Does Unification Help in Normalization?

Rakesh M. Verma and Wei Guo

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

Technical Report Number UH-CS-11-05

July 5, 2011

Keywords: Unification, Normalization.

Abstract

In this paper, we present a preprocessor for rules based on unification, which has the potential to enable faster search for potential redexes in normalization. We implement this idea in Laboratory for Rapid Rewriting (LRR) [1] and compare our method with ELAN [2] and Maude [3] using both favorable and unfavorable examples to demonstrate the performance.

Research supported in part by NSF grant CCF 0306475

Does Unification Help in Normalization?

Rakesh M. Verma and Wei Guo

Abstract

In this paper, we present a preprocessor for rules based on unification, which has the potential to enable faster search for potential redexes in normalization. We implement this idea in Laboratory for Rapid Rewriting (LRR) [1] and compare our method with ELAN [2] and Maude [3] using both favorable and unfavorable examples to demonstrate the performance.

Index Terms

Unification, Normalization.

I. Introduction

Fast rewriting is needed for equational programming, rewrite based formal verification methods, and symbolic computing systems. In any implementation of rewriting techniques efficiency is a critical issue [4]. The goals of this paper are to enhance the efficiency of the normalization by integrating a preprocessor for rules and to determine how much can unification help in future matching attempts in practice, especially when built-in operators such as arithmetic present complications. The immediate motivation is to effectively cut the time spent in traversing both the term and the rules in order to find a match. The preprocessor for rules utilizes the unification results obtained from a set of rules to facilitate matching. Our idea is applicable to any interpreter. Since we have been working on LRR , an interpreter for a rule-based programming language with an efficient history option, we use it as a platform to demonstrate our idea.

The rest of this paper is organized as follows. We first present some preliminaries including a brief introduction to LRR in Section 2. Then we discuss how unification helps in normalization in Section 3. The experimental results are presented in Section 4. In Section 5, we conclude the paper with some promising directions for future research.

II. Preliminaries

A *Term Rewriting System* is a set of rewriting rules, R, and a given term t_0. The objective is to compute a normal form of t_0, t_n. We denote the i^{th} 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 i^{th} 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*.

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 [5].

LRR is one of the interpreters for rule based programming. The input of LRR is a file representing the rules R and a file representing the given term t_0. It consists of a term graph interpreter TGR, and a term graph rewriter storing the history of its reductions, called Smaran, based on the Congruence Closure based Normalization Approach (CCNA) [6]. Smaran constructs signatures representing the terms and equivalence classes consisting of equivalent signatures. Please consult [6], [7] for more details. TGR uses Term Graph Rewriting which has no class or signature.

An extension of almost linear unification (ALU). The objective of unification is, given two terms l, r, to find a substitution σ such that $\sigma(l)$ and $\sigma(r)$ are syntactically identical. The ALU algorithm uses Directed Acyclic Graphs (DAGs) as the data structures of the terms and requires variables to be shared, which reduces the complexity from exponential to almost linear (please see [8] for more details). We extend the concept of unification as follows. Under strict unification, a constant never unifies with a function having at least one child. But if a function can be evaluated during the normalization and the constant is one of the possible results, we consider that they "unify".

Research supported in part by NSF grant CCF 0306475

For example, consider the Fibonacci function in appendix, either term $true$ or $false$ is the result of term $> (x, 1)$ and here we consider that $true$ and $false$ unify with $> (x, 1)$.

III. How Does ALU Help in Normalization

We find that many matches found in normalization happen between an instance $\sigma(r|_p)$ of a subterm $r|_p$ of a RHS r and an LHS l. This implies that the LHS unifies with this subterm of the RHS, since their variables are "effectively" disjoint. And if a subterm from a RHS can unify with a LHS, there is a great chance to find a match between the instance of the subterm and the LHS when the instance is built by the RHS. Before normalization starts, we add a preprocessor for rules which collects the unification results between LHS's and RHS's using ALU. In normalization procedure, we introduce a list, the ALU-list, to let the unification results help to find a match. In one step of normalization, instead of looking for a match by scanning all subterms of the term to be normalized and all the rules, our normalization procedure first looks for a match from the ALU-list. In Figure 1 below, in $t_{i-1} \rightarrow_{(i,j)} t_i$, a match is found between a subterm u of t_{i-1} and lhs_j. Then the subterm u is replaced by v, the instance of rhs_j, and we get t_i. Term v shares the same overall structure as rhs_j. If we know that a subterm $x = rhs_j|_p$ can unify with lhs_k, there is a great chance to find a match between term $w = v|_p$, the instance of term x, and lhs_k in the next step. In the $i + 1^{th}$ step, normalization can try the term w and lhs_k first. If a match is found, term w is replaced by the instance of rhs_k.

