Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

LOGIC PROGRAMMING FOR NUMERICAL COMPUTATION

Xinming Lin
Walter J. Karplus

June 1989
CSD-890042

Logic Programming for Numerical Computation

Xinming Lin and Walter J. Karplus
Computer Science Department
University of California, Los Angeles

Abstract

There have been many exciting advances in the field of logic programming. However, it has caught far less attention than it should in the community of scientific computing. One of the reasons is that presently available logic programming languages are not designed with numerical computation in mind. This paper addresses the need to specify numerical computation in a high-level declarative language which is natural to the user's thinking. A logic programming language is proposed, which allows many common tasks in numerical computation to be expressed naturally. The semantics of the language is discussed informally. An interpretation of the language in PROLOG is discussed. Applications in numerical computation are presented.

1 Introduction

Logic programming has gone through a lot since its first introduction in the early 70's. There have been many exciting advances in the field. However, the impression has been given that logic programming languages are solely the languages of Artificial Intelligence. Logic programming for numerical computation has not been appreciated (see [1]). The fact that logic is an abstraction of reasoning which exists in every kind of computation and different kinds of computations require different emphasis has been paid too little attention. It is the purpose of this paper to address the needs of numerical computation and to propose a solution.

Numerical computations are a wide class of computer applications that have been employed extensively in many scientific studies. Scientists and engineers have used them to do data analysis, to simulate systems and phenomena, and to make predictions. Among many different applications, certain steps are common in solving a problem [2,3]:

1. A physical phenomenon is studied.
2. A mathematical model is built to approximate the phenomenon.
3. A discrete approximation is made to simplify the model.
4. Numerical algorithms are chosen to decide a solution.
5. A program is written in a language that a computer can understand and is run to actually find the solution.

In the past, the last step has long been separated from the previous steps, because the language that a computer can understand is so much different from the languages that are employed in the early steps. A question then arises. Can we tell a computer what to do in just the same way that we convince ourselves that our methods will work. That is to say, can steps three, four, and five be combined in a high-level declarative language which is also close to the language of step two. The language proposed below is aimed toward that goal.

2 A Logic Programming Language with Functions LP($\mathcal{F}$)

2.1 The Syntax

The symbols of LP($\mathcal{F}$) are the following:

1. Real numbers `0 1.2 ...`
2. n-ary predicate variables for any natural number n `f/n g/n ...`
3. Predicate constants `=/2 +/3 -/3 true/0 false/0 ...`

4. Universal variables `X Y ...`

5. Other symbols `if , == ( ) [ | ]`

In LP($\mathcal{F}$), a n-ary function is considered as a (n+1)-ary predicate. The last argument is treated as the value of the function. Computations are represented as recursive functions that are, in turn, represented as predicates. In LP($\mathcal{F}$), predicates are first-order objects. A universal variable can take a number or a predicate as its value.

A *formula* in LP($\mathcal{F}$) is defined inductively by the following formation rules:

1. A predicate is a formula.

2. If **A** and **B** are formulas, so is **A , B**.

3. If **A** is a predicate and **B** is a formula, then **A `if` B** is a formula.

A formula is also called a program. Formulas of the form **A `if` B** are called if-formulas. As a convenience, the connective (`if`) is considered to have a higher precedence than the connective (`,`), so (`a if b , c`) is equivalent to (`(a if b) , c`). Therefore, (`a if b`) is a program, so are (`a`) and (`a if (b if c), a, b`).

The following notations are only syntactic sugar to write shorter and cleaner programs:

```
f(X) == a(X) + b(X)            ⇒
              f(X, F) if (a(X, A), b(X, B), F = A+B)
f(X) == a(X) + b(X) if c(X)    ⇒
              f(X, F) if (c(X), a(X, A), b(X, B), F = A+B)
a(X) + b(X) == f(X)            ⇒
              a(X, A) if (b(X, B), f(X, F), A+B = F),
              b(X, B) if (a(X, A), f(X, F), A+B = F)
f(d(e)) == g(a, b(c))          ⇒
              f(D, F) if (d(e, D), a(A), b(c, B), F = g(A, B))
```

2.2 Operational Semantics

Since the semantics of PROLOG has long been studied and understood (see [4]), the semantics of LP($\mathcal{F}$) can be interpreted in a PROLOG-like

language. This is not considered to be a replacement of a formal definition of the semantics but an intuitive beginning . The semantics of a formula **A**, denoted by **Eval[[A]]**, under an environment is a substitution and new environment pair, where an environment is just a series of if-formulas (refer [5,6]).

