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

BOP 0.2 MANUAL

D. S. Parker

June 1992
CSD-920027

Bop 0.2 Manual[1]

D. Stott Parker
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024-1596

`stott@cs.ucla.edu`

January 1992
Revised June 1992

[1]Copyright ©1990, 1991, 1992 D. Stott Parker
Unix is a trademark of AT&T Bell Laboratories.
Miranda is a trademark of Research Software Limited.
Development of Bop has been supported by NSF Grant IRI-8917907

Bop is a portable extension of Edinburgh-style Prolog environments, that integrates term rewriting with Prolog. Bop rests on a kind of conditional rewrite rules, that supports various term rewriting styles. The default rewriting mechanism is nondeterministic and lazy. It offers the following:

- control over the compilation of rewrite rules, permitting user-defined pattern matching and/or type checking.
- a rule compiler that produces fairly efficient Prolog code by avoiding redundant computation and exploiting features of modern Prolog systems such as clause indexing and delay primitives.
- ability to produce compiled rule output as a Prolog file that can subsequently be used independently in other applications.
- a full debugger/tracer with spypoint management for rewrite rules.
- a sizeable library of standard programs.
- a collection of demonstration programs, with a modest demo interaction shell encouraging exploration.
- a system written almost entirely in Prolog, intended to be portable across Edinburgh Prolog platforms. Bop has been run successfully on the C-Prolog, SICStus, and Quintus Prolog systems.

Bop was developed particularly in order to support research on stream processing and constraint processing, but it should also be of help in the exploration of other paradigmatic extensions of logic programming. This manual overviews the system and describes the primitives available, emphasizing its application to stream processing problems.

Contents

Chapter 1

Overview of Bop

Bop is an extension of Prolog that permits the user to write logic programs including rewrite rules. These rules all have the form

LHS => *RHS* :- *Condition*.

Here *Condition* is a logical condition that determines whether the rewrite rule should be used. Such rules are usually called *conditional rewrite rules*. There has been a great interest in conditional rewrite rules since the mid-1980's, since they combine many interesting paradigms in an apparently natural way. Among other things, it has been argued that the rules add the flavor of functional programming to logic programming, and encourage new styles of programming.

1.1 The Essence of Bop

Bop integrates conditional rewrite rules with Prolog, in a fairly thorough way, so the result is more coherent than a jumble of features or loosely-tied subsystems. In addition, Bop is intended as a portable and practical extension of Prolog, usable by Prolog programmers.

Bop offers the following:

- generalized conditional rewrite rules, and a nondeterministic, lazy term rewriting system. This system is flexible and accomodates various term rewriting styles, including *complete term rewriting* (Knuth-Bendix-like systems and systems with term orderings) and *regular term rewriting* (Huet-like confluent systems, lambda calculus, combinatory logic, systems with constructors).

- control over the compilation of rewrite rules, permitting user-defined pattern matching (outermost or innermost matching, syntactic or semantic unification, etc. – matching may be performed in any way the user desires, on an argument-by-argument basis) and/or type checking.

- a rule compiler that produces fairly efficient Prolog code, avoiding redundant computation and exploiting features of modern Prolog systems such as clause indexing and delay primitives.

- ability to produce compiled rule output as a Prolog file that can subsequently be used independently in other applications.

- a full debugger/tracer with spypoint management for rewrite rules.

- a sizeable library of commonly used functions.

- a collection of demo programs, with a modest demo interaction shell encouraging exploration.
- a system written almost entirely in Prolog and Bop, intended to be portable across Edinburgh Prolog platforms.[1]

Many systems have been proposed that integrate functional and logic programming. These systems usually begin by extending unification to include some kind of equality theory, often an equality theory implemented by term rewriting. Although it is very elegant and powerful, one problem with this approach is that it is usually too inefficient to be useful in practice. Another problem is that the proposals here have not been carried through to complete implementations, and they are not used.

Bop tries to provide a term rewriting capability in Prolog, a capability that is general and robust enough that it can be used in many applications (including the integration of logic and functional programming). In the same way that Definite Clause Grammars have been useful in grammar processing, serving as a mapping from grammar syntax to logic programs, Bop will hopefully be useful in term rewriting, serving as a mapping from Bop rules to logic programs. In order to be practical, Bop's rewriting scheme has been made very flexible, and can emulate diverse rewriting styles. Furthermore, instead of extending the unifier in all cases to include an equational theory, it relies on Prolog's syntactic unification where possible, and relies on extended unification only where required by the user (through the syntax of Bop rules).

Bop was developed particularly in order to support research on term rewriting, stream processing, and constraint processing. It rests on ideas that cut across all of these applications, which are strong enough to stand on their own. Hopefully Bop will also be of help in the exploration of other paradigmatic extensions of logic programming.

1.2 Programming in Bop

Bop rules take either of the forms

LHS => *RHS.*

LHS => *RHS* :- *PrologGoal.*

In these forms, *LHS* and *RHS* are Prolog terms that meet certain restrictions.

1.2.1 Completion Rules

A *completion* is a term that rewrites to itself. Examples of completions in Bop are `[]` and `[_|_]`. Completions are defined by completion rules, which look as follows:

```
[]      =>  [].
[H|T]   =>  [H|T].

0       =>  0.
s(X)    =>  s(X).

X       =>  X  :-  number(X).
```

In a completion rule, *LHS* and *RHS* are identical.

[1]As of this writing, Bop has been run successfully on the C-Prolog, SICStus, and Quintus Prolog systems.

1.2.2 Ordering Rules

Ordering rules state evaluation relationships between terms. Examples of Bop ordering rules are as follows:

```
append([],L)    => L.
append([H|T],L) => [H|append(T,L)].
```

Bop rules satisfy the following restriction:

If LHS is $f(t_1, \ldots, t_n)$, then each t_i is either a variable, or a completion.

Thus in the rules for append, the only arguments are the variable L or the completions [] and [H|T].

1.2.3 Bop Programs

A Bop program is a collection of Bop rules. Bop programs are easily developed in a recursive, functional style. For example, the following is a simple program for computing Ackermann's function:

```
0     =>  0.
s(X)  =>  s(X).

ack(0, 0)          =>  s(0).
ack(0, s(N))       =>  s(s(N)).
ack(s(M), 0)       =>  ack(M, s(0)).
ack(s(M), s(N))    =>  ack(M, ack(s(M),N)).
```

The first two rules define natural number values (in successor notation) as completions, and the remaining rules define Ackermann's function recursively. Had we wished to dispense with the successor notation and use Prolog integer arithmetic in its place, we could have written:

```
X  =>  X  :-  integer(X), X >= 0.

ack(0, 0)    =>  1.
ack(0, N)    =>  N1                  :-  N>=1, N1 is N+1.
ack(M1, 0)   =>  ack(M, 1)           :-  M is M1-1.
ack(M1, N1)  =>  ack(M, ack(M1,N))   :-  M is M1-1, N is N1-1.
```

With this program we find that ack(2,1) evaluates to 5, ack(2,2) to 7, ack(3,1) to 13, and so forth.

1.3 How Bop Works

Consider the following two rules, defining how lists are appended:

```
append([],L)    => L.
append([H|T],L) => [H|append(T,L)].
```

These rules are very much like the usual Prolog clauses, and are executable specifications of how one appends two lists.

Bop rules are compiled to Prolog clauses. The compilation process is straightforward. For example, the two rules above are translated into something functionally equivalent to the following Prolog code:

```
append(A,L) =>> B  :-  A =>> [],      L =>> B.
append(A,L) =>> B  :-  A =>> [H|T],  [H|append(T,L)] =>> B.
```

Here the operator `=>>` is something like the transitive closure of `=>` and can be read as 'evaluates to'. The predicate `=>>` can be used anywhere in a Prolog program in order to evaluate a functional expression, such as `append([1],append([2],[]))`.

The `=>>` rules above show how Bop integrates the benefits of a functional style with the benefits offered by logic programming and Prolog. For example, the Bop rules can operate non-deterministically, just like their Prolog `append` counterparts. Many *ad hoc* function- or rewrite rule-based systems have been proposed to incorporate backtracking, but Bop accomplishes this capability as a natural and immediate consequence of its integration with Prolog.

Bop implements *lazy evaluation.* With the rules above, the goal

```
?-  append([1,2,3],[4,5,6])  =>>  X.
```

yields the result

```
X =   [1|append([2,3],[4,5,6])].
```

That is, in one `=>>` step, only the head of the resulting appended list is computed. The tail `append([2,3],[4,5,6])` is called a *suspension* – it is a suspended computation that can be evaluated later if this is necessary. Demand-driven computation like this is referred to as lazy evaluation or delayed evaluation, and is quite useful in applications such as stream processing [1].

The cautious reader will have noticed that, in order for the `=>>` predicate above to work as we are claiming, we must add two clauses:

```
[]    =>> [].
[X|Y] =>> [X|Y].
```

These are the clauses produced by the Bop completion rules for `[]` and `[X|Y]`. Here both `[]` and `[X|Y]` serve as *constructors*; they are used to construct 'values' or 'data structures' (in this case streams). A program that defines completions only in order to build data structures is said to follow the *constructor discipline* [22]. Bop completion rules are almost always used to define constructors, but can be combined with ordering rules (violating the constructor discipline) if this is desired.

Bop rules blend with Prolog through the conditions in the rules. Thus in the Bop rule

```
[X|Xs] + [Y|Ys]  =>  [Z|Xs+Ys]  :-  Z is X+Y.
```

the condition `Z is X+Y` is treated by the Bop system as being a guard for the rule, and the resulting code produced is equivalent to

```
[X|Xs] + [Y|Ys]  =>>  A  :-  Z is X+Y,  [Z|Xs+Ys] =>> A.
```

By combining rewriting and Prolog computations judiciously, users can obtain programming power not available from Prolog or rewriting alone.

1.4 Format of Bop Source Files

Bop source files are designated by filenames of the form

basefilename`.bop`

where *basefilename* is arbitrary. These files are text files containing any of the following:

1. Bop rules
2. Directives
3. Prolog clauses

As mentioned above, Bop rules can have any of the following forms:

LHS `=>` *RHS.*

LHS `=>` *RHS* `:-` *PrologGoal.*

A *directive* has the format

`:-` *PrologGoal.*

where *PrologGoal* is a Prolog command. Directives are typically used to indicate

1. goals to be executed, or
2. additional files to be consulted.

These actions are performed immediately, at `consult` time. Directives that provide metadata are implemented as system predicates that have side effects. For example, the directive

```
:- bop_consult([myrules, library(math)]).
```

tells Bop to perform a *Bop consult* of the two files `myrules.bop` and `math.bop` (the latter of which should be found by searching through all the directories `L` currently defined by the Prolog goal `library_directory(L)` as in SICStus and Quintus Prolog.)

Note: The directive

`:- bop_ensure_consulted(`*File*`).`

should often be used in the place of the usual consult primitive to indicate that a consult of *File* is wanted, provided that it has not already been consulted. Note also that Bop's `consult` is very different than Prolog's `consult`. If a Prolog `consult` is needed, use

`:- consult(`*Files*`).`

Finally, any Prolog clause can appear in a Bop file.

For example, below is a valid Bop file for finding areas of convex polygons. The program works by first triangulating the polygon as shown in Figure 1. For example the polygon given by the stream of points

$$[(0,0),(1,4),(5,6),(6,3),(4,-1)]$$

corresponds to the set of triangles

$$\{\ ((0,0),(1,4),(5,6)),\ \ ((0,0),(5,6),(6,3)),\ \ ((0,0),(6,3),(4,-1))\ \}$$

having respective areas[2]

$$\{\ 7.0,\ 10.5,\ 9.0\ \}$$

and a total area of 26.5.

[2]The Heron formula for the area of a triangle whose sides have respective lengths a, b, c is given by

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$.

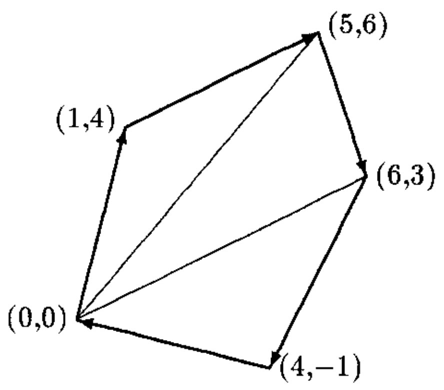

Figure 1.1: Triangulation of a Convex Polygon

```
:- [library(numbers)].   % numbers are completions
:- [library(math_pl)].   % include Prolog sqrt() function
:- [library(agg_stat)].  % include sum() aggregate function


total_area(Polygon) => sum(areas(triangles(Polygon))).

sample_polygon  =>  [ (0,0), (1,4), (5,6), (6,3), (4,-1) ].

triangles([])      =>  [].
triangles([P0|S])  =>  triangles1(S,P0).

triangles1([],_)       =>  [].
triangles1([P1|S],P0)  =>  triangles2(S,P0,P1).

triangles2([],_,_)        =>  [].
triangles2([P2|S],P0,P1)  =>  [tri(P0,P1,P2)|triangles2(S,P0,P2)].

areas([])                 =>  [].
areas([tri(P0,P1,P2)|S])  =>  [A|areas(S)]  :-  heron_area(P0,P1,P2,A).

heron_area(P0,P1,P2,Area) :-
        distance(P0,P1,A),
        distance(P1,P2,B),
        distance(P2,P0,C),
        S is (A+B+C)/2,
        AreaSq is S*(S-A)*(S-B)*(S-C),
        sqrt(AreaSq,Area).

distance((X1,Y1),(X2,Y2),Distance) :-
        DistanceSq is (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2),
        sqrt(DistanceSq,Distance).
```

The `triangles` function translates a stream of points into a stream of triples of points as desired.

1.5 Basic Use of the Bop System

The Bop directory has a number of subdirectories of interest:

/demos/	Demos of the Bop system
/lib/	Library files with useful rules
/doc/	Directory containing this documentation
/sda/	Material from Parker's 'Stream Data Analysis' paper [24]

In particular, however, in the Bop directory there is a Prolog saved program named bop containing the entire executable Bop system.

Assuming the rules for the polygon problem shown above are in the file polygon.bop, we could execute Bop as follows:

```
unix %  bop
...
Bop version 0.2,  Copyright (C) 1990, 1991, 1992  D. Stott Parker

Bop comes with ABSOLUTELY NO WARRANTY; for details type 'bop_no_warranty.'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'bop_copying.' for details.
For an introductory demo, type 'bop_demo.'

| ?- bop_consult(polygon).
{Bop consult: file polygon.bop...}
{Bop consult: file /bop/lib/numbers.bop...}
{/bop/lib/numbers.bop: consult complete, 20 msec 980 bytes}
{Bop consult: file /bop/lib/math_pl.pl...}
{/bop/lib/math_pl.pl: consult complete, 600 msec 19260 bytes}
{Bop consult: file /bop/lib/agg_stat.bop...}
{/bop/lib/agg_stat.bop: consult complete, 630 msec 31310 bytes}
{polygon.bop: consult complete, 1549 msec 63526 bytes}

yes
| ?- sample_polygon =>> P.

P = [(0,0),(1,4),(5,6),(6,3),(4,-1)] ?

yes
| ?- total_area(sample_polygon) =>> A.

A = 26.50000000000002 ?

yes
| ?- stream( triangles(sample_polygon) ) =>> S.

S = [tri((0,0),(1,4),(5,6)),tri((0,0),(5,6),(6,3)),tri((0,0),(6,3),(4,-1))] ?

yes
| ?- bop_fulltrace( triangles(sample_polygon) ).
-> triangles(sample_polygon)
-> triangles([(0,0),(1,4),(5,6),(6,3),(4,-1)])
-> triangles([(0,0),(1,4),(5,6),(6,3),(4,-1)])
-> triangles1([(1,4),(5,6),(6,3),(4,-1)],(0,0))
-> triangles1([(1,4),(5,6),(6,3),(4,-1)],(0,0))
-> triangles2([(5,6),(6,3),(4,-1)],(0,0),(1,4))
-> triangles2([(5,6),(6,3),(4,-1)],(0,0),(1,4))
-> [tri((0,0),(1,4),(5,6))|triangles2([(6,3),(4,-1)],(0,0),(5,6))]

yes
| ?- ^D
...
unix %
```

The various capabilities shown in this interaction will be discussed in detail later.

Chapter 2

Examples of Bop Programming

To illustrate the potential of Bop, we begin with some example programs. These programs are all available online, and can be used with the Bop demo system.

2.1 The Bop Demo System

To encourage exploration, Bop comes with a built-in 'demo' system. This system allows one to run one or more simple demonstration programs from a library that has grown over time. Once inside Bop, any of the currently available demos can be loaded and run by issuing the Prolog goal `?- bop_demo.` as follows:

```
unix % bop
...

Bop version 0.2,  Copyright (C) 1990, 1991  D. Stott Parker

Bop comes with ABSOLUTELY NO WARRANTY; for details type 'bop_no_warranty.'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'bop_copying.' for details.
For an introductory demo, type 'bop_demo.'

| ?- bop_demo.

Available Bop demos:

    csp          -    runnable version of Hoare's CSP specification language
    e            -    Compute e to any precision, in any radix
    fib          -    Fibonacci numbers
    fringe       -    Lazy comparison of trees
    genes        -    Simple DNA parsing
    grammars     -    Narrowing Grammar demos
    groups       -    Group expression simplification (Knuth-Bendix)
    hamming      -    Stream of Hamming numbers
    hull         -    Convex hull optimization
    inverses     -    simple two-variable constraints, via function inversion
    lottery      -    Stream database searches on lottery data
    lucid        -    Implementation of the Lucid stream processing language
    matrices     -    Lazy sparse linear algebra, with matrices as quadtrees
    oof          -    Object-Oriented Functional programming
    peano        -    Peano arithmetic
    primes       -    Stream of primes via sieves
    queens         -    N-queens solution for any N
    roman          -    Accept or generate valid roman numerals
    series         -    Power series manipulation
    slowsort       -    Slow-sorting via test-and-generate
    takeuchi       -    Takeuchi's doubly-recursive function

Please enter selection: e.
{Bop consult: file /r/miscl/prolog/bop/demos/e.bop...}
{/r/miscl/prolog/bop/demos/e.bop: consult complete, 410 msec 17860 bytes}

%----------------------------------------------------------------------------
% e.              Compute e to arbitrary precision, in any radix
% break.          Break to Prolog.
% exit.           Exit from running demo.
% halt.           Halt demo (and Prolog).
%----------------------------------------------------------------------------
Enter selection: e.


Computes   e = 2/1! + 1/2! + 1/3! + ...  by representing its fractional
part as a stream of coefficients of 1/i!:

                         0     1     1     1
        (e-2)     =     --- + --- + --- + --- + ...
                         1!    2!    3!    4!

We repeatedly multiply this fractional part by the radix (10, say),
and simplify the result so all coefficients ci are between 0 and i.
The resulting coefficient of 1! is then the next digit of e.

For radix 10 and 40 digits (fraction precision),
        e = 2.7182818284590452353602874713526624977572

Enter desired radix: 10.
Enter desired number of digits: 60.
2.718281828459045235360287471352662497757247093699959574966967
total CPU time 7470 msec
run this demo again? (y/n) n
run some other demo? (y/n) n
load some other demo? (y/n) n

yes
| ?- ^D
unix %
```

The examples below show a number of programs accessible through this demo system. Although this manual should illustrate the scope of Bop applications, there is no substitute for hands-on experience, so feel encouraged to run Bop and experiment.

2.2 Stream Processing

Today many applications deal with large quantities of data that take the form of a stream – an ordered sequence of data items. Streams arise in many ways: text files, genetic sequences, sound tracks, geological strata from drill holes, videotape, event histories, and on-line data.

The established stream processing paradigm consists in using transducers (functional transformations) to operate on streams. Under this approach databases are treated as streams, and queries or data analyzers are compositions of transducers that transform input data streams to output streams as needed.

For example, we can define a transducer named **diffs** that accepts a stream of numbers as input, and yields a stream of pairwise differences of these numbers as output. The program could be written as follows:

```
diffs([])     => [].
diffs([X|Xs]) => pairwise_diffs(Xs,X).

pairwise_diffs([],_)        => [].
pairwise_diffs([X|Xs],PrevX) => [DeltaX|pairwise_diffs(Xs,X)]
                                :-  DeltaX is (X-PrevX).
```

With this program, we could make the following evaluations:

```
diffs([1,4,9,16,25])          evaluates to  [3,5,7,9]
diffs(diffs([1,4,9,16,25]))   evaluates to  [2,2,2].
```

As mentioned earlier, the evaluation operator in Bop is **=>>**. This is a binary infix Prolog operator. Also, evaluation is *lazy* by default. If we load the program for **diffs** into Bop, then we could get the following Prolog interaction:

```
| ?- diffs([1,4,9,16,25]) =>> X.

X = [3|pairwise_diffs([9,16,25],4)] ?

yes
| ?- pairwise_diffs([9,16,25],4) =>> Y.

Y = [5|pairwise_diffs([16,25],9)] ?

yes
| ?-
```

Recursive evaluation of `pairwise_diffs([9,16,25],4)` will ultimately yield `[5,7,9]`.

