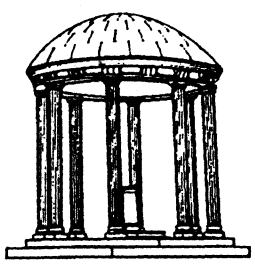
Equational Programming: A Unifying Approach to Logic and Functional Programming

Technical Report 85-030

1985

Bharat Jayaraman



The University of North Carolina at Chapel Hill
Department of Computer Science
New West Hall 035 A
Chapel Hill, N.C. 27514

Equational Programming: A Unifying Approach to Logic and Functional Programming

Bharat Jayaraman
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract

Functional and logic languages have many similarities, but there are significant differences between them that the integration of functional and logic languages is a challenging problem. The approach presented in this paper is called equational programming. We show that equations can be used to define many features of functional languages, such as abbreviations, patterns, set abstraction and infinite objects, as well as those of logic languages, such as the ability to invert functions, unify terms, and compute with logical variables and partially-defined values. A language called **EqL** is described that embodies this equational approach. The formal semantics of equations is given in terms of the *complete set of solutions* and the operational semantics is given in terms of two sets of reduction rules: $\rightarrow$-reductions and $\rightsquigarrow$-reductions. We refer to the latter form of reduction as *object refinement*. Equations are solved by gradually refining the values bound to the variables of the equation. This approach is amenable to parallel execution and also offers advantages in comparison to related techniques such as narrowing.

I. Introduction

Logic and functional languages have emerged as two promising disciplines in programming languages because of their high-level declarative specifications, elegant mathematical properties, and potential for highly parallel execution. Despite these similarities, each has strengths not readily possessed by the other. For example, functional languages support infinite data structures and higher-order functions, whereas logic languages support partially-defined values and nondeterministic specifications. Thus, the integration or "unification" of functional and logic programming has become a problem of considerable interest recently.

Most of the existing approaches to the integration of functional and logic languages may be considered to fall into two classes:

1. Languages that extend a logic language with functional capabilities. Within this class, one may further distinguish two subclasses:

   • Languages that are based on Horn clauses, such as Funlog [SY84], Eqlog [GM84], Qlog [K82], and the languages proposed by Tamaki [T84] and Barbuti et al [BBLM84].

   • Languages that are based on full first-order logic, such as Tablog [MMW84] and the language described by Hansson, et al [HHT82].

2. Languages that extend a functional language with logic capabilities. In comparison with languages of the preceding class, these languages are generally smaller, but less expressive. One may again distinguish two classes here:

   • Languages like LOGLISP [RS82], HOPE with absolute set abstraction [D83], and Scheme/L [SOS85] that support both deterministic as well as nondeterministic programs.

   • Languages like FPL [BDL82], FGL+LV [L85], Qute [SS83], and HASL [H84] that support only deterministic programs.

This paper proposes a novel approach to the problem of unifying functional and logic languages. The difference between our proposed approach and those listed above is that we use *equations* to attain the expressiveness of languages in the first class and the simplicity of languages in the second. We consider equations because they can be used to define, in a uniform way, common subexpressions, patterns, and infinite data structures, found in functional languages such as FGL, HOPE, ML, and KRC [KJLR80, BMS80, M84, T81], as well as to invert functions, unify terms and compute with partially-defined values as in

logic languages like Prolog [WPP77]. In fact, it is possible to define any Horn clause using equations. In addition, it is possible to directly express negation using equations. It is worth noting that unification in logic languages is really a special form of equation solution. Variables in such equations are *logical variables*, which obtain their values as a result of solving equations. We refer to the programming paradigm arising from programming with such equations as *equational programming*.

This paper also describes a novel execution strategy called *object refinement*. It is well-known that the evaluator for a non-strict functional language can be based on *outermost reduction* [KLP79, HO82]. However, the inclusion of logical variables and partially-defined values requires a more general reduction strategy. Two kinds of reduction rules for each primitive function are therefore proposed: $\rightarrow$-reduction is the usual reduction rule found in non-strict functional languages, and is performed whenever argument values are sufficiently defined for ordinary simplification. On the other hand, a $\rightsquigarrow$-reduction is performed when a $\rightarrow$-reduction cannot be applied, and refines argument values sufficiently to enable a $\rightarrow$-reduction. This latter process is termed *object refinement*, and can be seen as a generalization of ordinary reduction for partially-defined values, and also of conventional unification for expressions that are not restricted to pure terms. An equation is thus solved by progressively reducing its two expressions and refining objects bound to its logical variables.

Object refinement gives many opportunities for parallel execution and also has advantages over related techniques like *narrowing* [H80]. Object refinement is more efficient than narrowing because (a) reductions are performed only at the outermost level of a term, (b) information is associated with variables and there is no need to perform substitutions on terms, and (c) only the primitive functions perform this refinement. Also, object refinement can produce negative bindings for variables, whereas narrowing cannot.

The rest of this paper divides into the following sections: section II introduces the language EqL and presents several examples to show the versatility of equations for functional and logic programming; section III presents the formal semantics of the language in terms of the complete set of solutions; section IV presents the reduction rules of the evaluator and sketches the solution of equations; section V presents comparisons with related work; and section VI presents conclusions and directions for further work.

