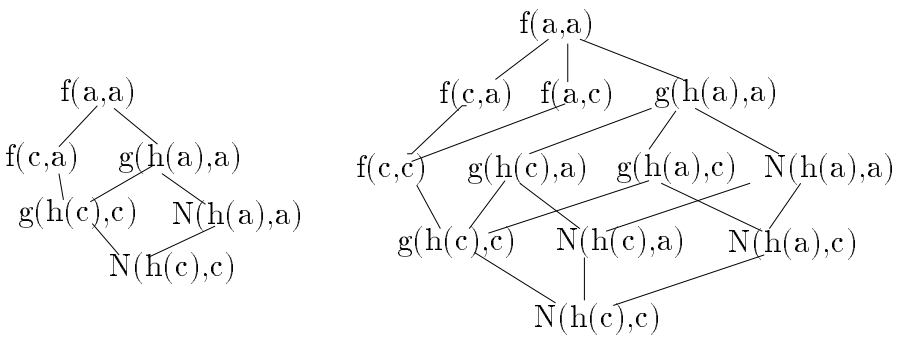
Intuitively, for a terminating and confluent TRS, we can use the corresponding dag TRS to compute the unique normal form for any term more efficiently, since several rewriting steps in tree rewriting are replaced by one dag rewriting step. Staple's speed-up theorem in [Sta80b] implies the following proposition:

Proposition 9 *Let R be a terminating and non-ambiguous TRS, t be any term, $|t|$ the length of the shortest derivation to compute t's normal form, and $|t_{dag}|$ the length of the shortest derivation to compute t_{dag}'s normal form. Then, $|t_{dag}| \leq |t|$.*

Non-left-linear

The "graph rewriting" [BEG87] and "jungle evaluation" [HP88a] have some trouble when they are used for non-left-linear rewriting systems. Our systems can be easily used for non-left-linear systems. The matching algorithm has no problem when dealing with non-linear terms represented by dags [PW78]. In dag representation, the same variables share the same node. Therefore, the dag rewriting for a non-left-linear rule is no different from the dag rewriting for a left-linear. To make clear, we give an example for a non-left-linear system. Meanwhile, to show how dag rewriting saves rewriting steps, we draw out for tree rewriting and dag rewriting all possible derivations that rewrite a given term to its normal form. See figure 4. We simply see that by dag rewriting, we have not only shorter derivations but also a much smaller term space (that are all terms in all derivations). This becomes important when we design strategies for optimization of rewriting. In the following section, we will give several efficient strategies.

Example. System=$\{f(x,x) \to g(h(x),x), g(x,y) \to N(x,y), a \to c, b \to d\}$ and term $f(a,a)$.



dag rewriting tree rewriting

Figure 4

4 Optimization of Dag Rewriting

Dag representation of rewriting systems can speed up the rewriting process, as described in the previous sections. In this section, we will consider how to speed up dag rewriting and propose several strategies for this purpose.

Optimization of rewriting has been discussed on tree representation [HL79]. Some researches [BL79][PMT74] have been done for dag rewriting, but they addressed only recursive programs, other than term rewriting systems that are our target.

Let t be a term and x a variable. $NumVar(x,t)$ denotes the number of occurrences of x in t. We define:

Variable-fewer TRS: For any variable x and any rule $l \to r$, $NumVar(x, l) \geq NumVar(x, r)$.

Variable-more TRS: For any variable x and any rule $l \to r$, $NumVar(x, l) \leq NumVar(x, r)$.

Variable-equal TRS: For any variable x and any rule $l \to r$, $NumVar(x, l) = NumVar(x, r)$.