Although the streams in this example are short, they could be infinitely long. For example, if we used `diffs` to compute the second-by-second changes in the NYSE price of IBM stock, we would obtain a stream with no imminent end. Many examples below manipulate infinite streams, and it is useful to have a primitive for printing these streams, or for printing part of these streams. Two useful primitives will be the following:

```
stream([])      =>  [].
stream([X|P])   =>  [X|S]  :-  (stream(P) =>> S).

print_stream([])     =>  [].
print_stream([X|P])  =>  [X|S]  :-  writeq(X), write('.'), nl,
                                    (print_stream(P) =>> S).
```

Here **stream** is an *eager* primitive that repeatedly evaluates its argument, as long as it successfully evaluates to a stream. Note that `print_stream` is almost identical to **stream**, but has the side-effect of printing. Finally, we can get only the first n items in a stream by defining a new primitive:

```
first(N,_)       =>  []                 :-  N=<0, !.
first(_,[])      =>  []                 :-  !.
first(N,[X|S])   =>  [X|first(N1,S)]    :-  N1 is N-1.
```

This primitive copies items X off its input to its output as long as its counter N remains positive. These primitives can be used together, so for example evaluating

```
print_stream( first(2,diffs([1,4,9,16,25])) )
```

will print 3 and 5.

With Bop, we can develop both conventional stream processing applications like this, as well as sophisticated unconventional applications. A number of demos are included that show how.

2.2.1 Primes

A Bop program is a collection of Bop rules. The Bop program below computes primes by Eratosthenes' method of sieves, using the dataflow expression of the problem apparently first proposed by McIlroy [17] and subsequently popularized by Kahn and MacQueen [14].

```
primes           =>  sieve(intfrom(2)).

intfrom(N)       =>  [N|intfrom(N1)]  :-  N1 is N+1.

sieve([P|S])     =>  [P|sieve(filter(P,S))].

filter(P,[X|S])  =>  if divides(P,X) then filter(P,S) else [X|filter(P,S)].

divides(P,X)     =>  true             :-  (X mod P) =:= 0.
divides(P,X)     =>  false            :-  (X mod P) =\= 0.
```

This program defines the stream of primes to be the result of sieving out multiples of primes from [2,3,4,5,...], i.e., the integers starting from 2. The initial value P in this stream is taken to be a prime by **sieve**, is placed in the output stream, and only non-multiples of P are passed from the input to the output. With this program, the **primes** demo produces this output, by printing the value of first(N,primes):

```
Enter desired number of primes: 15.
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 ].
total CPU time 140 msec
```

We can capture the execution of this program by writing a small program to print out the stream of primes produced, along with the suspensions of the stream.

```
| ?- consult(user).
| showStream(X) :-  X =>> [H|T], write(H-T), nl, showStream(T).
| {user consulted, 10 msec 486 bytes}

yes
| ?- showStream(primes).
2-sieve(filter(2,intfrom(3)))
3-sieve(filter(3,filter(2,intfrom(4))))
5-sieve(filter(5,filter(3,filter(2,intfrom(6)))))
7-sieve(filter(7,filter(5,filter(3,filter(2,intfrom(8))))))
11-sieve(filter(11,filter(7,filter(5,filter(3,filter(2,intfrom(12)))))))
13-sieve(filter(13,filter(11,filter(7,filter(5,filter(3,filter(2,intfrom(14))))))))
...
```

Some people are disappointed by the definition of `filter` above, preferring the following definition in Prolog:

```
filter(P,[X|S]) => filter(P,S)        :- 0 is X mod P, !.
filter(P,[X|S]) => [X|filter(P,S)].
```

This equivalent definition is perfectly acceptable with Bop, and operates more efficiently.

2.2.2 Hamming's Problem

Hamming numbers are integers expressible as $2^i 3^j 5^k$, for $i, j, k \geq 0$. They can be defined recursively as a stream with the following program:

```
hamming =>  [ 1 | mult235(hamming) ].

mult235(S) =>  sorted_merge( stream_mult(S,2),
                             sorted_merge( stream_mult(S,3),
                                           stream_mult(S,5) ) ).

sorted_merge([X|S],[Y|T]) => [X|sorted_merge(S,T)]      :- X=Y.
sorted_merge([X|S],[Y|T]) => [X|sorted_merge(S,[Y|T])]  :- X<Y.
sorted_merge([X|S],[Y|T]) => [Y|sorted_merge([X|S],T)]  :- X>Y.

stream_mult([X|S],N)      => [XN|stream_mult(S,N)]      :- XN is X*N.
```

With this definition, we can run the `hamming` demo to get the following value for `first(N,hamming)` with `N=20`:

```
Enter desired number of values: 20.
[ 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36 ].
total CPU time 350 msec
```

2.2.3 Fibonacci Numbers

Fibonacci numbers can be defined in *Lucid*-like style with the following program:

```
fib  =>  [1,1|fib+next(fib)].

next([_|S])  =>  S.

[] + []          =>  [].
[X|Xs] + [Y|Ys]  =>  [XY|Xs+Ys]  :-  XY is X+Y.
```

The `fib` demo allows us to print the first n Fibonacci numbers with the following interaction.

```
Enter how many Fibonacci numbers you wish: 15.
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 ].
total CPU time 1709 msec
```

2.2.4 Formal Power Series

The program described here was inspired by the extremely enjoyable paper [18]. Formal power series can be implemented as streams. Power series here are implemented as streams of numbers; the numbers are the coefficients of the power series. For example, the stream $[1, 0, 3, 0, 0, -1]$ denotes $(1 + 3x^2 - x^5)$. Also, [] represents 0.

We can define arithmetic operators `+`, `-`, `*`, `^` for power series with rules like the following:

```
[] + S          =>  S.
[A|As] + []     =>  [A|As].
[A|As] + [B|Bs] =>  [AB|As+Bs]    :- AB is A+B.

[] - S          =>  [0] - S.
[A|As] - []     =>  [A|As].
[A|As] - [B|Bs] =>  [AB|As-Bs]    :- AB is A-B.

[] * _          =>  [].
_ * []          =>  [].
[A|As] * [B|Bs] =>  [AB|([A]*Bs + As*[B] + [0|As*Bs])]  :- AB is A*B.

_ ^ N           =>  [1]           :- N = 0.
S ^ N           =>  S             :- N = 1.
S ^ N           =>  S^N1 * S^N2   :- integer(N), N > 2, N1 is N//2, N2 is N-N1.
```

This is enough to implement basic power series arithmetic! Expressions using these four operators can be evaluated incrementally using these definitions. Running the `series` demo shows this:

```
Please enter a power series expression.

Expression: [1,0,-3]^2 + [1,1]*[-1,6,1].
Enter desired length of series: 10.
Power Series: [ 0, 5, 1, 1, 9 ].
total CPU time 59 msec
```

The demo evaluates `first(10, [1,0,-3]^2+[1,1]*[-1,6,1] )` to get the result shown.

These basic definitions can be extended in many ways. With only a little refinement, to deal with special situations such as when `A` is 0 or 1, this implementation becomes fairly efficient. Also, we can add surprisingly concise definitions for various power series operators as in [18].

Reciprocals

The reciprocal $R = 1/P$ of the power series $P = 1 + x$ is $R = 1 - x + x^2 - x^3 \cdots = 1 - xR$. Similarly, the reciprocal $R = 1/P$ of $P = 1 + xQ$ will therefore be $R = 1 - xQR$. Thus, we can get a surprisingly compact definition for reciprocals:

```
reciprocal([1|As]) => [1|-As * reciprocal([1|As])]            :- !.
reciprocal([A|As]) => [Ainv|-As * [Ainv] * reciprocal([A|As])] :- Ainv is 1/A.
```

With this definition, we can compute reciprocals lazily. For example, to find the power series for $1/(1-2x)$, we can evaluate the expression `reciprocal([1,-2])`:

```
Expression: reciprocal([1,-2]).
Enter desired length of series: 15.
Power Series:
[ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 ].
total CPU time 259 msec
```

Derivatives and Integrals

Integrals and derivatives are simply ± 1 'shifts' of power series and multiplication by corresponding factors of i or $1/i$.

```
          derivative([])      =>  [].
          derivative([_|S])   =>  pairwise_products(S,ints(1)).

          integral(S)    =>  [0|pairwise_ratios(S,ints(1))].
          integral(C,S)  =>  [C|pairwise_ratios(S,ints(1))].

          ints(N)  =>  [N|ints(N1)]        :- N1 is N+1.
```

With these definitions, we can find the integral of $1/(1+x^2)$ (the arctangent series) by evaluating the expression `integral(reciprocal([1,0,1]))`:

```
   Expression:  integral(reciprocal([1,0,1])).
   Enter desired length of series: 15.
   Power Series:
   [ 0, 1.0, 0.0, -0.3333333333333332, 0.0,
     0.2, 0.0, -0.1428571428571428, 0.0, 0.1111111111111111,
     0.0, -0.09090909090909092, 0.0, 0.07692307692307689, 0.0 ].
   total CPU time 150 msec
```

Reversions

The *reversion* of a power series satisfies `reversion(P) = Q` if and only if `P(Q)` = *id*; i.e., reversion gives the formal inverse of a power series. For example, `reversion(tan)` = `arctan`.

Using an analysis similar to that for reciprocals above, the reversion of a power series whose constant coefficients are zero can be computed with the following definition:

```
  reversion([0|S]) => [0|R]  :-
                      S =>> [A|As],
                      bopDiv(1.0,A,Ainv),
                      R = [Ainv|((-R^2 * [Ainv] * compose(As,[0|R])))].
```

For example, the reversion of $(x+2x^3)$ is found by evaluating the expression `reversion([0,1,0,2])`:

```
   Expression: reversion([0,1,0,2]).
   Enter desired length of series: 10.
   Power Series:
   [ 0, 1.0, -0.0, -2.0, -0.0, 12.0, -0.0, -96.0, -0.0, 880.0 ].
   total CPU time 8720 msec
```

Note that the term `R` constructed by this program is circular! Nevertheless, the program still works. If use of circular terms causes concern, a definition like that for `reciprocal` can be used instead.

Transcendental Functions

Self-reference also makes its presence felt in transcendental functions. Power series are easily defined for standard transcendental functions, such as *exp*, *log1p*, *sin*, *cos*, *tan*, *sinh*, *cosh*, *tanh*, *arcsin*, *arccos*, *arctan* using identities from analysis. For example, the power series for the exponential function can be defined using integration and self-reference as follows:

```
  %------------------------------------------------------------------------
  %       Exponentials of power series
  %
  % Uses McIlroy's trick:
  %       exp(P(x))  =  int (d/dx exp(P(x))) dx  =  int (exp(P(x))*P'(x)) dx
```

```
%-----------------------------------------------------------------------

exp([])      =>  [1].
exp([0|As])  =>  integral(1, (exp([0|As])*derivative([0|As])) )  :-  !.
exp([A|As])  =>  [ExpA] * exp([0|As])                            :-  bopExp(A,ExpA).
```

The `series` demo allows inspection of the definitions for these functions:

```
Please select a function and its definition and power series will be printed.

Function: exp.
Definition: exp => exp(id)
Enter desired length of series: 10.
Power Series:
[ 1, 1.0, 0.5, 0.1666666666666666, 0.04166666666666666,
  0.008333333333333331, 0.001388888888888889, 0.0001984126984126983,
  2.480158730158729e-5, 2.755731922398589e-6 ].
total CPU time 210 msec
```

2.2.5 Stream Databases

Since the late 1970's, conventional wisdom regarding databases has been that typically databases should be modeled as *sets* of data items. However, the generality of this approach is limited.

It is not hard to see that often databases can be modeled more exactly as *streams* of tuples [24]. Bulk stream data arise in many important forms, including text files, sound tracks, event histories, genetic sequences, and so on. Data with any significant temporal component can usually be viewed as a stream. There is a serious need for a new approach in stream data processing. Today's database systems are not able to manage this data.

Database Processing as Stream Processing

An approach that is possible with Bop is to develop database systems using the established paradigm of stream processing, in which transducers (functional transformations) operate on streams. Under this approach databases are treated as streams, and queries or data analyzers are compositions of transducers that transform input data streams to output streams as needed.

With Bop, we can define many of the standard database query operators, such as `select`:

```
select([],_,_)              =>  [].
select([X|S],Pattern,Test) =>  select(S,Pattern,Test) :- \+ (X=Pattern,Test), !.
select([X|S],Pattern,Test) =>  [X|select(S,Pattern,Test)].
```

Very much like the `sieve` transducer, `select` removes elements `X` that fail to satisfy a given `Test` (Prolog goal), where variables in the test are bound by matching the input against a provided `Pattern`.

Let us use this transducer in a time series example. Assume the file `demos/njlottery.pl` contains time series data on the New Jersey lottery in the following format:

```
%       Year    Number  Payoff
%--------------------------------------
lot(    1975,   810,    190     ).
lot(    1975,   156,    120.5   ).
lot(    1975,   140,    285.5   ).
        ...     ...     ...
```

This data gives ordered New Jersey lottery results for three years (1975, 1976, 1980), along with the payoff in dollars. In this game one picks three digits between 0 and 9, and wins the payoff if the digits match those chosen by the state. The payoffs are determined by the number of winners, and look pretty modest for the risk involved. We can see how large the payoffs actually got by using the `select` transducer. With the query

```
select(file_terms('demos/njlottery.pl'),
        lot(_Year,_Winner,Payoff),
        Payoff > 600 )
```

the `lottery` demo yields the following:

```
lot(1975,499,869.5).
lot(1975,20,668.5).
lot(1975,77,640).
lot(1975,767,756).
lot(1975,919,637).
lot(1976,6,710).
lot(1976,966,720.5).
lot(1980,778,621).
```

Thus in the three years of daily lottery games in New Jersey for which we have data, only eight games paid more than $600. New Jersey is disappointing in many ways.

The SVP Database Model

It is actually possible to combine set and stream processing in a single model; in particular, the SVP model [25] shows this successfully. With SVP, *collections* of data (sets or streams) are modeled as binary trees, thus generalizing on both set indexing schemes and list or stream structure at the same time.

For example, the polygon example presented in the introduction can be rewritten in Bop for SVP collections. Here '`{}`' represents the empty collection, '`{X}`' represents the collection with one element `X`, '`<>`' is the cons-like binary tree constructor (an infix operator) used for collections, so `({1}<>{2})<>({3}<>{4})` is a balanced collection, and `{1}<>({2}<>({3}<>({4}<>{[]})))` is a (nil-terminated, list-like) stream. The following program now implements the polygon example:

```
total_area(Polygon)  =>  sum(areas(triangles(Polygon))).


triangles(S) =>  triangles1(S,tri(_,_,_)).

triangles1({},_)      =>  {}.
triangles1({P},Q)     =>  output_triangle(NQ)   :- next_triangle({P},Q,NQ).
triangles1(S1<>S2,Q)  =>  triangles1(S1,Q) <> triangles1(S2,NQ)
                                                :- next_triangle(S1,Q,NQ).

output_triangle(Q)  =>  {Q} :- complete_triangle(Q), !.
output_triangle(_)  =>  {}.

complete_triangle(tri(P0,P1,P2)) :- nonvar(P0), nonvar(P1), nonvar(P2).

next_triangle({P},tri(P0,_ ,P2),tri(P0,P2,P)) :- nonvar(P2), !.
next_triangle({P},tri(P0,P1,_ ),tri(P0,P1,P)) :- nonvar(P1), !.
next_triangle({P},tri(P0,_ ,_ ),tri(P0,P,_))  :- nonvar(P0), !.
next_triangle({P},tri(_ ,_ ,_ ),tri(P,_,_)).

areas({})                 =>  {0}.
areas({tri(P0,P1,P2)})    =>  {A}  :-  heron_area(P0,P1,P2,A).
areas(S1<>S2)             =>  areas(S1) <> areas(S2).
```

This SVP program and many others can be run with the **svp** demo.

2.3 Pattern Matching and Grammars

It is certainly not news that rewrite rules can be used to represent grammars. However, extremely powerful attribute-grammar-like programs can be written in Bop, programs that are more expressive than Definite Clause Grammars or other logic grammars.

2.3.1 Narrowing Grammar

Bop generalizes a rewriting system called *Narrowing Grammar* developed earlier by Chau and Parker [5]. The concepts of Narrowing Grammar are reviewed later in this manual, but they can be summarized here with a few examples.

Bop can be used as a grammatical formalism. The rules

```
(X+)   =>  X.
(X+)   =>  X , (X+).

([] , L)        =>  L.
([X|L1] , L2)   =>  [X|(L1 , L2)].
```

define regular expression closure ('Kleene +') and concatenation. With these rules, we can generate regular expressions: the expression `[a],([b,b]+)` will evaluate to `[a,b,b]`, `[a,b,b,b,b]`, etc. The arguments of the operators '+' and ',' can be any expression, so we can build arbitrary regular expressions with these.

This is only the beginning: we can define any parameterized algebra of patterns we like. For example, we can define the `number` pattern to produce a given number `N` of instances of a given pattern `P`.

```
number(P,N)      =>  number(P,N,0).

number(P,N,N)    =>  [].
number(P,N,M)    =>  P,number(P,N,M1)   :-   succ(M1,M).
```

For example, the expression `number(([bump]+,[turn]),2)` has the following execution with this grammar:

```
number(([bump]+,[turn]),2)
number(([bump]+,[turn]),2,0)

([bump]+,[turn]), number(([bump]+,[turn]),2,1)
([bump],[turn]), number(([bump]+,[turn]),2,1)
[bump|([],[turn])], number(([bump]+,[turn]),2,1)
[bump| (([],[turn]), number(([bump]+,[turn]),2,1))]
[bump| ([turn], number(([bump]+,[turn]),2,1))]
[bump| [turn| ([], number(([bump]+,[turn]),2,1))]]
[bump| [turn| number(([bump]+,[turn]),2,1)]]

[bump| [turn| (([bump]+,[turn]), number(([bump]+,[turn]),2,2))]]
[bump| [turn| (([bump],[turn]), number(([bump]+,[turn]),2,2))]]
[bump| [turn| [bump| (([],[turn]),number(([bump]+,[turn]),2,2))]]]]
[bump| [turn| [bump| [turn| number(([bump]+,[turn]),2,2)]]]]

[bump| [turn| [bump| [turn| []]]]]
```

The examples above show generation of languages from patterns. In fact, the same programs can be used to match very complex patterns. For example, we can define the grammar

```
...          =>  [].
...          =>  [_| ... ].

[] , S       =>  S.
[X|P] , S    =>  [X|(P,S)].

before(P,Q)  =>  ..., P, ..., Q, ... .
```

and then use it to match temporal patterns in an input stream, in which one pattern is constrained to appear before another. With little effort we can extend the vocabulary here to describe other patterns of interest.