II. Equational programming

II.1. Introducing EqL

We are developing a language called **EqL** which supports equational programming. In this language we employ a functional rather than a relational or clausal *syntax*, primarily because the components of equations are expressions which are composed of function applications. Some potential advantages of the functional notation are that many problems for which logic languages are used are more clearly formulated using functions than relations. Furthermore, functions possess *directional* information, not present in relations, which allow them to be more executed more efficiently [R84]. Since equations allow one to define patterns, we restrict our operation definitions to have simple variables as formal parameters. Also, since equations may have a set of solutions we provide a set notation similar to KRC; however, equations are used to define the "generators" and "filters". Our language is essentially pure, lazy [HM76] first-order LISP augmented with set expressions and equations.

The data values in this paper are limited primarily to a set of binary trees T which are defined using two constructors *leaf* and *cons* as follows:

1. $[\,] \in T$.
2. *leaf(a)* $\in$ T, where $a \in$ A, a finite set of atoms.
3. $x, y \in T \Rightarrow$ *cons(x,y)* $\in$ T.

As in LISP, lists are a special form of trees and are written using the [...] notation. The element [ ] stands for the empty tree and also the empty list. We introduce the *leaf* constructor for technical reasons (see section on operational semantics). However, in the examples that follow, we will write a list of two atoms 1 and 2 as [1 2] instead of [leaf(1) leaf(2)].

An EqL program consists of a set of operation definitions, each of which has the following form

$$\langle opname \rangle (\langle formals \rangle) \Leftarrow \langle expression \rangle,$$

where $\langle expression \rangle$ is any syntactically well-formed composition of functions, which includes the following three forms:

1. conditional-expression: **if** $\langle expression \rangle$ **then** $\langle expression \rangle$ **else** $\langle expression \rangle$.
2. set-expression: $\{\langle expression \rangle \mid \langle equations \rangle\}$.

3. where-expression: (⟨*expression*⟩ **where** ⟨*equations*⟩).

An equation has the form:

$$\langle expression \rangle = \langle expression \rangle.$$

In this paper, a set of equations ⟨*equations*⟩ forms a conjunctive set; that is, they are simultaneously true or false. Although disjunctive sets are useful and can be supported in EqL, we do not consider them in this paper. Parentheses and braces both serve to delimit the scope of logical variables appearing in equations. (There are no global variables in this language.) We briefly explain these two constructs:

- For a where-expression (⟨*expression*⟩ **where** ⟨*equations*⟩), if expressions in the equations are restricted to *terms* composed of *cons*, *leaf*, atoms and variables, then the set of equations has a unique solution (if it exists), which is its most general unifier. If the expressions are not just *terms* then any one of the set of possible solutions (if one exists) is chosen nondeterministically. In either case the result of the where-expression is the value of ⟨*expression*⟩ obtained after substituting for the variables in the equations according to the (chosen) solution. (If the equations have no solution *failure* is signalled, which in a sequential implementation could cause backtracking.)
- The result of a set-expression {⟨*expression*⟩ | ⟨*equations*⟩} is a set of values, where each value is produced by evaluating ⟨*expression*⟩ using one of the solutions of the set of equations. This set is represented as a list; the empty list [ ] is returned if it is detected that there is no solution. No order is defined among the different solutions, and because of outermost reduction this list is produced incrementally, as needed.

II.2. Examples

We now present several examples to show the capabilities of these constructs for functional and logic programming.

Example: Patterns

$$\begin{aligned} member(x,l) \Leftarrow\ & \textbf{if } null(l) \textbf{ then } false \textbf{ else} \\ & (\textbf{if } eq(x,h) \textbf{ then } true \textbf{ else } member(x,t) \\ & \textbf{where } cons(h,t) = l) \end{aligned}$$

This program shows one of the simplest uses of an equation, namely, to define a pattern which decomposes a list into its head and tail. The above definition is similar to the LISP definition of *member*, except for its use of the pattern.

Example: Inverting functions

$union(x, y) \Leftarrow$ **if** $null(x)$ **then** y **else**
(if $member(h, y)$ **then** $union(t, y)$ **else** $cons(h, union(t, y))$
where $cons(h, t) = x)$

$\{z \mid union([1\ 2], z) = [1\ 2\ 3]\}$

The above program defines the union of two sets x and y represented as lists, and obtains the list [[1 3] [2 3] [3]] as the value of z. This is a simple example of a logic program, illustrating how functions may be inverted.

Example: Infinite set of solutions

$append(x, y) \Leftarrow$ **if** $null(x)$ **then** y **else** $cons(car(x), append(cdr(x), y))$

$\{l \mid append(l, [1\ 2]) = append([1\ 2], l)\}$

The above definition is the customary LISP definition of *append*, using explicit selectors *car* and *cdr*. The set of solutions for the logical variable l is the infinite list [*nil* [1 2] [1 2 1 2] ...].

Example: Generators

$$cprod(s_1, s_2) \Leftarrow \{cons(x, y) \mid member(x, s_1) = true,\ member(y, s_2) = true\}$$

The above function *cprod* defines the cartesian product of two sets s_1 and s_2 represented as lists. This illustrates the use of equations as "generators" to generate elements of a set, as in KRC [T81].

Example: Set Difference

$$sdiff(s_1, s_2) \Leftarrow \{x \mid member(x, s_1) = true,\ member(x, s_2) = false\}$$

sdiff([1 2 3 6], [2 3 4 5])

The above program computes the set difference $s_1 - s_2$ of two sets s_1 and s_1 represented as lists. The above example illustrates the "generate and test" paradigm, and also shows how *negation* may be stated directly. The list [1 6] is computed at the top-level.

