

# Representing Logic Program Schemata in $\lambda$ Prolog

Timothy S. Gegg-Harrison

Department of Computer Science  
Winona State University  
Winona, MN 55987, USA  
tsg@vax2.winona.msus.edu

**Abstract.** Program schemata and programming techniques provide a mechanism for representing the essential characteristics of logic programs. By abstracting out common recursive control flow patterns, program schemata capture large classes of logic programs. Programming techniques represent common program components. By instantiating portions of program schemata with programming techniques, it is possible to generate arbitrary logic programs. In order to represent program schemata and programming techniques for any programming language, it is desirable to use a higher-order programming language as the representation language.  $\lambda$ Prolog is a higher-order logic programming language that extends Prolog by incorporating higher-order unification and  $\lambda$ -terms, making it an ideal logic programming language for representing logic program schemata and programming techniques. Because  $\lambda$ Prolog program schemata and programming techniques can be represented in  $\lambda$ Prolog, there is no need for the creation of an abstract meta-language in order to define and classify logic program schemata and programming techniques.

## 1 Introduction

The strength of logic programming languages is their duality of semantics. The declarative semantics of a logic programming language is based on logic, while the procedural semantics of a logic programming language is based on execution mechanisms (e.g., SLD-resolution + computation rule + search rule). Because the declarative and procedural semantics can be shown to be equivalent (at least for some classes of programs), it is possible for logic programmers to think less in terms of what processes the computer must go through and much more in terms of the logic of the problem itself and its possible solution. Logic programming promotes thinking about *what* the problem is rather than *how* to solve the problem. Problem solving becomes a process that is independent of the particular machine. Programs become easier to write and easier to debug because their structures more closely resembles the problem that is being solved rather some representation of the procedure required to solve it.

Since logic programming languages embody a declarative programming style and their syntax is extremely simple, it would seem that they should be easy to learn and easy to use. However, logic programming languages require their programmers to define

their programs recursively. Recursion is a very difficult concept at first, but once it is learned it becomes a very straightforward (and natural) problem solving technique. The trick is finding a method for representing common recursive control flow patterns. The solution is program schemata and programming techniques. Program schemata enable the creation of *conditional recursion* (i.e., a structured recursive equivalent to FOR and WHILE loops). Conditional iteration imposes structure on iterative programming languages, abstracting the essence of conditional repetition. Conditional recursion can serve the same role for logic programming languages.

Program schemata have proven useful in teaching recursive Prolog programming to novices [8], debugging Prolog programs [11], transforming Prolog programs [6], and synthesizing Prolog programs [5]. Furthermore, using program schemata to teach programming facilitates instruction that is tailored to the student's capabilities [9]. Program schemata enable improved instruction for novice programmers while at the same time promoting a structured programming style and the acquisition of abstract problem solving skills. In addition to being helpful for novice programmers, program schemata are essential to expert programmers. The key difference between experts and novices is not the size of their memory span, but rather their ability to chunk information together into meaningful units. Schemata provide a method of organizing meaningful information about complex domains. Experts have more and better problem schemata than novices. Novice programmers tend to categorize problems based on surface syntax-based features of the problem statement, while experts categorized problems with respect to a more abstract hierarchical organization of algorithms [1,16]. Thus, program schemata are essential to expert programmers.

By abstracting out common recursive control flow patterns, program schemata capture large classes of logic programs. Programming techniques represent common program components. By instantiating portions of program schemata with programming techniques, it is possible to generate arbitrary logic programs. In order to represent program schemata and programming techniques for any programming language, it is desirable to use a higher-order programming language as the representation language. Functional programming languages (e.g., Lisp, ML, Miranda, etc.) support higher-order functions. Most logic programming languages, on the other hand, do not have full support of higher-order predicates. Prolog supports first-order Horn clauses with only limited higher-order features.  $\lambda$ Prolog is a higher-order logic programming language that extends Prolog by incorporating higher-order unification and  $\lambda$ -terms [14]. Because of its support of higher-order Horn clauses [15],  $\lambda$ Prolog makes an excellent logic programming language for representing logic program schemata and programming techniques. In this paper, we present a set of logic program schemata and programming techniques. We begin by highlighting those features of  $\lambda$ Prolog which are useful in defining logic program schemata and programming techniques.