One concern here is how we can define two patterns that *overlap*. In Bop there one can do this with *pattern coroutining*. This allows us to match two patterns simultaneously. For example, with the grammar

```
[] // []          =>  [].
[X|P] // [X|S]    =>  [X|(P//S)].

partial_order_1 =>
   before([e1],[e2]) // before([e1],[e3])
   // before([e3],[e5]) // before([e3],[e4])
   // before([e2],[e6]) // before([e5],[e6])
   // before([e4],[e6]).
```

the pattern **partial_order_1** will match any of the following event streams:

```
[e1, e2, e3, e4, e5, e6]
[e1, e2, e3, e5, e4, e6]
[e1, e3, e2, e4, e5, e6]
[e1, e3, e2, e5, e4, e6]
[e1, e3, e4, e2, e5, e6]
[e1, e3, e5, e2, e4, e6]
[e1, e3, e4, e5, e2, e6]
[e1, e3, e5, e4, e2, e6].
```

2.3.2 Hoare's CSP

Hoare describes a trace-theoretic approach to specifying concurrent software in his book on *Communicating Sequential Processes* [11]. The CSP vocabulary chosen by Hoare can be translated to Bop almost symbol-for-symbol. For example, below are some sample definitions.

```
% Concatenation of traces:  S followed by T
[]^T => T.
[X|S]^T => [X|(S^T)].

% Repetition of traces:  S repeated N times
S**N => []  :- N =< 0, !.
S**N => S^(S**N1)  :-  N1 is N-1.

% Restriction of traces:  S restricted to (a set) A
[] restricted_to _ => [].
[X|S] restricted_to A => (S restricted_to A)     :- \+ member(X,A), !.
[X|S] restricted_to A => [X|(S restricted_to A)].

% Prefixes of traces:  S is a prefix of T
([] =< [])  => true.
([] =< [_|_])  => true.
([_|_] =< [])  => false.
([X|S] =< [Y|T]) => false     :-  X\==Y.
([X|S] =< [Y|T]) => (S =< T)  :-  X==Y.

% Similarity of traces:  S is like T with up to N symbols removed
is_like([],[],N) => true.
is_like([],[X|S],N) => true  :- #[X|S] =>> L, L =< N.
is_like([],[X|S],N) => false :- #[X|S] =>> L, L >  N.
is_like([X|S],[Y|T],N) => is_like(S,T,N) :- X == Y.
is_like([X|S],[Y|T],N) => is_like([X|S],T,N1) :- X \== Y, N1 is N-1.

% Subsequences of traces:  S is a subtrace of T
[] in [] => true.
[] in [_|_] => true.
[_|_] in [] => false.
[X|S] in [Y|T] => S in T  :- X == Y.
[X|S] in [Y|T] => [X|S] in T  :- X \== Y.

% Trace summary: the count of a symbol B in S
symbol_count([],_) => 0.
symbol_count([X|S],B) => N1 :- X==B,  symbol_count(S,B) => N, N1 is N+1.
symbol_count([X|S],B) => N  :- X\==B, symbol_count(S,B) => N.
```

Hoare also defines modest machines for processing traces. For example, here are some simple machines from pp.30–31:

```
 % A machine which offers a choice of two combinations of change for 5p

     ch5c => in5p -> ( out1p -> out1p -> out1p -> out2p -> ch5c
                     | out2p -> out1p -> out2p -> ch5c ).

 % A more complicated vending machine, which offers a choice of coins
 %      and a choice of food and change

     vmc  => (in2p -> (large -> vmc
                         |small -> out1p -> vmc)
              |in1p -> (small -> vmc
                         |in1p -> (large -> vmc
                                   |in1p -> stop))).

% A copying process

    copybit => mu X.(in_0 -> out_0 -> X
                    |in_1 -> out_1 -> X).
```

The csp demo can be used to get all possible short traces of a particular machine's execution:

```
Enter one of ch5c, vmct, vmc, vmcred, vms2, copybit:      vmc.
Enter length of traces to be generated:                   3.
Resulting tree of traces:
 in2p
      large
            in2p
                 large
                 small
            in1p
                 small
                 in1p
      small
            out1p
                  in2p
                  in1p
 in1p
      small
            in2p
                 large
                 small
            in1p
                 small
                 in1p
      in1p
           large
                 in2p
                 in1p
           in1p
total CPU time 49 msec
```

We are not aware of another implementation of CSP of this kind.

2.4 Search

Search is a fruitful application area for laziness: many search spaces permit the search frontier to be expanded *incrementally*, using arbitrary search criteria (heuristic best-first expansion, branch-and-bound, etc.). This is particularly easy to see when the search space consists of streams.

2.4.1 Permutations and Slowsort

Using the rules

```
perm([]) => [].
perm([X|L]) => insert(X,perm(L)).

insert(X,L) => [X|L].
insert(X,[Y|L]) => [Y|insert(X,L)].
```

we can generate all permutations of a list of length N, by exhaustively backtracking through them. For example:

```
Enter size of list to be permuted: 3.
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
total CPU time 50 msec
```

Using this nondeterministic function, slowsort can be defined via generate-and-test:

```
slowsort(L) => sorted( perm(L) ).

sorted([])    => [].
sorted([X|L]) => preceded_by(L,X).

preceded_by([],X)    => [X].
preceded_by([Y|L],X) => [X|preceded_by(L,Y)]  :-  X =< Y.
```

Note that this algorithm coroutines testing (**sorted**) with generation (**perm**), in a pipeline. This coroutining prunes away many unsuccessful generations at an early point, and achieves faster execution on the average than a brute force search:

```
List: [6,1,5,4,2,3].
Resulting list is: [1,2,3,4,5,6]
total CPU time 110 msec
```

Nevertheless, the execution is still very slow for large lists, so the program still richly merits the name 'slowsort'.

2.4.2 N Queens

The n-queens problem asks to find configurations of a $n \times n$-chessboard in which n queens are placed in such a way that no queen 'attacks' any of the others. The problem was studied by Gauss, among others. The following is a Bop program for finding solutions of the problem, again using a generate-and-test approach:

```
queens(L) => safe(perm(L)).

safe([])    => [].
safe([I|L]) => [I|safe(nodiagonal(L,I,1))].

nodiagonal([],_,_)    => [].
nodiagonal([J|L],I,D) => [J|nodiagonal(L,I,D1)] :- \+ diagonal(I,J,D), D1 is D+1.

diagonal(I,J,D) :- D is I-J.
diagonal(I,J,D) :- D is J-I.
```

With the list `L=[1,2,...,N]`, where `N` is the size of the chessboard, the program will find all non-attacking configurations of the queens.

```
[2,4,6,1,3,5]
[3,6,2,5,1,4]
[4,1,5,2,6,3]
[5,3,1,6,4,2]
total CPU time 1050 msec
```

2.4.3 General Search Strategies

The simple examples above show that search can be implemented as a lazy traversal of a space of possibilities. More generally, any search strategy can be programmed if we come up with an explicit representation for the search space, and implement operators to perform the traversal.

Permutations

We can model search spaces as streams of alternatives, which are search spaces that are expanded by one step. This is the basis for Paulson's interesting treatment in [26]. For example, the space of permutations of `[1,2,3]` could be expanded breadth-first in the following way. We begin initially with the empty permutation `[]`, then repeatedly expand each lists in the current space by prepending it with all possible values that will still give us a permuted sublist of `[1,2,3]`:

```
[]
[ [1], [2], [3] ]
[ [2,1], [3,1], [1,2], [3,2], [1,3], [2,3] ]
[ [3,2,1], [2,3,1], [3,1,2], [1,3,2], [2,1,3], [1,2,3] ]
```

Search strategies can be written as higher-order functions on search spaces. These strategies take an several arguments:

1. The search space remaining to be expanded. Initially, the search space is `[[]]`, consisting of a single empty partial permutation, and at any point subsequently it will be a stream of partial permutations. These partial permutations are placed on the output when they are 'full', i.e., they contain all `N` values.

2. The mapping that obtains the descendants of the current search space node. For example, with permutations the mapping finding descendants of a partial permutation `L` of the integers between `1` and `N` is

   ```
   nextvalue(N,L) => XL :- findall([X|L], (between(X,1,N),\+member(X,L)), XL).

   between(L,L,L)  :-  !.
   between(I,L,U)  :-  M is (L+U)//2, between(I,L,M).
   between(I,L,U)  :-  M is 1+(L+U)//2, between(I,M,U).
   ```

3. The predicate that determines whether input values should be passed the output. Here the predicate that determines whether permutations are 'full' (complete, of length n) is easily defined:

   ```
   isfull(N,Qs)   =>  false  :-  \+ length(Qs,N), !.
   isfull(_,_)    =>  true.
   ```

We can now implement breadth-first and depth-first search as follows:

```
dfs([],_,_) => [].
dfs([X|Xs],Desc,Pred) => if (Pred@X)
                            then  [X|dfs(append(DescX,Xs),Desc,Pred)]
                            else  dfs(append(DescX,Xs),Desc,Pred)
                         :- Desc@X =>> DescX.

bfs([],_,_) => [].
bfs([X|Xs],Desc,Pred) => if (Pred@X)
                            then  [X|bfs(append(Xs,DescX),Desc,Pred)]
                            else  bfs(append(Xs,DescX),Desc,Pred)
                         :- Desc@X =>> DescX.
```

These definitions are almost transparent. The idea is that depth-first search places the descendants of the current search space node at the *beginning* of the remaining space to be searched, while breadth-first search places them at the *end*. At each step we find the descendants `Desc@X` on the current node `X`, and depending on the value of the predicate `Pred@X` either include or omit `X` from the output. With these definitions, both expressions

```
dfs( [[]], nextvalue(3), isfull(3) )
bfs( [[]], nextvalue(3), isfull(3) )
```

yield the stream of permutations

```
[ [3,2,1], [2,3,1], [3,1,2], [1,3,2], [2,1,3], [1,2,3] ].
```

Note that these permutations are obtained last-digit-first, so the earliest permutations have last digit 1, and so forth.

This example shows the use of function application operator '`@`'. Since we want 'curried' expressions like `nextvalue(3)` and `isfull(3)` to be values until they are applied to some list, we must include definitions like the following:

```
nextvalue(N)   => nextvalue(N).

isfull(N)  =>  isfull(N).
```

Through all this, we see that we can define search strategies as higher-order functions on search spaces. We can also go one step further and extend them to functions on individual search space nodes:

```
depth_first(Desc,Pred) => depth_first(Desc,Pred).
depth_first(Desc,Pred,X) => dfs([X],Desc,Pred).

breadth_first(Desc,Pred) => breadth_first(Desc,Pred).
breadth_first(Desc,Pred,X) => bfs([X],Desc,Pred).

dfs_perms(N)  =>  depth_first(nextvalue(N),isfull(N),[]).

bfs_perms(N)  =>  breadth_first(nextvalue(N),isfull(N),[]).
```

With these definitions, `dfs_perms(3)` and `bfs_perms(3)` both evaluate to the same stream given above.

Palindromes

A *palindrome* is a word that equals its own reverse. We can set up a search for palindromes by incrementally generating words as described above, where descendants are incremental extensions of their parents, and the predicate used determines whether the word involved is a palindrome.

```
dfs_words => names((depth_first(words_ending_with,is_palindrome) @ [])).

bfs_words => names((breadth_first(words_ending_with,is_palindrome) @ [])).

words_ending_with(L) => [ [0'A|L], [0'B|L], [0'C|L] ].
words_ending_with => words_ending_with.

is_palindrome(L) => false :- \+ (reverse(L) =>> L), !.
is_palindrome(_) => true.
is_palindrome => is_palindrome.

names([]) => [].
names([L|Ls]) => [N|names(Ls)] :- name(N,L).
```

With this program we get the following output:

```
| ?- print_stream(first(20,dfs_words)) =>> _.
''.
'A'.
'AA'.
'AAA'.
'AAAA'.
'AAAAA'.
'AAAAAA'.
'AAAAAAA'.
'AAAAAAAA'.
'AAAAAAAAA'.
'AAAAAAAAAA'.
'AAAAAAAAAAA'.
'AAAAAAAAAAAA'.
'AAAAAAAAAAAAA'.
'AAAAAAAAAAAAAA'.
'AAAAAAAAAAAAAAA'.
'AAAAAAAAAAAAAAAA'.
'AAAAAAAAAAAAAAAAA'.
'AAAAAAAAAAAAAAAAAA'.
'AAAAAAAAAAAAAAAAAAA'.

yes
| ?- print_stream(first(20,bfs_words)) =>> _.
''.
'A'.
'B'.
'C'.
'AA'.
'BB'.
'CC'.
'AAA'.
'ABA'.
'ACA'.
'BAB'.
'BBB'.
'BCB'.
'CAC'.
'CBC'.
'CCC'.
'AAAA'.
```

```
'ABBA'.
'ACCA'.
'BAAB'.

yes
```

Depth-First Iterative Deepening Search

Following Paulson, we can extend the search strategies above to include such things as depth-first iterative-deepening (DFID) search. This repeatedly applies depth-first search up to a given depth bound K, which is incremented over time. Since DFID search is extremely slow when the increment chosen is small (in particular, when this increment is 1), the program here supports an arbitrary depth increment.

```
dfid(Desc,Pred) => dfid(Desc,Pred).
dfid(Desc,Pred,X) => dfid(1,Desc,Pred,X).

dfid(DepthIncrement,Desc,Pred) => dfid(DepthIncrement,Desc,Pred).
dfid(DepthIncrement,Desc,Pred,X) => deepen(0,DepthIncrement,Desc,Pred,X).

deepen(K,D,Desc,Pred,X) => dfs(K,D,Desc,Pred,X,deepen(KD,D,Desc,Pred,X))
         :- KD is K+D.


dfs(K,D,Desc,Pred) => dfs(K,D,Desc,Pred).
dfs(K,D,Desc,Pred,X) => dfs(K,D,Desc,Pred,X).

dfs(K,_,_,Pred,X,SF)    => if(Pred@X, [X|SF], SF) :- K = 0, !.
dfs(K,D,Desc,Pred,X,SF) => foldright(dfs(K1,D,Desc,Pred),Desc@X,SF) :- K1 is K-1.

foldright(_,[],SF) => SF :- !.
foldright(Op,[X|Xs],SF) => Op@X@foldright(Op,Xs,SF).

dfid_words => names(dfid(words_ending_with,is_palindrome,[])).
```

With this definition, we obtain the following result:

```
| ?- print_stream(first(20,dfid_words)) =>> _.
''.
'A'.
'B'.
'C'.
'AA'.
'BB'.
'CC'.
'AAA'.
'ABA'.
'ACA'.
'BAB'.
'BBB'.
'BCB'.
'CAC'.
'CBC'.
'CCC'.
'AAAA'.
'ABBA'.
'ACCA'.
'BAAB'.
```

N Queens With Arbitrary Search Strategy

Earlier we implemented the n-queens problem with depth-first search. Here, we can implement it using various strategies:

```
queens(N) => depth_first(nextqueen(N),isfull(N), []).

dfid_queens(D,N) => dfid(D,nextqueen(N),isfull(N), []).


nextqueen(N,Qs) => NQs
     :-  findall([NewQ|Qs],(between(NewQ,1,N),safequeen(Qs,NewQ)),NQs).

nextqueen(N) => nextqueen(N).

safequeen(Qs,NewQ) :- \+ member(NewQ,Qs), \+ diag(1,NewQ,Qs).

diag(I,NewQ,[Q|_])  :- I is NewQ-Q.
diag(I,NewQ,[Q|_])  :- I is Q-NewQ.
diag(I,NewQ,[_|Qs]) :- I1 is I+1, diag(I1,NewQ,Qs).
```

The results here are like those before:

```
| ?- print_stream(first(10,queens(8))) =>> _.
[4,2,7,3,6,8,5,1].
[5,2,4,7,3,8,6,1].
[3,5,2,8,6,4,7,1].
[3,6,4,2,8,5,7,1].
[5,7,1,3,8,6,4,2].
[4,6,8,3,1,7,5,2].
[3,6,8,1,4,7,5,2].
[5,3,8,4,7,1,6,2].
[5,7,4,1,3,8,6,2].
[4,1,5,8,6,3,7,2].

yes
| ?- print_stream(first(10,dfid_queens(4,8))) =>> _.
[4,2,7,3,6,8,5,1].
[5,2,4,7,3,8,6,1].
[3,5,2,8,6,4,7,1].
[3,6,4,2,8,5,7,1].
[5,7,1,3,8,6,4,2].
[4,6,8,3,1,7,5,2].
[3,6,8,1,4,7,5,2].
[5,3,8,4,7,1,6,2].
[5,7,4,1,3,8,6,2].
[4,1,5,8,6,3,7,2].

yes
| ?- print_stream(queens(6)) =>> _.
[5,3,1,6,4,2].
[4,1,5,2,6,3].
[3,6,2,5,1,4].
[2,4,6,1,3,5].

yes
| ?- print_stream(dfid_queens(3,6)) =>> _.
```

```
[5,3,1,6,4,2].
[4,1,5,2,6,3].
[3,6,2,5,1,4].
[2,4,6,1,3,5].
^C
Prolog interruption (h for help)? a
{ Execution aborted }
| ?-
```

The second query uses DFID search with a depth increment of 4. Of course, the searches here are slower than for the n-queens program developed earlier since the search strategy is implemented interpretively, and the search space is materialized in list structures rather than visited via backtracking. One also learns a fact about DFID search that is not often discussed: it is not easy to recognize when DFID search should halt! The point is that general search strategies can be programmed and used easily. Actually, a further point is that ML code is easily stolen using Bop.

2.5 Metaprogramming

As the previous example shows, it is possible to specify control of complex search processes straightforwardly with Bop. More generally, Bop is a fine system for developing interpreters. In fact, Bop has been intentionally developed with metaprogramming in mind.

2.5.1 A General Prolog Interpreter

Meta-circular interpreters for *all of* Prolog are ugly. This is primarily due to the inadequacy of the **clause** predicate and difficulty of interpreting unprincipled uses of the cut primitive. One surprising aspect of Bop is that, if cuts are restricted to principled uses with the conditional rules, then it becomes possible to write simple Bop/Prolog meta-interpreters, such as the following:

```
prove(G)          =>  true      :-  builtin(G), !, call(G).
prove(G)          =>  prove(B)  :-  (G=>B).
prove(\+ G)       =>  true      :-  \+ (prove(G) =>> true).
prove((A,B))      =>  prove(B)  :-  prove(A) =>> true.
prove((A->B))     =>  prove(B)  :-  prove(A) =>> true.
prove((A->B;_))   =>  prove(B)  :-  prove(A) =>> true.
prove((_->_;C))   =>  prove(C).
prove((A;_))      =>  prove(A).
prove((_;B))      =>  prove(B).
```

Note that the second line uses the **=>** predicate where Prolog interpreters would normally use **clause/2**! Since the conditions in Bop rules can include cuts, then as long as the conditions amount to *preconditions* specifying when a Bop rule can be selected, the interpreter here handles full Prolog. For example, the program

```
fib(N,1)  =>  true                                :-  N=<1, !.
fib(N,F)  =>  fib(N1,F1), fib(N2,F2), F is F1+F2  :-  N>1, N1 is N-1, N2 is N-2.
```

uses the rule conditions only as *guards* on rules. This can be thought of as augmenting the indexing provided by Prolog in clause selection. This program uses cuts only for these preconditions, and in such a situation, *the interpreter above is meta-circular*! It interprets cuts properly.

```
|?- prove(fib(14,X)) =>> Y.

X = 610,
Y = true ? ;
```

```
no
|?- prove(prove(fib(14,X))) =>> Y.

X = 610,
Y = true ? ;

no
|?-
```

The advantages of rewriting become apparent when we consider more sophisticated programs. Although the program above takes time exponential in n to complete, we can bring the time down to *linear* in n by adding one additional rule:

```
(fib(N1,F1), fib(N2,F2), Add)   =>   ((fib(N1,F2), More), Add)
        :-  (fib(N1,F1) => fib(N2,F2), More).
```

This rule is very subtle! It recognizes that evaluation of `fib(N,F)` causes evaluation of `fib(N1,F1)` and `fib(N2,F2)`, and that evaluation of `fib(N1,F1)` will cause `fib(N2,F2)` to be evaluated again redundantly. The rule merely consolidates the two `fib(N2,F2)` subgoals. The result is a huge decrease in complexity.

2.5.2 Constraint-Processing Metainterpreters

We can extend the interpreter above to take an environment argument `E`, in which we can save arbitrary constraints. Execution of a 'constraint' causes this environment to be updated, and bindings generated in the course of normal execution should cause this environment to be updated as well. For example, we can implement a Prolog system with the `freeze` primitive. Recall `freeze(X,G)` delays execution of the Prolog goal `G` until the term `X` becomes nonvariable. This delayed execution can be implemented in Bop as follows:

```
solve(G)  =>  solve(G,[]).

solve(G,           E)  =>  E            :- builtin(G), !, call(G).
solve(G,           E)  =>  solve(B,E1) :- (G=>B), thaw(E,E1).
solve(\+ G,        E)  =>  E            :- \+ (solve(G,E) =>> _).
solve((A,B),       E)  =>  solve(B,E1) :- solve(A,E) =>> E1.
solve((A->B),      E)  =>  solve(B,E1) :- solve(A,E) =>> E1.
solve((A->B;_),    E)  =>  solve(B,E1) :- solve(A,E) =>> E1.
solve((_->_;C),    E)  =>  solve(C,E).
solve((A;_),       E)  =>  solve(A,E).
solve((_;B),       E)  =>  solve(B,E).
solve(freeze(X,G),E)  =>  E1           :- var(X), !, insert(freeze(X,G),E,E1).
solve(freeze(_,G),E)  =>  solve(G,E).


thaw([],[]).
thaw([FG|E0],[FG|E]) :- FG=freeze(X,_), var(X), !, thaw(E0,E).
thaw([FG|E0],E)       :- FG=freeze(_,G), !, solve(G,E0) =>> E1, thaw(E1,E).
thaw([G|E0],[G|E])   :- thaw(E0,E).

insert(G,E,[G|E]).
```

With this interpreter we can write many constraint processing programs. For example, the classic SEND+MORE=MONEY demo is provided. It is easily arrives at a solution quickly, using a rule like `dif_numbers(X,Y) => freeze(X, freeze(Y, X \== Y))`.

Chapter 3

Bop and Rewriting

The rewriting field is large, and at first it is difficult to get a grasp on its development. Rewrite rules have been used as formula simplifiers in automated deduction systems for over twenty five years [3, 12].

