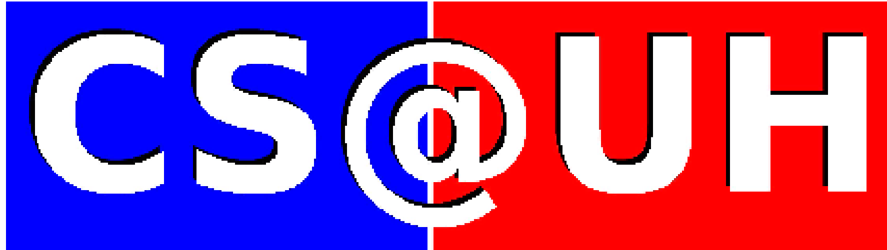


The Laboratory for Rapid Rewriting Version 3.0[1]

Rakesh M. Verma and Wei Guo

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

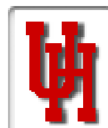
Technical Report Number UH-CS-12-01

May 10, 2012

Keywords: Interpreters for Rewriting, Memorization, Congruence-closure based rewriting, Term Graphs

Abstract

In this paper, we present the design and performance of the Laboratory for Rapid Rewriting system, LRR. Given a convergent or orthogonal rewrite system, R, and a term t, LRR computes the normal form of t whenever it exists. LRR consists of two interpreters: Smaran and TGR, which stands for Term Graph Rewriter. Both Smaran and TGR use term graphs with varying amounts of sharing. Smaran also stores the history of all rule applications in a very efficient data structure. A number of optimizations have been initiated in the implementation of LRR including a preprocessor for rules and the DS-list data structure. We give an overview of how to use the system, its core algorithms, data structures, optimizations and features. The performance of LRR on some benchmarks both favorable and unfavorable is presented and compared with two other interpreters Maude and Elan.



[1] Research supported in part by NSF grant CCF 0306475 and DUE 1062954

The Laboratory for Rapid Rewriting Version 3.0

Rakesh M. Verma and Wei Guo

University of Houston Department of Computer Science
4800 Calhoun Rd., Houston, TX 77004, USA
rmverma@cs.uh.edu
http://www.cs.uh.edu/~rmverma

Abstract. In this paper, we present the design and performance of the Laboratory for Rapid Rewriting system, LRR. Given a convergent or orthogonal rewrite system, R, and a term t, LRR computes the normal form of t whenever it exists. LRR consists of two interpreters: Smaran and TGR, which stands for Term Graph Rewriter. Both Smaran and TGR use term graphs with varying amounts of sharing. Smaran also stores the history of all rule applications in a very efficient data structure. A number of optimizations have been initiated in the implementation of LRR including a preprocessor for rules and the DS-list data structure. We give an overview of how to use the system, its core algorithms, data structures, optimizations and features. The performance of LRR on some benchmarks both favorable and unfavorable is presented and compared with two other interpreters Maude and Elan.

1 Introduction

Fast rewriting is needed for equational programming, rewrite based formal verification methods, and symbolic computing systems such as Maple/Mathematica. In any implementation of rewriting techniques efficiency is a critical issue [Hermann 91]. At the University of Houston (UH) we have been developing and evaluating the LRR system for fast rewriting. There are several motivations for LRR:

1. Theorem proving/formal verification – we have used LRR in a Knuth-Bendix completion procedure [Verma 99], and we have recently implemented CTL model checking with it.
2. As a testbed for innovating rewriting algorithms that are fast and efficient in practice – for example, we have experimented with different reduction strategies as described below and also different choices of data structures for congruence closure.
3. As an educational tool for teaching undergraduate and graduate students in several courses – in the declarative programming course at UH, students use it to learn equational programming and in the automata theory course we use it to illustrate tree automata. We have added a graphical interface to LRR called RuleMaker [Yu 08], so that students can draw finite state string or tree automata and run them. RuleMaker provides fine-grained control over LRR so that the student can view the results of every step. The advantage

of LRR is that it can be easily programmed for all kinds of automata instead of developing packages for specific automata from scratch.

In this paper, we give a self-contained introduction to LRR, highlighting its key original features and present its main innovations. These include: a tree interpreter Tree, more reduction strategies, an implementation of new data structure for speeding up normalization called DS-list and a preprocessor for rules. The DS-list and the preprocessor can be used to speed up any rewriting system. The performance of LRR on several benchmarks is given. A Linux version of LRR 3.0 and some examples can be downloaded from the first author's web page.

LRR 3.0 consists of a pure-tree (i.e., no sharing of common subexpressions) interpreter Tree, a term graph interpreter TGR, and a term graph rewriter that tables or stores the history of its reductions, called Smaran, based on the congruence closure normalization algorithm. TGR shares those subexpressions that match different occurrences of the same variable whereas Smaran has full sharing of common subexpressions. This algorithm treats rules as equations, hence it keeps equivalence classes of terms. Terms are represented implicitly via *signatures* and there is at most one special signature in each class called the *unreduced signature* of the class (for theoretical justification of the details of this algorithm please see [Verma 95,Bachmair 99]). The tabling component of LRR is useful in applications involving certain kinds of nonterminating systems including fixed-point computations, in dynamic programming, and in retracing/debugging, etc. The input to LRR is a program representing a convergent rewrite system (a different version allows orthogonal systems) and an input term. Similar to algebraic specification languages like ASF+SDF [Brand 02], ELAN [Borovansky 02] and Maude [Clavel 03] a program is composed from modules. Each module defines its own signature (set of variable, function symbols and constants) and rewriting rules. A module can import other modules. Terms in LRR are written in prefix form. LRR contains some predefined common datatypes such as, integers, booleans, strings, sets, etc. and operators including arithmetic, set, comparison, and logical operators. The efficient integration of these diverse datatypes into history-based rewriting is a challenging issue that we believe has been solved quite successfully in LRR.

LRR also includes a variant detector that can determine if a new term is an alphabetic variant of an existing term, which is usable with the history option. If so, the appropriate variant of the result computed for the existing term is used for further rewriting instead of starting from scratch. LRR provides a set of commands so that it can be called by other systems for symbolic computation. This currently requires the UNIX message passing mechanism. This feature of LRR was used to develop a system called KBS, which integrated an "off-the-shelf" Knuth-Bendix procedure with an earlier version of Smaran.

Operators can also be declared AC in LRR which supports AC matching of left-linear rules.

The rest of this paper is organized as follows. We first present some preliminaries including how to use LRR in Section 2. In Section 3, we describe the main algorithms in LRR and, in Section 4, we outline the data structures and

optimizations that have already been implemented. The experimental results are presented in Section 5. We conclude the paper with some promising directions for future research. Finally, we give an example to illustrate an optimization in LRR in the Appendix, which also includes the rules for the benchmarks on which LRR is compared with two other interpreters.

2 Preliminaries

A *signature* is a set $\mathcal{F}$ along with a function *arity*: $\mathcal{F} \rightarrow \mathbb{N}$. Members of $\mathcal{F}$ are called *function symbols*, and $arity(f)$ is called the *arity* of the function symbol f. Function symbols of arity zero are called *constants*. Let X be a countable set disjoint from $\mathcal{F}$ that we shall call the set of *variables*. The set $\mathcal{T}(\mathcal{F}, X)$ of *$\mathcal{F}$-terms over X* is defined to be the smallest set that contains X and has the property that $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, X)$ whenever $f \in \mathcal{F}$, $n = arity(f)$, and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, X)$. The set of function symbols with arity n will be denoted by $\mathcal{F}_n$; in particular, the set of constants is denoted by $\mathcal{F}_0$. We will use $root(t)$ to refer to the outermost function symbol of t. We will use $Var(t)$ to denote the set of variables appearing as subterms of a term t.