## 2 Classifying Logic Programs

The present work is motivated by our previous research on Prolog program schemata [7,10], the work by Brna and his colleagues on Prolog programming techniques [3,4], the work by Barker-Plummer on Prolog clichés [2], and the related work by Kirschenbaum and Sterling on Prolog skeletons and programming techniques [12,13]. The major shortcoming of each of these approaches is their reliance on an abstract meta-language to represent the program schemata and programming techniques. We propose the use of  $\lambda$ Prolog which alleviates the need for a meta-language.  $\lambda$ Prolog is an ideal choice for a representation language since it supports higher-order Horn clauses.  $\lambda$ Prolog schemata and techniques are represented as  $\lambda$ Prolog programs.

The basic syntactic conventions of  $\lambda$ Prolog are the same as those of Prolog: all legal statements must be in clausal form where  $:-$  represents implication, the comma represents conjunction, the semicolon represents disjunction, cut is represented by the exclamation mark, and identifiers that begin with an uppercase letter represent variables while identifiers that begin with a lowercase letter represent constants. The same set of built-in predicates for unifying terms and evaluating arithmetic expressions exist in  $\lambda$ Prolog. In addition to  $\lambda$ Prolog's support of predicate variables and  $\lambda$ -terms, the most notable difference between the syntax of the Prolog and  $\lambda$ Prolog is that  $\lambda$ Prolog uses a curried notation. Thus, the Prolog program `enqueue/3`:

```
enqueue([], E, [E]).
enqueue([H|T], E, [H|X]) :- enqueue(T, E, X).
```

which enqueues an element onto the end of a list would be written in  $\lambda$ Prolog's curried form as:

```
enqueue [] E [E].
enqueue [H|T] E [H|X] :- enqueue T E X.
```

$\lambda$ -terms are used in  $\lambda$ Prolog to represent predicate application and anonymous predicates (i.e., predicates that have no name associated with them). Predicate application is denoted in  $\lambda$ Prolog by juxtaposition. Anonymous predicates are denoted with  $\lambda$ -abstractions which have the form  $\lambda x.\rho(x)$  in  $\lambda$ -calculus and the form  $(X\backslash(\rho(X)))$  in  $\lambda$ Prolog and represents an anonymous predicate that has a single argument  $X$  which succeeds if  $\rho(X)$  succeeds where  $\rho(X)$  is an arbitrary set of  $\lambda$ Prolog subgoals. In addition to supporting  $\lambda$ -terms,  $\lambda$ Prolog also permits existential quantifiers.  $\lambda$ Prolog uses the keyword `sigma` to represent the existential quantifier  $\exists$  so the  $\lambda$ -term  $\lambda x.\lambda y.\exists z.(p\ x\ y\ z)$  would be coded in  $\lambda$ Prolog as  $(X\backslash Y\backslash(\text{sigma } Z\backslash(p\ X\ Y\ Z)))$  and represents an anonymous predicate that has two arguments  $X$  and  $Y$  which succeeds if  $p\ X\ Y\ Z$  succeeds for some  $Z$ .

Another important difference between Prolog and  $\lambda$ Prolog is that  $\lambda$ Prolog is a typed language.  $\lambda$ Prolog has several built-in types, including types for *int*, *bool*, *list*, and *o* (the type of propositions). If  $\tau_1$  and  $\tau_2$  are types then  $(\tau_1 \rightarrow \tau_2)$  is a type corresponding

to the set of functions whose domain and range are given by  $\tau_1$  and  $\tau_2$ , respectively. The application of  $T_1$  to  $T_2$  is represented as  $(T_1 T_2)$  and has the type  $\tau_1$  if  $T_1$  is a term of type  $(\tau_2 \rightarrow \tau_1)$  and  $T_2$  is a term of type  $\tau_2$ . If  $X$  is a variable and  $T$  is a term of type  $\tau'$ , then the abstraction  $(X : \tau \setminus T)$  is a term of type  $\tau \rightarrow \tau'$ .

