# AP: An Assertional Programming System

Mantis H.M. Cheng
Keitaro Yukawa

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1

Research Report CS-86-11
April 1986

# AP: An Assertional Programming System

*Mantis H.M. Cheng*
*Keitaro Yukawa*

Logic Programming and Artificial Intelligence Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada. N2L 3G1
Net address: {mhmcheng,kyukawa}@waterloo.csnet

## Abstract

We describe AP, a logic programming system based on Horn clauses with equality. AP subsumes relational programming systems based on Horn clauses or (conditional) equations; it allows relations and functions to be defined mutually recursively in a single program. AP supports most programming paradigms found in functional programming systems, e.g., higher order functions, lazy evaluation and stream processing. Inference involving equality is done not by special inference rules for equality but by SLD-resolution of equality axioms, thereby allowing the exploitation of Prolog implementation techniques. The equality axioms are chosen to result in small search spaces. Higher order functions are introduced in a way preserving the logical semantics of programs and avoiding computational intractability incurred by the use of higher order logic. By adopting the Schönfinkel-Curry function application operator, functions in AP are defined by first order equations. A prototype of AP has been implemented in Prolog; it is a realisation of *Logic Programming with Equations* as proposed by van Emden and Yukawa [27].

## 1 Introduction

We describe AP, an *assertional programming* system based on Horn clauses with equality. A working prototype of AP has been implemented in Waterloo Unix Prolog [3]. A "pure" AP program is a set of definite clauses in which equations are admitted as atomic formulae. Inference involving equality is done not by special inference rules for equality but by SLD-resolution of equality axioms. AP subsumes relational programming systems based on Horn clauses like Prolog and systems based on (conditional) equations [5,7,9,11,20,21,24,29]. AP supports most programming paradigms found in functional programming systems, e.g., higher order functions, lazy evaluation and stream processing. It allows relations and functions to be defined mutually recursively in a single program. Since AP has an equational subsystem, it can also be used for term rewriting, e.g., abstract data type specification and symbolic computation.

Recently there has been much work on the amalgamation of functional and logic programming [2,6,7,8,9,10,21,22,25,26]. The design of AP has been influenced by Eqlog [12]; we agree with Goguen and Meseguer that Horn clauses with equality is the right formalism for combining functional and logic programming. This approach retains logical semantics—computed answers, including results of function evaluations, can be justified in terms of logical consequence.

What does AP contribute to the amalgamation as compared with all the proposals cited above? Our answer is:

1. equations are computed by SLD-resolution of equality axioms,

2. higher order functions are introduced in such a way as to preserve the logical semantics of programs and to avoid computational intractability incurred by the use of higher order logic.

We regard (1) as especially important; it allows the exploitation of Prolog implementation techniques. All other proposals known to us attempting to incorporate equations into Horn clauses depend on certain special inference rules for equality, most notably narrowing [16]. As far as we know, such inference rules are no more efficient than SLD-resolution of equality axioms, and the implementation techniques for such inference rules are not as well developed as those for SLD-resolution.

This paper is organised as follows: Section 2 explains the background on equality, Section 3 provides an overview of programming in AP, Section 4 explains how higher order functions are introduced, Section 5 demonstrates the expressiveness of AP programs, Section 6 provides a logical basis of AP, Section 7 describes the implementation, and Section 8 discusses some possible future developments and extensions.

## 2 Background on Equality

The standard set $Eq$ of equality axioms (reflexivity, symmetry, transitivity and substitutivity for function symbols and predicate symbols) is a set of definite clauses:

$$
\begin{array}{ll}
Eq = \{ & \\
\quad x = x \leftarrow; & \text{(reflexivity)} \\
\quad x = y \leftarrow y = x; & \text{(symmetry)} \\
\quad x = z \leftarrow x = y, y = z; & \text{(transitivity)} \\
\quad f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \leftarrow x_1 = y_1, \ldots, x_n = y_n; & \text{(function substitutivity)} \\
\quad p(y_1, \ldots, y_n) \leftarrow x_1 = y_1, \ldots, x_n = y_n, p(x_1, \ldots, x_n); & \text{(predicate substitutivity)} \\
\quad \}. &
\end{array}
$$

By the completeness of SLD-resolution, any complete SLD-refutation procedure will find a refutation of

$$P \cup Eq \cup \{\leftarrow G\}, \tag{1}$$

whenever it is unsatisfiable, where $P$ and $\leftarrow G$ are a logic program and a goal clause possibly with equations. However, the search space of (1) is large and has many infinite branches and useless answers [27]. It is not amenable to efficient search by naive SLD-refutation procedures, e.g., typical Prolog interpreters which use the left-to-right depth-first search strategy.