The *size*, $|t|$, of a term t is the number of occurrences of variables and function symbols in t. So, $|t| = 1$ if t is a variable, and $|t| = 1 + \Sigma_{i=1}^{n}|t_i|$ if $t = f(t_1, \ldots, t_n)$. The *height* of a term t is 0 if t is a constant or a variable, and $1 + max\{height(t_1), \ldots, height(t_n)\}$ if $t = f(t_1, \ldots, t_n)$.

A *position* of a term t is a sequence of natural numbers that is used to identify the locations of subterms of t. The subterm of $t = f(s_1, \ldots, s_n)$ at position p, denoted $t|_p$, is defined recursively: $t|_\lambda = t$, where λ is the empty sequence, $t|_k = s_k$, and $t|_{k.l} = (t|_k)|_l$ for $1 \leq k \leq n$ and undefined otherwise [Radcliffe 10].

A *substitution* is a mapping $\sigma : X \rightarrow \mathcal{T}(\mathcal{F}, X)$ that is the identity on all but finitely many elements of X. Substitutions are generally extended to a homomorphism on $\mathcal{T}(\mathcal{F}, X)$ in the following way: if $t = f(t_1, \ldots, t_k)$, then (abusing notation) $\sigma(t) = f(\sigma(t_1), \ldots, \sigma(t_k))$. Oftentimes, the application of a substitution to a term is written in postfix notation. A term s *matches* a term t if there is a substitution σ such that $s\sigma = t$. Two terms s and t *unify* if there is a substitution σ such that $s\sigma = t\sigma$.

2.1 Term Rewrite Systems

A *rewrite rule* is a pair of terms, (l, r), usually written $l \rightarrow r$, where r does not contain any variables that do not appear in l (notice that if l is a constant, then r cannot contain any variables). For the rule $l \rightarrow r$, the *left-hand side* (LHS) is l, and the *right-hand side* (RHS) is r. Notice that if $l \rightarrow r$ and $l' \rightarrow r'$ are rules in some rule set, then we will assume (without loss of generality) that $(Var(l) \cup Var(r)) \cap (Var(l') \cup Var(r')) = \emptyset$. A rule, $l \rightarrow r$, can be applied to a term, t, if there exists a substitution, σ, such that $l\sigma = t'$, where t' is a subterm of t; in this case, t is rewritten by replacing the subterm $t' = l\sigma$ with $r\sigma$. The

process of replacing the subterm $l\sigma$ with $r\sigma$ is called a *rewrite*. A *root rewrite* is a rewrite where $t' = t$.

A *term rewrite system* (or *TRS*) is a pair, $(\mathcal{T}, R)$, where R is a finite set of rules and $\mathcal{T}$ is the set of terms over some signature, Σ. When the set of terms is clear from the context, we usually omit it and just refer to R itself as a term rewrite system. If we think of $\rightarrow$ as a relation, then $\stackrel{+}{\rightarrow}$ and $\stackrel{*}{\rightarrow}$ will denote its transitive closure, and reflexive and transitive closure, respectively. We will denote a root rewrite by $s \stackrel{r}{\rightarrow} t$ and a non-root rewrite by $s \stackrel{nr}{\rightarrow} t$. A *derivation* is a sequence of terms, $t_1, \ldots, t_n$, such that $t_i \rightarrow t_{i+1}$ for $i = 1, \ldots, n-1$; this sequence is often denoted by $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$. Given a rewrite system R, a term t is said to be in *normal form*, or a normal form, if no rule of R can be applied to it.

The input to LRR is a term rewriting system R and a given term t_0. The objective is to compute a normal form of t_0, t_n. We denote the ith rule as $rule_i : lhs_i \Rightarrow rhs_i$. We define that *the* i^{th} *step of the normalization* is a process that builds a new term t_i by applying $rule_j$ at a subterm of term t_{i-1}, in which $i \in \mathbb{N}, 0 < i \leq n$. We use $t_{i-1} \rightarrow_{(i,j)} t_i$ to denote the ith step of the normalization. Thus, the whole process of normalization can be denoted as a sequence, $t_0 \rightarrow_{(1,j)} t_1, ..., t_i \rightarrow_{(i+1,j')} t_{i+1}, ..., t_{n-1} \rightarrow_{(n,j'')} t_n$. Terms $t_1, ..., t_i, ..., t_{n-1}$ are called *intermediate results* [Verma and Guo 11b].

A *defined symbol* is a symbol that occurs as the root of some LHS in R. The rest of the symbols are called *constructors*. Note that predefined symbol such as mathematical, relational and set operators, etc., are neither defined symbols nor constructors since they will be evaluated eventually.

A rule $l \rightarrow r$ is *left-linear* if every variable appears no more than once in l and a rewrite system is left-linear if every rule is left-linear. Two rules $l \rightarrow r$ and $L \rightarrow R$ are *overlapping* if a non-variable subterm of l unifies with L. If the two rules are the same, then the subterm must be a proper subterm. A rewrite system is *orthogonal* if it is left-linear and non-overlapping. A rewrite system is *confluent* if for all terms s, t and u whenever $s \stackrel{*}{\rightarrow} t$ and $s \stackrel{*}{\rightarrow} u$, then there is a term v such that $t \stackrel{*}{\rightarrow} v$ and $u \stackrel{*}{\rightarrow} v$. Confluence implies that normal forms of terms are unique whenever they exist. A rewrite system is *terminating* if every rewrite sequence starting from every term is finite. A *convergent* rewrite system is both terminating and confluent. For more notions of rewriting we refer the reader to the excellent survey [Dershowitz 01].

2.2 Running LRR

The input of LRR is a module file with a .m extension representing the rules R and a term file with a .t extension representing the given term t_0. On the command prompt of a linux machine, use the command below to run LRR.

./lrr [OPTIONS]... MODULEFILE TERMFILE

And ./lrr - -help for help.

An LRR program contains one or more modules which may be stored in one or more module files. Each module has a unique module name. LRR is case

sensitive and all reserved words are in lower case. The syntax of a module file is as follows.

```
module   MODULENAME
rem      COMMENTS ;
import   IMPORTLIST ;
export   EXPORTLIST ;
var      VARLIST ;
func     FUNCLIST ;
rule     RULELIST ;
end module  MODULENAME
```

A module may contains IMPORT, EXPORT, VAR, FUNC and RULE sections the whose order cannot be changed. The MODULENAME given in "module" and "end module" must be the same. All sections are terminated by a semicolon.

IMPORTLIST is a list of names of other module files without the ".m" extension. The modules in the files will be imported. Filenames are separated by commas.

EXPORTLIST is a list of identifiers that are visible to other module files. The identifiers can be variables, constants, and functions separated by commas. Other modules must import this module to use the identifiers in the EXPORTLIST.

VARLIST is a list of names of variables separated by commas.

FUNCLIST is a list of names of functions separated by commas. Constants are defined here as functions with zero arity. The arity is defined in the parentheses after the function name. A function "f" with three parameters is defined below.

```
f ( 3 )
```

Functions of arity two can be declared associative and commutative (AC) in LRR using the reserved word ac after the list of functions. Predefined functions will be overridden if they are redefined in this section.

RULELIST is a list of rules separated by semicolons. Each rule is of the form below.

```
LeftHandSide => RightHandSide ;
```

Basic operators and functions are written in prefix order. Every variable used in the RHS must appear in the LHS. The module file for Fibonacci calculator is below.

```
module fib
rem fibonacci calculator;
import ;
export fib ;
var x;
func f(2), fib(1) ;
```

```
memo f, fib ;
rule    fib(x) => f(>(x,1),x) ;
        f(true,x) => +(fib(-(x,1)),fib(-(x,2))) ;
        f(false,x) => 1 ;
end module fib
```

Term files define the given terms. For example, a term file for Fibonacci calculator could contain the term

```
fib(20)
```

Basic datatypes and predefined identifiers are built into LRR. Integer, float, boolean, char, set, and untyped are supported datatypes. Predefined functions consist of set operations including union, intersection, insertion, deletion, membership, getting the nth element, counting the number of elements; regular arithmetic operators including addition, subtraction, multiplication, division, remainder, increment by one, decrement by one; comparison operators including greater than, less than, greater or equal, less or equal, equal, not equal and logical operators including and, or, xor, and not. Apart from integers and reals written in decimal notation, the predefined constants include $true$ and $false$.