$$\textbf{Eval}[[\textbf{A}]](e_1) = (\theta, e_2).$$

The semantics of a formula is defined inductively by the following rules:

1. **Eval[[A, B]]**(e) = **Eval[[**θ_1**B]]**(e_1), where **Eval[[A]]**(e) = (θ_1, e_1).
2. **Eval[[A if B]]**(e) = (ϵ, (e,**A if B**)), where ϵ is the empty substitution.
3. **Eval[[A]]**(e) = **Eval[[(**θ**B]]**(e), where e = (..., $\dot{\textbf{A}}$ **if B**, ...) and θ is the most general unifier between **A** and $\dot{\textbf{A}}$. The first if-formula is chosen if multiple choices exist, just for the sake of simplicity at this time.
4. **Eval[[true]]**(e) = (ϵ, e), where ϵ is the empty substitution.

In LP($\mathcal{F}$), predicates are first-order objects. Therefore, (`a(x) if (x if y)`) is a meaningful formula. A predicate is defined in the scope of its appearance. In the above example, (`a`) is defined in the whole formula but (`x`) and (`y`) are only defined in the sub-formula (`x if y`). Therefore,

$$\begin{aligned}
& \textbf{Eval}[[\texttt{a(x) if (x if y)}, \texttt{a(y)}, \texttt{y}]] \\
= \ & \textbf{Eval}[[\texttt{a(y)}, \texttt{y}]](\texttt{a(x) if (x if y)}) \\
= \ & \textbf{Eval}[[\texttt{y if } \mathfrak{y}, \texttt{y}]](\texttt{a(x) if (x if y)}) \\
= \ & \textbf{Eval}[[\texttt{y}]](\texttt{a(x) if (x if y)}, \texttt{y if } \mathfrak{y}) \\
= \ & \textbf{Eval}[[\mathfrak{y}]](\texttt{a(x) if (x if y)}, \texttt{y if } \mathfrak{y}),
\end{aligned}$$

where ($\mathfrak{y}$) is a predicate different from (`y`).

2.3 Semantic Unification

The unification is borrowed and modified from the unification used in CLP($\mathcal{R}$) to take into account of the semantics of arithmetic equality [7,8]. Unification of `X+1` and `3` will succeed with `X` unified to 2. Unification of

`X+Y` and `1` will be delayed until `X` and `Y` can be uniquely determined. For example, unification of `X-Y` and `2` lately will succeed with `X` unified to `1.5` and `Y` unified to `-0.5`. In addition, a predicate variable can be unified with another predicate variable, but two predicates wouldn't be unified if the predicate names have not been unified. For example, unification of `a(x)` and `b(x)` will fail if `a` and `b` was not unified before, while unification of `a(x)` and `a(y)` will succeed with `x` unified with `y`.

Moreover, a unification will be performed automatically between a predicate to be interpreted and the history of interpretation of that predicate. For example, in the program

```
x(0, 0) if true,
x(3, 9) if true,
x(I, X) if (x(I-1, X1), x(I+1, X2), X1-2*X+X2 = 0),
x(1, A).
```

the interpretation of `x(1, A)` is

$$\begin{aligned} & \mathbf{Eval}[[\mathtt{x(1,A)}]] \\ = \ & \mathbf{Eval}[[\mathtt{x(0,X1)}, \mathtt{x(2,X2)}, \mathtt{X1} - 2 * \mathtt{A} + \mathtt{X2} = 0]] \\ = \ & \mathbf{Eval}[[\mathtt{x(1,X3)}, \mathtt{x(3,X4)}, 0 - 2 * \mathtt{A} + \mathtt{X2} = 0, \mathtt{X3} - 2 * \mathtt{X2} + \mathtt{X4} = 0]] \\ = \ & \mathbf{Eval}[[0 - 2 * \mathtt{A} + \mathtt{X2} = 0, \mathtt{A} - 2 * \mathtt{X2} + 9 = 0]] \\ = \ & \{\mathtt{A}/3, \mathtt{X2}/6\}, \end{aligned}$$

where the environment is implicitly stated. The point is that `x(1, X3)` is automatically unified with `x(1, A)` when it is interpreted and therefore `X3` is unified with `A`.

