$\vec{LR}^2$: A Laboratory for Rapid Term Graph Rewriting

Rakesh Verma and Shalitha Senanayake
Computer Science Department
University of Houston
Houston, TX 77204
Ph: (713)743-3348
Fax: (713)743-3335
Email: rmverma@cs.uh.edu

Category: System Description

Abstract

In this system description, we present the current state of LR^2, a laboratory for developing and evaluating fast, efficient and practical rewriting techniques. We include a brief description of the main algorithm for storing history, selected optimizations, and some performance results and comparisons with related systems. We conclude with some promising directions for future research.

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 [3]. At the University of Houston we have been developing and evaluating the LR^2 system for fast rewriting for the last several years. There are two motivations for LR^2: we plan to use it for formal verification approaches that include rewriting, and as a testbed for innovating rewriting algorithms that are fast and efficient in practice. In this description we present the current state of LR^2, including a brief description of the main algorithm for storing history, selected optimizations, and some performance results and comparisons with related systems.

LR^2 consists of a term graph interpreter TGR, and a term graph rewriter that stores the history of its reductions, called Smaran,[1] based on the congruence closure approach. The input to LR^2 is a program representing a convergent rewrite system (a different version allows orthogonal systems) and an input term. Similar to algebraic specification languages like OBJ, ASF+SDF and ELAN a program is composed from modules. Each module defines its own signature and rewriting rules. A module can import other modules. Terms in LR^2 are written in prefix form. The language of LR^2 contains several built-in datatypes, viz., integers, floating-point arithmetic, booleans, characters, sets, multisets, and strings with associated operations. The set datatype supports the insertion, deletion, membership, union, etc., operations. The string datatype supports membership and indexing operations.

LR^2 also includes a variant detector that can determine if a new term is an alphabetic variant of an existing term, which is currently 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. LR^2 also allows a compact form for storing lists of arithmetic progressions as these occur frequently. Instead of storing the entire list, LR^2 stores the initial value, the final value and the difference. LR^2 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.

LR^2 provides a variety of options for controlling the amount of history that is stored by the system, if the user chooses the history option. The default option using Smaran is to save the results of each rewrite step in a compact data structure. The language of LR^2 allows annotating specific functions with the keyword "memo". This allows to save all the results of rewriting terms that have the specified function symbol at the root. The Delete (history) option in LR^2 allows to delete the entire history of rewrites performed so far except for the given term and its latest reduct after every i^{th} rewrite step, where i can be specified by the user. Another possibility allowed is to delete the entire history except the given term and the latest reduct as soon as the free memory available to the user drops to a user-specified percentage of the total available memory. Options can be combined in any way to suit the application. However, the delete option overrides the other options.

Operators can also be declared AC in LR^2. However, currently only left-linear rules with AC operators can be handled; the matching algorithm for nonlinear rules with AC operators is in the testing phase. The rest of this description is organized as follows. In Section 2, we give a description of the main algorithm for Smaran , in Section 3 we discuss selected optimizations, and in Section 4 we present some performance results and comparisons.

2 Smaran's Core Algorithm

The basic algorithm at the core of Smaran was proposed in [1] for orthogonal rules and later extended in [7, 9] to non-left-linear rewrite systems under fairly general conditions. The basic algorithm in turn is an extension of the well-known congruence closure algorithm (CCA) [2, 5, 6, 4]) for ground equations.

Recall that CCA divides the set of terms into numbered equivalence classes. Membership of a term in an equivalence class is decided by its *signature*. The signature of a term $f(t_1, \ldots, t_n)$ is the tuple $\langle f \ \#[t_1] \ \ldots \#[t_n] \rangle$, where $\#[t_i]$ is the number of the equivalence class containing the signature representing t_i. Equivalence class C *represents* term t if C contains a signature representing t. CCA operates by merging equivalence class representing terms whose equivalence follows from the given equations.

