Appendix E

Extended DCG notation: A tool for applicative programming in Prolog

1. Introduction

This appendix describes a preprocessor that simplifies purely applicative programming in Prolog. The preprocessor generalizes Prolog's Definite Clause Grammar (DCG) notation to allow programming with multiple accumulators. It has been an indispensable tool in the development of the Aquarius Prolog compiler. Its use is transparent in versions of Prolog that conform to the Edinburgh standard. The preprocessor and a user manual are available by anonymous ftp to arpa.berkeley.edu.

It is desirable to program in a purely applicative style, i.e. within the pure logical subset of Prolog. In that case a predicate's meaning depends only on its definition, and not on any outside information. This has two important advantages. First, it greatly simplifies verifying correctness. Simple inspection is often sufficient. Second, since all information is passed locally, it makes the program more amenable to parallel execution. However, in practice the number of arguments of predicates written in this style is large, which makes writing and maintaining them difficult. Two ways of getting around this problem are (1) to encapsulate information in compound structures which are passed in single arguments, and (2) to use global instead of local information. Both of these techniques are commonly used in imperative languages such as C, but neither is a satisfying way to program in Prolog, for the following reasons:

- Because Prolog is a single-assignment language, modifying encapsulated information requires a time-consuming copy of the entire structure. Sophisticated optimizations could make this efficient, but compilers implementing them do not yet exist.
- Using global information destroys the advantages of programming in an applicative style, such as the ease of mathematical analysis and the suitability for parallel execution.

A third approach with neither of the above disadvantages is extending Prolog to allow an arbitrary number of arguments without increasing the size of the source code. The extended Prolog is translated into standard Prolog by a preprocessor. This report describes an extension to Prolog's Definite Clause Grammar notation that implements this idea.

2. Definite Clause Grammar (DCG) notation

DCG notation was developed as the result of research in natural language parsing and understanding [Pereira & Warren 1980]. It allows the specification of a class of attributed unification grammars with semantic actions. These grammars are strictly more powerful than context-free grammars. Prologs that conform to the Edinburgh standard [Clocksin & Mellish 1981] provide a built-in preprocessor that translates clauses written in DCG notation into standard Prolog.

An important Prolog programming technique is the accumulator [Sterling & Shapiro 1986]. The DCG notation implements a single implicit accumulator. For example, the DCG clause:

```
term(S) --> factor(A), [+], factor(B), {S is A+B}.
```

is translated internally into the Prolog clause:

```
term(S,X1,X4) :- factor(A,X1,X2), X2=[+|X3], factor(B,X3,X4), S is A+B.
```

Each predicate is given two additional arguments. Chaining together these arguments implements the accumulator.

3. Extending the DCG notation

The DCG notation is a concise and clear way to express the use of a single accumulator. However, in the development of large Prolog programs I have found it useful to carry more than one accumulator. If written explicitly, each accumulator requires two additional arguments, and these arguments must be chained together. This requires the invention of many arbitrary variable names, and the chance of introducing errors is large. Modifying or extending this code, for example to add another accumulator, is tedious.

One way to solve this problem is to extend the DCG notation. The extension described here allows for an unlimited number of named accumulators, and handles all the tedium of parameter passing. Each accumulator requires a single Prolog fact as its declaration. The bulk of the program source does not depend on the number of accumulators, so maintaining and extending it is simplified. For single accumulators the notation defaults to the standard DCG notation.

Other extensions to the DCG notation have been proposed, for example Extraposition Grammars [Pereira 1981] and Definite Clause Translation Grammars [Abramson 1984]. The motivation for these extensions is natural-language analysis, and they are not directly useful as aids in program construction.

4. An example

To illustrate the extended notation, consider the following Prolog predicate which converts infix expressions containing identifiers, integers, and addition (+) into machine code for a simple stack machine, and also calculates the size of the code:

```
expr_code(A+B, S1, S4, C1, C4) :-
     expr_code(A, S1, S2, C1, C2),
     expr_code(B, S2, S3, C2, C3),
     C3=[plus|C4],      /* Explicitly accumulate 'plus' */
     S4 is S3+1.        /* Explicitly add 1 to the size */
expr_code(I, S1, S2, C1, C2) :-
     atomic(I),
     C1=[push(I)|C2],
     S2 is S1+1.
```

This predicate has two accumulators: the machine code and its size. A sample call is `expr_code(a+3+b,0,Size,Code,[])`, which returns the result:

```
Size = 5
Code = [push(a),push(3),plus,push(b),plus]
```

With DCG notation it is possible to hide the code accumulator, although the size is still calculated explicitly:

```
expr_code(A+B, S1, S4) -->
     expr_code(A, S1, S2),
     expr_code(B, S2, S3),
     [plus],        /* Accumulate 'plus' in a hidden accumulator */
     {S4 is S3+1}.  /* Explicitly add 1 to the size */
expr_code(I, S1, S2) -->
     {atomic(I)},
     [push(I)],
     {S2 is S1+1}.
```

The extended notation hides both accumulators:

```
expr_code(A+B) -->>
     expr_code(A),
     expr_code(B),
     [plus]:code,   /* Accumulate 'plus' in the code accumulator */
     [1]:size.      /* Accumulate 1 in the size accumulator */
expr_code(I) -->>
     {atomic(I)},
     [push(I)]:code,
     [1]:size.
```

The translation of this version is identical to the original definition. The preprocessor needs the following declarations:

```
acc_info(code, T, Out, In, (Out=[T|In]))./* Accumulator declarations */
acc_info(size, T, In, Out, (Out is In+T)).

pred_info(expr_code, 1, [size,code]).    /* Predicate declaration */
```

For each accumulator this declares the accumulating function, and for each predicate this declares the name, arity (number of arguments), and accumulators it uses. The order of the `In` and `Out` arguments determines whether accumulation proceeds in the forward direction (see `size`) or in the reverse direction (see `code`). Choosing the proper direction is important if the accumulating function requires some of its arguments to be instantiated.

5. Concluding remarks

An extension to Prolog's DCG notation that implements an unlimited number of named accumulators was developed to simplify purely applicative Prolog programming. Comments and suggestions for improvements are welcome.

6. References

[Abramson 1984]

H. Abramson, ''Definite Clause Translation Grammars,'' *Proc. 1984 International Symposium on Logic Programming,* 1984, pp 233-240.

[Clocksin & Mellish 1981]

W.F. Clocksin and C.S. Mellish, ''Programming in Prolog,'' *Springer-Verlag,* 1981.

[O'Keefe 1988]

R. A. O'Keefe, ''Practical Prolog for Real Programmers,'' *Tutorial 8, Fifth International Conference Symposium on Logic Programming, Aug. 1988.*

[Pereira 1981]

F. Pereira, ''Extraposition Grammars,'' *American Journal of Computational Linguistics,* 1981, vol. 7, no. 4, pp 243-255.

[Pereira & Shieber 1981]

F. Pereira and S. Shieber, ''Prolog and Natural-Language Analysis,'' *CSLI Lecture Notes 10,* 1987.

[Pereira & Warren 1980]

F. Pereira and D.H.D. Warren, ''Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks,'' *Journal of Artificial Intelligence,* 1980, vol. 13, no. 3, pp 231-278.

[Sterling & Shapiro 1986]

L. Sterling and E. Shapiro, ''The Art of Prolog,'' *MIT Press,* 1986.