Example: Recursive generators

$perms(l) \Leftarrow$ **if** $null(l)$ **then** [[ ]] **else**

$$\{cons(a,p) \mid member(a,l) = true$$
$$member(p, perms(sdiff(l,[a]))) = true\}$$

The above example defines the set of permutations of a list using a "recursive generator", and is again inspired by the KRC formulation of this problem [T82]. In this example we assume that *member* is defined using the LISP function *equal* rather than *eq*.

Example: Generate and test

$$primes() \Leftarrow \{p \mid member\ (p, numsfrom(2)) = true$$
$$divisors(p, nums(2, p-1)) = [\,]\}$$
$$numsfrom(n) \Leftarrow cons(n, numsfrom(n+1))$$
$$nums(l,h) \Leftarrow \textbf{if } l \geq h \textbf{ then } [\,] \textbf{ else } cons(n, nums(l+1,h))$$
$$divisors(p,l) \Leftarrow \{i \mid member(i,l) = true$$
$$mod(p,i) = 0\}$$

The function *primes* above illustrates the "generate and test" paradigm for defining the infinite set of primes. Note that the set of primes are not necessarily produced in ascending order, and because of outermost evalution the function *divisors* need produce only one divisor in order to show inequality with [ ].

Example: Fibonacci Sequence

$$addstr(s_1, s_2) \Leftarrow (cons(h_1 + h_2, addstr(t_1, t_2))$$
$$\textbf{where } cons(h_1, t_1) = s_1$$
$$cons(h_2, t_2) = s_2)$$
$$(fib \textbf{ where } fib = cons(1, cons(1, addstr(fib, cdr(fib)))))$$

The above program expresses a solution to a problem that is easily expressed in functional languages but not logic languages. The logical variable *fib* in the above equation denotes an infinite list in which each successive element is generated by adding its preceding two elements of the list. Such cyclic definitions of variables are common to many functional languages such as KRC [T81] and FGL [KJLR80], but are disallowed in most logic languages. (Colmerauer, however, shows how certain restricted forms of infinite trees called rational trees may be defined in Prolog without the occur-check [C82].)

Example: Sieve of Eratosthenes

$$sieve(l,p) \Leftarrow (\textbf{ if } divides(h,p) \textbf{ then } sieve(t,p) \textbf{ else } cons(h, sieve(t,p))$$

$$\textbf{where } cons(h,t) = l)$$

$$divides(n,p) \Leftarrow (\textbf{ if } h * h \geq n \textbf{ then } false \textbf{ else}$$
$$\textbf{if } iszero(mod(n,h)) \textbf{ then } true \textbf{ else } divides(n,t)$$
$$\textbf{where } cons(h,t) = p)$$

$$(p \textbf{ where } p = cons(2, sieve(numsfrom(3), p)))$$

The above program is a more efficient method of generating the primes, and is based on the Sieve of Eratosthenes. The last line in the above definition is similar to that in the Fibonacci example; that is, the sequence of primes produced is "fed back" to *sieve* in order to produce subsequent primes. The function *numsfrom* is the same as that in the preceding example.

Example: Difference Lists

$$dconc(d_1, d_2) \Leftarrow (cons(x,z) \textbf{ where } cons(x,y) = d_1$$
$$cons(y,z) = d_2)$$

$$(dconc(d_3, d_4) \textbf{ where } d_3 = cons(cons(1,\ cons(2,\ t)),\ t)$$
$$d_4 = cons(cons(3,\ cons(4,\ u)),\ u))$$

The above program expresses a solution to a problem that is easily expressed in logic languages but not functional languages. Here, simultaneous equations are used to effectively concatenate two lists, represented as *difference lists* [CG77], in constant time. The logical variable y in the first two equations effectively refines the object denoted by t, which is initially unbound, to *cons(cons(3, cons(4, u)), u)*. This refinement is done in constant time, independent of the size of the object denoted by x.

Example: Horn clauses

A simple example is given to illustrate the mechanical conversion of any Horn logic program to an EqL program. The following program is written using DEC-10 Prolog syntax [CM81]:

apd([ ], *X*, *X*).

apd(cons(H,T), Y, cons(H,Z)) :- *apd(T, Y, Z).*

?- *apd(X, Y,* [1,2,3,4]).

The converted EqL program would be

$$apd(A,B,C) \Leftarrow (A = [\,]$$

$$B = X$$
$$C = X)$$

$$apd(A,B,C) \Leftarrow (apd(T,Y,Z) \text{ where } A = cons(H,T)$$
$$B = Y$$
$$C = cons(H,Z))$$

$$\{cons(X,Y) \mid apd(X,Y,[1\ 2\ 3\ 4]) = true\}$$

The basic idea is to treat each predicate as a boolean-valued function. In general, a predicate p defined by k Horn-clauses would be translated into k EqL definitions with identical left-hand sides indicating that a nondeterministic choice is involved. Two other points to be noted here are: (1) The expression $(\langle equations \rangle)$ returns a boolean value indicating whether the set of equations has a solution or not. (2) The set of equations in each definition serve to unify the goal terms with those of the clause head.

III. Formal semantics

We now try to make precise the meaning of an EqL program. We shall restrict our attention in this paper to the formal semantics of a set of equations. Basically, an equation $[\![u = v]\!]$ is true if there a substitution ρ such that the *value* denoted by $([\![u]\!]\rho)$ is identical to that for $([\![v]\!]\rho)$. The semantics of an equation is then defined in terms of the *complete set of solutions*, and is similar to the complete set of unifiers relative to an equational theory [H80]. Our formalization is essentially a model-theoretic semantics [VK76, GM84]. We start by defining the domain G (for *ground* values) below:

1. $\perp$, [ ], $leaf(a) \in$ G, where $a \in$ A, the set of atoms.