$\lambda$ Prolog has a built-in type inference mechanism which gives its programmers the illusion that they are programming in a typeless language. Thus, the type system of  $\lambda$ Prolog serves as an aid to the programmer rather than an added layer of syntax. Lists and integers are handled the same way in  $\lambda$ Prolog as they are in Prolog. Unlike Prolog, however,  $\lambda$ Prolog supports separate types for propositions and booleans. The type  $o$  captures propositions and has the values `true` and `fail` and operations for conjunction, disjunction, and implication of propositions. The type  $bool$  captures boolean expressions and has the values `truth` and `false` and operations for conjunction and disjunction of booleans, and relationship comparisons (`<`, `=<`, `>`, `>=`). Note that because booleans are distinct from propositions, it is necessary to have the  $\lambda$ Prolog subgoal `truth is X < Y` in place of the Prolog subgoal `X < Y`.

### 3 Logic Program Schemata

It is possible to define a set of  $\lambda$ Prolog program schemata which enable the incorporation of conditional recursion into logic programming. Conditional iteration in the form of FOR and WHILE loops imposes structure on imperative languages, abstracting the essence of conditional repetition. FOR and WHILE loops are basic program schemata which capture commonly occurring imperative programming techniques. Conditional recursion serves the same role for logic programming languages.

Lists are a basic type in most logic programming languages, including Prolog and  $\lambda$ Prolog. As an example of a list processing task, assume we simply want to traverse a list of elements and count the number of elements in the list. A  $\lambda$ Prolog program which does this task could be written:

```
length [] 0.
length [H|T] Result :- length T X, Result is X + 1.
```

Assume we wanted to find the summation of all the elements of an arbitrary list of integers (e.g., the summation of the list [2,4,1,9,12,3] is 31). A  $\lambda$ Prolog program which does this task could be written:

```
sum [] 0.
sum [H|T] Result :- sum T X, Result is X + H.
```

A pattern seems to be arising here. The only difference between `sum/2` and `length/2` is that `sum/2` adds the first element to the sum of the remainder of the list whereas `length/2` merely increments the length of the remainder of the list by one. Indeed, this pattern is quite common among list processing tasks. Consider, for example, the task of finding the product of a list of numbers (e.g., the product of the list

[2,4,1,9,12,3] is 2592). Thus, we merely change the `Result is X + H` subgoal to `Result is X * H` in the body of the clause and we have a `product/2` program:

```
product [] 1.
product [H|T] Result :- product T X, Result is X * H.
```

Note, however, that `product/2` has an additional difference. The base case value was changed from 0 to 1 since 1 is the identity for multiplication. These programs are not the only ones that show this common pattern. Other programs which perform quite different tasks on lists are also members of this class of programs. Consider appending two lists together. A  $\lambda$ Prolog program for this task would look like:

```
append [] L L.
append [H|T] L Result :- append T L X, Result = [H|X].
```

The more common implementation of `append/3` has the `Result = [H|X]` subgoal unfolded into the head of the clause:

```
append [] L L.
append [H|T] L [H|X] :- append T L X.
```

One of the most commonly used program examples for recursive list processing is list reversal. Although there are other ways to write a list reversal program in  $\lambda$ Prolog, the one produced by most novice programmers is the following naive `reverse/2`:

```
reverse [] [].
reverse [H|T] Result :-
    reverse T X, append X [H] Result.
```

High-level programming languages were developed to make programming easier by abstracting out the essence of programming from the physical architecture of the machine on which the programs are executed. The progression of high-level programming languages over time has shown higher levels of abstraction. For example, control structures like FOR loops were included in imperative programming languages because it was discovered that looping structures were used throughout programs. A similar abstraction is possible for recursive programs. All of the programs presented so far fall into a class of programs that can be captured by the higher-order `reduceAll/4` program:

```
reduceAll [] Base Constructor Base.
reduceAll [A|B] Result Constructor Base :-
    reduceAll B C Constructor Base,
    Constructor A C Result.
```

which can be written using a single disjunctive clause:

```

reduceAll List Result Constructor Base :-
  (List = [], Result = Base);
  (List = [A|B],
   reduceAll B C Constructor Base,
   Constructor A C Result).

```

We can write each of the programs that we have seen so far as one line programs by simply invoking the `reduceAll/4` program. For example, we can write `append/3` more concisely as follows:

```

append A B C :- reduceAll A C
  (X\Y\Z\Z = [X|Y])
  B.