3 Core Algorithms

3.1 Tree and TGR

In Tree, the given term, intermediate results and the normal form are all stored as trees, which do not allow any sharing. Tree is the slowest algorithm in LRR. It is used as a reference point and for the applications in which the semantics of the rules would be affected by sharing.

TGR is similar to Tree but it allows sharing. For TGR, LRR uses Directed Acyclic Graphs (DAGs), not trees, to represent $t_0, t_1, ..., t_n$ so that the terms that match different occurrences of the same variable are shared.

3.2 Smaran

Smaran is the tabling component of LRR. The basic algorithm of Smaran extends the well-known congruence closure algorithm (CCA) for ground equations.

CCA divides the set of terms into numbered equivalence classes E. Membership of a term in an equivalence class is determined by its *signature* s. The signature of a term $f(t_1, ..., t_n)$ is the tuple $\langle f \ \#[t_1] \ ... \ \#[t_n] \rangle$, in which $\#[t_i]$ is the number of the equivalence class containing the signature representing t_i. We define that equivalence class E *represents* t if E contains a signature representing t. CCA operates by merging equivalence classes representing terms whose equivalence follows from the given equations [Verma 00].

To extend CCA for reduction we use the concept of a distinguished signature in every equivalence class named the *unreduced signature* [Chew 80], [Verma 89]

which is used to construct a distinguished term. It has been shown in [Chew 80], [Verma 89] that it is enough to examine this term to select useful rule instances, and that it is sufficient to check this term for irreducibility [Verma 00]. If it is irreducible, the class representing the term contains a normal form. Hence an instance of a LHS represented by a reduced signature does not result in any progress towards normal form. Also, terms represented by reduced signatures cannot be normal forms. Whenever there is a substitution σ such that a rule $lhs_i \Rightarrow rhs_i$ matches the distinguished term $t = \sigma(lhs_i)$ of a class, E, the signature representing t is marked reduced in E and the class representing $t' = \sigma(rhs_i)$ (if any) is merged with E. If there is no class representing t', Smaran constructs a signature representing t', inserts the signature into E, and marks it the unreduced signature of E.

Smaran starts by constructing the signature s of t_0. Then s is inserted into a class and marked the unreduced signature of the class. Smaran tracks the number of this class throughout the process of reduction. Signatures of terms are constructed bottom-up. To illustrate the algorithm, we use Fibonacci calculator as an example. We number the rules in a top-down order for convenience and use the symbol '*' to indicate unreduced signature. Let $t_0 = fib(2)$.

The initial set of classes is:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle^*\}$$

In the 1st step of normalization, LRR calls matching function to find a match between the unreduced signature of any class and the LHS of any rule and a match between class 1 and lhs_1 occurs. While building the instance of the RHS rhs_1, a signature representing the instance $\sigma(rhs_1)$ is created and inserted into class 1 as its unreduced signature. Here we do not show signatures, related to the built-in datatypes that can be evaluated directly and LRR also does not store them in equivalence classes. At the end of 1st step, the classes are below:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle^*\} \quad 2 : \{1^*\} \quad 3 : \{true^*\}$$

In the 2nd step, class 1 matches lhs_2. The instance of rhs_2 is $fib(1) + fib(0)$ which cannot be evaluated. The signature representing this is constructed, inserted into class 1 and marked as its unreduced signature. At the end of 2nd step classes are below:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 4,\ 6\rangle^*\} \quad 2 : \{1^*\} \quad 3 : \{true^*\}$$

$$4 : \{\langle fib\ 2\rangle^*\} \quad 5 : \{0^*\} \quad 6 : \{\langle fib\ 5\rangle^*\}$$

In the 3rd step, class 4 matches lhs_1, and in the 4th step, class 4 matches lhs_3. The term $fib(1)$ which is represented by class 4 reduces to 1 represented by class 2. Thus class 4 and class 2 are merged into, say 2. The classes at the end of the 4th step are below:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 2,\ 6\rangle^*\} \quad 2 : \{\langle fib\ 2\rangle,\ \langle f\ 7\ 2\rangle,\ 1^*\}$$

$$3 : \{true^*\} \quad 5 : \{0^*\} \quad 6 : \{\langle fib\, 5\rangle^*\} \quad 7 : \{false^*\}$$

Note LRR has updated the signatures which contains class 4 to contain class 2. After the 5th and the 6th steps, the term $fib(0)$, which is represented by class 6 reduces to 1 represented by class 2. Thus class 6 and class 2 are merged, say into 2. Now we have:

$$0 : \{2^*\} \quad 1 : \{\langle fib\, 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 2,\ 2\rangle^*\}$$

$$2 : \{\langle fib\, 2\rangle,\ \langle fib\, 5\rangle,\ \langle f\ 7\ 2\rangle,\ \langle f\ 7\ 5\rangle,\ 1^*\} \quad 3 : \{true^*\} \quad 5 : \{0^*\} \quad 7 : \{false^*\}$$

The unreduced signature of class 1 can be evaluated to term 2, which is in class 0. Thus class 1 and class 0 are merged, say into 0. At the end of the 6th step, we get:

$$0 : \{2^*,\ \langle fib\, 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 2,\ 2\rangle\} \quad 2 : \{\langle fib\, 2\rangle,\ \langle fib\, 5\rangle,\ \langle f\ 7\ 2\rangle,\ \langle f\ 7\ 5\rangle,\ 1^*\}$$

$$3 : \{true^*\} \quad 5 : \{0^*\} \quad 7 : \{false^*\}$$

No more matches are found in LRR. Hence Smaran checks for the existence of a normal form of t_0. The unreduced signature of class 1 is 2 which is irreducible. Therefore, the normal form of $fib(2)$ is 2. Note that LRR needs no more computation to reduce $fib(fib(2))$ because it is represented by the signature $\langle fib\ 0\rangle$ in class 0. Its normal form is also 2. On the other hand, an interpreter that does not store history would calculate $fib(2)$ twice to get the normal form. The compact data structure helps exploit the advantages of storing history and can also speed up normalization [Verma 00].

3.3 Reduction Strategies

There are six reduction strategies in version 3.0: (i) the original reduction strategies for Smaran and TGR, (ii) an efficient version of leftmost-outermost for left-linear rules for Tree and leftmost-outer for TGR, Smaran, (iii) a hybrid version of aspects of Smaran original strategy and leftmost-outer strategy for Smaran, (iv) the DS-list strategy which will be described in detail in Section 4, (v) ALU-list strategy which will be discussed in Section 4, and (vi) a combination of DS-list and ALU-list for Smaran and TGR.

The original reduction strategy for Smaran and TGR on a successful match immediately attempts to reduce the instance of the RHS and its descendants. If LRR finds no match, it backtracks all the way to the root of intermediate results.

The leftmost-outermost reduction strategy for Tree is a pure strategy which upon a successful match does not attempt to reduce the instance of the RHS and immediately backtracks to the node m levels up, where $m = min(max\{height(lhs)|lhs \rightarrow rhs \in R\}, level\ of\ rhs\ instance)$. It correctly implements leftmost-outermost in pure-tree but not necessarily outermost in TGR or Smaran due to sharing.

The hybrid version of original Smaran and leftmost-outer reduction strategy for Smaran attempts to reduce the instance of the RHS but not its descendants before backtracking.

The DS-list reduction strategy is based on the DS-list in which a pointer to the *current* node is always maintained. In normalization, this pointer is moving from the current node to the next, trying to match the term represented by the node. Upon a successful match, LRR updates the list by adding pointers to any new resultant subterms with defined symbols as the roots, and by deleting pointers to any obsolete nodes representing the terms that were erased by the normalization. We will discuss it in more detail in Section 4.

