Goal Solving as Operational Semantics

Nachum Dershowitz
Department of Computer Science
University of Illinois
Urbana, IL 61801, USA
nachum@cs.uiuc.edu

Abstract

To combine a functional or equational programming style with logic programming, one can use an underlying logic of Horn clauses with equality (as an interpreted predicate symbol) and (typed) terms. From this point of view, the most satisfying operational semantics would search for solutions to equations or predicates. "Narrowing" and many of its variants are complete mechanisms for generating solutions. Such a melded language is more expressive than either paradigm alone: functional dependencies are explicit; "multi-valued" functions can be better expressed as predicates; nested functions can be evaluated without recourse to search (backtracking); (non-constructor) terms can serve as arguments to predicates; functions can be inverted; nonterminating functions can be programmed in a terminating fashion; goals can be simplified in a "don't care" manner; "functional" negation can prune searches. Moreover, the availability of backtracking and existential ("logic") variables provides an alternative to infinite data structures ("streams").

1 Introduction

Functional programming and logic programming are two relatively new styles of programming, each with its own set of advantages. Today, they are both well-established paradigms, each with its own set of devotees.

Since the mid-eighties, a fair amount of effort has been invested in combining the best of both worlds. Though many good ideas have been forthcoming, acceptance of this work has been hampered by slow implementations, on top of the natural tendency to make do with what one already has available.

In what follows, we reiterate the beauty of using equational Horn clauses for programs and show how conditional equations and goal-solving obviate the need for lazy evaluation.

For example, given the program

$$\begin{array}{rcl} primes & \rightarrow & sift(from(2)) \\ from(n) & \rightarrow & n : from(n+1) \\ sift(x:z) & \rightarrow & x : sift(filter(z, \lambda y(x|y))) \\ filter(p, y:z) & \rightarrow & y : filter(p,z) \ :\!- \ \neg p(y) \\ filter(p, y:z) & \rightarrow & filter(p,z) \ :\!- \ p(y) \\ x|y & :\!- & z \times y = x \end{array}$$

with appropriate type declarations and standard definitions of multiplication and negation, solving the goal $p_1 : p_2 : p_3 : z =^? primes$ should yield the first three primes.

2 Functional and Equational Programming

In functional programming languages, one expresses programs as function definitions and computes by supplying an expression to be evaluated. Function applications are "expanded" according to the definitions until all defined function symbols are eliminated and a "constructor" term is obtained. In pure Lisp, a conditional construct is used for defining functions. A program for interleaving two lists could be written as follows:

$$inter(x,y) \rightarrow if\ x = nil\ then\ y\ else\ car(x) : inter(y, cdr(x))$$

In ML, and most modern functional languages, patterns are used to delineate the various cases in a definition; thus one can write instead something like:

$$\begin{array}{rcl} inter(nil, y) & \rightarrow & y \\ inter(z:x, y) & \rightarrow & z : inter(y, x) \end{array}$$

Miranda [39] allows repeated variables in patterns to indicate that a clause of the definition applies only when two arguments are identical. Functional languages are typically higher-order and allow for functions to be passed as arguments to other functions, but patterns cannot be used to distinguish one functional argument from another. Most functional languages impose a type discipline. Much recent work has gone into the efficient implementation of functional languages.

The operational semantics of such a language are "logically complete" if whenever the definitions imply that a given variable-free ("ground") term s is equal to a constructor term t, the output t will be returned when s is given as input. The mutual exclusiveness of the different cases in function definitions guarantees that s is equal to at most one constructor term t. It is well known that a "lazy" ("outermost fair", "call by name") evaluation strategy—as employed in SASL [38] or Miranda [39]—is required for completeness, and that an "eager" (innermost, "call by value") strategy—as in Lisp or ML—can lead to an infinite computation, without ever finding t. Thus, an interpreter programmed in a lazy language will terminate whenever the computation it

is mimicking terminates. Computation with infinite structures ("streams") is often touted as another advantage of lazy evaluation [11, 20]: only that part of the structure that is needed for a particular computation need be evaluated.