If a redex is not a proper subterm of any redex, it is called an *outermost redex*. If a redex does not contain any proper subterm as a redex, it is called an *innermost redex*. Let t be a term and t_{sub} a proper subterm of t. The path from the root of t to the root of t_{sub} (excluding the root of t_{sub}) is called the *root path* of t_{sub} in t, denoted by *root-path*(t_{sub}, t). E.g., for term $t = f(g(h(a, f(a, b))), c)$, *root-path*$(f(a, b), t)$ is *f-g-h*, and *root-path*(c, t) is f. Let t be a term, t_{sub} a proper subterm in t, and t_{sub} also a redex. If there is a rule $l \to r$ and a variable x in l such that *root-path*(x, l) matches a substring of *root-path*(t_{sub}, t), we call t_{sub} a *variable redex* (of rule $l \to r$) in t. E.g., TRS=$\{f(g(x), a) \to h(x, d),\ b \to c\}$, in term $t = f(g(g(a, f(a, b))), c)$, b is a redex because of the second rule, *root-path*$(x, f(g(x), a))$ is *f-g* where $f(g(x), a)$ is the left side of the first rule, and *root-path*(b, t) is *f-g-f*, so *f-g* is a substring of *f-g-f* and b is a variable redex in t. Using these definitions, we define the following strategies:

Outermost strategy. To rewrite a term, choose an outermost redex.

Variable-delay strategy. To rewrite a term, if there are non-variable-redexes, choose a non-variable-redex to rewrite; otherwise, all of redexes are variable redexes and we choose a variable redex to rewrite.

Innermost strategy. To rewrite a term, choose an innermost redex.

In the previous sections, a derivation is a sequence of rewriting steps. To simplify terminology, from now on, a derivation is defined as a sequence of rewriting steps to rewrite a term to its normal form. Any two derivations that rewrite the same term to its same normal form are said *equivalent*. An *optimal* derivation is the shortest one among all equivalent derivations. The ith rewriting in derivation D is denoted by $D[i]$. $D[i, j]$ $(i < j)$ denotes all rewritings from $D[i]$ through $D[j]$. A derivation in which at each step the outermost strategy is used is called a *outermost derivation*. The *variable-delay derivation* and *innermost derivation* are defined in the same way.

The paper [Li90] has studied the optimization of tree rewriting for terminating, left-linear and non-ambiguous TRSs. The major results are:

(1) For variable-equal TRS, any derivation is optimal;

(2) For variable-more TRS, an innermost derivation is optimal;

(3) For variable-fewer TRS, any derivation D has an equivalent outermost derivation D' such that $|D'| \leq |D|$;

(4) For variable-fewer TRS, any outermost derivation D has an equivalent variable-delay derivation D' such that $|D'| \leq |D|$.

In this paper, we will consider validity of these strategies on dag rewriting and also extend these results to non-left-linear TRSs. Our discussion is based on the result obtained in the last section that if a tree TRS is terminating and non-ambiguous, then the corresponding dag TRS is terminating and non-ambiguous. We will discuss left-linear TRSs first, and then consider non-left-linear cases. We assume, if without any other comments, that any TRS to be used is terminating and non-ambiguous, and that any TRS and any term are in dag representation.

A rule r is said *root-rewrites* to a term t if the redex is t itself. Let rule r rewrite term t. If r root-rewrites t, we call r is *applied on* t; if r root-rewrites a proper subterm of t, we call r is *applied inside* t; if we don't care whether this rewriting occurs on or inside t, we say r is *applied to* t. Let s be a proper subterm of t. If r root-rewrites a subterm of t which is a proper superterm of s, we say the rewriting *occurs above* s in t. Let s be a proper subterm of term t. Any proper subterm of t whose root is not on the root path of s in t is called a *general sibling* of s.

4.1 Variable-More TRSs

In tree rewriting, for left-linear and variable-more TRSs, any innermost derivation is optimal. But in dag rewriting, for the same subclass of TRSs, any derivation is optimal (no condition "innermost"). That means, for this subclass of TRSs, we don't need to worry about the order of rewritings on redexes, and we can use any derivation to get the normal form with the shortest length.