For this reason, we abandon $Eq$ and use instead two other sets of equality axioms (cf. Section 7). The equality axioms are chosen so that the Prolog interpreter finds efficiently the desired SLD-refutations for a restricted but useful class of Horn clauses with equality. They are the extensions of the equality axioms presented in [27].

## 3 Programming in AP

AP accepts the following types of inputs: programs, terms to be evaluated, queries, system directives and macro definitions.

A "pure" AP program is a set of definite clauses in which equations are admitted as atomic formulae. An AP program may contain any nonlogical features inherited from the underlying Prolog system.

Functions are defined by a set of definite clauses of the form (which we call conditional equations):

$$F = E \leftarrow C_1, C_2, \ldots, C_n; \qquad (n \geq 0).$$

When $n = 0$, the implication symbol "$\leftarrow$" is omitted. An equation on the left of "$\leftarrow$" is used directionally, from left to right.

For example, the following two clauses define the factorial function `fac`:

```
| fac(0) = 1 ;
| fac(N) = N * fac(N-1) <- N > 0 ;
```

The infix operators "*" and "-" denote the arithmetic multiplication and subtraction functions, and ">" denotes the *greater than* relation. "|" is the AP system prompt. Any identifier which begins with an upper case letter is a logical variable, otherwise it is a constant, function or predicate symbol. To compute the factorial of 5, we enter:

```
| fac(5) ;          % a function application
120
| a = 5 ;           % say that a is equal to 5
| fac(a) ;
120
```

Evaluable functors may occur as arguments of predicates, both in conclusions and in conditions.

```
| jekyl = hyde ;
| father(john) = hyde ;
| member(father(john), birthday_club) <- ;     % a fact
| ?member(jekyl, birthday_club) ;              % a query
yes
```

Macros are defined by the relation *is*. A macro definition of the form "E is F" is used to replace every occurrence of E in subsequent input terms by the term F. All such textual replacements are done at compile time—before execution.

## 4 Higher Order Functions

Higher order functions are introduced into AP without using higher order logic. This is done by adopting the Schönfinkel-Curry function application operator [4,23] and defining functions by first order equations.

The admissible terms in AP programs are defined as follows:

1. a variable is a term,
2. a constant symbol is a term,
3. if $t$ and $u$ are terms, so is $t : u$.

":" is an infix left-associative binary function symbol representing the function application operator; the term $t : u$ is the result of applying $t$ to $u$. Parentheses may be used for grouping. Such terms are known as *curried* terms. *All* terms in an AP program are curried. As curried notation may be unfamiliar to the user, AP accepts an alternative notation "$f(t_1, ..., t_n)$" for "$f : t_1 : \cdots : t_n$" when $f$ is a constant symbol or a variable.

For example, AP translates the second clause for the factorial function shown in Section 3 into:

```
| fac:N = *:N:(fac:(-:N:1)) <- N > 0 ;
```

The `twice` function, commonly expressed in $\lambda$-calculus as $twice = \lambda f.\lambda x.f(f(x))$, is defined in AP using the equation:

```
| twice:F:X = F:(F:X) ;
| succ(X) = X+1 ;                  % the successor function
| twice:twice:twice:succ:1 ;       % a sample application
17
```

The `compose` function, $compose = \lambda g.\lambda f.\lambda x.g(f(x))$, is defined in AP as:

```
| compose:G:F:X = G:(F:X) ;
| compose:(twice:succ):fac:3 ;     % a sample application
8
```

## Semantics of Higher Order Functions

Let $P$ be a set of definite clauses. Let $\equiv$ be the congruence relation over the Herbrand universe $H_P$ defined by:

$$\forall t, u \in H_P, \quad t \equiv u \quad \overset{def}{\longleftrightarrow} \quad P \cup Eq \models t = u,$$

where $Eq$ is the standard equality axioms. The *standard model*, due to Goguen and Meseguer [12], has as domain the quotient domain $H_P/\equiv$. Let $[t]$ be the congruence class of $t$. An $n$-place relation $p$ holds of $([t_1], ..., [t_n])$ in this model iff $P \cup Eq \models p(t_1, ..., t_n)$. A functor $f$ is assigned the function $f^*$ such that $f^*([t_1], ..., [t_n]) = [f(t_1, ..., t_n)]$.

