

AD-A171 009

A REWRITE RULE MACHINE MODELS OF COMPUTATION FOR THE
REWRITE RULE MACHINE(U) SRI INTERNATIONAL MENLO PARK CA
COMPUTER SCIENCE LAB J GOGUEN ET AL. JUL 86

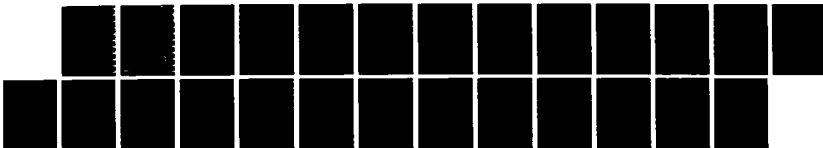
1/1

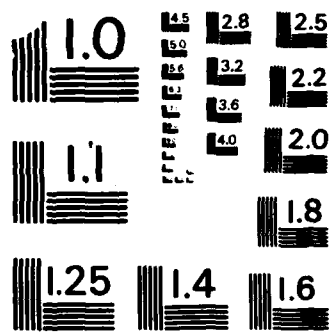
UNCLASSIFIED

N00014-85-C-0417

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A171 089

SRI International



DTIC FILE COPY

A REWRITE RULE MACHINE

Models of Computation for the Rewrite Rule Machine

Final Report
July 1986

By: Joseph Goguen, Program Manager
Claude Kirchner, International Fellow
José Meseguer, Senior Computer Scientist

SRI Project ECU 1243

Prepared for:

Office of Naval Research
Information Sciences Division
800 N. Quincy St.
Arlington, VA 22217-5000

Attn: Dr. Charles Holland, Code 1133

Contract No. N00014-85-^C~~B~~-0417

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
(415) 328-6200
TWX: 9100-373-2046
Telex: 334486

333 Ravenswood Ave • Menlo Park, CA 94025
415 326-6200 • TWX 910-373-2046 • Telex 334-486

9

3

DTIC
ELECTE
AUG 14 1986
S D E

86 7 29 124

Models of Computation for The Rewrite Rule Machine *

Joseph Goguen, Claude Kirchner[†], José Meseguer

SRI International, Computer Science Lab

July 9, 1986

Abstract: A new model of computation, *concurrent tree rewriting*, is proposed as a bridge between easily programmed Ultra High Level Languages (UHLLs) featuring implicit concurrency, and an advanced parallel architecture of unprecedented performance, the Rewrite Rule Machine (RRM) architecture. At the highest level of abstraction, computation is understood as rewriting a tree at multiple sites concurrently. Less abstractly, such a (possibly very large) tree can be partitioned into fragments that are assigned to different processors, with each processor doing concurrent rewriting on its own fragment of the tree; this gives the second level, *partitioned concurrent rewriting*. After introducing the basic concepts and properties of the model, we discuss tradeoffs between tree and directed acyclic graph (dag) data representations; we also study partitioned concurrent rewriting, including tree and rule partitioning, and discuss evaluation strategies as a flexible control mechanism for concurrent rewriting. The mathematical definitions are gathered in an appendix.

1 Introduction

This report documents recent progress on models of computation for the Rewrite Rule Machine (RRM) project at SRI International. The next paragraphs of the introduction discuss the overall goals of the RRM project, software issues that our model solves, and the model of computation. Section 2 studies the most abstract level of concurrent rewriting, and Section 4 treats partitioned tree rewriting. Tradeoffs between the tree and directed acyclic graph representations are discussed in Section 3, and rewriting strategies in Section 5. For an overview of the RRM project, the reader is referred to [5]. Results on simulation are reported in [13]. The architectural design, including designs for tree representation and rewriting inside each processor, is documented in [11]

*Supported by Office of Naval Research Contract N00014-85-C-0417.

[†]On leave from CNRS (Centre de Recherche en Informatique de Nancy) France.



On For	
A&I	<input checked="" type="checkbox"/>
ed	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special

A-1

1.1 Goals of the RRM Project

The purpose of the RRM project is to design and test a prototype general purpose, high-speed, large-scale, parallel computer architecture that is especially suitable for symbolic computation with ultra-high-level languages (UHLLs). One promising application area is the support of powerful program development environments, including "intelligent" editors, compilers, libraries of reusable software, rapid prototyping tools, formal specification languages, program verifiers, debuggers, test case generators, etc. Artificial Intelligence applications are also promising, including robot planning, natural language processing, high level vision, and expert systems. A third promising area is hardware simulation.

The RRM will consist of a large number of processors, each with custom VLSI to process tree-structured data independently and very efficiently. The UHLLs considered will combine object-oriented, functional, and logic programming, plus other powerful features, including parameterized programming, graphical programming, sophisticated error handling, and powerful type systems. Compilers will convert UHLL programs into sets of rewrite rules for execution on the RRM.

1.2 Software Issues

From the software point of view, tree processing means that manipulations can easily be described in a way that is independent of the order of execution, and that also provides ample opportunities for concurrent execution. The basic mode of tree processing is called *tree* (or *term*¹) *rewriting* (or replacement or reduction), and refers to the replacement of one subtree by another, whenever a tree-structured template is matched. A *rewrite rule* consists of two such templates, one for the subtree to be replaced, and another that determines what it is to be replaced by.

1.2.1 Programmability

We feel that *programmability* is one of the most critical issues blocking further progress in parallel computation: it does little good (except for very homogeneous problems) to provide lots of processors if the programmer has to explicitly assign processes to processors. We feel that the best approach to combining hardware efficiency with programming ease and flexibility is to have a model of computation that provides a simple bridge between a powerful UHLL and the hardware itself. We argue that *tree rewriting* is such a bridge. As shown in this report, concurrency is implicit in the rewrite rules themselves and can be directly exploited by the model and the architecture without any explicit concurrency constructs in the programming language. Work on programming language semantics shows how to implement advanced languages with tree rewriting. We have taken OBJ2 [4] as our basis. This is a very advanced functional UHLL based on tree rewriting with a uniquely powerful generic module facility and type system. This basis has been extended to include logic programming [6] and object-oriented programming [7].

¹In this presentation we will often use the word "tree" as a synonym for "term," except when discussing data representations for terms, where we will distinguish between tree and dag representations.

1.3.2 Tree-Structured Data and Computation

There are many applications in which the data are naturally tree-structured (in the sense that there is a natural hierarchy, with a "root" node at the top and with one or more branches from each node that is not a tip node), and for which tree rewriting is a highly efficient and natural mode of execution. Some examples of naturally occurring tree structures are:

- Menus, such as occur in interactive graphics,
- Expressions, such as $(A + B)(A^2 + 3)$ and, more generally, programs,
- Natural language syntax, and many other structures that occur in natural or artificial languages, such as plans and explanations,
- Most generally, any abstract data type.