Lemma 1 *In a derivation constructed by a left-linear and variable-more TRS, any redex does not disappear unless it is rewritten.*

Proof: Suppose D is a derivation constructed by a left-linear and variable-more TRS R, s is a redex after $D[i]$, and $D[i+1]$ rewrites $t(\neq s)$.

Case 1. t is inside s. t must be a variable redex of s. Suppose s is a redex of rule r. Since R is left-linear, after rewriting of t, s remains a redex of rule r.

Case 2. t is above s. s must be a variable redex of t. Since R is variable-more, s remains a redex after rewriting of t.

Case 3. t is a general sibling of s. Obviously s remains a redex after rewriting of t. □

Theorem 1 *In dag rewriting, for a left-linear and variable-more TRS, given any term, any derivation for that term is optimal.*

Proof: Suppose D and D' are two derivations for term t and $|D'| < |D|$. Suppose $D[1,i] = D'[1,i]$ and $D[i+1] \neq D'[i+1]$. $D[i+1]$ rewrites s and $D'[i+1]$ rewrites s'. By lemma 1, s remains a redex until it is rewritten in the subsequent derivation of D'. Suppose s is rewritten in D' by rule r at step $D'[j](i < j)$. All function symbols

(and constants, if there are) in s matched with function symbols in the left side of r are called the *critical part* of s. Note that those subterms in s matched with variables of r are not in the critical part.

Suppose $j = i + k$ (i, j defined above). Let t_{i+p} be the term after step $D'[i+p]$ in D' ($0 \leq p \leq k-1$). Root-rewrite s in t_i by rule r, t_i becoming t'_0. Consider applying $D'[i+1], D'[i+2], ..., D'[i+k-1]$ to t'_0. Let t'_p be the term after rewriting step $D'[i+p](0 \leq p \leq k-1)$. We prove by induction that t_{i+p} and t'_p ($0 \leq p \leq k-1$) are almost the same except the critical part.

Basis. $p = 0$. Remember that t_{i+0} is rewritten to t'_0 by applying rule r to the root of s in t_{i+0}. Because r is variable-more and left-linear, and r is represented by a dag, only the critical part is changed. Hence, the claim holds for the basis.

Induction. Suppose the claim holds for $p-1(0 < p \leq k-1)$. That means, t_{i+p-1} and t'_{p-1} are almost the same except the critical part. Suppose s^* in t_{i+p-1} is root-rewritten at step $D'[i+p]$. We have three cases.

Case 1. s^* is inside s. s^* must be a variable redex of s, which is not in the critical part, that means, s^* is in t'_{p-1}. Hence $D'[i+p]$ can be applied to s^* in t'_{p-1}. Therefore t_{i+p} and t'_p are almost the same except the critical part.

Case 2. s^* is above s. s must be a variable redex of s^*. Suppose s is rewritten by rule r and the position of s^* in t_{i+p-1} is p. Let s^{**} be the subterm in t'_{p-1} at position p. s^* and s^{**} are almost the same except the critical part which is under s, so rule r can be applied on s^{**}. Since s is a variable redex in s^{**} of rule r, the critical part is not changed.

Case 3. s^* is a general sibling of s. Obviously, $D'[i+p]$ can be applied to both t_{i+p-1} and t'_{p-1} and t_{i+p} and t'_p are almost the same except the critical part.

The claim is proved. We have known that t_{i+k-1} and t'_{k-1} are almost the same except the critical part. Note that $D'[j]$ (i.e. $D'[i+k]$) root-rewrites s. The term obtained by this rewriting to t_{j-1} (i.e. t_{i+k-1}) is exactly the same as t'_{k-1} because the critical part is changed. The significance is that if we move the rewriting on s at $D'[j]$ to just in front of $D'[i+1]$, all the rewritings from $D'[i+1]$ through the end of D' are not affected. By this way, we construct a new derivation D'' such that D'' is equivalent to D', $|D''| = |D'|$, and $D''[1, i+1] = D[1, i+1]$. Keeping on this way, we can eventually construct a derivation D''' such that D''' is equivalent to D', $|D'''| = |D'|$, and D''' is a prefix of D. This is impossible since D''' is equivalent to D. □