Equational programming ("rewriting") extends the notion of functional programming by allowing one to use a set of directed equations rather than just function definitions. *Rewrite rules* are used to replace "equals-by-equals" in a specified direction. Rules are repeatedly applied to any term containing a subterm that matches a left-hand side; when a rule matches, the matched subterm is replaced by the corresponding instance of the right-hand side. Functions need not be total. The output one seeks is the normal forms (unrewritable expressions) of (i.e. equal to) given input terms, but normal forms need not be constructor terms.

The difference between functional and equational programming can be seen in a program for conjunctive normal form. In the equational approach, one merely writes equations for eliminating double negations, for DeMorgan's Laws and for distributivity of "or" over "and", and uses them to rewrite expressions into normal form. In this case, the symbols for "not," "and," and "or" are not defined functions, and the rules (like $\neg(x \wedge y) \rightarrow (\neg x) \vee (\neg y)$) have nested occurrences of them on the left. In the functional approach, one would consider those connectives to be constructors, and define a function *cnf* with cases like $cnf(\neg(x \wedge y)) \rightarrow cnf(\neg x \vee \neg y)$, as well as $cnf(x \wedge y) \rightarrow cnf(x) \wedge cnf(y)$ and $cnf(x \vee y) \rightarrow dist(cnf(x), cnf(y))$, where *dist* is an auxiliary function that "multiplies out" its two arguments.

Though equations are an important means for specifying properties of functions operating on data, conditional equations (that is, equational Horn clauses) provide an even more versatile programming paradigm. A *(conditional) rule* is an equational implication in which the equation in the conclusion is oriented and conditions are either atoms or equations. A rule $l \rightarrow r \ :\!- \ c_1, \ldots, c_n$ is applied to a term $t[l\sigma]$ containing an instance $l\sigma$ of the left-hand side l of the rule if $c_i\sigma$ can be rewritten to true (if the condition is an equation, then both sides must rewrite to the identical term), for each condition c_i, in which case $t[l\sigma]$ is replaced by $t[r\sigma]$. (If there are no conditions c_i, the rule is "unconditional.") We call sets of such rules *(standard conditional) rewrite systems*. Conditional rewriting is well-behaved if recursively evaluating conditions always terminates. An input term is repeatedly rewritten according to the rules; when (and if) no rule applies, the resultant *normal form*, is considered the "value" of the initial term. The following (along with unconditional rules for subtraction and inequality) constitutes a (nondeterministic) program for greatest common divisor:

$$\begin{array}{rcll} gcd(0, x) & \rightarrow & x & \\ gcd(x, s(y)) & \rightarrow & gcd(x - s(y), s(y)) & :\!- \ x > y \\ gcd(s(x), y) & \rightarrow & gcd(y - s(x), s(x)) & :\!- \ y > x \end{array}$$

For completeness, one needs to find the normal forms whenever they

exist. *Confluence* of a rewrite system is a property that ensures that no term has more than one normal form. For a survey of the theory of conditional (and unconditional) rewriting, see [26].

There are two approaches to completeness of equational programming, both of which demand confluence for ground terms. In analogy with functional programming, one can insist that left-hand side patterns do not overlap and that no variable appears more than once in a pattern. *Orthogonal* systems (a) have no variables that do not also appear on the left-hand side, (b) are "left-linear," and (c) no left-hand side unifies with a renamed nonvariable subterm of a left-hand side (other than with itself). The *gcd* program is not orthogonal. Unconditional orthogonal systems are confluent, even for nonterminating sets of rules, and an outermost rewriting strategy is guaranteed to reach a normal form if such exists, for which reason, orthogonal systems are popular in equation-based programming languages [33]. (For efficiency reasons, one may want to impose additional restrictions on the form of left-hand side patterns [32].) A conditional rewrite system is *normal* if one side of each condition in each rule is a ground normal form (like $true$). Orthogonal normal systems are also confluent (see [26, Theorem 3.0.10]).

