Towards a Broader Basis for Logic Programming†

Bharat Jayaraman

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
U.S.A.

Telephone: (716) 636-3180/3194
E-Mail: bharat@cs.buffalo.edu

Abstract

Logic programming is generally taken to be synonymous with relational programming; however, this paper argues why a broader basis for logic programming is needed, why existing combinations of equations, relations, and functions do not entirely meet the need, and a promising approach for the broader basis. The proposed broader basis consists of three logical forms: equations, relations, and subset assertions, along with an accompanying set of matching and unification operations. Equations provide the ability to define deterministic operations without Prolog's *cut*. Subset assertions can be used to define more declaratively Prolog's *setof* as well as transitive-closure operations, both of which are traditionally expressed using *assert* and *retract*. Because functions defined by subset assertions must be invoked with ground arguments, formulating relational clauses by subset assertions allows one to declaratively specify which arguments of a relation are ground, thereby obviating the need for mode declarations. Several examples are given to illustrate the approach. It is shown that this broader language is amenable to a very efficient implementation; in many cases, it will be potentially more efficient than Prolog. A language called **SuRE**—for **Su**bsets, **R**elations, and **E**quations—is being implemented based on these ideas.

† This research was supported in part by grant DCR-8603609 from the National Science Foundation.

1. Why is a Broader Basis Needed?

Prolog has been immensely successful as a programming language, but it has too many defects to be considered successful as a *logic* programming language. Prolog departs from the ideal of logic programming in important ways, especially by having an unsound and incomplete proof system, but also by allowing the programmer to indicate control information explicitly (the notorious *cut*) and perform global side-effects (using *assert* and *retract*). Not only do these features undermine the logical nature of the language, they also make it difficult to run Prolog programs in parallel; also, many static analysis techniques depend crucially on the absence of these extra-logical features. Certain other features of Prolog are also incorrect from a logical standpoint, e.g. Prolog's *not* operator, or do not have a clear formulation in logic, e.g. the *setof* construct for collecting all solutions to a query. The presence of meta-logical operations, such as *var* and *nonvar*, also make programs hard to reason about.

The underlying thesis in this paper is that the presence of extra-logical features in Prolog and the need to make semantic compromises for the sake of efficiency (i.e., unsound and incomplete implementations) are indications of the limitations of the relational paradigm as an *exclusive* basis for logic programming. A substantial portion of most Prolog programs is deterministic [B89, DW89]; hence, these portions could be formulated more clearly using an equational or functional style without the *cut* to specify determinacy—note that clause-indexing obviates only some uses of the *cut*. The equational or functional style is also preferable when a relation can be meaningfully run in only a certain *mode*—as a function from certain input arguments to certain output argument, e.g., sorting a list of numbers or summing the leaves of a tree. Many uses of *assert* and *retract* correspond to implementations of transitive-closure operations, memoization, or collecting results from alternative search paths, as in the *setof* construct. In an earlier paper [JP89], it was shown how subset assertions allow a more declarative formulation of such uses. Thus the goal of this research is to show how an integrated language based on equations, subsets and relations can serve as a declarative and efficient substitute for many uses of the impure features of Prolog. The objective here is not to replace the relational paradigm, but to supplement it with logical forms that are amenable to correct implementation without semantic compromises.

The remainder of this paper is organized as follows. Section 2 briefly surveys previous work on equations, functions and relations, and shows why they do not completely suffice as a broader basis for logic programming. Section 3 describes the basic approach, summarizing and clarifying our earlier papers [JP87, JN88, JP89]. Section 4 present several examples that elucidate how this broader language avoids many common uses of the extra-logical operations. Section 5 is devoted to conclusions, status, and further work.

2. Related Work and Desiderata

Relational programming is not the only form of logic programming; as alluded to earlier, equations and functions are possible approaches. Below is a brief survey of the major logic programming paradigms that have been proposed—see references [BL86, DL86] for a survey of several specific approaches—and a discussion of why they do not completely suffice as a broader basis for logic programming. In short, they are based on features that are either not amenable to efficient compilation or they do not address how the extra-logical features such as *cut*, *assert*, *retract*, *setof*, etc., can be avoided.

