Term-Rewriting Implementation of Equational Logic Programming

Michael J. O'Donnell
The University of Chicago

ABSTRACT

In 1975 I started a small project to explore the consequences of implementing equational programs with no semantic compromises. Latest results include a compiler that executes exactly the logical consequences of an equational program, with run-time speed comparable to compiled Franz LISP. This paper describes the accomplishments of the project very briefly, concentrating on shortcomings and directions for future work.

1 Introduction

The most common approach to providing semantics for programming languages is to regard a program as the definition of a collection of functions. In some cases great ingenuity is required to construct the unique function associated with each symbol in a program. Inputs and outputs are regarded as *values* in the domains of the defined functions, and the input/output behavior of the implementation of a program is expected to be exactly the function associated with some designated symbol in the program. I prefer, at least at the foundational level, to regard a program as an utterance in a dialogue between a *person* and a *computer,* and to explain its meaning in the style of mathematical logic preceding Computer Science. Inputs and outputs are additional utterances in the same dialogue, so they inhabit the syntactic world of the program, rather than a separate semantic world. Such a view leads to the following schema for *Logic Programming.*

> A *program* is a set of assertions in some logical language.
> An *input* is a question, specifying syntactically a set of possible (not necessarily correct) *answers* [BS76]
> A *correct output* for a given input is a logical consequence of the program that answers the input question.
> A *sound implementation* of a program is code that reads questions, and produces only correct answers. A *complete implementation* is one that produces a correct answer whenever one exists.

Prolog is the best-known example of Logic Programming. Aside from certain semantic compromises, it fits the schema above, with a program being a set of Horn clauses in the pure (*i.e,* equalityless) first-order predicate calculus, questions being of the form, "For what $x_1, \ldots, x_i$ does

$C[x_1, \ldots, x_i]$ hold?", with corresponding answers of the form "$C[\tau_1, \ldots, \tau_i]$". Languages with a Functional Programming flavor may also be viewed as Equational Logic Programming languages, as follows.

> An *Equational Logic Program* consists of a set of universally quantified equations.
> A term containing no instance of a left-hand side of an equation is in *normal form.*
> An *input* is a question of the form "What is τ?" for some ground term τ.
> An answer to such a question is an equation "$\tau = \nu$" where ν is a ground term in normal form.

Occurrences of instances of left-hand sides of equations are called *redexes*, rewriting of redexes to corresponding right-hand sides is called *reduction.* Inputs and outputs are, of course abbreviated in practice. In Prolog, the question, "For what $x_1, \ldots, x_i$ does $C[x_1, \ldots, x_i]$ hold?" is presented as "$C[x_1, \ldots, x_i]$", and the answer "$C[\tau_1, \ldots, \tau_i]$" is returned as "$x_1 = \tau_1, \ldots, x_i = \tau_i$". In Equational Logic Programming the question, "What is τ?" is presented as "τ", and the answer "$\tau = \nu$" is returned as "ν".

In 1975, I started a small project to explore the consequences of implementing Equational Logic Programming with no semantic compromises. Christoph Hoffmann, Paul Chew, Robert Strandh, Paul Golick, and Giovanni Sacco all collaborated in various ways. [ODo87] describes the project in more detail. In essence, we produced a programming language whose programs are sets of equations which, when treated as rewrite rules, are *regular* systems of rules [Klo80], *i.e.,* they are nonoverlapping and left-linear. [ODo77] proves that regular systems have the *confluence,* or *Church-Rosser property,* which is sufficient to guarantee the completeness of term-rewriting as an implementation of Equational Logic Programming. At first, I encountered great skepticism about the potential efficiency of such a language, so the project to date has emphasized efficient implementation with no compromises in the semantics.

We now have a compiler, produced by Robert Strandh, that clearly establishes that acceptable performance is achievable. The compiler accepts regular systems of equations, with a small additional restriction called *strong left-sequentiality* [HO79] based on Huet and Lévy's *strong sequentiality* [HL79], that allows left-right sequential processing similar to, but actually more general than, leftmost-outermost reduction. Compiled programs are guaranteed to behave precisely as required by the description of Equational Logic Programming above. The requirement of completeness leads to the uniform use of *outermost reduction,* which is often called *lazy evaluation* [HM76] [FW76]. Some pragmatically essential notational extensions are available, including the conventional arithmetic primitives, but all such extensions are precisely equivalent semantically to large sets of equations. Fast pattern-matching techniques, using precomputed finite-automaton transition tables, guarantee that the run-time cost of a single rewriting step is independent of the number and complexity of left-hand sides of equations — in contrast to the sequential search techniques used in most Prolog implementations.