The ALU-list reduction strategy controls the reduction based on results from a preprocessor for rules which unifies every subterm in every RHS with every LHS. In the ALU-list, a current pointer is maintained. During normalization, the pointer moves from the current node to the next, trying to match the term represented by the node with some, not all, LHSs according to the unification results. Upon a successful match, LRR updates the ALU-list by adding pointers to new resultant subterms according to the unifications, and deleting pointers to any stale nodes. Details will be discussed in Section 4.

The combination of the DS-list and the ALU-list reduction strategy gives the ALU-list strategy higher priority than DS-list strategy. During reduction, a subterm in the instance of RHS is checked by the ALU-list strategy before the DS-list strategy. Terms deleted from the ALU-list are checked by the DS-list strategy. Only when the ALU-list is empty, the DS-list strategy takes over the reduction.

4 Optimizations

In order to achieve efficiency in LRR, we have implemented both high-level and low-level optimizations. We now discuss some of the key optimizations in LRR. The low-level optimizations consist of integer encoding, elimination of dependency lists, memory allocation and hashing. The high-level optimizations include implicit evaluation, discrimination trees, don't-reduce signatures/terms, the DS-list and the ALU-list.

4.1 Integer Encoding

All strings, variables, defined symbols, and classes are encoded by numbers to improve efficiency. LRR encodes all strings appearing in the input files by numbers, which eliminates many string operations that are replaced by integer operations. In Smaran the access to the substitution for a variable is needed both for the creation of the instance of the RHS upon a successful match and for consistency checking since the same variable appearing in different places must be instantiated to the same term or class. To achieve efficient access, LRR uses the integer encoding. The integer of a variable is calculated once and is stored

along with the variable. Accessing the value of the variable is now done in constant time by direct indexing, using the integer encoding of the variable as the index [Verma 93]. Every class E is given a number for efficiency. This is useful in finding the unreduced signature of a class.

4.2 Elimination of Dependency Lists

In the latest version of LRR, we have completely eliminated dependency lists. These lists kept track of all the signatures that depended on a class. This information was needed to update signatures when a class is merged with another class. By changing the signature data structure to contain references to classes instead of class numbers, we avoid this expensive book-keeping and updating.

However, elimination of dependency lists causes a problem with AC operators, because this information is useful for the flattening process. Hence, to speed up the flattening process and avoid unnecessary traversals of signatures, we keep a reference counter to track the number of times a signature appears in the data structure. The counter for every signature is initially one and subsequent searches for the signature and class unions cause it to increase. If the counter is more than one and the signature is reduced, only then a traversal is required otherwise no traversal is needed for flattening. However, a traversal is still needed to identify duplicates if a subterm is reduced to an already existing subterm. This is detected by a successful search of the result when we are constructing its signature.

4.3 Implicit Evaluation

To improve efficiency, LRR predefines regular arithmetic operators, comparison operators and logical operators. Such operations are evaluated automatically. Our experience has shown that implementing these operations using rules is grossly inefficient and increases the number of signatures dramatically [Verma 93]. One important optimization is that LRR immediately evaluates the signatures in Smaran and terms in TGR with mathematical or relational operators as roots whose parameters are available directly. LRR stores only results and discards those signatures or terms. The number of the signatures and terms are reduced further and the normalization gets faster.

Another important optimization is **bottom-up strategy**. LRR uses bottom-up algorithms to compute predefined operations, which improves the efficiency of built-in datatypes. Take the Fibonacci calculator as an example which requires relatively more mathematical operations than reductions. The speed-up over the top-down algorithms ranges from 5 to 10 according to the given term. We use a queue to store pointers to the built-in operators which are stored in a binary tree. When LRR builds the instance of the RHS from bottom up, these operators are added into the queue. After the instance is constructed, LRR evaluates the queue.

The procedure for integrating built-in operations with reductions works fairly efficiently even though it is quite simple. It basically checks the queue for evaluations only when the reduction strategy in effect cannot find any more matches.

4.4 Discrimination Tree

LRR introduces discrimination trees for matching, which show substantial improvement in time for large sets of rules and/or terms that requires more than $50,000$ reductions. A variable list for tracking substitutions for variables to handle consistency requirements is also added. We represent the discrimination tree in a novel way; LRR does a breadth-first scan of the set of rules and links every level of the tree to remove costly recursive calls. Recall the integer encoding of strings, and variables, we use the integer encodings of the variables to index into the variable list. For example, for the input term $sieve(from(2, 2000), 500)$, which constructs a list of 2000 numbers starting from 2 and then extracts up to 500 primes from it, the reduction in time is almost 25% when discrimination trees are used.

Hence, for each rule in LRR, its LHS is stored top-down and in a discrimination tree to speed up matching and its RHS is stored bottom-up to speed up building the instance of RHS. These representations facilitate iterative construction, as opposed to recursive procedures, of right-hand instances and top-down matching of left-hand sides.

4.5 Don't-reduce Signatures/Terms

This is an optimization to cut down on unproductive traversals of the intermediate term during normalization. Don't-reduce signatures are defined as follows: i) All constructor constants are don't-reduce signatures; ii) $f(t_1, ..., t_n)$ is a don't-reduce signature if f is a constructor and the unreduced signatures of classes $\#[t_1]$ ... $\#[t_n]$ are don't reduced signatures. Don't-reduce signatures represent a subclass of normal forms. Hence no rules can match them, which means that LRR does not need to find any matches below the don't-reduce signatures. Any reduction procedure can skip the examination of large portions of the signature graph as useless for finding new matches. It is obviously unnecessary for LRR to traverse below a class having a normal form to find any matches. However, we must attempt to match all rules against the unreduced signature of a class to detect normal forms, which is more difficult than the detection of a don't-reduce signature because it can be done bottom-up without matching (recall that we build instance of RHS in a bottom-up manner). Furthermore built-in operations make detection of normal forms more complicated since an unreduced signature depending on a class whose unreduced signature is an unevaluated mathematical signature does not match any rules and may or may not match after the mathematical signature is evaluated. The idea of don't-reduce signatures also applies to TGR and Tree by focusing on terms not signatures.

4.6 Memory Allocation and Hashing

Because of the potentially large number of terms in TGR and signatures and classes in Smaran LRR uses free lists for various data structures including signatures, classes, the implicit evaluation queue to recycle space. Also we have our own routines for allocating and deallocating memory.

We also use new efficiently computable algorithms to hash strings and signatures. Better distribution of values and fewer collisions are obtained by the new algorithms.

4.7 The DS-list

In the latest version of LRR we have implemented the DS-list in both Smaran and TGR. The DS-list is a circular doubly linked list that contains pointers to terms or signatures that have defined symbols as roots. A pointer to the *current* node in the list is always maintained [Verma 04].

Reduction with the DS-list is relatively straightforward. As the given term t_0 is parsed, the DS-list is initialized to have pointers to t_0's subterms that have defined symbols at the top. During normalization, the current pointer moves from the current node to the next, attempting to match the term represented by the node. Upon a successful match, the list is updated. When building the instance of the RHS, pointers to new subterms with defined symbols as roots in the instance are inserted into the DS-list. Also pointers to stale terms that were deleted by the normalization are removed. For example, consider that the expression $f(g(a), b)$ reduces in one step to a, where f, g, b are defined symbols and a is a constructor symbol. Then, pointers to the subterms $f(...)$, $g(a)$ and b should all be removed from the DS-list, since these defined subterms have been erased. After update procedure finishes, normalization continues traveling around the DS-list, attempting matches. Reduction completes when the DS-list is empty or when a complete traversal around the list ends without any matches. Since the list can grow or shrink when a match is found, the *efficient* detection of a complete traversal around the list without any matches requires tracking whether any insertions were made into the DS-list or not.

