Extended DCG notation:
A tool for applicative programming in Prolog

User Manual

1. Introduction

This manual describes a preprocessor for Prolog that adds an arbitrary number of arguments to a predicate without increasing the size of the source code. The hidden arguments are of two kinds:

(1) Accumulators, useful for results that are calculated incrementally in many predicates. An accumulator expands into two additional arguments per predicate.

(2) Passed arguments, used to pass global information to many predicates. A passed argument expands into a single additional argument per predicate.

The preprocessor has been tested under C-Prolog and Quintus Prolog. It is being used by the author in program development, and is believed to be relatively bug-free. However, it is still being refined and extended. The most recent version is available by anonymous ftp to arpa.berkeley.edu or by contacting the author. Please let me know if you find any bugs. Comments and suggestions for improvements are welcome.

2. Using the preprocessor

The preprocessor is implemented in the file `accumulator.pl`. It must be consulted or compiled before the programs that use it. In Prologs that conform to the Edinburgh standard, such as C-Prolog or Quintus Prolog, the user-defined predicate `term_expansion/2` is called when consulting or compiling each clause that is read. With this hook the use of the preprocessor is transparent.

Clauses to be expanded are of the form `(Head-->>Body)` where `Head` and `Body` are the head and body of the clause. The head is always expanded with all of its hidden arguments. Table 1 summarizes the expansion rules for body goals. In the table, `Goal` denotes any goal in a clause body, `Acc` denotes an accumulator, `Pass` denotes a passed argument, and `Arg` denotes either an accumulator or a passed argument. Hidden arguments of body goals that are not in the head have default values which can be overridden. For compatibility with DCG notation the accumulator `dcg` is available by default. If-then-else is not handled in this version.

The preprocessor assumes the existence of a database of information about the hidden parameters and the predicates to be expanded. Three relations are recognized: a declaration for each predicate, each accumulator, and each passed argument. These relations can be put at the beginning of each file (in which case their scope is the file) or stored in a separate file that is consulted first (in which case their scope is the whole program).

A short example gives a flavor of what the preprocessor does:

```
% Declare the accumulator 'castor':
acc_info(castor, _, _, _, true).

% Declare the passed argument 'pollux':
pass_info(pollux).

% Declare three predicates using these hidden arguments:
pred_info(p, 1, [castor,pollux]).
pred_info(q, 1, [castor,pollux]).
```

Table 1 — Expansion rules for the preprocessor

Body goal	Action
`{Goal}`	Don't expand any hidden arguments of `Goal`.
`Goal`	Expand all of the hidden parameters of `Goal` that are also in the head. Those hidden parameters not in the head are given default values.
`Goal:L`	If `Goal` has no hidden arguments then force the expansion of all arguments in `L` in the order given. If `Goal` has hidden arguments then expand all of them, using the contents of `L` to override the expansion. `L` is either a term of the form `Acc`, `Acc(Left,Right)`, `Pass`, `Pass(Value)`, or a list of such terms. When present, the arguments `Left`, `Right`, and `Value` override the default values of arguments not in the head.
`List:Acc`	Accumulate a list of terms in the accumulator `Acc`.
`List`	Accumulate a list of terms in the accumulator `dcg`.
`X/Arg`	Unify `X` with the left term for the accumulator or passed argument `Arg`.
`Acc/X`	Unify `X` with the right term for accumulator `Acc`.
`X/Acc/Y`	Unify `X` with the left and `Y` with the right term for the accumulator `Acc`.
`insert(X,Y):Acc`	Insert the arguments `X` and `Y` into the chain implementing the accumulator `Acc`. This is useful when the value of the accumulator changes radically because `X` and `Y` may be the arguments of an arbitrary relation.
`insert(X,Y)`	Insert the arguments `X` and `Y` into the chain implementing the accumulator `dcg`. This inserts the difference list `X-Y` into the accumulated list.

```
pred_info(r, 1, [castor,pollux]).

% The program:
p(X) -->> Y is X+1, q(Y), r(Y).
```

This example declares one accumulator, one passed argument, and three predicates using them. The program consists of a single clause. The preprocessor is used as follows: (bold-face denotes user input)

```
% cprolog
C-Prolog version 1.5
| ?- ['accumulator.pl'].
accumulator.pl consulted 9780 bytes 1.7 sec.

yes
| ?- ['example.pl'].
example.pl consulted 668 bytes 0.25 sec.

yes
| ?-
```

Now the predicate `p(X)` has been expanded. We can see what it looks like with the `listing` command:

```
| ?- listing(p).

p(X, S1, S3, P) :- Y is X+1, q(Y, S1, S2, P), r(Y, S2, S3, P).
```

(Variable names have been changed for clarity.) The arguments `S1`, `S2`, and `S3`, which implement the

accumulator `castor`, are chained together. The argument `P` implements the passed argument. It is added as an extra argument to each predicate.