Run-time performance of the current system is better than compiled Franz LISP for many examples. Franz LISP is not considered to be an especially fast implementation, but it is clearly an acceptable one, so there is no longer any question that usefully efficient Equational Logic Programming languages may be implemented by term-rewriting, although the limits of performance

cannot be known until a lot more attention is given to coding details and optimizations. More thorough descriptions of the current system are available in [HOS85] [ODo87] [ODo85]. In this paper I concentrate on shortcomings of the language, and directions for further research.

2 Regular Systems are Too Limited

It is easy to write equational programs that look very similar to LISP programs. In particular, if each equation has the form $f(\vec{x}) = \tau$, then τ is essentially a lazy LISP definition of the function f, using a somewhat prettier notation. Such minor notational improvement hardly justifies the effort of developing a new programming language. In particular, the large amounts of work that went into clever pattern-matching techniques is wasted on such a program, as a very naive strategy indexing on the head symbol does just as well. It is easy to transfer some of the conditional structure of τ above into multiple left-hand sides, *e.g.* $f(x) = cond(null(x), \tau_1, \tau_2)$ may be replaced by $f([]) = \tau_1'$ and $f([x|y]) = \tau_2'$ (I use Prolog notation for lists in this paper, even when discussing LISP), but such definition by cases still seems like a minor notational change, and certainly does not exercise the pattern-matcher very rigorously. Unfortunately, although the regular systems of equations are theoretically much more general than LISP programs, typical examples of concepts that do not code nicely into LISP are also troublesome to express with regular sets of equations.

For example, consider a LISP program for polynomial addition. A polynomial $a_m X^m + \cdots + a_0 X^0$ is represented by the list $[a_0, \ldots, a_m]$. A simple LISP function, perhaps called $POLYPLUS$, goes down a pair of lists adding corresponding elements, and eliminating trailing zeroes. The trouble with such a program is that it represents the concept of addition in three different ways, depending on the computational intention. First, there is the standard LISP function $PLUS$ to add two integer numerals. Next, there is the user-defined $POLYPLUS$ to add polynomials, which mathematically is the same sort of addition, merely applied to an expression with unknowns. Finally, each *cons* in a list of coefficients represents an addition and a multiplication, since the list of coefficients $[a_0, \ldots, a_m]$ is really just an encoding of the *Horner-rule form* $a_0 + X \times (a_1 + X \times (\cdots (a_{m-1} + X \times a_m) \cdots))$. It would be much more natural merely to take equations from a high-school algebra text, and massage them to reduce sums of Horner-rule forms to a single Horner-rule form. A first attempt seems to succeed with

1) $(i + X \times a) + (j + X \times b) = (i + j) + X \times (a + b)$
2) $(i + X \times a) + j = (i + j) + X \times a$
3) $i + (j + X \times b) = (i + j) + X \times b$
plus the equations defining addition of numerals

where i and j range over integer numerals, a and b are unqualified variables, and X is an atomic symbol standing for the variable of the polynomial (not to be confused with a variable in the equations). These equations code nicely into an executable equational program [ODo85], but they fail to remove trailing zeroes. The conceptually natural way to eliminate the zeroes is to add the two equations

4) $X \times 0 = 0$
5) $a + 0 = a$

but this introduces overlaps. For example, equations 2 and 4 overlap in $(7+X\times 0)+23$. There is a regular solution, but it is rather ugly [ODo85], and involves sneakily encoding control information into semantically irrelevant aspects of the form of a polynomial.