To extend CCA for normalization we make use of the concept of a distinguished signature in each equivalence class called the *unreduced signature* [1, 7]. Using this signature we construct a distinguished term. It has been shown in [1, 7, 9] that it is enough to examine this term to select useful rule instances, and that it is sufficient to check this term for irreducibility for rewrite systems satisfying fairly general conditions. If it exists and is irreducible, then the class containing this term has a normal form. Whenever a rule instance $A \rightarrow B$ can be applied to the distinguished term of a class, C, because of a match, the signature representing A is marked reduced and the class representing B (if any) is merged with C. If there is no class representing B, then a signature representing B is constructed, inserted into C, and marked the unreduced signature of C.

[1] A preliminary, slower version of Smaran with fewer options was demonstrated at RTA 93.

The algorithm starts by constructing the signature of the given term. This signature is then inserted into a class and marked the unreduced signature of the class. This class number is tracked throughout the process of normalization. Signatures of terms are constructed in the obvious bottom-up way. We illustrate this algorithm with a small example. Consider the convergent rewrite system, $\{1 : fib(x) \rightarrow f(x > 1, x), 2 : f(true, x) \rightarrow fib(x-1) + fib(x-2), 3 : f(false, x) \rightarrow 1\}$, that uses built-in arithmetic to define the fibonacci function and let the input term be $fib(2)$.

The initial set of classes is: $0 : \{2^*\}\ 1 : \{\langle fib\, 0\rangle^*\}$ (the symbol '*' indicates unreduced signatures). Now the match procedure is called to find a match between the unreduced signature of any class and the left-hand side of any rule. A match occurs between class 1 and rule 1. Now, the signature representing the corresponding instance of the right-hand side of rule 1 is constructed, inserted into class 1 and marked as its unreduced signature. Note that here we do not show the signatures, corresponding to the built-in datatypes, that can be evaluated directly. At this stage the classes are as follows:

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

During the next iteration class 1 matches rule 2. The right-hand side instance is $fib(1) + fib(0)$. The signature representing this is created and inserted into class 1 as its new unreduced signature. At the end of the second iteration the new/changed classes are as follows:

$$1 : \{\langle fib\, 0\rangle, \langle f\ 3\ 0\rangle, \langle +\ 4\ 6\rangle^*\}\ 4 : \{\langle fib\, 2\rangle^*\}\ 5 : \{0^*\}\ 6 : \{\langle fib\, 5\rangle^*\}$$

After two rewrite steps, using rules 1 and 3 respectively, the term $fib(1)$ represented by class 4 reduces to 1, which is represented by class 2. Hence classes 4 and 2 are merged, say into 2, and we have:

$$0 : \{2^*\}\, 1 : \{\langle fib\, 0\rangle, \langle f\ 3\ 0\rangle, \langle +\ 2\ 6\rangle^*\}\, 2 : \{\langle fib\, 2\rangle, \langle f\ 7\ 2\rangle, 1^*\}\, 3 : \{true^*\}\, 5 : \{0^*\}\, 6 : \{\langle fib\, 5\rangle^*\}\, 7 : \{false^*\}$$

Note that signatures containing the class number 4 have been updated to contain class number 2. After two more rewrite steps, the term $fib(0)$ represented by class 6 reduces to 1. Hence classes 6 and 2 are merged, say into 2, and we get:

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

Now, the unreduced signature of class 1 can be evaluated to yield the term 2, which is in class 0. Hence, classes 1 and 0 are merged, say into 1, and we get:

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

No more matches are found. Therefore, the algorithm checks for the existence of a normal form for the given term. Class 1 contains the unreduced signature *2*, which is irreducible. Thus the normal form of $fib(2)$ is 2. Note that if the normal form of $fib(fib(2))$ is needed, no more computations are needed since this term is also represented by the signature $\langle fib\, 0\rangle$ in class 1. It is simply extracted from class 1. The normal form of this term is also 2. On the other hand, an interpreter that does not store history would compute $fib(2)$ twice to normalize this term. This compact data structure helps exploit the advantages of storing history and can also speed up normalization.

3 Select Optimizations

In the design and implementation of LR^2 we have attempted to achieve efficiency through both high-level optimizations such as the normalization strategy reported earlier in [8] and also low-level optimizations such as encoding all strings by integers to replace string operations with less expensive integer operators [8]. For lack of space, we mention here only a few of the key optimizations in LR^2.