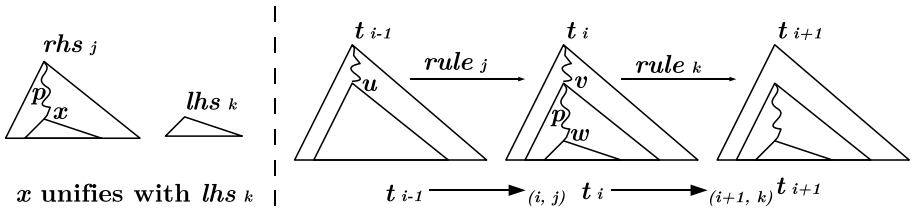

Fig. 1. Unification results can help in normalization

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 RHS's and much of the overall structure of terms can be safely predicted from the RHS's (the exceptions are the variable substitutions and unexplored parts of the intermediate terms). If before normalization, we know the unification results between each subterm in every RHS and each LHS, we can use the results to find the matches in normalization efficiently. But not all the unification results lead to successful matches. In this case, and for normalizing t_0, LRR calls original Smaran or TGR to find matches.

A preprocessor for rules tries to unify every subterm in every RHS with every LHS and stores the successful results denoted by a list of pairs (C, P) before normalization. In Figure 1, subterm x from rhs_j unifies with lhs_k. We say that $rule_k$ is a *candidate* and for this example $C = k$. We define the position a *point*, which for this example would be stored in $P = p$. Storing term x in the node does not help normalization but storing P does because normalization needs to find w by following P from v. The preprocessor stores the indexes of the rule in C and uses a single linked list to store P which is essentially a sequence of natural numbers. There may be more than one pair for each RHS. For every RHS, the preprocessor uses a single linked list to store the nodes.

The ALU-list is a singly-linked list to store the information obtained from the unification results during the normalization. Each node in the list is a 3-tuple (i, c, s), where i indicates the i^{th} 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. We use stack operations to implement a depth-first order in normalization. In one step, tuples obtained from the nodes of the RHS that is applied in this step are pushed into the ALU-list. In the next step, the ALU-list pops a tuple (i', c', s') and tries to match the term s' and $lhs_{c'}$. If they match, LRR continues normalization. If not, the ALU-list pops the next tuple. When the ALU-list is empty, normalization goes back to the original algorithm searching for new match. In LRR, normalization goes back to Smaran or TGR. In Figure 1, rhs_j has a node (k, p) in which $x = rhs_j|_p$. Normalization locates term $w = v|_p$ and pushes the tuple (i, k, w) into the ALU-list. If the tuple is popped at the $i + 1^{th}$ step, $rule_k$ is applied at term w to form term t_{i+1}.

TABLE I
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	2.2301	2.6398	1.8197	1.7829
bintree(380)	0.1152	0.0044	0.0936	0.0160	0.0144	0.0116	0.0116
dfa(1363)	0.0016	0.0000	0.0008	0.0540	0.0540	0.0396	0.0424
fib(20)	1.4416	0.0272	0.0000	0.0000	0.0004	4.4851	4.5507
merge(20000)	17.8455	0.0404	70.2658	0.0460	0.0524	0.0300	0.0296
qsort(1800)	66.6180	1.1872	30.7426	10.0294	8.8518	3.4254	3.3874
rev(19900)	66.6304	0.0380	129.6359	0.0484	0.0548	0.0328	0.0332
rfrom(19996)	1.7005	0.0408	44.1588	0.0384	0.0408	0.0224	0.0220
sieve(10000)	169.6300	0.4900	29.6235	1.5889	1.7709	0.8277	0.8257

A. Optimizations

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

Mutually exclusive detection is a method to cut unnecessary insertions into the ALU-list caused by the extension of ALU. For example, both terms $true$ and $false$ are considered as candidates for the term $> (x, 1)$. We add two nodes in unification. But only one tuple will succeed in matching. So, by evaluating the term $> (x, 1)$ before pushing a candidate into the ALU-list, normalization picks up only the "right" tuple.