Integration of DS-list into Smaran proceeds as follows. Each node in the DS-list contains a pointer to a class whose unreduced signature is labeled by a defined symbol in the current term being reduced.

As the given term is parsed, the existing code calls the function to insert the signatures into classes. This initializes the DS-list to contain pointers to classes of subexpressions with defined symbols at the root occurring in the given term.

Since terms are represented by classes, it is possible to simply track the creation, modification, and union of classes to determine the operations to be performed on the DS-list. Monitoring the classes for three simple conditions is the only measure necessary to maintain the list. First, if the unreduced signature of a newly created class is a defined signature, then the class should be inserted into the DS-list. Second, when inserting a new unreduced signature which is a constructor into a pre-existing class and that class number is marked *current* in

the DS-list, this class is deleted from the list. Third, when merging two classes, if the unreduced signature of the resultant class is labeled by a constructor, then the pointer to the class whose unreduced signature was reduced is removed from the DS-list. Deletions during (cascading) unions may occur anywhere within the DS-list and therefore could require a costly linear search through the list. To avoid repeated searching it is best to batch the deletions. Finally, classes containing defined unreduced signatures representing defined subterms that are erased during a reduction could also be deleted from the DS-list, but this can be even more expensive to determine than deletions caused by cascading unions. For this reason, we chose not to implement this last deletion condition. Since the only time a class is added to the DS-list is when it is created, and since classes are shared, the DS-list naturally contains only one reference to each class and so terms are not repeated.

Integrating into TGR is similar to the integration into Smaran. The DS-list is initialized to have pointers to the defined subterms in the given term. The DS-list is implemented on top of the DAG data structure of the intermediate result itself by updating the links to subterms. During reduction, the current pointer goes from current node to the next, trying to match the term represented by the node. The list is also updated by adding pointers to new defined subterms in the instance of RHS. DS-list deletion for TGR is subtle. At first glance it seems that it can be implemented efficiently by storing the pointers to the defined subterms of the current term (which may be a subterm of the current term for normalization) being matched. If the match is successful, then the stored list of pointers can be used to delete the stale subterms from the DS-list. If the match is unsuccessful, then we just reclaim the space for this temporary list of references. However, note that if a defined subterm is below a variable, i.e., in the substitution part, in the LHS it may never be traversed. Moreover, if this variable also appears in the corresponding RHS, e.g., if the rule is $f(x) \to g(x)$, then, for efficiency, the defined subterm should remain in the DS-list whereas if the variable is erased, e.g., if the rule is $f(x) \to a$, then the defined subterm should be deleted from the DS-list.

Because of these subtleties, a comprehensive DS-list deletion procedure for TGR is not yet implemented. This means that some time could be wasted in reducing the stale subterms. At present the deletion happens only when the reference to be deleted is pointed to by current. In future, the required information about variable erasure could be made part of the preprocessor for rules that we describe in the next section.

Optimizations for the DS-list can be made on the basic DS-list structure presented above. The first optimization is to delete (or to avoid insertion of) normal forms with defined symbols at the root. However, this requires matching the rules at least once against all the defined subterms of the expression. An easier to implement version of this optimization is to do it only for defined expressions that contain only constructors (or variables) at non-root occurrences. These can be computed easily by combining the don't-reduce optimization mentioned above with one round of matching at the top.

A second optimization involves moving both forward and backward through the DS-list when searching for matches by using two pointers into the list instead of moving in one direction. Empirically, it was found that this generally reduces the time spent in traversing the DS-list looking for a match when the DS-list size is beyond a certain threshold.

A third optimization for Smaran is to modify the behavior of a class union on the DS-list. Deleting every class which loses its unreduced signature in the union would require a costly linear search for every class union in which this occurs. Even if deletions are batched, one complete scan through the list may be required in the worst case. It is cheaper to simply do nothing for the non-current classes to be deleted instead of searching for them. The classes could be deleted while LRR traverses the list searching for matches (This "smarter" deletion is left for the future.).

4.8 The ALU-list

The latest optimization in LRR is the ALU-list. The ALU-list is a preprocessor for rules that tries to take advantage of the fact that a match of a rule inside the template part of a new RHS instance implies that an LHS of a rule unifies with a non-variable subterm of an RHS of a possibly different rule. These unifications can be determined and stored for static rule sets and for dynamic rule sets when a new rule is created. We have implemented the ALU-list in both Smaran and TGR. The ALU-list is a singly linked list that contains pointers to terms according to the unification results. LRR always maintains a pointer to the *current* node in the ALU-list.

An extension of almost linear unification (ALU) is to extend the regular ALU algorithm which uses a DAG to store terms and requires variables to be shared (Please see [Baader 99] for details) by allowing unifications between terms with predefined operators at the top and their possible results. To illustrate this, consider that the expression $>(x, 1)$ in the Fibonacci calculator. Term $true$ or $false$ is the result. Hence we consider that $>(x, 1)$ unifies with $true$ and $false$.

How does ALU help in normalization? We find that if a subterm $t = r|_p$ from a RHS r can unify with a LHS l, there is a great chance to find a match between the instance of t, denoted as $\sigma(t)$, and l when the $\sigma(t)$ is built by r. Hence before reduction starts, we add a preprocessor for rules which unifies every subterm in every RHS with every LHS using ALU and stores the successful unification results. During reduction, we introduce the ALU-list and let it help to find a match based on the successful unifications. To find a match in a step of reduction, LRR starts from *current* node of the list instead of scanning all subexpressions of the term to be normalized and all rules. To illustrate this, consider the unification result and two steps of normalization in Figure 1. The preprocessor has the information that a subterm $x = rhs_j|_p$ unifies with lhs_k. In step $t_{i-1} \rightarrow_{(i,j)} t_i$, LRR finds a match between a subterm u of t_{i-1} and lhs_j. Then v, the instance of rhs_j replaces the subterm u. We get t_i. Term v obviously shares the same overall structure as rhs_j and x unifies with lhs_k. Hence there is a great chance that term $w = v|_p$, the instance of term x matches lhs_k in the

next step. In the $i+1^{th}$ step, reduction directly tries the term w and lhs_k. If a match is found, term w is replaced by the instance of rhs_k.

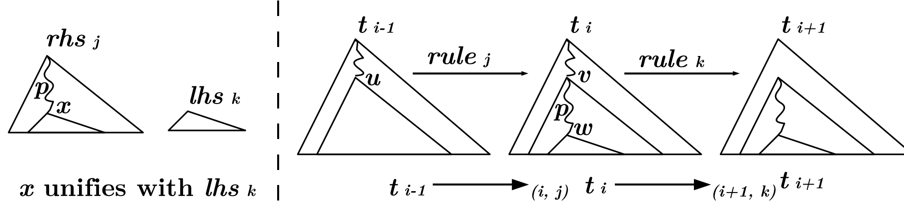


Fig. 1. Unification results can help in normalization

We give a concrete example in the Appendix to illustrate the operation of the ALU-list.

Actually, significant parts of all the intermediate results, $t_1, ..., t_i, ..., t_{n-1}$ and the normal form t_n are constructed from the RHSs and much of the overall structure of terms can be safely predicted from the RHSs (the exceptions are the variable substitutions and unexplored parts of the intermediate terms). However, not all the successful unifications guarantee successful matches. The ALU-list alone cannot control the reduction and under some cases, LRR must revert to traversing the term.

