That scripting language called Prolog

Sergei Winitzki

Types, Theorems, and Programming Languages

June 16, 2014

The four programming paradigms

Paradigm	Example	Programs are...	Difficulty is in...	Best used in...
Imperative	Fortran	lists of commands	order of updates	numerics
Functional	Lisp	expressions	level of abstraction	compilers
Logic	Prolog	sets of predicates	runtime behavior	symbolic AI
Constraint	$\TeX$	data + constraints	conflicting modules	specific domain

- Objects, type systems, modules, concurrency, etc., are merely *features* added on top of a *chosen paradigm*

A definition of "declarative"

Programming is "declarative" when *specifications are programs.*
More formally:
A language L is **declarative for an application domain** D if:

- The domain D has a *good* specification formalism F
  - "good" = *visual, pragmatically convenient,* complete for the domain D
- There is an obvious *syntactic* transformation from F to L
- The resulting program *implements the specification*

Less formally:

- A declarative language is a "perfect DSL" for the given domain

Example: declarative FORTRAN 77

- Application domain: numerical mathematical expressions
- Specification: a mathematical formula involving *numbers* and *functions*
- Example specification:

$$f(x,p,q) = \frac{\sin px}{x^2} - \frac{\sin^2 qx}{x^3}$$

  - Implementation: `F(X,P,Q)=SIN(P*X)/X**2-(SIN(Q*X))**2/X**3`

- For more complicated tasks, FORTRAN is not declarative

$$\tilde{X}_k = Y_k - \sum_{j=k+1}^{n} A_{kj} X_j, \quad \forall k \in [1..n]$$

```
      X(N)=Y(N)/A(N,N)
      DO 10 K=N-1,1,-1
         S=0.
         DO 20 J=K+1,N
20          S=S+A(K,J)*X(J)
10       X(K)=(Y(K)-S)/A(K,K)
```

(example code, 1987)

Example: declarative Haskell 98

- Application domain: recursively defined, algebraic data structures
- Specifications: inductive definitions of functions on ADTs
- Example (from R. Sedgewick, *Algorithms in C*, 1998)

Definition 5.1 *A* **binary tree** *is either an external node or an internal node connected to a pair of binary trees, which are called the left subtree and the right subtree of that node.*

This definition makes it plain that the binary tree itself is an ab-

```
data BTree α = BTNode α | BTVertex α (BTree α) (BTree α)
```

Example: declarative Haskell 98, continued

Definition 5.6 *The* **level** *of a node in a tree is one higher than the level of its parent (with the root at level 0). The* **height** *of a tree is the maximum of the levels of the tree's nodes. The* **path length** *of a tree is the sum of the levels of all the tree's nodes. The* **internal path**

```
height ::  BTree α → Int
height (BTNode _) = 0
height (BTVertex _ t1 t2) = 1 + max (height t1) (height t2)
```

Example: non-declarative Haskell

For a different application domain, Haskell is *not* declarative!

- Downloading data from server (from "*Real World Haskell*", 2008)

```
download dbh logf =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
       logf "Download complete."
    where procPodcast pc =
              do logf $ "Considering " ++ (castURL pc)
                 episodelist <- getPodcastEpisodes dbh pc
                 let dleps = filter (\ep -> epDone ep == False)
                             episodelist
                 mapM_ procEpisode dleps
          procEpisode ep =
              do logf $ "Downloading " ++ (epURL ep)
                 getEpisode dbh ep
```

Prolog as a DSL for logic puzzles

All jumping creatures are green. All small jumping creatures are martians. All green martians are intelligent.
Ngtrks is small and green. Pgvdrk is a jumping martian.
Who is intelligent? (inpired by S. Lem, *Invasion from Aldebaran*)

```
small(ngtrks). green(ngtrks).
martian(pgvdrk). jumping(pgvdrk).
green(X) :- jumping(X).
martian(X) :- small(X), jumping(X).
intelligent(X) :- green(X), martian(X).
main :-
  intelligent(X), format('~w is intelligent.~n', X), halt.
```

```
$ swipl -o martians -q -t main -c martians.pl
$ ./martians
pgvdrk is intelligent.
```