The alternative approach requires that all computation strategies terminate in a normal form. A conditional system is *decreasing* if there exists a well-founded extension $\succ$ of the rewrite relation for which terms are greater than subterms and every instance of a left-hand side is greater than the corresponding instance of each condition. Decreasing systems exactly capture the finiteness of recursive evaluation of terms. Uniqueness is ensured if the system passes a "critical pair" test [26, Definition 3.0.14], which states that confluence of (finite) terminating systems can be effectively tested by checking that both sides of each of a finite set of equations have the same normal form, but this is not a necessary condition for ground confluence [26, Theorem 3.0.15].

Without the decreasing condition, the critical pair test is insufficient, though it does apply when no left-hand side unifies with a proper subterm of any left-hand side [26, Theorem 3.0.15]. Note that interpreting Horn clauses as conditional rewrite rules (rewriting predicates to $true$) gives a system satisfying this constraint, because predicate symbols are never nested in the "head" of a clause. Furthermore, all critical pairs pass the test, since all right-hand sides are the same. This also applies to pattern-directed functional languages in which defined functions are not nested on left-hand sides.

Whereas the first approach suffers from strong syntactic restriction of orthogonality; the second has the disadvantage of requiring termination, leaving interpreters, streams, and other nonterminating programs behind. For other methods of establishing conditional confluence, see [15, 26].

3 Functional-Logic Programming

Various proposals have been set forth for combining features of functional programming and logic (relational) programming; surveys include [1, 15, 35]. Functional notation and an evaluation mechanism are borrowed from functional programming; assignment by unification and goal-solving are borrowed from logic programming. One can, for instance, begin with a functional language like Lisp and add backtracking and unification. In fact, some early languages for Artificial Intelligence, like PLANNER [18] and Qlisp [40], had these facilities (and more). One can, alternatively, add set constructs to functional languages [22, 27, 33, and others]. Or one can go in the opposite direction, starting with Prolog and adding function definitions. Such languages normalize terms before attempting unification, but do not use the definitions to instantiate free variables during goal reduction.

Adding equality to a Horn-clause logic language is trivial, since the axioms of equality (reflexivity, symmetry, transitivity, and functional reflexivity) are Horn. Depth-first search would be useless in this case, but any complete Horn strategy would do. This is in the spirit of Green's [13] original work on extracting answers from resolution proofs. Unfortunately, the axioms for equality lead to hopelessly inefficient computations. To combine a functional or equational programming style with logic programming, one can use an underlying logic of Horn clauses with equality (as an interpreted predicate symbol) and (typed) terms. From this point of view, the most satisfying operational semantics would search for solutions to equations or predicates. Paramodulation (unifying one side of an equation with a nonvariable subterm of a clause and replacing with the other side) is a complete method of handling equality in resolution-based theorem provers, but without a sense of direction to the equations, such an approach does not capture the notion of evaluation present in functional languages. Several language proposals in the early eighties provided interfaces between resolution-based goal reduction and function evaluation, but were inherently incomplete; others were complete but were more like theorem-provers than programming languages.

A "logical" programming language ought to have a simple declarative semantics, and a sound and complete operational semantics. That is, each statement should have a local declarative meaning and each procedural step should follow logically from their collective meaning. Therefore, languages that combine features of functional and logic programming in a unified way, and for which any logically satisfiable goal is solvable, are more appealing, than combinations of the disparate operational mechanisms.

One can use the "in-then-else" construct of functional languages to capture the implications of Horn clauses and use goal solving for "logic variables." alternatively, conditional systems provide a natural bridge between functional programming, based on equational semantics, and logic-programming, based on Horn clauses. See [35].

Narrowing [36, 37] is a "linear" restriction of paramodulation akin to the SLD-strategy in Horn-clause logic [2]. Whereas paramodulation uses both sides of an equation in the same way, narrowing is more directed—unifying with left-hand sides only—thereby taking the direction of rewriting into account. In the conditional case, a rule $l \rightarrow r \ :- \ c$ may be applied to a goal $t[s]$ if a *nonvariable* subterm s unifies with the left-hand side l via most general unifier σ. (Variables in l and t are standardized apart.) The resultant subgoals are $t[r]\sigma, c\sigma$. Solving the condition $c\sigma$ may result in additional variable bindings.