In fact, tree structure is a fully general form of data structuring, since any computable function can be seen as a computation on tree-structured data that rewrites subtrees into other subtrees. In particular, such processes as selecting a particular item from a menu, evaluating an arithmetic expression, verifying a program, assigning a meaning to a sentence, editing a program, constructing a plan, and compiling or interpreting a program can all be conveniently described as tree rewriting processes. The feasibility of writing nontrivial programs with rewrite rules has been shown by experience with programming languages like OBJ, Hope, Miranda, and FP, and by the work of Hoffman and O'Donnell. For example, several different language interpreters have been written in OBJ, including one for OBJ.

1.3 Models of Computation

Previous tree rewriting studies have addressed the sequential case. In this report we develop the basic concepts for concurrent tree rewriting, i.e., the rewriting of a tree at multiple sites concurrently. This amounts to a new model of computation with some properties analogous to sequential tree rewriting and with new properties of its own. An issue of crucial importance for concurrent tree rewriting is whether to represent data as trees or as directed acyclic graphs (dags). Important efficiency and communication tradeoffs, and even more general ways of performing concurrent rewriting appear, when considering the choice of trees vs. dags. Since the RRM architecture provides a network of processors such that each can do parallel tree rewriting and several processors can cooperate in rewriting a large tree, a natural, more concrete, refinement of the model is to consider partitioned concurrent term rewriting. Important issues for partitioned rewriting include: (i) finding appropriate criteria for tree partitioning; (ii) efficiency issues related to the partitioning of the rules, since each processor does not need, and cannot store, the entire set of rewrite rules that make up a large program, and not all the rules are equally expensive; (iii) communication issues when rewriting takes place at the border separating two or more fragments of the tree. An additional problem that we address is rewriting strategies. Strategies can be used as a flexible control mechanism that, while still allowing concurrency, avoids useless computations that could take up considerable resources.

2 Concurrent Tree Rewriting

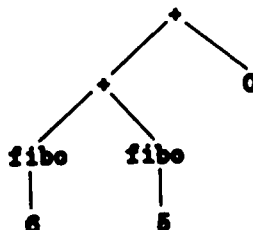
This section provides an informal introduction to concurrent tree rewriting. Although the style is informal and intuitive, the ideas introduced here can be made mathematically precise. Indeed, formal definitions, although not essential to understand the main ideas, are nevertheless important in order to develop the theory with mathematical rigor. Mathematical definitions for all the concepts introduced in this section can be found in the Appendix.

In the concurrent tree rewriting model of computation, data are trees with nodes labelled by function symbols and leaves labelled by constants, and programs are sets of *equations* that are interpreted as left to right *rewrite rules*. The left- and righthand sides of an equation are trees with nodes labelled by function symbols and leaves labelled by constants or *variables*. A variable can be instantiated by any tree of the type given to the variable; such an instantiation of a set of variables is called a *substitution* or *match*.

Computation is *tree rewriting* (or *reduction*) by *matching* the lefthand side of a rewrite rule to a subtree of the tree to be evaluated and then *replacing* that subtree by a corresponding instance of the righthand side of the rule. For example, by instantiating the variable N to the constant 6 , the lefthand side of the rewrite rule

$$(*) \text{ fibo}(N) \rightarrow \text{fibo}(N - 1) + \text{fibo}(N - 2)$$

(where *fibo* is the function symbol for Fibonacci numbers) can be matched to $\text{fibo}(6)$ in the term $(\text{fibo}(6) + \text{fibo}(5)) + 0$ which can be also represented by the tree

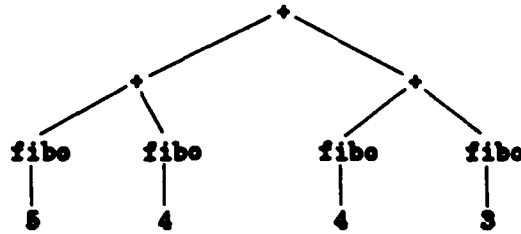


The subterm $\text{fibo}(6)$ where the match has occurred is called a *redex*. The redex subterm can then be replaced by the corresponding instance of the righthand side, so that the original term is rewritten to the term $((\text{fibo}(6-1) + \text{fibo}(6-2)) + \text{fibo}(5)) + 0$. Rewriting by applying one rule at any one location at a time is called *sequential tree rewriting*. If the rewrite rule $(*)$ had been applied to $\text{fibo}(5)$ instead, one step of sequential rewriting would have yielded $(\text{fibo}(6) + (\text{fibo}(5-1) + \text{fibo}(5-2))) + 0$.

Notice that the rule $(*)$ could instead have been applied concurrently to both $\text{fibo}(6)$ and $\text{fibo}(5)$, yielding

$$((\text{fibo}(6-1) + \text{fibo}(6-2)) + (\text{fibo}(5-1) + \text{fibo}(5-2))) + 0.$$

We call this *concurrent tree rewriting*. In concurrent tree rewriting several rules can simultaneously be applied and several matches for each one of those rules can be rewritten in a single step of concurrent computation. For example, by applying the rule $(*)$ concurrently in a first step, and then applying both the rule $N + 0 \rightarrow N$ and a rule for subtraction in a second concurrent step, we can transform the original tree into the tree

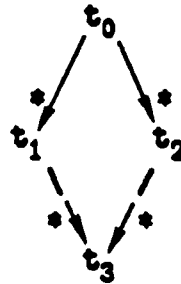


in two steps of concurrent rewriting. This process continues until there are no more matches, then the expression is said to be *reduced* or in *normal form*. This simple example shows that tree rewriting is by its very nature concurrent. We emphasize that the concurrency is implicit in the rewrite rules themselves, and *no explicit concurrency constructs are required in the language*. We see this as a major advantage.

A set of rules is called *terminating* if all possible ways of concurrently rewriting a term do eventually stop in a normal form (the normal form reached in each case may in general be different). Some equations are nonterminating and should not be used as rewrite rules; for instance, a commutativity law $X + Y = Y + X$ would lead to the infinite chain of rewritings

$$\text{fibo}(6) + \text{fibo}(5) \rightarrow \text{fibo}(5) + \text{fibo}(6) \rightarrow \text{fibo}(6) + \text{fibo}(5) \rightarrow \dots$$

For functional computations, one expects the final result not to depend on the particular chain of rewritings that led to it, i.e., one expects all normal forms to be equal. This property is guaranteed to hold if the rules satisfy the *Church-Rosser property* illustrated by the diagram below



where the starred arrows denote rewriting sequences of 0, 1, or more steps of (possibly concurrent) rewriting. The Church-Rosser property says that any two rewriting sequences starting at the same term (solid arrows) can always be reconciled by further rewriting (dotted arrows). For nondeterministic computations, however, it is sensible to allow reaching different final results, and the Church-Rosser property should not be expected to hold.

