An Abstract Concurrent Machine for Rewriting

Nachum Dershowitz
University of Illinois
Urbana, IL 61801
U.S.A.
nachum@cs.uiuc.edu

Naomi Lindenstrauss
The Hebrew University
Jerusalem 91904
Israel
naomil@humus.huji.ac.il

Abstract

Term rewriting corresponds to reduction in applicative languages; narrowing of terms corresponds to goal reduction in logic languages. An abstract machine is described for rewriting and narrowing. It has been implemented in Flat Concurrent Prolog, but could be coded in any system in which processes are capable of creating other processes and communicating with each other.

1 Introduction

Term rewriting is a powerful computational paradigm. On the one hand, it has the full power of Turing machines, because they can be formulated as rewrite systems, and on the other hand, it lends itself to parallel execution, since different subterms of a term can be rewritten concurrently.

A *rewrite rule* over a set of (first-order) terms is an ordered pair of terms, which we write as $l \rightarrow r$. A *rewrite system* is a finite set of rules. Given a rewrite system R, we say that a term s *rewrites* to a term t at some position p, denoted by $s \rightarrow_R t$, if there is a rule $l \rightarrow r$ in R and a substitution σ, such that the subterm of s at position p is $l\sigma$ and t is s with that subterm replaced by $r\sigma$. The position at which a rewrite applies is called a *redex.* A term is said to be in *normal form* if no rewrite rule can be applied to it. It should be stressed that any variables in s remain uninstantiated by rewriting.

We also consider a more general operation, known as "narrowing", in which a rule is applied if its left-hand side *unifies* with any subterm of s. More precisely, we say that a term s *narrows* to a term t (at position p), denoted $s \rightsquigarrow_R t$, if there is a rule $l \rightarrow r$ in R such that σ is the most general unifier of the subterm of s (at p) with l, and t is the result of applying σ to s and then replacing that subterm $l\sigma$ of $s\sigma$ with $r\sigma$. If $s\sigma$ is different from s, we will call it a *proper* narrowing. We assume that unification always treats variables in rules as distinct from those in the term.

In this paper, we show how different rewriting and narrowing strategies can be implemented on an abstract concurrent machine. The way our abstract machine works for rewriting is that any given term t that we wish to rewrite has a process assigned to it. The process associated with a term may spawn other processes associated with its arguments, and so on. Thus, a dynamic system of processes will develop, with processes being created and terminating, until finally only the root process will remain, containing a term that is the normal form of t. The advantage of this scheme is that the division of the rewriting task into subtasks is done automatically in correspondence with the structure of the term that is to be rewritten. The only communication is between a process and those processes directly spawned by it, so if the processes are assigned to different processors, only a minimal amount of inter-processor communication is necessary. The scheme can incorporate various strategies of concurrent rewriting, including innermost-first, top-down, and outermost-first strategies. We can also use different strategies for different subterms, and the E-strategies of [GoKiMe 86] can be easily incorporated. The advantage of our design is its conceptual simplicity, the minimal amount of inter-processor communication necessary, and, as the examples show, good performance. Another advantage is that all processors (in the rewriting part) share the same program, as in the systolic approach to concurrent programming (see [Shap 87]).

It turns out that this form of concurrent rewriting can give a significant speed-up compared to sequential rewriting. Our abstract machine is powerful enough to compute Fibonacci numbers or solve the Towers of Hanoi problem in linear time, to merge and sort in polylogarithmic time, to compute algebraic expressions of length n in average $O(\sqrt{n})$ time, and to evaluate Boolean expressions of length n in average constant time (of course under the assumption that the primitive operations of the abstract machine take constant time). Moreover, it "discovers" by itself the parallel bits algorithm for adding two numbers in time logarithmic in their length. Also, it solves some problems of coordination that arise in parallel execution.