Reduction with ALU-list needs the successful unifications collected by the preprocessor. We use a pair (C, P) to denote the unification results. In Figure 1, lhs_k unifies with subexpression x from rhs_j. We define that $rule_k$ is a *candidate* and here $C = k$. We define the position a *point* and here $P = p$. We store the position not the term x because reduction needs to find w by following P from v. For every RHS, LRR uses a singly linked list to store the pairs. The first step of normalization is carried out by traversing the term since at this point there is no information from the previous step. When building t_1, the ALU-list is initialized to contain pointers to 3-tuple (i, c, s), where i indicates the ith step of normalization, c indicates a candidate, in which $c = C$, and s represents the term that is possible to match lhs_c, such as $w = v|_P$ in Figure 1. During normalization, the current node (i', c', s') pops out and the current pointer moves to the next node, attempting to match $lhs_{c'}$ with the term represented by s'. Upon a successful match, the list is updated. New nodes pointing to the instances of the subterms that unify with LHSs are added. Pointers to stale nodes are deleted, which will be discussed in the optimization section. After update procedure finishes, normalization continues traveling around the ALU-list. Since not every unification result leads to a successful match, when the ALU-list is empty, LRR must resort to term traversal to find the next match. When the ALU-list is not empty, it controls the normalization procedure. The ALU-list cannot either start the reduction or end the reduction. However it helps reduction in between.

4.9 Optimizations for the ALU-list

In order to improve the efficiency of integration of the preprocessor and the normalization procedure, we implemented the following optimizations.

Elimination of path lists saves the traversal time when LRR locates the term s in the instance of the RHS to store it into the 3-tuple which will be added into the ALU-list. The preprocessor used to have a list for keeping the path info of the point P. During reduction, LRR followed the path to find s. Much time is wasted on traversal. The current preprocessor flags the term t at the point P in the RHS. Building the instance needs two pointers: one traverses the RHS and the other traverses the instance simultaneously. When the former one visits t which was flagged by the preprocessor, the latter pointer visits the instance $s = \sigma(t)$ and LRR stores s.

Mutually exclusive detection cuts unnecessary addition into the ALU-list caused by the extension of the almost linear-time unification algorithm as mentioned above. The preprocessor considers every possible result of a built-in function as a candidate. Every pair containing a possible result will be added in to the ALU-list by the preprocessor. However, only one tuple will succeed in matching. Therefore, LRR calls mutually exclusive detection after evaluating the instance of the predefined functions to push the "right" tuple into the list.

Candidate elimination contains three ways to remove tuples from the ALU-list. *Same point elimination* cuts unnecessary matching attempts. Tuples that contains same i and same s apply at the same point in the same instance. Once we match the first tuple among these tuples, the remaining tuples are deleted since the intermediate term probably will change in the next normalization step. *Descendants elimination* also cuts unnecessary matching attempts. If the parent matches, LRR will not match its children since the intermediate term probably will change. *Changed signature check* cuts unnecessary matching attempts when the ALU-list works with Smaran. LRR checks if the unreduced signature of the class in the current tuple has changed since the tuple was pushed into the ALU-list. If yes, LRR pops the tuple directly without trying to match.

The V-list helps the ALU-list reduction strategy to scan the unexplored parts of the intermediate terms caused by the variable substitutions. It is a singly-linked list with current pointer. When building t_1, the V-list is initialized to contain pointers to the instance of the variable that occurs in the RHS and is instantiated. During reduction, the list is updated. Pointers to the instances of variables are added into the list. Only when the ALU-list is empty, the current pointer of the V-list moves to the next node, looking for a match for the term represented by the node. Since the V-list contains the pointer to the term without the candidate information, LRR traverses the term to find the next match for normalization. V-list controls the reduction when the ALU-list strategy cannot advance and routes the reduction to the unexplored parts instead of tracking back all the way to the top.

5 Experimental Results

All of the optimizations described above are mature except for DS-list deletion and the preprocessor for rules, which still have some room for improvement. A Linux version of LRRv3.0 and some examples can be downloaded from the first author's web page and also `http:/www.cs.uh.edu/~evangui`. We compare LRR against Maude 2.6 32-bit version, which can be found at `http://maude.cs.uiuc.edu/download`, and the ELAN interpreter 3.6g, which can be found at `http://webloria.loria.fr/equipes/protheo/SOFTWARES/ELAN/manual/index-manual.html`.

Performance Results. We present the experimental results on nine benchmarks (rules can be found the Appendix) to illustrate the level of efficiency. LRR is implemented in C and runs on Linux. Normalization times are on a 2.67GHz Intel i5 560M Ubuntu 10.10 Linux kernel 2.6.35-22 system with 8GB of memory using gcc compiler (v. 4.4.5) with optimization level 3. We are aware of the difficulties of comparing different software systems. Each benchmark for three systems uses exactly the same algorithm. Rules for the benchmarks are of course semantically identical. Syntactic differences are due to differences in the rule specifications for the three interpreters. Table 1 shows the average results of 10 executions in seconds for nine benchmarks, which can be found at the URL given above.

Table 1. Experimental Results on Normalization Time

Benchmark	ELAN	Maude		LRR			
		w/o memo	w/ memo	Smaran	Smaran+ALU	TGR	TGR+ALU
binsort(1500)	164.2228	0.6936	463.6586	1.2165	1.7305	0.8669	1.2885
bintree(380)	0.1152	0.0044	0.0936	0.0092	0.0092	0.0072	0.0060
dfa(1363)	0.0016	0.0000	0.0008	0.0312	0.0312	0.0240	0.0260
fib(20)	1.4416	0.0272	0.0000	0.0000	0.0000	3.8658	3.7494
merge(20000)	17.8455	0.0404	70.2658	0.0296	0.0380	0.0152	0.0196
qsort(1800)	66.6180	1.1872	30.7426	7.6769	7.0912	1.7753	2.3857
rev(19900)	66.6304	0.0380	129.6359	0.0308	0.0372	0.0156	0.0200
rfrom(19996)	1.7005	0.0408	44.1588	0.0236	0.0268	0.0092	0.0148
sieve(10000)	169.6300	0.4900	29.6235	1.1249	1.3365	0.4108	0.5624

Tables 2 and 3 give more detailed information about the various strategies available in Smaran and TGR respectively. In these tables, LMOM represents the approximation of leftmost-outermost mentioned above.

Table 2. Experimental Results based on Smaran Strategy

Benchmark	Smaran				
	Smaran	DS List	LMOM	ALU List	DS List + ALU List
binsort(1500)	1.2165	1.6129	1.4505	1.7305	2.1873
bintree(380)	0.0092	0.0032	0.0052	0.0092	0.0056
dfa(1363)	0.0312	0.0020	0.0004	0.0312	0.0020
fib(20)	0.0000	0.0000	0.0000	0.0000	0.0000
merge(20000)	0.0296	0.0372	4.3471	0.0380	0.0440
qsort(1800)	7.6769	6.2944	7.4089	7.0912	7.3593
rev(19900)	0.0308	0.0360	0.0332	0.0372	0.0432
rfrom(19996)	0.0236	0.0208	0.0244	0.0268	0.0304
sieve(10000)	1.1249	1.3781	1.3061	1.3365	1.5781
Sum.	10.1427	9.3846	14.5758	10.3006	11.2499

Table 3. Experimental Results based on TGR Strategy

Benchmark	TGR				
	TGR	DS List	LMOM	ALU List	DS List + ALU List
Binsort(1500)	0.8669	1.0957	1.0593	1.2885	1.2581
bintree(380)	0.0072	0.0064	0.0044	0.0060	0.0060
dfa(1363)	0.0240	0.0276	0.0000	0.0260	0.0000
fib(20)	3.8658	3.9447	3.9999	3.7494	3.8110
merge(20000)	0.0152	0.0116	2.4930	0.0196	0.0180
qsort(1800)	1.7753	2.3257	2.3641	2.3857	2.2389
rev(19900)	0.0156	0.0144	0.0160	0.0200	0.0188
rfrom(19996)	0.0092	0.0076	0.0108	0.0148	0.0140
sieve(10000)	0.4108	0.8897	0.5196	0.5624	0.5348
Sum.	6.9900	8.3234	10.4671	8.0724	7.8996

From Table 1, even though we find that Maude without memo is the fastest option in most benchmarks, Smaran and/or TGR are close. It is interesting to see that Smaran is not far behind even in examples that do not use history, despite saving the entire history of rule applications. ELAN interpreter runs slow in most cases. We are aware that the ELAN project focuses more on the compiler than the interpreter. Maude with memo runs faster for fib(20) and dfa but is much slower for the other benchmarks tested. The preprocessor does not completely beat TGR or Smaran. Apparently there is some inefficiency in the implementation of the preprocessor. We think we can improve it in the following ways. First, we plan to write a new function for matching since we have a great accuracy in prediction. The new function should explore the unification results deeper. The other, when the preprocessor cannot initiate a match, LRR should find the next match in a more efficient way. Although, the preprocessor of rules runs slower than original methods in most examples, but it cuts the unnecessary matching attempts significantly. Although it does not yet control the normalization independently, the percentage of successful matches is relatively high.