If a lefthand (resp. righthand) side of a rule has only one instance of each of its variables, the rule is called *left-* (resp. *right-*) *linear*. The rule (*) above is an example of a left-linear rule. Left-linear rules are easier to match than non-left-linear ones, since there is no need to check that different occurrences of a variable are instantiated to the same subterm for left-linear rules.

When doing concurrent rewriting, care has to be taken of the case when two lefthand sides match in two redexes so that the two lefthand sides partially overlap each other.

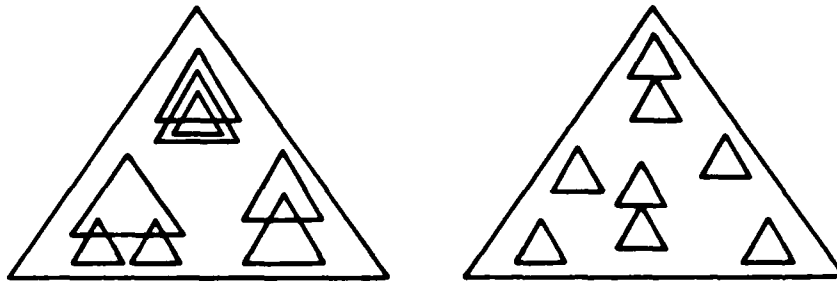


Figure 1: Overlapping and nonoverlapping sets of redexes

Rules for which this happens are called *overlapping*, or *self-overlapping* if overlapping occurs with the same rule. For instance, the associativity rule

$$(X + Y) + Z \rightarrow X + (Y + Z)$$

overlaps with itself, so that it has the entire expression $((5 + 7) + 9) + 7$ and the subterm $(5 + 7) + 9$ as redexes. This poses a problem for the well-definedness of concurrent rewriting because there is a "clash" of redexes due to overlapping; this is illustrated in Figure 1. However, later in this report we shall see that such clashes can be tolerated if a dag representation is chosen for terms.

3 Trees versus Dags

An important question is how to represent trees at the hardware level. An important choice is whether or not to use dags (i.e., directed acyclic graphs), which permit sharing of identical subtrees. We have considered the advantages and disadvantages of dag versus strict tree structure for abstract concurrent tree rewriting and for partitioned tree rewriting².

The problem of choosing between a tree representation and a dag representation for terms is not a new one. Several studies on dag rewriting already exist, including work such as [1] and [3]. But, to the best of our knowledge, none of the previous studies deals with concurrent rewriting or has architectural concerns in mind.

3.1 Trees and Dags

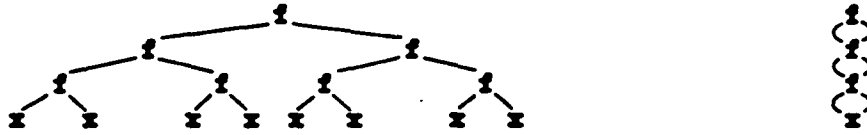
We have already discussed and given examples of tree representations. The dag representation generalizes this. Notice that a tree is a particular kind of directed acyclic graph, or dag, having a unique entering node (i.e. a unique node which is not the target of any edge) and such that each other node is in the target of exactly one edge. Each dag with nodes labelled by function symbols and having

²The tree vs. dag question is very important at different levels of modelling, with different tradeoffs appearing at each level. Simulation results will be used to decide at what levels trees or dags should be chosen.

an entering node from which all other nodes are reachable can be transformed into a (labelled) tree by successively splitting the nodes that are the targets of several arrows. We then view all the dags that can be transformed into the same tree as different representations of the same term. For instance, the term

$$f(f(f(x, x), f(x, x)), f(f(x, x), f(x, x)))$$

has, among other, the following two representations:



3.2 Comparison of the Two Representations

3.2.1 Space Occupation

Obviously the dag representation is more space efficient than the tree representation, which had more nodes. There is always a dag representation of maximum space efficiency, i.e., having a minimum number of nodes, called a **fully shared dag representation**. The dag on the right in the example in section 3.1 is fully shared. For that example, the space (number of nodes) occupied by the tree was $2^4 - 1$ as opposed to 4 for the dag representation. Thus, the space of the tree representation could be in an exponential relation to the space of the dag representation. In general, however, a term will not be sufficiently regular to allow an exponential gain in space by full sharing. Statistics on the space efficiency of the dag representation for a collection of examples are given in [13].

3.2.2 Criteria Related to Matching

In order to detect a match, one may have to do either of the following:

test for equality of subterms. For example, to match the non-left-linear rule $x + x \rightarrow z$ equality of the two subterms matching the variable x needs to be checked for the tree representation. In a fully shared dag representation checking for such an equality is trivial. Another example is given later (see Figure 3).

handle simultaneous read accesses to the same node. For example, matching the rule $f(h(x), h(g(y))) \rightarrow q(x, y)$ to the dag



will involve a simultaneous read access to the node labelled h . Such a concurrent read must be supported by the hardware in all cases for dag representations, even for nonoverlapping rules (note that the rule in the example is not self-overlapping). For the tree representation, simultaneous read access to a node can happen only when matching overlapping rules.

The first point is clearly an advantage of the dag representation and especially of the fully shared dag representation, where the test of equality is equivalent to the test of identity. But since rewriting of fully shared dags cannot be implemented efficiently, the gain will be only partial. Nevertheless, a successful test for equality can, in the case of a dag representation, be used to increase the amount of sharing, thus freeing storage resources.

The second point shows a partial advantage of the tree representation since, even if this kind of simultaneous read access may appear for trees in the case of self-overlapping rules, it will be much less frequent than for dags.

3.2.3 Criteria Related to Tree Replacement

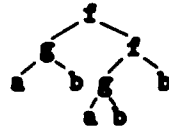
We now consider the problem from the point of view of tree replacement. There are again several aspects:

number of copies needed. Many rules, such as the distributive rule

$$x * (y + z) \rightarrow (x * y) + (x * z)$$

are not right-linear. For dags, duplication of a variable by rewriting involves only modification of a pointer, but for trees one has to copy the duplicated subtrees, which can be very large.