Narrowing can be simulated in Prolog by decomposing terms, as first done in [3]. A logic-programming language with narrowing-like operational semantics was first suggested in [4], but only unconditional rules were used (conditionals were encoded as equivalences); equational Horn clauses were first used in Eqlog [12], RITE [8, 23], and SLOG [10], and have more recently been implemented in Babel [31]. For additional references and comparisons with other suggestions, see the discussion in [8].

To force conditions to be evaluated (or solved) before the branches of a conditional construct, some authors impose a leftmost strategy (e.g. [34]); for conditional systems, one can let narrowing go through only after establishing that the conditions hold (e.g. [8, 10, 12]).

Narrowing and many of its variants are complete mechanisms for generating solutions, in that a solution at least as general as any that satisfies the query can always be found (solutions that are provably equal are considered to be the same). More specifically, with ground confluence, any irreducible solution to a goal can be found by narrowing [21, 25]. (An "irreducible" solution assigns normal forms to each variable.) Limiting one to irreducible solutions is justifiable in a functional setting, since the "values" one is looking for are always constructor terms. The orthogonal approach to equational programming leads to a lazy, outermost narrowing strategy [34], as in the Babel language [31], which is complete in that case.

Alternative narrowing derivations must be explored if completeness is to be assured, since deterministically choosing one possible narrowing over others will not guarantee that solutions will be found. Restrictions and variations of narrowing that do preserve completeness are summarized in [15]. For example, the idea encompassed by the restriction of narrowing to so-called "basic" positions is that when one is looking for irreducible solutions, one can ignore paths that narrow within what was a variable of the original goal. There are more general semantic unification methods as well as refinements of narrowing (see [15]). Top-down methods (e.g. [9, 19]) are particularly appealing. We take the liberty henceforth of referring to all equation-solving methods that make use of confluence by the generic term "narrowing."

Additional superfluous narrowing paths can be avoided by making a distinction between constructor symbols and defined ones (assuming that terms built entirely from constructors are irreducible) [10, for example]. Two terms

headed by different constructors can never be equal; when headed by the same constructor, they are equal if, and only if, their respective arguments are equal. See [15] for more details and references to more refined methods of detecting unsatisfiable goals.

4 Equational-Logic Programming

It would make sense to allow all equational Horn clauses that have *computational* meaning in a program. We should not design a language restricting expressibility of what seems natural just to limit the amount of backtracking that might be necessary (full narrowing as opposed to outermost). Indeed, we see a danger in announcing the narrowing strategy to the programmer, who might be tempted to stray from the strictures of logic, under some assumption of the operational behavior.

With (ground confluent and) terminating rules, there is no need for an lazy evaluation strategy to ensure that a value for a term will be reached. Just as an innermost evaluation is appropriate, a narrowing derivation that mimics it suffices [9]. This narrowing derivation, however, might itself not be innermost. SLOG [10] always chooses the leftmost-innermost narrowing path, and, hence, is complete only in certain situations.

The terminating approach to equational programming suggests a "normalizing" narrowing strategy [7, 8, 16, 19]. This has the advantage of completeness without strong syntactic restrictions, but termination and ground confluence are undecidable properties; thus, completeness is dependent on the programmer's writing a "correct" program. One need not be afraid to predicate completeness on undecidable properties of programs. Ground confluence is a consistency requirement, meaning that different ways of evaluating the same ground term cannot result in distinct values. The correctness of a program is undecidable in any case, so it is the programmer who shoulders the responsibility. Saying that a program may not output an answer if it is nonterminating or nonconfluent is no different from saying to a programmer using an ordinary language that a nonterminating program may produce no output.