Prolog in a nutshell 1: symbols, predicates, rules

- Outer-level lowercase symbols are **logical predicates**
  - All other lowercase symbols are symbolic constants
- All capitalized symbols are **quantified logical variables** (LVs)
- LVs on the left imply $\forall$; free LVs on the right imply $\exists$
- The symbol `:-` means implication $\leftarrow$, the comma means "*and*"
- Rules can be given in any order; no declarations
- Examples:
  `green(X) :- jumping(X)` means $\forall X : green(X) \leftarrow jumping(X)$
  `path(A,B) :- path(A,C), path(C,B)` means
  $\forall A, B : [\exists C : path(A, C) \wedge path(C, B) \rightarrow path(A, B)]$
- `main :- intelligent(X)` means: prove that $\exists$`X:intelligent(X)`

Prolog in a nutshell 2: variables

- "Martians" recap:

```
small(ngtrks). green(ngtrks).
martian(pgvdrk). jumping(pgvdrk).
green(X) :- jumping(X).
martian(X) :- small(X), jumping(X).
intelligent(X) :- green(X), martian(X).
```

- `main :- intelligent(X)` means: prove that $\exists$`X:intelligent(X)`
- The Prolog engine will *prove existence* of X by backtracking search! (we can say "`trace`" and follow the search)

Prolog in a nutshell 2: backtracking search

"Martians" recap:

```
small(ngtrks). green(ngtrks).
martian(pgvdrk). jumping(pgvdrk).
green(X) :- jumping(X).
martian(X) :-
small(X), jumping(X).
intelligent(X) :-
green(X), martian(X).
?- intelligent(X).
```

```
[trace]  ?- intelligent(X).
  Call: (6) intelligent(_G2442)
  Call: (7) green(_G2442)
  Exit: (7) green(ngtrks)
  Call: (7) martian(ngtrks)
  Call: (8) small(ngtrks)
  Exit: (8) small(ngtrks)
  Call: (8) jumping(ngtrks)
  Fail: (8) jumping(ngtrks)
  Fail: (7) martian(ngtrks)
  Redo: (7) green(_G2442)
  Call: (8) jumping(_G2442)
  Exit: (8) jumping(pgvdrk)
  Exit: (7) green(pgvdrk)
  Call: (7) martian(pgvdrk)
  Exit: (7) martian(pgvdrk)
  Exit: (6) intelligent(pgvdrk)

X = pgvdrk .
```

- The proof may fail, or may succeed in more than one way
- LVs are assigned by unification and unassigned on backtracking
- Once assigned, a logical variable is *immutable*
  - This is called "resolution of Horn clauses" and "unification"

Horn clauses and "resolution"

- Consider a restricted fragment of predicate logic:
  - Expressions are conjunctions of "Horn clauses":
    $$(a \wedge b \wedge ... \wedge c \rightarrow d) \wedge (p \wedge q \wedge ... \wedge r \rightarrow s) \wedge (\text{True} \rightarrow w) \wedge ...$$
  - Disjunction is expressible through Horn clauses:
    $$(a \vee b) \rightarrow c = (a \rightarrow c) \wedge (b \rightarrow c)$$
  - Only $\forall$ quantifiers are allowed, and only outside:
    $$\forall X \forall Y : a(X, Y) \wedge b(Y) \rightarrow c(X)$$
- Prolog syntax: quantifiers are implicit; implication points leftward
- "Resolution": the "b" is eliminated from two clauses

$$(c \leftarrow a \wedge p \wedge q) \Leftarrow \begin{cases} b \leftarrow a \wedge p \\ c \leftarrow b \wedge q \end{cases}$$

Examples: recursive predicates

- Factorial predicate: fact(N,F) holds if $F = N!$

```
fact(1,1).
fact(N,F) :- M is N-1, fact(M,E), F is E*N.
```

- Fibonacci predicate: fibo(N,F) holds if F is N-th Fibonacci number

```
fibo(0,1). fibo(1,1).
fibo(N,F) :- M1 is N-1, fibo(M1,E1),
     M2 is N-2, fibo(M2,E2), F is E1+E2.
```

- Instead of computing F through calling a function, we *prove existence* of a value F such that some predicate holds.
- Prolog has *only predicates* — no declarations, expressions, functions, variables, or static types
  - Note: `M is N-1` resembles an expression but is actually a predicate!
  - it means `is(M,'-'(N,1))` , where `'-'` is an infix data constructor!