There is one example — weak reduction in the Combinator Calculus, with the binary function *APPLY* appearing both leftmost and nonleftmost in left-hand sides — where the perfect system of equations is regular but not LISP-like [ODo85]. There is also potential benefit in complex left-hand sides and sophisticated pattern matching if an equational program is used to represent the information in a database. Buneman and Frankel [BF79] have proposed equational databases as an alternative to relational ones. Equational Logic Programming seems to have the same natural connection to equational databases that Prolog has to relational databases. The normal Prolog strategy of sequential search works very badly for programs representing databases, but the pattern-matching tables used in the equational programming implementation look a lot like trie indexes already, and might lead straightforwardly to an efficient implementation.

It appears, however, that a really desirable equational programming language should allow benign overlaps that do not destroy the Church-Rosser property. Of course, the Church-Rosser property is undecidable, so decidable sufficient conditions are required. The Knuth-Bendix procedure [KB70] solves this problem for systems in which every reduction sequence leads to normal form. Their procedure detects overlaps (called *critical pairs*), constructs examples where a term α reduces by overlapping equations to β and γ, then reduces β and γ to see if they reach a common normal form. This procedure does not halt if either of β and γ has no normal form, but it could certainly be converted to an algorithm that explored a finite number of reductions of β and γ and searched for common, not necessarily normal, forms. Unfortunately, even if all such βs and γs reduce to common forms, this only establishes the *local Church-Rosser property* — when α reduces to β and to γ *in one step,* then β and γ reduce to a common form. The full Church-Rosser property applies also to many-step reductions. The local Church-Rosser property implies the full Church-Rosser property only for terminating systems. Further theoretical research is needed to determine whether some variation of the Knuth-Bendix procedure can provide a useful generalization of equational programming allowing some nonregular systems of equations.

Although nothing seems to be known about useful extensions of the Knuth-Bendix procedure to nonterminating systems, there are two paradigmatic sources of benign overlap that may serve as a guide. Many overlaps are the result of an equation expressing the associativity of an operator, for example $(i+j)+k=i+(j+k)$. This equation overlaps itself in $((1+2)+3)+4$. There is a lot of good literature on special techniques to deal with associativity, but no general technique that captures associativity naturally as a special case. Note that we may wish to require associativity of a function defined implicitly by a term schema, as well as requiring it of functions with explicit one-symbol names. The other typical benign overlap that occurs in equational definitions involves the interaction of a distributive property with a cancellation, identity or idempotence property. For example, $i\times(j+k)=i\times j+i\times k$ overlaps with $i+0=i$ in $2\times(5+0)$. There are probably no essentially new techniques for guaranteeing the Church-Rosser property required to deal with the typical cases, merely a careful choice of existing techniques. In particular, finite termination usually holds for the subsystem involving the overlapping equations. Some relatively easy analysis of the interaction of nonterminating regular systems with terminating, but nonregular, Church-Rosser

systems might prove to be very valuable.

Sufficient relaxation of the nonoverlapping restriction will allow programming techniques that go well beyond those available in the regular systems of equations. There are also many natural equations, such as $equal(x,x) = true$, that violate the left-linearity constraint. Such violations seem to be less common, and less crucial in practice, and the theoretical results needed to deal with them appear to be quite difficult. Paul Chew [Che81] proved that nonoverlapping, but not necessarily left-linear, equations produce unique normal forms, but they do not always have the Church-Rosser property. Uniqueness of normal forms, without the Church-Rosser property, is not enough to guarantee completeness of a term-rewriting implementation, since it may be necessary to back out of a nonterminating reduction path in order to get on one that leads to normal form. Given nice ways of guaranteeing the Church-Rosser property without left-linearity, there are still serious problems in implementing such systems, since there are implicit equality tests involved in determining whether an equation is applicable (Section 7).

3 Parallelism in Equational Programs

The equational programs accepted by the current implementation are all sequential, in the sense that in every term not in normal form, there is at least one outermost redex (called an *index* by Huet and Lévy) that must be reduced in order to reach normal form. A correct implementation *may* choose to work on several redexes in parallel, but a sequential strategy that always reduces an index is guaranteed to find a normal form whenever it exists, without wasting any reductions. In contrast, consider the *parallel or* equations $or(true,x) = true$, $or(x,true) = true$. Faced with a term of the form $or(\alpha,\beta)$, there is no way in general to choose which of the subterms α or β to reduce, and work on either may prove to be wasted. Intuitively, a parallel (or at least interleaved) reduction of α and β seems to be required.