Simplification, that is rewriting via terminating rules, is a very powerful feature, particularly when defined function symbols are allowed to be arbitrarily nested in left-hand sides. We suggest user-defined, unbacktrackable, eager ("don't care") simplification prior to each backtrackable lazy ("don't know") narrowing step. Eager simplification without backtracking has the potential of eliminating many otherwise nondeterministic choice points (by eliminating narrowable variables from goals) and leading to dramatically improved performance of functional-logic programs [8, 10]; see [15]. It is, of course, important not to incur heavy costs in searching for applicable rewrites. It is possible to minimize the overhead involved in various ways, including taking advantage of the fact that rewrites have only local impact and rewrites that fail only because of a mismatch with a free variable signify

a potential narrowing [23]. Assuming ground confluence and termination, any strategy can be used for simplification. Narrowing, always preceded by simplification, is complete for ground confluent terminating systems (without "extra" variables appearing in the condition or right-hand side of a rule, but not on the left-hand side). Exactly how much simplification is performed before each narrowing step is a matter of taste, since completeness is not affected by this decision. Narrowing only fully simplified goals has been advocated by [4, 8, 10, 16, 19] and others.

With a combined language, the need for some "added features" is ameliorated. For example, multi-valued functions can be better modeled by predicates than with nondeterministic rewriting, as in [24]. Goal-solving can then find the different values satisfying the relation. Functional dependency is in the syntax. Negation can be handled by incorporating negative information in the form of rewrite rules, which are then used to simplify subgoals to *false* [8, 10, 15]. This approach allows some unsatisfiable goals to be pruned.

The form taken by answers is another issue. Suppose we are given rules for converting a propositional formula e into conjunctive normal form. As a goal, we cannot just write $e =^? z$, since e itself is a solution for z, but is not in the desired form. We could program an explicit predicate, say $nf(z)$, that checked if z is in the desired form, and add it to the goal, or we could impart a "weak" meaning to the equals sign in the goal, namely that they are equal *constructor* terms [8, 15]. Rules for this constructor equality can be generated automatically. A similar approach is to use "directed" goals of the form $e \rightarrow^? z$, similar to the equations in "normal" conditions, meaning that we are looking for a z that is the normal form of e [9].

5 Stream Programming

The desirability of incorporating infinite structures in a functional-logic language has been widely asserted [2, 30]. Unfortunately, we are presented with a tradeoff between the benefits of lazy evaluation of orthogonal, nonterminating rules and those of eager simplification with terminating, nonorthogonal rules.

Starting from the lazy approach, one can allow some additional simplification rules. These rules must be terminating (so that they may be applied eagerly without worry), and should be true in the initial model of the defining equations (so that they preserve uniqueness of normal forms). These requirements are, however, insufficient to guarantee completeness. For example, one cannot just add the terminating and true rule $0 \rightarrow a$ to the orthogonal program $a \rightarrow 0$, since 0 is no longer a normal form. Similarly, lazy narrowing with $a \rightarrow b, b \rightarrow 0$ combined with eager rewriting using $b \rightarrow a$ will not compute the normal form 0 of a. One could add a "shortcut rule" like $inter(y, nil) \rightarrow y$ to the interleave program, since for all ground terms y, the program rewrites $inter(y, nil)$ (in many steps) to y. Situations in which

simplification by terminating rules can be combined with lazy narrowing with nonterminating rules are described in [14].

Starting from a terminating system, one can employ a lazy trick to simulate streams [8, Section 4.5] (cf. tricks for forcing strict evaluation), but the semantics of tricks may not be all that clear.

There is a relatively unexplored alternative: The justification for infinite data structures in a functional setting is that there is no *a priori* way of knowing how much of the structure one will need in any given computation. But in a logic setting—with existentially quantified goals—one need not know in advance how far to go. Instead, one can define a function that computes a fraction of the structure, up to some bound supplied as an argument. Instead of a lazy $from(n) \to n : from(s(n))$, one uses the terminating $from_{s(k)}(n) \to n : from_k(s(n))$. To use the structure, one simply *solves* for a sufficiently large bound. The same idea can be used for interpreters.