We sometimes require that a rewrite system be terminating (that is, there is no infinite sequence $t_1 \rightarrow_R t_2 \rightarrow_R \ldots$), and/or satisfy the ground Church-Rosser property (that is, if $t \rightarrow_R \cdots \rightarrow_R t_1$ and $t \rightarrow_R \cdots \rightarrow_R t_2$ for variable-free term t, then there is an s such that $t_1 \rightarrow_R \cdots \rightarrow_R s$ and $t_2 \rightarrow_R \cdots \rightarrow_R s$). With these requirements, every ground (variable-free) term has a unique normal form that does not depend on the order in which rewrite steps are taken, and the rewriting algorithms we describe return that normal form. Without termination, a particular rewriting strategy might go on forever, not finding any normal form; with termination but without the Church-Rosser property, it will find only one out of potentially many normal forms. For a survey of rewriting, see [DerJou 90].

Following [JoDer 89], the version of our abstract machine that we describe here alternates between normalizing (rewriting to normal form) and narrowing. In this approach, called *normal narrowing*, rewriting steps are never retraced and are given preference over (proper) narrowing steps. Normal narrowing is guaranteed to find a solution to a goal whenever there is one only if the given system is terminating and Church-Rosser. Since, as we will see, the machine essentially searches a tree for solutions obtainable by narrowing, various strategies from the spectrum between depth-first, which is simpler but not complete, to breadth-first, which uses more resources, can be entertained.

Concurrent term rewriting has been investigated by [GoKiMe 86], [SePaRa 89], and others. Unlike [SePaRa 89], we do not make any assumptions regarding the system other than the termination and Church-Rosser properties needed for completeness. On the other hand, we do not consider all possibilities of concurrent rewriting as in [GoKiMe 86], only those that correspond to configurations of our abstract machine. In [GoKiMe 86], a step of concurrent rewriting with respect to a non-overlapping set of redexes W is defined as the result of rewriting at those redexes with a bottom-up sequential strategy, and a maximal concurrent rewriting is defined as concurrent rewriting with W maximal in the set of all non-overlapping subsets of redexes with the partial order of inclusion. But, as they state, "It is not practical to implement maximal concurrent rewriting, and even if it were, it would sometimes be undesirable." Even in the sequential case, finding an optimal rewriting sequence is NP-complete [LiPal 89].

We have implemented the abstract machine in Flat Concurrent Prolog (for the details, see [DerLin 90]), but it may be coded in any system in which processes are capable of creating other processes and communicating with each other.

2 Rewriting

The main feature of the abstract machine is a capability to create processes, with a channel of communication between each process and its creator. The rewriting algorithm considers terms as processes that may spawn other processes associated with their arguments. Thus, the rewriting task is broken down into subtasks corresponding to the structure of the given term.

The input to a new process is a term, for which the process and its offspring will look for a normal form and then terminate. All processes run the same program, which depends on the strategy. Each process can either rewrite its term at the root (that is, apply rewrite rules applicable to the whole term), or create processes for its arguments and wait for their answers. (In the FCP implementation these are implemented as perpetual processes.) Given a term t to rewrite, a process associated with it is created. This process may create, according to the rewriting strategy, processes associated with the arguments of t (e.g. if $t = f(t_1, \ldots, t_n)$, these will be processes associated with $t_1, \ldots, t_n$), and so on.

2.1 Basic version

The processes can do several things, depending on the state they are in. In the basic version, they can be in one out of four states, which we call 'self', 'children', 'wait', and 'stop'. The initial state is determined when the process is created and will differ with different rewriting strategies, such as innermost-first, top-down, etc. How the state will change also depends on the rewriting strategy.

A process in the state 'self' tries to rewrite itself at the root according to some rewrite rule. What happens next is determined by "annotated" rules. Instead of the usual rewrite rules of the form

$$Left \longrightarrow Right$$

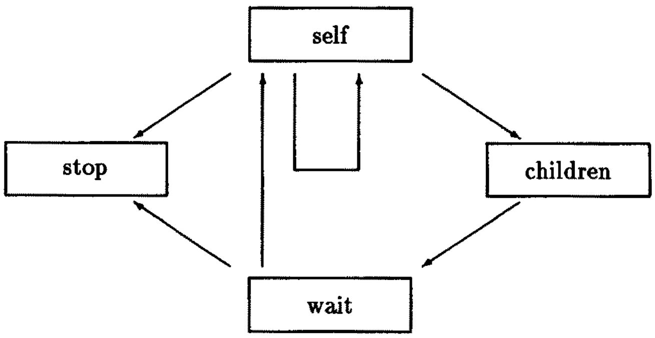
meaning that *Left* can be rewritten to *Right*, we use rules of the form

$$Left \longrightarrow Right \qquad : State$$

where *State* specifies into what state the process will go after successfully applying the rule.