4.2 Left-Linear TRSs

We have shown that for a left-linear and variable-more TRS, all derivations have the same length. But for a left-linear and variable-fewer TRS, for a term, we may have

derivations to compute its normal form with different lengths. The reason is that some variable (say x) which is in the left side of a rule does not occur in the right side of the rule, so that when the rule is applied, the redex matched by x may disappear without rewriting and different order of rewritings have different effects. For example, given system $\{f(x,y) \to N, a \to b\}$ and term $f(a,b)$, we have two derivations with different lengths: $f(a,b) \to N$ and $f(a,b) \to f(b,b) \to N$.

In [Li90], we have shown that for left-linear and variable-fewer any derivation has an equivalent outermost derivation which has no more rewriting steps. In other words, for any derivation, we may find an equivalent outermost derivation which are more efficient. With dag representation, we can lift the condition "variable-fewer" to get the same result, that means, the result is valid for any left-linear TRS (don't forget non-ambiguity condition).

Theorem 2 *In dag rewriting, for a left-linear TRS, any derivation D has an equivalent outermost derivation D' such that $|D'| \leq |D|$.*

Proof: Let R be a linear TRS, D a derivation, and tail-length(D) the length of the subderivation in D from the first non-outermost rewriting through the end.

We use induction on tail-length(D) to prove:

Claim: D is equivalent to an outermost derivation D' such that tail-length(D')=0 and $|D'| \leq |D|$.

Basis. tail-length(D)=0. Trivial.
Induction. Suppose the claim holds for the derivation with smaller tail-length function value than D. Suppose the first non-outermost rewriting occurs on the redex s. There must be an outermost redex s_{out} containing s properly. Since R is non-ambiguous, s must be a variable redex in s_{out}. Suppose the rule appliable on s_{out} is r which has variables $x_1, ..., x_k$ at the left side and the subterms matched with $x_1, ..., x_k$ are $s_1, ..., s_k$. s is a subterm of one of $\{s_1, ..., s_k\}$. Suppose D_{tail} is the subderivation of D from the rewriting of s through the end. We discuss in cases what rewriting will happen to s_{out} or above s_{out} in D_{tail}. Let D_{head} denote the remaining subderivation in D except D_{tail}.

Case 1. All rewritings to s_{out} or above are applied only on or inside $s_1, ..., s_k$. This case is impossible since r will be appliable, even in the case that R is non-left-linear. Because identical terms will be rewritten to a same term by the confluence property. In the following cases, suppose r' is the first rule applied on a redex that is not a subterm of any one of $s_1, ..., s_k$ and r' is applied to or above s_{out} .

Case 2. r' is applied on a proper subterm of s_{out}. r' superposes r, contrary to the condition that R is non-ambiguous, so this case is impossible.

Case 3. r' is applied on s_{out}. Before r' is applied, all rewritings to s_{out} must be on or inside $s_1, ..., s_k$ by case 2. Suppose s_i becomes $s_i'(1 \leq i \leq k)$ and s_{out} becomes s_{out}'

just before r' is applied.

Because R is left-linear, obviously x_i matches $s'_i (1 \leq i \leq k)$ and r can be applied on s'_{out}. Since R is non-ambiguous, only one rule can be applied to a term. Therefore, $r' = r$.

We move rewriting by r in front of the rewriting on s, D becoming D'. We suppose that in D the rewriting on s is $D[i]$ and the rewriting on s_{out} is $D[j] (i < j)$. Suppose s_{out} is rewritten to s''_{out} by r in D' and s'_{out} is rewritten to s'''_{out} by r in D. s''_{out} and s'''_{out} are the same except the changes to $s_1, ..., s_k$. Since R is left-linear and is represented by dags such that the different occurrences of the same variable at right sides of rules point to the same node, we can assume the distinct variables at the right side of r are $x_{i_1}, ..., x_{i_j} (1 \leq i_1 < ... < i_j \leq k)$. In D', all $s_{i_1}, ..., s_{i_j}$ are rewritten to $s'_{i_1}, ..., s'_{i_k}$ by the same rewritings between $D[i]$ and $D[j]$ in D. All other rewritings between $D[i]$ and $D[j]$ occur on the general siblings of s_{out} and they are not affected by moving of the rewriting on s_{out}, so they can be applied in D'. Hence, D' is equivalent D and $|D'| \leq |D|$. tail-length(D') < tail-length(D). By induction hypothesis, the claim holds for D'.