The known techniques for analyzing sequentiality [HL79] [ODo85] [HOS85] [HO79] extend naturally to techniques for detecting a small number of cases in which parallel reduction is required. That is, *index sets* may be identified, where in order to reach normal form at least one redex in each set must be reduced, but it is not apparent which. Sequential systems are precisely those in which there is always at least one singleton index set. When all index sets have multiple elements, the essential idea is to choose some index set, and fork off parallel processes to handle each member of the set.

In principle, there is a simple implementation of this idea, but the precise space and time overhead of the implementation is extremely important in practice, and this has never been analyzed. On the one hand, the amount of state information for each process, and the data structure for maintaining the set of active processes, must be very small. On the other hand, due to sharing of subterms, the goals of several subprocesses may be the same through long subcomputations, and it is crucial to detect this and avoid repeating the work. Because of the nontrivial depth of left-hand sides, not all processes working at the same node behave in the same way, so detection of process equivalence is not quite trivial. A careful design of data structures and algorithms satisfying the constraints above has never been done, and is crucial to the practicality of inherently parallel equational programming. The application of index sets to efficient computation on truly parallel

machines is another intriguing line for future study. Given information about the availability of idle processors, it should be possible to use the index sets to choose a reasonable subset of redexes to reduce, balancing the possibility of slowing down convergence to normal form by not processing a needed redex against the possibility of wasting a processor on an irrelevant redex.

4 Incremental Input and Output

The run-time input/output interface of the currently implemented Equational Logic Programming system is quite primitive — a completely defined finite term is provided as input, and the system produces the normal form, if any, as output. So, the demand-driven lazy evaluation used internally by the system is not available at the interface. The first and obvious step is to produce a potentially infinite output term as its parts become known. Such incremental output is already implemented in the kernel of the system, because it is actually easier to program than batched output. Part of the theory of sequentiality involves criteria for determining that a certain symbol in a term is *stable* — that is, that no future reduction steps can possibly change it. It is elementary to output the next symbol in a developing normal form as soon as it becomes stable, even though the entire normal form is not known. In the current implementation, the pretty-printer does not support incremental output, so output can be viewed incrementally only in its crude form. This problem is very simple to solve, in principle. Incremental reading of the input as it is needed is also very easy to program, but that has not yet been done [ODo85].

A more serious problem conceptually is the arbitrariness of the left-to-right prefix order in which symbols of a term appear in conventional notation. The current implementation's treatment of incremental output, and the easy version of incremental input, require terms to be processed in that fixed order. Some prefix order is reasonably natural, but the left-to-right one may not be. A very flexible interface would involve a dialogue, in which the consumer of a term specifies dynamically the next symbol to produce, subject only to the constraint that an argument may not be seen before the symbol of which it is an argument. A nice open topic for research is the design of a protocol for such dialogues — perhaps it should involve a language for moving multiple cursors around a term, and querying the symbol at a designated cursor. It is tempting to allow the consumer to determine also when certain subterms are shared, but that idea presents very tricky design problems. Also, parallel queries may be desirable. Finally, it should be possible to reevaluate an input term incrementally after a *change* (as opposed to an extension). Such a facility would make equational programs very attractive for defining semantic back ends to structure editors.

An amusing semantic problem arises when input and output terms are processed incrementally. There is no way to guarantee, when producing a single symbol of output, that the remainder of the normal form is well defined and finite. In fact, it is very natural and convenient to request output of an infinite list — *e.g.,* the list of all primes — and to interrupt the process after seeing enough. Similarly, the producer of input cannot be prevented from extending the input term infinitely. While it seems perfectly clear what we want an implementation to do with infinite input and output terms, the semantics of Equational Logic Programming do not explain what sorts of infinite incremental behaviors are correct. The most obvious idea for extending the semantics is to suppose that, in every model, each infinite term denotes a value, just as do finite terms. Providing

such values seems to require continuity assumptions for all functions, and leads to semantics that cannot be realized computationally. For example, given the equations

$$a = [0|a];\ b = [0|b]$$

the inputs a and b both produce the same output — the infinite list of 0s. It is *not,* however, a logical consequence of these equations that $a = b$, and it is clearly undecidable and nonenumerable in general whether two terms are equal in the sense of producing the same infinite result (this is just the program equivalence problem).