Note that the effect of narrowing with the indexed rule is the same as lazy rewriting with the original. The extra parameter stays always just a variable. Furthermore, narrowing creates no extra choice points, since there is only one rule for the stream.

When streams are used on the right-hand side of other rules, as in $g(x, y) \to nth(x, from(y))$, extra variables become necessary. These can be eliminated using the method of [17], for example, giving $g_{s(k)}(x, y) \to nth(x, from_k(y))$. Incorporating bounds can be beneficial when one can use them to cut down on computations that will not in the end contribute (as with the Sieve of Eratosthenes), but in general bounds seem like an unnecessary hassle.

A third option is to isolate cases in which streams can be combined with terminating systems and for which eager simplification plus lazy narrowing of the stream rules works. It can't always work: fair outermost narrowing might not solve $g(a, c(a)) =^? 0$, given the rule $g(x, x) \to 0$ and stream $a \to c(a)$.

So, under what conditions can one combine arbitrary unbacktrackable simplification with lazy narrowing, without explicitly programming the bound? We describe the case of a terminating unconditional rewrite system R and a stream-like rule S. Suppose R is ground confluent and contains no occurrences of the symbol f. Let S contain one (confluent) rule of the form $f(x_1 \ldots x_n) \to e$, where e contains all of the arguments x_i. Stream definitions, like $ones \to 1 : ones$ or $from(n) \to n : from(s(n))$, take that form.

We claim that any normalization strategy that is outermost-fair towards $R \cup S$ is complete when R is left-linear. (Were R orthogonal, this would come as no surprise.) Normalized narrowing will not loop for stream-like examples, since simplification will rewrite a term to the selected element of the stream as soon as that becomes possible.

Suppose s has normal form t. Let S' be S with an extra argument added to each occurrence of f. The f on the left becomes $f_{s(n)}$ and those on the

right become f_n (where n is a new variable). We first show that $R \cup S'$ is terminating. Let $\hat{t}$ denote the S'-normal form of a term t. Since S' is terminating and confluent, the normal form always exists and is unique. Suppose the transformed combination is not terminating and there is an infinite sequence $t_1 \to t_2 \to t_3 \to \cdots$ of rewrite steps. Consider the terms $\hat{t_1}, \hat{t_2}, \ldots, \hat{t_i}, \hat{t_{i+1}}, \ldots$. If t_i rewrites via S' to t_{i+1}, then by confluence of S' we have $\hat{t_i} = \hat{t_{i+1}}$. Since there are no f's in R, it can be shown that if R rewrites t_i to t_{i+1}, then $\hat{t_i}$ goes to $\hat{t_{i+1}}$ in zero or more R steps. Since S' does not destroy any argument x_i of f, we must actually have at least one R step. Thus, were the combined system not terminating, there would exist an infinite sequence with R alone.

Since the combined system is terminating, by Knuth's Critical Pair Lemma [26, Lemma 1.4.11], it is also confluent. Thus any derivation from s leads to the same normal form, call it t'. Since S applies to any subterm headed by an f, the term t, which is in S normal form, cannot contain any f's. There must exist a number N such that if we replace each subterm $f(u_1, \ldots, u_m)$ in s with $f(u_1, \ldots, u_m, s^N 0)$, then any derivation in $R \cup S'$ from this new term s' leads to t. We know that a normalizing outermost strategy for $R \cup S'$ will narrow s' to t (or to something more general). The only catch is that what is fair for S' may be unfair for S (which has enabled positions that S' might no longer have). But at some point R must be able to eliminate all f's to get t. Being left-linear, R cannot see any outstanding f's far below where it applies. This implies that we can use outermost narrowing with simplification by R.

6 Other Aspects

Space limitations prevent us from doing more than listing other desiderata for a equational-logic language of the future.

- If conditional rules are to generalize Horn-clause programming, extra variables should be permitted in conditions. The (declarative) meaning of such a statement is that the consequent holds for any value of the extra variable that satisfies the conditions in which it appears. A program can sometimes be transformed to circumvent this problem [17] by introducing the extra variable into the "head." Alternatively, we can revise the definition of decreasing rewriting in the extra-variable case, and say to insist that solution to the new variables be in *normal form*. Additional useful completeness results are needed ([28] is one such).