2. $x, y \in$ G $\Rightarrow cons(x,y) \in$ G.

The partial ordering $\sqsubseteq$ on G is defined as follows:

1. $(\forall x)\ \perp \sqsubseteq x$.

2. $x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2 \Rightarrow cons(x_1,y_1) \sqsubseteq cons(x_2,y_2)$.

Furthermore, the limit of every infinite chain $d_1 \sqsubseteq d_2 \ldots$ is also in the domain. We use $\perp$ to denote "nontermination".

Let $\mathcal{E}$ be the following set of equations:

$$[\![u_1 = v_1 \ \ldots \ u_n = v_n]\!].$$

We first define Γ, the *set of ground solutions* of $\mathcal{E}$ with respect to a program P:

$$\Gamma = \{\rho \mid P \models \mathcal{E}\ \rho\},$$

where the logical consequence "$\models$" is defined by

$$\mathcal{P} \models \mathcal{E}\ \rho \Longleftrightarrow \mu_{\mathcal{P}}([\![u_1]\!]\rho) \equiv \mu_{\mathcal{P}}([\![v_1]\!]\rho)\ \wedge \ldots \wedge\ \mu_{\mathcal{P}}([\![u_n]\!]\rho) \equiv \mu_{\mathcal{P}}([\![v_n]\!]\rho).$$

The function $\rho \in \mathit{Subst}$, where $\mathit{Subst} = [\mathrm{Var} \mapsto \mathrm{G}]$, is the substitution function for variables in the equations. $([\![u]\!]\ \rho)$ is a textual-substitution operation, and yields the expression resulting from substituting for the variables in u according to ρ. The function $\equiv$ tests whether two ground values in G are identical.

The function $\mu_{\mathcal{P}} : \mathit{Exp} \mapsto \mathrm{G}$ gives the meaning for the set of all ground expressions Exp with respect to a program $\mathcal{P}$†. It assumes that each primitive function has a predefined meaning and each function symbol gets its denotation from the set of continuous functions $[\mathrm{G} \mapsto \mathrm{G}]$. $\mu_{\mathcal{P}}$ essentially defines the meaning of expressions. We omit its details, but instead present below the semantics of where-expressions and set-expressions as two illustrative cases:

Assuming Γ is a nonempty set of ground solutions to a set of equations $\mathcal{E}$, the semantics of a where-expression is given by

$$\mu_{\mathcal{P}}([\![(\mathit{exp}\ \textbf{where}\ \mathcal{E})]\!]) = \mu_{\mathcal{P}}([\![\mathit{exp}]\!]\ \rho) \text{ for some } \rho \in \Gamma.$$

If Γ is the empty set then the semantics of the where-expression is $\top$, which denotes "failure".

The semantics of a set-expression is given by

$$\mu_{\mathcal{P}}([\![\{\mathit{exp} \mid \mathcal{E}\}]\!]) = \{\mu_{\mathcal{P}}([\![\mathit{exp}]\!]\ \rho) \mid \rho \in \Gamma\}.$$

We illustrate the semantics of an equation by an example. Consider

$$[\![\mathit{append}(x, [1\ 2]) = \mathit{append}([1\ 2], x)]\!].$$

Suppose the substitution function $\rho = \{x \leftarrow [\ ]\}$. Then, the result of applying the textual substitution operations to each expression of the equation is:

$$[\![\mathit{append}(x, [1\ 2])]\!]\rho = [\![\mathit{append}([\ ], [1\ 2])]\!]$$
$$[\![\mathit{append}([1\ 2], x)]\!]\rho = [\![\mathit{append}([1\ 2], [\ ])]\!].$$

Next, the meaning of each of the resulting expression is given by $\mu_{\mathcal{P}}$ below, where we assume $\mathcal{P}$ is the program containing the definition of *append*:

$$\mu_{\mathcal{P}}([\![\mathit{append}([1\ 2], [\ ])]\!]) = [1\ 2]$$
$$\mu_{\mathcal{P}}([\![\mathit{append}([\ ], [1\ 2])]\!]) = [1\ 2].$$

---

† In general, permitting nondeterministic definitions as in the last example of section 2 requires that $\mu_{\mathcal{P}}$ map expressions to a set of values rather than a single value.

Since the resulting values are identical, the substitution $\rho = \{x \leftarrow [\,]\}$ is a solution to the equation.

The *set of ground solutions* Γ does not fully capture the meaning of equations because logic programs, such as difference lists, require solutions in terms of *partially-defined values* (or *nonground values*). Although a partially-defined value can be modelled by an equivalence class of ground values, it is convenient to introduce them explicitly in the formal semantics since this would facilitate showing the correctness of the operational semantics. For this reason we define the *complete set of solutions*, Σ, as follows:

$$(\forall \rho \in \Gamma)(\exists \sigma \in \Sigma)\ \sigma \leq \rho \quad \wedge \quad (\forall \sigma \in \Sigma)(\exists \rho \in \Gamma)\ \sigma \leq \rho$$

where $\sigma_1 \leq \sigma_2 \iff (\forall x)\sigma_1(x) \sqsubseteq \sigma_2(x)$, and the substitution functions σ_1 and σ_2 are from the set $[Var \mapsto \mathrm{P}]$, where P is the domain of partially-defined values (defined in the next section) and $\sqsubseteq$ is the partial ordering on P.