2.4 Structured Data Types

Structured data types, especially lists, have been used extensively in PROLOG as a recursion mechanism, whereas in LP($\mathcal{F}$) recursive functions have taken over the role. Recursive functions are more powerful than lists as a recursion mechanism because lists are only one-dimensional while functions can be multi-dimensional. However, sometimes it is desirable to have structured data objects, such as in the case of input and output. In LP($\mathcal{F}$), functions are untyped and can take not only numbers but also structures

as arguments and can return either as its value. A list is simply a structure that is composed of two parts. Unification on structures in LP($\mathcal{F}$) is the same as in PROLOG. The addition of two lists is like:

```
add_list([], [], []) if true,
add_list([X| A], [Y| B], [Z| C]) if (Z = X+Y, add_list(A, B, C))
```

A structure can also contain a function. In

```
a(K, A) if (x(K, X), A = [X| a(K-1)]),
```

a defines itself recursively.

3 Applications in Numerical Computation

Now, it is time to see how numerical computation can be expressed in LP($\mathcal{F}$). The first example is the LU-decomposition of a matrix.

```
lu(x, y) if (
        a(I, J, 0) == x(I, J),
        a(I, K, K) == a(I, K, K-1)/a(K, K, K-1)
                if I > K,
        a(I, J, K) == a(I, J, K-1)-a(I, K, K)*a(K, J, K-1)
                if (I > K, J > K),
        y(I, J) == a(I, J, I-1) if I =< J,
        y(I, J) == a(I, J, J) if I > J
),
```

where function x represents the input matrix and y represents the output matrix. In the above LP($\mathcal{F}$) program, the output matrix is specified as a certain relation of the input matrix. There is no destructive computations as in imperative programming languages.

The second example is to solve the initial-value problem:

$$dy/dt = -y + t + 1,\ 0 \leq t \leq 1,\ y(0) = 1.$$

An integration algorithm using a backward Euler's method is used to solve the problem. The reason a backward algorithm is used is to demonstrate the power of the automatic unification discussed in section 2.3.

```
integ_euler(y, f, H) if
        y(K) == y(K-1)+f(K)*H,
H = 0.1,
f(K) == -y(K)+ K*H+1,
y(0) == 1,
integ_euler(y, f, H).
```

The interpretation of `y(1, Y)` would be

$$\begin{aligned}
& \mathbf{Eval}[[\mathtt{y(1,Y)}]] \\
= \ & \mathbf{Eval}[[\mathtt{y(0,Y0)}, \mathtt{f(1,F)}, \mathtt{Y} = \mathtt{Y0} + \mathtt{F} * 0.1]] \\
= \ & \mathbf{Eval}[[\mathtt{f(1,F)}, \mathtt{Y} = 1 + \mathtt{F} * 0.1]] \\
= \ & \mathbf{Eval}[[\mathtt{y(1,Y1)}, \mathtt{F} = -\mathtt{Y1} + 1 * 0.1 + 1, \mathtt{Y} = 1 + \mathtt{F} * 0.1]] \\
= \ & \mathbf{Eval}[[\mathtt{F} = -\mathtt{Y} + 1 * 0.1 + 1, \mathtt{Y} = 1 + \mathtt{F} * 0.1]] \\
= \ & \{\mathtt{F}/0.0909, \mathtt{Y}/1.009\},
\end{aligned}$$

where the environment is implicitly stated.

The last example is to solve a Poisson's equation:

$$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = xe^y,\ 0 < x < 2,\ 0 < y < 1,$$

with the boundary conditions

$$\begin{aligned}
u(0,y) = 0,\ u(2,y) = 2e^y,\ 0 \le y \le 1, \\
u(x,0) = x,\ u(x,1) = ex,\ 0 \le x \le 2
\end{aligned}$$

The discretization uses the three-point finite difference method. It is known that the discretized problem is a system of linear algebraic equations, which can be solved by the semantic unification.

```
deriv_2nd(K, f, g, H) if (
        2*f(I, J)+ H*H*g(I, J) == f(I-1, J)+f(I+1, J) if K = 1,
        2*f(I, J)+ H*H*g(I, J) == f(I, J-1)+f(I, J+1) if K = 2
),
N = 20,
H = 2/N,
M = 10,
```

```
K = 1/M,
x(I) == I*H,
y(I) == I*K,
u(0, J) == 0,
u(N, J) == c(N, J),
u(I, 0) == x(I),
u(I, M) == c(I, M),
deriv_2nd(1, u, a, H),
deriv_2nd(2, u, b, K),
a(I, J) + b(I, J) == c(I, J),
c(I, J, C) if (x(I, X), y(J, Y), C = X*exp(Y)).
```

4 Implementation Issues

An interpreter for LP($\mathcal{F}$) is being written in Quintus Prolog [9]. The purpose of the interpretation is to demonstrate the consistency and feasibility of the language as well as to provide a language machine to actually run LP($\mathcal{F}$) programs. In the interpretation the efficiency is not the main concern, although it is not totally ignored.

