A Parallel Completion Procedure for Term Rewriting Systems [1]

Katherine A. Yelick
University of California at Berkeley

Stephen J. Garland
MIT and The Hebrew University

Abstract. We present a parallel completion procedure for term rewriting systems. Despite an extensive literature concerning the well-known sequential Knuth-Bendix completion procedure, little attention has been devoted to designing parallel completion procedures. Because naive parallelizations of sequential procedures lead to over-synchronization and poor performance, we employ a transition-based approach that enables more effective parallelizations. The approach begins with a formulation of the completion procedure as a set of transitions (in the style of Bachmair, Dershowitz, and Hsiang) and proceeds to a highly tuned parallel implementation that runs on a shared memory multiprocessor. The implementation performs well on a number of standard examples.

1 Introduction

We describe a parallel completion procedure for term rewriting systems. A sequential completion procedure was formulated first by Knuth and Bendix [13]. Extensions, modifications, and applications to algebra, theorem proving, and data type induction are described by Buchberger [3] and Dershowitz [5].

Performance is an important factor that limits the applicability of completion procedures, and of term rewriting systems in general. We show how parallelism can lead to significantly better performance. Opportunities for parallelism abound, because completion is not inherently sequential. But straightforward parallelizations of the Knuth-Bendix procedure perform poorly. Careful algorithm and data structure design is needed, as in sequential completion procedures, to avoid superfluous work. How parallel tasks are scheduled must be tuned, because the order in which steps are performed plays a crucial role in performance.

This paper is divided as follows. Section 2 defines the completion problem. Section 3 describes the issues that arise in finding good parallel solutions. Section 4 presents transition axioms for a completion procedure using the inference rules of Bachmair, Dershowitz, and Hsiang [2]. Section 5 transforms these axioms into ones suitable for parallel implementation. Section 6 describes the implementation. Section 7 describes its performance. Sections 8 and 9 describe related work and summarize our results.

2 The Completion Problem

We assume a familiarity with the notions of terms and substitutions. An equation is an unordered pair of terms, written $(s \doteq t)$. We write $(s \doteq t)$ when we want to

[1] Both authors were supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, by the National Science Foundation under grant CCR-8910848, and by the Digital Equipment Corporation. K. Yelick was supported in part by AT&T and by the University of California at Berkeley. S. Garland was supported in part by a Fulbright Lectureship.

Authors' addresses: K. A. Yelick, Computer Science Division, University of California, Berkeley, CA 94720. e-mail: yelick@cs.berkeley.edu. S. J. Garland, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. e-mail garland@lcs.mit.edu.

distinguish an orientation; i.e., $(s \doteq t)$ is either $(s \doteq t)$ or $(t \doteq s)$.

A rewrite rule is an ordered pair of terms, written $s \rightarrow t$. A term rewriting system R is a set of rewrite rules, which defines a relation $\rightarrow_R$ on terms such that $s \rightarrow_R t$ (*s rewrites to t*) if and only if there is a rule $l \rightarrow r$ in R and a substitution σ such that s contains σl as a subterm (l is said to *match* this subterm) and t is formed by replacing the occurrence of σl by σr. Let $\rightarrow_R^+$, $\rightarrow_R^*$, and $\leftrightarrow_R^*$ be the transitive, reflexive transitive, and reflexive symmetric transitive closures of $\rightarrow_R$. Since $\leftrightarrow_R^*$ is symmetric, it is well-defined even when R is replaced by a set of equations E. The relation $\leftrightarrow_E^*$ is the *equational theory of E* and is also denoted by E^*; i.e., an equation $(s \doteq t)$ is in E^* if and only if $s \leftrightarrow_E^* t$.

R is *confluent* if, whenever $r \rightarrow_R^* s$ and $r \rightarrow_R^* t$, there is a term u such that $s \rightarrow_R^* u$ and $t \rightarrow_R^* u$. R is *noetherian* if $\rightarrow_R^+$ contains no infinite chains. R is *convergent* if it is both confluent and noetherian. If R is convergent, then for any term t there exists a unique irreducible (i.e., unrewritable) term $t\downarrow_R$ (called the *normal form* of t) such that $t \rightarrow_R^* t\downarrow_R$. When R is simply noetherian, we write $t\downarrow_R$ to mean some one of the possibly many normal forms of t. When R is convergent, $s \leftrightarrow_R^* t$ if and only if $s\downarrow_R = t\downarrow_R$.

Some applications of term rewriting require convergent systems; others attempt to establish convergence by creating new rules in a process known as *completion*. Such applications raise the following related problems.

Definition. Given a set of equations E, the *finite completion problem* is to find a convergent rewriting system R such that $\leftrightarrow_R^*$ and $\leftrightarrow_E^*$ are the same relation.

For some some sets of equations E, no finite R exists that solves the finite completion problem. In such cases, it is sometimes useful to find approximations R_i to E, in the following sense.

Definition. Given a set of equations E, the *completion problem* is to produce a (possibly infinite) sequence of rewriting systems $R_0, R_1, \ldots$ such that each R_i is noetherian, $\rightarrow_{R_i}^*$ is contained in $\leftrightarrow_E^*$, and, for any equation $(s \doteq t)$ in E^*, there exists i such that for all $j > i$, s and t have unique normal forms in R_j, and $s\downarrow_{R_j} = t\downarrow_{R_j}$.