So, I prefer to regard the infinite output η for the finite input α, not as an assertion of the infinite equation $\alpha = \eta$ but rather as an abbreviation for the infinite conjunction of finite equations

$$(\exists \vec{x}_1 \,.\, \alpha = \eta_1[\vec{x}_1]) \wedge (\exists \vec{x}_2 \,.\, \alpha = \eta_2[\vec{x}_2]) \wedge \cdots$$

where the $\eta_i[\vec{x}_i]$s are larger and larger approximations to the infinite term η, with each $\vec{x}_i$ being a list of distinct variables used in the unknown positions. So, with the equation $a = [0|a]$, and the input a, the output $[0, 0, \ldots]$ abbreviates

$$(\exists x \,.\, \alpha = [0|x]) \wedge (\exists x \,.\, \alpha = [0, 0|x]) \wedge \cdots$$

The notion of correctness follows immediately from this form, but there is some problem in defining what is an infinite answer to a question "what is α?" The basic idea is to let an answer be any finite or infinite conjunction of the form above, where each $\eta_i[\vec{x}_i]$ is in normal form, and remains in normal form whenever normal forms are substituted for $\vec{x}_i$ (the tempting alternative of merely requiring that every reduction of every instance $\eta_i[\vec{\tau}]$ retains the form $\eta_i[\vec{\tau'}]$ leads to obvious uncomputabilities). Furthermore, we should require an implementation to find the *maximal* correct answer under the instance ordering of infinite terms with variables, else the completely undefined output would always be acceptable. If the input α and the output η are both infinite, then the appropriate interpretation of the output is

$$\begin{array}{l}(\forall \vec{w}_1 \,.\, \exists \vec{x}_1 \,.\, \alpha_1[\vec{w}_1] = \eta_1[\vec{x}_1]) \wedge \cdots \wedge (\forall \vec{w}_1 \,.\, \exists \vec{x}_{f(1)} \,.\, \alpha_1[\vec{w}_1] = \eta_{f(1)}[\vec{x}_{f(1)}]) \wedge \\ (\forall \vec{w}_2 \,.\, \exists \vec{x}_{f(1)+1} \,.\, \alpha_2[\vec{w}_2] = \eta_{f(1)+1}[\vec{x}_{f(1)+1}]) \wedge \cdots \\ \vdots\end{array}$$

Where the $\alpha_i[\vec{w}_i]$s are larger and larger approximations to the infinite term α, and similarly for η as before, and f is some strictly increasing function. Notice how the function f is used to decouple the rates of production of the input and output. In general, α_i is the portion of the input processed to determine $\eta_{f(i)}$. I allow $\eta_{i+1} \equiv \eta_i$, so each new input symbol does not necessarily produce new output. The natural extension of the current implementation is not complete for the semantics suggested above, because input and output terms are always presented in left-right prefix order. There is no great technical difficulty in adapting the order of presentation to the results of rewriting, but it is hard to devise a good humanly-readable notation for the results using sequential input and output of strings. Rather, interactive input and output seems to be required.

5 Modularity

For practical purposes, a major failing of the current Equational Logic Programming Language is the complete lack of high-level programming support. Some facilities, such as type checking, are elementary to add as preprocessing steps. While it is not hard to design some reasonably useful modular constructs, it is very difficult to find modular constructs that are simultaneously semantically natural, practically convenient, and feasible to implement efficiently. First, we require combining operations that operate on the *meanings* of equational programs, rather than their texts. The most natural representative of the meaning of a program P seems to be a triple consisting of the term language defined by P, the set of models satisfying the equations in P, and the set of normal forms. In [ODo85] I tried to develop semantic constructs for combining such triples. Aside from some glaring errors in the definitions, the basic idea seems to be a failure, because combinations of regular systems are not necessarily regular, nor are they Church-Rosser. What is worse, regularity seems to be inherently a property of the program text, and not just its meaning. So, there seems to be no way of guaranteeing the good behavior of a combination of programs, without looking at their textual interactions. Goguen *et al.* have developed some useful modular constructs for the equational language OBJ, but they do not follow any discipline, such as regularity, for guaranteeing completeness [BG80] [FGJM85].