Extensions of Prolog to include term rewriting, equational logic, or functional logic programming typically either introduce a more general equality relation as an additional predicate, or extend the unifier from the usual syntactic unification procedure to solve some more general equational theory. In both cases the equality theory is usually 'directed' – i.e., the equations are used only one way, in which more complex expressions are equated to simpler expressions – giving an equational system which can be viewed as an evaluator. Bop follows the first approach; the compendium [15] gives an good survey of current work on extended unifiers.

Bop rests on *conditional term rewriting*. Its implementation, with completion rules and ordering rules, and other features such as the **schema** directive, supports various kinds of term rewriting. Specifically, Bop supports *complete term rewriting* (Knuth-Bendix-like systems and systems with term orderings) and *regular term rewriting* (Huet-like confluent systems, lambda calculus, combinatory logic, systems with constructors). We show examples of these kinds of term rewriting below.

3.1 Conditional Term Rewriting and Bop

Conditional rewrite rules arise naturally from equational logic, or the combination of logic and functions. For example, consider the rule

$$\textit{for all real } X,\ \textit{power}(X,0) = 1 \textit{ if } X \neq 0$$

which asserts that the arithmetic expression *power(X,0)* can be simplified to *0* as long as X is nonzero.

In the past decade these rules have attracted increasing amounts of attention at international conferences and workshops on rewriting systems, automated deduction, and programming languages. One reason for interest in them is that they offer an elegant way to integrate rewriting, functional programming, and logic programming. Conditional rewrite rules have been featured recently by a number of proposed programming systems, including RITE [8, 13], SLOG [9], and K-LEAF [16]. See also [2, 6] for surveys of integrations of functional and logic programming.

Bop in particular is the successor of a number of rewrite rule systems developed at UCLA, particularly F* and Log(F) [20, 21], Narrowing Grammar [4], and FLOP [23]. These UCLA systems were all initially inspired by the work of Sanjai Narain, beginning in 1984 with his paper on implementing lazy evaluation in Prolog [19]. Bop is a second-generation system, extending these earlier systems with several years' of experience and further study of rewriting.

3.2 Narrowing and Conditional Narrowing

In the context of combining rewriting with logic programming, one frequently hears the term *narrowing.* The predicate `=>>` implements a kind of narrowing in Bop. In order to avoid confusion, we review here what narrowing is, and what conditional narrowing is.

Narrowing is a mechanism for rewriting a term, in which the term is *unified* with the head of rule whenever the rule is applied (or more precisely, attempted to be applied, since the unfication can fail). Narrowing was considered by Slagle, Lankford, Fay, and Hullot in the mid-1970's, and conditional rules were investigated by Brand and Lankford slightly later, but the combination of narrowing, conditional rewrite rules, logic programming, and functional programming has been studied only recently.

An excellent tutorial on narrowing can be found in [28]. We can define narrowing formally in the following way. Let p, q be terms where p is not a variable, and let s be a nonvariable subterm of p (which we write $p = r[s]$). If there exists a rule *LHS* `=>` *RHS* (which we assume has no variables in common with p), for which there is a most general unifier θ of *LHS* and s, and $q = (r[RHS])\theta$ (the result of replacing s by *RHS* and applying the substitution θ), then we say *p narrows to q.*

A *narrowing* is then a sequence of terms

$$p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$$

such that p_i narrows to p_{i+1} for $1 \leq i \leq n-1$.

Let us cement our understanding of narrowing with an example. If we are given a set of rules such as

```
     0 + X  =>  X.
  s(Y) + Z  =>  s(Y+Z).
```

then a *narrowing* of the term `s(0)+s(0)` is:

$$\begin{array}{clc}
\boxed{\texttt{s(0)+s(0)}} & & \\
\downarrow & \texttt{s(Y)+Z => s(Y+Z)} & \{\ \texttt{Y} \leftarrow \texttt{0},\ \texttt{Z} \leftarrow \texttt{s(0)}\ \} \\
\boxed{\texttt{s(0+s(0))}} & & \\
\downarrow & \texttt{0+X => X} & \{\ \texttt{X} \leftarrow \texttt{s(0)}\ \} \\
\boxed{\texttt{s(s(0))}} & &
\end{array}$$

The result of the narrowing is the term `s(s(0))`. The rules used in the narrowing are shown in the center, and the bindings generated are shown on the right in this diagram.

The narrowing just shown is actually a *reduction*, since the terms rewritten at each step are ground terms (closed terms, terms without variables). Narrowing permits the rewriting of terms *with* variables. For example, one narrowing of the term `s(M)+s(0)` is:

$$\begin{array}{clc}
\boxed{\texttt{s(M)+s(0)}} & & \\
\downarrow & \texttt{s(Y)+Z => s(Y+Z)} & \{\ \texttt{Y} \leftarrow \texttt{M},\ \texttt{Z} \leftarrow \texttt{s(0)}\ \} \\
\boxed{\texttt{s(M+s(0))}} & & \\
\downarrow & \texttt{0+X => X} & \{\ \texttt{M} \leftarrow \texttt{0},\ \texttt{X} \leftarrow \texttt{s(0)}\ \} \\
\boxed{\texttt{s(s(0))}} & &
\end{array}$$

Furthermore, with the term `s(M)+N`, we have multiple (nondeterministic) narrowings. Two narrowings are:

$$\begin{tikzcd}
\boxed{\texttt{s(M)+N}} \arrow[d, "\texttt{s(Y)+Z => s(Y+Z)} \quad \{\ \texttt{Y} \leftarrow \texttt{M},\ \texttt{Z} \leftarrow \texttt{N}\ \}"] \\
\boxed{\texttt{s(M+N)}} \arrow[d, "\texttt{0+X => X} \quad \{\ \texttt{M} \leftarrow \texttt{0},\ \texttt{X} \leftarrow \texttt{N}\ \}"] \\
\boxed{\texttt{s(N)}}
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{\texttt{s(M)+N}} \arrow[d, "\texttt{s(Y)+Z => s(Y+Z)} \quad \{\ \texttt{Y} \leftarrow \texttt{M},\ \texttt{Z} \leftarrow \texttt{N}\ \}"] \\
\boxed{\texttt{s(M+N)}} \arrow[d, "\texttt{s(Y)+Z => s(Y+Z)} \quad \{\ \texttt{M} \leftarrow \texttt{s(Y)},\ \texttt{Z} \leftarrow \texttt{N}\ \}"] \\
\boxed{\texttt{s(s(Y+N))}} \arrow[d, "\texttt{0+X => X} \quad \{\ \texttt{Y} \leftarrow \texttt{0},\ \texttt{X} \leftarrow \texttt{N}\ \}"] \\
\boxed{\texttt{s(s(N))}}
\end{tikzcd}$$

The example above describes narrowing for rules without conditions. For conditional rewrite rules such as

LHS => *RHS* :- *Condition*

narrowing is quite similar, but first performs unification with *LHS* and then checks the logical *Condition*. For example, the term `power(3,0)` can be narrowed using the rule

```
power(X,0)  =>  1  :-  dif(X,0).
```

by first unifying `X` with `3`, checking that `3` is different from `0`, and producing the result `1`. Thus we have a narrowing

$$\texttt{power(3,0)} \ \rightarrow\ \texttt{1}$$

of one step. More generally, the narrowing could have multiple steps. For example, using also the rule

```
power(1,E)  =>  1.
```

we get the two-step conditional narrowing

$$\texttt{power(power(3,0),5)} \ \rightarrow\ \texttt{power(1,5)} \ \rightarrow\ \texttt{1.}$$

3.3 Narrowing vs. Reduction

It is important to be aware of the difference between narrowing and *reduction*, which until the 1980's provided the standard operational semantics of rewriting systems. With reduction, ground terms (terms having no variables) are rewritten to ground terms; and also, in a rule (*LHS* => *RHS*), one typically must guarantee that every variable in *RHS* also appears in *LHS*. With narrowing, by contrast, arbitrary first-order terms are rewritten to first-order terms, and fewer restrictions are made on *LHS* and *RHS* Furthermore, a narrowing produces bindings θ that, in a Prolog system, can affect the term p_1 initially submitted for rewriting.

The distinction between reduction and narrowing is significant, as some computations are possible with narrowing that are not with reduction. The ability to obtain bindings is powerful. For example, with the rule

```
        power(X,0)  =>  1  :-  dif(X,0).
```

there is a *narrowing*

```
                                   power(2,P)  →  1
```

that results in the binding `P = 0`, telling us that (2^P) simplifies to 1 *provided that P* is 0. However, there is no such *reduction*, since `power(2,P)` is not a ground term.

Today a number of popular functional programming languages, notably ML and Miranda, use a rewrite rule-oriented syntax. However, these languages do not yet offer what is provided by Bop:

1. These functional languages use *reduction*, not narrowing.
2. These languages rest on *rewrite rules*, not conditional rewrite rules. As a consequence, they are obliged to provide if-then-else constructs that obscure the readability of the rules. Furthermore, they lose the power of the relational style, which allows non-procedural statement of descriptions (conditions), in a way approximating natural language – the basis of the success of database query languages.
3. These languages offer modularization mechanisms that are more aimed at software engineering (specifically encapsulation, data hiding, and strong typing) than at flexibility and exploration of context paradigms in data access. Although these mechanisms can be added to almost any language, their addition typically causes concessions in flexibility that we do not want to make yet. Many people do not seem to have grasped that the main strength of Prolog is the great *expressiveness* it provides with reasonable performance. Bop is a further step along the line of expressive, reasonable performance languages.

3.4 Complete Term Rewriting

In complete term rewriting, term rewrite rules are applicable at any position within a term. They are to be applied in arbitrary order until no rule is applicable; 'completeness' of a set of rules guarantees that the result of this rewriting is the same, regardless of the selection of rules or the order in which they were appplied.

Below we show the classic group expression simplification rules of Knuth-Bendix. These rules are confluent, and are sufficient for simplifying any expression involving group multiplication, inverses, and identities.

```
        :- op(250,yf,inverse).

        1 inverse               =>   1.
        X inverse inverse       =>   X.
        X inverse * ( X * Y )   =>   Y.
        ( X * Y ) inverse       =>   Y inverse * X inverse.
        1 * X                   =>   X.
        X inverse * X           =>   1.
        X * 1                   =>   X.
        X * X inverse           =>   1.
        X * ( X inverse * Y )   =>   Y.
        ( X * Y ) * Z           =>   X * ( Y * Z ).

        ( X * Y )               =>   ( X * Y ).     % completion
        X inverse               =>   X inverse.     % completion
```

For example, the term

```
        p * (p inverse) * (q * r) inverse * (q * r) * s
```

is rewritten to s as follows (recall * is left associative):

```
        (((p * p inverse) * (q * r)inverse) * (q * r)) * s
        ((1 * (q * r)inverse) * (q * r)) * s
        ((1 * (r inverse * q inverse)) * (q * r)) * s
        ((r inverse * q inverse) * (q * r)) * s
        (r inverse * (q inverse * (q * r))) * s
        (r inverse * r) * s
        1 * s
        s
```

With Bop, the final two completion rules handle the situation where none of the other rules is still applicable. These rules are highly nondeterminate, but because of the way search is performed (applying any possible rule before resorting to the completion rule), the *first* completion obtained by the rules will be the Knuth-Bendix normal form.

3.5 Regular Term Rewriting

The SKI-calculus of combinatory logic is probably the prime example of what are called *applicative term rewriting systems,* term rewriting systems with a special binary application operator. The SKI-calculus can be implemented in Bop as follows: [1]

```
        i => i.
        i(X) => X.

        k => k.
        k(X) => k(X).
        k(X,Y) => X.

        s => s.
        s(X) => s(X).
        s(X,Y) => s(X,Y).
        s(X,Y,Z)  =>  (X@Z) @ (Y@Z).
```

In this implementation, Bop fixed-point rules are used to implement normal forms. All normal form terms (terms that cannot be evaluated further) are defined to be fixed points. Bop evaluation of these terms leaves them unchanged. Subsequent evaluation will result, however, if these terms are applied to arguments.

[1]Note that F @ X is function application, applying the function F to the argument X. Evaluation of this causes evaluation of the term *F(X)*, constructed from F and X using Prolog's metalogical term-building primitives, where arguments are appended at the right. For example, when evaluating `k @ 1` Bop evaluates `k(1)`, and when evaluating `k(1) @ 2` Bop evaluates `k(1,2)`.

Chapter 4

Formal Syntax and Semantics of Bop

The previous sections have given a high-level view of programming in Bop. In this section we show that Bop rules can be thought of as implementing well-known equational, term rewriting or functional programming concepts. Understanding the semantics of Bop programs from these perspectives permits reasoning about Bop programs, and clarifies strengths and weaknesses of Bop.

The semantics of Bop can be presented in several natural ways:

- Bop rules are equations, and the semantics of Bop programs are those of equational logic;
- Bop rules are inequalities that define a partial order => on terms, and the semantics of Bop programs are those of inequational logic;
- Bop rules define a binary relation => of Bop-narrowing on terms, and programmers can use this relation and its composition closure =>>. The composition closure is the least solution of the fixed-point equation

  =>> $=$ (=> $\circ$ =>>)

  where $\circ$ denotes composition of relations. In other words, if we think of => as a mapping, then =>> is the fixed-point iteration of =>.
- Bop rules are function definitions, and programmers can use both the definition relation => and the functional evaluation relation =>>, which evaluates a functional expression to 'normal form' (a term all of whose subterms are completions).
- Bop syntax is just a convenient shorthand for first-order logic, and any Bop rule can be converted directly to a Horn clause. The semantics of a Bop program are those of the corresponding logic program.

Thus one can view a Bop program in several very different, but ultimately equivalent, ways. That Bop unifies these diverse semantics is a strength; it unifies aspects of logic programming, functional programming, term rewriting, and iteration in a simple way. We will show later that further interpretations can be given to Bop rules, including viewing them as grammar rules and as a limited kind of object-oriented method definitions.

4.1 The Syntax of Bop Rules

Bop rules take the form

LHS => *RHS* :- *Condition.*

where *LHS* and *RHS* are first-order terms, and *Condition* is a Prolog goal; this *Condition* can be omitted. The syntax of Bop rules is restricted in one important way: *all nonvariable arguments in LHS must be completions.*

We will assume here without loss of generality that all Bop rules are *linear*, i.e., no variable can appear more than once in *LHS*. There is no loss of generality here since a nonlinear rule like

```
f(X,h(X,Y,X)) => g(X) :- q(X,Z).
```

can be converted to the equivalent linear rule

```
f(V1,h(V2,Y,V3)) => g(X) :- V1 =>> X, V2 =>> X, V3 =>> X, q(X,Z).
```

where `V1`, `V2`, and `V3` are newly-introduced variables. These two rules are equivalent in the sense that equalities implicit in the *LHS* of the first rule are made explicit in the condition of the second. Since the semantics of equality among duplicate *LHS* variables can vary, the assumption of linear rules is made here for clarity. Bop actually does permit nonlinear rules like the one shown above, interpreting them as a shorthand for the corresponding linear rule. However, for this section we will assume that Bop rules are always linear.

Every Bop rule must necessarily be one of the following two kinds:

1. *Completion rules*
   A Bop rule in which *LHS* and *RHS* are identical is a *completion rule.*

   In all term rewriting systems, the process of rewriting will terminate on some terms. In the literature these terms are variously referred to as 'normal forms', 'constructor terms', 'values', and so forth. Bop terms that rewrite to themselves (i.e., terms on which rewriting has no effect, or are fixed-points under rewriting) are called *completions.*

   Completions are declared with completion rules. Examples of predefined completions in Bop are: `true`, `false`, `[]`, `[_|_]`. These are defined by the following completion rules:

   ```
   true    =>  true.
   false   =>  false.
   []      =>  [].
   [H|T]   =>  [H|T].
   ```

   Conditional completion rules are also permissible. For example, the following rule declares all numbers to be completions:

   ```
   X       =>  X         :-  number(X).
   ```

   This rule does not declare variables to be completions. Variables do not rewrite to themselves with this rule – at least not with the definition expected here for `number(X)`. As the rule is written, only numbers are declared to be completions.

   In principle, all terms could be declared to be completions in Bop with the reflexive rule

   ```
   X       =>  X.
   ```

   However, doing this is not a good idea because it would make most programs highly nondeterminate. With this rule every term rewrites to itself, and with any other rules Bop will produce a large number of results (completions, 'normal forms') for most terms being rewritten.

   Many term rewriting systems follow the convention that rewriting terminates when no applicable rule can be found. In Bop, a different convention is followed:

(a) In Bop, rewriting *fails* when no applicable rule can be found.

(b) In Bop, rewriting terminates (*succeeds*, reaches a fixed point) when a completion rule can be applied.

This convention fits the Prolog control model and more generally the logic programming proof search model. Also, it permits interesting variations on term rewriting system control schemes.

2. *Ordering rules*
Rules that are not completions are called Bop ordering rules. Various examples of ordering rules are as follows:

```
if(true,X,_)   =>  X.
if(false,_,Y)  =>  Y.

lazy_and(X,Y)  =>  if(X,Y,false).
lazy_or(X,Y)   =>  if(X,true,Y).
not(X)         =>  if(X,false,true).

power(0,E) => 0  :-  dif(E,0).
power(1,E) => 1.
power(X,0) => 1  :-  dif(X,0).
power(X,E) => Y  :-  number(X), integer(E), E>=1, pow(X,E,Y).

append([],L)       =>  L.
append([X|L1],L2) =>  [X|append(L1,L2)].

int_between(L,L)  =>  L.
int_between(L,U)  =>  int_between(L,M)  :-  M is (U+L)//2.
int_between(L,U)  =>  int_between(M,U)  :-  M is 1+(U+L)//2.

ints(N)  =>  [N|ints(N1)]  :-  N1 is N+1.
```

These rules illustrate how ordinary term rewriting rules (`if`, `lazy_and`, `lazy_or`, `not`), conditional rewriting rules (`power`), nondeterministic rules (when its arguments are unbound `append` also may succeed in more than one way, and `int_between` succeeds in more than one way when its first argument is less than its second) and rules generating infinite structures (`ints`) can be written in Bop.

4.2 Equational Semantics of Bop

Each Bop rule can be viewed as equations between terms, and can be interpreted in the manner typical for equational logic programs, logic programs in which the equality predicate figures prominently (and is often the *only* predicate).

In first-order logic, the axioms of equality can be presented as follows:

Reflexivity axiom

$$\forall x \quad x = x.$$

Symmetry axiom

$$\forall x \forall y \quad x = y \ \leftarrow \ y = x.$$

Transitivity axiom

$$\forall x \forall y \forall z \quad x = z \ \leftarrow \ x = y, \ y = z.$$

Incremental function substitutivity axioms
For each n-ary function symbol f, and for each i, $1 \leq i \leq n$:

$$\forall x_1 \ldots \forall x_n \forall y_i \quad f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n) \ \leftarrow \ x_i = y_i.$$

Predicate substitutivity axioms
For all n-ary predicate symbols p:

$$\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n \quad p(x_1, \ldots, x_n) \ \leftarrow \ p(y_1, \ldots, y_n), \ x_1 = y_1, \ \ldots, \ x_n = y_n.$$

The first-order theory generated by these axioms is called Eq.

We can define the semantics of Bop in terms of equational logic by considering rules to be conditional equations. The rule

$$LHS \texttt{=>} RHS \texttt{:-} q.$$

can be considered to be the conditional equation

$$LHS = RHS \ \leftarrow \ q.$$

This relationship with equational logic immediately gives a semantics for Bop: the semantics for logic programs with the equality relation. Essentially all that is needed to make a set of Bop conditional equations like this into a full logic program is to augment them with the axioms of equality.

In fact, Bop does not use all of the axioms of equality. Its default equality theory is intentionally *not complete*:

- Bop's equality theory omits the symmetry axiom. This omission is typical of computational implementations of equality, especially equality theories that implement functions, since function definitions are 'directed' equations.

- Bop's equality theory omits the reflexivity axiom. In its place, all useful instances of the reflexivity axiom must be provided by completion rules.

- Bop's equality theory can omit some function substitutivity axioms. Specifically, for a given function symbol f of arity n, if there is no Bop ordering rule for f whose *LHS* has a completion appearing in the i-th argument position, then the i-th substitutivity axiom for f

$$f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n) \ \leftarrow \ x_i = y_i$$

  is omitted. We say that f is *lazy* in its i-th argument in this case. For example, in Bop the 'cons' function symbol `[_|_]` is defined by default only by the completion rule shown above, so it is lazy in both its arguments. Similarly since `if` is defined by the ordering rules

```
if(true,X,_)   =>  X.
if(false,_,Y)  =>  Y.
```

  it is lazy in its second and third arguments, and the corresponding function substitutivity axioms are omitted.