Prolog in a nutshell 2: syntax extensions

- Syntax for data structures: "passive predicates" (a distinct namespace)
  `left(pair(X,Y), X). right(pair(X,Y), Y).`
  - since `pair(X,Y)` is *inside* a predicate, it must be data
  - however, `pair(X,Y)` can contain *free* logical variables
  - and so can contain *any* other structures (no typechecking!):
    `pair(pair(X,Y),pair(Y,pair(Z,T)))`
- Syntax sugar for lists: `[]` or `[a,b,c]` or `[a,b,c|[]]`
  - Similar to Haskell's `a:b:c:[]`
  - Examples:

```
head([X|Xs], X). tail([X|Xs], Xs).  empty([]).
length([], 0).
length([X|Xs],N) = length(Xs,M), N is M+1.
```

- User-defined syntax: infix, prefix, postfix
  `op(400, xfy, *).  op(300, yfx, ^).  op(200, xf, :!).`

What is "unification"?

- Unification is Prolog's way of "assigning values to variables"
- Pattern-matching of recursive structures, fitting unassigned variables
- Assignment is propagated to other rules:

```
like(1). like(2).
like(you_know(Y,X)) :- like(X), like(Y).
really(A,B) :- like(A), like(you_know(A,B)).
```

- The predicate really(1,2) holds because like(you_know(Y,X)) is unified with like(you_know(1,2)) to set $Y = 1$ and $X = 2$

“Pointers made safe”: difference lists

- Appending ordinary lists takes $O(n)$ operations:
```
l_append([], X, X).
l_append([X|Xs], Y, [X|Zs]) :- l_append(Xs, Y, Zs).
```
- To optimize, we need a “pointer to the end of the list”!
  - “having a pointer” = a part of a data structure is *not yet assigned*
  - solution: a free LV is exposed, unified with some part of the data
- **Difference lists**: `pair([a,b,c|X],X)` or `[a,b,c|X]-X`
```
op(500, xfy, -).  empty(X-X).
dl_append(X-Y,Y-Z,X-Z). /* O(1) operations!  */
```
```
[trace] ?- dl_append( [a,b,c|A]- A, [d,e|B]-B, C).
dl_append([a, b, c|_G2447]-_G2447, [d, e|_G2456]-_G2456, _G2463)
dl_append([a, b, c, d, e|_G2456]-[d, e|_G2456], [d, e|_G2456]-_G2456,
    [a, b, c, d, e|_G2456]-_G2456)

A = [d, e|B],   C = [a, b, c, d, e|B]-B.
```

"Pointers made safe": difference lists

How this works in more detail:

- We have the program `dl_append(X-Y,Y-Z,X-Z).`
- We have the query `?- dl_append([a,b,c|A]-A, [d,e|B]-B, C).`
- The value of `Y` can be unified with the query structure only if we assign `Y=A` and also `Y=[d,e|B]` and `Z=B`.
- Thus we must have `A=[d,e|B]`
- Therefore `X` must be assigned `[a,b,c|A]=[a,b,c,d,e|B]`
- Finally `C=X-Z=[a,b,c,d,e|B]-B`

“Pointers made safe”: queues

- Implement insertion at end of queue:

```
list_to_queue(L,Q-Y) :- l_append(L,Y,Q). /* O(n) */
q_insert_at_end(E, Q-[E|Y], Q-Y). /* O(1) */
q_head(E, [E|Q]-Y, Q-Y).
```

```
[trace]
?- q_insert_at_end( n, [a,b,c|X]-X, Q ).
X = [n|_G1726],    Q = [a, b, c, n|_G1726]-_G1726.
?- q_head( X, [a,b,c,n|Y]-Y, Z).
X = a,     Z = [b, c, n|Y]-Y.
```

“Pointers made safe”: iterators

- A list iterator: set at begin, increment, check if at end
- Split the list into a pair of queues
- Effectively, we have a pointer to the *middle* of a list:

```
:- op(600, xfx, ^^).
plist_at_begin(X-X ^^ A-B). plist_at_end(A-B ^^ X-X).
plist_incr(A-[X|B] ^^ [X|C]-D, A-B ^^ C-D).
```

```
[trace]  ?- plist_incr([a,b|X]-X ^^ [c,d,e|Y]-Y, P).
plist_incr([a, b|_G1978]-_G1978 ^^ [c, d, e|_G1990]-_G1990, _G1999)
plist_incr([a, b, c|_G2113]-[c|_G2113] ^^ [c, d, e|_G1990]-_G1990,
    [a, b, c|_G2113]-_G2113 ^^ [d, e|_G1990]-_G1990)
X = [c|_G2113],   P = [a, b, c|_G2113]-_G2113  ^^ [d, e|Y]-Y.
```