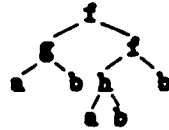
direct modification of node labels. Such a modification is correct in the case of trees but not in the case of dags (except for the node corresponding to the top of the redex). For instance, rewriting the tree representation



by means of the rewrite rule



can be accomplished just by modifying the node at occurrence 2.1, to obtain



whereas for the dag representation



a local copy is required. The result is:

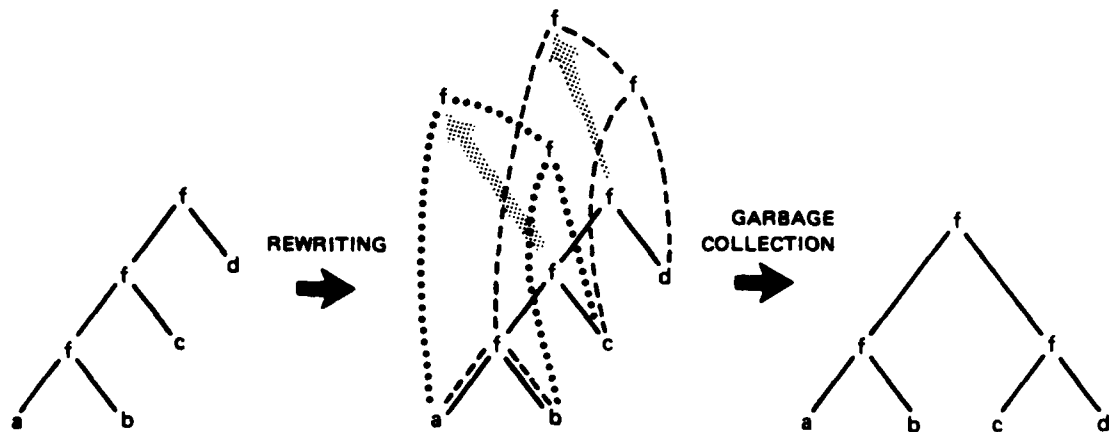
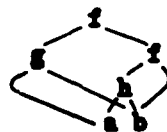


Figure 2: Concurrent dag rewriting with the associativity rule.



overlapping of rules. By implementing a graph rewriting which modifies only the node at the top of the redex, one can rewrite concurrently without worrying about self-overlapping rules. A natural example using the associativity rule $((xfy)fz) \rightarrow (xf(yfz))$ is described in Figure 2. Notice that this kind of rewriting with self-overlapping rules makes no sense for terms or trees.

3.2.4 Other Considerations

Other issues are also affected by choice of representation:

1. Freeing unreferenced items will be simpler for the tree representation, since something like reference counts will be needed for the dag representation.
2. Simultaneous attempts to read a value needed for finding a match or for testing equality of subterms, may induce delays in the case of the dag representation. However, that problem may be solved at the hardware level.

Figure 3 gives examples that illustrate the different criteria of comparison that we have discussed between tree and dag representations of a term.

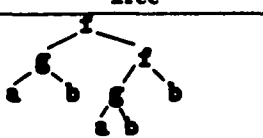
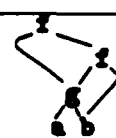
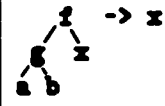




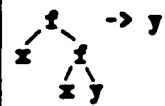
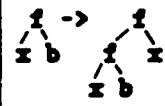
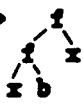
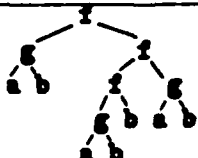

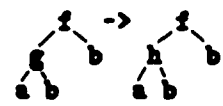
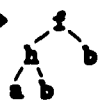

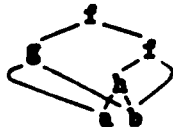
	Tree	Dag
		
 $\rightarrow x$	The normal form b is reached in one step of concurrent rewriting.	Depending on how the reduction of dags is implemented, the normal form (b) is reached in one (resp. two) step(s) of concurrent rewriting. The second step would be needed if simultaneous read access to a node is sequentialized. In this case we obtain first  and then b .
 $\rightarrow c$	The normal form  is reached by one step of concurrent rewriting at occurrences 1 and 2.1.	The normal form  is reached by one step of concurrent rewriting at occurrence 2.1.
 $\rightarrow y$	One needs to test equality of the subtrees at occurrences 1 and 2.1. The result b is obtained in one step of concurrent rewriting at occurrence ϵ .	The same result is obtained with the same rewriting but the equality test is trivial
 \rightarrow 		
 \rightarrow 	By only modifying the cell at occurrence 2.1, one obtains 	The previous trick can not be used and local copying is required. The result is 

Figure 3: Impact of the term representation on rewriting

3.2.5 Summary

The following table summarizes the points we have discussed.

	Tree	Dag
Advantages	<ul style="list-style-type: none"> • There is no shared structure and thus no overhead due to multiple read access to a node. 	<ul style="list-style-type: none"> • Allows (possibly maximal) sharing and thus is more space-efficient. • Testing equality of subtrees is generally efficient and is trivial in the case of maximal sharing. • No copying is needed when subtrees are duplicated. • There is no need to avoid overlapping in concurrent rewriting.
Drawbacks	<ul style="list-style-type: none"> • Testing equality of subtrees is expensive. • Copying of duplicated subtrees is needed, which can be expensive for large trees. • Overlapping redexes require special treatment. 	<ul style="list-style-type: none"> • Rewriting may require local duplication in the dag.

4 Partitioned Term Rewriting

In order to be able to execute large programs on the RRM, both the terms to be reduced and the set of rules constituting the program have to be partitioned:

- Terms should be partitioned among different processors because:
 - The size of each processor is limited,
 - Each processor is working in a SIMD mode and in general the terms are not homogeneous (in the sense that the function symbols appearing in different parts of the tree may be quite different) so that the potential for parallelism cannot be exploited if the tree is not partitioned.
- The set of rules is partitioned because:
 - The size of each processor's rule memory is limited,
 - For efficiency purposes it is appropriate:
 - * To increase the number of successful matches by flow analysis, which allows localizing the set of rules that can possibly apply on a given term,
 - * To isolate as much as possible the rules involving node or processor communication such as:
 - Non-left-linear rules which require testing for equality of sub-terms,
 - Overlapping rules, which require mutual exclusion of matching in a given neighborhood.

All these points are developed and discussed below.

4.1 Partitioning of Rules

4.1.1 Stratification by Rule Complexity

The complexity that we have in mind regards difficulty in matching a rule. The concurrent matching process is simplest for sets of left-linear nonoverlapping rules; it becomes more complex with the presence of non-left-linear or self-overlapping rules. The worst case is a rule that is not left-linear and, in addition, overlaps with itself.

A left-linear rule is less complex than a nonlinear one. For instance, rule (0) in the rational numbers example (Figure 4) is the only left-linear rule. To decide whether rule (0) matches the term

$$(\dagger) (3 / (2 / (3 * 7))) / (3 / (2 / (5 * 7)))$$

at the top, it is not enough to see that the two subterms below the top $/$ symbol are nonzero rationals; one has to check for equality of those two subterms, namely $3 / (2 / (3 * 7))$ and $3 / (2 / (5 * 7))$. Left-linear rules are simpler because matching only requires local inspection of the tree in a region the size of the rule's lefthand side. For instance, rule (1) matches at the top by instantiating R to $3 / (2 / (3 * 7))$, R' to 3 , and S' to $2 / (5 * 7)$; no further inspection of those three subterms is required.