For the Smaran subsystem of LR^2, we have experimented with alternative representations and chosen the most efficient one. We have replaced the use of several large arrays in the previous version, by a more dynamic data structure that involves a single array, with interconnected lists for each class, and balanced search trees. A signature is stored only in one location and all other data structures address this location. We changed the components of a signature from class numbers to actual pointers to other signatures. A union operation is a constant-time operation implemented by setting a pointer from the root of one class to the root of the other. Of course, after several union operations, the chains of pointers that must be followed can become quite long. Hence we employ the technique of path compression on these chains, when obtaining the class number of a signature.

Bottom-up algorithms for built-in data types. The second optimization that is crucial to the efficiency of built-in datatypes is the use of bottom-up algorithms for computations involving the built-in datatypes. The result of this change can be seen for instance in the Fibonacci program, which requires relatively more arithmetic computations than reductions. Here the speed-up over the previous version ranges from 5 to 10 depending on the input term.

Matching and Construction of Signatures. The third major optimization is the use of discrimination trees for matching in LR^2, with a variable list for keeping track of the substitutions for variables to handle consistency checks. The discrimination tree representation is novel in that we do a breadth-first scan of the set of rules and each level of the tree is linked to eliminate expensive recursive calls. Since all strings are encoded as integers in LR^2, the integer encodings for the variables, which are saved in the record for the variables, are used for indexing into the variable list. The use of discrimination trees gives substantial reduction in time for large sets and/or terms requiring over 50000 reductions. For

Program	TGR (s)	No. of classes	Smaran (s)	Reductions	Computations
$Fib(800.0)$	–	1604	1.99	1602	3198
$huff(700)$	18.38	167803	22.74	327585	163675
$qsort(500)$	16.01	254004	33.77	628253	250501
$primes(800)$	21.46	326836	39.72	651314	656529

Table 1: Experimental Results

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 time is almost 25% when discrimination trees are used. We have also adopted a dual representation of rules in LR^2 in which the left-hand sides are stored top-down and right-hand sides are stored bottom-up. With this representation construction of the right-hand side instance for TGR or its signature for Smaran on a successful match is accomplished more efficiently in a bottom-up manner without any expensive recursions. Finally, we have implemented incremental matching algorithms in LR^2 for enhanced efficiency in theorem proving applications. In such applications, new rules may be created or existing rules deleted, hence we have designed the discrimination tree structure so that it can efficiently support deletion and insertion operations.

Dependency lists. The fourth optimization that is crucial to the Smaran subsystem in LR^2 is the handling of dependency lists. For the identification of signatures that must be changed when a class is unioned with another class, e.g., the signature $\langle + \ 4 \ 6 \rangle$ must be changed to $\langle + \ 2 \ 6 \rangle$ when class 4 is merged into 2 in the above example, we associated a dependency list with each class in Smaran. All signatures that must be modified in the event that this class changes are put on the class's dependency list. When a class C is merged into another class, signatures on C's dependency list are examined and modified. The previous version of Smaran handled dependency lists in a completely eager manner.

In the latest version we have modified the structure and implemented a lazier algorithm. One optimization is that we identify classes on which potentially large number of signatures can depend, e.g., the classes containing the boolean constants $true$ and $false$ in Section 2, and do not create dependency lists for these classes and ensure that these classes are not merged into any other class. More importantly, we take into account the sizes of the dependency lists to decide which class to union into the other. This turns out to be more efficient than weighted union with the sizes of the classes. We also keep track of the location of each signature in the dependency list, which eliminates subsequent searches.

Memory Allocation and Hashing. LR^2 recycles space using free lists for various data structures and it has its own memory allocation and deallocation routines. Finally, for the hashing of strings and signatures we use improved hashing algorithms that are efficiently computable. Practical studies have shown that we get better distribution of values with fewer collisions with the new algorithms. These involve the use of randomization.