- Computing by forward reasoning from the program [4, 8] is a possibility that is particularly attractive in a database setting. Better yet, the recent work of [28] allows for both bottom-up and top-down computing.

- The language must be typed. An example of a language that also incorporates higher-order functions is λ-Prolog [29]. Whether one wants to solve for functional variables needs to be resolved.

- The language should have mechanisms for information hiding and inheritance; an "order-sorted" logic underlies Eqlog [12], for example.

- Other than for infinite data structures and coding interpreters, it is hard to imagine good uses of nonterminating systems. An exception may be "fairly" terminating systems like a gcd program with a rule $gcd(x, y) \rightarrow gcd(y, x)$. Complete methods are wanting.

- Ventures into the realm of non-Horn reasoning are always tempting. Besides negation, limited disjunction may be possible [8, 27].

- Constraint programming is an important development and may be a more comfortable setting in which to incorporate reasoning with directed equalities.

- In an equational language, it may not be clear how individual equations should be oriented. Existing methods for establishing termination and confluence are geared to the significantly simpler case of unconditional rules. An alternative would be to use forms of ordered Horn-clause resolution like [5, 28]. They could allow for a measure of direction in the form of an ordering (the "control" part) supplied to the prover by the programmer along with unoriented equational Horn clauses (the "logic" part).

- Rather than provide an explicit ordering, a futuristic alternative may be to have the user supply a definition of the "shape" of the desired normal forms, and let the system choose an ordering that shows progress towards the desired forms with each step of rewriting. For example, given various propositional identities and a definition of conjunctive normal form, the system should push negations down to the literal level and disjunctions down to clauses.

- The efficient implementation of combined languages is, of course, of paramount importance. Some of the recent work is discussed in [15].

- Rewriting and narrowing lend themselves, in a natural fashion, to "and-" and "or-" parallelism, respectively. One (simulated) parallel implementation of this combination is reported on in [6].

References

[1] Marco Bellia and Giorgio Levi. The relation between logic and functional languages: A survey. *J. of Logic Programming*, 3(3):217–236, October 1986.

[2] Pier Giorgio Bosco, Elio Giovannetti, and Corrado Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[3] P. Deransart. An operational algebraic semantics of PROLOG programs. Internal report, Institut National Recherche en Informatique et Automatique, Le Chesnay, France, 1983.

[4] Nachum Dershowitz. Computing with rewrite systems. Technical Report ATR-83(8478)-1, Information Sciences Research Office, The Aerospace Corp., El Segundo, CA, January 1983.

[5] Nachum Dershowitz. Ordering-based strategies for Horn clauses. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 118–124, Sydney, Australia, August 1991.

[6] Nachum Dershowitz and Naomi Lindenstrauss. A parallel implementation of equational programming. In *Proceedings of the Fifth Jerusalem Conference on Information Technology*, pages 426–435, Jerusalem, Israel, October 1990. IEEE Computer Society.

[7] Nachum Dershowitz, Subrata Mitra, and G. Sivakumar. Equation solving in conditional AC-theories. In H. Kirchner and W. Wechler, editors, *Proceedings of the Second Conference on Algebraic and Logic Programming (Nancy, France)*, volume 463 of *Lecture Notes in Computer Science*, pages 283–297, Berlin, October 1990. Springer-Verlag.

[8] Nachum Dershowitz and David A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford Press, Oxford, 1988.

[9] Nachum Dershowitz and G. Sivakumar. Solving goals in equational languages. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems (Orsay, France)*, volume 308 of *Lecture Notes in Computer Science*, pages 45–55, Berlin, July 1987. Springer-Verlag.

[10] Laurent Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the Symposium on Logic Programming*, pages 172–184, Boston, MA, July 1985. IEEE.

[11] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In Michaelson and R. Milner, editors, *Proceedings of the Third EATCS International Colloquium on Automata, Languages and Programming*, pages 257–284, Edinburgh, Scotland, 1976. Edinburgh University Press.