Self-overlapping rules are harder to match in parallel than rules that do not overlap with themselves. This is because two matches of the same rule can overlap with each other imposing an additional communication overhead to arbitrate such conflicts. For instance, rule (1) below overlaps with itself and also matches the left subterm of the term (\dagger) above by instantiating R to 3 , R' to 2 , and S' to $3 * 7$. Rule (1) also matches the right subterm of (\dagger) in a completely similar way. Thus, attempting to match rule (1) in parallel to the term (\dagger) will require communication to resolve the contention between those three overlapping matches. An even worse case appears when the rule is both non-left-linear and self-overlapping; then, both communication costs (for deciding equality of subterms and for resolving contention of overlapping matches) have to be paid. For instance, the rule

$$R' * ((1 / R') * R) = R$$

is an example of a non-left-linear, self-overlapping rule.

Given a set of rules, we may want to stratify the set according to rule complexity and then partition each stratum into maximal sets of nonoverlapping rules whenever possible. In this way, we can obtain a partition of the set of rules into subsets that is optimal from the efficiency point of view. More efficient rules could be tried first and less efficient rules could be isolated and adequately postponed. Thus, sets of rules can be organized from more to less efficient according to the following categories:

1. Sets of left-linear, nonoverlapping rules
2. Sets of non-left-linear, nonoverlapping rules
3. Sets of left-linear self-overlapping rules
4. Sets of non-left-linear self-overlapping rules.

```

obj RAT is
  protecting INT .
  sorts NzRat Rat .
  subsorts Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op _- : Rat -> Rat .
  op _- : NzRat -> NzRat .
  op _+ : Rat Rat -> Rat [assoc comm] .
  op _+ : NzRat NzRat -> NzRat [assoc comm] .
  vars R S : Rat .
  vars R' S' I' : NzRat .
  eq : R' / R' = 1 . -- 0
  eq : R / (R' / S') = (R * S') / R' . -- 1
  eq : (R / R') / S' = R / (R' * S') . -- 2
  ceq : J' / I' = quot(J',gcd(J',I')) / quot(I',gcd(J',I'))
        if gcd(J',I') /= 1 . -- 3
  eq : R / 1 = R . -- 4
  eq : 0 / R' = 0 . -- 5
  eq : R / (- R') = (- R) / R' . -- 6
  eq : - (R / R') = (- R) / R' . -- 7
  eq : R + (S / R') = ((R * R') + S) / R' . -- 8
  eq : R + (S / R') = (R * S) / R' . -- 9
jbo

```

Figure 4: The rational numbers example

In the rational numbers example (Figure 4), the only non-left-linear rule is rule (0). The overlap between rules is summarized in the table below:

Rule	Overlaps with
0	1-9
1	0-9
2	0-9
3	0-2, 4, 6-9
4	0-3, 5, 7-9
5	1-2, 4, 6-9
6	0-3, 5, 7-9
7-9	0-6

We can then form the following stratified partition of the set of rules 0-9:

1. Non-overlapping and left-linear: {7,8,9}, {3,5}, {6,4}
2. Non-left-linear and nonoverlapping: {0}
3. Self-overlapping and left-linear: {1,2}

4.1.2 Rule Restriction by Flow Analysis

This is a method to reduce the amount of rules to be tried for concurrent matching. It can be very useful as a way of maximizing the number of successful matches when reducing a given tree. It can also help using the rule storage resources associated with each processor in an efficient way, minimizing rule communication cost. Additionally, it may even provide a useful heuristic for tree partitioning, since by clustering of the flow graph, occurrence of critical function symbols that mark transitions between function clusters could be identified. The method should be seen as complementary to rule stratification by complexity; combining the two methods together one obtains a stratification of rules by complexity such that only those rules that could potentially apply to a given problem are represented.

The key idea is to group together rules having the same function symbol at the top of their lefthand side, and to relate such sets of rules by flow analysis of their corresponding function symbols. Here are two simple notions for flow analysis in this context:

Definition 1 *A function symbol f is said to weakly flow into a function symbol g if there is a rule with f at the top of its lefthand side such that g occurs somewhere in its righthand side; if, in addition, g does not occur in the lefthand side of the rule, then f is said to strongly flow into g , i.e., strong flow is a subrelation of weak flow.*

The diagram in Figure 5 gives the flow graph for the equations of the rational numbers example. Notice that `gcd` and `quot` are integer functions; i.e., the flow graph cuts across different OBJ modules.

We can use such a flow graph at compile time to determine what rules will be needed to reduce a given tree. For instance, by flow analysis of the rules for the rational numbers, a compiler could determine that only the rules {0,1,2,3,4,9} and the rules for `gcd` and `quot` will be needed to reduce the term

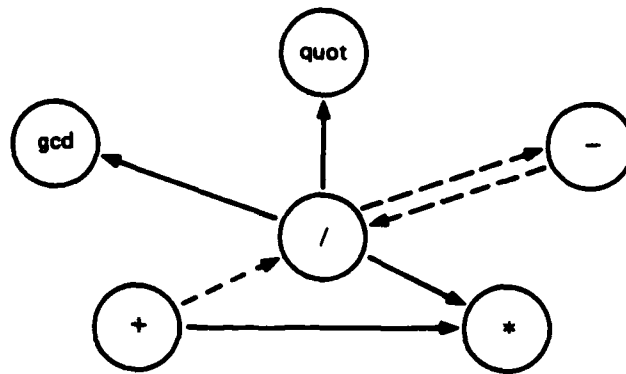


Figure 5: Flow graph for the equations of the rational numbers example

$$(3 / 6) / (7 * (16 / 12)).$$

Thus, to reduce such a term we can restrict the original partition by rule complexity for the rational numbers example to obtain a smaller partition:

1. Nonoverlapping and left-linear: {3}, {4}, {9}
2. Non-left-linear and nonoverlapping: {0}
3. Self-overlapping and left-linear: {1,2}

Restricting the set of rules by means of flow analysis, as in the above example, has two obvious advantages:

- The rate of successful matches will increase, since rules that will always fail are excluded,
- The storage of rules in a processor is facilitated, since fewer rules have to be stored.