- **Equations** [O85]

The use of *equations* as a logic programming language has been advocated by O'Donnell [O85]. An important restriction on equational queries is that they may not contain any variables, i.e., they are *ground* queries. Thus, it is not necessary to use unification; a simpler, "one-way" matching suffices. If the equations are regular, i.e., left-linear and non-overlapping, a form of leftmost outermost rewriting strategy is a sound, complete and efficient implementation. Thus, deterministic computations can be formulated in this framework without resort to extra-logical operations such as the *cut*. However, this paradigm of equations is too restrictive to serve as a complete basis for logic programming.

- **Relations and Equations** [GM84, JLM84, DP85]

It seems natural to combine the efficiency of equations with the programming expressiveness of relations. However, existing combinations [GM84, JLM84, DP85] of these two logical forms extend the power of ordinary unification to *unification modulo an equational theory*, also known as *E-unification* or *narrowing*. A thorough analysis of E-unification is given by Gallier and Raatz [GR86]. While the E-unification approach is very elegant from a theoretical standpoint, it is very difficult to realize in practice, even for the "well-behaved" case [GR86] where the program is partitioned into a set of relational clauses and a set of equations such that an equational term never appears in the head of a conditional clause. Thus, it is very important for the compiler to know in advance the equational theory with respect to which unification must be performed, so that it can make necessary optimizations.

- **Conditional Equations** [DP85, F84, LP87, J88, DO88]

A different way to enhance the equational programming paradigm is to use equations not just for simplifying (or reducing) an expression, but also for solving equational goals of the form $expr_1 = expr_2$. The term *conditional equations* was introduced by Dershowitz and Plaisted [DP85] to refer to programs of the form $e = f$:- p, where p is in general a conjunction of equational or relational goals. A relational clause p :- q, r can be viewed as a conditional equation $p = true$:- $q = true$, $r = true$, and thus conditional equations provide a uniform approach to equational and relational programming. As noted in [GR86], solving equational goals in these languages is essentially the E-unification problem, although approaches such as EqL [J88] and K-LEAF [LP87] make the problem more tractable by distinguishing constructors and functions. An important point to note about all these approaches is that, by allowing equations to be used for both reduction and narrowing, a compiler cannot easily detect when an equational goal should be reduced and when it should be narrowed. Without this distinction, the potential efficiency of using equations is lost.

- **Functions and Relations** [AN87]

While functional programming languages, such as Miranda [T85], do provide the same advantages as equational languages, the problems with relational programming are not addressed by integrating functional and relational programming. The reason for this claim is that a proper integration of these two forms must account for the higher-order and infinite objects of functional languages. The problem with this combination is that using functional expressions within the relational sublanguage requires equality to be defined over infinite and higher-order objects. The needed higher-order unification here, unlike that in λProlog [NM88], is not even recursively eumerable. This problem is handled in Le Fun [AN87] by failing when two higher-order objects are tested for equality. While this might be a practical approach, it should be noted that this inference is, strictly speaking, unsound. However, one important feature of Le Fun is worth noting: to obtain the efficiency of functions, they should be used primarily for non-backtrackable execution.

- **Desiderata for a Broader Logic Programming Language**

In view of the computational difficulties with the aforementioned combinations of functions, relations and equations, we propose the criteria in designing a broader logic programming language:

(i) E-unification with respect to an arbitrary user-supplied theory should not be supported; it is too difficult to implement efficiently. (The success of languages like CLP [JL87] and λProlog [NM88] lend support to this criteria.)

(ii) Relations cannot be combined with functions without compromising the correctness of equality; they can be combined with equations because the latter do not contain higher-order or infinite objects.

(iii) Equational languages should make the distinction between constructors and other function symbols, in order to facilitate efficient implementation (especially for narrowing).

(iv) When equational goals occur in relational clauses, the compiler should be able to detect in most cases when the goal requires reduction and when it requires narrowing.

Even with these restrictions, there are substantial technical problems to be solved before reaching a practical system.