For example, consider the two equations defining the concatenation of two difference list $cons(cons(1, cons(2, t)), t)$ and $cons(cons(3, u), u)$:

$$[\![cons(x, y) = cons(cons(1, cons(2, t)), t)$$
$$cons(y, z) = cons(cons(3, u), u)]\!].$$

Let $\rho = \{x \leftarrow cons(cons(1, cons(2, cons(3, u))), u), y \leftarrow cons(3, u), z \leftarrow u\}$.

Although there are infinitely many ground solutions—one for each ground value assigned to the variable u in ρ—the complete set of solutions has a single element, namely, ρ.

It should be noted that in general there is no decision procedure for an arbitrary set of equations because of the existence of unsolvable problems such as Hilbert's Tenth problem. That is, it is not possible to construct an algorithm that will always terminate reporting either that it has found all complete solutions or that there is no solution. Therefore, our goal in defining an operational semantics in the next section is to devise a method that will compute a complete set of solutions *assuming termination.*

IV. Operational Semantics

IV.1. Reduction Rules

Before presenting the reduction rules, we first define the domain of partially-defined values P as follows:

1. $\bot, \phi_t \in \mathrm{P}$.

2. $leaf(x) \in \mathrm{P}$, where $x \in \mathrm{A} \cup \{\phi_a\}$.

3. $x, y \in \mathrm{P} \Rightarrow cons(x, y) \in \mathrm{P}$.

The semantic elements ϕ_a and ϕ_t require some motivation. The subscripts a and t stand for "atom" and "tree" respectively. Informally, these denote the partially-defined value "don't care" for atoms and trees. (A similar element ϕ_b exists for the set of booleans B.) An unbound variable is said to have a value ϕ_a, ϕ_b or ϕ_t. It should be noted that these values play a different role from $\perp$ which denotes "nontermination".

The partial ordering on this domain is given below:

1. $\perp \sqsubseteq \phi_a, \perp \sqsubseteq \phi_b, \perp \sqsubseteq \phi_t$.

2. $(\forall x \in \mathrm{A})\ \phi_a \sqsubseteq x, \quad (\forall x \in \mathrm{B})\ \phi_b \sqsubseteq x, \quad (\forall x \in \mathrm{P})\ \phi_t \sqsubseteq x$.

3. $x_1 \sqsubseteq x_2 \ \wedge\ y_1 \sqsubseteq y_2 \Rightarrow cons(x_1, y_1) \sqsubseteq cons(x_2, y_2)$.

The partial ordering defined above determines the allowable *object refinements.* A variable v bound to an object whose value is p may be refined only to a value q such that $p \sqsubseteq q$. Note that we assume *object uniqueness* as in [L85]: the object bound to the variable v remains the same; it is the value of the object that is refined from p to q. For example, in the expression **if** *null(x)* **then** ... **else** ..., assuming x were initially unbound, its value would be refined to [ ] in the **then** part and either *leaf(a)* or $cons(x_1, x_2)$ in the else part, where $a \leftarrow \phi_a$, $x_1 \leftarrow \phi_t$ and $x_2 \leftarrow \phi_t$. The possibility of more than one refinement corresponds to nondeterministic choice in logic languages.

The operational semantics of the language is based on *outermost reduction.* Since partially-defined values are permitted, two kinds of reduction rules for each primitive function are needed: $\rightarrow$-reduction is the usual reduction rule found in non-strict functional languages, and is performed whenever argument expressions reduce to values that are sufficiently defined for ordinary simplification. A $\rightsquigarrow$-reduction is performed when a $\rightarrow$-reduction cannot be performed, and corresponds to the case when an argument expression reduces to an unbound variable. A $\rightsquigarrow$-reduction refines the value of the object bound to the variable sufficiently to enable a $\rightarrow$-reduction. Thus, all expressions are reduced by outermost $\rightarrow$-reductions and $\rightsquigarrow$-reductions to either an atom, or a boolean, or an unbound variable, or an expression of the form $cons(e_1, e_2)$, where e_1 and e_2 are expressions.

Figure 1 shows the $\rightarrow$-reduction rules for a few primitive functions. The rules *cdr* can be defined similar to that for *car*, and no rules are shown for *cons* because it does not evaluate its arguments. All arguments are shown in their maximally reduced forms.

Figure 1. Sample $\rightarrow$-Reduction Rules

$car(\bot) \rightarrow \bot$	$atom(\bot) \rightarrow \bot$
$car(leaf(e_1)) \rightarrow \top$	$atom(leaf(e_1)) \rightarrow true$
$car(cons(e_1, e_2)) \rightarrow e_1$	$atom(cons(e_1, e_2)) \rightarrow false$
$car(x) \rightarrow \top, x \notin \mathrm{P}$	$atom(x) \rightarrow \top, x \notin \mathrm{P}$
$null(\bot) \rightarrow \bot$	**if** $\bot$ **then** e_1 **else** $e_2 \rightarrow \bot$
$null([\,]) \rightarrow true$	**if** *true* **then** e_1 **else** $e_2 \rightarrow e_1$
$null(leaf(e_1)) \rightarrow false,$	**if** *false* **then** e_1 **else** $e_2 \rightarrow e_2$
$null(cons(e_1, e_2)) \rightarrow false$	**if** x **then** e_1 **else** $e_2 \rightarrow \top$, if $x \notin \mathrm{B} \cup \{\phi_b, \bot\}$
$null(x) \rightarrow \top$, if $x \notin \mathrm{P}$	

Note: In the above rules, e_i is any expression and $\top$ denotes failure.