If the finite completion problem can be solved for E, the resulting rewriting system provides a decision procedure for E^*. If the completion problem can be solved, the resulting sequence of rewriting systems provides a semidecision procedure. However, there may not be a solution to the completion problem. For example, if E consists of a single commutativity axiom, no noetherian rewriting system has the same equational theory as E.

More general formulations of the completion problem increase the number of sets of equations for which solutions exist. Some generalizations allow function symbols in R that are not in E, requiring only that $\leftrightarrow_R^*$ be a conservative extension of $\leftrightarrow_E^*$. Others include completion modulo equations (such as commutativity) [17]. Although these generalizations fall in the class addressed by our approach, we do not consider them in this paper.

Solving the completion problem involves generating additional rules, if necessary, to ensure confluence. Such rules can be found using *unification* and *critical-pairing*. Two terms s and t are *unifiable* if there is a substitution σ (called their *unifier*) such that $\sigma s = \sigma t$. If $\sigma s = \sigma t$, and for all other unifiers σ', there is a substitution τ

such that $\sigma' = \sigma \circ \tau$, then σ is a *most general unifier* of s and t. If two terms are unifiable, then they have a most general unifier. A substitution τ is a *renaming* if, for all variables v in the domain of τ, $\tau(v)$ is a variable. Most general unifiers are unique up to renaming, i.e., up to composition with renaming substitutions.

Let r_1 and r_2 be the rewrite rules $s \to t$ and $l \to r$. Assume r_1 and r_2 have no variables in common. Then $(s' \doteq t')$ is a *critical pair* between r_1 and r_2 if some σ unifies l with a non-variable subterm of s, s' is formed from σs by replacing the subterm σl by σr, and t' is σt. Let $crit(r_1, r_2)$ be the set of critical pairs between r_1 and r_2, with r_2's variables renamed, if necessary, to avoid conflicts with r_1's. Let $crit\_all(R)$ be $\bigcup\{crit(r_1, r_2) : r_1, r_2 \in R\}$. Note that $crit\_all(R)$ contains $crit(r, r)$ for all r in R. Both *crit* and *crit_all* are unique up to variable renaming.

Solving the completion problem also involves establishing that a rewriting system is noetherian, which itself is an undecidable problem. The most common approach to proving that a system is noetherian is to use a *reduction ordering* on terms, i.e., a monotonic well-founded ordering that is stable under substitution [4]. If $>$ is a reduction ordering and $l > r$ for every rule $l \to r$ in R, then R is noetherian. Completion procedures generally employ a fixed reduction ordering and halt with failure if this ordering is not powerful enough to orient some equations that arise during completion. This prompts the following definition.

Definition. Given a set of equations E and a reduction ordering $>$, a *completion procedure* produces a possibly infinite sequence $R = \langle R_0, R_1, \ldots\rangle$ of rewriting systems such that:

1. Each R_i is provably noetherian; i.e., $l > r$ for all $l \to r$ in R_i.
2. Each R_i is consistent with E; i.e., $\leftrightarrow^*_{R_i}$ is contained in $\leftrightarrow^*_E$.
3. If some R_i solves the finite completion problem, then the procedure halts with success, and this R_i is the last element in R.
4. If no R_i solves the finite completion problem, then either (a) the procedure halts with failure or (b) R solves the general completion problem, i.e., R is infinite, and, for any equation $s \leftrightarrow^*_E t$, $s\downarrow_{R_i} = t\downarrow_{R_i}$ for all sufficiently large i.

The first two conditions are *safety properties*, which prevent completion procedures from producing rewriting systems that do not terminate, or that incorrectly reduce two terms to a common normal form. The last two conditions are *liveness properties*, which require completion procedures to achieve certain results.

Condition (4a) classifies trivial procedures, which always halt with failure, as completion procedures. Completion procedures typically differ on the set of inputs for which they fail, and their ability to resist failure is one of the qualities by which they are judged. Completion procedures can be made *failure resistant* by allowing the reduction ordering to be enlarged in restricted ways [8]. Completion procedures can be made *unfailing* by leaving some equations unordered and restricting the application of rewrite rules [2]. Because these generalizations do not raise interesting new questions concerning parallelism, we do not consider them in this paper.

3 Design Issues for Parallel Completion Procedures

The completion problem provides many opportunities for parallelism. At the same time, it presents many pitfalls. Ideally, a parallel completion procedure running on n processors should be close to n times faster than a well-tuned sequential completion procedure running on a single processor. Such speed-ups, however, may be difficult to attain. Because of synchronization overhead, tasks cannot be too fine-grained and parallel data structures must minimize synchronization bottlenecks. Because all processors must be kept equally busy to achieve maximum performance, tasks cannot be so coarse-grained as to prevent effective load-balancing. And because highly parallel processes can exhibit nondeterministic behavior, care must be taken to maintain correctness while optimizing performance.

We explore these issues by discussing various ways to parallelize the sequential Knuth-Bendix completion procedure [13]. This procedure has two alternating phases: *internormalization*, which rewrites and eliminates equations and rules, and *critical pairing*, which creates new rules. Although the requirements for completion procedures do not mention internormalization, experience with sequential completion procedures has shown that it is essential for good performance.