The result is that Bop's equality theory is always a fragment $Eq' \subseteq Eq$. The precise fragment is determined by the rules provided as input. Completion rules determine what fragment of reflexivity is available, and ordering rules determine what fragment of substitutivity is available. The full equality theory (even the symmetry axiom, in fact) can be made available if desired, but is omitted by default.

We can formalize the compilation of Bop rules in terms of the equality theory Eq'. Compilation can be performed by partially evaluating Bop rules with the equality axioms shown above. We can think of the equality theory Eq' as an interpreter for Bop rules, and the effect of this interpreter can be incorporated into the compiled Bop rules by using partial evaluation.

First, the compilation algorithm leaves each completion rule

$$LHS = LHS \quad \leftarrow \quad q$$

undisturbed. Second, ordering rules (which we assume here are linear) undergo two transformations:

1. *Resolution with transitivity*
   The equation
   $$LHS = RHS \quad \leftarrow \quad q$$
   is resolved with the middle literal in the transitivity rule
   $$x = z \quad \leftarrow \quad x = y,\ y = z$$
   to yield the consequence
   $$LHS = z \quad \leftarrow \quad q,\ RHS = z.$$
   Bearing the left-to-right computation rule of Prolog in mind, it is significant here that the equation $RHS = z$ is placed rightmost in the clause.

2. *Resolution with incremental function substitutivity*
   The equation is resolved against the incremental substitutivity axioms for each argument in which a non-variable term appears. If the equation is
   $$f(t_1, \ldots, t_n) = z \quad \leftarrow \quad q,\ RHS = z$$
   we resolve with the final literal of transitivity to get
   $$x = z \quad \leftarrow \quad x = f(t_1, \ldots, t_n),\quad q,\quad RHS = z.$$
   If t_i is not a variable in position i, we resolve the second literal with the i-th incremental substitutivity axiom
   $$f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n) \quad \leftarrow \quad x_i = y_i$$
   to obtain
   $$f(t_1, \ldots, t_{i-1}, x_i, t_{i+1}, \ldots, t_n) = z \quad \leftarrow \quad x_i = t_i,\ q,\ RHS = z.$$
   We repeat this for each nonvariable argument $i_1, \ldots, i_m$, and in the end obtain the following clause:
   $$f(x_1, \ldots, x_n) = z \quad \leftarrow \quad x_{i_1} = t_{i_1},\ \ldots,\ x_{i_m} = t_{i_m},\ q,\ RHS = z.$$
   Again, bearing the computation rule of Prolog in mind, it is significant that the subgoals $x_{i_j} = t_{i_j}$ are included in left-to-right order, and appear before the subgoals q, $RHS = z$.

This compilation process guarantees various important properties:

1. If Bop's fragment of equality is Eq', and the Bop program is BP, the result of compilation is a logic program LP such that
$$Eq' \cup BP \vdash LP.$$
Every clause produced using this two-step transformation will be a consequence of the original rules and the equality axioms. We have used only resolution with equality axioms in producing the new rules.

2. The converse is not true, since the equality axioms are not inferrable from the compiled axioms. However we do have a useful approximation of the converse:
$$LP \cup \{ x = x \leftarrow \} \vdash BP.$$
That is, every original Bop equation is a consequence of the compilation *and reflexivity*. Specifically, resolving away the subgoals $x_{i_m} = t_{i_m}$ and $RHS = z$ against the reflexivity axiom $x = x \leftarrow$ ultimately transforms any clause in LP to its original form $LHS = RHS \leftarrow q$ in BP. Thus, with reflexivity, the compiled equations are *equivalent* to the original Bop equations.

3. The compiled equations can be treated as a specialized equality theory in their own right. That is, we can consider the compiled clauses as the definition of the equality relation '=', removing the original axioms of equality.

   Under certain conditions the models of the compiled equations are the same as the models of the original equations and axioms of equality.

4. Using SLD-resolution with the Prolog selection rule, the compiled equations implement *lazy narrowing*, in which the leftmost outermost redex is always selected. Laziness stems not only from the redex selection, but also from the partial use of incremental substitutivity, which avoids use of the equality relation for arguments that were variable in the original Bop rule.

5. Bop's two varieties of rules (completion and ordering rules) permits various programming schemes:

   - If a function symbol appearing as the *LHS* of a completion rule *does not* appear as the *LHS* of any ordering rule, the symbol effectively implements a *constructor* [22]. Rewriting of any constructor term terminates immediately.
   - If a function symbol appearing as the *LHS* of a completion rule *does* also appear as the *LHS* of a ordering rule, the symbol effectively implements a rewriting termination scheme. When the completion rule appears before the ordering rule, we get a *lazy* rewriting scheme again – but in this case provides another kind of laziness, in which the completion rule is chosen first if possible. On the other hand, when the ordering rule appears before the completion rule, we get a scheme that attempts to use transitivity first, and reflexivity afterwards if transitivity fails.

4.3 Partial Ordering Semantics of Bop

We can adapt the semantics of Bop for equational logic to work for partial orders as well.

The standard axioms of partial orders in first-order logic are as follows:

- *Reflexivity axiom*

$$\forall x \quad x \sqsubseteq x.$$

- *Antisymmetry axiom*

$$\forall x \forall y \quad x = y \;\leftarrow\; x \sqsubseteq y,\; y \sqsubseteq x.$$

- *Transitivity axiom*

$$\forall x \forall y \forall z \quad x \sqsubseteq z \;\leftarrow\; x \sqsubseteq y,\; y \sqsubseteq z.$$

- *Function monotonicity axioms*
  For all n-ary function symbols f, if f is $(\sqsubseteq_A, \sqsubseteq_B)$-monotone in its i-th argument $(1 \leq i \leq n)$, then

$$\forall x_1 \ldots \forall x_n \forall y_i \quad f(x_1, \ldots, x_i, \ldots, x_n) \sqsubseteq_B f(x_1, \ldots, y_i, \ldots, x_n) \;\leftarrow\; x_i \sqsubseteq_A y_i.$$

- *Predicate monotonicity axioms*
  For all n-ary predicate symbols p, if p is $\sqsubseteq_A$-monotone in its i-th argument $(1 \leq i \leq n)$, then

$$\forall x_1 \ldots \forall x_n \forall y_i \quad p(x_1, \ldots, x_i, \ldots, x_n) \;\leftarrow\; p(x_1, \ldots, y_i, \ldots, x_n),\; x_i \sqsubseteq_A y_i.$$

In much the same way we developed equational semantics for Bop above, we can develop an inequational semantics. There are two differences.

First, the symmetry axiom is no longer available with inequational semantics. However, this is not only not a problem, it is an advantage. The 'directed' or 'oriented' view of equality discards symmetry anyway. The antisymmetry axiom merely asserts that when two terms x and y are less than or equal to each other, they are equal. This can be viewed either as a constraint on sets of Bop rules (there are no 'loops'; equality is interpreted as syntactic identity) or as a (partial) definition of equality relation. Both make sense here, but we prefer the former, which views antisymmetry as a constraint on Bop rules that prevents infinite loops.

Second, monotonicity axioms have replaced the substitutivity axioms. This is both interesting and appropriate: a function or predicate may or may not be monotone in any given argument, and the ordering $\sqsubseteq_A$ can vary from argument to argument. In short, the inequality theory used for each argument can vary. This is actually useful in programming, and is implemented by the **schema** directive in Bop. Just as omission of a substitutivity axiom yielded laziness in the equational semantics, omission of a monotonicity axiom will have the same effect here.

4.4 Narrowing Semantics of Bop

Bop provides a restricted kind of narrowing, which (for lack of a better term) we will call *Bop-narrowing.* The binary relation `=>` defined by Bop rules leads to a one-step narrowing on terms, and the evaluation relation `=>>` is the multi-step narrowing relation obtained by composing Bop-narrowing steps until reaching a completion. Intuitively, this composition closure will be the least solution of the fixed-point equation

$$\texttt{=>>} \;=\; (\; \texttt{=>} \circ \texttt{=>>} \;)$$

where $\circ$ denotes composition of relations. Let us explain more precisely how.

Bop-narrowing extends the NU-narrowing of [4, 5]. We say that *p narrows to q in one Bop-narrowing step* (with unifier θ) if either:

1. there exists a rule

$$\textit{LHS} \texttt{ => } \textit{RHS} \texttt{ :- } \textit{Condition}$$

(where *Condition* can be trivial) for which p and *LHS* unify, and the goal

:- *unify(p,LHS), Condition*

succeeds[1] with unifier θ, so that $q = RHS\theta$; *or*

2. • there exists a rule *LHS* => *RHS* :- *Condition* for which

$$\begin{aligned} p &= f(s_1,\ldots,s_n) \\ LHS &= f(t_1,\ldots,t_n) \end{aligned}$$

but the goal :- *unify(p,LHS), Condition* does not succeed;

• for some i, s_i narrows to s_i' in one Bop-narrowing step with unifier θ, and:

$$f(s_1\theta,\ \ldots,\ s_{i-1}\theta,\ s_i',\ s_{i+1}\theta,\ \ldots,\ s_n\theta) \quad = \quad q.$$

A *Bop-narrowing* is a narrowing

$$p_1,\ p_2,\ \ldots,\ p_n$$

where, for each i, p_i narrows to p_{i+1} in a Bop-step. We call this *a Bop-narrowing to a completion* if p_n is a completion and p_{n-1} is not.

This definition of Bop-narrowing gives an operational semantics for Bop rules. In a Bop-narrowing step starting at p, we either unify p with the head of some rule, or else try to reduce the distance between p and the head of some rule by recursively performing a Bop-step on one of p's arguments. In other words: outermost narrowing of the term p is performed if possible, and if not, then an inner subterm is narrowed.

The Bop rule

$$f(t_1,\ldots,t_n) \texttt{ => } RHS \leftarrow q$$

is translated to the clause

$$f(x_1,\ldots,x_n) \texttt{ =>> } z \leftarrow x_{i_1} \texttt{ =>> } t_{i_1},\ \ldots,\ x_{i_m} \texttt{ =>> } t_{i_m},\ q,\ RHS \texttt{ =>> } z$$

where $i_1,\ldots,i_m$ are the indices of nonvariable terms in $t_1,\ldots,t_n$. We can prove that, with this translation, *the predicate* =>> *is Bop-narrowing to a completion.* The goal

$$\leftarrow \quad f(s_1,\ldots,s_n) \texttt{ =>> } g$$

succeeds with substitution θ iff there is a Bop-narrowing from $f(s_1,\ldots,s_n)$ to $g\theta$, and $g\theta$ is a completion. The proof is a simple adaptation of that in [5].

The definition above does not specify how inner redex selection works, i.e., which argument s_i is selected in a Bop-narrowing step. As shown in [4] the freedom to select any argument is important in guaranteeing completeness. In the Prolog implementation of Bop, however, a *left-to-right argument selection strategy* is used. From left-to-right, arguments are unified if this is possible, and if not Bop-narrowing is applied to them. This left-to-right selection is implemented by the translation just shown.

Recapitulating, then, if we think of the => predicate as defining one-step Bop-narrowings (this is not accurate, but intuitive), then the =>> predicate defines multi-step Bop-narrowings to a completion. In this way, we can justify the earlier description of =>> as the 'composition closure' of =>.

[1]The predicate *unify* can be defined by the clause *unify(X,X)*.

4.5 Coercions: the Intuitive Operational Semantics of Bop

Having described Bop's semantics in several formal ways, let us point out another, less formal, way of capturing the semantics that captures how it is often used in practice.

We mentioned earlier that completions in Bop are often used as constructors, to define values or data structures. When Bop programs follow this constructor discipline [22], they are intuitively understood as manipulating two kinds of terms: *evaluated terms* (constructors) and *unevaluated terms* (terms that are not guaranteed to be evaluated). Basically, this means that for each value type T we define (via constructors), there are two types: T, and $evaluates\_to(T)$. For example, the *stream* type is defined via the constructors `[]` and `[_|_]`, and the first argument of the `append` function must be of type $evaluates\_to(stream)$.

Programming this way is quite natural. One thinks of unification as a time for performing *coercions* from type $evaluates\_to(T)$ to type T, for each argument that has a defined pattern. Thus we extend unification to a 'unification with coercion' that evaluates input terms if this is explicitly requested by the rule.

Coercion semantics can be reconciled with the other semantics given earlier. For example, with equational semantics we can view the directed equality relation as a coercion.

The concept of coercions can be extended further, and include non-constructor completions. Consider the following mixture of completion rules and ordering rules:

```
square(L)            =>  square(L).
square(L)            =>  rhombic(L,L).
square(L)            =>  rectangle(L,L).

rhombic(B,H)         =>  rhombic(B,H).
rhombic(B,H)         =>  parallelogram(B,H) :- H =< B.

rectangle(B,H)       =>  rectangle(B,H).
rectangle(B,H)       =>  parallelogram(B,H).

parallelogram(B,H)   =>  parallelogram(B,H).
```

Here `square(L)` is a completion, but not a constructor, as it also evaluates to `parallelogram(L,L)`. These rules define a type hierarchy of parallelograms. With it, the term `square(7)` defines a value that can also be coerced to the other values `rhombic(7,7)`, `rectangle(7,7)`, `parallelogram(7,7)`. Potentially then, coercion semantics can implement certain kinds of inheritance, as well as other modes of programming we can devise using mixtures of completion and orderin rules.

Chapter 5

Compilation of Bop Programs

In this section we describe how Bop programs are translated to Prolog. The basic translation is very easy to understand. For the sake of performance, however, several optimizations transform the basic compiled program into a program that reflects partial evaluation and exploits Prolog indexing.

In addition to the basic compilation of Bop rules, a variety of user-selectable options are available for the Bop-to-Prolog translation. These options are provided in the form of *directives*. Compilation of Bop rules, like that for Prolog rules, proceeds in several steps:

1. obtaining a term from the input file
2. term expansion (macro expansion)
3. directive processing (if any)
4. basic translation and optimization.

The following directives can be included in Bop source files in order to obtain special compilations:

```
:- schema(F/N,Schema).
:- memo F/N.
```

Currently these directives must appear *before* all rules involving the function. If they are encountered afterwards, their effect is unpredictable.

5.1 Basic Compilation

Bop programs can incorporate arbitrary Prolog constructs in their conditions. In order to reason about these programs, an operational semantics for Bop in terms of Prolog is needed. This is easily provided since the compilation mapping from Bop to Prolog is straightforward.

Any Bop ordering rule

$$\texttt{f(t}_1\texttt{,...,t}_n\texttt{) => g :- q.}$$

can be compiled to the clause

$$\texttt{f(X}_1\texttt{,...,X}_n\texttt{) => Y :- X}_{i_1}\texttt{ =>> t}_{i_1}\texttt{, ..., X}_{i_m}\texttt{ =>> t}_{i_m}\texttt{, q, g =>> Y.}$$

where $\texttt{t}_{i_1}$, ..., $\texttt{t}_{i_m}$ are the nonvariable terms among the argument terms $\texttt{t}_1$, ..., $\texttt{t}_n$, and $\texttt{X}_{i_1}$,...,$\texttt{X}_{i_n}$ are corresponding newly-introduced variables.

5.2 Optimization

Some optimizations are applied to the logic program resulting from the basic compilation above. A program such as

```
append([],L)     =>  L.
append([H|T],L)  =>  [H|append(T,L)].
```

is first compiled to the form suggested above:

```
append(A,L) =>> B  :-  A =>> [],     L =>> B.
append(A,L) =>> B  :-  A =>> [H|T], [H|append(T,L)] =>> B.
```

The first optimization is to factor common goals from the clauses. For example, the goal `A =>> ...` can be factored, yielding the following:

```
append(A,L)  =>> B  :-  A =>> A1, append_1(A1,L,B).
    append_1([],    L,Y)  :-  L =>> Y.
    append_1([X|T],L,Y)  :-  [X|append(T,L)] =>> Y.
```

Where the compilation described earlier is a kind of unfolding, the technique used here is a kind of folding. This compilation improvement is important, because otherwise equality goals are spawned for the same argument multiple times (like the first argument to append here). This quickly becomes intolerable in a practical system.

The compilation produces code that tries to minimize redundant evaluations. That is, the compiled code tries to evaluate terms only once. All rules are translated the same way, using a simple left-to-right decision tree on the evaluated values of the arguments. The resulting behavior will be determinate or not depending on argument instantiation patterns.

For example, the rules

```
f(C,[])     => g([],C).
f(C,[A|L]) => h(g(A,C),L).
```

are translated to:

```
(f(X,Y) =>> Z) :- !, f_1(X,Y,Z).

   f_1(X,Y,Z) :- (Y =>> T), f_2(X,T,Z).
      f_2(C,[],Z)     :- (g([],C) =>> Z).
      f_2(C,[A|L],Z) :- (h(g(A,C),L) =>> Z).
```

and if the second argument is bound these rules will be determinate. The (non-determinate) rules

```
f([],D)       =>  g([],D).
f(C,[A|L])    =>  h(g(A,C),L).
f([],[])      =>  f([],[]).
f([B|L],[])  =>  f(B,L).
```

are translated to:

```
(f(X,Y) =>> Z) :- !, f_1(X,Y,Z).

   f_1(X,Y,Z) :- (X =>> S), f_2(S,Y,Z).
      f_2([],D,Z) :- (g([],D) =>> Z).
      f_2(X,Y,Z)  :- (Y =>> T), f_3(X,T,Z).
         f_3(C,[A|L],Z)  :- (h(g(A,C),L) =>> Z).
         f_3([],[],f([],[])).
         f_3([B|L],[],Z) :- (f(B,L) =>> Z).
```

Note that the third rule is a completion rule. Completion rules are always translated to Prolog clauses with no subgoals except the condition originally attached to the rule.

It is clear from this translation that each argument is evaluated *at most once*. Also, arguments are evaluated *as needed* by rules top-to-bottom, left-to-right. Furthermore, reduction of all of the needed arguments occurs *before* actual unification with the rule head.

5.3 Treating the Compilation of Laziness with Care

Earlier we considered a small program for computing Ackermann's function:

```
0      =>  0.
s(X)   =>  s(X).

ack(0, 0)        =>  s(0).
ack(0, N)        =>  s(N).
ack(s(M), 0)     =>  ack(M, s(0)).
ack(s(M), s(N))  =>  ack(M, ack(s(M),N)).
```

This program works very well. With it, we get the following listing and execution:

```
| ?- bop_listing(ack).
ack(0,0)=>s(0).
ack(0,A)=>s(A).
ack(s(A),0)=>ack(A,s(0)).
ack(s(A),s(B))=>ack(A,ack(s(A),B)).

ack(A,B)=>>C:-!,ack_1(A,B,C).
   ack_1(A,B,C):-A=>>D,B=>>E,ack_2(D,E,C).
      ack_2(0,0,A):-s(0)=>>A.
      ack_2(0,A,B):-s(A)=>>B.
      ack_2(s(A),0,B):-ack(A,s(0))=>>B.
      ack_2(s(A),s(B),C):-ack(A,ack(s(A),B))=>>C.

yes
| ?- ack( s(s(0)), s(0) ) =>> X.

X = s(s(s(s(s(0))))) ?

yes
```

There is an important point about the way it is written: the first rule forces both arguments to `ack` to be evaluated, so neither argument is lazy. Thus we always get *call by value* evaluation of `ack` expressions.

Now, suppose that we had written this function slightly differently, noticing that the first two rules can be consolidated into a single rule:

```
lack(0, N)        =>  s(N).
lack(s(M), 0)     =>  lack(M, s(0)).
lack(s(M), s(N))  =>  lack(M, lack(s(M),N)).
```

This program is lazy in the second argument for the first rule; `lack` is *not* equivalent to `ack`! Note that is compilation is quite different, and its execution is different as well:

```
| ?- bop_listing(lack).
lack(0,A)=>s(A).
lack(s(A),0)=>lack(A,s(0)).
lack(s(A),s(B))=>lack(A,lack(s(A),B)).

lack(A,B)=>>C:-!,lack_1(A,B,C).
   lack_1(A,B,C):-A=>>D,lack_2(D,B,C).
      lack_2(0,A,B):-s(A)=>>B.
      lack_2(A,B,C):-B=>>D,lack_3(A,D,C).
         lack_3(s(A),0,B):-lack(A,s(0))=>>B.
         lack_3(s(A),s(B),C):-lack(A,lack(s(A),B))=>>C.

yes
| ?- lack( s(s(0)), s(0) ) =>> X.

X = s(lack(s(0),lack(s(0),0))) ?

yes
```

The laziness of the first rule manifests itself in the solution `X`, whose subterms are not completions and will require further evaluation to obtain the value `s(s(s(s(0))))`. The solution `X` is perfectly 'correct', but is lazy.