In the standard model of an AP program, the function application operator ":" denotes the function such that $[t] : [u] = [t : u]$, and a ground term $t$ denotes $[t]$. A congruence class $[t]$ can be thought of as *representing* a function

$$f_{[t]} : H_P/\equiv \longrightarrow H_P/\equiv$$

such that

$$f_{[t]}([u]) = [t : u], \quad \forall [u] \in H_P/\equiv .$$

Clearly, $f_{[t]}([u]) = [t] : [u], \forall [t], [u] \in H_P/\equiv$. For this reason, every object in the universe of an AP program, i.e., a congruence class $[t]$, can be thought of as representing a unary function producing a unary function, i.e., $f_{[t]}$ [19].

We can define relations, and solve equations, over whatever functions we define, e.g., `twice`, `compose`, and whatever partial applications of such functions, e.g., `twice:succ`, `compose:fac`. (Note that such terms do exist in the Herbrand universe of the program.) We believe that this opens an interesting view of functions in logic programming.

## 5 Programming Examples

In this section, we show by examples how a declarative mixture of functional and logic programming is achieved in AP.

### Higher Order Functions

The following higher order function `accumulate` computes:

$$F(A)\ Op\ F(Next(A))\ Op\ \cdots\ Op\ F(B), \quad A \leq B,$$

where $Op$ is a binary operation and $F$ is a unary function, and when $A > B$, `Init` is the value of `accumulate` (cf. page 56 of [1]).

```
| accumulate(Op,Init,F,A,Next,B) = Init <- A > B ;
| accumulate(Op,Init,F,A,Next,B) =
|    Op(F(A),accumulate(Op,Init,F,Next(A),Next,B)) <- A =< B ;
| square(X) = X * X ;
| succ(X) = X + 1 ;
| accumulate(+,0,square,1,succ,5) ;   % a sample application
55
```

### Equation Solving

AP can solve some simple equations. Note that every object in an AP program represents a function. `[]` denotes the empty list; `[H|T]` denotes a nonempty list with `H` as the head and `T` as the tail.

```
| app([],L) = L ;                  % the append function
| app([H|T],L) = [H|app(T,L)] ;
| ?app(X,[c,d]) = [a,b,c,d] ;      % sample queries
X = [a,b]
| ?app([a|X],[c,d]) = app([a,b,c],[d]) ;
X = [b]
| map(F,[]) = [] ;                 % the higher order map function
| map(F,[X|L]) = [F(X)|map(F,L)] ;
| succ(X) = X + 1 ;
| square(X) = X * X ;
| map(succ,[1,2,3]) ;                 % sample applications and queries
[2,3,4]
| ? map(F,[1,2,3]) = [2,3,4] ;
F = succ
| ? [1,4,9] = map(F,[1,2,3]) ;
F = square
| ? M(succ,[1,2,3]) = [2,3,4] ;
M = map
```

## Stream Processing

The following set of equations and macros defines the infinite stream of Fibonacci numbers. The functions delay and force are those described in [13] (cf. pages 214–239), which are primitives incorporated into the equality axioms which give the effect of applicative order reduction (cf. Section 7).

```
| cons_stream(X,Y) is [X|delay(Y)] ;      % a macro
| head([X|Y]) = X ;
| tail([X|Y]) = force(Y) ;
% we assume that the streams S1 and S2 are infinite
| sum(S1,S2) = cons_stream(head(S1)+head(S2),sum(tail(S1),tail(S2))) ;
| fibs = cons_stream(1,cons_stream(1,sum(fibs,tail(fibs)))) ;
% compute the 6th Fibonacci number
| head(tail(tail(tail(tail(tail(fibs)))))) ;
8
```