```

We can also write a  $\lambda$ Prolog implementation of `reverse/2` using only `reduceAll/4` as well:

```

reverse A B :- reduceAll A B
  (F\G\H\H\reduceAll G H (X\Y\Z\Z = [X|Y])) [F]
  [].

```

This implementation of `reverse/2` shows the notion of *nested recursion*, which is synonymous to nested FOR loops in imperative languages. An alternative implementation of `reverse/2` takes advantage of a pre-existing procedure definition for `append/3` and invokes it directly rather than redefining it:

```

reverse A B :- reduceAll A B
  (X\Y\Z\append Y [X] Z)
  [].

```

In order to capture a larger class of programs, we must generalize `reduceAll/4` to permit arbitrary termination conditions. We also need to generalize the base case *value* to a base case *predicate* so we have more generality in our base case clause. Furthermore, we need to have an arbitrary list destructor. We can produce the higher-order program `reduce/6` which represents this more robust program schema:

```

reduce List Result Destructor Constructor Stop Base :-
  (Stop List, Base List Result);
  (Destructor List A B,
   reduce B C Destructor Constructor Stop Base,
   Constructor A C Result).

```

Some explanation of the  $\lambda$ Prolog `reduce/6` program schema is in order. It contains six arguments. The first argument is the primary input, while the second argument is the primary output. As will be explained later, the primary input and output can be either simple or structured terms, but they are both first-order terms. The remaining four arguments are second-order terms representing arbitrary  $\lambda$ Prolog

predicates. The first of these arguments defines the process for destructing the input, while the second of these arguments defines the process for constructing the output. The last two arguments are used to define the terminating condition, defining the process to identify the terminating condition and the process which defines how to construct the output for the terminating condition, respectively. Some examples should help clarify `reduce/6`.

Consider `append/3` again. For an arbitrary query `append A B C`, the primary input is A and primary output is C. The destructor predicate decomposes the input into the head element and the tail of the list. This process can be defined with the anonymous predicate  $(X\backslash Y\backslash Z\backslash(X = [Y|Z]))$ . Likewise, the constructor predicate for `append/3` composes a new list and can be defined with the anonymous predicate  $(X\backslash Y\backslash Z\backslash(Z = [X|Y]))$ . The terminating condition occurs whenever the input list becomes empty and can be defined with the anonymous predicate  $(X\backslash(X = []))$ . As can be seen in the base case clause of the definition of `append/3`, the terminating output value should be assigned to B. Combining all this together produces the following definition for `append/3`:

```
append A B C :- reduce A C
  (X\Y\Z\X = [Y|Z])
  (X\Y\Z\Z = [X|Y])
  (X\X = [])
  (X\Y\Y = B).
```

In a similar fashion, we can define `reverse/2` (which reverses the elements of a list) using `reduce/6`:

```
reverse A B :- reduce A B
  (X\Y\Z\X = [Y|Z])
  (X\Y\Z\append Y [X] Z)
  (X\X = [])
  (X\Y\Y = []).
```

We can even rewrite `reduceAll/4` (or any of the other programs we've seen so far) using `reduce/6`:

```
reduceAll A B C D :- reduce A B
  (X\Y\Z\X = [Y|Z])
  C
  (X\X = [])
  (X\Y\Y = D).
```

All of the programs that we have seen so far fall in the class of *global list processing* programs since they process the entire input list. Some programs only process part of the input list. For example, `insert/3` takes a sorted list and an element and inserts the element in its correct position in the list, stopping whenever it finds an element in the input list that is larger than the element being inserted or the input list

becomes empty. A  $\lambda$ Prolog program for `insert/3` can be written using `reduce/6`:

```
insert A B C :- reduce A C
  (X\Y\Z\ (X = [Y|Z]))
  (X\Y\Z\ (Z = [X|Y]))
  (X\(\sigma V\(\sigma W\ (
    (X = []);
    (X = [V|W], truth is B < V)
  ))))
  (X\Y\ (Y = [B|X])).
```

The main difference between `insert/3` and all of the previous programs is the terminating condition. There are actually two terminating conditions. The correct position for insertion of a given element into a sorted list is either immediately in front of the first element in the list which is larger than the given element or at the end of the list if every element in the list is smaller than or equal to the given element. The use of `sigma` identifies the existence of variables which satisfy the terminating condition. Specifically, the  $\lambda$ -term  $(X\(\sigma V\(\sigma W\ ((X = []); (X = [V|W], \text{truth is } B < V))))$  represents an anonymous one-argument predicate which succeeds if either its argument is an empty list (i.e., all of the elements in the original list are smaller than or equal to the given element) or if there exist variables  $V$  and  $W$  such that the predicate's argument is unifiable with  $[V|W]$  where  $V$  is larger than the given element (i.e.,  $V$  is the smallest element in the original list which is larger than the given element).