Case 4. r' is applied above s_{out}. By case 2, before r' is applied, all rewriting to s_{out} must be on or inside $s_1, ..., s_k$. If s_{out} were not a variable redex of r', r would superpose r'. Therefore s_{out} is a variable redex of r'. Suppose the term being rewritten by r' is t. By non-ambiguity, all proper subterms in t not corresponding to variables of r' are normal forms. Suppose derivation D_{sub} reduces a proper subterm t' of t to its normal form and one step of D_{sub} is in D_{tail}. Obviously, tail-length(D_{sub}) < tail-length(D). By induction hypothesis, we can obtain an outermost derivation D'_{sub} equivalent to D_{sub} and $|D'_{sub}| \leq |D_{sub}|$. Since t' must be a general sibling of s_{out} and s, all rewritings to t' has no effect to s, so we can move all steps of D'_{sub} in front of the rewriting step of s, and obtain a derivation equivalent to D with smaller tail-length function value. Using the induction hypothesis again for the modified D, we can prove the claim holds for D. Therefore we can assume that in D, all derivation steps used in rewriting all proper subterms in t not corresponding to the variables in r' are in D_{head}. In this case, r' is appliable at the beginning of D_{tail}, but this is contrary to the supposition that r is an outermost redex at the beginning of D_{tail}.

We have enumerated all cases. The claim does hold. The theorem immediately follows the claim. □

Even outermost strategy is not able to guarantee to generate the optimal derivation. One of examples is as follows: system $f(x, b) \to N, a \to b, c \to d$ and term $f(c, a)$. Both c and a are outermost redexes, but which one is rewritten first has effect on the length of derivation. c first: $f(c, a) \to f(d, a) \to f(d, b) \to N$ and a first: $f(c, a) \to f(c, b) \to N$.

The variable-delay strategy, the strategy better than outermost strategy, is given

in [Li90] for left-linear and variable-fewer TRSs. With dag representation, the "variable-fewer" condition can be lifted, so that we have a more general result for variable strategy. Before we go to the formal description, we need a lemma.

Lemma 2 *Let D be an outermost derivation of a non-ambiguous TRS and t a non-variable outermost redex after some steps of D. In D, t remains a non-variable outermost redex until it is rewritten.*

Proof: Suppose after $D[i]$ in D, t is a non-variable outermost redex. If t is no longer a non-variable outermost redex after $D[i+1]$, there are only two possibilities: t itself is rewritten at step $D[i+1]$ or a redex t' which is above t is rewritten at step $D[i+1]$. For the lemma, we only need to discuss the latter case. Because t is a non-variable redex, t is unifiable with a non-variable subterm of t', which is contrary to the condition that the TRS is non-ambiguous. □

Theorem 3 *In dag rewriting, for a left-linear TRS, any outermost derivation D has an equivalent variable-delay derivation D' such that $|D'| \leq |D|$.*

Proof: . Let R be a linear TRS, D an outermost derivation, and tail-length(D) denote the length of the subderivation of D from the first non-variable-delay rewriting through the end.

We use induction on tail-length(D) to prove:

Claim: D is equivalent to a variable-delay derivation D' such that tail-length(D')=0 and $|D'| \leq |D|$.