While I still hope that a thoroughly satisfactory modular treatment of Church-Rosser equational programs can be found, there seems to be more immediate promise in abandoning the Church-Rosser property. Given my commitment to completeness, abandonment of the Church-Rosser property requires a semantics that is no longer based on equational logic. A natural alternative is *subset logic.* Let every term represent a *set* of values, where functions are required to operate pointwise (*i.e.*, $f(S) = \bigcup\{ f(\{x\}) \mid x \in S \}$). Instead of the equality relation, use the subset relation to express programs. Subset logic is complete with the *reflexive, transitive,* and *substitution* rules of equality, omitting the *symmetric* rule. So, nondeterministic term rewriting from left to right is a complete implementation for subset logic, with no restrictions on the rules. Technically, it doesn't matter whether the left side of a rule is a subset of the right, or *vice versa,* as long as the direction is always the same. Intuitively, it seems more natural to think of reduction as producing a subset of the input term, since then a term may be thought of as denoting a set of possible answers. Thus, a single line of a Subset Logic Program looks like $\alpha \supseteq \beta$. Note that normal forms do not necessarily denote singleton sets, although it is always possible to construct models in which they do.

6 Indeterminate Computations

Subset Logic Programming naturally supports programs with indeterminate answers, since normal forms are not unique, and the logic programming schema allows any of possibly many correct answers to be produced. It would also be possible to require the set of all correct answers to be output, but it would be more natural to view a system of this sort as equational, since the output set is unique. While Equational Logic Programming extends naturally to infinite inputs and outputs, without changing its application to finite terms, such extension of Subset Logic Programming is

more subtle. If only finite terms are allowed, then infinite computations may be regarded as a sort of failure, and a finite normal form must be found whenever one exists. If incremental output of possibly infinite normal forms is desired, then there is no effective way to give precedence to the finite forms when they exist. The most natural idea seems to be to follow all possible reduction paths, until one of them produces a stable symbol for output. Whenever such a symbol is output, all reduction paths producing different symbols at the same location are dropped. Only a reduction path that has already generated all of the symbols that have been output is allowed to generate further output. The details work out essentially the same as with infinite outputs for equational programs, merely substituting $\supseteq$ for $=$. It is not at all clear whether this logically natural notion of commitment to a symbol, rather than a computation path, is useful. I cannot find a natural semantic scheme to support the more conventional sort of commitment to a computation path, although a user may program in such a way that multiple consistent paths never occur.

Efficient implementation of Subset Logic Programming presents a very interesting challenge. It is obviously unacceptable to explore naively the exponentially growing set of reduction sequences. A satisfying implementation should take advantage of local Church-Rosser behavior to prune the search space down to a much smaller set of reductions that is still capable of producing all of the possible outputs. The correct definition of the right sort of local Church-Rosser property is not even known. It is *not* merely the Church-Rosser property for a subset of rules, since two rules that do not interfere with one another may interfere differently with a third, so that the order in which the two noninterfering rules are applied may still make a difference to the outcome. Pattern-matching and sequencing techniques must be generalized as well, and a good data structure designed to represent simultaneously the many different reductions under consideration as compactly as possible. Sharing of equivalent subterms becomes problematic since it may be necessary to reduce two different occurrences of the same subterm in two different ways. Significant solutions to these problems could be well worth the effort, since Subset Logic Programming would capture the useful indeterminate behavior of Prolog, while avoiding the repeated generation of the same solution by slightly different paths.

7 Proving Equality and Solving Equations

While reducing input terms to normal forms is useful, and sufficient in principle for all computation, there are other sorts of derivations from equational programs that could be very useful. Term rewriting is already widely used for testing equality, using systems of rules that are guaranteed always to produce unique normal forms [HO80]. A significant extension of these techniques to systems without termination could be very valuable. Even when applied to terminating systems, techniques designed for nonterminating systems are often more efficient, since they must avoid unnecessary and potentially infinite subcomputations. Three substantial technical problems arise in such a generalization. First, the Knuth-Bendix procedure for checking, and in some cases producing, the Church-Rosser property fails completely in the presence of infinite reduction sequences, as discussed in Section 2. In addition, although the Church-Rosser property guarantees that if $\alpha = \beta$ holds, then it may be proved by reducing α and β to some common form, sequencing techniques that suffice for producing normal forms may fail to prove equalities of terms without normal forms.