4.2 Tree Partition

Each of the processors of the RRM has a limited capacity, so that terms exceeding a certain size cannot be stored in a single processor. This means that, when trees get too big, they have to be partitioned so that some upper fragment of the tree remains in the original processor whereas subtrees below that fragment are shipped to other available processors. Besides being a need imposed by a processor's storage capacity, partitioning of a tree may in fact be advantageous to increase the amount of concurrent rewriting. This is due to the possibly non homogeneous structure of a large tree, so that portions of the tree that are distant from each other may involve very different function symbols. Assuming that each processor will do concurrent rewriting in a SIMD mode, lack of homogeneity would limit parallelism, since match attempts for a rule can succeed only in some fragment of the tree. If trees are partitioned into relatively homogeneous parts, the amount of concurrent rewriting

can increase, since now all fragments of the tree can be active doing concurrent rewriting with the rules appropriate for each fragment.

We shall address two issues that arise in tree partitioning. The first regards criteria that should be used to partition a tree, i.e., when and where should a tree be partitioned; the second has to do with communication and reconfiguration problems when term rewriting takes place across a partition boundary, thus involving several processors.

4.2.1 Criteria for Tree Partitioning

Trees should not be partitioned at random; rather, the criteria used should be to try to maximize parallelism and to minimize communication between the fragments of the partition (i.e., interprocessor communication, since each fragment will be stored inside a different processor). Regarding tree size, there should be a certain size threshold, related to the maximal storage capacity of a processor, so that tree partitioning begins after that threshold has been reached, if other considerations do not force it before then. In addition to size, the following factors should also be taken into account in a tree partitioning strategy:

- *Expected rate of tree growth associated to a subterm.* This can be guessed by inspection of both the subterm and the equations associated to its function symbols (more generally, equations of other function symbols closely related in the flow analysis graph to those in the term) (cf. Section 4.1.2). Information on this matter could also be the subject of annotations given by the programmer as for strategies (cf. Section 5). When a subterm with high growth rate appears in the tree, such a subterm could be sent to another processor, especially if the original tree already exceeds a certain size.
- *Flow Analysis Information* The function symbol flow graph introduced in Section 4.1.2 may be used to detect transitions to a different "homogeneous component" of the tree, for which rules different than the ones used so far will apply. Function symbols that show strong flow relations with each other could be grouped together into clusters, with each cluster corresponding to a different "homogeneous component" of the tree. If a subterm marks the transition to a different homogeneous component, such a subterm, together with the rules for the new component, could be sent to a new processor. However, such transitions may be hard to detect if the clustering of function symbols does not provide sufficient separation between homogeneous components, and more experience with the flow analysis technique described in Section 4.1.2 will be needed to assess its potential for tree partitioning.

As with other issues in this report, design of a tree partitioning strategy that takes advantage of the above factors should be based on experimental results with an ample collection of examples. The approach to simulation described in [13] should be extended to partitioned tree rewriting in order to provide the necessary experimental basis.

4.2.3 Term Rewriting Across Fragments of a Partition

Trees should be partitioned so as to minimize interprocessor communication. However, interprocessor communication may be unavoidable, due to matching attempts that need to inspect a portion of the tree in a boundary between two or more fragments stored in different processors. Blocking such an attempt could result in failure to attain the final result of a computation. Two related questions arise in this context:

1. How should matching attempts across a boundary between tree fragments be handled?
2. How should the tree be reconfigured after a successful match across a boundary?

Regarding the first question, two alternatives that can be considered are: (i) to ship a fragment of the tree up, to the parent processor requesting the match, and (ii) to ask the child processor to do part of the match. The second alternative seems preferable, since match attempts fail most of the time and may require unbounded inspection of the child subtree for non-left-linear rules. If the first alternative is chosen, unnecessary communication cost will be incurred in many cases when a match did not exist. Also, shipping a subtree up will generally increase the number of links between processors, since a link to the top of the child subterm would, in the first alternative, have to be replaced by links to all the children trees of the tree been shipped up, thus increasing the amount of interprocessor communication.

The configuration question can be considered assuming a tree representation or a dag representation. In the following we will discuss an example for the dag case; the tree case is similar, but the reader should be aware of the limitations of the tree representation discussed in Section 3.

The example is illustrated in Figure 6. The rule $f(g(x), y) \rightarrow k(r(x, x), h(y))$ will match across trees 2-4. Since tree 3 originally had two links from two parent trees, the function symbol g has to be recopied after rewriting so that the link with tree 1 remains consistent. How should the righthand side be partitioned between trees 2 and 4? One option is to insert it exclusively within tree 2; this, however, would unnecessarily increase by one the number of links between trees. A better solution that would, on the average, tend to balance the amount of tree growth after a rewriting on a boundary would be to partition the righthand side bringing down as much of it as possible. This leads to the solution in Figure 6, and (in this case) avoids increasing the number of links. In general, the amount of space available in each processor may dictate a different strategy for reconfiguring the tree by insertion of the righthand side.

5 Strategies

For most computations, the order of evaluation does not affect the final result. This informal fact has a formal counterpart in the *Church-Rosser* property of a set of rewriting rules guaranteeing that different evaluations of the same tree can always be reconciled by further evaluation. The Church-Rosser property holds for concurrent rewriting if and only if it holds for sequential rewriting, and indeed this property,

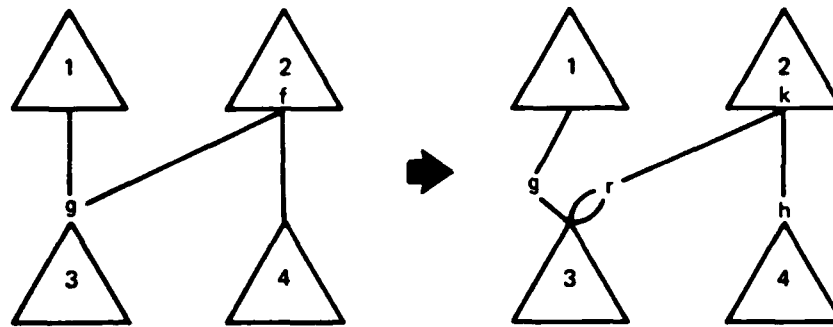


Figure 6: Example of reconfiguration of a tree after rewriting

since it allows rewritings to be done in any order, ensures the fully concurrent nature of tree rewriting. However, there are reasons that may make advisable imposing certain control mechanisms, called evaluation strategies, on the order of evaluation, although evaluation itself remains concurrent. These reasons include:

- Space efficiency, since certain rewritings may perform unnecessary computations that highly increase the size of the tree.
- Termination of computations that in general may not necessarily terminate but where the order of evaluation matters for finding a final result when there is one.
- Concurrency control purposes, when term rewriting is used as a method to implement communication among processes.