If we use the equality axioms which give the effect of normal order reduction (cf. Section 7), we can dispense with delay and force.

```
| head([X|Y]) = X ;
| tail([X|Y]) = Y ;
| sum(S1,S2) = [head(S1)+head(S2)|sum(tail(S1),tail(S2))] ;
| fibs = [1,1|sum(fibs,tail(fibs))] ;
| head(tail(tail(tail(tail(tail(fibs)))))) ;
8
```

## Abstract Data Type Specification

The following set of clauses defines an operation insert and a relation member on the abstract data type *binary search tree*:

```
| insert(Item,nil) = node(nil,Item,nil) ;
| insert(Item,node(L,Item,R)) = node(L,Item,R) ;
| insert(Item,node(L,Key,R)) = node(insert(Item,L),Key,R) <-
|         Item < Key ;
| insert(Item,node(L,Key,R)) = node(L,Key,insert(Item,R)) <-
|         Item > Key ;
| member(Item,node(L,Item,R)) <- ;
| member(Item,node(L,Key,R)) <- Item < Key, member(Item,L) ;
| member(Item,node(L,Key,R)) <- Item > Key, member(Item,R) ;
| ?member(4,insert(2,insert(4,insert(3,nil)))) ;   % a query
yes
```

## Term Rewriting

The following set of equations, which is confluent and terminating, specifies the theory of Groups [18]:

```
| X + 0 = X ;
| 0 + X = X ;
| X + (-X) = 0 ;
| (-X) + X = 0 ;
| (X + Y) + Z = X + (Y + Z) ;
| -(0) = 0 ;
| -(-X) = X ;
| -(X + Y) = (-X) + (-Y) ;
| X + (-X + Y) = Y ;
| (-X) + (X + Y) = Y ;
| (d + (c + (-c))) + (-(0 + (-a))) ;   % a sample rewriting
d + a
```

## Sorting

The following program sorts a list of numbers with respect to a function $F$ evaluated at these numbers, i.e., the sorted list $[x_1, x_2, \ldots, x_n]$ has the property that $F(x_1) \leq F(x_2) \leq \cdots \leq F(x_n)$.

```
| isort(F,[],[]) <- ;
| isort(F,[Head|Tail],Sorted) <-
|        isort(F,Tail,NewTail),
|        insert(F,Head,NewTail,Sorted) ;
| insert(F,H,[],[H]) <- ;
| insert(F,H,[X|Y],[H,X|Y]) <- F(H) =< F(X) ;
| insert(F,H,[X|Y],[X|Y']) <- F(H) > F(X), insert(F,H,Y,Y') ;
| f(X) = 4 + 2*X - X*X ;
| ? isort(f,[1,5,-1],L) ;      % a sample query
L = [5,-1,1]
```

## 6 Logical Basis of AP

We present the correctness, and the completeness under *ground confluence* (see below), of a set of equality axioms from which the two sets of equality axioms used for the implementation are derived. Throughout this section, $P$ denotes a set of definite clauses in which equations are admitted as atomic formulae.

Let $Eq'$ be the following set of equality axioms:

$$\begin{aligned} Eq' = \quad \{ \quad & eq_1(x,y) \leftarrow eq_2(x,z), eq_2(y,z); & (2)\\ & eq_2(x,x) \leftarrow; & (3)\\ & eq_2(x,z) \leftarrow eq_3(x,y), eq_2(y,z); & (4)\\ & eq_3(x,y) \leftarrow x = y; & (5)\\ & eq_3(f(x_1,\ldots,x_i,\ldots,x_n), f(x_1,\ldots,y_i,\ldots,x_n)) \leftarrow eq_3(x_i,y_i); & (6)\\ & \qquad (\forall\, 1 \le i \le n) & \\ \}. \end{aligned}$$

(6) is included for all non-constant function symbols. The relation $eq_3$ is the one-step reduction over ground terms (to be defined below), $eq_2$ is the transitive-reflexive closure of the one-step reduction and $eq_1$ is the relation *$x$ and $y$ reduce to some common term.*

We define the *homogeneous form* of a clause:

$$p(t_1, ..., t_n) \leftarrow B_1, ..., B_m; \quad m, n \ge 0,$$

to be:

$$p(x_1, ..., x_n) \leftarrow x_1 = t_1, ..., x_n = t_n, B_1, ..., B_m;$$

where $x_1, ..., x_n$ are $n$ different variables not occurring in the original clause. To use $Eq'$ to compute with $P$, we transform $P$ into $P_T$ by performing the following two steps:

1. replace each clause in $P$, except for conditional equations (i.e., clauses whose conclusion is an equation), by the homogeneous form,

2. replace, in the set resulting from Step 1, every equation $M = N$ in the condition of a clause by $eq_1(M, N)$.