3. Proposed Approach: Subsets, Relations and Equations

The language to be described builds upon our earlier work on combining subsets, equations and relations [JP89]. We described two paradigms in these papers: subset-equational programming (combining subsets and equations) and subset-relational programming (combining subsets and relations). We also gave their formal model-theoretic and operational semantics. This paper explores the integration of these two paradigms and shows how the resulting language can help avoid many of the extra-logical features in Prolog. This paper describes the integrated language informally, using several examples. A formal description of the language is easy to give—it is essentially a consolidation of our earlier semantic descriptions [JP89, J90]—but this is not the main goal of this paper. Before describing the integrated language in section 4, we briefly review the main features of subset-equational and subset-relational programming.

3.1 Subset-Equational Programming

A program assertion in this paradigm may take one of two forms†:

† The implementation [N88] actually uses the keyword `contains` instead of $\supseteq$, and the keyword `phi` instead of ϕ

$$f(\textit{terms}) = \textit{expression}$$
$$f(\textit{terms}) \supseteq \textit{expression}$$

where *terms* correspond to the data objects of the language, and are built up from atoms, variables, and constructors, whereas *expressions* may additionally contain user-defined functions. In addition to the usual first-order terms of Prolog, the language also provides set-terms and set matching. By providing subset assertions with a *collect-all* capability, the meaning of a set-valued function *f* operating on ground terms is equal to the union of the respective sets defined by the different subset assertions for *f*. The top-level query is of the form

? *ground-expr.*

The meaning of this query is the ground term g such that *ground-expr* $= g$ is a logical consequence of the *completion* of the program, i.e., augmenting all subset assertions defining some function with equality assertions that capture the *collect all* capability of these subset assertions. Because the declarative semantics makes a distinction between terms and expressions, i.e., variables may be quantified only over terms rather than expressions, it can be shown that depth-first search and innermost reduction are *not* semantic compromises, but indeed correct implementation techniques for the language [J89].

In order that program assertions define a unique outcome for any query expression, we require for *equational assertions* (i) that the left-hand side not overlap with the left-hand side of any other assertion, and (ii) when set constructors occur on the left-hand side, the result should be independent of which one of the potentially many matches is selected. Other less restrictive conditions are possible, but we shall assume the above conditions, for the sake of specificity.

• **Set Matching**: Because arguments to functions are ground terms, function application requires matching, rather than unification. The matching algorithm is a form of *set matching* because of the presence of set constructors: ϕ (empty set), $\{\_\}$ (singleton set), or $\{\_ \mid \_\}$. The notation $\{x \mid s\}$ refers to a set in which x is one element and t is the remainder of the set. This form of set matching reduces to a-c matching if the pattern $\{x \mid s\}$ is treated as $\{x\} \cup s$ where only the associative and commutative properties of $\cup$ are used in the matching. When set patterns appear on the l.h.s. of a subset assertion, each of the potentially many matches must be separately considered in reducing the r.h.s. expression to a resulting set, and the union of all such resulting sets must be formed.

• **Simple Subset-Equational Programs**:

$\texttt{crossproduct}(\{\texttt{x} \mid \_\}, \{\texttt{y} \mid \_\}) \supseteq \{[\texttt{x} \mid \texttt{y}]\}$

$\texttt{intersect}(\{\texttt{x} \mid \_\}, \{\texttt{x} \mid \_\}) \supseteq \{\texttt{x}\}$

$\texttt{perms}(\phi) = \{[\ ]\}$

$\texttt{perms}(\{\texttt{x} \mid \texttt{t}\}) \supseteq \texttt{distr}(\texttt{x}, \texttt{perms}(\texttt{t}))$

$\texttt{distr}(\texttt{x}, \{\texttt{h} \mid \_\}) \supseteq \{[\texttt{x} \mid \texttt{h}]\}$

(Our convention is to begin variables with lower-case letters and constants with upper-case letters.) Often definitions can be stated in a compact, non-recursive manner using subset assertions and set-terms, because much of the iteration over sets is moved into the matching process. In `crossproduct`, `intersect` and `distr`, there is no need to explicitly indicate the case when the argument sets are empty; the result is the empty set by virtue of an emptiness-as-failure assumption. The reference [J90a] also discusses the use of subset-equational programming in representing rules of an expert system, and shows how the collect-all capability is particularly useful in defining the results of exhaustive backchaining systems such as MYCIN [BS84].