The control mechanism that we have been exploring is that of a tree rewriting strategy. The notion of *E-strategy* (*E* is for *evaluation*) was introduced in sequential OBJ2 [4] as a powerful and flexible way to control the order of evaluation, so as to improve efficiency of execution. For example, given a three-argument operator f with strategy [1 2 0 3] and an expression $f(t_1, t_2, t_3)$ to be reduced, the first argument t_1 (indicated by the number 1) must be reduced first, say to t'_1 , before reducing the second argument t_2 to t'_2 (indicated by the number 2); then we must rewrite at the top (indicated by the number 0) of $f(t'_1, t'_2, t_3)$ before finally going on to reduce t_3 . This kind of sequential evaluation is not appropriate for concurrent rewriting; fortunately, however, the interpretation of *E-strategies* can be generalized from the sequential case to the concurrent case. For the example given above, we would begin by evaluating all three major subtrees of $f(t_1, t_2, t_3)$ concurrently, until its first and second subtrees are reduced; then we would apply rules to the top of the resulting tree before going on to reduce the third argument further (if needed). More generally concurrent *E-strategies* can be provided in the three following ways, listed in increasing order of amount of control being imposed on the computation:

Concurrency with priority. It is the strategy described previously: concurrency is not affected until the arguments having priority are normalized, and then

one action (in the previous example the reduction on top) is given the highest priority.

Concurrency with rendez-vous. Reduction is executed concurrently but some subterms must all be in normal form before reduction at the top is performed. For example if the operator f has the strategy [123|0] then no reduction on top of the tree can be done before t_1 and t_2 and t_3 are all normalized.

Exclusivity. That is the strategy which assures the exclusivity of reduction at one occurrence. For instance, if the operator f has the strategy [!1!2!3!0] it means that the first argument t_1 must be first reduced before reductions in the others subtrees t_2, t_3 and on top are permitted. Then t_2 itself is exclusively reduced, next t_3 , and at last the resulting tree $f(t'_1, t'_2, t'_3)$ itself is reduced.

We have found that such concurrent rewriting strategies can yield significant savings of space without reducing the amount of useful concurrency.

More generally, it appears that OBJ with E-strategies can be used to specify quite general concurrent processes, for example, protocols. If so, this should be an important advance in specification technology.

A Appendix: Concurrent rewriting

A.1 Definitions

Our definitions and notations are consistent with those of G.Huet and D.Oppen [9]. Given a set X of variables and a graded set F of function symbols, the free F -algebra over X is denoted $T(F, X)$ and its elements are called terms. Similarly many-sorted terms can be defined as in [12]. Terms can be viewed as functions from the free monoid on the natural numbers denoted N^* to $F \cup X$. The domain of the term t considered as a function is denoted $O(t)$ and is called the set of occurrences of t . For example, $t(\epsilon)$ is the top symbol of the term t . $\text{Var}(t)$ denotes the set of variables of t , $t|_m$ the subterm of t at occurrence m (for $m \in O(t)$), and $t_{[m \rightarrow t']}$ the term obtained by replacing $t|_m$ by t' in t . A term is linear iff for any $x \in \text{Var}(t)$, x has only one occurrence in t . The set of nonvariable occurrence in a term is denoted $\bar{O}(t)$.

Substitutions σ are endomorphisms of $T(F, X)$ with a finite domain $D(\sigma)$. A substitution σ is denoted by $(x_1 \leftarrow t_1), \dots, (x_n \leftarrow t_n)$. We denote by $\sigma|_W$ the restriction of the substitution σ to the subset W of X . If Σ is a set of substitutions then $\Sigma|_W = \{\sigma|_W \mid \sigma \in \Sigma\}$ is the set of elements of Σ restricted to W .

An axiom is a pair $\{t, t'\}$ of terms denoted $t = t'$. A rewrite rule is a pair $\{t, t'\}$ of terms, denoted $t \rightarrow t'$, such that $\text{Var}(t')$ is a subset of $\text{Var}(t)$. A set of rewrite rules is called a Term Rewriting System (TRS). A rewrite rule $t \rightarrow t'$ is called left-linear (resp. right-linear) iff t (resp. t') is a linear term i.e. no variable occurs more than once in t (resp. t'). For example, if x is a variable, c is a constant, and $*$ is a binary operator, then the rule $x * c \rightarrow x$ is left-linear.

If R is a TRS, the rewriting relation \rightarrow^R is defined as follows: $t \rightarrow^R t'$ iff there exists an occurrence m in $\bar{O}(t)$, a rule $l \rightarrow r$ in R and a substitution σ such that $t|_m = \sigma(l)$ and $t' = t_{[m \rightarrow \sigma(r)]}$. $t|_m$ is called a redex in t at occurrence m under the rewrite rule $l \rightarrow r$. For t fixed, a redex may be represented by the 3-tuple (m, l, r) .

For example, for the previous rule $x * e \rightarrow x, (1 * 2) * e$ is a redex in the term $2 * (2 * ((1 * 2) * e))$ at occurrence 2.2. We write $t \rightarrow_{[m, \sigma, l \rightarrow r]}^R t'$ if we want to make explicit the occurrence, the substitution, and the rule involved in the rewriting.

A nonvariable term t overlaps a term t' at occurrence $m \in \bar{O}(t')$ iff there exists a substitution θ such that $\theta(t) = \theta(t'|m)$. For example the term $x * e$ overlaps at occurrence 1 the term $(2 * y) + 3$. A set R of rules is called **nonoverlapping** iff for any pair of rules $l \rightarrow r, l' \rightarrow r'$ in R l and l' do not overlap each other at any (nonvariable) occurrence. Otherwise the set is said to be **overlapping**. A rule $l \rightarrow r$ is called **self-overlapping** iff l overlaps itself at an occurrence different from ϵ ; associativity is a typical example of a self-overlapping rule.

Given a set A of equations, the A -equality relation (denoted $=_A$) is the smallest congruence relation closed under instantiation and generated by the set of axioms A . \vdash_A denotes one step of axiom application.

We denote by \leq_A the subsumption preorder on $T(F, X)$ defined by: $t \leq_A t'$ iff $t' =_A \sigma(t)$ for a substitution σ called a **match** from t to t' . Composition of substitutions σ and ρ is denoted by $\sigma.\rho$. Given a subset V of X , we define $\sigma \leq_A \sigma' [V]$ iff $\sigma' =_A \sigma''.\sigma [V]$ for a substitution σ'' .

Let t be a term and R a term rewriting system. Let $R(t) = \{(u_i, l_i, r_i)\}$ the set of all the redexes in t under R ; i.e.

$$(u_i, l_i, r_i) \in R(t) \Leftrightarrow l_i \rightarrow r_i \in R \text{ and } \exists \sigma \text{ s.t. } t|_{u_i} = \sigma(l_i)$$

Definition 2 A subset W of $R(t)$ is said to be **nonoverlapping** (or **non-conflicting** or **consistent**) iff for any redexes (u, l, r) and (u', l', r') in W ,

- $u|u'$ (i.e. u and u' are incomparable) or
- $u < u'$ and $\exists v \in O_{\text{var}}(l)$ s.t. $u.v \leq u'$ where $O_{\text{var}}(l)$ is the set of variable occurrences in l :

$$O_{\text{var}}(l) = \{\lambda | \lambda \in O(l) \text{ and } l(\lambda) \in X\}$$

This definition is illustrated by Figure 1 in the main text.