Candidate elimination contains three ways to delete tuples from the ALU-list. *Same point elimination* is a method to cut unnecessary matching attempts. Tuples having the same value of i and same value of s apply at the same point of the same instance of the RHS. Once we find the first match from these tuples, which have the same value of s, the intermediate term probably will change in the next reduction step and the remaining tuples are deleted. *Descendants elimination* also cuts unnecessary matching attempts. If the parent succeeds in matching, the matching attempts for its children are unnecessary since the intermediate term probably will change. *Changed signature check* cuts unnecessary matching attempts only when the preprocessor works with Smaran. Smaran checks whether the unreduced signature of the class has changed since the tuple containing the class was added into the ALU-list. If yes, LRR deletes the tuple.

IV. EXPERIMENTAL RESULTS

The preprocessor for rules has still some room for improvement. A Linux version of LRR v3.0 and some examples can be downloaded from `http:/www.cs.uh.edu/~evangui`. LRR v3.0 provides: i) the original Smaran and TGR, ii) a preprocessor for rules with original Smaran and with original TGR. In the reference [1], [9], [6], several optimizations are discussed including structure sharing, and CCNA. We use Maude 2.6 32-bit version which can be found at `http://maude.cs.uiuc.edu/download`. We use 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 in [10] for lack of space) 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 same algorithm. Rules in the benchmark are 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. 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. We plan to track the positions of variables in a RHS and direct LRR to try terms covered by the variables rather than traversing from the root. 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.

Related work. We did an extensive search for rule-based programming interpreters using the papers [4], [11] and the Rewriting Page on the web, but we have been unable to find any interpreter that includes any such application of unification to speed up the matching process during normalization. Apart from Maude, in [11] 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 [12], which does not include built-ins and could only handle a string of length 819 in the dfa example.

V. Discussion and Future Work

We presented a preprocessor for rules, a method to improve the efficiency of normalization. The preprocessor beats the earlier strategy in accurately finding the next match. We plan to implement the preprocessor in a more efficient way and try to use more information from unification to help speed up the normalization even more.

Acknowledgments. We want to thank S. Senanayake, J. Thigpen, and H. Shi for initial work on LRR, and Z. Liang for some examples.

References

[1] R. Verma and S. Senanayake, "LR^2 : A laboratory for rapid term graph rewriting," in *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, 1999, pp. 252–255.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, "Elan: A logical framework based on computational systems," *Electr. Notes Theor. Comput. Sci.*, vol. 4, pp. 35–50, 1996.

[3] P. L. M. Clavel, S. Eker and J. Meseguer, "Principles of maude," in *Electronic Notes in Theoretical Computer Science*, J. Meseguer, Ed., vol. 4. Elsevier Science Publishers, 1996, pp. 65–89.

[4] M. Hermann, C. Kirchner, and H.Kirchner, "Implementations of term rewriting systems," *The Computer Journal*, vol. $34(1)$, pp. 20–33, 1991.

[5] N. Radcliffe and R. M. Verma, "Uniqueness of Normal Forms is Decidable for Shallow Term Rewrite Systems," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. Lodaya and M. Mahajan, Eds., vol. 8. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 284–295. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2871

[6] R. M. Verma, "Smaran: A congruence-closure based system for equational computations," in *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, 1993, pp. 457–461.

[7] H. Shi, "Intergrating associative and commutative matching in the LR^2 laboratory for fast, efficient and practical rewriting techniques," Master's thesis, University of Houston, 2000.

[8] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1999.

[9] R. Verma, "Static analysis techniques for equational logic programming," in *Proceedings of the 1st ACM SIGPLAN Workshop on Rule-based Programming*, 2000.

[10] W. Guo and R. Verma, "Does unification help in normalization?" University of Houston Computer Science Department, Tech. Rep. UH-CS-11-05, 2011.

[11] M. Vittek, "A compiler for nondeterministic term rewriting systems," in *RTA*, 1996, pp. 154–167.

[12] J. W. Klop, V. van Oostrom, and F. van Raamsdonk, "Combinatory reduction systems: Introduction and survey," *Theor. Comput. Sci.*, vol. 121, no. 1&2, pp. 279–308, 1993.