From Tables 2 and 3 it is clear that DS-list strategy is the winner for Smaran but not for TGR, which seems to be a direct consequence of the incomplete DS-list deletion algorithm that hurts TGR but not Smaran since Smaran only considers unreduced signatures for potential matches. So the only consequence of an incomplete DS-list deletion algorithm for Smaran is a DS-list that is longer than necessary and the time lost in traversing a longer than optimal list, which appears to be small from the timings.

Other Related Work. We did an extensive search for rule-based programming interpreters using the papers [Hermann 91,Vittek 96] and the Rewriting Page on the web, but we have been unable to find any other interpreters that come close in terms of the range of optimizations in LRR. Apart from Maude, in [Vittek 96] a compiler for rules is described, but there is no comparable effort on speeding up normalization. The only other interpreter that we could find is CRSX [Klop 93], which does not include built-ins and could only handle a string of length 819 in the dfa example.

6 Conclusions and Future Work

In this paper we have presented LRR an efficient interpreter for rule-based programming and term rewriting, and described the key optimizations that make it efficient. LRR is unique in its efficient tabling component Smaran and a relatively efficient non-tabling component as well. As far as future work is concerned, the ALU-list can be made more efficient, the DS-list deletion algorithm can be completed and support for AC operators can be provided in the DS-list and ALU-list structures. Furthermore, the backtracking procedure can take advantage of unifications between RHSs and subterms of the LHSs, the converse of the ALU-list procedure.

Acknowledgments. We want to thank Jieh Hsiang, K.B. Ramesh, S. Kolli for initial work on Smaran, S. Senanayake and H. Shi for work on LRR, J. Thigpen for work on DS List, and Z. Liang for the CTL model checking program.

References

[Baader 99] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press, (1999)

[Bachmair 99] Bachmair, L., Ramakrishnan, C., Ramakrishnan, I. and Tiwari, A.: Normalization via Rewrite Closures. Lecture Notes in Computer Science pp. 190–204. (1999)

[Borovansky 96] Borovansky, P., Kirchner, H., Moreau, P.E., Vittek, M.:ELAN: A logical framework based on computational systems. In: Proceedings of the first international workshop on rewriting logic. Asilomar (1996).

[Borovansky 02] Borovansky, P., and others: ELAN User Manual. (2002)

[Brand 02] van den Brand, M., Heering, J., Klint, P., Olivier, P.: Compiling Rewrite Systems: The ASF+SDF Compiler. ACM Transactions on Programming Languages and System pp. 334 – 368. (2002)

[Chew 80] Chew, P.: An Improved Algorithm for Computing with Equations. focs, pp. 108 – 117. (1980)

[Clavel 00] Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude. Electronic Notes in Theoretical Computer Science. (2000)

[Clavel 03] Clavel, M., Durán, F., Eker, S. and others: The Maude 2.0 System. Rewriting Techniques and Applications (RTA 2003) Lecture Notes in Computer Science pp. 76 – 87. (2003)

[Dershowitz 01] N. Dershowitz and D. Plaisted. Rewriting. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier Science, 2001.

[Hermann 91] Hermann, M., Kirchner, C., Kirchner, H.: Implementations of term rewriting systems. The Computer Journal, 34(1), pp. 20–33. (1991)

[Klop 93] Klop, J.W., Oostrom, V.V., and Raamsdonk, F.V.: Combinatory Reduction Systems: Introduction and Survey, Theoretical Comp. Sci. 121, pp. 271-308 (1993).

[Radcliffe 10] Radcliffe, N., Verma, R.: Uniqueness of Normal Forms is Decidable for Shallow Term Rewrite Systems. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 284–295. (2010)

[Shi 00] Shi, H.: Integrating associative and commutative matching in the LR^2 Laboratory for fast, efficient and practical rewriting techniques. University of Houston. (2000)

[Verma 89] Verma, R.: Equations, Nonoblivious Normalization, and Term Matching Problems. State University of New York at Stony Brook. (1989)

[Verma 93] Verma, R.: Smaran: A congruence-closure based system for equational computations. In: Proceedings of the 5th International Conference on Rewriting Techniques and Applications, pp. 457–461. (1993)

[Verma 00] Verma, R.: Static Analysis Techniques for Equational Logic Programming. In: Proceedings of the 1st ACM SIGPLAN Workshop on Rule-based Programming. (2000)

[Verma 95] Rakesh M. Verma. A theory of using history for equational systems with applications. *Journal of the ACM*, 42(5):984–1020, 1995. Also in the 32nd IEEE FOCS Symposium, 1991.

[Verma and Guo 11] Verma, R. and Guo, W.: Does Unification Help In Normalization? University of Houston Computer Science Department Technical Report, UH-CS-11-05, June 2011.

[Verma and Guo 11b] Verma, R., Guo, W.: Does Unification Help in Normalization. The International Workshop on Unification (2011)

[Verma 99] Verma, R., Senanayake, S.:LR^2 : A Laboratory for Rapid Term Graph Rewriting. In: Proceedings of the 10th International Conference on Rewriting Techniques and Applications, pp. 252–255. (1999)

[Verma 04] Verma, R., Thigpen, J.: DS-forest: A Data Structure for Fast Normalization and Efficiently Implementing Strategies. In: Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming. pp. 45 – 49. (2004)

[Vittek 96] Vittek, M. :A Compiler for Nondeterministic Term Rewriting Systems. In: Proceedings 7th Conference on Rewriting Techniques and Applications. pp. 154-168 New Brunswick, New Jersey, USA (1996).

[Yu 08] Wenshan Yu and Rakesh M. Verma. Visualization of rule-based programming. In *ACM SAC, Graphics and Visualization Track*, 2008.

Appendix

6.1 A Concrete Example

To illustrate the details of the ALU-list reduction strategy, we use the Fibonacci calculator as an example. We label the rules as below.

$$fib(x) \Rightarrow f(>(x,1),x) \tag{1}$$

$$f(true,x) \Rightarrow +(fib(-(x,1)), fib(-(x,2))) \tag{2}$$

$$f(false,x) \Rightarrow 1; \tag{3}$$

The preprocessor tries to unify every subterm of every RHS with every LHS. Remember we extend the ALU algorithm, $f(>(x,1),x)$ in rhs_1 unifies with lhs_2:$f(true,x)$, lhs_3:$f(false,x)$. Since we eliminate the path info, we flag the term,$f(>(x,1),x)$ for which we build a list of candidates lhs_2 and lhs_3. In $rule_2$, $fib(-(x,1))$ has a candidate lhs_1 and $fib(-(x,2))$ has a candidate lhs_1. Before elimination of the path info, we build a list of two pairs $(2,\lambda)$, $(3,\lambda)$ for $rule_1$ and a list of two pairs $(1,(1))$, $(1,(2))$ for $rule_2$.