Let $\Delta(t)$ be the set of all nonoverlapping subsets of redexes in t .

Definition 3 The relation \rightarrow^{R_1} of concurrent rewriting is then defined by

$$t \rightarrow_W^{R_1} t' \Leftrightarrow \begin{cases} W = \{(u_i, l_i, r_i) | 1 \leq i \leq n\} \in \Delta(t) \\ \text{and} \\ i < j \Rightarrow u_i \not\prec u_j \\ \text{and} \\ t \rightarrow_{[u_1, l_1, r_1]}^R t_1 \rightarrow \dots \rightarrow_{[u_n, l_n, r_n]}^R t' \end{cases}$$

Note that the last condition specifies the result of applying the set of redexes in W with a bottom-up sequential strategy.

In the definition above it is possible to define the result t' using the notion of *residual* due to Church as defined in [8] or [2].

Definition 4 Let t a term, R a TRS, and $w_i = (u_i, l_i, r_i), i = 1, 2$ two redexes of R in t . Then the residual $w_1 \setminus w_2$ of w_1 by w_2 is the set of redexes defined by:

1. If w_1 does not cover w_2 (i.e. $u_1|u_2$ or $u_1 < u_2$ and the redexes are nonoverlapping) then $w_1 \setminus w_2 = \{w_1\}$;
2. If $\exists v \in O_{var}(l_2)$ such that $u_1 = u_2.v.v'$ then $w_1 \setminus w_2 = \{(u, l_1, r_1) | u = u_2.u'.v' \text{ with } u' \in O_{var}(r_2, l_2|v)\}$;
3. Otherwise $w_1 \setminus w_2 = \emptyset$.

This notion is now extended to W , a set of redexes of R in t . Let $w = (u, l, r)$ be any redex of R in t . The residual of W by reducing the redex w is

$$W \setminus w = \bigcup_{w' \in W} w' \setminus w.$$

This allows defining the notion of concurrent reduction by

$$t \xrightarrow{R_W} t' \Leftrightarrow \begin{cases} t \xrightarrow{R_{[u,l,r]}} t'' \text{ for } (u, l, r) \in W \\ \text{and} \\ t'' \xrightarrow{R_{[W \setminus (u,l,r)]}} t' \end{cases}$$

Definition 5 The relation of maximal concurrent rewriting \rightarrow^{R_1} is:

$$t \xrightarrow{R_1} t' \Leftrightarrow \begin{cases} t \xrightarrow{R_W} t' \\ \text{and} \\ W \text{ maximal in } (\Delta(t), \subseteq) \end{cases}$$

The relation of maximally concurrent rewriting $\rightarrow^{R_1^{max}}$ is:

$$t \xrightarrow{R_1^{max}} t' \Leftrightarrow \begin{cases} t \xrightarrow{R_1} t' \\ \text{and} \\ |W| \text{ maximal in } \Delta(t) \text{ for } \leq \end{cases}$$

A.2 Properties of Concurrent Rewriting

Definition 6 Let R be a relation on the set of terms, $\xrightarrow{\cdot}^R$ and $\xleftarrow{\cdot}^R$ be respectively its reflexive transitive and symmetric reflexive transitive closures.

R is terminating or noetherian iff there is no sequence of the form

$$t_0 R t_1 \dots t_n R t_{n+1} \dots$$

R is Church-Rosser iff

$$\forall t_1, t_2; t_1 \xleftarrow{\cdot}^R t_2 \Rightarrow \exists t' \text{ such that } t_1 \xrightarrow{\cdot}^R t' \text{ and } t_2 \xrightarrow{\cdot}^R t'.$$

Proposition 1

- If \rightarrow^R is terminating so is concurrent rewriting, and in particular \rightarrow^{R_1} .
- If \rightarrow^R is Church Rosser, then so is concurrent rewriting, and thus \rightarrow^{R_1} .

It is not easy to relax the above termination hypothesis as shown by the following,

Definition 7 A TRS is weak terminating iff every term has a normal form.

Example of weak terminating system (Barendregt, Huet): Let R be given by

$$\begin{array}{l} F(x, x) \rightarrow A \\ G(x) \rightarrow F(x, G(x)) \\ C \rightarrow G(C) \end{array}$$

- The first rule is non-left-linear.
- Note that C has the two normal forms A and G(A) (among others), and that R is locally confluent [10] (but not Church-Rosser).
- The concurrent relation $R^{\#}$ associated to R is not terminating. For example

$$F(C, G(C)) \rightarrow_{R^{\#}} F(G(C), F(G(C), G(G(C)))) \rightarrow_{R^{\#}} \dots$$

References

- [1] M. Bauderon and B. Courcelle. *Graph expressions and graph rewritings*. Technical Report, Université de Bordeaux 1, 1985.
- [2] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. Reynolds, editors, *Application of Algebra to Language Definition and Compilation*, Prentice Hall, 1985.
- [3] C. Dwork, P. Kanellakis, and L. Stockmeyer. *Parallel Algorithms for Term Matching*. Technical Report, MIT, 1986.
- [4] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages Conference*, 1985.
- [5] J.A. Goguen, C. Kirchner, S. Leinwand, J. Meseguer, and T. Winkler. *Progress Report on the Rewrite Rule Machine*. Technical Report, SRI International, 1986.
- [6] J.A. Goguen and J. Meseguer. EQLOG: equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming. Functions, Relations and Equations*, Prentice Hall, 1986.
- [7] J.A. Goguen and J. Meseguer. Extensions and foundations of object-oriented programming. In *SIGPLAN Notices*, 1986.
- [8] G. Huet and J.J. Levy. *Computations in Non-ambiguous Linear Term Rewriting Systems*. Technical Report, INRIA Laboria, 1979.
- [9] G. Huet and D. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, Academic Press, 1980.

- [10] J.P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, to appear. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City, 1984.
- [11] S. Leinwand and J.A. Goguen. *Architectural Options and Testbed Facilities for the Rewrite Rule Machine*. Technical Report, SRI International, 1986.
- [12] J. Meseguer and J.A. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, Cambridge University Press, 1985.
- [13] T. Winkler, S. Leinwand, and J.A. Goguen. *Simulation of Concurrent Rewriting*. Technical Report, SRI International, 1986.

END

DITIC

9 - 86