In the state 'children', the process creates children corresponding to its arguments and then goes into state 'wait' to wait for their answers. After the answers have arrived, or if there are no children, it applies a test, supplied with the rules, to decide into which state to go next. There is also a state 'stop', in which a process sends the value of its term to its parent and terminates. This usually happens when it is clear that the term associated with the process has reached its normal form.

The following diagram describes the transitions:



In addition to giving the annotated rewrite rules, one has also to specify what the initial state of a process will be and what state a process will go into once it receives answers from its children. (These will be different for different rewriting methods, and may also depend on the term.)

2.2 Different Strategies

When there is more than one redex in a term, there are alternative orders in which rewriting can take place. In a parallel situation, it is often possible to apply rules at more than one redex at the same time. When one redex is above another, it is usually not practical to do both at once, since the result of applying one rule can make the other inapplicable.

One possible strategy is the innermost-first approach in which a subterm is rewritten at the root only if none of its subterms can be rewritten. This has the advantage of maximizing the amount of concurrency possible at "disjoint" redexes. To get this manner of rewriting with our machine, the initial state of each process is taken as 'children'; a process that has received the answers from its children goes into state 'self', and if no rule applies it sends its term to its parent and terminates. All rules are of the form

$$Left \longrightarrow Right \qquad : children$$

When all instances of *Right* are in normal form, one can use the more efficient rule

$$Left \longrightarrow Right \qquad : stop$$

An alternative is a top-down approach, in which we first try to rewrite a term by a rule that applies at the root, and only if this is not possible are children created. The process waits for them to return the normal forms of its arguments. For this strategy, the initial state of each process is 'self'; a process that has received answers from its children goes into state 'self', unless the answers have not changed the term, in which case it goes into state 'stop'. The rules take the form

$$Left \longrightarrow Right \qquad : self$$

Both these strategies are valid in the sense that one is guaranteed to end up with the normal form of the initial term. With arbitrary strategies, one must ascertain that the rigid choices made will not cause the machine to stop without reaching a normal form. For rewriting strategies compare [Küch 82], who uses the term "admissible" instead of what we call "valid", and [Stick 83].

To give an example in which one strategy is clearly better than another—consider a finite ring with operations *add* and *mult* given by appropriate rules. Suppose we also have rules

$$\begin{aligned} mult(x,0) &\longrightarrow 0 \\ mult(0,x) &\longrightarrow 0 \end{aligned}$$

In this case, the top-down strategy is better for *mult* because, if t is some complicated multiplicative expression and we try to rewrite $mult(t,0)$, the top-down strategy will immediately use the appropriate rule and rewrite it to 0, while the innermost-first strategy will first reduce t to normal form, bottom-up. For terms containing both *add* and *mult*, it may be advantageous to use a top-down strategy for subterms with main operator *mult* (that is, start in state 'self') while using an innermost-first strategy for subterms with main operator *add* (that is, start with state 'children').

There are several ways in which the performance of the basic machine can be improved.

- There is no need to create a process for a term whose main operator is a "free constructor", that is, an operator for which there are no rewrite rules. For example, if a process associated with the term $f(h(t_1), g(t_2))$ wants to create children and h and g are constructors, it is enough to create processes for t_1 and t_2.