Other classic *partial list processing* programs include `member/2` and `position/3`. Each of these predicates can be written using `reduce/6`. The predicate `member/2` is a predicate that produces no outputs, it merely succeeds if the given element is a member of the input list or fails if the given element is not a member of the input list. Thus, the  $\lambda$ Prolog implementation of `member/2` has a dummy variable in place of the output argument and has the subgoal `true` in place of the recursive and base case constructors in its invocation of `reduce/6`:

```
member A B :- reduce A Dummy
  (X\Y\Z\ (X = [Y|Z]))
  (X\Y\Z\ (true))
  (X\(\sigma W\ (X = [B|W])))
  (X\Y\ (true)).
```

The predicate `position/3` takes an input list and an element and returns the position of the element in the input list:

```
position A B P :- reduce A P
  (X\Y\Z\ (X = [Y|Z]))
  (X\Y\Z\ (Z is Y + 1))
  (X\(\sigma W\ (X = [B|W])))
  (X\Y\ (Y = 1)).
```

Rather than reducing a list by combining each element with the result of reducing the remainder of the list, sometimes it is desirable to map a function predicate across all the elements of a list. For example, we may want to double all of the elements in a list. In order to double all of the elements of a list, we must first apply a function predicate that doubles each element and then put the doubled element in the front of the list produced by doubling all the elements in the remainder of the list. In general, the predicate `map/3` can be used to apply an arbitrary binary function predicate to each element of a list:

```
map A B C :- reduce A C
            (X\Y\Z\X = [Y|Z])
            (X\Y\Z\(\sigma W\B X W, Z = [W|Y]))
            (X\X = [])
            (X\Y\Y = []).
```

We can write `doubleAll/2` using this `map/3` predicate:

```
doubleAll A B :- map A (X\Y\Y is 2 * X) B.
```

Another common higher-order predicate is `filter/3`. The predicate `filter/3` takes a unary predicate and a list and filters out all elements from the list that do not satisfy the predicate. For example, we may want to filter out all non-positive numbers from a list of numbers. We can write `filter/3` using `reduce/6`:

```
filter A B C :- reduce A C
              (X\Y\Z\X = [Y|Z])
              (X\Y\Z\((B X, Z = [X|Y]); Z = Y))
              (X\X = [])
              (X\Y\Y = []).
```

We can write `positivesOnly/2` using this `filter/3` predicate:

```
positivesOnly A B :- filter A (X\truth is X > 0) B.
```

It is also possible to capture programs that construct more than one output or have more than one input list that is being decomposed. Consider the task of partitioning an input list into two output lists which contain all the elements that are less than or equal to a given partitioning element and one which contains all elements that are strictly greater than the given partitioning element. A  $\lambda$ Prolog implementation of `partition/4` can be written using `reduce/6` by combining the two output lists into a single list which is manipulated appropriately by the constructor predicates:

```

partition A B C D :- reduce A [C,D]
  (X\Y\Z\X = [Y|Z])
  (X\Y\Z\(\sigma V\(\sigma W\
    (false is X > B, Y = [V,W], Z = [[X|V],W]);
    (truth is X > B, Y = [V,W], Z = [V,[X|W]])
  )))
  (X\X = [])
  (X\Y\Y = [[B],[ ]]).

```

Notice the use of disjunction in the constructor predicate to permit putting the current element on the appropriate output list. The subgoal  $Y = [V,W]$  breaks the output into its two output lists  $V$  and  $W$ . If the current element is less than or equal to the partitioning element then the current element is added to the first output list with the subgoal  $Z = [[X|V],W]$ . Otherwise, it is added to the second output list with the subgoal  $Z = [V,[X|W]]$ .