For instance, to prove that $f(\alpha) = f(\beta)$, it may suffice to reduce α and β to a common form, or it may be necessary to apply a reduction involving the outermost symbol f to one or both sides. At first it appears that only *complete* reduction sequences — which reduce every redex — can be sure of doing enough reductions, at the cost of much wasted work. Finally, even if enough reductions are tried, a proof may be missed because the two terms are out of synchronization, and the current state of one agrees with some old forgotten state of the other, so we may have to search through many different reduction sequences to find a common form.

The first problem, of guaranteeing the Church-Rosser property, may be solved by restricting to regular systems of equations, although that restriction seems too strong even for programming applications, and is likely to be even less desirable for theorem proving. Significant generalization of the Knuth-Bendix procedure in this direction is completely open. The second problem, sequencing, is not solved, but a promising direction is apparent. Jayaraman noticed that, when trying to prove $\alpha = \beta$, as long as the outermost symbols disagree, only outermost reductions need be tried [Jay85]. Once the outermost symbols agree, it is necessary to explore in parallel further outermost reductions and a decomposed problem of proving equality of corresponding arguments. Within each argument the same reasoning applies. It seems likely that disagreement of the outermost symbols will be very common, so this observation can save a lot of steps. A remaining problem, not treated by Jayaraman, is to implement the required parallelism efficiently. In particular, the globally outermost step will often be the same as one of the outermost steps within a subproblem, so an efficient implementation should take care not to repeat work that is required in several different ways. There appears to be some convergence between these ideas based on term rewriting, and recent work on resolution-based equational theorem proving [DH86].

The final problem, of avoiding search through many different reduction sequences, is probably solved by a careful application of Paul Chew's *directed congruence closure* algorithm [Che80]. Congruence closure was originally intended to provide decision procedures for theories of finite sets of equational postulates, with no variables. The essential idea is to construct an undirected graph whose nodes represent subterms of the left- and right-hand sides of postulated equations, and of the terms being tested for equality, and whose edges represent known equalities. Initially, the only edges are those given by the postulates. Then, the transitive and congruence properties of equality are applied as closure operations, adding edges to the graph (symmetry is handled implicitly by using undirected edges). It is easy to see that all equalities between terms corresponding to nodes in the graph that are logical consequences of the postulates will be generated by this procedure. Kozen noticed that there is a polynomial time algorithm for congruence closure [Koz77], and Downey, Sethi and Tarjan developed the theoretically most efficient algorithms [DST80], but the basis for practical work, and the inspiration for Chew's study of congruence closure, comes from Nelson and Oppen, who used a theoretically slower, but for most applications better, algorithm in automatic theorem provers [NO80]. For Equational Logic Programming, the congruence closure technique looks very attractive, because it never evaluates the same term twice. Unfortunately, the technique does not apply directly, because of the use of variables in equational programs.

In principle, it is possible to extend congruence closure to equations with variables, by enumerating the ground instances of these equations, and performing the closure operation on the larger and larger finite sets of equations without variables so generated. In conversation, I learned

that Oppen had applied such a method quite satisfactorily to theorem proving. For programming language implementation, the potentially exponential waste of enumerating unnecessary instances is not acceptable. Chew showed that it is possible to generalize the data structures used in congruence closure to represent a directed graph of known reductions, and to extract efficiently from those data structures the most reduced equivalent to a given term, using all possible applications of all instances of reduction rules considered so far. Part of that term may be undefined, but the well-defined part is always sufficient to determine the next outermost reduction that would be made by a term rewriter. Instead of actually performing a reduction, Chew's *directed congruence closure* procedure adds the next instance of a reduction rule that would be applied to an initially empty finite set of ground instances, and performs congruence closure on the result [Che80].