Lazy evaluation is specified implicitly in Bop, by the appearance of variable arguments in rules. Be careful with functions like `lack` that are lazy in an argument in the first rules, but not in later ones. At times these functions provide precisely what is wanted. In stream computations for example, laziness is beneficial, and a function `f` defined by rules like

```
f([],S)           =>  ...
f([X|Xs],[])      =>  ...
f([X|Xs],[Y|Ys])  =>  ...
```

provides the laziness we want. At other times laziness is not really wanted, and it must be avoided.

Bop extends Prolog unification in a straightforward, predictable way. One look at the *LHS*s of a collection of rules gives a clear understanding of which arguments are evaluated, the order in which they are evaluated, and what their values are expected to be. Multiple reductions of a particular argument are never performed. This approach is somewhat 'operational': ordering of rules and of arguments can affect performance. The same is true of Prolog, however! Since Bop is intended as a practical extension of Prolog for the present, the operational predictabilty of Bop should be seen as a strength, and not a weakness. Bop has both a formal semantics and a practical operational semantics that integrates smoothly with Prolog.

Bop provides a variety of explicit ways to force eagerness in what would otherwise be a lazy computation. For example, the primitive `stream(E)` eagerly evaluates `E` into a Prolog list. A similar 'eager consumer' primitive can be written for each data structure of interest. Also, the second version of `ack` above can be forced to evaluate both arguments with the `schema` directive:

```
:-  schema ack( eager(=>>), eager(=>>) ).

ack(0, N)         =>  s(N).
ack(s(M), 0)      =>  ack(M, s(0)).
ack(s(M), s(N))   =>  ack(M, ack(s(M),N)).
```

With this directive, the program behaves like the first version, using call by value. We explain this directive now.

5.4 Function Schemas: the `schema` directive

Schemas have been included in this version of Bop in order to facilitate the writing of certain kinds of rules, and to encourage experimentation on argument conversion schemes. The many approaches to conditional rewriting that have been proposed [7] suggest that no single compilation scheme will be sufficient. The schema directive is used to declare how the arguments of a function are to be treated. With it one can not only specify the type of an argument, but also how the argument should be converted from its input type to another type for use within the function. Effectively, the schema directive allows the programmer to modify the compilation of each rule, adding type conversion and/or type checking goals for arguments as needed.

Schema declarations can take either of the following forms:

`:- schema` *Function*(*ArgType1*,...,*ArgTypeN*).
`:- schema` *Function*(*ArgType1*,...,*ArgTypeN*) `=>` *ResultType*.

These schema directives may appear anywhere in the program, but govern only those rules that appear *after* the directive.

Schema argument type declarations relate the types of the actual arguments passed to the function with the 'formal arguments' introduced and used in the rule. In other words, the *ArgType* definitions specify binary relationships between actual and formal arguments. In Bop, these definitions can be specified as follows:

ArgType → *Conversion*
ArgType → `lazy`(*Conversion*)
ArgType → `eager`(*Conversion*)

ResultType → *Conversion*

Conversion → *TypeConversionRelation*
Conversion → *TypeConversionRelation*(*TypeCheckRelation*)

TypeConversionRelation → (name of any Prolog predicate of arity 2)

TypeCheckRelation → (name of any Prolog predicate of arity 1)

Any Prolog predicate here should be *side-effect free*, so that multiple invocations all have the same effect. Otherwise the predicates are unrestricted, however, and can backtrack or perform meta-logical tests of their arguments.

In Bop, arguments can be either *eager* or *lazy*. The default is *lazy*, and the following declarations are equivalent:

Conversion
`lazy`(*Conversion*)

An eager argument undergoes type conversion before any subsequent rule is applied. A lazy argument does not undergo type conversion until it is needed.

In Bop, type conversion for an argument is said to be 'needed' in two situations:

1. An argument needs type conversion if it appears with a nonvariable value in the head of a rule:

```
        if(true,S,_)  =>  S.
```

In this rule, the first argument must undergo type conversion, since it appears with the atomic value `true`. The second and third arguments need not undergo type conversion, however, since they appear as variables.

2. An argument needs type conversion if it is unified with (any subterm of) another argument:

```
np(N,P,P) => propernoun(N).
```

In this rule, the second and third arguments must undergo type conversion, and the resulting two values are to unify.

In Bop, actual arguments X passed to the function defined by the **schema** stands in the *Type-ConversionRelation* relation to the corresponding formal argument Y used by the function. In other words, after the schema

:- **schema** *FunctionName(TypeConversionRelation).*

appears, the rule

FunctionName(X) => *FunctionBody(X).*

where X needs type conversion behaves equivalently to the rule

FunctionName(X) => *FunctionBody(Y)* :- *TypeConversionRelation(X, Y).*

(The compiler generates code such that the type conversion between X and Y is done only once, and not in each rule.) Also, after the schema

:- `schema`*FunctionName(TypeConversionRelation(TypeCheckRelation)).*

appears, the rule

FunctionName(X) => *FunctionBody(X).*

where X needs type conversion behaves equivalently to the rule

FunctionName(X) => *FunctionBody(Y) :- TypeConversionRelation(X, Y), TypeCheck-Relation(Y).*

Effectively, the schema directive allows the programmer to modify the compilation of each rule.

The default *ArgType* used in the absence of any schema directive is `=>>`. (or `lazy(=>>)` – which is equivalent). This indicates that the predicate `=>>` is to be used to convert the input argument to a fixed point term. No type checking of the resulting term is performed.

Commonly used type relations are as follows:

TypeConversionRelation → `=`
TypeConversionRelation → `=>>`

TypeCheckRelation → `var`
TypeCheckRelation → `nonvar`
TypeCheckRelation → `atom`
TypeCheckRelation → `atomic`
TypeCheckRelation → `number`
TypeCheckRelation → `integer`

The = conversion relation leaves an argument undisturbed, simply unifying the actual argument X with the formal argument Y.

Some examples of *ArgType* definitions:

```
:-  schema  np( =, =, = ).
:-  schema  if( eager(=>>), lazy(=>>), lazy(=>>) ).
:-  schema  anOrdinaryTwoArgumentBopFunction( =>>, =>> ).
:-  schema  vector_sum( stream, stream ).
:-  schema  sentenceParser( (=), =(parse_tree), stream ).
:-  schema  employee_id( =(person), =(atom), =>>(salary) ).
:-  schema  integral( =>>, =>>, rewriting_join ).
:-  schema  server( message_queue_input(nonempty_queue) ).

stream(X,Y) :- stream(X) =>> Y.

parse_tree( tree(_,_) ).

person(X) :- X is_a person.
salary(X) :- number(X),  0 < X,  X < 200000.

rewriting_join(X,Y) :- X =>> V, Y =>> V.
```

The schema directive can be justified in terms of the inequational semantics and coercion semantics given earlier. The directive specifies how arguments to functions are to be interpreted. With inequality semantics, the directive specifies how the function is monotone. With coercion semantics, it specifies how arguments are coerced.

The schema directive also permits Bop to support various sophisticated kinds of rewriting. In addition to the coercions shown above, we could use *AC-unification* and *λ-unification*, or whatever else is needed. These unification methods are very expensive, and wholesale replacement of the Prolog unifier by them is almost always a very bad idea as far as performance is concerned. The schema directive allows expensive unifiers (or equational theories) to be used *where, and only where, they are needed.*

5.5 Memoized Functions: the memo directive

Functions can be requested to be 'memoed':

```
:- memo F/N.
```

'Memoization' is an old AI programming technique in which

- values of a function are saved in a lookup table when they are produced;
- all evaluations of the function are preceded by a search through the lookup table, which avoids the evaluation if the function value is found.

In other words, memo functions pay the price of maintaining a stored table of previous computations, under the expectation that some future computations will be avoided and that the savings will justify this effort.

In Bop, memo functions are currently implemented with the Prolog internal database (record) mechanism. See the file lib/ctrl_memo.bop for complete details.

Chapter 6

Contexts, Fops, and Bop Term Expansion

We mentioned earlier that Bop offers the ability to define rules that apply in a given context (module, situation, world, etc.). These rules have the form:

Context: LHS => *RHS* :- *Condition.*

where *Context* may be any term. The meaning of this rule is that the contextual expression

Context : LHS

is rewritten to *RHS* provided that *Condition* holds, just as the usual semantics of these rules would suggest. For example, the following rules can be entered:

```
reg_polygon(N,L) : area        => A  :- reg_polygon_area(N,L,A).
reg_polygon(N,L) : perimeter   => P  :- P is N*L.
reg_polygon(N,L) : description => [a_regular_polygon].
```

6.1 Fops

We will call the *Context* in the example above a *fop.* It can be thought of as a function, a binary relation, an object (in the object-oriented programming sense), a module, and as other things as well. The Bop compiler produces relatively efficient code for fop rules, and this enables a variety of programming styles: modular, object-oriented, and so forth.

The '`:`' operator behaves like functional application. The rules shown above apply the definition of `reg_polygon` to an argument (`area`, etc.). The operator '`:`' is left associative by default – so `p:q:r:s` represents `(((p:q):r):s)`.

Both arguments of '`:`' are evaluated, so completion rules are needed before the rules above will operate properly. We can get a working program with the following:

```
area              =>  area.
perimeter         =>  perimeter.
description       =>  description.

equilateral(L)    =>  reg_polygon(3,L).
square(L)         =>  reg_polygon(4,L).
pentagon(L)       =>  reg_polygon(5,L).
reg_polygon(N,L)  =>  reg_polygon(N,L).
```

The final four rules define regular polygons of various types. This gives an inheritance mechanism, so for example we will get the following interaction:

```
| ?-  pentagon(3) : perimeter  =>>  X.

X = 15
yes
```

Another example shows how fops can be used to replace the functional application operator '@'. Earlier we defined the SKI calculus using '@' in the following way:

```
i => i.
i(X) => X.

k => k.
k(X) => k(X).
k(X,Y) => X.

s => s.
s(X) => s(X).
s(X,Y) => s(X,Y).
s(X,Y,Z)  =>  (X@Z) @ (Y@Z).
```

Intuitively, we should be able to translate this to use ':' as follows:

```
i : X       =>  X.
k(X) : Y    =>  X.
s(X,Y) : Z  =>  (X:Z) : (Y:Z).
```

However, the arguments of ':' are evaluated, just as are the arguments of '@'. So, we must again provide the completion rules that make the SKI combinators themselves evaluable:

```
i           =>  i.
i : X       =>  X.

k           =>  k.
k : X       =>  k(X).
k(X)        =>  k(X).
k(X) : _    =>  X.

s           =>  s.
s : X       =>  s(X).
s(X)        =>  s(X).
s(X) : Y    =>  s(X,Y).
s(X,Y)      =>  s(X,Y).
s(X,Y) : Z  =>  (X:Z) : (Y:Z).
```

6.2 Bop Term Expansion

Contexts, fops, and the ':' operator, are implemented in Bop by exploiting term expansion. Many useful features, like the memo feature, can be implemented with term expansion, and very general macro expansion features can be built with it as well.

6.2.1 Operation of Term Expansion

Bop's term expansion capability is like that of Prolog, but has some important differences.

After Bop reads a term `X` while performing a `bop_consult` or `bop_compile`, it first invokes `bop_expand_term(X,Y)` to obtain the term `Y` that ultimately will be compiled and asserted. The predicate `bop_expand_term(X,Y)` performs several kinds of term expansion in series:

1. `bop_term_expansion(X,X1)`
   user-defined Bop term expansion;

2. `contextExpansion(X1,X2)`
   expansion of context statements, as explained below;

3. `expand_term(X2,Y)`
   standard Prolog/user term expansion.

There are two very important caveats concerning `bop_expand_term`: First, *it always succeeds at least once*, but *it can succeed more than once* via backtracking. (This differs from `expand_term/2` in Prolog, which succeeds exactly once.) When it succeeds more than once, *one* input term expands to *multiple* terms for assertion. This feature might be used in many ways, but is used in particular for handling the treatment of Bop context statements, which expand into a sequence of index traversal clauses.

Second, *the user-defined predicate* `bop_term_expansion/2` *should have no side effects.* It is always run twice on each input because of the first requirement, that `bop_expand_term` always succeed at least once. Thus side-effects can, and probably will, have undesired results.

6.2.2 Expansion of Context Statements

Term expansion performs *context expansion*, which we can explain with a few examples as follows. Given a statement

```
(r(X,Y) : Z => t(X,Y,Z) :- C)
```

Bop produces the expansion

```
r(A,B):E => 'r:'(E,A,B).
'r:'(Z,X,Y) => t(X,Y,Z) :- C.
```

The statement

```
(r(X,Y) : s(U,V) => t(X,Y,U,V) :- C)
```

is expanded to

```
r(A,B):E => 'r:'(E,A,B).
'r:'(s(U,V),X,Y) => t(X,Y,U,V) :- C.
```

Finally, the statement

```
(r(X,Y) : t => t :- C)
```

in which `t` is nonvariable is expanded to the single Bop rule

```
r(A,B):t => r(A,B):t  :- C.
```

6.2.3 Expansion of Frames

Bop provides a simple extension of the context/fop mechanism called the *frame*, which permits grouping of statements with a factored context or fop. A frame has the form:

```
Fop  :  {
           Bop rule ,
           ... ,
           Bop rule ,
        }.
```

For example, the rules for `reg_polygon` can be factored as follows:

```
reg_polygon(N,L) : {
        area          =>  A  :-  reg_polygon_area(N,L,A),
        perimeter     =>  P  :-  P is N*L,
        description   =>  [a_regular_polygon]
}.
```

Frames are treated simply as factored expansions. Bop expands the frame

```
r(X,Y): {
        (s(U,V) => t(X,Y,U,V) :- C),
        (u(P,Y,R) => v(X,P) :- D)
}.
```

as though the two rules

```
r(X,Y) : s(U,V) => t(X,Y,U,V) :- C.
r(X,Y) : u(P,Y,R) => v(X,P) :- D.
```

were input instead.

Frames can be nested as well. Given the nested frame

```
r(X,Y): {
        (u(P,Y,R) => v(X,P) :- D),

        s(U,V):{
                (q(W) => t(X,Y,U,V,W) :- C),
                (p(V) => o(U,X,W) :- E)
        }
}.
```

Bop behaves as though the following were input:

```
r(X,Y) : u(P,Y,R) => v(X,P) :- D.
r(A,B) : s(C,D)        =>  'r:s'(C,D,A,B).
'r:s'(C,D,A,B)         =>  'r:s'(C,D,A,B).  % !!!
'r:s'(U,V,X,Y) : q(W)  =>  t(X,Y,U,V,W) :- C.
'r:s'(U,V,X,Y) : p(V)  =>  o(U,X,W) :- E.
```

Although with each of these expansions it might be reasonable to produce coercion statements for the fop in question, like

```
r(A,B) => r(A,B).
```

where `r/2` is the fop name as above, Bop does not do this currently. The ':' operator evaluates both its arguments, and without coercions like this will fail. However, as indicated above some applications like inheritance make it desirable for the fop to be evaluated without necessarily being a completion.

Chapter 7

Bop System Predicates

The Bop system provides a number of Prolog predicates for managing and executing Bop rules. These predicates are of the following kinds:

- File consultation/compilation
- Function definitions and listing
- Bop initialization
- Execution tracing
- Stream output

In addition to these predicates, Bop permits a number of transducers that are commonly used. These are stored in the library subdirectory, and are not described here (yet).

7.1 File Consultation and Compilation

Bop imitates Prolog with basic `consult`-like primitives that read in 'Bop files', files containing directives, Bop rules, and Prolog clauses. Bop rules are translated into Prolog clauses for the predicate `=>>`/2, which can be invoked from Prolog with the goal

BopTerm `=>>` *Result*.

`bop_consult(`*File*`)`
: Consult and translate *File*`.bop` into Prolog clauses.

  Just as with `consult`, `expand_term` is invoked for each term read in. In particular, the Bop rule translator attempts to apply user definitions of `term_expansion` before actually translating a rule.

`bop_consult([`*File*`|`*Files*`])`
: Consult and translate multiple Bop files.

`bop_reconsult(`*File*`)`
: Reconsult and translate *File*`.bop`, clearing all previous definitions of functions encountered.

`bop_reconsult([`*File*`|`*Files*`])`
: Reconsult and translate multiple Bop files.

`bop_flisting(`*File*`)`

Send a bop_listing (with all pertinent directives) to *File*`.pl`. This file can subsequently be consulted, or compiled with Prolog's `compile` or `fcompile` predicates.

`bop_compile(`*File*`)`

Translate *File*`.bop` and then Prolog-compile the result using the Prolog `compile` predicate.

`bop_compile([`*File*`|`*Files*`])`

Translate multiple Bop files and Prolog-compile the result.

`bop_fcompile(`*File*`)`

Translate and perform a Prolog `fcompile`, giving *File*`.ql`. (SICStus Prolog only.)

7.2 Function Definitions and Listings

`bop_listing`

Produces a listing of the entire set of defined functions, along with Prolog clauses that have been `bop_consulted`.

`bop_listing(`*FunctionName*`)`

Produces a listing for all functions of the indicated name. For example, with the rules for primes we could do the following:

```
| ?- bop_listing(primes).
primes => sieve(intfrom(2)).

primes=>>A:-!,primes_1(A).
   primes_1(A):-sieve(intfrom(2))=>>A.

yes
| ?- bop_listing(filter/2).
filter(A,[B|C]) => filter(A,C) :- 0 is B mod A, !.
filter(A,[B|C]) => [B|filter(A,C)].

filter(A,B)=>>C:-!,filter_1(A,B,C).
   filter_1(A,B,C):-B=>>D,filter_2(A,D,C).
      filter_2(A,[B|C],D):-0 is B mod A,!,filter(A,C)=>>D.
      filter_2(A,[B|C],D):-[B|filter(A,C)]=>>D.

yes
```

Note that the original rules are listed along with their translations.

`bop_listing(`*FunctionName/Arity*`)`

Produces a listing for the function of the indicated name and arity.

`current_function(`*FunctionName, FunctionFunctor*`)`

A table of all defined functions, giving both their name (an atom) and their functor (a term of the appropriate arity, with unbound arguments).

`function_value_goal`(*FunctionFunctor, ResultValue, PrologGoal*)

A table of all currently defined defined functions, which can be used to find their value. The Bop system does not use this table directly, but it is provided for user convenience. For example, to find the value `Y` of `f([],X,2)`, we can execute the Prolog goal:

```
?- function_value_goal( f([],X,2), Y, Goal ),  call(Goal).
```

After `bop_consulting` the rules for primes above, we would obtain the following table:

```
| ?- listing(function_value_goal).

function_value_goal([], [], true).
function_value_goal([A|B], [A|B], true).
function_value_goal(true, true, true).
function_value_goal(false, false, true).
function_value_goal(fail, fail, fail).
  ...
function_value_goal(primes, A, primes_1(A)).
function_value_goal(intfrom(A), B, intfrom_1(A,B)).
function_value_goal(sieve(A), B, sieve_1(A,B)).
function_value_goal(filter(A,B), C, filter_1(A,B,C)).

yes
| ?-
```

Note the first six entries are provided initially, and the last four entries permit us to use the translated Prolog clauses.

`function_completion`(*BopTerm, PrologGoal*)

A table of all currently defined fixed point terms. The Bop system does not use this table directly, but it is provided for user convenience. Initially, something equivalent to the following are defined:

```
| ?- listing(function_completion).

function_completion([], true).
function_completion([A|B], true).
function_completion(true, true).
function_completion(false, true).
function_completion(fail, fail).
  ...

yes
| ?-
```

7.3 Bop Initialization

`bop_init`

Initializes the Bop system. Currently this causes the file `/lib/init.bop` to be consulted (using `bop_consult`). Among other things, this results in the following rules being entered:

```
[]      =>  [].
[H|T]   =>  [H|T].
true    =>  true.
false   =>  false.
fail    =>  fail      :-  fail.
```

See Section 6 for more details.

`bop_reinit`
Reinitializes the Bop system.

`bop_reinit(`*File*`)`
Reinitializes the Bop system using the indicated file instead of the default file `init.bop`.

`bop_reinit([`*File*`|`*Files*`])`
Reinitializes the Bop system using the indicated files instead of the default file `init.bop`.

`bop_clear`
Removes all currently defined Bop rules.

`bop_clear(`*FunctionName/Arity*`)`
Removes all currently defined Bop rules for the function indicated.

`bop_clear(`*Functor*`)`
Removes all currently defined Bop rules for the function indicated.

