Logic Programming and Software Engineering - Implications for Software Design

Leon Sterling[1] and Ümit Yalçinalp[2]
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, OH, 44106, U.S.A.

Abstract

Logic programming is a programming paradigm with potential to contribute to software engineering. This paper is concerned with one dimension of that potential, the impact that experience with developing logic programs can have on software design. We present a logic programming perspective on programming patterns, systematic program development, design for provability, and the paradigm of meta-programming.

1. Abstractions help in developing complex software

The essential challenge of software engineering is how to build and maintain complex software. The challenge has continued unabated over many years. Computer use is increasing, and with increased computer use has come increased user knowledge, and rising expectations of software reliability and usability.

Opinions vary as to whether current software engineering practice can meet expectations. Harel (1992) sounded an optimistic tune. While acknowledging earlier cautionary writing of Brooks (1987) and Parnas (1985) that there is no `silver bullet' for the problems software developers face, Harel claims that computer science is advancing and software production can become more reliable.

The history of computer science has shown that progress in software development has come through better abstractions. Logic programming (Kowalski, 1979) is an abstraction introduced in the 1970’s. It has taken time for its value to be appreciated in software engineering. It is the purpose of this paper, and more generally this special issue, to argue that logic programming has something to offer.

The key abstraction introduced in logic programming is the logical variable and the use of unification as a uniform means of computation. Unification abstracts away many data manipulation details, making programs more concise, and easy to read and write. This will be explicitly pointed out in later sections. Non-determinism is also present, which abstracts away implementation details of specific search techniques in some applications.

This paper discusses in detail one dimension of the potential of logic programming to contribute to software engineering - the impact that experience with developing logic programs can have on software design. The key innovation of the paper is explicitly identifying patterns that have emerged within logic programming, and pointing out their relationship to the current interest in programming patterns (Gamma et al., 1995). Another

[1] Current Address: Department of Computer Science, University of Melbourne, Parkville, Vic. 3052, Australia, email: leon@cs.mu.oz.au
[2] Current Address: Sybase Corporation, CA, U SA, email:umit@sybase.com

discussion of the potential of logic programming to contribute to software engineering can be found in (Ciancarini and Levi, 1992).

Patterns in logic programming constitute reusable components and are sketched in Section 2. The patterns are easy to recognize in logic programs due to the high level of abstraction afforded by unification and the logical variable. We will discuss this more in the context of Program 4 for finding a path between nodes in a graph. In Section 3 we show how patterns can form the basis for a systematic program development method which aids maintainability.

Logic programming draws its source from logic. Indeed the theme for this stream of papers is logic engineering. Conceived appropriately, using logic programs is a formal method. A strength of formal methods is the ability to prove programs correct and to construct programs that are provably correct. How the program patterns can guide a correctness proof is discussed in Section 4, along with implications towards design of programs for provability.

Section 5 presents the paradigm of meta-programming. We believe the essence of meta-programming is building an abstraction of a language and developing an interpreter for the abstraction. We explain our non-standard view of meta-programming and argue that logic programming is especially amenable to introducing and exploiting abstractions. The theme of powerful abstraction explicit here is implicit in the paper.

These are not the only issues relevant to software design from experience in program development with logic programming languages. The specific areas were chosen to give a flavor of what logic programming can contribute, and because they reflect our recent research. Other interesting issues are related to module systems (Bugliesi et al., 1994), and the interactions between types and modes reflected, for example, in the development of the language Mercury (Somogyi et al., 1996).

Logic programming has contributed to software engineering more broadly. Other papers in this special issue speak for themselves, but we explicitly mention the following:

- Process modeling
- Rapid prototyping and exploratory programming
- Program transformation
- Formal methods

We conclude this section with the observation that there may be growing acceptance of logic programming within software engineering. There has certainly been increased visibility in the form of this special issue, and the special issue of International Journal of Software Engineering and Knowledge Engineering (Ciancarini and Sterling, 1996) arising out of the workshop on Application of Logic Programming to Software Engineering following the Eleventh International Conference on Logic Programming held in Italy in 1994. There has also emerged a stream of applications presented in the Prolog 1000 database and presented at the series of international conferences on practical applications of Prolog and Constraint Logic Programming in 1992, 1994, 1995, and 1996 in London and Paris.

2. Program patterns

Patterns have been widely acknowledged as being important in crafting complex systems in areas such as architecture and machine design. Recent attention has focussed on design patterns for software engineering, particularly associated with the widespread interest in object-oriented programming. To some extent, patterns have been a theme throughout the evolution of programming languages. Subroutines and macros can certainly be viewed as patterns, and were introduced to allow reusability within a single piece of software. Current emphasis is on patterns that can be reused between software systems.