In object-oriented terminology the declarations of hidden parameters correspond to classes with a single method defined for each. Declarations of predicates specify the inheritance of the predicate from multiple classes, namely each hidden parameter.

3. Declarations

3.1. Declaration of the predicates

Predicates are declared with facts of the following form:

```
pred_info(Name, Arity, List)
```

The predicate `Name/Arity` has the hidden parameters given in `List`. The parameters are added in the order given by `List` and their names must be atoms.

3.2. Declaration of the accumulators

Accumulators are declared with facts in one of two forms. The short form is:

```
acc_info(Acc, Term, Left, Right, Joiner)
```

The long form is:

```
acc_info(Acc, Term, Left, Right, Joiner, LStart, RStart)
```

In most cases the short form gives sufficient information. It declares the accumulator `Acc`, which must be an atom, along with the accumulating function, `Joiner`, and its arguments `Term`, the term to be accumulated, and `Left` & `Right`, the variables used in chaining.

The long form of `acc_info` is useful in more complex programs. It contains two additional arguments, `LStart` and `RStart`, that are used to give default starting values for an accumulator occurring in a body goal that does not occur in the head. The starting values are given to the unused accumulator to ensure that it will execute correctly even though its value is not used. Care is needed to give correct values for `LStart` and `RStart`. For DCG-like list accumulation both may remain unbound.

Two conventions are used for the two variables used in chaining depending on which direction the accumulation is done. For forward accumulation, `Left` is the input and `Right` is the output. For reverse accumulation, `Right` is the input and `Left` is the output.

To see how these declarations work, consider the following program:

```
% Example illustrating the difference between
% forward and reverse accumulation:

% Declare the accumulators:
acc_info(fwd, T, In, Out, Out=[T|In]).   % Forward accumulator.
acc_info(rev, T, Out, In, Out=[T|In]).   % Reverse accumulator.

% Declare the predicates using them:
pred_info(flist, 1, [fwd]).
pred_info(rlist, 1, [rev]).

% flist(N, [], List) creates the list [1, 2, ..., N]
flist(0) -->> [].
flist(N) -->> N>0, [N]:fwd, N1 is N-1, flist(N1).

% rlist(N, List, []) creates the list [N, ..., 2, 1]
rlist(0) -->> [].
rlist(N) -->> N>0, [N]:rev, N1 is N-1, rlist(N1).
```

This defines two accumulators `fwd` and `rev` that both accumulate lists, but in different directions. The

joiner of both accumulators is the unification `Out=[T|In]`, which adds `T` to the head of the list `In` and creates the list `Out`. In accumulator `fwd` the output `Out` is the left argument and the input `In` is the right argument. This builds the list in ascending order. Switching the arguments, as in the accumulator `rev`, builds the list in reverse. A sample execution gives these results:

```
| ?- flist(10, [], List).

List = [1,2,3,4,5,6,7,8,9,10]

yes
| ?- rlist(10, List, []).

List = [10,9,8,7,6,5,4,3,2,1]

yes
| ?-
```

If the joining function is not reversible then the accumulator can only be used in one direction. For example, the accumulator `add` with declaration:

```
acc_info(add, I, In, Out, Out is I+In).
```

It can only be used as a forward accumulator. Attempting to use it in reverse results in an error because the argument `In` of the joiner is uninstantiated. The reason for this is that the predicate `is/2` is not pure logic: it requires the expression in its right-hand side to be ground.

3.3. Declaration of the passed arguments

Passed arguments are declared as facts in one of two forms. The short form is:

```
pass_info(Pass)
```

The long form is:

```
pass_info(Pass, PStart)
```

In most cases the short form is sufficient. It declares a passed argument `Pass`, that must be an atom. The long form also contains the starting value `PStart` that is used to give a default value for a passed argument in a body goal that does not occur in the head. Most of the time this situation does not occur.

4. Tips and techniques

Usually there will be one clause of `pred_info` for each predicate in the program. If the program becomes very large, the number of clauses of `pred_info` grows accordingly and can become difficult to keep consistent. In that case it is useful to remember that a single `pred_info` clause can summarize many facts. For example, the following declaration:

```
pred_info(_, _, List).
```

gives all predicates the hidden parameters in `List`. This keeps programming simple regardless of the number of hidden parameters.