• **Stratified Subset-Equational Programs**: We also showed how the paradigm of subset-equational programming can be easily extended to also support *closure functions*, i.e., functions that define transitive-closure sets. In general, closure functions are useful in many areas, especially dataflow analysis in compilers. In order to give a simple declarative semantics and a corresponding correct operational semantics in terms of *memo-tables* [M68] (or extension tables), we introduced the class of *stratified subset-equational programs* [JP89, JP90]. An example illustrating closure functions is the program below for finding the set of reachable nodes of a graph, represented as a set of ordered pairs, starting from some given node.

```
reach(v, g) ⊇ adjacent(v, g)
reach(v, g) ⊇ allreach(adjacent(v, g), g)
allreach({x | _}, g) ⊇ reach(x, g)
adjacent(v, {[v, w] | _ }) ⊇ {w}
```

If the above program is treated as a simple subset-equational program, it will be nonterminating when run on an input graph that has a cycle. If, on the other hand, the programmer identified the definitions `reach` and `allreach` as closure functions, using some form of program annotation, it is possible to avoid the infinite loop using a memo-table. There is a need to identify the closure functions, so that memoization is restricted to those cases where it is needed.

In general, *stratified subet-equational programs* consist of closure and non-closure functions partitioned into several levels, such that all closure functions at a given level are defined in terms of one another using *subset-monotonic* functions, but may be defined in terms of any (closure or non-closure) function from a lower level. A set-valued function g is said to be *subset-monotonic* in a particular argument iff $\mathrm{s}_1 \subseteq \mathrm{s}_2$ imples $g(\ldots, \mathrm{s}_1, \ldots) \subseteq g(\ldots, \mathrm{s}_2, \ldots)$. This restriction on the composition of closure functions is needed in order to give a provably correct operational semantics in terms of memo-tables.

3.2. Subset-Relational Programming

Subset-relational programming is a paradigm of programming with subset and relational assertions. This work is relatively more recent than the preceding work on subset-equational programming. In simple subset-relational programming, only relational goals may occur in the body of an assertion; in stratified subset-equational programs, equality goals may also occur in the body provided they refer to clauses from a lower strata. The general forms of simple subset-relational programs are as follows:

$$\mathtt{f}(terms) \supseteq s$$

$$\mathtt{f}(terms) \supseteq s \quad \texttt{:-} \quad \mathtt{p}_1(terms), \ldots, \mathtt{p}_n(terms)$$

$$\mathtt{p}(terms)$$

$$\mathtt{p}(terms) \quad \texttt{:-} \quad \mathtt{p}_1(terms), \ldots, \mathtt{p}_n(terms)$$

Note that these definite clauses can contain set-terms. The declarative meaning of a subset assertion is that, for all its ground instantiations, the function `f` operating on ground terms contains a ground set s if the condition in the body, $\mathtt{p}_1(terms), \ldots, \mathtt{p}_n(terms)$, is true. By incorporating a collect-all capability, the result of a query such as

$$\mathtt{f}(ground\text{-}terms) = term$$

is obtained by first collecting all solutions to the expression $\mathtt{f}(ground\text{-}terms)$ and then unifying the result with $term$. A top-level query may also be of the form $\mathtt{p}(terms)$. The operational semantics uses set matching with subset assertions and, because clauses may have set-terms, *set unification* with relational assertions. Arguments to a function defined by subset assertions must be ground, and each derivation must be terminating (either successful or finitely-failed) in order to obtain a result for the equational goal shown above. Furthermore, there should be only a finite number of derivations, because all sets in this framework are finite. Note that these conditions are analogous to those needed for the correctness of negation-as-failure in relational programming [L87]. A few examples will serve to illustrate the paradigm.