The abstraction level of logic programming languages makes certain patterns easy to see and reuse. Unification is a key factor. Taking a pattern-directed view can make logic programs easy to build. In this section, we discuss patterns that have emerged, primarily in Prolog, and comment at the end on the relationship to other languages. .

2.1 Skeletons and shells

We loosely identify two classes of programming patterns for logic programming: skeletons and shells discussed in this subsection, and techniques discussed in the next subsection (Sterling and Kirschenbaum, 1993), (Sterling and Shapiro, 1994).

The first class of patterns are reusable programs which we have called *skeletons* . Skeletons constitute an essential control flow of a program. Useful skeletons are

- data structure traversers,
- grammars, and
- interpreters.

The most common data structure for logic programs is the list, and many programs are based on skeletons for traversing lists. These have been described elsewhere, notably in (Sterling and Kirschenbaum, 1993) and (Sterling and Shapiro, 1994). A running example we will use in this paper is a program for graph traversal. Program 1 contains the program `connected`/2 which is a skeleton for traversing a graph specified by a collection of `edge`/2 facts. We use a simple example here for pedagogic reasons. The program (pattern) is the transitive closure of the `edge` relation, and checks connection via depth-first search inherited from Prolog’s computation model.

```
connected(X,Y) ← edge(X,Y).
connected(X,Y) ← edge(X,Z), connected(Z,Y).
```

Program 1 A skeleton for traversing a graph

Program 2 contains a fragment of a definite clause grammar (DCG) for parsing a Pascal-like programming language. The first rule says that a statement is an identifier, followed by :=, followed by an expression. DCGs are in fact syntactic sugar for Prolog, and most Prolog systems translate DCGs directly into Prolog.

```
statement → identifier, [:=], expression.
statement → [if], test, [then], statement, [else], statement.
statement → [while], test, [do], statement.
```

Program 2 Fragment of a grammar for parsing a simple programming language

Skeletons are complete logic programs which run even if all they do is check types. It is also useful to think of incomplete programs where the details are to be filled in. A *shell* is a skeleton where not all predicates in the skeleton are fully developed. Loosely,

Shell = Code + comments + specifications for the missing code

A shell is refined by including the code for the missing goals in the body, or by adding and deleting new goals in the body of a clause.

An example of a shell is the game playing structure presented in Figure 1. Note that many other predicates need to be filled in for the shell to be useful. But the essential logic has been captured. The shell is discussed in detail in Chapter 20 of 'The Art of Prolog' (Sterling and Shapiro, 1994), and it is refined and reused for games of nim and kalah in Chapter 21 of that text. A colleague, Andrew Davison, told me this was a good starting point for a game playing tutorial he developed in C.

play(Game) ← Play game with name Game

Predicates that need to be filled in are:
initialize/3 which initializes the game board, and the person who is to play next;
display_game/2 which displays the current position of the game from the perspective of the next player;
game_over/3 which checks if the game is over;
announce/1 which announces the result;
choose_move/3 which chooses a move;
move/3 which makes the move updating the board;
next_player which gives the succession of moves.

```
play(Game) ←
     initialize(Game,Position,Player),
     display_game(Position,Player),
     play(Position,Player,Result).

play(Position,Player,Result) ←
     game_over(Position,Player,Result) ->
          announce(Result)
     ;  choose_move(Position,Player,Move),
        move(Move,Position,Position1),
        display_game(Position1,Player),
        next_player(Player,Player1),
        play(Position1,Player1,Result).
```

Figure 1 Shell for playing games

2.2 Techniques and enhancements

Techniques are the second class of patterns. They capture basic Prolog programming practices, such as building a data structure or performing calculations in recursive code. Unlike skeletons, techniques are not programs but can be conceived as a family of operations that can be applied to a skeleton to produce a program.

Informally, a programming technique interleaves some additional computation around the control flow of a skeleton program. The additional computation might calculate a value or produce a side effect such as screen output. Syntactically, techniques may rename predicates, add arguments to predicates, add goals to clauses, and/or add clauses to programs. Applying a technique to a skeleton yields an *enhancement.*. Enhancements preserve the basic computational behavior of the skeleton.

The *calculate* and *build* techniques both compute something (a value or a data structure) while following the control flow of the skeleton. An extra argument is added to the defining predicate in the skeleton, and an extra goal is added to the body of each recursive clause. In the case of the *calculate* technique, the added goal is an arithmetic calculation; in the case of the *build* technique, the goal builds a data structure. In both cases, the added goal relates the extra argument in the head of the clause to the extra argument(s) in the body of the clause.