3.1 Granularity

The simplest approach to designing a parallel completion procedure is to keep the overall sequential structure of the Knuth-Bendix procedure, but to use fine-grained parallelism for low level computations. For example, we can use theoretically efficient parallel procedures to determine if two terms match during internormalization. Yet this helps little in practice: even with lightweight threads, terms must be huge—with hundreds or thousands of operators—before the gains outweigh the overhead of starting and synchronizing parallel tasks. Although very large terms arise in some applications, they are not the norm.

At a higher level of granularity, we can parallelize the operations of rewriting and computing critical pairs, for example, by attempting to match and unify different subterms in parallel. But this also requires very large terms to justify the overhead.

At better levels of granularity, we can (and do) attempt to rewrite a given term by several rewrite rules, or to compute critical pairs between different rules in parallel.

3.2 Load Balancing

A further way to parallelize the sequential completion procedure is to perform the two processes of internormalization and critical pairing in parallel, thereby creating a two-stage pipeline with a feedback loop from the critical pairing process to the internormalization process. Additional parallelism can be used within the two stages to try to balance the pipeline. Implementing this design taught two lessons that led us to abandon developing parallel programs from conventional sequential ones.

The first lesson is that the amount of time spent in normalization far exceeds the time spent in critical pairing. The magnitude of the difference—a factor of 20 is not unusual—limits the potential speedup of the pipeline to 5%. Furthermore, on a six-processor machine, dedicating one sixth of the processing power to 5% of the work is not a good use of resources.

The second lesson is that, even with many processors, performance instability between the two stages makes it difficult to balance the pipeline: an expensive critical pairing stage that generates many new rules is generally followed by an expensive internormalization stage. On typical iterations, the ratio of the time spent internormalizing to the time spent critical pairing varied between 1/2 and 79. We can address these specific performance problems, for example, by having the critical pairing stage work on more rules to make it relatively more expensive, by further optimizing internormalization to make it less expensive, or by dividing the stages into parallel subtasks and dynamically allocating additional processors. But these solutions avoid the real problem: there is a synchronization point after each iteration of the pipeline, when each stage gets new rules from the other. The situation is even worse without the pipeline, there being two synchronization points per iteration, one after internormalization and one after critical pairing.

Unnecessary synchronization points are artifacts of basing parallel programs on sequential programs. Hence we use a different, transition-based approach [23] that allows parallelism both between and within critical pairing and internormalization.

3.3 Correctness

Achieving performance while maintaining correctness requires care in choosing algorithms and data structures. For example, if we attempt to rewrite both sides of an equation in parallel, then failed rewrites must not modify the terms being rewritten, and successful simultaneous rewrites must not interfere with one another.

When internormalizing in parallel, we must be careful to prevent two rules that are the same up to the names of their variables from reducing one another to trivial rules, i.e., to rules with identical left and right sides.

4 High-level Transition Axioms for Completion

We base the design of our parallel completion procedure not on traditional sequential procedures, but on a reformulation of the original Knuth-Bendix procedure [13] by Bachmair, Dershowitz and Hsiang [2] as a set of nondeterministically-applied transition axioms. Figure 1 presents transition axioms for a completion procedure similar to the inference rules in [2]. The state consists of a set of equations E and a set of rewrite rules R. Initially, E holds the user's input and R is empty. (Our formulation differs slightly from that in [2], which assumes that neither E nor R contains elements that differ only by renamings. Such an assumption is difficult to implement.) The procedure stops if and when all guards are false, e.g., because the *fail* transition sets both E and R to *ordering_failure*.

The transition axioms preserve the invariant that each rule $s \rightarrow t$ in R is ordered with respect to $>$, i.e., $s > t$. An important property of reduction orderings is that if $s \rightarrow t$ and r both satisfy this invariant, and if r rewrites t to t', then $s \rightarrow t'$ also satisfies the invariant; hence *right_reduce* preserves the invariant. Because *left_reduce* may not satisfy the invariant, and because it may reduce a rule to a triviality, it turns a rewritten rule into an equation.

The requirement $\neg$*age_prevents*$(s \rightarrow t, r)$ in the guard for *left_reduce* prevents the completion procedure from removing all instances of a redundant rewrite rule. As noted earlier, when two rules are renamings of one another, either can reduce the other to a triviality. The details of this problem are not important for this paper, but the solution affects our presentation. We associate an *age* with each rewrite rule and define *age_prevents*(r_1, r_2) to be true if r_1 is older than r_2 and the left sides of r_1 and r_2 are renamings of each other. Using the age of rules is mentioned in [2], and the validity of this solution was confirmed by Dershowitz [6].

Any procedure that performs a fair interleaving of these actions solves the completion problem. CP fairness ensures both that every equation appearing in E is