• **Setof**: Simple subset-relational programs can be used to simulate Prolog's `setof` feature in a more declarative manner. For example, given the usual `append/3` relation,

```
append([ ], X, X)
append([H|T], Y, [H|Z])  :- append(T, Y, Z)
```

the Prolog goal `setof([X|Y], append(X,Y, [1,2,3]), Answer)` for defining the different partitions of the list `[1,2,3]` may be expressed by the following subset assertion and query.

```
parts(list) ⊇ {[x|y]}  :- append(x, y, list)
?  parts([1,2,3]) = answer.
```

• **Set-terms in Relations**: The use of set terms in relations makes possible some interesting definitions. The member relation on sets, called `member2` below, can be used to verify set membership in the usual way.

```
member2(x, {x | _}).
```

The above predicate can also be used to insert an element in a set, if not already present, via a goal such as

```
?  member2(10, s).
```

Unlike the case with the Prolog `member` predicate, there is only *one solution* to the above goal, i.e., `s` $\leftarrow$ {`10` | _}. The `member2` predicate may be successively invoked to add an element to `s` if not already present.

Below is another definition of the permutations of the elements of a set.

```
set-to-list(φ, [ ])
set-to-list({x | s}, [x | t]) :- set-to-list(s, t)
permutations(set) ⊇ {list}  :- set-to-list(set, list)
```

Because the argument to `permutations` is assumed to be ground, the first argument in the invocation of `set-to-list` in the body of `permutations` will also be ground. Because the matching of an n-element set against the second assertion for `set-to-list` would yield n different matches, each of these matches is separately considered in recursively reducing the body of `set-to-list`. In this manner, all permutations to a top-level invocation of `permutations` are computed.

• **Negation-as-Failure:** Negation-as-failure is also compatible with this paradigm. For example, the set-difference of two sets can be stated as follows.

`diff(s1, s2)` $\supseteq$ `{d} :- member2(d, s1), not member2(d, s2).`

Each solution to `d` will be used to instantiate the set {`d`}, so that the result of a call such as

?`diff`({1, 2, 3, 4}, {1, 3, 5, 6}) = `answer`

will be the set {2, 4}. Note that the meaning of negation-as-failure in this context is more general than that in Prolog. In general, `not` G will succeed when the search tree for G is finitely failed after considering the derivation from each clause as well as all possible choices (due to the set pattern {`x` | `t`}) within any particular clause.

In [JP89], we discussed stratified subset-relational programs, where it is permissible to have equality goals on the r.h.s. of a subset or relational assertion, provided such goals refer to subset assertions from a lower strata. This makes possible multi-level collect operations.

A number of researchers have advocated the use of sets for functional and logic programming. In comparison with these approaches, the main significance of the proposed approach is that very concise, clear and efficient set operations are possible when they are formulated with subset assertions and much of the iteration over sets is moved into the matching process. Furthermore, subset assertions combine in a natural way with equational and relational assertions, thus making possible a broader basis for logic programming.

4. SuRE: A language integrating Subsets, Relations and Equations

In this section, we will examine the integration of subset-equational and subset-relational programming in a language called SuRE. It subsumes the above two paradigms, and, also permits equational and subset goals in the bodies of subset and relational assertions. An equational goal is of the form

`f`(*terms*) = *term*,

where the function f can be defined either through equational or subset assertions, and the arguments to f must be ground at the time of invocation. That is, equational goals will be used for pure reduction, not narrowing. An extension of the paradigm to allow narrowing will be considered in future work, but, as noted in section 2, this extension will be considered successful only if a distinction between reduction and narrowing can be made at compile-time. A subset goal is of the form

$\texttt{f}(\mathit{terms}) \supseteq \{\mathit{term}\}$,

where, once again, f obeys the same restrictions as in an equational goal. A subset goal specifies the enumeration of one element at time from the set defined by f, and specifies a limited form of lazy evaluation with the added benefit that intermediate sets are avoided—this capability has been implemented in simple subset-equational programs, but was used in the limited context when it is known that a function distributes over union in a certain argument position [JP87, JN88, J90]. One possible extension would be to allow a goal of the form $\texttt{f}(\mathit{terms}) \supseteq \mathit{set\text{-}term}$, in order to allow more general lazy evaluation. The implementation issues of this feature have not yet been examined, hence this extension is not considered further in this paper.

A few examples will serve to illustrate the broader framework.