Examples of the *calculate* and *build* techniques applied to the program for `connected` are given as the programs `count` and `path` below respectively in Programs 3a and 4.

```
count(X,Y,2) ← edge(X,Y).
count(X,Y,N) ← edge(X,Z), count(Z,Y,N1), N is N1+1.
```

Program 3a An enhancement of the graph traverser that counts nodes

```
count(X,Y,N) ← count_ac(X,Y,2,N).

count_ac(X,Y,N,N) ← edge(X,Y).
count_ac(X,Y,Ac,N) ←
     edge(X,Z), Ac1 is Ac+1, count_ac(Z,Y,Ac1,N).
```

Program 3b An enhancement of the graph traverser that counts nodes using accumulate-calculate

```
path(X,Y,[X,Y]) ← edge(X,Y).
path(X,Y,[X|P]) ← edge(X,Z), path(Z,Y,P).
```

Program 4 An enhancement of the graph traverser that builds a path

Let us consider how Program 4 for `path` illustrates the power of the logic variable and unification. Consider the query `path(a,b,P)?` asking for a path between nodes a and b. Unification with the head of the recursive clause for `path` will allocate a list for the path, set the head of the list to be a, in short manage all the data handling. Program 4 is multiple use. A query `path(a,b,[a,c,b])?` will check whether the path [a,c,b] joins a and b. Unification handles the list comparison, and the logic variable has specified that the first node and the head of the list must be the same.

The accumulator technique adds two arguments to the defining predicate in the skeleton to allow for global variables in a program. The first argument is used to record the current value of the variable in question and the second contains the final result of the computation. The base case relates the input and output arguments, often via unification.

Analogous to *calculate* and *build*, there are two types of accumulator technique: *accumulate-calculate* and *accumulate-build*. An example of the accumulate-calculate technique is given in Program 3a. One difference between *calculate* and *accumulate-calculate* is in the need to add an auxiliary predicate. Another is that goals and initial values need to be placed differently. There is an analogous *accumulate-build* technique. The reader can rewrite Program 4 to use accumulate-build as an exercise.

Other examples of useful techniques are adding a depth bound to a computation, and adding an accumulator to keep track of previously visited nodes in a search application. This technique is applied to Program 1 to yield Program 5.

```
connected(X,Y) ← connected_enh(X,Y,[X]).

connected_enh(X,Y,Visited) ← edge(X,Y).
connected_enh(X,Y,Visited) ←
        edge(X,Z),
        not member(Z,Visited),
        connected_enh(Z,Y,[Z|Visited]).
```

Program 5 An enhancement keeping track of nodes visited previously

Program 6 is the result of applying the *build* technique to Program 2 to generate a parse tree. We give another example of the build technique for two reasons. First, to emphasize that techniques are patterns. The operation of creating Program 4 from Program 1 should be seen as the same pattern as creating Program 6 from Program 2. Second, we give a forward pointer for the discussion of meta-programming in Section 5. DCGs are an example of a meta-linguistic abstraction. The patterns of skeletons and techniques are ideal for meta-programming.

```
statement(assign(Id,E)) →
        identifier(Id), [:=], expression(E).
statement(if(Test,T,E)) →
  [if], test(Test), [then], statement(T), [else], statement(E).
statement(while(Test,S)) →
        [while], test(Test), [do], statement(S).
```

Program 6 Fragment of a grammar plus parse tree

2.3 Composition

Two enhancements of the same skeleton share computational behavior. They can be combined into a single program which combines the functionality of each separate enhancement. Techniques can be developed independently and subsequently combined automatically. The (syntactic) operation for combining enhancements is called *composition*. This is similar in intent to function composition where separate functionalities are combined into a single function.

Program 7 shows the result of composition of Programs 4 and 3a, `path` and `count`. Note that the operation is syntactic. The extra arguments in `path_count` are copied verbatim from their respective programs, as are the extra goals in the recursive clause. The algorithm is described in 'The Art of Prolog' and Prolog code given for performing composition.

```
path_count(X,Y,[X,Y],2) ← edge(X,Y).
path_count(X,Y,[X|P],N) ←
        edge(X,Z), path_count(Z,Y,P,N1), N is N1+1.
```

Program 7 Graph traversal building path and counting nodes

One application of composition is to merge the behavior of 'parallel loops'. This can be done by program transformation using fold/unfold operations. However, due to the non-determinism of Prolog, composition is a more general operation. It reflects a program construction approach to combining programs which is natural for programmers.