[12] Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[13] C. Cordell Green. *The Application of Theorem-Proving to Question-Answering*. Ph.d., Department of Computer Science, Stanford University, Stanford, CA, 1969. Reprinted by Garland, New York [1980].

[14] Michael Hanus. Combining lazy narrowing and simplification. In *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming (Madrid, Spain)*, volume 844 of *Lecture Notes in Computer Science*, pages 370–384, Berlin, 1994. Springer-Verlag.

[15] Michael Hanus. The integration of functions into logic programming: From theory to practice. *J. Logic Programming*, 19&20:583–628, 1994.

[16] Michael Hanus. Lazy unification with simplification. In *Proc. 5th European Symposium on Programming (Edinburgh, Scotland)*, volume 788 of *Lecture Notes in Computer Science*, pages 272–286, Berlin, Germany, 1994. Springer.

[17] Michael Hanus. On extra variables in (equational) logic programming. In *Proc. Twelfth International Conference on Logic Programming*, pages 665–679. MIT Press, 1995.

[18] Carl Hewitt. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Technical report, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, February 1972. AI-TR-258.

[19] Steffen Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1989.

[20] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[21] Jean-Marie Hullot. Canonical forms and unification. In R. Kowalski, editor, *Proceedings of the Fifth International Conference on Automated*

Deduction (Les Arcs, France), volume 87 of *Lecture Notes in Computer Science*, pages 318–334, Berlin, July 1980. Springer-Verlag.

[22] Bharat Jayaraman and Frank S. K. Silbermann. Equations, sets, and reduction semantics for functional and logic programming. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 320–331, Cambridge, MA, August 1986.

[23] N. Alan Josephson and Nachum Dershowitz. An implementation of narrowing. *J. Logic Programming*, 6(1&2):57–77, January/March 1989.

[24] Stéphane Kaplan. Rewriting with a nondeterministic choice operator. In B. Robiinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming (Saarbücken, F.R.G.)*, volume 213 of *Lecture Notes in Computer Science*, page 351ff., Berlin, March 1986. Springer-Verlag.

[25] Stéphane Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *J. Symbolic Computation*, 4(3):295–334, December 1987.

[26] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

[27] John W. Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Logic Programming Symposium*, Cambridge, MA, 1994. MIT Press.

[28] Christopher Lynch. Oriented equational logic programming is complete. Unpublished report, Centre de Recherche en Informatique de Nancy, Nancy, France, 1995.

[29] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simle unification. In P. Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming (Tübingen, F.R.G.)*, volume 475 of *Lecture Notes in Computer Science*, pages 253–280, Berlin, December 1989. Springer-Verlag.

[30] Juan Jose Moreno-Navarro. Expressivity of functional-logic languages and their implementation. In M. Alpuente and R. Barbuti, editors, *Joint Conference on Declarative Programming GULP-PRODE'94*. GULP (Italian ALP Chapter), Universidad Politecnica Valencia, Servicio de publicaciones Universidad Politecnica de Valencia, September 1994.

[31] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo. Logic programming with functions and predicates: The language Babel. *J. Logic Programming*, 12:191–223, 1992.

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

[33] Michael J. O'Donnell. Equational logic programming. In Dov Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, *Logic Programming*, chapter 2. Oxford University Press, to appear.

[34] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the Symposium on Logic Programming*, pages 138–151, Boston, MA, July 1985. IEEE.

[35] Uday S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 3–36. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[36] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *J. of the Association for Computing Machinery*, 21(4):622–642, 1974.

[37] V. F. Turchin. Equivalent transformation of recursive functions defined in the language Refal. In *Sympos Teoria Yazykov i Metody Prostroenia System Programirowania*, pages 31–42. Alushta-Kiev, 1972. in Russian.

[38] David A. Turner. SASL language manual. Technical report, University of Kent, Canterbury, U.K., 1976.

[39] David A. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[40] B. Michael Wilber. A Qlisp reference manual. Technical note 118, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA, March 1976.