Figure 2 shows the $\rightsquigarrow$-reduction rules for the same primitives. Again, arguments are assumed to have been maximally reduced. It may be noted that for these primitives the $\rightarrow$-reduction and $\rightsquigarrow$-reduction rules together cover all cases, i.e., all values in domain P.

Figure 2. Sample $\rightsquigarrow$-Reduction Rules

$$\text{if } x = \phi_t, \quad atom(x) \rightsquigarrow \begin{cases} true, & \text{with } \{x \leftarrow leaf(x_1), x_1 \leftarrow \phi_a\}, \text{ or} \\ false, & \text{with } \{x \leftarrow cons(x_1, x_2),\ x_1 \leftarrow \phi_t,\ x_2 \leftarrow \phi_t\} \end{cases}$$

$$\text{if } x = \phi_t, \quad car(x) \rightsquigarrow x_1, \text{ with } \{x \leftarrow cons(x_1, x_2),\ x_1 \leftarrow \phi_t,\ x_2 \leftarrow \phi_t\}$$

$$\text{if } x = \phi_t, \quad null(x) \rightsquigarrow \begin{cases} true, & \text{with } \{x \leftarrow [\,]\}, \text{ or} \\ false, & \text{with } \{x \leftarrow leaf(x_1), x_1 \leftarrow \phi_a\}, \text{ or} \\ false, & \text{with } \{x \leftarrow cons(x_1, x_2),\ x_1 \leftarrow \phi_t,\ x_2 \leftarrow \phi_t\} \end{cases}$$

$$\text{if } x = \phi_b, \quad \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \rightsquigarrow \begin{cases} e_1, & \text{with } \{x \leftarrow true\}, \text{ or} \\ e_2, & \text{with } \{x \leftarrow false\} \end{cases}$$

Note: In the above rules, e_i is any expression, and x_1 and x_2 are new variables that do not occur elsewhere in the program.

The $\rightsquigarrow$-reduction rules are determined from the partial ordering on the domain and from the $\rightarrow$-reduction rules. In general, an object may be refined to any of the *next more defined* values in the domain. The one exception is that refinements involving $\bot$ are not allowed. For example, the rules in Figure 2 that refine an unbound variable to a *cons* expression bind the arguments of *cons* to ϕ_t rather than to $\bot$. This is referred to as *object refinement* and is shown in Figure 2 by $\{variable \leftarrow value\}$.

IV.2. Solution of Equations

We informally present here the solution of equations by an example. Our example shows the solution to a single equation, but since more equations are added as the solution proceeds, the discussion is applicable to a set of equations as well. For simplicity, we omit the *leaf* constructor for the atoms 1 and 2 in this example.

Consider the equation

$$append([1\ 2], x) = append(x, [1\ 2]). \tag{0}$$

Recall that it has an infinite number of solutions: [ ], [1 2], [1 2 1 2], etc. The above equation is just an abbreviation for

$$append(cons(1, cons(2, [\ ])), x) = append(x, cons(1, cons(2, [\ ]))). \tag{1}$$

The expression on the left of equation (1) $\rightarrow$-reduces to

$$cons(car(l_1), append(cdr(l_1), x))$$

where $l_1 \leftarrow cons(1, cons(2, [\ ]))$.

The expression on the right has three possible $\rightsquigarrow$-reductions, corresponding to the three possible refinements for $null(x)$ where $x \leftarrow \phi_t$:

Refinement	Reduced Expression
1. $x \leftarrow [\ ]$	$cons(1, cons(2, [\ ]))$
2. $x \leftarrow leaf(x_1), x_1 \leftarrow \phi_a$	$cons(car(x), append(cdr(x), l_2))$ where $l_2 = cons(1, cons(2, [\ ]))$
3. $x \leftarrow cons(x_1, x_2)$ $x_1 \leftarrow \phi_t$ $x_2 \leftarrow \phi_t$	$cons(car(x), append(cdr(x), l_2))$ where $l_2 = cons(1, cons(2, [\ ]))$

All three refinements would be considered in an actual implementation—the first in fact leads to an overall solution—but we present here the most interesting case, which is the equation resulting from refinement 3:

$$cons(car(l_1), append(cdr(l_1), x)) = cons(car(x), append(cdr(x), l_2)) \tag{2}$$

where $l_1, l_2 \leftarrow cons(1, cons(2, [\ ]))$, $x \leftarrow cons(x_1, x_2)$, $x_1 \leftarrow \phi_t$, and $x_2 \leftarrow \phi_t$.

This leads us to our **first equation-solution rule**: *Whenever an equation is of the form* $cons(e_1, e_2) = cons(e_3, e_4)$, *two new equations are generated:* $e_1 = e_3$, *and* $e_2 = e_4$. (A

similar rule involving the constructor *leaf* may be defined.) The two new equations for our example are

$$car(l_1) = car(x) \tag{3}$$

$$append(cdr(l_1), x) = append(cdr(x), l_2). \tag{4}$$

Equation (3) further reduces to $1 = x_1$, at which time x_1 gets bound to 1. This reflects a **second equation-solution rule**: *Whenever an equation is of the form $v = e$ or $e = v$, where v is an unbound variable and e is either an atom or boolean or [ ] or $leaf(e_1)$ or $cons(e_1, e_2)$, v is bound to e.*

Equation (4) is solved similar to equation (1). The next solution obtained is

$$\{ x \leftarrow cons(x_1, x_2), \quad x_1 \leftarrow 1, \quad x_2 \leftarrow cons(x_3, x_4), \quad x_3 \leftarrow 2, \quad x_4 \leftarrow [\,] \}$$

The other solutions are obtained if x_4 were refined to $cons(x_5, x_6)$ and the resulting equations solved.