A transformed goal clause $G_T$ is obtained from $G$ by replacing every equation $M = N$ in $G$ by $eq_1(M, N)$.

The ***reduction*** relation $\rightarrow_P$ associated with $P$ is defined on the set of all ***ground*** terms, i.e., the Herbrand universe of $P$, as follows.

**Definition** For all ground terms $M$ and $N$,

$$M \rightarrow_P N \quad \overset{def}{\longleftrightarrow} \quad P_T \cup Eq' \models eq_3(M, N). \quad \Box$$

When $P$ is a set of equations which can be used as a term rewriting system [14,15], the reduction $\rightarrow_P$ coincides with the usual definition of reduction restricted to ground terms. When $P$ is a set of clauses which can be used as a set of ***conditional rewrite rules*** in the sense of Kaplan [17], the reduction $\rightarrow_P$ coincides with the reduction defined by Kaplan for conditional rewrite rules, restricted to ground terms. Let $\overset{*}{\rightarrow}_P$ be the transitive-reflexive closure of $\rightarrow_P$ .

**Definition** $\rightarrow_P$ is said to be ***ground confluent*** iff for all ground terms $M$, $N$, and $S$, $S \overset{*}{\rightarrow}_P M$ and $S \overset{*}{\rightarrow}_P N$ imply that there is a ground term $T$ such that $M \overset{*}{\rightarrow}_P T$ and $N \overset{*}{\rightarrow}_P T$. A logic program $P$ is said to be ground confluent iff $\rightarrow_P$ is ground confluent. $\Box$

**Definition** A ground term $M$ is a *P-canonical* (or *P-normal*) term iff there is no term $N$ such that $M \rightarrow_P N$. For all ground terms $M$ and $N$, $N$ is a $P$-canonical form of $M$ iff $N$ is $P$-canonical and $M \overset{*}{\rightarrow}_P N$. $\Box$

The following two theorems state the correctness, and the completeness under ground confluence, of $P_T \cup Eq'$ with respect to $P \cup Eq$, where $Eq$ is the standard equality axioms.

**Theorem** (Correctness of transformed programs and $Eq'$) Let $P$ be a logic program and $G$ a goal clause $\leftarrow B_1, ..., B_m$. Let $\theta$ be any computed answer substitution for $P_T \cup Eq' \cup \{G_T\}$. Then

$$P \cup Eq \models \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

**Theorem** (Completeness under ground confluence of transformed programs and $Eq'$) Let $P$ be a ground confluent logic program and $G$ a goal clause $\leftarrow B_1, ..., B_m$. Let $\theta$ be any substitution for the variables of $G$ such that

$$P \cup Eq \models \forall(B_1 \wedge \cdots \wedge B_m)\theta.$$

Then there is an SLD-refutation of $P_T \cup Eq' \cup \{G_T\}$ with the computed answer substitution $\sigma$ such that $\theta = \sigma\gamma$ for some substitution $\gamma$.

# 7 Implementation

## The Equality Axioms

$Eq'$ presented in the previous section is amenable to execution by a Prolog interpreter. But it has the disadvantage that, given a goal clause of the form $\leftarrow eq_1(t,x)$ or $\leftarrow eq_1(x,t)$, $x$ is instantiated to $t$, if the clauses are selected in the order shown. To prevent this, it is necessary to switch the order of the clauses (3) and (4) so that the Prolog interpreter performs reduction until a term has been reduced to a canonical form. It is also necessary to add a clause (i.e., the second clause in $Eq_n$ below) to prevent reduction of variables.