- It is possible to mark terms which have already reached their normal form. Suppose we want to rewrite $f(t_1, t_2)$ and t_1 is in normal form. Then, when the process associated with $f(t_1, t_2)$ creates children, it is enough to create a child only for t_2.
- We may incorporate the concurrent E-strategies of [GoKiMe 86]. If f is an operator of arity n, then a concurrent E-strategy for f is a subset I of $\{1, \ldots, n\}$ with the operational interpretation that the arguments indexed by I must be fully reduced before a rule is applied to a term with main operator f. For example, if we want to rewrite $if(c, a, b)$, we may work on the first argument c until we get an answer from that child, and only then apply the appropriate rule for $if(\mathbf{true}, a, b)$ or $if(\mathbf{false}, a, b)$.

These variations can be expressed within the framework of our abstract machine by replacing the state 'children', in which child-processes for all arguments are created, by more specific states *arguments*(I), where I can be as in the definition of the E-strategies or a description by occurrences of positions of subterms for which we want to create processes.

2.3 More powerful variations

In the basic version of the machine, all processes run the same program, and there is no difference between them except for differences caused by the terms with which they are associated. More powerful versions can be conceived, but are more complicated to implement.

For instance, one can implement a parallel version of outermost-first. In the sequential case, outermost-first rewriting will rewrite at the first (leftmost) redex at which a rule applies if one goes over the term considered as a tree in a depth-first manner. One can define a step of parallel outermost-first rewriting as applying rules to *all* redexes that are not contained within other redexes. For this case, one can conceive a machine that will work as follows: First, the root process tries to rewrite itself. If this is impossible, it creates children, and they create children, and so on, until children are reached that can rewrite their term at the root, or else their term is an irreducible constant. Such children transmit their results to their parents and terminate. Any other process that receives answers from all its children also sends its result to its parent and terminates. Thus, the root process always receives an answer in finite time. When it does not change its term, it knows it has reached a normal form and can terminate. Otherwise, it will proceed as before. In this version, the behavior of a process depends on its position in the tree.

Another possibility is for one process to be able to tell other processes to terminate. For instance, a process may launch all its children but not wait for answers from all of them. Instead, it proceeds when it has enough information, and tells those offspring processes that have become superfluous to terminate. In such cases, there must be a possibility for a process to transmit a halt signal to its offspring. (In the FCP version, all communication between processes is done via shared variables.) For example, if one rewrites $if(C, A, B)$, and C has been rewritten to *true*, one can stop working on B; or if one computes $mult(A, B)$ and has discovered that A rewrites to 0, there is no longer any need to rewrite B; or if one rewrites $and(A, B)$ and A has been rewritten to $false$, there is no need to work on B. This variation can be incorporated within our framework by replacing the 'wait' state, in which a process waits for answers from all its children, by states $await(Condition)$. In the above examples, these would be, respectively, the states $await(child_1 = \mathbf{true} \vee child_1 = \mathbf{false})$, $await(child_1 = 0 \vee child_2 = 0)$, $await(child_1 = \mathbf{false} \vee child_2 = \mathbf{false})$. When *Condition* becomes true, a rule can be applied at the root of the term, and the superfluous processes are halted.

This last strategy is more complicated than all the ones we encountered before. At each stage of the computation, there is a tree of processes, some of which are waiting and some are busy rewriting their term by applying a rule at its root. With the other strategies, the busy processes are exactly those at the leaves of the process tree. Here, we do not wait for answers from all sons, but proceed to rewrite the parent term at the root the moment we have enough information to do so. In other words—inner nodes may also be active.

With certain simple rewrite systems, we may have busy nodes throughout the process tree. (Usually, this has to be avoided, and that is the reason for using non-overlapping redexes in the concurrent term rewriting of [GoKiMe 86], and for the coordination between parents and children in the case of our concurrent machine.) It may happen that the moment a child has discovered what its main operator will be, it can already return this partially evaluated result to its parent, who may proceed to use it for rewriting at the root. Also, if the main operator of a term becomes a constructor, it is possible to terminate its process after connecting its children in an appropriate way to its parent. This will be illustrated later in an example of search trees and skew heaps.