State Components

E : set[equation] + ordering_failure
R : set[rule] + ordering_failure

Initially

E = user input
R = $\emptyset$

Transition Axioms

simplify % Apply one rewrite step to either side of an equation
$(s \doteq t) \in E \ \&\ t \rightarrow_R t' \Rightarrow$
$E := E - (s \doteq t) + (s \doteq t')$

delete % Delete trivial equation
$(s \doteq s) \in E \Rightarrow$
$E := E - (s \doteq s)$

orient % Convert equation into rewrite rule using reduction ordering
$(s \doteq t) \in E \ \&\ s > t \Rightarrow$
$E := E - (s \doteq t) \ \&\ R := R + (s \rightarrow t)$

right_reduce % Apply one rewrite step to right side of a rule
$(s \rightarrow t) \in R \ \&\ t \rightarrow_R t' \Rightarrow$
$R := R - (s \rightarrow t) + (s \rightarrow t')$

left_reduce % Apply one rewrite step to left side of a rule
$(s \rightarrow t) \in R \ \&\ r \in R \ \&\ s \rightarrow_{\{r\}} s' \ \&\ \neg\text{age\_prevents}(s \rightarrow t, r) \Rightarrow$
$R := R - (s \rightarrow t) \ \&\ E := E + (s' \doteq t)$

deduce % Add one critical pair between rules in R to E
$(s \doteq t) \in \text{crit\_all}(R) \Rightarrow$
$E := E + (s \doteq t)$

fail % Halt if there is an unorderable, nontrivial, normalized equation
$(s \doteq t) \in E \ \&\ s \neq t \ \&\ s\downarrow_R = s \ \&\ t\downarrow_R = t \ \&\ s \not> t \ \&\ t \not> s \Rightarrow$
$E, R := \text{ordering\_failure}$

Liveness

CP fairness: for any nonterminating execution $(E_0, R_0), (E_1, R_1), \ldots$ and any i, $\cap_{j>i} E_j = \emptyset$ and, if $e \in \cap_{j>i}\text{crit\_all}(R_j)$, then some renaming of e is in some E_k.

CP termination: if $(E_0, R_0), (E_1, R_1), \ldots$ is nonterminating, then no R_i solves the finite completion problem for E_0.

Figure 1: High-level transition axioms for completion

eventually ordered, simplified, or deleted, and also that every critical pair is eventually added to E. CP termination ensures that the completion procedure terminates if and when it solves the finite completion problem.

The completion procedure can fail if there is a nontrivial equation e in E that can be neither ordered nor rewritten, since no fair execution can leave e in E forever. Most implementations, however, will run indefinitely with an unorderable equation as long as there is other work to do. These procedures are not CP fair, but this is an academic point: finite machine resources will eventually take care of termination.

5 Implementable Transition Axioms for Completion

The transition axioms in Figure 1 are not appropriate for direct implementation. To make them implementable, we must find ways to ensure liveness, balance their granularity, and simplify their guards. At the same time, we seek to enhance performance by avoiding repetitive work. As in good sequential completion procedures, we want to avoid normalizing an equation or rule multiple times by the same set of rules, to avoid computing critical pairs for a given pair of rules more than once, and to avoid computing critical pairs between unnormalized rules.

5.1 Liveness

In sequential completion procedures, liveness is generally established by proving that appropriate invariants are true at different points of control. For example, nested loops might normalize all equations with respect to all rules, and invariants about which equations are normalized with respect to which rules may depend on the control point within each loop.

In our parallel completion procedure, we establish liveness instead by dividing the state (i.e., the sets of equations and rules) into components that satisfy appropriate invariants, e.g., about which equations and rules have been normalized with respect to which other rules, and which rules have had their critical pairs computed. A significant part of the design effort for parallel completion involves defining appropriate state components and invariants.

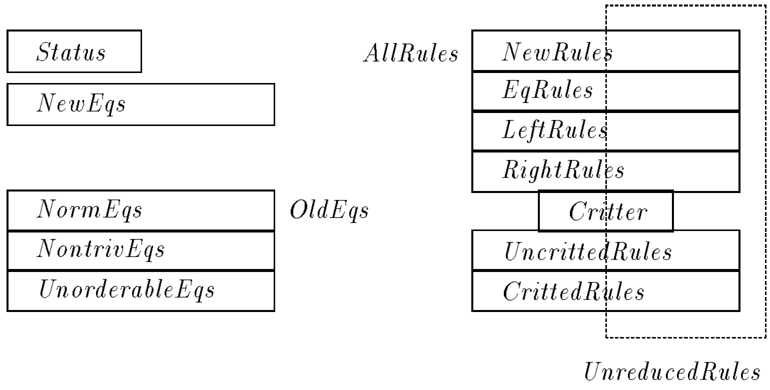
5.2 Granularity

Most transitions in Figure 1 perform a single rewrite step or compute a single critical pair. They incur considerable overhead, and are too fine-grained to be useful on a small number of processors. Therefore, we seek to increase their granularity.

Since actions that are too coarse-grained result in insufficient parallelism, we avoid both extremes by defining transitions that apply a single rule, at most once, to (at most) every rule or equation in the system, or that apply all rules as many times as possible to a single equation or rule. We avoid transitions that apply all rules to all equations.

5.3 Simple Guards

We simplify the guards for the transitions by turning them into inexpensive tests for the existence of an element in a state component or into tests involving the value of a scalar variable. The guards in Figure 1 involve more costly tests: a term being in normal form, an equation being orderable, or an equation being a critical pair between two rules. Fortunately, the same technique that helps with liveness can be



Initially, *NewEqs* = user input, *Status* = *running*, others = $\emptyset$.

Figure 2: Venn diagram of state components for completion

used to simplify guards: we put equations and rules into different state components, and we define invariants for these components that match the guards of Figure 1.

Figure 2 shows our state components. All but one is implemented as a *queue*; the queues *OldEqs*, *AllRules*, and *UnreducedRules* share elements with other queues, but the rest are mutually disjoint. The queues can simply be semiqueues, which ensure only that anything enqueued will eventually be dequeued. We use FIFO queues. The "usual" path that data (i.e., rules and equations) take in Figure 2 is from *NewEqs* to *OldEqs* to *AllRules* to *CrittedRules*. A final state component, *Status*, can have one of two values: *running* or *ordering_failure*.