4 Results and Comparisons

Because of space limitations, we present only a few experimental results to illustrate the level of efficiency achieved by the LR^2 system. LR^2 is implemented in C and runs on Sun/DEC workstations.

The following table summarizes results for four benchmarks: Fib - given in Section 2, $huff$ - a function that computes huffman codes for a file containing n distinct characters, $qsort$ - a quicksort program for a list of n numbers, and $primes$ - to compute the first n prime numbers. The input term for primes is $sieve(from(2, 6000), 800)$. Construction of the list $from(2, 6000)$ takes 1.42 seconds for Smaran and 0.42 seconds for TGR and *is* included in the results. Results for quicksort are for the worst case, when the input list is given in the reverse order. Note that history is not useful at all for $huff$, $primes$, and $qsort$.

To illustrate the options of LR^2, we ran it with the history and delete options but all functions except a dummy were specified "no memo" for $sieve(from(2, 20000), 2500)$ achieving: reductions 5259141, computations 5276867, number of classes 272643, and a time of 219.64 seconds (nearly 24000 reductions per second).

Comparison with other systems. We understand the difficulties of comparing software systems using experimental results, which can be sensitive to the choice of benchmarks, architectures, implementation language, etc. However, to illustrate the level of efficiency achieved by LR^2 we compared it with other interpreters.

In particular, we compared LR^2 with the ELAN interpreter on several benchmarks including those listed in Table 1 and found that LR^2 with history option is between 15 to 80 times faster for linear list reversal with list sizes ranging from 1000 to 4000, for $primes$ LR^2 (with history) is about 15 times faster for first 500 primes, and similar results were obtained

Example	Reveal	Reveal+Smaran	Reveal+TGR
exgroup	0.7	0.5	0.3
exgroupl	0.5	0.4	0.4

Table 2: Experimental Results (in seconds) of Comparison with Reveal

for other programs.[2] We also compared LR^2 with the Reveal (version 1.0) prover, which is written in C. Results for some completion examples distributed with Reveal are summarized in Table 2.

Since Reveal does not have built-in arithmetic we tried only a linear list reversal program and found that Reveal took total time of 6.46 seconds to reverse a list of 100 elements and 22.91 seconds to reverse a list of 150 elements as compared to LR^2's (with history) total timings of 1.2 seconds.

5 Discussion and Future Work

In this paper we have presented LR^2 along with some useful extensions and optimizations. We have developed some ideas on how to statically analyze rewrite systems and determine those functions for which history is likely to be useless and those for which history is likely to be useful [10]. We plan to incorporate these in future versions of LR^2. Future versions of LR^2 will include lazier handling of dependency lists (we have identified a class of signatures that need never be added to dependency lists [10]), more sophisticated reduction strategy [10], and more flexibility to the user in choosing strategies (e.g., leftmost innermost/outermost).

Acknowledgements. We thank K.B. Ramesh and S. Kolli for initial contributions to Smaran.

References

[1] P. Chew. An improved algorithm for computing with equations. In *Proc. IEEE Symp. on Foundations of Computer Science*, volume 21, pages 108–117, 1980.

[2] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

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

[4] D. Kozen. Complexity of finitely presented algebras. In *Proc. Ninth ACM Symposium on Theory of Computing*, pages 164–177, 1977.

[5] G. Nelson and D.C. Oppen. Fast decision algorithms based on congruence closure. *Journal of the ACM*, 27:356–364, 1980. Also in the 18th IEEE FOCS, 1977.

[6] R.E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31, 1984.

[7] Rakesh M. Verma. *Equations, Nonoblivious Normalization, and Term Matching Problems.* PhD thesis, State University of New York at Stony Brook, 1989.

[8] Rakesh M. Verma. Smaran: A congruence closure based system for equational computations. In Claude Kirchner, editor, *Proc. Conf. on Rewriting Techniques & Applications*, 1993.

[9] 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.

[10] Rakesh M. Verma. Static analysis for high-performance rewriting. Technical report, University of Houston, 1997.

[2] We are aware that the focus of the ELAN project is more on compilation

6 Appendix A

The Fibonacci program given in Section 2 naturally benefits from history. The other three programs are examples where no history can be reused except if the same term is repeated.