Appendix

A Concrete Example

To illustrate the details, we use a concrete example in LRR which computes Fibonacci numbers (we use Fibo for short).

$$fib(x) \Rightarrow f(>(x,1),x) \quad (1)$$

$$f(true,x) \Rightarrow +(fib(-(x,1)),fib(-(x,2))) \quad (2)$$

$$f(false,x) \Rightarrow 1; \quad (3)$$

The normalization process using TGR is (under a depth-first left-most order) 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 1^{st} step, LRR calls Smaran or TGR to initiate matching because the preprocessor has no information. $rule_1$ is picked up and $t_1 = f(true, 2)$. Then, the preprocessor knows that the term $f(> (x, 1), x)$ unifies with lhs_2:$f(true, x)$, and lhs_3:$f(false, x)$. It looks like LRR tries to match $f(true, 2)$ with $f(true, x)$ and $f(false, x)$. But in Section III.A, we show that LRR picks up only $f(true, x)$ while the original LRR attempts to match $f(true, 2)$ with all 3 rules. LRR picks up $rule_2$ and gets $t_2 = +(fib(1), fib(0))$. The preprocessor still knows that the terms $fib(-(x, 1))$ and $fib(-(x, 2))$ unify with lhs_1:$fib(x)$. Under a depth-first left-most order, LRR starts from $fib(1)$ and succeeds in matching $fib(1)$ with $fib(x)$ while the original LRR traverses from the root of $+(fib(1), fib(0))$ searching for a match. LRR picks $rule_1$ and gets $t_3 = +(f(false, 1),\ fib(0))$. After 6 steps, LRR stops at the normal form, 2.

In ALU, for each RHS, it is possible that some subterms unify with multiple rules. Thus, there may be more than one pair for each RHS. After LRR parses all the rules and before it starts the normalization, for each RHS, the preprocessor tries to unify every subterm with every LHS and stores the results for each RHS. In Fibo, $f(> (x, 1), x)$ unifies with lhs_2:$f(true, x)$, lhs_3:$f(false, x)$; $fib(-(x, 1))$ and $fib(-(x, 2))$ unify with lhs_1:$fib(x)$. So, $rule_1$ has a list of two pairs $(2, \lambda)$, $(3, \lambda)$. $rule_2$ has a list of two pairs $(1, (1))$, $(1, (2))$.

In normalization, for the combination of Smaran and ALU, s in the 3-tuple (i, c, s) is the number of the class containing the term, indicating the unreduced signature of the class. For the combination of TGR and ALU, s is the term.

In Fibo, $rule_2$ is used in the 2^{nd} step. So LRR (using TGR) gets two nodes from $rule_2$, $(1, (1))$, $(1, (2))$, creates two tuples $(2, 1, fib(1))$ and $(2, 1, fib(0))$, and pushes them into the ALU-list. In the 3^{rd} step, $(2, 1, fib(1))$ is at the top of the ALU-list. So LRR pops the tuple and tries to match $fib(1)$ with lhs_1. Since they match, LRR builds t_3 and evaluates the built-in operations. LRR creates 2 tuples $(3, 2, f(false, 1))$, $(3, 3, f(false, 1))$ but only pushes $(3, 3, f(false, 1))$ into the ALU-list because of mutually exclusive detection.

Optimizations

Mutually exclusive detection. We find that the extension of ALU brings us some overhead. For example, both terms $true$ and $false$ are considered as candidates for the term $> (x, 1)$. Thus, two nodes are stored by the preprocessor. But eventually only one rule will succeed in matching. So, it is better to push the "right" tuple only into the ALU-list in normalization. LRR builds the instance of the RHS and evaluates the built-in operations before it pushes tuples. It is easy to detect the value of $> (x, 1)$ and pick up the "right" tuple. In most cases, the terms allowed by the extension of ALU, such as $true$ and $false$, are mutually exclusive. In ALU, our method adds two fields to the node which becomes (C, P, LOC, VAL). LOC indicates the position of the term $l|_{LOC}$ in a LHS l. VAL indicates the value of the term. Normalization copies LOC and VAL into the tuple which becomes (i, c, s, loc, val). LRR picks up the "right" tuple in which val is equal to the value of the instance $\sigma(rhs_c|_{loc})$. So far, the detection can detect $true$, $false$, and : (nonempty list), nil (empty list). And the LOC can only stores one integer which indicates the n^{th} child of the root of a RHS. In our benchmarks, this detection is sufficient. In Fibo, nodes under $rule_1$ turn to $(2, (\lambda), (1), true)$, $(3, (\lambda), (1), false)$ indicating the 1st child of the root of lhs_2 is $true$, and the 1st child of the root of lhs_3 is $false$. In the 2^{nd} step, LRR detects the value of $true$, and thus picks the tuple $(1, 2, true, (1), true)$ obtained from the node $(2, (\lambda), (1), true)$ and drops the tuple $(1, 3, true, (1), false)$ which is obtained from the node $(3, (\lambda), (1), false)$.