The state components that contain equations have the following invariants. *OldEqs* contains the same elements as *NormEqs*, *NontrivEqs*, and *UnorderableEqs*. All equations in *OldEqs* are normalized with respect to *AllRules* − *NewRules*. No equation in *NontrivEqs* or *UnorderableEqs* is trivial. No equation in *UnorderableEqs* can be ordered by the reduction ordering.

The state components that contain rewrite rules have the following invariants. *UnreducedRules* is a subset of *AllRules*, which has the same elements as the seven other queues shown in Figure 2. If $s \rightarrow t$ is in *AllRules*, then $s > t$. The left side of every rule in *AllRules* is normalized with respect to *AllRules* − *NewRules* − *EqRules*. The right side of every rule in *AllRules* − *UnreducedRules* is normalized with respect to *RightRules*, *Critter*, *UncrittedRules*, and *CrittedRules*. *Critter* contains at most one rule; all critical pairs between this rule and itself have been added to *NewEqs* (but may no longer be there). For any r_1 and r_2 in *UncrittedRules* or *CrittedRules*, $crit(r_1, r_2)$ has been added to *NewEqs*. All critical pairs between the rule in *Critter* and those in *CrittedRules* have been added to *NewEqs*. (*UncrittedRules* is so-called because critical pairs have not yet been computed between its rules and *Critter*.)

Figures 3 and 4 contain directly implementable transition axioms for completion. These transitions move rules from one queue to another, maintaining the above invariants. In their descriptions, $rewritable(Q, r)$ contains the elements of Q that can be rewritten using r; $rewrites(Q, r)$ contains these elements, each rewritten once by r; $left\_reducible(Q, r)$ contains those $s \rightarrow t$ in Q such that $\neg age\_prevents(s \rightarrow t, r)$ and s can be rewritten by r; $left\_reduced(Q, r)$ contains these rules, with s rewritten

Transition Axioms

normalize_eqn

  $e = \text{head}(\text{NewEqs}) \Rightarrow$

    $\text{NewEqs} := \text{NewEqs} - e$ &

    $\text{NormEqs} := \text{NormEqs} + e\downarrow_{AR}$ &

    $\text{OldEqs} := \text{OldEqs} + e\downarrow_{AR}$

filter_eqn

  $(s \doteq t) = \text{head}(\text{NormEqs}) \Rightarrow$

    $\text{NormEqs} := \text{NormEqs} - (s \doteq t)$ &

    if $(s \neq t)$ **then** $\text{NontrivEqs} := \text{NontrivEqs} + (s \doteq t)$

orient_eqn

  $(s \doteq t) = \text{head}(\text{NontrivEqs}) \Rightarrow$

    $\text{NontrivEqs} := \text{NontrivEqs} - (s \doteq t)$ &

    if $(s > t)$ **then** $\text{NewRules} := \text{NewRules} + (s \rightarrow t)$

      & $\text{AllRules} := \text{AllRules} + (s \rightarrow t)$

      & $\text{OldEqs} := \text{OldEqs} - (s \doteq t)$

     elseif $(t > s)$ **then** $\text{NewRules} := \text{NewRules} + (t \rightarrow s)$

      & $\text{AllRules} := \text{AllRules} + (t \rightarrow s)$

      & $\text{OldEqs} := \text{OldEqs} - (s \doteq t)$

     else $\text{UnorderableEqs} := \text{UnorderableEqs} + (s \doteq t)$

back_simplify

  $r = \text{head}(\text{NewRules}) \Rightarrow$

    $\text{NewRules} := \text{NewRules} - r$ &

    $\text{EqRules} := \text{EqRules} + r$ &

    $\text{NewEqs} := \text{NewEqs} + \text{rewrites}(\text{OldEqs}, r)$ &

    foreach X **in** (NormEqs, NontrivEqs, UnorderableEqs, OldEqs)

     $X := X - \text{rewritable}(X, r)$

left_reduce

  $r = \text{head}(\text{EqRules}) \Rightarrow$

    $\text{EqRules} := \text{EqRules} - r$ &

    $\text{LeftRules} := \text{LeftRules} + r$ &

    $\text{NewEqs} := \text{NewEqs} + \text{left\_reduced}(\text{AllRules}, r)$ &

    foreach X **in** (NewRules, EqRules, LeftRules, RightRules, Critter,
      UncrittedRules, CrittedRules, UnreducedRules, AllRules)

     $X := X - \text{left\_reducible}(X, r)$

right_reduce

  $r = \text{smallest}(\text{LeftRules}) \Rightarrow$

    $\text{LeftRules} := \text{LeftRules} - r$ &

    $\text{RightRules} := \text{RightRules} + r$ &

    $\text{UnreducedRules} := \text{UnreducedRules} + \text{right\_reduced}(\text{AllRules}, r)$
      $- \text{right\_reducible}(\text{UnreducedRules}, r)$ &

    foreach X **in** (NewRules, EqRules, LeftRules, RightRules, Critter,
      UncrittedRules, CrittedRules, AllRules)