4.1. Transitive Closures

Two examples will illustrate the integration of stratified subset-equational and stratified subset-relational programming. It is desirable to re-formulate the `reach` definition of section 3.1 so that the graph is specified by an `edge` relation rather than a set of ordered pairs. This way the graph will not be entered in the memo-table and we minimize the cost of memo-table search.

```
reach(v) ⊇ adjacent(v)
reach(v) ⊇ allreach(adjacent(v))
allreach({x | _}) ⊇ reach(x)
adjacent(v) ⊇ {w} :- edge(v, w)
edge(1, 2).      edge(2, 3).      edge(3, 4).      edge(4, 1).
```

The above definitions should be stratified into three levels: `edge` at level 1, `adjacent` at level 2, and `reach` and `allreach` at level 3. The closure functions are, as before, `reach` and `allreach`, and must be declared through suitable annotations. It is well-known that defining transitive closures in Prolog efficiently requires the use of `assert` and `retract` [O88]. It should be mentioned that nested expressions in the body of a subset (or an equational) assertion are flattened to reflect the innermost reduction order. For example, the second assertion for `reach` is syntactic sugar for the following:

```
reach(v) ⊇ {s2}  :-  adjacent(v) = s1, allreach(s1) = s2.
```

The following example illustrates the integration a little better; it is a program defining the *reaching definitions* in a program flow graph, which is computed by a compiler during its optimization phase [AU77]:

```
out(b) ⊇ diff(in(b), kill(b))
out(b) ⊇ gen(b)
in(b) ⊇ allout(pred(b))
allout({p | _}) ⊇ out(p)
```

where `kill(b)`, `gen(b)`, and `pred(b)` are pre-defined set-valued functions specifying the relevant information for a given program flow graph and basic block `b`. The closure functions in the above example are `in`, `out`, and `allout`, and these must be declared by annotations. The set-difference function, `diff`, which was defined earlier, is subset-monotonic in its first argument, and hence its use in defining the closure function `out` is correct.

4.2. A More Thorough Integration

The following formulation of the N queens problem makes use of equational, subset and relational assertions as well as equational, subset and relational goals.

```
solve(n) ⊇ {answer} :-
           n > 0,
           solvefrom(1, n, φ) ⊇ {answer}
solvefrom(col, n, safeset) =
           if col > n then {safeset} else placequeen(col, n, safeset)
placequeen(col, n, safeset) ⊇ {answer} :-
           gen(n) ⊇ {row},
           not attacking(pos(col, row), safeset),
           solvefrom(col+1, n, {pos(col, row) | safeset}) ⊇ {answer}
gen(n) = if n = 1 then {1} else {n | gen(n - 1)}.
```

```
attacking(pos(c1, r1), {pos(c2, r2) | _}) :-
            overlaps(c1, r1, c2, r2) = true

overlaps(c1, r1, c2, r2) =
            (r1 = r2) or abs(c1 - c2) = abs(r1 - r2)
```

There are several noteworthy points about the above program:

(i) By using subset assertions rather than relational assertions to define `solve` and `placequeen`, we have made explicit those arguments that are intended to be ground; thus, subset assertions are a declarative alternative to mode declarations.

(ii) Subset goals such as $\mathtt{solvefrom}(1, \mathtt{n}, \phi) \supseteq \{\mathtt{answer}\}$ allow the selection of one element at a time from the generated set. If we are interested in collecting all elements of the set, we would use the equational goal $\mathtt{solvefrom}(1, \mathtt{n}, \phi) = \mathtt{answer}$.

(iii) Equational assertions such as `solvefrom`, `overlaps` and `gen` make available the familiar expression notation and `if-then-else` for greater clarity and efficiency.

(iv) Negation-as-failure is also compatible with this broader framework.

(v) Because functions defined by equational and subset assertions must be invoked with ground terms, the compiler can make use of this knowledge to generate more efficient code. Propagation of groundness information at compile-time is greatly facilitated by the presence of equational and subset assertions. In the above example, the compiler can verify that the arguments to all equational and subset goals are ground, and also can simplify expressions such as `n > 0` and `col+1`.