Same point elimination It is possible that more than one LHS unify with one subterm in a RHS. Therefore the RHS gets more than one node with same P value in unification and more than one tuple with same s value are pushed into the ALU-list. However, from these tuples, once we find the first match, the value of s probably changes in the next step of normalization and the rest tuples have little chance to lead to a new match. So, we just delete the rest tuples. In ALU, we add one more field, $SAME$, into the node. Different points are represented by

different values of $SAME$. Normalization copies the value of $SAME$ to make a new tuple $(i, c, s, same)$. During the normalization, the ALU-list may contain tuples from different steps of normalization. We find that the tuples in the ALU-list are in the lexicographical order of $(i, same)$ in LRR. So after a successful match, normalization advances to the next tuple and eliminates the tuples with the same value of i and the same value of $same$. The same point elimination stops when a tuple with a different i or a different $same$ is met. In Fibo, if not considering about mutually exclusive detection, in the 2^{nd} step, after a matching between $f(true, 2)$ and lhs_2, we can delete the tuple $(1, 3, true, same)$.

Descendants elimination. In ALU, we find that a parent and its descendants in a RHS may unify with either one or more rules. In normalization, if the parent succeeds in matching with term t, the matching attempts for its descendants are unnecessary because in the following steps of normalization t probably changes. In LRR, nodes are stored in a depth first order, so are tuples. It is easy for LRR to locate descendants upon a successful match. We add a LVL into the node indicating the level of the subterm in the RHS. We copies this value to the tuple as lvl. Upon a successful match from tuple (i, c, s, lvl), we keep eliminating the next tuple (i', c', s', lvl') if $i' = i$ and $lvl' > lvl$.

Changed signature check. In Smaran, LRR constructs the signature to represent the term and the class to hold equivalent signatures. Smaran tries to match the unique unreduced signature of a class and LHS's. In the tuple (i, c, s), s stores the class of the subterm in t_i not the unreduced signature. In fact, unreduced signatures keep changing during the normalization. LRR checks whether the unreduced signature of class s has changed since the tuple was added into the ALU-list. If yes, LRR will delete the tuple. If not, LRR will try to match. TGR does not need this check since in TGR s represents the term which never changes.

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.

$$ins(x, nil) \Rightarrow node(nil, x, nil) \quad (5)$$
$$ins(x, node(l, v, r)) \Rightarrow instest(x, >(x, v), <(x, v), l, v, r)) \quad (6)$$
$$instest(x, false, true, l, v, r) \Rightarrow node(ins(x, l), v, r) \quad (7)$$
$$instest(x, true, false, l, v, r) \Rightarrow node(l, v, ins(x, r)) \quad (8)$$
$$instest(x, false, false, l, v, r) \Rightarrow node(l, v, r) \quad (9)$$
$$cat(:(x, y), z) \Rightarrow :(x, cat(y, z)) \quad (10)$$
$$cat(nil, z) \Rightarrow z \quad (11)$$
$$binsort(:(x, y)) \Rightarrow bs(ins(x, nil), y) \quad (12)$$
$$bs(n, :(x, y)) \Rightarrow bs(ins(x, n), y) \quad (13)$$
$$bs(n, nil) \Rightarrow makelist(n) \quad (14)$$
$$makelist(node(l, v, r)) \Rightarrow cat(makelist(l), :(v, makelist(r))) \quad (15)$$
$$makelist(nil) \Rightarrow nil \quad (16)$$

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 n^{th} 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.

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

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

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

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

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