The formalism of annotated rewrite rules can be used for enforcing certain ways of rewriting in systems which would not be terminating if all ways of rewriting would be allowed. For instance, if we want to create a stream of primes by the Sieve of Eratosthenes method, we use a rule

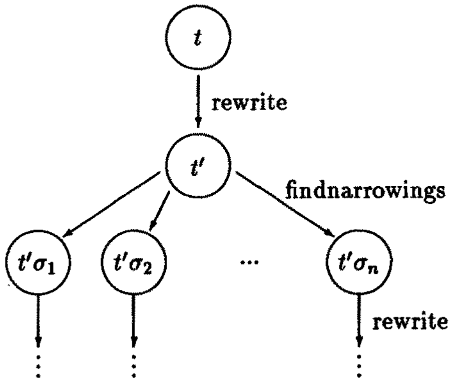
$$integers(n) \longrightarrow n \cdot integers(n+1) \qquad : stop$$

The added state prevents non-termination. (In [TamSa 83] and [DerPl 88], the same problem is solved by introducing new function symbols and using narrowing.)

3 Narrowing

The problem we face in this section is the following: Given a rewrite system R and a term t, is it possible to find a substitution σ such that $t\sigma$ can be rewritten to *true*? Note that solving one part of a conjunctive goal $and(s, t)$, by finding that $t\sigma$ reduces to *true*, does not mean that one has made progress, since there may be no instance of $s\sigma$ that holds, in which case alternative solutions to t must be considered.

The approach we adopt here is the following: First, we rewrite t by the methods of the previous section as much as possible, treating the variables that appear in t as if they were constants. In this way, we obtain a term t', which cannot be rewritten further. Next, we call a primitive operation, $findnarrowings(t')$, which returns all substitutions $\sigma_1, \ldots, \sigma_n$ such that $t'\sigma_i$, $i = 1, \ldots, n$, can be rewritten. The operation $findnarrowings$ can be done in parallel: for each position in the term and rule in the program, check if the subterm at that position can be narrowed by the particular rule. For each of the $t'\sigma_i$, we recursively apply the same process we applied to t. In such a way, we get a "narrowing" tree



in which we have to search for a node that is *true* (or a variable, for which *true* can be substituted).

The situation here is very similar to that faced with when one wants to search the and-or tree determined by a logic program. One must choose a strategy that lies somewhere between a depth-first search, which is usually efficient but incomplete, and a breadth-first search, which is very wasteful of resources when looking for only one solution.

Given narrowing substitutions $\sigma_1, \ldots, \sigma_n$, the narrowing algorithm must decide which to consider first. For this purpose, we supply a predicate *choose*$([\sigma_1, \ldots, \sigma_n])$ that returns those narrowing substitutions that will be considered first. Only if these substitutions lead to failure will other substitutions be considered. For a depth-first search, *choose*$([\sigma_1, \ldots, \sigma_n])$ returns $[\sigma_1]$. For breadth-first, it should return the list $[\sigma_1, \ldots, \sigma_n]$. The operation *choose* can also apply heuristic considerations, but one must take care lest completeness (finding a solution if there is one) is lost.

4 Examples

In this section, we give examples both of rewriting and narrowing strategies. Further examples, for instance, merging and sorting in polylogarithmic time and the computation of algebraic expressions of length n in average $O(\sqrt{n})$ time, can be found in [DerLin 90].

4.1 Towers of Hanoi

The Towers of Hanoi problem can be formulated for the basic version of the machine via rules:

$$\begin{array}{l} hanoi(0,x,y,z) \longrightarrow move(x,y) \quad : stop \\ hanoi(s(n),x,y,z) \longrightarrow \\ \quad t(hanoi(n,x,z,y), move(x,y), hanoi(n,z,y,x)) \\ \qquad\qquad\qquad\qquad\qquad\qquad : children \end{array}$$

initial state 'self', and a test that says that a process that has received answers from its children goes into state 'stop'.