The interpretation is basically a realization of the operational semantics discussed in section 2.2. The environment is stored in the database. An if-formula is asserted into the database whenever it is interpreted. The naming and unification of predicate variables are implemented by tables. The history of predicates that have been interpreted is stored as a binary tree for automatic unification. A list is used to store the delayed linear algebraic equations and are solved by Gaussian elimination. Non-linear algebraic equations are not considered at this time.

```
%
% eval(formula)
%
eval(A) :-
        empty_btree(Empty),
        interpret(A, Empty, L), solve(L, []).
% solve(equation_list, delayed_list)
```

```
% solves a system of linear algebraic equations
% by Gaussian elimination

%
% interpret(formula, history, delayed_equations)
%
interpret(true, _, []).
interpret((A , B), E, L) :-
        !, interpret(A, E, L1),
        interpret(B, E, L2), append(L1, L2, L).
interpret(((X == Y) if B), _, []) :-
        substitute(X, U, UL), substitute(Y, V, VL),
        connect(VL, R1), and(R1, U=V, R), create(UL, B, R).
% substitute(term, term, predicate_list)
% creates a list of predicates as the body of an if-formula.
%
interpret((A if B), _, []) :-
        assertz(lambda(A, B)).
interpret((X == Y), E, L) :-
        interpret(((X == Y) if true), E, L).
interpret((X = Y), E, L) :-
        substitute(X, U, UL), substitute(Y, V, VL),
        append(UL, VL, L0), connect(L0, R),
        interpret(R, E, L1),
        unify(U, V, L2), append(L1, L2, L).
% unify(term, term, delayed_list)
% uses the semantic unification to take into
% account the semantics of arithmetic equality.
%
interpret(A, _, []) :-
        predicate_property(A, built_in),
        call(A).
interpret(Goal, _, []) :-
        lambda(Goal, true).
interpret(Goal, E, L) :-
        old_query(Goal, E, L).
```

```
% old_query(predicate, history, delayed_list)
% checks if automatic unification with the
% history of the interpreted predicates can be
% performed.
%
interpret(Goal, E, L) :-
        functor(Goal, F, N), functor(Head, F, N),
        lambda(Head, Body),
        unify_args(1, N, Goal, Head, L1),
        add_btree(E, F, Goal, NewE),
        interpret(Body, NewE, L2), append(L1, L2, L3),
        solve(L3, L), assertz(lambda(Goal, true)).
% assert interpreted predicates to prevent redundant
% interpretations.
```

5 Summary

A high-level logic programming language is designed for the purpose of numerical computation. Common features in numerical computation are handled naturally in the language so that the user can concentrate on the high-level thinking of his problem. The language incorporates functions as a recursive mechanism which is more powerful than lists. Semantic unification allows computations to be expressed in algebraic equations. Modulation is facilitated through nested if-formulas. Computations are driven by demand and supported by unification.

Logic programming for numerical computation opens a new programming methodology in scientific computing. To the user, a high level declarative language can be used to describe his problem in a way which is close to his mathematical modeling of the physical phenomenon. The actual implementation is transparent to him. To the language implementor, a declarative language allows optimization and parallel processing to take full advantage of them, independent of the applications. Unlike imperative programming languages which introduce man-made execution and data dependencies, logic programming languages leave computations in a declarative form that awaits the language implementor to exploit as much efficiency as he can.

Acknowledgement

We want to thank Professor D. Stott Parker for his teaching the first author logic programming. We also appreciate Bob Tisdale's interest on the project and his corrective reading of the paper. The project is supported by Lawrence Livermore National Laboratory under the grant UCLLNL B056074.

References

[1] W. F. Clocksin. A technique for translating clausal specifications of numerical methods into efficient programs. *The Journal of Logic Programming*, 5:231–242, 1988.

[2] V. Vemuri and Walter J. Karplus. *Digital Computer Treatment of Partial Differential Equations.* Prentice-Hall, 1981.

[3] Roger W. Hockney and James W. Eastwood. *Computer Simulation Using Particles.* McGraw-Hill International Book Company, 1981.

[4] Krzysztof R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery*, 29(3):841–862, July 1982.

[5] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[6] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International (UK) Ltd, 1987.

[7] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of POPL-87*, Munich, 1987.

[8] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In *Logic Programming: Proceedings of the 4th International Conference*, pages 196–218, The MIT Press, 1987.

[9] Richard O'Keefe. *Practical Prolog for Real Programmers.* Fifth International Conference Symposium on Logic Programming, 1988. Tutorial No.8.