     $X := X - \text{right\_reducible}(X, r) + \text{right\_reduced}(X, r)$

Figure 3: Directly implementable transitions, Part I

right_normalize

$(s \rightarrow t) = \text{head}(\text{UnreducedRules}) \Rightarrow$

$\quad \text{UnreducedRules} := \text{UnreducedRules} - (s \rightarrow t)$ &

$\quad$ **foreach** X **in** (NewRules, EqRules, LeftRules, RightRules, Critter, UncrittedRules, CrittedRules, AllRules)

$\quad\quad$ **if** $(s \rightarrow t) \in X$ **then** $X := X - (s \rightarrow t) + (s \rightarrow t\downarrow_{\text{AllRules}})$

add_critical$_1$

$r_1 = \text{head}(\text{Critter})$ & $r_2 = \text{head}(\text{UncrittedRules}) \Rightarrow$

$\quad \text{UncrittedRules} := \text{UncrittedRules} - r_2$ &

$\quad \text{CrittedRules} := \text{CrittedRules} + r_2$ &

$\quad \text{NewEqs} := \text{NewEqs} \cup \text{crit}(r_1, r_2)$

add_critical$_2$

$\text{UncrittedRules} = \emptyset$ & $r = \text{head}(\text{RightRules}) \Rightarrow$

$\quad \text{UncrittedRules} := \text{Critter} \cup \text{CrittedRules}$ &

$\quad \text{CrittedRules} := \emptyset$ &

$\quad \text{Critter} := \{r\}$ &

$\quad \text{RightRules} := \text{RightRules} - r$ &

$\quad \text{NewEqs} := \text{NewEqs} \cup \text{crit}(r, r)$

fail

$\text{UnorderableEqs} \neq \emptyset \Rightarrow$

$\quad \text{Status} := \text{ordering\_failure}$

Liveness

Weak fairness: *fail* must be executed if its guard remains true.

Figure 4: Directly implementable transitions, Part II

once by r. *right_reducible*(Q, r) contains those $s \rightarrow t$ in Q such that t can be rewritten by r; *right_reduced*(Q, r) contains these rules, with t rewritten once by r.

Note that when *orient_eqn* creates a new rule, the equations in *OldEqs* must be renormalized. *Back_simplify* moves an equation rewritten by the new rule to *NewEqs*, where it will be renormalized by *normalize_eqn*. *Left_reduce* moves rewritten rules back to *NewEqs*, where they can be deleted or oriented in the other direction. *Right_reduce* puts rewritten rules into *UnreducedRules*, where they will be renormalized by *right_normalize*.

The liveness property in Figure 4 is *weak fairness* between *fail* and all other actions. Thus, an unorderable equation may exist in *UnorderableEqs* for a long time before the completion process fails, but unless the equation is moved or deleted, the process must eventually halt with failure. Note that there is no fairness requirement between any of the other actions.[2]

The axioms in Figures 3 and 4 can be viewed as an implementation of Figure 1, and standard refinement mapping techniques [1] applied to prove correctness. In these methods, the set of possible executions of the implementation are shown to be a subset of the executions of the specification, with a refinement mapping used to associate states of one with states of the other.

[2] Weak fairness is stronger than necessary, because the guard on *fail* may be continuously true even though *UnorderableEqs* has no persistent element. Moreover, the fairness condition may be dropped entirely if the semidecision procedure property is not desired.

5.4 Designing for Performance

Among the lessons learned from sequential implementations of completion are that internormalization enhances performance, but redundant normalization is inefficient. The *UnreducedRules* queue helps balance these conflicting requirements. When the right side of a rule is rewritten, it may no longer be in normal form with respect to the other rules; but equations and other rules that were previously unrewritable by this rule remain unrewritable. Hence we implement a "reminder" that the rule must be right-normalized by placing it in *UnreducedRules*, while at the same time leaving it in the queue to which it belonged so as not to lose the information contained in the invariant for that queue.

Although we designed our state components and transitions to avoid extra normalizations and critical pairing, the design does not prevent this from happening when objects become outdated. This trade-off keeps communication overhead low.

We implement *RightRules* as a priority queue, which is a special case of a semiqueue in which the "smallest" element is dequeued first. We choose this implementation because rules dequeued from *RightRules* will be used to compute critical pairs, and sequential implementations have demonstrated the practical importance of computing critical pairs between small rules before larger ones.

6 The Implementation

A two-step implementation based on the transition axioms in Figures 3 and 4 follows the transition-based approach described in [23]. First, we implement each transition axiom by a transition procedure. Our approach requires that these transition procedures appear atomic with respect to one another: any concurrent execution of the procedures must be equivalent to a sequential execution in which each procedure invocation takes a step specified by some transition axiom. Second, we implement a scheduler to execute the transition procedures in parallel. The scheduler, which is a parallel program that runs on all processors, guarantees liveness; being application-specific, it gives far better performance than would be achieved by allowing the operating system to schedule tasks. The weak liveness property in Figure 4 gives us the freedom to choose a scheduling order that performs well.

An easy way to make the transition procedures appear atomic is to make them atomic, e.g., to have them run within critical regions. But this leads to a completely sequential, albeit nondeterministic, completion procedure. Thus, significant algorithm and data structure design goes into implementing the transition procedures so that they appear atomic when they are actually highly concurrent.