4.3. Nonground Terms

The previous examples did not make use of nonground terms in any essential way. The example below for sorting a list of numbers shows the use of nonground terms in conjunction with equations and relations.

```
sort(list) ⊇ {answer } :-
                    dsort(list, [answer | [ ]]).

dsort([ ], [x|x]).
dsort([mid | list], [sorta | tail])  :-
                    part(list, mid) = [left | right],
                    dsort(left, [sorta | [mid | sortb]]),
```

```
                     dsort(right, [sortb | tail]).
part([ ], m) = [[ ] | [ ]].
part([h | t], m) = if m>h then [[h | a] | b] else [a | [h | b]]
                     where
                          part(t, m) = [a, b].
```

The above program takes an ordinary list of numbers as input to `sort`, but employs difference-lists in the body of `dsort` for efficient concatenation of the two sorted sublists. As in the case of `solve` in section 4.2, using a subset assertion rather than a relational assertion for `sort` allows us to indicate which arguments are expected to be ground. Because equational assertions must be invoked with ground arguments and must return ground results, the compiler can infer that the first argument of `dsort` is always ground. Consequently, the compiler can also determine that omission of the occurs-check for the second argument of `dsort` is safe. (In Prolog, omission of occurs-check when working with difference lists can give rise to wrong answers [L87].)

Since `sort` is deterministic, a typical goal using `sort` would be

`sort(`*ground-list*`) = {answer}`,

indicating thereby that the resulting set is expected to have only one solution. In view of its determinacy, one might be tempted to define `sort` using the *equational* assertion

```
sort(list) = answer  :- dsort(list, [answer | [ ]]).
```

However, the use of relational goals in equational assertions is not legal in SuRE, because it is not possible in general to guarantee that the relational goal in the body will not produce more than one solution. (This is precisely the restriction of "well-behavedness" referred to in [GR86].)

5. Conclusions and Further Work

The many shortcomings of Prolog when viewed as a pure logic programming language are well-known. The need for a pure logic programming language is based on pragmatic reasons; considerable research has gone into static analysis of logic programs as well as parallel execution, but the success of this work depends crucially on the availability of a declarative language. The underlying premise in this paper is that relational programming is not sufficient to serve as an exclusive basis for logic programming. The paper contains a survey of other possible approaches, such as equations, conditional equations, combinations of relations and equations, functions and relations, etc., and explains why they do not suffice. They are either not amenable to efficient compilation or they do not address how the extra-logical features such as *cut*, *assert*, *retract*, *setof*, etc., can be avoided.

The proposed broader logic programming uses a combination of equational, subset and relational assertions. Equational assertions are intended primarily for deterministic computations, and would serve to obviate many uses of the *cut*. They will be used for exclusively for reduction (not narrowing), and hence would be amenable to efficient compilation. The most novel feature in this work is the subset assertion, which was described in my earlier papers. Subset assertions combine well with both equational and relational assertions: When combined with equations, they yield very concise and efficient formulation of set operations; they also can be used to specify transitive-closure operations, which are a declarative analog of many uses of memo-tables. When subset assertions are combined with relations, they provide a declarative treatment for Prolog's *setof* construct, which corresponds to many uses of *assert* and *retract*. By specifying a relational clause as a subset clause, one can state which arguments are intended to be ground; thus subset assertions are a declarative alternative to mode declarations. The ability to treat these features in a declarative manner is an important advance of the proposed approach over previous work. Several examples are presented as evidence of the declarativeness and potential efficiency of the approach.

An implementation for simple subset-equational programs has been developed [J90], but not for the other features. The next step in this research is a more thorough description of the **SuRE** language along with a complete semantic definition. This will be followed by a sequential implementation of the language based on an extension of the WAM. As mentioned in the paper, the compiler for this language must make essential use of the static analysis techniques, especially groundness analysis. Groundness analysis is required for an efficient implementation of negation-as-failure, hence the required mechanisms are not any more complex than those already needed.

Finally, it should be noted that this paper does not claim to have a declarative and efficient solution to *all* reasonable uses of the non-logical features of Prolog, although many frequently occurring uses of these features can be handled with equations and subset assertions. A systematic study of several real Prolog programs is being conducted to discover cases that are not easily accomodated, which in turn would serve as a guide to further refining the language. Many problematic features remain unaddressed in this paper, e.g., input-output and the meta-logical predicates. The challenge, in my opinion, is not to formalize the current uses of predicates like *var* and *nonvar*—since programs using these features are hard to reason about—but to devise constructs that would obviate (at least the frequently occurring) uses of these features.