The directed congruence closure technique has never been tested in practice, although it appears to be susceptible to efficient implementation. Such an implementation could turn out to be very valuable, possibly subsuming all of the special techniques that have been developed for "memoing." In fact, once the directed congruence closure procedure has processed a set of instances of rules, the resulting data structure represents, in a compressed form, all of the possible results of applying all possible subsets of those instances to all possible occurrences in the term being reduced. Thus, an apparently exponential, and in some cases where a finite set of instances produce a reduction loop, infinite, search space can be managed efficiently. An extension of such techniques incorporating derived equations between nonground terms would be very interesting and useful. Once the theoretical problems regarding which derived equations to add are solved, Strandh's incremental algorithm for generating pattern-matching tables [Str84] will probably provide a good technique for adding such rules on the fly.

Beyond proving equalities, it would be very useful to be able to solve equalities between terms with variables. This is precisely the problem of unification modulo a set of equations, and has been studied extensively in the attempt to combine Prolog with Functional Programming, but without definitive solution. Given a good solution to the problem of proving equalities between ground terms, a solution to the unification problem may probably be derived by running the prover as if the two terms were ground, then, whenever the procedure tries to look at a variable position, considering possible instantiations. It is easy to show that only instantiations that immediately either match the other term, or contribute to a redex, need be considered.

References

[BF79] O. P. Buneman and R. E. Frankel. FQL — a functional query language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 52–57, SIGMOD, May–June 1979.

[BG80] R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332, 1980.

[BS76] N. D. Belnap and T. B. Steel. *The Logic of Questions and Answers.* Yale University Press, 1976.

[Che80] L. P. Chew. An improved algorithm for computing with equations. In *21st Annual Symposium on Foundations of Computer Science*, pages 108–117, IEEE, 1980.

[Che81] L. P. Chew. Unique normal forms in term rewriting systems with repeated variables. In *13th Annual ACM Symposium on Theory of Computing*, pages 7–18, 1981.

[DH86] V. J. Digricoli and M. C. Harrison. Equality-based binary resolution. *Journal of the ACM*, 33(2):253–289, 1986.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[FGJM85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *12th Annual Symposium on Principles of Programming Languages*, pages 52–66, SIGPLAN and SIGACT, 1985.

[FW76] D. Friedman and D. Wise. Cons should not evaluate its arguments. In *3rd International Colloquium on Automata, Languages and Programming*, pages 257–284, Edinburgh University Press, 1976.

[HL79] G. Huet and J.-J. Lévy. *Computations in Non-ambiguous Linear Term Rewriting Systems.* Technical Report 359, IRIA, 1979.

[HM76] P. Henderson and J. H. Morris. A lazy evaluator. In *3rd Annual Symposium on Principles of Programming Languages*, pages 95–103, SIGPLAN and SIGACT, 1976.

[HO79] C. M. Hoffmann and M. J. O'Donnell. Interpreter generation using tree pattern matching. In *6th Annual Symposium on Principles of Programming Languages*, pages 169–179, SIGPLAN and SIGACT, 1979.

[HO80] G. Huet and D. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems*, Academic Press, 1980.

[HOS85] C. M. Hoffmann, M. J. O'Donnell, and R. I. Strandh. Implementation of an interpreter for abstract equations. *Software — Practice and Experience*, 15(12):1185–1203, 1985.

[Jay85] B. Jayaraman. *Equational Programming: A Unifying Approach to Functional and Logic Programming.* Technical Report 85-030, The University of North Carolina, 1985.

[KB70] D. E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 127–146, Pergammon Press, Oxford, 1970.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* PhD thesis, Mathematisch Centrum, Amsterdam, 1980.

[Koz77] D. Kozen. Complexity of finitely presented algebras. In *9th Annual ACM Symposium on Theory of Computing*, pages 164–177, 1977.

[NO80] G. Nelson and D. C. Oppen. Fast decision algorithms based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[ODo77] M. J. O'Donnell. *Computing in Systems Described by Equations.* Volume 58 of *Lecture Notes in Computer Science*, Springer-Verlag, 1977.

[ODo85] M. J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, MA, 1985.

[ODo87] M. J. O'Donnell. Survey of the equational logic programming project. In *Colloquium on Resolution of Equations in Algebraic Structures*, 1987.

[Str84] R. I. Strandh. *Incremental Suffix Trees with Multiple Subject Strings.* Technical Report JHU/EECS-84/18, The Johns-Hopkins University, 1984.