Thus we use the following set $Eq_n$ of equality axioms, to be adjoined to a transformed AP program.

```
Eqn = {
        eq1(X,Y) ← eq2(X,Z), eq2(Y,Z) ;
        eq2(X,Y) ← is_var(X), eq(X,Y) ;              % "eq" is unification
        eq2(X,Z) ← eq3(X,Y), eq2(Y,Z) ;              % transitivity
        eq2(X,X) ← ;                                 % reflexivity
        eq3(X,Y) ← X = Y ;                           % using equations
        eq3(X1:X2,Y1:X2) ← eq3(X1,Y1) ;              % substitutivity
        eq3(X1:X2,X1:Y2) ← eq3(X2,Y2) ;
        }.
```

When executed by a Prolog interpreter, $Eq_n$ gives the effect of normal order (leftmost-outermost) reduction. For any set $P$ of definite clauses, the first SLD-refutation by the Prolog interpreter of $P_T \cup Eq_n \cup \{\leftarrow eq_1(t,x)\}$ or of $P_T \cup Eq_n \cup \{\leftarrow eq_1(x,t)\}$, where $t$ is ground, instantiates $x$ to a canonical form of $t$.

If there is no need to use infinite data structures, it is preferable to use equality axioms which give the effect of applicative order (innermost) reduction, because it is generally more efficient. For this reason, we have:

$Eq_a$ = {
```
    eq1(X,Y) ← eq2(X,Z,F1), eq2(Y,Z,F2) ;
    eq2(X,Y,_) ← is_var(X), eq(X,Y) ;          % "eq" is unification
    eq2(X,Z,yes) ← eq3(X,Y,yes), eq2(Y,Z,F) ;  % transitivity
    eq2(X,X,no) ← ;                            % reflexivity
    eq3(X:Y,Z,F) ←                             % substitutivity
                eq2(X,X1,F1), eq2(Y,Y1,F2),
                eq4(X1:Y1,Z,F3),               % function application
                or(F1,F2,F3,F) ;               % F is "no" if F1,F2 and F3 are "no"
    eq3(X,Y,yes) ← X = Y ;
    eq4(X,Y,yes) ← X = Y ;                     % using equations
    eq4(X,X,no) ← ;                            % rewrite to itself
    }.
```

In this case to avoid infinite derivations, we need an extra argument indicating whether the term has been rewritten by any reduction steps. As before, for any set $P$ of definite clauses, the first SLD-refutation by the Prolog interpreter of $P_T \cup Eq_a \cup \{\leftarrow eq_1(t,x)\}$ or of $P_T \cup Eq_a \cup \{\leftarrow eq_1(x,t)\}$, where $t$ is ground, instantiates $x$ to a canonical form of $t$. Note that $Eq_a$ is still usable for simple equation solving.

## The Interpreter

The inference engine is the Prolog interpreter. Given an AP program $P$, the parser generates the transformed program $P_T$ (cf. Section 6). Queries are then interpreted by this Prolog program:

```
solve( (Goal,Goals) ) ← try_solve( Goal ), solve( Goals ) ;
solve( Goal ) ← try_solve( Goal ) ;
try_solve( (M = N) ) ← eq1( M, N ) ;          % solve equations
try_solve( G ) ← G ≠ (M = N), prove( G ) ;    % solve non-equational goals
```

## 8 Discussion

We have described a working prototype of AP and demonstrated its expressiveness with examples chosen from different applications. We have yet to gain experience on using both functional and logic programming in one system; we believe that AP will become a useful tool. This section discusses some possible extensions and future developments of AP.

## Parallelism

Since our equality axioms are definite clauses, AND- and OR- parallelism in logic programming remain a feature in AP. Examining $Eq_n$ and $Eq_a$, we note that

1. in $Eq_n$ and $Eq_a$, AND-parallelism can be used to find a common reduced form (the clause for $eq_1$), and OR-parallelism can be used to process multiple conditional equations defining the same function,

2. in $Eq_n$, OR-parallelism can be used to find a unifying subterm (the clauses for $eq_3$),

3. in $Eq_a$, AND-parallelism can be used to reduce subterms (the first clause for $eq_3$).

We take these possibilities as a vindication for our claim made earlier: exploitation of Prolog implementation techniques.