Bop can be used several ways:

1. As an interactive environment for running/debugging Bop code.
2. As a compiler for Bop files.

Bop's use as a compiler is illustrated by the following sessions:

```
% bop
Bop version 0.2
...
?- bop_consult([f1,f2]),  %  Consult 'f1.bop', 'f2.bop'.
   bop_flisting(out12),   %  Write result to 'out12.pl'.
   bop_reinit,            %  Clear and reinitialize Bop.
   bop_compile([f3,f4]),  %  Consult 'f3.bop', 'f4.bop',
                          %  write a flisting, execute bop_clear,
                          %  and Prolog-compile the flisting.
   save_program(sys34).   %  Save the current state in 'sys34'.
```

The above session first compiles two bop files, printing the resulting compilation to a file, and then compiles two others after reinitializing. More sophisticated usage might be as follows:

```
% bop
Bop version 0.2
...
?- bop_clear,             %  Clear ALL current Bop information.
   consult('out12.pl'),   %  Prolog-consult the earlier flisting.
                          %  NOTE:  ALWAYS consult flisting files,
                          %        NEVER bop_consult them:
                          %  'out12.pl' still contains '=>' rules,
                          %  so bop_consult would compile them!
                          %  ALSO: bop_clear before consulting flistings
    ... ,                 %  Do something with the result...
    bop_reinit(myinit),   %  Clear all current Bop information,
                          %  and initialize with 'myinit.bop'
                          %  instead of library(init), the default.
    bop_consult('p.pl'),  %  Consult 'p.pl', ALSO keeping track of all
                          %  predicates defined for future flistings.
    bop_compile(f5),      %  Consult 'f5.bop', make flisting for
                          %  files myinit,p,f5, then bop_clear and
                          %  Prolog-compile the flisting.
    ... .                 %  Do something with the result...
```

Note that `bop_flisting(F)` writes all the current Bop clauses & directives to '`F.pl`', which can be subsequently consulted, compiled, fcompiled, etc. by any Prolog system. Whenever this Prolog system also has the Bop system loaded, `bop_clear` should be called before loading '`F.pl`'.

The predicates `bop_consult`, `bop_reconsult`, and `bop_compile` all initially execute

:- `bop_ensure_loaded(`*InitialBopSource*`).`

when they begin, where *InitialBopSource* is the file or list of files defining the predefined Bop rules (rules users will not want to type in each time). The default *InitialBopSource* is `library(init)`, a file containing basic Bop rules for `[]`, `[_|_]`, etc. Users can override this default with a command like

```
:- bop_reinit([myinit, library(math), 'startup.pl']).
```

to change *InitialBopSource* to be the three files `myinit.bop`, `BOP_DIR/lib/math.bop`, `startup.pl` then execute `bop_clear` and `bop_init`.

7.4 Execution Tracing

Two simple kinds of tracing are provided for Bop rules.

`bop_trace(`*BopTerm*`)`
`bop_trace(`*BopTerm, ResultingValue*`)`

Prints a trace of the recursive evaluations of subterms of *BopTerm*, optionally returning its resulting value. The trace is indented according to the recursion level. For example, with the rules defined earlier for primes, we could obtain the following:

```
| ?- bop_trace(primes).
-> primes
->  sieve(intfrom(2))
->   intfrom(2)
->    [2|intfrom(3)]
=>    [2|intfrom(3)]
=>   [2|intfrom(3)]
->   [2|sieve(filter(2,intfrom(3)))]
=>   [2|sieve(filter(2,intfrom(3)))]
=>  [2|sieve(filter(2,intfrom(3)))]
=> [2|sieve(filter(2,intfrom(3)))]

yes
| ?-
```

The trace lists both each term that is evaluated (passed as a first argument to `=>>/2`), and below this term the ultimate value that the term is evaluated to. Since terms can evaluate to themselves, mild redundancies can appear in the trace, as this example shows.

Prolog-style Call/Exit/Redo/Fail control over the trace is also provided. In the trace above, the single arrows (`->`) indicate a call of `=>>`, while the double arrows (`=>`) indicate an exit of `=>>`. In general, the following symbol conventions are used:

Symbol	Port of `=>>/2`
`->`	call
`=>`	exit
`<-`	redo
`<=`	fail

By using **bop_leash** (see below), users can control interaction at each of these ports.

`bop_fulltrace(`*BopTerm*`)`
`bop_fulltrace(`*BopTerm, ResultingValue*`)`
Like **bop_trace**, but repeatedly presents the trace for the entire *BopTerm* being evaluated. For example, again with the rules for primes we would obtain:

```
| ?- bop_fulltrace(primes).
-> primes
-> sieve(intfrom(2))
-> sieve([2|intfrom(3)])
-> sieve([2|intfrom(3)])
-> [2|sieve(filter(2,intfrom(3)))]

yes
| ?-
```

As this example shows, this trace can again be mildly redundant.

`bop_spy` *FunctionName*
`bop_spy(`*FunctionName*`)`
`bop_spy` *FunctionName/Arity*
`bop_spy(`*FunctionName/Arity*`)`
Puts a Bop spypoint on the indicated function, or on all functions with the indicated function name. If a spypoint already exists, this is indicated. The user is subsequently prompted whenever a function currently with a spypoint is executed.

`bop_nospy` *FunctionName*
`bop_nospy(`*FunctionName*`)`
`bop_nospy` *FunctionName/Arity*
`bop_nospy(`*FunctionName/Arity*`)`
Removes a Bop spypoint on the indicated function, or on all functions with the indicated function name. If no spypoint exists, this is indicated.

`bop_leash(`*LeashingMode*`)`

Changes the current leashing mode for tracing. The leashing mode is a list of ports selected from `call`, `exit`, `redo`, `fail`, optionally preceded by minus signs (which inhibit prompting at the port). Initially, the leashing mode is set with

```
bop_leash([-call,-exit]).
```

This mode is reflected in the traces shown earlier. In these traces, only call and exit ports were displayed, and no user interaction was requested when these ports were encountered.

`bop_debugging`

Lists all current information on Bop leashing, spypoints, etc.

7.5 Bop Flags

Two commands are used to control the setting of flags used by Bop to control interaction with the user:

`bop_flag(`*Flag,Old*`)`
`bop_flag(`*Flag,Old,New*`)`

The `bop_flag` predicate is like the `prolog_flag` predicate provided by some Prolog systems. Here `Flag` can be queried, as with the two-argument version of the predicate, or set, as with the three-argument version. The currently defined flags and their values are listed below:

`unknown`

Acceptable values: `fail`, `trace` (default: `trace`).
When `fail`, attempted evaluation of any undefined function simply fails quietly. Otherwise, a warning is printed and the Prolog debugger (`trace`) is entered.

`redefine_warnings`

Acceptable values: `on`, `off` (default: `on`).
When `off`, redefinitions of Bop functions and Prolog predicates by consult or reconsult of several files are dealt with silently. Otherwise, whenever a function `F` or predicate `P` is encountered in a (re)consulted file that was already defined by another file, a warning is presented to the user, and the redefinition is resolved interactively.

For example, if the files `a.bop` and `b.bop` both contain definitions for `append/2`, we could get the following interaction:

```
| ?- bop_consult(a),  bop_consult(b).
{Bop consult: file a.bop...}
{a.bop: consult complete, 140 msec 1702 bytes}
{Bop consult: file b.bop...}

The Bop function append/2 is defined by two files.
    Old File: a.bop
    New File: b.bop
Do you want the new definition to replace the old? (y/n) n
Do you want the new definition included after the old? (y/n) n
{b.bop: consult complete, 120 msec 552 bytes}

yes
| ?-
```

With this interaction we preserve the definition of `append` in `a.bop`, but otherwise include the rules in `b.bop`.

`single_var_warnings`
Acceptable values: `on`, `off` (default: `on`).
When `off`, 'Prolog style check' violations are ignored. Otherwise, rules having only one occurrence of any (non-anonymous) variable are displayed along with a warning message. (The style check is worth keeping enabled, as it catches what is probably the largest single source of errors in Prolog programming.)

`system_notices`
Acceptable values: `on`, `off` (default: `on`).
When `off`, no Bop system notices (messages in curly braces '`{...}`') are printed.

7.6 Stream Output

Streams have been used so frequently with Bop up to this point that primitives for printing streams have been included in the system.

`print_list(`*BopTerm*`)`
`print_list(`*BopTerm, FileName*`)`
Assuming that *BopTerm* evaluates to a stream, this prints its incremental evaluation *in standard Prolog list format.* If *FileName* is included, the output is sent to this file.

For example:

```
| ?- print_list(append([a,b],[c,d])).
[ a, b, c, d ].

yes
| ?-
```

This is achieved by repeatedly evaluating to yield a term of the form `[H|T]`, printing `H`, and continuing with `print_list(T)`. Termination occurs if the result of evaluation is `[]`. The evaluation done here is determinate, without backtracking, commiting to the first successful evaluation at each step, even if the term in question has multiple (nondeterminate) values.

`print_stream(`*BopTerm*`)`
`print_stream(`*BopTerm, FileName*`)`
Like `print_list`, but prints the incremental evaluation of *BopTerm* as a sequence of Prolog terms terminated by 'full stops'. For example:

```
| ?- print_stream(append([a,b],[c,d])).
a.
b.
c.
d.

yes
| ?-
```

If *FileName* is included, the output is sent to this file.

Chapter 8

The Bop Library

8.1 Builtin Primitives

In the file lib/init.bop are a number of builtin primitives. This file essentially contains a frequently used collection of Bop rules. These rules are (implicitly) loaded by the Bop system, and in most cases will already be available to you when you begin to execute with Bop. This section reviews the primitives in this file.

Bop also permits users to override the init.bop file with their own file, so that any initial definitions can be used. In this way Bop can be tailored to any user's tastes.

8.1.1 Standard Completion Terms

Initially, the following completions are defined:

```
[] => [].
[H|T] => [H|T].

true => true.
false => false.

fail => fail  :-  fail.
```

8.1.2 Undefined Functions

When an undefined function is evaluated, Bop invokes the predicate trap_unknown_function. Its initial definition is:

```
trap_unknown_function(F,Value) :-
      prolog_flag(unknown,trace),
      functor(F,FName,Arity),
      \+ current_function(FName,F),
      printNotice(['Warning:  The function ',FName/Arity,' is undefined.']),
      bopSpyStart(F,Value,FName,Arity,Command),
      call(Command).
```

Since the initialization file can be modified, this predicate can be changed when necessary.

8.1.3 Control Primitives

Bop provides a basic set of control primitives:

(*BopTerm* :- *PrologGoal*)

The :- function evaluates a term after first checking a logical condition. It can be defined with one Bop rule:

```
(BopTerm :- PrologGoal)  => BopTerm  :-  call(PrologGoal).
```

(*BopTerm* => *Term*)
(*BopTerm* =>> *Value*)

These functions allow Bop to retain the results of using Bop rules They can be defined as follows:

```
(BopTerm =>  Term)   => Term    :-  (BopTerm =>  Term).
(BopTerm =>> Value)  => Value   :-  (BopTerm =>> Value).
```

repeat(*BopTerm*)

The **repeat** function simply creates an infinite choice point for a given term:

```
repeat(X)   =>  X.
repeat(X)   =>  repeat(X).
```

if_then_else

Bop has several if-then-else variants. They are easily defined using Bop itself:

```
if(true,X)              =>  X.

if_then(true,X)         =>  X.

if(true,X,_)            =>  X.
if(false,_,Y)           =>  Y.

if_then_else(true,X,_)  =>  X.
if_then_else(false,_,Y) =>  Y.
```

Beyond this, Bop permits the user to write if-then-else statements in an Algol-like syntax by exploiting the Prolog operator feature. In other words, the following are equivalent:

```
         if(Test,Expr1,Expr2)
      if Test then Expr1 else Expr2
```

This is implemented in the following way:

```
:- op(900,fy,(if)).
:- op(901,xfy,(then)).
:- op(900,yfx,(else)).

if(B) => if(B).
else(T,E) => else(T,E).
```

```
if B then T else E      =>  if_then_else(B,T,E) :- !.

if B then T             =>  if_then(B,T).
```

Unfortunately, parentheses must be used to enforce precedence for these operators, since *nested if-then-else statements will not parse properly otherwise.* This is an inherent weakness of Prolog's operator precedence parsing.

8.1.4 Functional Application/Higher-order Mappings

Function @ *BopTerm*

The @ operator applies a Bop function to an argument. It is typically applied with lambda terms, which are defined using the '\' operator; for example, `X\sqrt(X)` is the Bop representation of $\lambda x \,.\, sqrt(x)$. The operation of '@' is not sophisticated (or very efficient), being defined as follows:

```
:- op(800,yfx,(@)).
:- op(700,xfy,(\)).

(X\F)       => (X\F).

(X\FX) @ X0 => FX0   :-  !, betaSubstitution(X,FX,X0,FX0).
F @ X0      => FX0   :-  F =.. L, append(L,[X0],LX0), FX0 =.. LX0.
```

8.1.5 Basic Stream Operators

The initialization file includes a few essential operators on streams.

`member(`*S*`)`

Yields (nondeterministically, via backtracking, the members of stream *S*.

`append(`*S1*,*S2*`)`

The stream concatentation of *S1* and *S2*. This primitive works like the usual **append** predicate, but for streams.

`reverse(`*S*`)`

The reverse of the stream *S*.

`first(`*N*,*S*`)`

The initial substream of *N* elements of *S*. If *S* is shorter than *N*, then the result is *S*.

`stream(`*S*`)`

The entire stream yielded by *S*, generated eagerly. This function is used often! It is also used often in conjuction with `first`, so for example

```
stream(first(10,primes))
```

yields the list of the first 10 prime numbers eagerly, with no suspension.

`print_stream(`*S*`)`

The entire stream yielded by *S*, generated eagerly, with the side-effect of printing to the current output in a 'stream' (readable file) format. For example:

```
| ?- print_stream(append([a,b],[c,d])) =>> _.
a.
b.
c.
d.

yes
| ?-
```

`print_list`(*S*)

The entire stream yielded by *S*, generated eagerly, with the side-effect of printing to the current output, with a list format. For example:

```
| ?- print_list(append([a,b],[c,d])) =>> _.
[ a, b, c, d ].

yes
| ?-
```

8.1.6 Dirty Things

Bop also provides the following control primitives, which cannot be defined with ordinary Bop rules. They should be avoided, and resorted to only when necessary.

`quote`(*BopTerm*)
`=`(*BopTerm*)

These two functions are identical in behavior, and subvert the Bop convention that all results be completions. The `quote` function can be defined:

```
quote(BopTerm)  =>>  BopTerm.
```

Often `quote(T)` can be avoided simply by using a *bona fide* completion (like `[T]`) to inhibit evaluation.

`!`(*BopTerm*)

The `!` function simply cuts after evaluating its argument:

```
!(BopTerm)  =>>  Value  :-  BopTerm =>> Value, !.
```

`postcondition`(*BopTerm*,*PrologGoal*)

The `postcondition` function checks *PrologGoal* after evaluating *BopTerm*:

```
postcondition(BopTerm,PrologGoal) => Value  :-
                BopTerm =>> Value, call(PrologGoal).
```

8.2 The Math Libraries

The Bop library contains the following files:

- `math.bop`
  Bop functions for all basic math operations. These include the constant definitions from `math.h` in Un*x:

```
e                 =>  2.7182818284590452354.
log2_e            =>  1.4426950408889634074.
log10_e           =>  0.43429448190325182765.
ln_2              =>  0.69314718055994530942.
ln_10             =>  2.30258509299404568402.
pi                =>  3.14159265358979323846.
pi_over_2         =>  1.57079632679489661923.
pi_over_4         =>  0.78539816339744830962.
one_over_pi       =>  0.31830988618379067154.
two_over_pi       =>  0.63661977236758134308.
two_over_sqrt_pi  =>  1.12837916709551257390.
sqrt_2            =>  1.41421356237309504880.
sqrt_1_over_2     =>  0.70710678118654752440.
```

and the standard math library functions:

```
X^Y, sqrt(X),
log1p(X), log10(X), log(X),
exp(X), expm1(X),
sin(X), cos(X), tan(X),
asin(X), acos(X), atan(X),
sinh(X), cosh(X), tanh(X),
floor(X).
```

Includes `math_pl.pl` and `math_arith.bop`.

- `math.c`
  The Prolog foreign functions for the Un*x `libm` math library.

- `math_arith.bop`
  Defines the following Bop expressions for Prolog numbers:

  `-X, X*Y, X+Y, X-Y, X/Y.`

  Includes `math_arith_pl.pl` and `numbers.bop`.

- `math_arith_pl.pl`
  Defines the following Prolog predicates for Prolog numbers:

  `bopMinus(X,Z), bopMinus(X,Y,Z), bopPlus(X,Y,Z), bopTimes(X,Y,Z), bopDiv(X,Y,Z).`

  These are all implemented with the Prolog `is` predicate, but work like constraints: any two of the three arguments may be defined.

- `math_pl.pl`
  Defines the following Prolog predicates:

```
bopPow(X,Y,XtoY), bopSqrt(X,SqrtX),
bopLog1p(X,Log1pX), bopLog10(X,Log10X), bopLog(X,LogX),
bopExp(X,ExpX), bopExpm1(X,Expm1X),
bopSin(X,SinX), bopCos(X,CosX), bopTan(X,TanX),
bopAsin(X,AsinX), bopAcos(X,AcosX), bopAtan(X,AtanX),
bopSinh(X,SinhX), bopCosh(X,CoshX), bopTanh(X,TanhX),
bopFloor(X,FloorX),
```

This file includes the Prolog foreign function interface for the Un*x `libm` math library.

- `math_sqrt.pl`
  Defines the predicate `sqrt(X,Y)` in Prolog, avoiding the loading in of the standard math library (at a price in performance).
- `numbers.bop`
  Defines Prolog numbers to be completions:

```
        X  =>  X  :-  number(X), !.
```

8.3 Aggregates

8.3.1 agg_moments.bop

`count_avg`(*Stream*)
  Takes a numeric *Stream* and yields `[Count,Avg]`.

`count_avg_variance`(*Stream*)
  Takes a numeric *Stream* and yields `[Count,Avg,Variance]`.

`count_avg_stddev`(*Stream*)
  Takes a numeric *Stream* and yields `[Count,Avg,Stddev]`.

`moments1`(*Stream*)
  Takes a numeric *Stream* and yields `[moment_values(N,XbarN)]`.

`moments2`(*Stream*)
  Takes a numeric *Stream* and yields `[moment_values(N,XbarN,M2)]`.

`moments3`(*Stream*)
  Takes a numeric *Stream* and yields `[moment_values(N,XbarN,M2,M3)]`.

`moments4`(*Stream*)
  Takes a numeric *Stream* and yields `[moment_values(N,XbarN,M2,M3,M4)]`.

8.3.2 agg_quantiles.bop

`quantiles`(S,N)
  The ordered stream of N values that would partition the result of sorting the numeric stream S into N equal segments.

`median`(S)
  The true median of the numeric stream S.

`approx_median`(S)
  An approximate median of the random numeric stream S, obtained by an algorithm of Pearl [27].

`approx_quantile`(S,Q)
  An approximate Q-th quantile of the random numeric stream S, obtained by an algorithm of Pearl [27].

8.3.3 agg_stat.bop

`length(`*S*`)`
: The length of stream *S*.

`count(`*S*`)`
: The length of stream *S*.

`min(`*S*`)`
: The least item of a numeric stream *S*.

`max(`*S*`)`
: The greatest item of a numeric stream *S*.

`sum(`*S*`)`
: The sum of all items in a numeric stream *S*.

`sumsq(`*S*`)`
: The sum of squares of all items in a numeric stream *S*.

`aggregate(`*Op*,*S*`)`
: The numeric result obtained by evaluating the expression that interjects the binary, right-associative operator *Op* between all elements of *S*.

InfixOp	Resulting computation
`+`	sum of all elements in *S*
`*`	product of all elements in *S*
`/\`	logical *and* of all elements in *S*
`\/`	logical *or* of all elements in *S*
`count`	count of all elements in *S*
`min`	minimum of all elements in *S*
`max`	maximum of all elements in *S*
`sum`	total of all elements in *S*
`avg`	average of all elements in *S*
`sumsq`	sum of squares of all elements in *S*

8.4 Control Functions

8.4.1 ctrl_backtrack.bop

`backtracking(`*S*`)`
: Causes items in stream *S* to be saved as they are found, using the undo/1 primitive, which records them on the Prolog trail. Backtracking causes these saved items to be recorded and reused. *S* is not actually allowed to backtrack itself; instead we reuse the first stream that *S* yields.

`commit_backtracking(`*S*`)`
: Announces that failure after this point should not cause recovery of earlier parts of the stream whose current tail is *S* (i.e., that the earlier part of the stream no longer need be saved).

8.4.2 ctrl_memo.bop

`memo`(*S*,*Id*)

The stream *S*, recording items as they are obtained. *Id* is the (atomic) name of the memo entry for *S*. When *Id* is initially unbound, it is bound with a new memo identifier. When *Id* is bound, it is taken to be an existing memo identifier. This function permits backtracking, as well as multiple accesses of a stream by different threads.

Two Prolog primitives can be used to manage memoization:

```
current_memo(S,Id)  :-  stream S has been memo'ed as Id.
clear_memo(Id)      :-  all memo information for Id is removed.
```

Note: Since memo stream terms are `record`'ed currently, they should be ground.