2.4 Other LP work

The skeletons and techniques presented in this paper are all Prolog examples. Skeletons and techniques can be as easily identified for other logic programming languages, as discussed in Kirschenbaum, Michaylov and Sterling (1996). They claim that programming patterns should be identified when a (logic-based) language is first used, in order to encourage systematic, effective program development. This learning approach should be stressed during teaching. They show that the skeletons and techniques for Prolog can be extended to constraint logic programming languages, notably CLP(R) (Jaffar et al., 1992), concurrent logic programming languages such as Flat Concurrent Prolog (Shapiro, 1987) and Strand (Foster and Taylor, 1989), and higher order logic program languages, in particular λ-Prolog (Nadathur and Miller, 1988). Applying these ideas to functional programming languages is currently being studied.

Recently, Gegg-Harrison (1995) and Naish (1996) presented skeletons and techniques in terms of higher order predicates. The approach has some elegant predictive power, but is probably only accessible to more advanced students.

There have been attempts to characterize logic programming patterns using schemas. This paper shares intent with Gegg-Harrison (1991), but is both simpler and easier to generalize. The skeletons are simpler than schemata because they are not overly general. Students (and programmers) think in terms of specific examples not schemas, which we emphasize by dealing with real but skeletal programs, rather than second-order predicates. Techniques are easily adapted to other skeletons, which is not possible in his schemas.

O'Keefe's text (1990) also discusses Prolog programming patterns in terms of schemas, albeit different ones to Gegg-Harrison. Again our preference is for concrete programs. Fuchs and colleagues also present schemas for program transformation (Fuchs and Fromherz, 1991).

2.5 Other paradigms

The design patterns espoused in the book by (Gamma et al., 1995) are closer to skeletons than techniques, especially in their higher order characterization. In particular, the template pattern described in the book is reminiscent of a shell as discussed in Section 2.1. The strategy pattern is also similar to a shell. Our sense is that the logic programming perspective does not match exactly with classes and instances, but is similar in spirit, especially if a higher order view is taken.

Characterizing patterns via higher order functions as mentioned above is similar to work on higher order programming in functional languages. Techniques such as foldl and foldr can achieve the same effect as application of programming techniques. Loop merging in functional languages achieves the same effect as composition, though non-determinism is an added issue in logic programming.

Research has also been performed on merging similar behaviors in imperative languages. Reps and colleagues use slicing to combine effects (Reps, 1990). Reps' work is similar in intent to composition. In our opinion, the patterns are more elegant in the logic programming context.

3. Systematic development and maintenance

3.1 Stepwise enhancement

The identification of skeletons and techniques arose from investigation into what might constitute a standard program development methodology for Prolog. Despite attractive features, Prolog has not been widely adopted for software engineering. A possible factor is that standard development practices have not been adapted to Prolog. The method of *stepwise enhancement*, an adaptation of stepwise refinement, addresses this weakness. It permits a systematic construction of Prolog programs, while exploiting Prolog's high-level features.

Stepwise enhancement consists of three steps:

1. Identify the skeleton program constituting the control flow of the program.
2. Create enhancements using standard programming techniques.
3. Compose the separate enhancements to give the final program.

Developing a program is typically straightforward once the skeleton is decided. Knowing what skeleton to use is less straightforward and must be learned by experience, which is true for any design task. However, by splitting up the program development into three steps, the design process is simplified and given structure.

A tutorial example of using stepwise enhancement to develop a simple program is given in Chapter 13 of (Sterling and Shapiro, 1994). A more elaborate example developing an expert system shell is given in Chapter 17 of the text. Other expert system examples can be found in (Yalçinalp, 1991). A detailed example of code developed using stepwise enhancement is a Prolog tracer, reported in (Lakhotia, Sterling and Bojantchev, 1995).

Stepwise enhancement has also been useful for developing programs for software testing. A skeleton parser for Pascal (of which Program 2 is a part) was instrumented to give def-use chains and block numbering. The experience is reported in (Sterling et al. 1992). A novice Prolog programmer adapted the program for instrumenting data coverage testing to work on C code rather than Pascal. The programmer was successful due to the structuring of the problem and by being able to adapt patterns.