There is a **third equation-solution rule** which is applicable when both expressions reduce to unbound variables: *Whenever an equation is of the form $v_1 = v_2$ and both v_1 and v_2 are unbound variables, both v_1 and v_1 are bound to the same object, whose value is ϕ_t.*

An interesting property of the equation-solution rules is that they effectively perform unification when the expressions are restricted to just *terms*. Another is that rule 1 gives opportunities for parallel execution. However, access to common variables across equations that are solved in parallel must be synchronized so as not to lose any refinements. We refer to the parallelism arising from recursively decomposing an equation into sub-equations as *cons-parallelism*, which is a special form of *and-parallelism*. The different refinements of a variable can be examined in parallel and gives rise to *or-parallelism*.

V. Related Work

The term *equational programming* was first introduced by Hoffman and O'Donnell [HO82, O85], who used it to refer to a style of function definitions by equations and a simple semantics based on the logical consequences of equality. Unlike the logical variables of EqL equations, the variables here are universally quantified and the equations here are similar to abstract data type specifications. Restrictions on the left-hand sides of equations are placed in order to insure determinacy. The term equational programming has been recently used by Dershowitz and Plaisted to refer to a style of programming with conditional rewrite rules [DP85] which provides the capability of first-order functional and

logic programming in a uniform and elegant way. Conditional expressions and equations in EqL give all the expressive power of conditional rewrite rules, and some additional capabilities: because variables in EqL equations may be defined cyclically, certain efficient definitions of infinite data structures are possible, as illustrated in the Fibonacci example.

Object refinement combines the advantages of several recently proposed approaches for executing logic languages. We summarize below the important similarities and differences: Our reduction of expressions is similar to the pattern-driven lazy reduction of Funlog [SY84] and the reduction strategy of FGL+LV [L85], except that no explicit unification is performed in our approach. It is also related to Berkling's ϵ-reduction [B85] for reducing sets of equations that are composed of pure terms. The programs considered by Berkling are essentially the same as EqL programs that are mechanically derived from Horn-clauses (see example of section II.2).

Object refinement is also related to the evaluation mechanism of Prolog with equality, proposed by Kornfeld [K83]. In this language, the programmer may specify so-called Ω-terms and equality theorems which get invoked whenever two terms don't unify syntactically. These equality theorems in effect specify explicitly how refinements are to be made. In EqL, these refinements are made automatically and are determined solely by the primitive functions.

Perhaps the most closely related approach is that of *narrowing* [H80]. It differs in two ways form narrowing: (a) no explicit unification is performed, and (b) reductions occur only at the outermost level of a term. Note that narrowing is not applicable to a language like EqL which has only simple variables as formal parameters. Also narrowing cannot handle negative information. Recently, two variations of narrowing have been proposed: conditional narrowing [DP85] and lazy narrowing [R85]. Object refinement combines the generality of conditional narrowing (being applicable to conditional expressions) with the efficiency of lazy narrowing (being based on outermost reduction). The simplication steps of conditional narrowing are analogous to our $\rightarrow$-reductions, and the narrowing steps are analogous to our $\rightsquigarrow$-reductions.

VI. Conclusions and Further Work

EqL supports functional programming more directly than logic programming because of its functional syntax. However, our examples show that many logic programming paradigms are easily stated in the language; in fact any Horn logic program can be di-

rectly converted to an EqL program. EqL operations have only simple variables as formal parameters, i.e., there are no patterns, because patterns can be realized quite easily by equations, as shown in the examples above. We provide set expressions because equations may possess a set of solutions. In this respect, our approach is similar to that of LOGLISP and HOPE with absolute set abstraction. The 'findall' predicate of Prolog [CM81] and the various forms of 'all solutions' surveyed by Naish [N85] are also related. In comparison with these and the earlier cited works, the main contribution of our language lies in demonstrating the clarity and power of *equations* for functional and logic programming.

We are at present investigating the parallel implementation of EqL. An interesting aspect here is the implementation of set-expressions. In this implementation, we represent the tree of alternatives arising from nondeterministic choices explicitly as a tree of *frames*, where each frame contains a partially-solved set of equations and corresponding variable bindings. In this approach, and-parallelism arises within a single frame whereas or-parallelism arises across different frames. Whenever a refinement is performed and there is more than one choice, a frame "splits" into several frames, one for each choice. The frames for equations that yield a solution are incorporated into the list of solutions; those that do not are deleted. Furthermore, the list of solutions is produced incrementally, as needed.

Also under development is a formal correctness proof of the operational semantics. There is a strong similarity between this proof and that of theorem 2 of Hullot [H80] for the complete set of unifiers. Essentially, the argument is as follows: If ρ is a solution to a set of equations $\mathcal{E}$ and if all sequences of $\rightarrow$-reductions emanating from $\mathcal{E}\rho$ terminate, then one can "project" each step of a sequence of $\rightsquigarrow$-reductions from E on to a corresponding step of a sequence of $\rightarrow$-reductions from $\mathcal{E}\rho$. The proof is inductive and depends on the correctness of the $\rightsquigarrow$-reductions for each of the primitive functions.

References

[A84] H. Abramson, "A Prological Definition of HASL, a Purely Functional Language with Unification Based on Conditional Binding Expressions." In *New Generation Computing* **2**, 1984, pp. 3–35.