## Incorporating Equality Axioms

Our equality axioms have the status of a user program. This made the correctness of the implementation obvious and the construction of the prototype easy. Although the system is efficient enough for teaching purposes, the equality axioms have to be incorporated into the interpreter for applications demanding more efficiency.

## Introducing Lambda Expressions

Lambda expressions allow the use of functions without definition and locality of expressions within terms. Their utility has been demonstrated by a functional language like Scheme [1]. Also, we may want the system to return a lambda expression whenever it is more informative (e.g., $\lambda$x.succ(succ(x)) instead of `twice:succ`). Work is in progress on introducing lambda expressions into AP.

## Compiling AP

Compiling logic programs or functional programs into conventional machine instructions is well-understood [13,28]. What are suitable machine models onto which AP programs can be compiled?

# Acknowledgements

Professor van Emden provided us with an incentive to build AP. We thank the following people for commenting on an earlier draft of this paper: Romas Aleliunas, Randy Goebel, David Poole and Maarten van Emden. This research is partially supported by the Natural Sciences and Engineering Research Council.

## References

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

[2] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, pages 119–132, Lecture Notes in Computer Science, vol. 213, Springer-Verlag, 1986.

[3] M.H.M. Cheng. *Design and Implementation of the Waterloo Unix Prolog Environment.* Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, 1984.

[4] H.B. Curry and R. Feys. *Combinatory Logic.* Volume 1, North-Holland, Amsterdam, 1958.

[5] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 37–70, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[6] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Relations, and Equations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[7] N. Dershowitz and D.A. Plaisted. Logic programming cum applicative programming. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 54–66, IEEE, July 1985. Boston, Massachusetts.

[8] M. Dincbas and P. Vanhentenryck. *Extended Unification Algorithms for the Integration of Functional and Logic Languages.* manuscript, European Computer-Industry Research Centre, Munich, West Germany, 1986.

[9] L. Fribourg. Oriented equational clauses as a programming language. *The Journal of Logic Programming*, 1(2):165–177, 1984.

[10] U. Furbach and S. Hölldobler. Modelling the combination of functional and logic programming languages. *Journal of Symbolic Computation*, 2(2):123–138, 1986.

[11] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, ACM, January 1985. New Orleans, Louisiana.

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

[13] P. Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall International, London, 1980.

[14] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[15] G. Huet and D.C. Oppen. Equations and rewrite rules, a survey. In R.V. Book, editor, *Formal Language Theory, Perspectives and Open Problems*, pages 349–405, Academic Press, New York, 1980.

[16] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R.A. Kowalski, editors, *5th Conference on Automated Deduction*, pages 318–334, Lecture Notes in Computer Science, vol. 87, Springer-Verlag, 1980.

[17] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2,3):175–193, 1984.

[18] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, Pergamon Press, Oxford, 1970.

[19] A.R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1982.

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

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

[22] J.A. Robinson and E.E. Sibert. LOGLISP: motivation, design and implementation. In K.L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 299–313, Academic Press, London, 1982.

[23] M. Schönfinkel. On the building blocks of mathematical logic. In J. van Heijenoort, editor, *From Frege to Gödel, A Source Book in Mathematical Logic, 1879-1931*, pages 355–366, Harvard University Press, Cambridge, Massachusetts, 1967.

[24] G. Smolka. FRESH: a higher-order language with unification and multiple results. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 469–524, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[25] A. Srivastava, D. Oxley, and A. Srivastava. An(other) integration of logic and functional programming. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 254–260, IEEE, July 1985. Boston, Massachusetts.

[26] H. Tamaki. Semantics of a logic programming language with a reducibility predicate. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 259–264, IEEE, February 1984. Atlantic City, New Jersey.

[27] M.H. van Emden and K. Yukawa. *Logic Programming with Equations.* Technical Report CS-86-05, Department of Computer Science, University of Waterloo, Waterloo, Ontario, October 1986. revised version.

[28] D.H.D. Warren. *Implementing PROLOG — Compiling Predicate Logic Programs.* D.A.I Research Report No. 39, Department of Artificial Intelligence, University of Edinburgh, 1977.

[29] J.-H. You and P.A. Subrahmanyam. Equational logic programming: an extension to equational programming. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 209–218, ACM, January 1986. St. Petersburg Beach, Florida.