3.2 Enhancement Structures

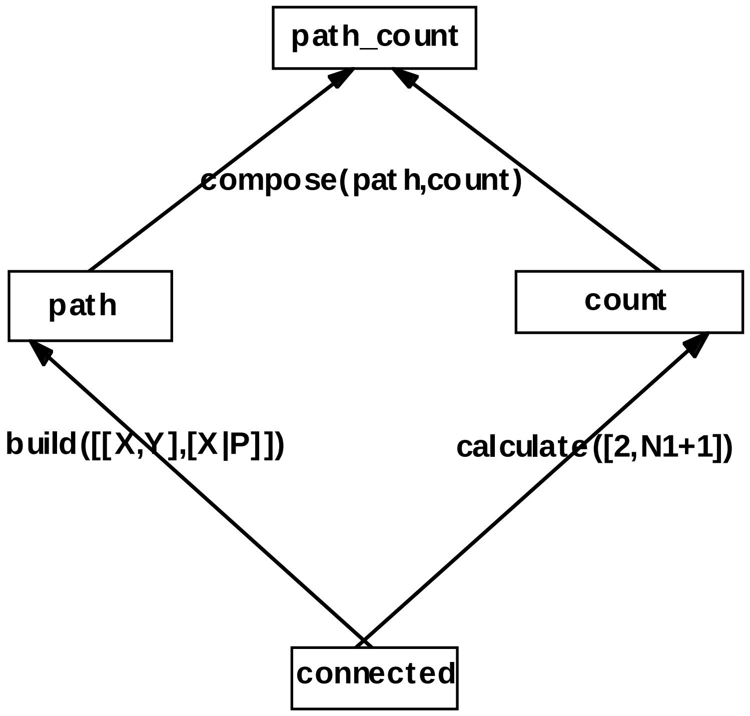

Figure 2: Example enhancement structure

Enhancement structures are a graphical way of presenting the relationship between programs which emphasize patterns. Figure 2 shows an enhancement structure for our running example of `path_count`. Nodes of the enhancement structure are programs, typically skeletons and their enhancements. Arcs represent the application of a programming technique and are labelled by a parameterized representation of the technique applied. The name is the technique and the arguments are what needs to be filled in to complete the enhancement.

Enhancement structures can help guide a proof of correctness for programs developed via stepwise enhancement. This will be sketched in Section 4.2.

3.3 Maintaining enhanced programs

Consider changing a program developed using stepwise enhancement. If the only change is to a single technique, all the programmer need do is create a new enhancement embodying the changed technique. The sequence of enhancements and compositions can which built the original program can be replayed to give the new final program. The modifications and replay could readily be moderated through a user interface operating directly on the enhancement structure.

Facilitating program modification by replaying operations that have not changed is certainly viable. A prototype system was demonstrated at a logic programming workshop in 1991 which replayed a series of programming enhancements. The system was based on a poorer version of techniques proposed by Lakhotia (1989).

3.4 Support tools

The PT[3] environment supports logic program development via stepwise enhancement. Its design and prototype implementation in C++ is described in (Sterling and Sen, 1992). The environment can be tailored to either Prolog or CLP(R). Via a mouse-driven interface, a user can create extensions by applying techniques to skeletons. The user is prompted to fill in values for arguments and supply extra goals with the interface stepping through the code to the appropriate places.

The prototype currently facilitates construction of programs concerned with recursive data structure traversal. The environment allows the application of four techniques to skeletons: *calculate*, *build*, *accumulate-calculate* and *accumulate-build*. Seven skeletons are pre-defined with the environment, and the user can add others.

Composition of extensions created in the environment is supported, and limited error checking has been incorporated. Implicit in PT's interface is a representation of techniques. Each technique is parameterized for each skeleton to know where to add values. For example, the *build* technique adds an extra argument to each essential predicate in the skeleton. A new goal is given for each rule where the extra argument in the head is determined by the arguments in the body.

The PT programming environment supports simple program maintenance. The simplicity of program construction in the environment carries over to small modifications.

4. Proving Prolog programs correct

4.1 Specifications of Prolog programs

We do not advocate using first-order logic as a specification language. Still it is necessary to have a specification, that is a document which explains the behavior of a program sufficiently so that the program can be used without the code having to be read. We believe that a specification should be the primary form of documentation and be given for each procedure in a program.

A suggested form for a specification of a Prolog program is given in Figure 3. It consists of a procedure declaration, effectively giving the name and arity of the predicate, a series of type declarations about the arguments, a relation scheme, and other important information such as modes of use of the predicate and multiplicities of solutions in each mode of use. Most relevant here is the relation scheme.

The relation scheme is a precise statement in English which explains the relation computed by the program. It should be stressed that relation schemes must be precise statements. We believe that proving properties of programs should proceed in the way of mathematics where proofs are given by precise statements in an informal language. Our view of specifications is heavily influenced by Deville (1990).

---

[3] An unimaginative acronym for Prolog Tool

```
procedure  p(T1, T2, ..., Tn)

        Types: T1: type1
               T2: type2
                .     .
                .     .
               Tn: typen

        Relation scheme:

        Modes of use:

        Multiplicities of solution:
```

Figure 3: Template for a specification

4.2 A sample proof of correctness

We sketch a sample proof that a Prolog program is correct with respect to its specification. The proof heavily depends on the way the program was derived. The derivation is described by the concept of an enhancement structure presented in Section 3.2.