Now consider the task of merging two sorted lists into a single sorted list. This task requires decomposing two input lists. A  $\lambda$ Prolog implementation of `merge/3` can be written using `reduce/6` by combining the two input lists into a single input list which is manipulated appropriately by the destructor predicate:

```

merge A B C :- reduce [A,B] C
  (X\Y\Z\(\sigma U\(\sigma V\(\sigma W\
    (X = [[Y|U],[V|W]], truth is Y < V, Z = [U,[V|W]]);
    (X = [[V|W],[Y|U]], false is Y > V, Z = [[V|W],U])
  )))
  (X\Y\Z\Z = [X|Y])
  (X\(\sigma W\((X = [[ ],W]); (X = [W,[ ]])))
  (X\Y\((X = [[ ],Y]); (X = [Y,[ ]])).

```

Disjunction is used in the destructor predicate in the definition of `merge/3` to enable removing the first element from only one of the input lists. The element is removed from whichever input list has the smallest element. If the smallest element is contained in the first input list then it is "identified" with the subgoals  $X = [[Y|U], [V|W]]$ ,  $\text{truth is } Y < V$  and "removed" with the subgoals  $X = [U, [V|W]]$ ,  $\text{false is } Y > V$ . Likewise, if the smallest element is contained in the second input list (or if the first element in both input lists is identical) then it is "identified" with the subgoal  $X = [[V|W], [Y|U]]$  and "removed" from the second input list with the subgoal  $X = [[V|W], U]$ .

As a last example of the use of `reduce/6`, consider the task of sorting a list of elements. One common method of sorting is insertion sort. Insertion sort is a global list processing program. For each element that is processed, insertion sort ensures that the element is inserted into its correct position in the output list using the predicate `insert/3`. A  $\lambda$ Prolog implementation of `insertion_sort/2` using `reduce/6` can be written:

```

insertion_sort A B :- reduce A B
  (X\Y\Z\(X = [Y|Z]))
  (X\Y\Z\(insert Y X Z))
  (X\(X = []))
  (X\Y\(Y = [])).

```

It is also possible to write `insertion_sort/2` in terms of `merge/3` instead of `insert/3`. The advantage of this is that `merge/3` is associative and therefore enables the use of the accumulator version of `reduce/6` discussed in the next section. The following implementation of `insertion_sort/2` uses `merge/3` instead of `insert/3`:

```

insertion_sort A B :- reduce A B
  (X\Y\Z\(X = [Y|Z]))
  (X\Y\Z\(merge Y [X] Z))
  (X\(X = []))
  (X\Y\(Y = [])).

```

## 4 Constructing Logic Program Schemata

In the previous section, we identified some program schemata and showed that they capture a wide class of programs. Another important pattern that occurs in programs is the application of programming techniques. One programming technique is to decompose the input list from the back rather than the front as in the following inverse implementation of `reverse/2`:

```

reverse A B :- reduce A B
  (X\Y\Z\(append Z [Y] X))
  (X\Y\Z\(Z = [X|Y]))
  (X\(X = []))
  (X\Y\(Y = [])).

```

Another programming technique is the use of accumulator pairs to build recursive arguments. The use of accumulator pairs is a commonly known programming technique and has appeared throughout the literature as a method of improving program efficiency. For example, the efficiency of `reverse/2` can be improved by accumulating the result:

```

reverse A B :- reduce A [B,[]]
  (X\Y\Z\(X = [Y|Z]))
  (X\Y\Z\(sigma V\(sigma W\
    Z = [V,W], Y = [V, [X|W]]
  )))
  (X\(X = []))
  (X\Y\(sigma W\Y = [W,W])).