References

[AN87] H. Ait-Kaci and R. Nasr, "Le Fun: Logic, equations and Functions," In *1987 IEEE Symposium on Logic Programming*, San Francisco, pp. 17-23, 1987.

[AU77] A. Aho and J.D. Ullman, "Principles of Compiler Design," Addison-Wesley, 1977.

[B89] D. Bowen, "Determinacy Checking: Choice Points Considered Harmful," In *Workshop on Logic Programming Environments*, Cleveland, Ohio, October 1989, pp. 17-33.

[BL86] M. Bellia and G. Levi, "The Relation between Logic and Functional Languages: A Survey," In *J. of Logic Programming*, vol. 3, pp.217-236, 1986.

[DL86] D. DeGroot and G. Lindstrom, "Logic Programming : Functions, Equations and Relations," Prentice-Hall, 1986.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, pp. 54–66.

[DW89] S.K. Debray and D.S. Warren, "Functional Computations in Logic Programs," In *ACM TOPLAS*, vol. 11, no. 3, pp. 451-481, July 1989.

[GM84] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.*, Vol. 2, pp. 179–210, 1984.

[GR86] J.H. Gallier and S. Raatz, "SLD-Resolution Methods for Horn Clauses with Equality based on E-Unification," In *IEEE Symp. on Logic Programming*, Salt Lake City, 1986, pp. 168-179.

[J88] B. Jayaraman, "Semantics of EqL," In *IEEE Transactions on Software Engineering,* vol. 14, no. 4, pp. 372-380, April 1988.

[J89] B. Jayaraman, "Subset-Equational Programing," Submitted for journal publication.

[J90] B. Jayaraman, "Implementation of Subset-Equational Programs," To appear in *Journal of Logic Programming.*

[J90a] B. Jayaraman, "Subset-Equational Programming in Intelligent Decision Systems," Invited paper for a special issue of *Journal of Computers and Mathematics with Applications*, to appear 1990.

[JL87] J. Jaffar, J.-L. Lassez, "Constraint Logic Programming," In *14th ACM POPL*, pp. 111-119, Munich, West Germany, 1987.

[JLM84] J. Jaffar, J.-L. Lassez, M.J. Maher, "A theory of complete logic programs with equality," In *J. of Logic Programming*, pp. 211-223, 1984.

[JN88] B. Jayaraman and Nair, A., Subset-logic Programming: Application and Implementation, In *Fifth Int'l Logic Programming Conference*, Seattle, August 1988, pp. 843-858.

[JP87] B. Jayaraman and D. A. Plaisted, "Functional Programming with Sets," In *3rd Intl. Conf. on Functional Programming and Computer Architecture*, Portland, Sept. 1987, pp. 194-210.

[JP89] B. Jayaraman and D.A. Plaisted, "Programming with Equations, Subsets, and Relations," In *North American Logic Programming Conference*, Cleveland, October 1989, pp. 1051-1068.

[JP90] B. Jayaraman and D.A. Plaisted, "Semantics of Stratified Subset-Equational Programming," submitted for journal publication, 1990.

[L87] J.W. Lloyd, "Foundations of Logic Programming," Springer-Verlag, 1987.

[LP87] G. Levi, C. Palamidessi et al, "A Complete Semantic Characterization of K-LEAF, A Logic Language with Partial Functions," In *1987 IEEE Symposium on Logic Programming*, San Francisco, pp. 318-327, 1987.

[NM88] G. Nadathur and D. Miller, "An Overview of λProlog," In *5th Int'l Logic Programming Conference*, Seattle, August 1988.

[N88] Nair, A., Compilation of Subset-logic Programs, M.S. Thesis, University of N. Carolina at Chapel Hill, December 1988.

[O85] M.J. O'Donnell, "Equational Logic as a Programming Language," MIT Press, 1985.

[O88] R.A. O'Keefe, "Practical Prolog for Real Programmers," Tutorial Notes from *5th Intl. Logic Programming Conference*, Seattle, August 1988.

[T85] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Conf. on Functional Prog. Langs. and Comp. Arch.*, Nancy, France, Sep. 1985, pp. 1-16.

[W83] D. H. D. Warren, "An Abstract Instruction Set for Prolog", Tech. Note 309, SRI International, 1983, 28 pages.