The enhancement structure can help guide a proof of correctness for programs developed via stepwise enhancement. The proof suggests leveraging the structure of the program development to help the final program. For example, with the program for `path_count` given in the enhancement structure of Figure 2, the following steps will constitute the proof.

1. Establish that the `connected` program is correct using a straightforward induction argument.
2. Prove that `path` is correct relative to `connected` using the correctness of the programming technique.
3. Prove that `count` is correct relative to `connected` using the correctness of the programming technique.
4. Since `path_count` is the composition of `count` and `path`, and composition preserves correctness, `path_count` is correct.

A sample proof that `count` is correct relative to `connected` is sketched in the appendix, where an appropriate specification is given. The proof depends on the Computation Extension Theorem proved in (Kirschenbaum et al., 1993).

4.3 Design for provability

Structuring proofs of correctness to correspond to how the program was developed can be extended for more elaborate examples. This leads to the notion of *design for provability*. The way the program is built is shaped by the intention to prove it correct.

The first author has been gathering anecdotal evidence about the efficacy of design for provability. Together with Liming Cao, Leon Sterling modified a program for partitioning a number so that it could be proved correct. Thinking about having to prove the program correct greatly improved the program. We also were forced to clarify the main data structure that would be needed in the design.

Proofs of correctness have been sketched for several other examples. The largest are an object-oriented database language, and a translator from Z to Prolog (Sterling et al., 1996). In both these cases, thinking of the need to prove the program correct improved the code.

5. Meta-Programming

... the establishment of new descriptive languages ... is particularly important in computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators.

(Abelson & Sussman, 1985)

5.1 What is Meta-Programming?

The paradigm of meta-programming has emerged as a mature entity within logic programming since the pioneering work of Bowen and Kowalski (1982). Prolog, as a logic programming language, supports the paradigm. Manipulating programs is easy, and rapid prototyping of new meta-linguistic abstractions is a normal style of development.

It is worth exploring definitions of meta-programming. The commonly accepted definition of meta-programming, given for example in (Sterling and Shapiro, 1994) and in the foreword to (Abramson and Rogers, 1989) is ‘the writing of programs that treat other programs as data.’ This definition is an accurate description, however, it does not convey the essence of program development using meta-programming.

We believe that meta-programming is best viewed as a (relatively) new paradigm for programming complex systems that has emerged in both functional and logic programming. A *paradigm* for our purposes, as discussed in (Floyd, 1987), is an approach to writing programs. According to Floyd, a paradigm is a collection of methods and/or techniques to facilitate the construction of certain classes of programs. Examples of paradigms are structured programming, rule-based programming, and dynamic programming. Each of them are effective for a class of programming problems.

In our opinion, the essence of meta-programming is building a language which abstracts and interprets other languages. The purpose of meta-programming is to facilitate manipulation of object programs, either by syntactically manipulating them or executing them. This leads to a different definition of meta-programming.

‘Meta-programming is building an abstraction of an object language and developing an interpreter for the abstraction.’

5.2 Applications

Logic programming and Prolog as its most mature language have played a role in application development. Prolog is particularly appropriate for manipulating symbols. Where the symbols are programs, we have the literal sense of meta-programming. Applications which are oriented to symbol manipulation such as parsing and compiling are most suitable. An excellent case for the use of Prolog for parsing and compiling has been made by Cohen and Hickey (1992). Prolog has also made some impact in software testing. Here the specification of an abstract data type is manipulated to generate a computationally feasible set of test cases (Dauchy and Marre, 1991). Again the power to manipulate symbols at a high level is very important.

In previous research, we have shown the power of language abstraction for expert systems, both for tools and specific applications. The case is eloquently made in (Yalçinalp, 1991) of how to use Prolog for knowledge-based systems via a meta-programming approach.

Meta-programming is good for compiling one language to another as well as providing a means of abstraction for defining a language and its interpretation. There are two examples from developments at Quintus. The first extends and formulates a language and paradigm on top of Prolog. Schachte and Saab (1994) describe the Quintus objects package, which was possible to build precisely because of the meta-programming approach. We believe this is a model for software development. The second example is abstract SQL compilation to native SQL for a relational database layer. After developing the abstract SQL, there was an interface to all the major databases. Adding them was not a real problem except linking with specific database libraries.