Basis. tail-var-length(D)=0. Trivial.
Induction. Suppose the claim holds for the outermost derivation with smaller tail-length function value than D. Suppose after $D[i]$ in D there is a non-variable outermost t_{non} and a variable outermost t_{var} and the latter is rewritten at step $D[i]$. Before $D[i]$, all derivation steps are applied by variable-delay strategy. By lemma 2, t_{non} will be rewritten in D at step $D[j]$ $(i < j)$. Construct an intermediate derivation as follows.

D'' : Move rewriting on t_{non} in front of the rewriting on t_{var}.

All rewritings between $D[i]$ and $D[j]$ in D can be done between $D'[i+1]$ and $D'[j]$ in D', because those redexes being rewritten between the steps are all outermost and they cannot be inside or above t_{non}, so the rewriting on t_{non} does not affect them. Therefore D'' is equivalent to D and has the same length. By theorem 2, the subderivation between $D''[i+1]$ and the end has an equivalent outermost derivation with equal or smaller length, say it is D_{tail}. Concatenating $D''[1,i]$ with with D_{tail}, we obtain D', which is equivalent to D, is outermost, and has smaller tail-length function value. By induction hypothesis, the claim holds for D'. □

4.3 Non-Left-Linear TRSs

For term rewriting, one of the great advantages of dag representation is that different occurrences of a variable in a term point to the same node. In the last subsection, taking advantage that several occurrences of a variable at right side of rules share one node, we showed that any derivation is optimal for non-ambiguous and variable-more TRSs and that outermost strategy and variable-delay are good candidates for efficient normal form computation for any non-ambiguous TRS. In this subsection, we will take advantage that the different occurrences of a variable at left side of rules share one node, which will lead us to new discoveries for non-left-linear TRSs. We have noted that most of research on efficient term rewriting before focused on left-linear TRSs.

Let rule $l \to r$ rewrite term t and l be a non-linear term. Specifically, suppose variable x occurs in l multiple times. In dag representation, all occurrences of x in l share one node. When l matches a subterm t' of t, all occurrences of x in l match a same subterm of t'. Due to this fact, all discussions on rewriting by left-linear TRSs in the last subsection are valid for rewriting by non-left-linear TRSs. Let's see an example. Most of proofs in the last subsection used "swapping" of two rewritings: Let R be a non-ambiguous and left-linear TRS, t a redex and t' a redex inside t, i.e., t' is a proper subterm of t. By non-ambiguity of R, t' must be a variable redex of t. Since R is left-linear, the variables at left side of rules are different and they may match different subterms. No matter when t' is rewritten, t is always a redex. That means, rewriting on t' has no effect on rewriting on t. Hence, rewritings on t and t' can be swapped in any order. In dag rewriting, all occurrences of a variable at left side of a rule match a same subterm. Rewriting on t' has no effect on rewriting on t, so they can be swapped in any order. By comparison, if a non-left-linear TRS is represented by trees, we cannot do this "swapping". For example, given system $\{f(x,x) \to N, a \to b\}$ and term $f(a,a)$. $t = f(a,a)$ is a redex and t' = left son a of f is a redex inside t. If t' is rewritten to b, we don't have any redex of the form $f(...)$ furthermore.

The advantage of dag representation for non-left-linear TRSs described above can be used in all proofs for the non-left-linear version of theorem 1, theorem 2, and theorem 3. That is, the "left-linear" condition can be lifted. We don't repeat the proofs here. Therefore, we have

Theorem 4 *In dag rewriting, for a non-ambiguous and variable-more TRS, given any term, any derivation for that term is optimal.*

Theorem 5 *In dag rewriting, for a non-ambiguous TRS, any derivation D has an equivalent outermost derivation D' such that $|D'| \leq |D|$.*

Theorem 6 *In dag rewriting, for a non-ambiguous TRS, any outermost derivation D has an equivalent variable-delay derivation D' such that $|D'| \leq |D|$.*

5 Conclusions

We have designed a dag representation for terms and term rewriting systems, which save space and rewriting steps as most as possible in the view that redundant space and rewritings are avoided. The term represented by a tree is uniquely mapped to a dag. This simple corresponding relationship between tree representation and dag representation provides us a clear base to establish relationship for important properties of rewriting system between tree rewriting and dag rewriting. We have provided efficient strategies for dag rewriting. With dag representation, we have designed efficient strategies for non-left-linear rewriting systems.