8.5 Filters

8.5.1 filter_indices.bop

`select_indices`(*Indices*,*S*)

The substream of *S* determined by the (ordered) stream of offsets *Indices*. For example, if *Indices* is `[2,7,18]`, then the result will be a stream containing the second, seventh, and eighteenth items in stream *S*.

8.5.2 filter_random.bop

`random_subsequence`(*S*)

One randomly-selected substream of *S*.

`random_combination`(*S*,*M*)

One randomly-selected subsequence of *S*, of size *M*.

`random_partition`(*S*)

One randomly-selected partition `[L|R]` of *S*.

`random_permutation`(*S*)

One randomly-selected permutation of *S*.

8.5.3 filter_select.bop

`select`(*S*,*Pattern*,*Test*)

The substream of *S* containing those items that both match *Pattern* and satisfy the Prolog goal *Test*.

`selectUntil`(*S*,*Pattern*,*Test*)

The prefix stream of *S* items that either do not match *Pattern* or do not satisfy the goal *Test*.

8.5.4 filter_uniq.bop

`uniq`(*S*)

The substream of *S* of all items with adjacent duplicates removed. Equivalent to `uniq(`*S*`,all)`.

`uniq`(*S*,*Option*)

The substream of *S* obtained according the value of *Option*:

`uniq(`*S*`,all)`	the (sub)stream of items with duplicates removed
`uniq(`*S*`,singles)`	the stream of items appearing without duplicates
`uniq(`*S*`,multiples)`	the stream of items appearing only with duplicates
`uniq(`*S*`,count_all)`	the stream of (N-X) pairs; N is count of duplicates
`uniq(`*S*`,count_singles)`	the stream of (1-X) pairs where X had no duplicates
`uniq(`*S*`,count_multiples)`	the stream of (N-X) pairs where X had duplicates

8.6 Enumerators

An *enumerator* is a function that yields all members of a particular set, one at a time, *via backtracking.*

8.6.1 enum_combinatorial.bop

`subsequence(`*Stream*`)`
: Lazily enumerates all subsequences of *Stream.* That is, subsequences are produced incrementally.

`eager_subsequence(`*Stream*`)`
: Eagerly finds all subsequences of *Stream.* That is, a complete subsequence is produced each time.

`combination(`*Stream,Number*`)`
: Lazily enumerates all combinations of *Number* values chosen from *Stream.* Here *Number* must be a nonnegative integer.

`eager_combination(`*Stream,Number*`)`
: Eagerly enumerates all subsequences of size *Number* from *Stream.*

`partition(`*Stream*`)`
: Enumerates all possible binary partitions `[P1|P2]` of *Stream,* where `P1` and `P2` are subsequences of *Stream.*

`eager_partition(`*Stream*`)`
: Like `partition`, but eager.

`permutation(`*Stream*`)`
: Enumerates all possible permutations of *Stream.*

`eager_permutation(`*Stream*`)`
: Like `permutation`, but eager.

8.6.2 enum_filestream.bop

`file_term(`*Filename*`)`
: Enumerates terms read from *Filename.*

`file_byte(`*Filename*`)`
: Enumerates bytes read from *Filename.*

`instream_term(`*ReadStream*`)`
: Enumerates terms read from open Prolog *ReadStream.*

`instream_byte(`*ReadStream*`)`
: Enumerates bytes read from open Prolog *ReadStream.*

8.6.3 enum_number.bop

`integer_between`(*LowerBound, UpperBound*)
: Enumerates all integers *I* between *LowerBound* and *UpperBound.*

8.6.4 enum_prefix.bop

`array_term`(*Array*)
: Enumerates entries from (functor) *Array.*

`frontier_term`(*Term*)
: Enumerates leaf/external subterms of *Term.*

8.6.5 enum_window.bop

`window`(*N,S*)
: Enumerates contiguous substreams ('windows') of size *N* in the stream *S.*

8.7 Generators

A *generator* is a function that yields all members of a particular set, one at a time, *in a stream.*

8.7.1 gen_filestream.bop

`file_terms`(*Filename*)
: Enumerates terms read from *Filename.*

`file_bytes`(*Filename*)
: Enumerates bytes read from *Filename.*

`instream_terms`(*ReadStream*)
: Enumerates terms read from open Prolog *ReadStream.*

`instream_bytes`(*ReadStream*)
: Enumerates bytes read from open Prolog *ReadStream.*

8.7.2 gen_number.bop

`constant_stream`(*C*)
: The infinite stream *[C, C, C, ...].*

`constant_stream`(*C,N*)
: The stream *[C, C, ..., C]* of length *N.*

`ramp`(*M*)
: The infinite stream *[M, M+1, M+2, ...].*

`ramp`(*M,N*)
: The stream *[M, M+1, M+2, ..., N].*

`step`(*M,K*)
: The infinite stream *[M, M+K, M+2K, ...].*

`ramp(`*M,K,N*`)`
: The stream *[M, M+K, M+2K, ..., M+nK]*, where $M+nK+K > N$. The same stream as would be generated by '**for** $i = M$ **step** K **until** N'.

8.7.3 gen_random.bop

`random_unit_reals`
: Generates random real numbers in the unit interval.

`random_unit_reals(`*Seed1,Seed2,Seed3*`)`
: Generates random real numbers in the unit interval, using *Seed1, Seed2, Seed3* as initial random seed values.

`random_reals(`*A,B*`)`
: Generates random reals in the interval $[A,B]$.

`random_integers(`*M,N*`)`
: Generates random integers in the interval $[M,N]$.

`random_zero_ones`
: Generates random 0-1 values.

`random_booleans`
: Generates random `true-false` values.

8.8 Stream Operators

8.8.1 map_project.bop

In the following functions, *Index* is always a (small) integer, specifying an argument index. *Indices* are lists of term argument addresses; each term argument is either an integer (giving an argument index), or a list of integers (giving a path of argument indices). For example, `[2, [1,2,4], 3]` is acceptable as *Indices.* It selects a list consisting of: the 2nd argument, the 4th argument of the 2nd argument of the 1st argument, the 3rd argument of a given term.

`project_arg(`*S,Index*`)`
: The stream of *Index*'th arguments from terms in stream *S*.

`project_key_arg(`*S,Index*`)`
: The stream of *(Arg-X)* pairs from items *X* in *S*, where *Arg* is the *Index*'th argument of *X*.

`project_args(`*S,Indices*`)`
: The stream of indexed arguments from terms in *S*.

`project_key_args(`*S,Indices*`)`
: The stream of *(Key-X)* pairs from items *X* in *S*, where *Key* is the list of arguments of *X* given by *Indices.*

8.8.2 ops_compare.bop

`compare(`*X, Y*`)`
: The result *C* from among `<`, `=`, `>` obtained by the Prolog goal `compare(`*C*`,`*X*`,`*Y*`)`.

`stream_compare(`*S,T*`)`
: The stream of pairwise `compare` results applied to terms from the streams *S* and *T*.

8.8.3 ops_join.bop

`keysorted_natural_join`(S,T)
: The *join* of the streams obtained by **`keysorting`** S and T.

`keysorted_equijoin`(S,T)
: The *equijoin* of the streams obtained by **`keysorting`** S and T.

`equijoin_on_key_arg`(S,*SIndex*,T,*TIndex*)
: The keysorted equijoin of S and T, where equality is tested by comparing subterms at position *SIndex* of terms in S with subterms at position *TIndex* of terms in T.

`equijoin_on_key_args`(S,*SIndices*,T,*TIndices*)
: The keysorted equijoin of S and T, where equality is tested by comparing the list of subterms at positions *SIndices* of terms in S with the list of subterms at positions *TIndices* of terms in T.

8.8.4 ops_sets.bop

`sorted_union`(S,T)
: The stream resulting from collectively merging the streams for S and T, which should be already sorted. Duplicates are omitted from the result.

`sorted_intersection`(S,T)
: The stream resulting from selectively merging the streams for S and T, which should be already sorted. Duplicates are omitted from the result.

`sorted_difference`(S,T)
: The stream of terms in stream S that do not appear in stream T, where both streams should be sorted.

`union`(S,T)
: The stream `sorted_union(sort(`S`,`T`))`.

`intersection`(S,T)
: The stream `sorted_intersection(sort(`S`,`T`))`.

`difference`(S,T)
: The stream `sorted_difference(sort(`S`,`T`))`.

8.8.5 sorting.bop

`sort`(S)
: The result of sorting stream S. Duplicates are omitted from the result.

`keysort`(S)
: The result of keysorting stream S, which should be a stream of *Key-Term* pairs. Duplicates are *not* omitted from the result.

8.8.6 sorting_ranks.bop

`ranks`(S)
: A stream of indices (a permutation of *[1,2,...,N]*, where N = *length(S)*), such that the i-th result item gives the rank of the i-th element of S.

Chapter 9

Narrowing Grammar

Logic programming has led to new insights into parsing. A *logic grammar* has rules that can be represented as Horn clauses which can then be executed for either acceptance or generation of the language specified.

Narrowing Grammar is a new kind of logic grammar that combines concepts from logic programming, rewriting, lazy evaluation, and specific logic grammar formalisms such as Definite Clause Grammar (DCG). A Narrowing Grammar is a finite set of Bop rewrite rules. What differentiates Narrowing Grammar from Bop is that there is a style, and a predefined set of operators or functions, with which the rules are written.

Reddy [28] proposed interpreting rewrite rules using narrowing. In narrowing, variables of a rewritten term can be bound, and therefore narrowing can serve as a basis for extended logic grammars in the same way that unification has served up to this point.

By virtue of its implementation in Bop, Narrowing Grammar is directly executable, like many logic grammars, and can be executed both as a generator and as an acceptor. Unlike many logic grammars, Narrowing Grammar also permits higher-order specification and modular composition, and provides lazy evaluation by virtue of its rewriting strategy. Lazy evaluation is important in certain language acceptance situations, such as in coroutined matching of multiple patterns against a stream.

Narrowing Grammar offers the following general capabilities.

- Support for *complex* pattern matching.
- Generation/acceptance in the same mechanism.
- Programming/rewriting in the same mechanism.
- Some new perspectives on grammatical formalisms.

Below, we offer only a brief summary of narrowing grammar; for more information, see [4, 10, 5]. These references rely on NU-narrowing, which is the restriction of Bop-narrowing to the case where the conditions on rewrite rules are always trivial.

9.1 Examples of Narrowing Grammar

It is perhaps easiest to understand Narrowing Grammar by studying a few examples. In this section we present examples that illustrate some of the more interesting capabilities.

All the examples involve a particular kind of Bop definition. The rule

```
    p => [a,b,c].
```

defines a *pattern* (nonterminal symbol) p that *generates* the stream (terminal string) [a,b,c]. Thus Bop's handling of streams permits it to implement grammars easily.

Regular Expression Grammar

In Bop we can quickly develop rules for regular algebra:

```
[]        =>  [].

[H|T]     =>  [H|T].

(X;Y)     =>  X.
(X;Y)     =>  Y.

(X+)      =>  X.
(X+)      =>  X,(X+).

([],L)       =>  L.
([H|T],L)    =>  [H|(T,L)].
```

Here ',' is an operator that implements concatenation (append) of patterns. If pattern p generates [a,b,c], and pattern q generates [d,e], then the pattern p,q generates [a,b,c,d,e].

To understand pattern generation more deeply, consider only the rules for Kleene '+' and concatenation. With these, we can obtain the following Bop-narrowing:

```
     Rewritten term               Rule used
  ============================================================
  ([a]+,[b])
                                  (X+) => X,(X+).
  (([a],[a]+),[b])
                                  ([H|T],L) => [H|(T,L)].
  ([a|([],[a]+)],[b])
                                  ([H|T],L) => [H|(T,L)].
  [a|(([],[a]+),[b])]
                                  ([],L) => L.
  [a|([a]+,[b])]
                                  (X+) => X.
  [a|([a],[b])]
                                  ([H|T],L) => [H|(T,L)].
  [a,a|([],[b])]
                                  ([],L) => L.
  [a,a,b]
```

Narrowing grammar can also be used for pattern recognition (parsing, acceptance), in which we match a pattern P against a stream S. To accomplish this, we introduce a primitive match(P,S) that succeeds only when P matches a prefix of S:

```
match([],S) => S.
match([X|L],[X|S]) => match(L,S).
```

Thus we use match as a parser, as the following narrowing shows:

```
match( ([a]+,[b]), [a,a,b] )
match( (([a],[a]+),[b]), [a,a,b] )
match( ([a|([],[a]+)],[b]), [a,a,b] )
match( [a|(([],[a]+),[b])], [a,a,b] )
match( (([],[a]+),[b]), [a,b] )
match( ([a]+,[b]), [a,b] )
match( ([a],[b]), [a,b] )
match( [a|([],[b])], [a,b] )
match( ([],[b]), [b] )
match( [b], [b] )
match( [], [] )
[]
```

Higher-Order Patterns

Narrowing grammar has several very novel features. One is the ability to pass patterns as arguments to other patterns, allowing us effectively to implement *higher-order patterns*. For example, if we define

```
number(P,N)    =>  number(P,N,0).

number(P,N,N)  =>  [].
number(P,N,M)  =>  P,number(P,N,M1)   :-  succ(M1,M).
```

then `number(P,N)` is a pattern that will generate (or match) a stream of `N` consecutive copies of `P`. For example, we can use it to generate a stream with the following narrowing:

```
number(([a]+,[b]),2)
number(([a]+,[b]),2,0)
([a]+,[b]), number(([a]+,[b]),2,1)
([a],[b]), number(([a]+,[b]),2,1)
[a|([],[b])], number(([a]+,[b]),2,1)
[a| (([],[b]), number(([a]+,[b]),2,1))]
[a| ([b], number(([a]+,[b]),2,1))]
[a| [b| ([], number(([a]+,[b]),2,1))]]
[a| [b| number(([a]+,[b]),2,1)]]
[a| [b| (([a]+,[b]), number(([a]+,[b]),2,2))]]
[a| [b| (([a],[b]), number(([a]+,[b]),2,2))]]
[a| [b| [a| (([],[b]),number(([a]+,[b]),2,2))]]]]
[a| [b| [a| [b| number(([a]+,[b]),2,2)]]]]
[a| [b| [a| [b| []]]]]
```

Coroutined Pattern Matching

Another interesting feature of Narrowing grammar is its ability to implement coroutined pattern matching. Many logic grammars are also not lazy, yet lazy evaluation can be very important in parsing. It allows coroutined recognition of multiple patterns against a stream. Bop permits the integration of lazy evaluation with logic grammars, and to our knowledge this is new.

With the definition of the `//` operator

```
([X|Xs] // [X|Ys]) => [X|Xs//Ys].
([] // []) => [].
```

we coroutine two patterns. The compound pattern `(P // Q)` simultaneously matches or generates `P` and `Q`. The operator `//` yields the stream pattern `[X|(Xs//Ys)]` when its two argument patterns narrow, respectively, to `[X|Xs]` and `[X|Ys]`. The effect is 'coroutined' narrowing of the two argument patterns, since the arguments are recursively constrained to narrow to streams whose heads match. Ultimately `//` constrains both argument patterns to narrow to the same stream.

For example, consider the grammar:

```
s_abc => ab_c // a_bc.

ab_c => pair([a],[b]), [c]*.
a_bc => [a]*, pair([a],[b]).

pair(X,Y) => [].
pair(X,Y) => X, pair(X,Y), Y.
```

This grammar imposes simultaneous (parallel) context-free constraints ($a^n b^n c^*$ and $a^* b^n c^n$) on the streams generated or matched by the pattern `s_abc`. It thus produces the stream of values $a^n b^n c^n$. That is, it relies on the fact that

$$\{a^n b^n c^n \mid n \geq 0\} \quad = \quad \{a^n b^n c^* \mid n \geq 0\} \ \cap \ \{a^* b^n c^n \mid n \geq 0\}.$$

There is much more to say about Narrowing Grammar. For a formal treatment and further examples, see [4, 10, 5].

Acknowledgement

Bop owes a great deal to many people. First, Sanjai Narain came up with the initial inspiration for implementing lazy evaluation in Prolog so elegantly, and developing the Log(F) system – whose influence is still reflected by Bop in many ways. Working with him has been a great source of inspiration. Paul Eggert has been a unfailing generator of enthusiasm, and a wonderful tester of ideas; many improvements in Bop's design are results of discussions with him. Also, Bop is now 'copylefted freeware' thanks to him. Lewis Chau and Eugene Paik clarified many issues concerning the development of rewriting systems in Prolog. Lewis was responsible for pushing the Narrowing Grammar idea to completion, and exploring the advantages of narrowing over reduction. Many other students at UCLA have also contributed to Bop, including Bijan Arbab, Arman Bostani, Mike Coleman, Ed Dergharapetian, Lars Hagen, Shen-Tzay Huang, Ted Kim, Deborah Ogg, Prasad Ram, and Xin Wang. Finally, Maarten van Emden helped the development of Bop enormously with his generous hospitality. Central issues concerning the semantics of Bop were resolved, and most of this manual was completed, during a visit with his logic programming group at the University of Victoria during August 1991.

Bibliography

[1] H. Abelson and G. Sussman. *The Structure and Analysis of Computer Programs.* MIT Press, Boston, MA, 1985.

[2] M. Bellia and G. Levi. The Relation Between Logic and Functional Languages: A Survey. *J. Logic Programming*, 3:185–215, 1986.

[3] B. Buchberger and R. Loos. Algebraic Simplification. *Computing*, Suppl. 4:11–43, 1982. In *Computer Algebra: Symbolic and Algebraic Computation*, ed. by B. Buchberger, G.E. Collins, R. Loos.

[4] H.L. Chau. Narrowing Grammar: A Comparison with Other Logic Grammars. Technical report, MIT Press, Cleveland, OH, October 1989.

[5] H.L. Chau and D.S. Parker. Narrowing Grammar. Technical report, MIT Press, Lisbon, Portugal, September 1989.

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

[7] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical Conditional Rewrite Systems. In *Proc. Conference on Automated Deduction (CADE-9)*, pages 538–549, New York, 1988. Springer-Verlag (LNCS #310).

[8] N. Dershowitz and D.A. Plaisted. Logic Programming cum Applicative Programming. In *Proc. Symp. on Logic Programming*, pages 54–66, Boston, 1985.

[9] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition. In *Proc. Symp. on Logic Programming*, pages 172–184, Boston, 1985.

[10] Ph.D. Thesis H.L. Chau. Narrowing Grammar: A Lazy Functional Logic Formalism for Language Analysis. Technical report, UCLA Computer Science Dept., September 1989.

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[12] G. Huet and D. Oppen. Equations and Rewrite Rules: A Survey. In R.V. Book, editor, *Formal Languages: Perspectives and Open Problems.* Academic Press, 1980.

[13] A. Josephson and A. Dershowitz. An Implementation of Narrowing. *J. Logic Programming*, 6(1-2):57–77, 1989.

[14] G. Kahn and D.B. MacQueen. Coroutines and Networks of Parallel Processes. In *Proc. IFIP 1977*, pages 993–998. North-Holland, 1977.

[15] C. Kirchner. *Unification.* Academic Press, San Diego, CA, 1990.

[16] G. Levi, C. Palamidessi, P.G. Bosco, E. Giovanetti, and C. Moiso. A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions. In *Proc. Symp. on Logic Programming*, pages 318–327, San Francisco, 1987.

[17] M.D. McIlroy. Coroutines. Technical report, Bell Telephone Laboratories, Murray Hill, NJ, 1968.

[18] M.D. McIlroy. Squinting at Power Series. *Software – Practice and Experience*, 20(7):661–683, July 1990.

[19] S. Narain. A Technique for Doing Lazy Evaluation in Logic. *J. Logic Programming*, 3(3):259–276, October 1986.

[20] S. Narain. LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation. Technical report, Rand Corporation, Santa Monica, CA, 1987.

[21] S. Narain. LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation. Technical report, UCLA Computer Science Dept., 1988.

[22] M.J. ODonnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, MA, 1985.

[23] E.S. Paik. FLOP: Functional Logic Programming. Technical report, UCLA Computer Science Dept., December 1988.

[24] D.S. Parker. Stream Data Analysis in Prolog. In L. Sterling, editor, *The Practice of Prolog.* MIT Press, Cambridge, MA, 1990.

[25] D.S. Parker, E. Simon, and P. Valduriez. SVP – A Model Capturing Sets, Streams, and Parallelism. In *Proc. 18th Intnl. Conf. on Very Large Databases*, Vancouver, 1992. Full version available as UCLA Technical Report CSD-9200020, April 1992.

[26] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[27] J. Pearl. A Space-Efficient On-Line Method of Computing Quantile Estimates. *J. Algorithms*, 2(2):164–177, 1981.

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