```

which produces the well-known accumulator version (or difference-list and railway-shunt versions which are accumulator versions that merely hide the accumulator within

the output argument) of `reverse/2`. It is also possible to write an accumulator version of `reverse/2` using `reduceAll/4`:

```
reverse A B :- reduceAll A [] (X\Y\Z\((Y = [X|Z]))) B.
```

by using the second argument (`Result`) as the accumulator and using the last argument (`Base`) as the final result. In fact, any global list processing program with an associative constructor that can be written in pure  $\lambda$ Prolog can be defined using `reduceAll/4`. We need to formally define the notion of an associative predicate for predicates with three arguments. For example, the predicate `append/3` is associative:

```
append [] L L.
append [H|T] L [H|X] :- append T L X.
```

There are two ways of appending three lists `A`, `B`, and `C` together. We can either append `B` to the end of `A` and then append `C` to end of this list or we can append `C` to the end of `B` and then append this list to the end of `A` (i.e., `append A B X`, `append X C D` or `append B C Y`, `append A Y D`). Formally, the predicate `p/3` is *associative* if the following holds:

$$\forall U, V, W, X, Y, Z ((p\ Y\ W\ X\ \text{iff}\ p\ U\ Z\ X) \\ \text{if } (p\ U\ V\ Y\ \wedge\ p\ V\ W\ Z))$$

The predicate `merge/3` given in the previous section is also associative. Note that the subgoal `X = [Y|Z]` could be replaced by the subgoal `append [Y] Z X` so such subgoals can be thought of as associative as well. Subgoals of the form `X is Y  $\oplus$  Z` where  $\oplus$  is either addition (+) or multiplication (\*) are also associative. Note, however, that predicates that contain the arithmetic evaluator `is/2` cannot be written using `reduceAll/4` because they are dependent on subgoal order. For example, although `insertion_sort/2` is a global list processing program and its constructor `merge/3` is associative, it is not possible to define `insertion_sort/2` using `reduceAll/4` because the righthand side of its `is/2` subgoals will contain unbound variables. For such programs, it is necessary to define an accumulator version of `reduceAll/4` which serves the same function as `reduceAll/4`, but reverses the order of its subgoals:

```
reduceAllAcc List Acc Constructor Result :-
  (List = [], Result = Acc);
  (List = [A|B],
   Constructor A Acc C,
   reduceAllAcc B C Constructor Result).
```

We can define all of the global list processing programs with associative constructors presented in the previous section using `reduceAllAcc/4`. For example, `length/2`, `reverse/2`, and `insertion_sort/2` can be defined using `reduce-`

AllAcc/4:

```
length A B :- reduceAllAcc A 0
             (X\Y\Z\Z is Y + 1)
             B.

reverse A B :- reduceAllAcc A []
             (X\Y\Z\Z = [X|Y])
             B.

insertion_sort A B :- reduceAllAcc A []
                    (X\Y\Z\merge [X] Y Z)
                    B.
```

The predicate `reduceAllAcc/4` corresponds to the notion of reducing the input from the left while the predicate `reduceAll/4` corresponds to reducing the input from the right. Thus,  $\lambda$ Prolog schemata enable a generic description of a problem solution that is independent of the direction of the input reduction. Programming techniques represent components of programs that can be applied to general program schemata to produce more specialized program schemata like the `reduceAllAcc/4`, `reduceAll/4`, and `reduce/6`.

There are two major types of programming techniques: control flow techniques and context techniques. Control flow techniques are applied to program schemata as a way of defining the basic recursive control flow of the program schema. One example of this type of programming technique is the *accumulator* technique which was just presented. Two other common recursive control flow techniques are *single\_recursion* and *double\_recursion* which enable the creation of singly recursive and doubly recursive program schemata, respectively.

Context techniques are applied to program schemata as a way of defining a context for arguments. The three most common context techniques are *same*, *decompose*, and *compose* which define an argument to be the same across recursive calls, decrease in size across recursive calls, and increase in size across recursive calls, respectively. The techniques *list\_subgoal* and *list\_head* are special forms of *decompose* and *compose* which apply to list arguments. Each of these techniques are highlighted in the following paragraphs.

Programming techniques constitute components of programs and program schemata which enables the generation of specialized program schemata from more generalized program schemata. For example, it is possible to apply techniques to the following general  $\lambda$ Prolog program schema:

```
schema A1 ... An :- Goals.
```

to produce the `reduce/6` program schemata or any of the programs given in this paper. We begin with a 6-argument version of this schema and apply the *single\_recursion* technique. The *single\_recursion* technique is used to split the body of a clause into a

disjunction of two sets of subgoals. One of the subgoals contains base case clauses, while the other set of subgoals contains a recursive call to the predicate. Applying this technique to the general  $\lambda$ Prolog program schema produces:

```

schema A1 A2 A3 A4 A5 A6 :-
  Base;
  (Before, schema B1 B2 B3 B4 B5 B6, After).