We have given efficiency analysis for each step in rewriting a term. But more complexity analysis for comparing tree rewriting and dag rewriting is interesting. In general, the more number of subterms are shared, the more efficient the dag rewriting is. We may consider about the average complexity. Matching two dags is efficient. How to find a redex in a dag term is another interesting topic. Since occurrences of subterms in a term are much fewer if the term is represented by a dag, the redex finding process will be much more efficient.

Acknowledgements

I would like to thank Zvi Kedem for guide and support, and Krishna Palem for helpful discussions.

References

[BEG87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glaueert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep, "Term graph rewriting," *PARLE'87 (Parallel Architecture and Language Europe)*, Vol. II, LNCS 259, pp.141-158, 1987.

[BL79] G. Berry and J.-J. Levy, "Minimal and optimal computation of recursive programs", *JACM* 26, pp.148-175, 1979.

[CKS87] C.Choppy, S.Kaplan and M.Soria, "Algorithmic complexity of term rewriting systems," *2nd Int. Conf. on Rewriting Techniques and Applications*, pp.256-270, 1987.

[Der85] N. Dershowitz, "Computing with Rewrite Systems," *Information and Control* 65, pp.122-157, 1985.

[GM86] J.A. Goguen and J. Meseguer, "EQLOG: Equality, types and generic modules for logic programming", in (D. DeGroot and G. Lindstrom, Eds.) *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, Englewood Cliffs, NJ, pp.295-363, 1986.

[HKP88] A. Habel, H.-J. Kreowski and D. Plump, "Jungle evaluation," *Fifth workshop on Specification of Abstract Data Types*, *LNCS 332*, pp.92-112, 1988.

[HL79] G. Huet and J.-J. Levy, "Computations in nonambiguous linear rewriting systems", INRIA Tec. Report 359, 1979.

[HP88a] B. Hoffmann and D. Plump, "Jungle evaluation for efficient term rewriting," *International Workshop on Algebraic and Logic Programming, LNCS 343*, pp.191-203, 1988.

[HP88b] B. Hoffman and D. Plump, "Jungle evaluation for efficient term rewriting," Report No. 4/88, Fachbereich Mathematik und Informatik, Universitat Bremen, 1988.

[KB70] D.E. Knuth and P.B. Bendix, "Simple Word Problems in Universal Algebras," in (Leech, J. ed.) *Computational Problems in Abstract Algebra*, Pengamon, Oxford U.K., pp.263-297, 1970.

[Hu80] J.M. Hullot, "Canonical forms and unification," *5th Conf. on Automated Deduction, LNCS 87*, pp.318-334, 1980.

[Li90] K. Li, "Complexity of rewriting and optimization of rewriting," *Proc. of 2nd Conf. on Algebraic and Logic Programming*, available as in LNCS, 1990.

[PMT74] G. Pacini, C. Montangero and F. Turini, "Graph representation and computation rules for typeless recursive languages," *ICALP'74, LNCS 14*, pp.157-169, 1974.

[PW78] M.S. Paterson and M.N. Wegman, "Linear Unification", *JCSS 16*, pp.158-167, 1978.

[Rob71] J.A. Robinson, "Computational logic: the unification computation," *Machine Intelligence 6*, pp.63-72, 1971.

[Sta80b] J. Staples, "Speeding up subtree replacement systems," *J. of Theor. Comp. Sci. 11*, pp.39-47, 1980.

[SY86] P.A. Subrahmanyam and J. You, "FUNLOG: A computational model integrating logic programming and functional programming", in (D. DeGroot and G. Lindstrom, Eds.) *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, Englewood Cliffs, NJ, pp.157-198, 1986.