Instead of describing our concurrent data structures here, we describe some of their properties and the kinds of parallelism they admit. We lock sparingly; the few critical regions are typically only a few instructions long. For example, each queue has two locks—one on the pointer to its head, the other on its pointer to the tail; each lock is held only long enough to increment its pointer. The queues contain pointers to rewrite rules and equations; the extra level of indirection allows rules and equations to be in several queues at once, as indicated by the overlapping regions in the Venn diagram of Figure 2.

Rewrite rules are modified in place, for example, by the *right_normalize* procedure, which must replace a rule by a right-normalized rule in each of seven queues. This procedure is implemented by modifying the rule in *UnreducedRules*, which is a shared copy of the rules in the other queues.

Rewrite rules are modified without locking. The code relies on the underlying shared memory, which guarantees that if a memory location is read at the same time it is written, then the read will observe either the value before or the value after the write—it will not observe an intermediate nonsensical value. A rewrite rule is represented by two locations, and simultaneous rewrites to the two terms are allowed. If two simultaneous rewrites are done to the same term, one may be lost. However, this loss does not affect correctness. It may affect performance in that the lost rewrite may have to be redone, but lost rewrites rarely happen in practice, and the effect on performance is not significant.

The scheduler that runs on each processor consists of a loop that repeatedly invokes transition procedures. Most of the design decisions in the scheduler are related to performance tuning, and are discussed in Section 7 along with the performance results. The problem of detecting termination within the scheduler is nontrivial, because it requires taking a global snapshot of the system to determine whether all guards are simultaneously false. If a process examines the queues in turn to see whether they are empty, by the time it finishes new elements may have been enqueued. Therefore, an agreement algorithm must be run between the schedulers on the separate processors to ensure that all continue running until they agree there is no work to do.

7 Scheduling and Performance

Experiments show that the best scheduler executes the transition procedures in a loop with the order given by Figures 3 and 4. This scheduler repeatedly invokes the same transition procedure as long as possible, then moves on to the next. The only exceptions are the *add_critical* procedures, which it executes only once before repeating the other procedures. This choice was motivated by experience with sequential completion procedures: performance, both in time and space, is better if equations and rules are kept in normal form. Thus, critical pair computations are stopped as soon as renormalization is required.

Figure 5 shows the performance of this scheduler on a Firefly with six CVAX processors when completing typical sets of equations taken from the term rewriting literature. The first column shows the number of milliseconds taken by the program running on one processor. The other columns show the relative performance with more processors. These figures were obtained by averaging five executions for each example; they show that speedups are better for the larger examples.

The first example generates a complete set of rewrite rules for group theory. The next provides some simple axioms for arithmetic, which *fib4* extends to compute the fourth Fibonacci number. Completing *homomorphism* establishes the fact that a map from one group into another that preserves group multiplication also preserves inverses and the identity. Completing *group56* produces a complete presentation of a group of order 56. The last example is due to Ursula Martin.

The time required to complete *group56* depends dramatically on the order in which critical pairs are computed, because one particular critical pair eliminates most of the other rules [16]. A round-robin scheduler, which executes each transition procedure once before going on to the next, seems to exhibit super-linear speed-ups for this example, gaining a factor of eight with two processors and fifteen with six.

	1 (ms)	1 : 2	1 : 3	1 : 4	1 : 5	1 : 6
group	1718	2.0	2.4	2.1	4.1	2.9
arith	4666	1.9	2.7	3.4	3.3	2.9
fib4	6603	1.8	3.1	2.9	2.2	3.5
homomorphism	11980	2.0	2.8	2.5	4.2	4.2
group56	18796	2.2	2.3	3.1	3.7	3.8
domino	44162	2.0	2.9	3.7	3.8	5.1

Figure 5: Transition-based completion using the best scheduler

However, the round-robin scheduler takes more time in all cases than the scheduler we use. Hence looking at speedups without considering absolute performance is a very bad way to tune parallel programs. Furthermore, leaving scheduling to the operating system is likely to produce terrible results.

Performance depends not just on the way the completion procedure schedules tasks, but also on the way the operating system allocates processes to processors. The times in Figure 5 were obtained using a system scheduler that does not move processes from one processor to another, thereby preserving cache context. When processes are allowed to move, performance is significantly worse: the speed-ups with six processors were about 65% of those in Figure 5.

Our parallel completion procedure is consistently faster, even when running on one processor, than the completion procedure provided by the Larch Prover [12]. While this comparison is somewhat unfair to the Larch Prover, which has more functionality, the comparison does show that our parallel implementation is fast enough to be of practical use.

These performance results are encouraging. For the larger examples, performance continues to improve as processors are added. This does not guarantee scalability beyond a small number of processors, but neither does it provide evidence against scalability.

8 Related Work

Dershowitz and Lindenstrauss [7] describe an abstract concurrent machine for rewriting a term by a set a rules, and they use it to compare various rewriting strategies. The parallelism in their model is very fine grained. They associate a process with each operator in a term and try to minimize the normalization time for that term. This is quite different from our implementation, which emphasizes larger grained parallelism; it allows parallel rewriting during internormalization, but does not intentionally schedule multiple rewrites of the same term.

Some sequential completion procedures [11, 14, 15] also divide the state into components, either to achieve better performance or to facilitate reasoning about correctness. The data structures used for our state components differ from these, however, because they are concurrent and have more control information to allow for the additional executions that come from parallelism.