Parsing with Definite Clause Grammars

- Top-down parsing with unlimited backtracking...

```
expr ::== term | term oper expr | '(' expr ')'
oper ::== 'and' | 'or' | ...
term ::== 'true' | 'false' | ...
```

- ...is similar to Prolog's evaluation strategy on *token queues*:

```
expr(Ts) :- term(Ts).
expr(Ts-X) :- term(Ts-T1), op(T1-T2), expr(T2-X).
oper([T|X]-X) :- T='and'.  oper([T|X]-X) :- T='or'.  /* etc.  */
term([T|X]-X) :- T='true'.  term([T|X]-X) :- T='false'.  /* etc.  */
```

- Syntactic sugar (`-->`) is provided to avoid writing out the queues:

```
expr --> term.  expr --> term, oper, expr.  expr --> ['('], expr, [')'].
oper --> ['and'].  oper --> ['or'].  term --> ['true'].  term --> ['false'].
/* test:  */ ?- expr(['true', 'and', 'false', 'or', 'true'], []).
```

- Nonterminals can have extra arguments and call Prolog code
  - No need for Lex/Yacc (but they do other grammars...)
  - Can parse some non-CFG grammars ("attributed" grammars)

Interpreters

- A lambda-calculus interpreter in 3 lines of Prolog?...

```
:- op(1190, xfy, ~>).  :- op(1180, yfx, @).
comp(A~>B, A~>B). comp( (A~>B) @ A, B).
comp(A@B@C, R) :- comp(A@B,S), comp(S@C,R). comp(A@B, A@B).
```

- ...um, not really (lacking α-conversion and multiple steps)

```
?- comp( (X~>Y~>X) @ 1, Result).
Result = (Y~>1) .  /* okay!  */
?- comp( (X~> X@X) @ (Y~>Y), R1), comp(R1 @ 1,Res).
X = Y, Y = (Y~>Y), R1 = ((Y~>Y)@ (Y~>Y)), Res = ((Y~>Y)@1) .  /* ???  */
```

- A Prolog module with about 15 clauses achieves this (hacky!):

```
:- use_module(lambda).
id - (X ~> X). const - (C ~> _ ~> C).
ycomb - (F ~> (X ~> F @ (X @ X)) @ (X ~> F @ (X @ X))).
fac - ( F ~> N ~> iff @ (N F = const; F = (const @ id) ).
?- lambda((y @ fac @ 4), Result).
Result = 24
```

Prolog and relational databases

Alice lost her umbrella on Monday. Brian lost his glasses on Tuesday. Chris lost his pen on Monday. Dennis lost his watch on Monday and his pen on Tuesday.
Whoever loses something on Monday will be unlucky for the whole week.
Who is unlucky for the whole week and also lost something on Tuesday?

```
lost(alice, umbrella, monday).  lost(brian, glasses, tuesday).
lost(chris, pen, monday).
lost(dennis, watch, monday).  lost(dennis, pen, tuesday).
unlucky(X) :- lost(X, _, monday).
?- unlucky(Who), lost(Who, _, tuesday).
```

- Our predicates are either facts or non-recursive inferences
- We will need *no backtracking* (unlike the "Martians" example)
- A query such as `?- unlucky(X)` will usually return many results

Prolog and relational databases

- We have seen this before... (it is called "SQL")

lost:	person	object	day
	alice	umbrella	monday
	brian	glasses	tuesday
	chris	pen	monday
	dennis	watch	monday
	dennis	pen	tuesday

unlucky:	person
	alice
	chris
	dennis

- Our Prolog code...

```
unlucky(X) :- lost(X, _, monday).
?- unlucky(Who), lost(Who, _, tuesday).
```

- ...is translated into SQL as:

```
CREATE VIEW unlucky AS SELECT person FROM lost WHERE lost.day='monday';
SELECT person FROM unlucky NATURAL JOIN lost WHERE lost.day='tuesday';
```