Given a term $hanoi(N,A,B,C)$, where N is an integer written in successor notation (e.g. $N = s(s(s(0)))$), and rewriting it to its normal form, we get a list of the moves to perform in order to move $s(N)$ disks from pole A to pole B using auxiliary pole C, subject to the well-known requirements. Note that we use here a notation for lists that is different from the usual notation, with the help of a constructor t of arity 3. This representation gives a more balanced tree than the usual binary representation, which is completely lopsided. (Cf. [GoKiMe 86, p. 3] for the importance of replacing list structures by more balanced structures in a concurrent environment.) In this case, the sequential time for rewriting $hanoi(N,A,B,C)$ is $\Theta(2^N)$, while our parallel method takes $\Theta(N)$.

4.2 A three-element group

Suppose we have a group of three elements a, b, and e (the identity), with rewrite rules that define a multiplication operator, and two additional rules that say that the third power of any element is the identity. We can compute with these rules, that is reduce multiplicative terms to their normal form which is their value.

For the top-down strategy the rules are as follows:

$$\begin{array}{lll} mult(x, mult(x,x)) \longrightarrow e & & : stop \\ mult(mult(x,x),x) \longrightarrow e & & : stop \\ mult(e,x) \longrightarrow x & & : self \\ mult(x,e) \longrightarrow x & & : self \\ mult(a,a) \longrightarrow b & & : stop \\ mult(a,b) \longrightarrow e & & : stop \\ mult(b,a) \longrightarrow e & & : stop \\ mult(b,b) \longrightarrow a & & : stop \end{array}$$

The initial state of processes will be 'self', and a process that receives answers from its children goes into state 'self', if they have changed it, and into state 'stop', otherwise.

In this case, the machine will find the normal form of $mult(t, mult(t,t))$ for any complicated term t by one relatively expensive step of rewriting (it must ascertain that the t's are identical), while an innermost-first strategy will create processes for each multiplicative expression in the term.

4.3 Parallel bits algorithm

The parallel bits algorithm enables one to add two numbers in parallel, given their bit-representation, in time proportional to the logarithm of their length. As we shall see, our machine invents this algorithm by itself.

Suppose we are given binary numbers with 2^n bits represented as balanced binary trees with the help of a constructor t. (As we have already pointed out, in a parallel environment, lists have to be represented in a more balanced way than with *cons*.) If, for example, $n = 2$ and the number is 1101, we represent it as $t(t(1,1), t(0,1))$. The result of adding two such numbers is a number of the same kind and possibly a carry bit. We represent the result as a pair $[Carry, Sumtree]$, where $Sumtree$ is the tree representing the 2^n lower bits of the sum, and $Carry$ is the $2^n + 1$st bit. We can formulate rewrite rules for performing the addition of two numbers represented as trees of same size:

$$\begin{array}{ll}
carry([x,y]) \longrightarrow x & \\
rest([x,y]) \longrightarrow y & \\
t([x,y],z) \longrightarrow [x,t(y,z)] & \\
if(1,x,y) \longrightarrow x & \\
if(0,x,y) \longrightarrow y & \\
add(0,0,0) \longrightarrow [0,0] & add(1,0,0) \longrightarrow [0,1] \\
add(0,0,1) \longrightarrow [0,1] & add(1,0,1) \longrightarrow [1,0] \\
add(0,1,0) \longrightarrow [0,1] & add(1,1,0) \longrightarrow [1,0] \\
add(0,1,1) \longrightarrow [1,0] & add(1,1,1) \longrightarrow [1,1] \\
add(c,t(x,y),t(u,v)) \longrightarrow & \\
\quad t(if(carry(add(c,y,v)), add(1,x,u), add(0,x,u)), rest(add(c,y,v))) &
\end{array}$$

(The third rule takes care of "pulling out" the carry. The arguments of *add* are the old carry and the two numbers added, and it rewrites into a pair consisting of the new carry and the lower bits of the result.)

The number of parallel rewrite steps when adding two numbers of 2^n digits is linear in n, and therefore logarithmic in the length of the numbers.