[B85] K. Berkling, "Epsilon-Reduction: Another View of Unification," In Proc. of *IFIP TC-10 Working Conference on Fifth Generation Computer Architecture*, UMIST, Manchester, July, 1985.

[BBLM84] R. Barbuti, M. Bellia, G. Levi, and M. Martelli, "On the Integration of Logic Programming and Functional Programming." In *Internatl. Symp. Logic Programming, IEEE*, Atlantic City, 1984, pp. 160–166.

[BDL82] M. Bellia, P. Degano, and G. Levi, "The Call by Name Semantics of a Clause Language with Functions." In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 281–295.

[BMS80] R. M. Burstall, D. B. MacQueen, D. T. Sanella, "HOPE: an experimental applicative language." In *1980 ACM LISP Conference*, pp. 136–143.

[C78] K. L. Clark, "Negation as Failure." In *Logic and Data Bases*, Ed. H. Gallaire and J. Minker, Plenum Press, New York, 1978, pp. 293–322.

[C82] A. Colmerauer, "Prolog and infinite trees," In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 231-251.

[CG77] K. L. Clark and S. Gregory, "A First-order Theory of Data and Programs." In *Information Processing*, 1977, pp. 939–944.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog.* Springer-Verlag, New York, 1981.

[D83] J. Darlington, "Unifying Functional and Logic Languages." Internal Report, Imperial College, London, 1983.

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

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

[H80] J-M. Hullot, "Canonical Forms and Unification." In *Proc. 5th Workshop on Automated Deduction*, Springer Lecture Notes, 1980, pp. 318–334.

[HHT82] A. Hansson, S. Haridi, and S.-A. Tärnlund, "Properties of a Logic Programming Language." In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 267–280.

[HM76] P. Henderson and J. H. Morris, "A Lazy Evaluator." In *Third ACM POPL*, 1976, pp. 95–103.

[HO82] C. M. Hoffman and M. J. O'Donnell, "Programming with Equations." *ACM TOPLAS* **4**, No. 1 (January 1982) pp. 83–112.

[JKK83] J.-P. Jouannaud, C. Kirchner, H. Kirchner, "Incremental Construction of Unification Algorithms in Equational Theories." In *ICALP*, Barcelona, 1983, pp. 361-373.

[K82] H. J. Komorowski, "QLOG—The Programming Environment for PROLOG in LISP." In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 315–322.

[K83] W. A. Kornfeld, "Equality for Prolog." In *8th IJCAI*, Karlsruhe, West Germany, 1983, pp. 514–519.

[K84] T. Khabaza, "Negation as Failure and Parallelism." In *Internatl. Symp. Logic Programming, IEEE*, Atlantic City 1984, pp. 70–75.

[KJLR80] R. M. Keller, B. Jayaraman, G. Lindstrom, D. Rose, "FGL Programmer's Guide," AMPS Technical Memo No. 1, University of Utah, July, 1980.

[L85] G. Lindstrom, "Functional Programming and the Logical Variable." In *12th ACM POPL*, New Orleans, Jan 1985, pp. 266–280.

[MMW84] Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: The Deductive-Tableau Programming Language," In *ACM Symp. on LISP and Functional Programming*, Austin, 1984, pp. 323–330.

[N85] L. Naish, "All Solutions Predicates in Prolog," In *Symp. on Logic Programming*, Boston, 1985, pp. 73-77.

[O85] M. J. O'Donnell, "Equational logic as a programming language," M.I.T. Press, 1985.

[R65] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *JACM* **12**, pp. 23-41, 1965.

[R84] U. S. Reddy, "On the Relationship between Logic and Functional Languages," Draft of article to appear in *Functional and Logic Programming*, eds. D. DeGroot and G. Lindstrom, 1984.

[R85] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages." In *1985 Symp. on Logic Programming*, Boston, 1985, pp. 138–151.

[RS82] J. A. Robinson and E. E. Sibert, "LOGLISP: Motivation, Design, and Implementation." In *Logic Programming*, Ed. K. L. Clark and S.-A. Tärnlund, Academic Press, 1982, pp. 299–313.

[SOS85] A. Srivastava, D. Oxley, A. Srivastava, "An(other) Integration of Functional Programming." In Proc. 1985 Symposium on Logic Programming, Boston, 1985, pp. 254-260.

[SS83] M. Sato and T. Sakurai, "Qute: A Prolog/Lisp Type Language for Logic Programming." In *8th IJCAI*, Karlsruhe, West Germany, 1983, pp. 507–513.

[SY84] P. A. Subrahmanyam and J-H. You, "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logical Programming." In *Internatl. Symp. Logic Programming, IEEE*, Atlantic City, 1984, pp. 144–153.

[T81] D. A. Turner, "The semantic elegance of applicative languages," In *ACM Symp. on Func. Prog. and Comp. Arch.*, New Hampshire, October, 1981, pp. 85-92.

[T84] H. Tamaki, "Semantics of a Logic Programming Language with a Reducibility Predicate." In *Internatl. Symp. Logic Prog., IEEE*, Atlantic City, 1984, pp. 259–264.

[VK76] M. H. van Emden and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language." *J. ACM* **23**, No. 4 (1976) pp. 733–743.

[WPP77] D. H. D. Warren, F. Pereira, and L. M. Pereira, "Prolog: the Language and Its Implementation Compared with LISP." *SIGPLAN Notices* **12**, No. 8 (1977) pp. 109–115.