Prolog, Datalog, and SQL

	SQL	**Datalog**	**Prolog**
recursive data	?		+
recursive queries		+	+
Horn clauses	+	+	+
control flow			+
functions	+		
typed values	+		
named values	+		

Prolog + functions + types = **functional-logic programming**

Functional-logic programming in Mercury

- Mercury is Prolog + some features of ML
  - Immutable values, static type inference, algebraic polymorphism
  - all predicates and all arguments are labeled with "modes"
  - static detection of errors in predicate use
  - functions are deterministic predicates with strict modes
  - products, sums, labeled records, higher-order functions
  - I/O by unique types
  - modules and signatures à la Standard ML
  - type classes à la Haskell
  - standard library: arrays, multithreading, etc.
  - can compile to high-performance machine code, C, Java, Erlang

A taste of Mercury

- Read an integer and print its factorial

```
:- module f.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.
:- implementation.
:- import_module int.
:- pred fact(int::in, int::out) is det.
:- import_module list, string.
fact(N,F) :- ( if N =< 1 then F = 1 else fact(N-1, A), F = A*N ).
main(!IO) :- io.read_line_as_string(Result, !IO),
  ( if Result = ok(String),
       string.to_int(string.strip(String), N)
    then
       io.format("fact(%d) = %d\n", [i(N), i(fact(N))], !IO)
    else
       io.format("That isn't a number...\n", [], !IO)
  ).
```

Prolog: A perfect scripting language

- No declarations, runtime ("dynamic") typing, immutable values
- Data constructors and predicates have user-defined infix syntax
- Pattern-matching on recursive data structures, with backtracking
- Metaprogramming, reflection, self-modifying code
- Easy to do embedded or external DSLs (monadic top-down parsing)
- ISO standard since 1995, many free implementations
    - typically with a REPL and a compiler to high-performance VM
- Interface to databases, networking, multithreading, GUI, ...
- Core language is "small"; full language is hard to use?

Compiling Prolog to a virtual machine

- Warren's Abstract Machine (WAM) is still the most influential
- Instructions assume that WAM manages stack and heap
- Compilation of core Prolog to WAM is relatively straightforward:

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).

concatenate/3: switch_on_term C1a,C1,C2,fail

C1a:    try_me_else C2a              % concatenate(
C1:     get_nil A1                   %    [],
        get_value A2,A3              %    L,L
        proceed                      % ).

C2a:    trust_me_else fail           % concatenate(
C2:     get_list A1                  %    [
        unify_variable X4            %       X|
        unify_variable A1            %       L1], L2,
        get_list A3                  %    [
        unify_value X4               %       X|
        unify_variable A3            %       L3]) :-
        execute concatenate/3        % concatenate(L1,L2,L3).
```

(D. H. D. Warren, 1983)

Conclusions and outlook

- Declarative programming = creating a good DSL for your domain
- It is easy to pick up Prolog, after seeing SQL and Haskell
- Prolog makes building DSLs easy (both internal and external DSLs)
- Mercury = Prolog + types + functions
- Prolog is almost forgotten, but its legacy lives on

Suggested reading

Free implementations I used:

- SWI-Prolog
- The Mercury programming language

A great pedagogical introduction to Prolog and Datalog:

- D. Maier, D. S. Warren. *Computing with logic: Logic programming with Prolog*. Addison-Wesley, 1988

Advanced Prolog programming:

- E. Shapiro, L. Sterling. *The art of Prolog*. MIT, 1999
- T. Van Le. *Techniques of Prolog programming*. Wiley, 1993
- R. O'Keefe. *The craft of Prolog*. MIT, 1990

Implementation (the WAM is still an important source of inspiration):

- H. Aït-Kaci. *Warren's Abstract Machine (WAM)*. MIT, 1991
- W. F. Clocksin. *Design and simulation of a sequential Prolog machine*. New Gen. Comp., 3 (1985), p. 101 (describes the ZIP machine)

Summary

- What is "logic programming" and "constraint programming"
- Prolog in a nutshell
- How Prolog "makes pointers safe"
- Why Prolog was the ultimate scripting language for AI (backtracking search, interpreters, and DSLs for free)
- What is "functional-logic programming" (a taste of the programming language Mercury)