6. Conclusions

Taking a pattern view of programming is helpful. There is no doubt that skeletons and techniques can help in teaching the effective use of logic programming languages. Within the classroom, emphasizing patterns has been valuable for teaching effective Prolog programming. Students are able to follow more complicated Prolog programs and the quality of code in student projects has increased. Graduate students find this approach useful for explaining code to others, and in the cases of meta-interpreters cited earlier complicated programs were more easily developed.

Finally, we reiterate our belief that abstraction is key to future development of complex software systems. It makes it easy to identify programming patterns so programming solutions can be re-used. Also more literally, domain-specific abstractions can be rapidly prototyped and developed using a meta-programming approach. It will be a cornerstone in any applications we are involved with.

Acknowledgments

This work was supported by the National Science Foundation under grants CCR-9000387 and CCR-93-03484. It was greatly influenced by active discussions with the ProSE group at Case Western Reserve University, especially Marc Kirschenbaum and Ashish Jain. The first author acknowledges support of his current location, the Department of Computer Science of the University of Melbourne. The second author acknowledges colleagues at Quintus.

References

Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

Abramson H. and Rogers, M. (eds.), *Meta-Programming in Logic Programming*, MIT Press, 1989.

Bowen, K.A. and Kowalski, R. Amalgamating Language and Metalanguage in Logic Programming, in *Logic Programming,* (eds. Clark, K.L. and Tarnlund, S.-A.), Academic Press, 1982.

Brooks, F.P. Jr. No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer*, **20**(4), pp. 10-19, 1987

Bugliesi, M., Lamma, E., and Mello, P. Modularity in Logic Programming, *J. Logic Programming*, **19/20**, pp. 443-502, 1994

Ciancarini, P. and Levi, G. What is Logic Programming good for in Software Engineering? in *Advances in Software Engineering and Knowledge Engineering*, Ambriola, V. and Tortora, G. (eds.), pp. 109-134, World Scientific Press, 1992

Ciancarini, P. and Sterling, L.S. (eds.) Applications of Logic Programming in Software Engineering, *International Journal on Software Engineering and Knowledge Engineering*, **6**(1), 1996

Dauchy, P. and Marre, B. Test Data Selection from Algebraic Specifications, Proc. 3rd European Soft. Eng. Conf., LNCS **550**, pp. 80-100, Springer-Verlag, 1991

Deville, Y. *Logic Programming: Systematic Program Development*, Addison Wesley, 1990

Floyd, R.W. The Paradigm of Programming, in *ACM Turing Award Lectures - The First Twenty Years - 1966-1985*, pp. 131-142, ACM Press, 1987.

Foster, I. and Taylor, S. *Strand: New Concepts in Parallel Processing,* Prentice Hall, 1989

Fuchs, N. and Fromherz, M. Schema-based Transformations of Logic Programs, Proc. 5th International Workshop on Logic Program Synthesis and Transformation, Proietti, M. (ed.), pp. 111-125, Springer-Verlag, 1991.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*, Addison-Wesley, 1995

Gegg-Harrison, T. Learning Prolog in a Schema-Based Environment, *Instructional Science,* **20**:173-192, 1991.

Gegg-Harrison, T. Representing Logic Program Schemata in λProlog, Proc. 12th International Logic Programming Conference (ed. L. Sterling), pp. 467-481, MIT Press, 1995

Harel, D. Biting the Silver Bullet: Towards a Brighter Future for System Development, *IEEE Computer*, 1992

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. The CLP($\mathcal{R}$) language and system, *ACM Trans. on Programming Languages and Systems*, **14**(3), pp. 339-395, 1992

Kirschenbaum, M., Sterling, L.S. and Jain, A. Relating Logic Programs via Program Maps, Annals of Mathematics and Artificial Intelligence, **8**, pp. 229-245, 1993

Kirschenbaum, M., Michaylov, S. and Sterling, L.S. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages, Proc. 19th Australian Computer Science Conference, Melbourne, 1996

Kowalski, R. *Logic for Problem Solving*, Elsevier-North Holland, 1979.

Lakhotia, A. Incorporating Programming Techniques into Prolog Programs, in *Proc. 1989 North American Conference on Logic Programming*, (eds. Lusk, E. and Overbeek, R.), pp. 426-440, MIT Press, 1989

Lakhotia, A., Sterling, L. and Bojantchev, D. Development of a Prolog Tracer by Stepwise Enhancement, Proc. Third International Conference on Practical Applications of Prolog, Paris, pp. 371-393, 1995

Naish, L. Higher Order Logic Programming in Prolog, Draft Computer Science Technical Report, University of Melbourne, 1996.

Nadathur, G. and Miller, D. An overview of λ-Prolog, in Proc. 5th International Conference and Symposium on Logic Programming, Kowalski, R. and Bowen, K. (eds.), pp. 810-827, MIT Press, 1988

O'Keefe, R. *The Craft of Prolog*, MIT Press, 1990.