4.4 Boolean expressions without variables

As an example of the use of more powerful versions of the machine for rewriting, consider Boolean expressions made up of the connectives *or*, *and*, *not*, and truth constants, *true*, *false*. The length of such an expression is the number of the above symbols appearing in it. If we consider all binary trees with n nodes as having the same probability, then the average parallel cost of evaluating such an expression by our machine using "clever" waiting (that is, when we abandon rewriting of the other argument once one argument of an *and* has been rewritten to *false* or one argument of an *or* has been rewritten to *true*) tends to a constant as n grows large. (For the proof, based on the generating functions approach of Flajolet, see [DerLin 89].) It should be remarked, however, that with a "clever" sequential evaluation (that is, when we do not evaluate the second argument if evaluation of the first argument is sufficient for evaluation of the whole expression), we get a similar result, only with a larger constant.

4.5 Search trees and skew heaps

This example shows how the rewriting approach can simplify parallelizing algorithms. The example of skew heaps is similar to the simpler example of insertion of elements into a binary search tree given in [GoKiMe 86]. Its interest lies in showing that, while the basic operations of our concurrent machine are quite complicated, it is much simpler to parallelize algorithms with it, because the machine gets the problem in a form that preserves its "meaning".

Concurrent operations on skew heaps were treated in [Jon 89]. There, an implementation of priority queues is considered, which allows items to be enqueued and dequeued in logarithmic amortized time, but allows new operations to begin after only constant time on a MIMD machine. The implementation uses the skew heaps of [SleTar 86]. Enqueuing and deleting items from the heap is done with the help of the *meld* operation which unites two heaps. In [Jon 89], a sequential Pascal program is first given, which achieves the melding by means of pointer adjustments, and then this program is transformed into a concurrent program which uses semaphores to implement what is called there "the bubble of mutual exclusion". We can formulate the melding of two heaps, represented as binary trees of the form $t(Left, Root, Right)$, by means of the following rewrite rules:

$$\begin{aligned}
&if(true,x,y) \longrightarrow x\\
&if(false,x,y) \longrightarrow y\\
&meld(t(x,y,z),nil) \longrightarrow t(x,y,z)\\
&meld(nil,t(x,y,z)) \longrightarrow t(x,y,z)\\
&meld(t(x,y,z),t(x',y',z')) \longrightarrow if(y \le y',\\
&\qquad t(meld(z,t(x',y',z')),y,x),t(meld(t(x,y,z),z'),y',x'))
\end{aligned}$$

(We assume a condition $y \le y'$ is immediately rewritten to *true* or *false*, as the case may be.) The strategy is the following: A process with main operator *meld* tries to rewrite its term at the root. If this is impossible, it creates children and waits for their partially evaluated answers. It immediately rewrites its term at the root when this becomes possible. In that case, its main operator becomes a constructor, so no rule will ever apply to it at the root. If it used the last rule, it creates a child for the argument with main operator *meld*. In all cases, it terminates after reporting its result (possibly only partially evaluated, but with suitably connected answer channels) to its parent.

In our case, the primitive machine operations are much more complicated than in the implementation via pointer adjustments and semaphores. On the other hand, things are much simpler conceptually, since the idea of the algorithm is not lost by translating it to the machine. The "bubble of mutual exclusion" is realized by itself—the machine will not try to rewrite the term $meld(meld(H_1, H_2), H_3)$ at the root simply because no rewrite rule applies at the root.

4.6 Satisfying Boolean expressions

As an example of narrowing, suppose we have Boolean expressions made up of the operators *and*, *or*, *not*, truth values, and variables, with the rewrite rules

$$\begin{array}{rcl@{\qquad}rcl}
and(true, x) & \longrightarrow & x & and(x, true) & \longrightarrow & x \\
and(false, x) & \longrightarrow & false & and(x, false) & \longrightarrow & false \\
or(true, x) & \longrightarrow & true & or(x, true) & \longrightarrow & true \\
or(false, x) & \longrightarrow & x & or(x, false) & \longrightarrow & x \\
not(true) & \longrightarrow & false & not(false) & \longrightarrow & true \\
not(and(x, y)) & \longrightarrow & or(not(x), not(y)) & not(or(x, y)) & \longrightarrow & and(not(x), not(y))
\end{array}$$

Given an expression, we want to find a substitution for its variables so that the expression rewrites to *true*. (This is the satisfiability problem.)