Program 1 *Huffman codes.*

$$\begin{aligned}
BuildHuff(:(Huff(weight1, tree1), nil)) &\rightarrow tree1 && (1)\\
BuildHuff(:(x, :(y, remainder))) &\rightarrow && (2)\\
BuildHuff(Insert(remainder, Combine(x, y))) & && (3)\\
Insert(nil, item) &\rightarrow :(item, nil) && (4)\\
Insert(:(Huff(weight2, tree2), remainder), Huff(weight1, tree1)) &\rightarrow && (5)\\
f(<(weight1, weight2), weight1, tree1, weight2, tree2, remainder) & && (6)\\
f(true, weight1, tree1, weight2, tree2, remainder) &\rightarrow && (7)\\
:(Huff(weight1, tree1), :(Huff(weight2, tree2), remainder)) & && (8)\\
f(false, weight1, tree1, weight2, tree2, remainder) &\rightarrow && (9)\\
:(Huff(weight2, tree2), Insert(remainder, Huff(weight1, tree1))) & && (10)\\
Combine(Huff(weight1, tree1), Huff(weight2, tree2)) &\rightarrow && (11)\\
Huff(+(weight1, weight2), :(tree1, tree2)) & && (12)
\end{aligned}$$

Program 2 *Quicksort program.*

$$\begin{aligned}
sort(nil) &\rightarrow nil && (13)\\
sort(:(x, y)) &\rightarrow append(sort(smaller(x, y)), append(:(x, nil), sort(larger(x, y)))) && (14)\\
smaller(x, nil) &\rightarrow nil && (15)\\
smaller(x, :(y, z)) &\rightarrow f(<(x, y), x, y, z) && (16)\\
f(true, x, y, z) &\rightarrow smaller(x, z) && (17)\\
f(false, x, y, z) &\rightarrow :(y, smaller(x, z)) && (18)\\
larger(x, nil) &\rightarrow nil && (19)\\
larger(x, :(y, z)) &\rightarrow g(<(x, y), x, y, z) && (20)\\
g(true, x, y, z) &\rightarrow :(y, larger(x, z)) && (21)\\
g(false, x, y, z) &\rightarrow larger(x, z) && (22)\\
append(nil, x) &\rightarrow x && (23)\\
append(:(x, y), z) &\rightarrow :(x, append(y, z)) && (24)\\
list(x) &\rightarrow h(>(x, 0), x) && (25)\\
h(true, x) &\rightarrow :(x, list(-(x, 1))) && (26)\\
h(false, x) &\rightarrow nil && (27)
\end{aligned}$$

Program 3 *Sieve of Eratosthenes. This is a naive sieve program to compute the first n prime numbers. The first three rules build a list of y numbers starting with x. The filter rules remove all the multiples of a number from a given list. The sieve rules use the filter rules to build a list of primes after the multiples have been removed. Sieve is always called with a list beginning with 2.*

$$\begin{aligned}
from(x, y) &\rightarrow ffrom(y > 0, x, y) && (28)\\
ffrom(false, x, y) &\rightarrow nil && (29)\\
ffrom(true, x, y) &\rightarrow cons(x, from(x + 1, y - 1)) && (30)\\
filter(n, nil) &\rightarrow nil && (31)\\
filter(n, cons(x, l)) &\rightarrow ffilt(x\%n = 0, n, x, l) && (32)\\
ffilt(true, n, x, l) &\rightarrow filter(n, l) && (33)\\
ffilt(false, n, x, l) &\rightarrow cons(x, filter(n, l)) && (34)\\
sieve(x, 0) &\rightarrow nil && (35)\\
sieve(nil, y) &\rightarrow nil && (36)\\
sieve(cons(x, l), y) &\rightarrow fsieve(y > 0, x, l, y) && (37)\\
fsieve(false, x, l, y) &\rightarrow nil && (38)\\
fsieve(true, x, l, y) &\rightarrow cons(x, sieve(filter(x, l), y - 1)) && (39)
\end{aligned}$$