Slaney and Lusk [20] describe a parallel closure computation, which divides the problem of completing $R = \{r_1, \ldots, r_n\}$ into tasks that compute critical pairs between r_i and $\{r_1, \ldots, r_i\}$, adding any results not subsumed by R to a new list S, and a single task that moves rules from S to R, incrementing n and rechecking the rest of S for subsumption. Because this method fails to address issues (such as internormalization) that affect the total amount of work performed, any speed-ups it exhibits are not very meaningful. For example, if two critical pairing tasks generate members s_1 and s_2 of S such that s_2 subsumes s_1, but not *vice versa*, a task will be spawned to compute critical pairs between s_1 and all members of R, without there being any way to abort such an expensive task when it is found to be superfluous.

A completion procedure that has been parallelized successfully is Buchberger's algorithm for computing Gröbner Bases [3]. Like us, Ponder observed performance problems when using a straightforward parallelization of Buchberger's algorithm [18]. Vidal [22] redesigned the data structures and rearranged the top-level procedure to get superior speed-ups. His procedure does internormalization in a separate sequential phase. It is therefore simpler than our procedure, but less efficient, since unreduced polynomials may cause extraneous computation.

9 Summary and Conclusions

Completion is a complicated problem, and the design of parallel completion procedures differs in nontrivial ways from the design of sequential procedures. The nondeterministic description of completion in terms of transition axioms given by Bachmair *et al* [2] is a good starting point for a parallel design, particularly because it avoids some of the unnecessary serialization inherent in parallelizing a conventional sequential program. But there is a large step between their description and the directly implementable transition axioms that describe our procedure. In particular, the liveness requirement (CP fairness) must be encoded into data structures without creating too many serialization points.

Our design technique appears promising as a means of parallelizing other applications that are highly irregular in their data structures, control structures, and communication patterns. Examples of problems that have been difficult to parallelize by other means include circuit verification and simulation, certain sparse matrix problems, and many symbolic applications.

Our implementation runs on a shared memory multiprocessor and performs well when completing typical sets of equations, generally achieving speedups of $4n/5$ or better when run with n processors. We are currently porting our implementation to an eight-processor Sequent, to test the scalability of our implementation.

We believe the completion procedure will scale to more processors as long as there is sufficient data to keep the transition procedures busy. Because completion often generates hundreds of intermediate rewrite rules, and because of the combinatorial nature of applying these rules, an implementation for upwards of a hundred processors should be interesting. There does not appear to be sufficient parallelism, however, to support tens of thousands of processors. To utilize that many processors, one would need to parallelize the processes of rewriting, matching, and unification.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. Research Report 29, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1988.

[2] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357. IEEE, 1986.

[3] B. Buchberger. History and basic features of the critical pair/completion procedure. *Journal of Symbolic Computation*, 3(1&2):3–38, February/April 1987.

[4] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[5] N. Dershowitz. Completion and its applications. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume II: Rewriting Techniques, pages 31–86. Academic Press, New York, 1989.

[6] N. Dershowitz, 1990. Private communication.

[7] N. Dershowitz and N. Lindenstrauss. An abstract machine for concurrent term rewriting. In *Proceedings of the Second International Conference on Algebraic and Logic Programming, Berlin*. LNCS, October 1990.

[8] D. Detlefs and R. Forgaard. A procedure for automatically proving termination of a set of rewrite rules. In *Proceedings of the First International Conference on Rewriting Techniques and Applications, Dijon, France*, pages 255–270. LNCS 202, May 1985.

[9] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, June 1984.

[10] C. Dwork, P. C. Kanellakis, and L. Stockmeyer. Parallel algorithms for term matching. *SIAM Journal of Computing*, 17(4):711–731, August 1988.

[11] S. J. Garland and J. V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1991.

[12] S. J. Garland, J. V. Guttag, and J. L. Horning. Debugging Larch Shared Language specifications. Technical Report 60, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1990.

[13] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, Oxford, 1970.

[14] P. Lescanne. Completion procedures as transition rules + control. In *TAPSOFT '89*, pages 28–41. LNCS 351, 1989.

[15] P. Lescanne. Orme, an implementation of completion procedures as sets of transition rules. In M. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 661–662. LNCS 449, 1990.

[16] U. Martin. Doing algebra with REVE. Technical report, University of Manchester, Manchester, England, 1986.

[17] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, Apr. 1981.

[18] C. Ponder. Evaluation of performance enhancements in algebraic manipulation systems. Technical Report UCB/CSD-88/438, Computer Science Division, University of California, Berkeley, CA 94720, 1988.

[19] R. Ramesh and I. Ramakrishnan. Optimal speedups for parallel pattern matching in trees. In *Proceedings of the 2nd International Conference on Rewriting Techniques and Applications, Bordeaux, France*, pages 274–285. LNCS 256, May 1987.

[20] J. K. Slaney and E. W. Lusk. Parallelizing the closure computation in automated deduction. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 28–29. LNCS 449, 1990.

[21] R. M. Verma and I. Ramakrishnan. Tight complexity bounds for term matching problems. *Information and Computation*, 1990.

[22] J.-P. Vidal. The computation of Gröbner bases on shared memory multiprocessors. Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, 1990.

[23] K. A. Yelick. *Using Abstraction in Explicitly Parallel Programs.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA 02139, December 1990. Also appeared as MIT/LCS/TR-507, July 1991.