When we rewrite the expression, the *not*'s are pushed towards the leaves, so that we reach a normal form in which the argument of each *not* is a variable.

Now we can apply different narrowing strategies for ordering the narrowing instatiations of variables:

- Goal-directed strategy—we know that $and(s, t)$ rewrites to *true* iff both s and t rewrite to *true*. We also know that for $or(s, t)$ to rewrite to true it is sufficient if one of its arguments does. Now suppose that, after the preliminary rewriting has pushed the *not*'s inward, the expression considered as a tree contains an or-chain, that is, a path from the root to a leaf or a *not* preceding a leaf all of whose nodes are labelled by *or*'s. Then we have an immediate solution. If we have an and-chain, we get a necessary substitution, which, after rewriting, results in a smaller tree.

- Substitution for variables that occur near to the root.

- Substitution for variables that appear many times.

- A combination of the two previous strategies, which gives to each variable a priority which consists of the sum of the weights of its appearances, where the weight of an appearance at distance d from the root is $1/2^d$.

It turns out that the probability for an or-chain, if we consider trees with n nodes of degree 2 and give equal probability to families of trees that have the same structure as far as the nodes of degree 2 are concerned, approaches approximately 0.42 as $n \longrightarrow \infty$. (This can be shown by the generating functions method we used in [DerLin 89].) The case of an and-chain is equally probable.

Acknowledgements

We are grateful to Ehud Shapiro and the anonymous referees for their helpful suggestions.

References

[DerJou 90] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems", Chap. 6 in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, Vol. 2, North-Holland, 1990.

[DerLin 89] N. Dershowitz and N. Lindenstrauss, "Average time analyses related to logic programming", *Logic Programming: Proceedings of the 6th International Conference*, MIT Press, pp. 369–381, 1989.

[DerLin 90] N. Dershowitz and N. Lindenstrauss, "A parallel implementation of equational programming", *Proceedings of the 5th Jerusalem Conference on Information Technology*, Oct. 1990, to appear.

[DerPl 88] N. Dershowitz and D. A. Plaisted, "Equational programming", in J. E. Hayes, D. Michie, and J. Richards, eds., *Machine Intelligence 11*, pp. 21–56, 1988.

[GoKiMe 86] J. Goguen, C. Kirchner, and J. Meseguer, "Concurrent term rewriting as a model of computation", *Proceedings of a Workshop on Graph Reduction*, Santa Fé, NM, Lecture Notes in Computer Science **279**, Springer, Berlin, pp. 53–93, 1986.

[JoDer 89] N. A. Josephson and N. Dershowitz, "An implementation of narrowing", *Journal of Logic Programming* **6**(1&2), pp. 57–77, 1989.

[Küch 82] W. Küchlin, "Some reduction strategies for algebraic term rewriting", *ACM SIGSAM Bull.* **16**(4), pp. 13–23, 1982.

[Jon 89] D. W. Jones, "Concurrent operations on priority queues", *Communications of the ACM* **32**, pp. 132–137, 1989.

[LiPal 89] K. Li, "Complexity of Term Rewriting", Technical Report 464, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY, November 1989.

[Shap 87] Shapiro, E., "Systolic programming: A paradigm of parallel processing", in E. Shapiro, ed., *Concurrent Prolog Collected Papers*, Vol. 1, pp. 207–242, 1987.

[SePaRa 89] R. C. Sekar, S. Pagawi, and I. V. Ramakrishnan, "Transforming strongly sequential rewrite systems with constructors for efficient parallel execution", *Rewriting Techniques and Applications: 3rd International Conference*, Lecture Notes in Computer Science **355**, pp. 404–418, 1989.

[SleTar 86] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps", *SIAM J. Comput.* **15**, pp. 52–69, 1986.

[Stick 83] M. E. Stickel, "A note on leftmost innermost term reduction", *ACM SIGSAM Bull.* **17**(3&4), pp. 19–20, 1983.

[TamSa 83] H. Tamaki and T. Sato, "Program Transformation Through Meta-shifting", *New Generation Computing* **1**, pp. 93–98, 1983.