Parnas, D. Software Aspects of Strategic Defense Systems, Comm. ACM, **28**, pp. 1326-1335, 1985

Schachte, P. and Saab, G. Efficient Object-Oriented Programming in Prolog, Proc. Second International Conf. on Practical Applications of Prolog, London, pp. 471-496, 1994

Shapiro, E. (ed.) Concurrent Prolog, MIT Press, 1987

Somogyi, Z., Henderson, F., and Conway, T. The execution algorithm of Mercury: an efficient purely declarative logic programming language, *J. Logic Programming*, 1996 (to appear)

Sterling, L., Harous, S., Kirschenbaum, M., Leis, B. and White, L. Developing Software Testing Programs Using Stepwise Enhancement Proc. Intl. Conf. Software Engineering, Melbourne, May, 1992

Sterling, L. and Kirschenbaum, M. Applying Techniques to Skeletons, in *Constructing Logic Programs*, (ed. J.M. Jacquet), pp. 127-140, Wiley, 1993.

Sterling, L., Ciancarini, P. and Turnidge, T. On the Animation of 'non executable' Specifications by Prolog, *International Journal on Software Engineering and Knowledge Engineering*, **6**(1), 1996

Sterling, L.S. and Sitt Sen, Chok A Tool to Support Stepwise Enhancement in Prolog, Workshop on Logic Programming Environments, pp. 21-26, Vancouver, October, 1993

Sterling, L.S. and Shapiro, E.Y. *The Art of Prolog*, 2nd edition, MIT Press, 1994.

Yalçinalp, L.Ü. Meta-Programming for Knowledge-Based Systems in Prolog, Ph.D. Thesis, Case Western Reserve University, 1991.

Appendix

Specification for `count(X,Y,N)`

Types: X, Y: constants (representing nodes of a graph);
N: A non-negative integer

Relation scheme: N is the number of nodes contained in a path connecting nodes X and Y in a graph.
N.B. The number of nodes in the path is one more than the number of edges in the path,

Modes of use: X, Y are inputs, and N is an output;
X,Y,N are all inputs;

Multiplicities of solution: The predicate fails if the nodes are not connected.
If there are k distinct paths, then there are k solutions.

Proof of relative correctness of `count/3`

Lemma: The length N computed by `count` for a goal G = `count(X,Y,N)` is one plus the number of times, S, that an `edge/2` goal is selected in the SLD-refutation for G.

Proof: By induction on S.

Base case: S = 1. The only refutation, i.e. sequence of quadruples (A_k,σ_k,C_k,R_k) ending in the empty resolvent ,where `edge` is selected once is

```
(count(X,Y,N), {X1=X,Y1=Y,N=2}, count(X1,Y1,2):-edge(X1,Y1),
                                          edge(X,Y))
(edge(X,Y), {X=X2,Y=Y2}, (edge(X2,Y2)), T)
```

and hence N equals 2.

Inductive case: Assume the lemma is true for when `edge/2` is selected k times. The value for k+1 is at least two, and thus in any refutation the recursive clause must be chosen. A refutation for G = `count(X,Y,N)` starts with the sequence

```
 (count(X,Y,N), {X1=X,Y1=Y,N=N1},
    count(X1,Y1,N1):-edge(X1,Z1),count(Z1,Y1,N11),N1 is N11+1,
        (edge(X,Z1), count(Z1,Y1,N11), N is N11+1))
```

By induction the refutation for `count(Z1,Y1,N11)` selects an `edge/2` fact k times, and N11 equals k+1.

In the refutation for G, `edge/2` is selected one more time and N equals k+2

Completeness: Let G = `count(X,Y,N)` be in the intended meaning for count. This means that N-1 edges connect the node X to the node Y. So it must be true that G' = `connected(X,Y)` is in the intended meaning of `connected`. Since `connected` is correct, there exists a refutation for G' which lifts to one for `count(X,Y,N')`, by the Computation Extension Theorem of (Kirschenbaum et al. 1993). Furthermore, the refutation chooses an `edge/2` goal N-1 times, since the additional goals are only arithmetic calculations. That the lifted refutation produces the correct value for N' follows immediately from the lemma.

Correctness: Suppose G = `count(X,Y,N)` is in the meaning of `count`, i.e. there is a refutation for G. By the lemma, there exists a refutation of G where N is one plus the number of times `edge/2` is selected. Interpreting this for the graph, N is the number of nodes contained in a path connecting nodes X and Y, and thus G is in the intended meaning of `count`.