The normalization process using TGR under a depth-first left-most order and the process using Smaran discussed above is below:

$$\begin{aligned} fib(2) &\rightarrow_{(1,1)} f(true,2) \\ &\rightarrow_{(2,2)} +(fib(1), fib(0)) \\ &\rightarrow_{(3,1)} +(f(false,1), fib(0)) \\ &\rightarrow_{(4,3)} +(1, fib(0)) \\ &\rightarrow_{(5,1)} +(1, f(false,0)) \\ &\rightarrow_{(6,3)} 2 \end{aligned} \tag{4}$$

In the 1st step, since the ALU-list has not been initialized due to the lack of information, LRR calls Smaran or TGR to find a match. LRR picks $rule_1$ and $t_1 = f(true, 2)$. When building the instance $f(true, 2)$ in a bottom-up manner with the unification info, LRR checks every subterm in the instance. If it is flagged as a *point*, LRR updates the ALU-list. Without Mutual Exclusive Detection, LRR adds two 3-tuples $(1, 2, s)$ and $(1, 3, s)$, in which s indicates the term $f(true, 2)$. With the detection, only one 3-tuple $(1, 2, s)$ is added.

In the 2nd step, LRR pops the 3-tuple $(1, 2, s)$ from the list and tries to match lhs_2 with $f(true, 2)$ while the the original LRR attempts to match $f(true, 2)$ with all 3 rules. After the successful match, LRR tries to clean the list. However, the ALU-list is empty now. LRR builds the instance and adds $(2, 1, s')$ where s' indicates the term $fib(1)$ and $(2, 1, s'')$ where s'' indicates the term $fib(0)$ into the list.

In the 3rd step, under a depth-first left-most order, LRR pops $(2, 1, s')$ first and match $fib(1)$ with lhs_1 successfully while the original LRR traverses from the root of $+(fib(1), fib(0))$ searching for a match. LRR tries to eliminate candidates but finds nothing to remove. $t_3 = +(f(false, 1), fib(0))$. After 6 steps, LRR stops at 2 which is the normal form.

In the ALU-list strategy based on Smaran, s in the 3-tuple (i, c, s) stores the number of the class representing the term and the pointer to the unreduced signature in step i. In later steps, LRR pops this tuple and calls changed signature check to check if the current unreduced signature stays as same as the one in step i. If yes, LRR matches the unreduced signature of the class with lhs_c. If no, LRR abandons the tuple and pops the next one if there is. In the ALU-list strategy based on TGR s is the pointer to the term.

6.2 Benchmarks

We use nine benchmarks for ELAN, Maude, and LRR. The rules are listed below. All benchmarks for 3 interpreters are also available at `http:/www.cs.uh.edu/~evangui`.

1. binsort. Binary insertion sort. This program sorts a list by inserting values into a binary search tree.

$$ins(x, nil) \Rightarrow node(nil, x, nil) \tag{5}$$

$$ins(x, node(l, v, r)) \Rightarrow instest(x, >(x, v), <(x, v), l, v, r)) \tag{6}$$

$$instest(x, false, true, l, v, r) \Rightarrow node(ins(x, l), v, r) \tag{7}$$

$$instest(x, true, false, l, v, r) \Rightarrow node(l, v, ins(x, r)) \tag{8}$$

$$instest(x, false, false, l, v, r) \Rightarrow node(l, v, r) \tag{9}$$

$$cat(:(x, y), z) \Rightarrow :(x, cat(y, z)) \tag{10}$$

$$cat(nil, z) \Rightarrow z \tag{11}$$

$$binsort(:(x, y)) \Rightarrow bs(ins(x, nil), y) \tag{12}$$

$$bs(n, :(x, y)) \Rightarrow bs(ins(x, n), y) \tag{13}$$

$$bs(n, nil) \Rightarrow makelist(n) \tag{14}$$

$$makelist(node(l,v,r)) \Rightarrow cat(makelist(l), :(v, makelist(r))) \quad (15)$$
$$makelist(nil) \Rightarrow nil \quad (16)$$

2. bintree. This program inserts a value into a binary search tree.

$$ins(x,nil) \Rightarrow node(nil,x,nil) \quad (17)$$
$$ins(x,node(l,v,r)) \Rightarrow instest(x, >(x,v), <(x,v), l, v, r) \quad (18)$$
$$instest(x,false,true,l,v,r) \Rightarrow node(ins(x,l),v,r) \quad (19)$$
$$instest(x,true,false,l,v,r) \Rightarrow node(l,v,ins(x,r)) \quad (20)$$
$$instest(x,false,false,l,v,r) \Rightarrow node(l,v,r) \quad (21)$$

3. dfa. This program simulates a deterministic finite automaton.

$$a(q0) \Rightarrow q1 \quad (22)$$
$$b(q0) \Rightarrow q0 \quad (23)$$
$$a(q1) \Rightarrow q0 \quad (24)$$
$$b(q1) \Rightarrow q1 \quad (25)$$

4. fib. This program calculates the nth Fibonacci numbers. Please refer to the concrete example

5. merge. This program merges two lists into one.

$$merge(nil,nil) \Rightarrow nil \quad (26)$$
$$merge(:(x,y),nil) \Rightarrow :(x,y) \quad (27)$$
$$merge(nil,:(x,y)) \Rightarrow :(x,y) \quad (28)$$
$$merge(:(x,y),:(u,v))) \Rightarrow :(x,:(u,merge(y,v))) \quad (29)$$

6. qsort. This program implements quicksort on a list of natural numbers.

$$cat(:(x,y),z) \Rightarrow :(x,cat(y,z)) \quad (30)$$
$$cat(nil,z) \Rightarrow z \quad (31)$$
$$sort(nil) \Rightarrow nil \quad (32)$$
$$sort(:(x,y)) \Rightarrow cat(sort(smaller(x,y)), cat(:(x,nil),sort(larger(x,y)))) \quad (33)$$
$$smaller(x,nil) \Rightarrow nil \quad (34)$$
$$smaller(x,:(y,z)) \Rightarrow f(<(x,y),x,y,z) \quad (35)$$
$$f(true,x,y,z) \Rightarrow smaller(x,z) \quad (36)$$
$$f(false,x,y,z) \Rightarrow :(y,smaller(x,z)) \quad (37)$$
$$larger(x,nil) \Rightarrow nil \quad (38)$$
$$larger(x,:(y,z)) \Rightarrow g(<(x,y),x,y,z) \quad (39)$$
$$g(true,x,y,z) \Rightarrow :(y,larger(x,z)) \quad (40)$$
$$g(false,x,y,z) \Rightarrow larger(x,z) \quad (41)$$

7. rev. This program reverses a list.

$$rev(x) \Rightarrow apprev(x, nil) \tag{42}$$
$$apprev(:(x, y), z) \Rightarrow apprev(y, :(x, z)) \tag{43}$$
$$apprev(nil, w) \Rightarrow w \tag{44}$$

8. rfrom. This program outputs a list of natural numbers in a reverse order.

$$rfrom(x, y) \Rightarrow rffrom(>(y, 0), x, y) \tag{45}$$
$$rffrom(true, x, y) \Rightarrow :(x, rfrom(-(x, 1), -(y, 1))) \tag{46}$$
$$rffrom(false, x, y) \Rightarrow nil \tag{47}$$

9. sieve. This program outputs a list of prime numbers from a list of natural numbers greater than 1.

$$fsieve(true, x, l, y) \Rightarrow :(x, sieve(filter(x, l), -(y, 1))) \tag{48}$$
$$fsieve(false, x, l, y) \Rightarrow nil \tag{49}$$
$$filter(n, :(x, l)) \Rightarrow ffilt(=(\%(x, n), 0), n, x, l) \tag{50}$$
$$filter(n, nil) \Rightarrow nil \tag{51}$$
$$ffilt(true, n, x, l) \Rightarrow filter(n, l) \tag{52}$$
$$ffilt(false, n, x, l) \Rightarrow :(x, filter(n, l)) \tag{53}$$
$$sieve(:(x, l), y) \Rightarrow fsieve(>(y, 0), x, l, y) \tag{54}$$
$$sieve(nil, y) \Rightarrow nil \tag{55}$$
$$sieve(x, 0) \Rightarrow nil \tag{56}$$