```

We can apply the *same* technique to the last four arguments producing:

```

schema A1 A2 A3 A4 A5 A6 :-
  Base;
  (Before, schema B1 B2 A3 A4 A5 A6, After).

```

Now we can instantiate the `Base` variable in the base portion and the `Before` and `After` variables in the recursive portion of this clause to the subgoals  $(A5\ A1, A6\ A1\ A2), A3\ A1\ A\ B1,$  and  $A4\ A\ B2\ A2,$  respectively. The resultant clause is identical (modulo variable renaming) to `reduce/6`. By beginning with a 4-argument  $\lambda$ Prolog program schema, we can produce the global list processing `reduceAll/4` schema. After applying the *single\_recursion* technique, we have:

```

schema A1 A2 A3 A4 :-
  Base;
  (Before, schema B1 B2 B3 B4, After).

```

We can apply the *same* technique to the last two arguments producing:

```

schema A1 A2 A3 A4 :-
  Base;
  (Before, schema B1 B2 A3 A4, After).

```

Applying the *list head* technique to the first argument produces:

```

schema A1 A2 A3 A4 :-
  Base;
  (Before, A1 = [A|B1], schema B1 B2 A3 A4, After).

```

Now we can remove `Before` (or instantiate it to `true`) and instantiate `Base` and `After` to  $(A1 = [], A4 = A2)$  and  $(A3\ A\ B2\ A2),$  respectively. The resultant clause is identical (modulo variable renaming) to `reduceAll/4` which captures all global list processing programs (e.g., `append/3` and `reverse/2`). By beginning with a 4-argument  $\lambda$ Prolog schemata, we can produce the `reduceAllAcc/4` schema. After applying the *single\_recursion* technique, we have:

```

schema A1 A2 A3 A4 :-
  Base;
  (Before, schema B1 B2 B3 B4, After).

```

We can apply the *same* technique to the last two arguments producing:

```

schema A1 A2 A3 A4 :-
  Base;
  (Before, schema B1 B2 A3 A4, After).

```

Now we can instantiate the `Base` and `Before` variables to  $(A1 = [], A2 = A4)$  and  $(A1 = [A|B1], A3 = A2 B2)$ , respectively. After removing `After` (or instantiating it to `true`), the resultant clause is identical (modulo variable renaming) to `reduceAllAcc/4`.

To this point, all of the programs and schemata that we have presented have had a single recursive subgoal in the recursive clause. Many well-known algorithms require the use of multiple recursive subgoals (e.g., quicksort). In order to accommodate these programs, it is necessary to support a *double\_recursive* technique which permits the creation of a recursive clause with two recursive subgoals. Applying this technique to a 5-argument schema leads to:

```

reduceDouble List Result Divide Constructor Stop Base :-
  (Stop List, Base List Result);
  (Divide List A B,
   reduceDouble A AR Divide Constructor Stop Base,
   reduceDouble B BR Divide Constructor Stop Base,
   Constructor AR BR Result).

```

which can be used to produce *divide and conquer* programs like `quicksort/2`. The important thing to note about our approach to generating  $\lambda$ Prolog program schemata is that there is no need for a meta-language to represent the programming techniques. Furthermore, all intermediate programs are valid (albeit not very useful)  $\lambda$ Prolog programs.

## 5 Conclusion

Imperative programming languages provide their programmers with a set of structured programming constructs. There are currently no structured constructs, however, in logic programming languages. Conditional iteration in the form of `FOR` and `WHILE` loops imposes structure on imperative languages, abstracting the essence of conditional repetition. `FOR` and `WHILE` loops are basic program schemata which capture commonly occurring imperative programming techniques. Conditional recursion serves the same role for logic programming languages.

In this paper, we have argued that it is possible to incorporate a structured style of programming into logic programming languages by exploiting program schemata and

programming techniques. Program schemata capture the notion of conditional recursion which serves the same role for logic programming languages that conditional iteration does for imperative languages. Programming techniques correspond to common program components within program schemata. Representing program schemata and programming techniques requires a higher-order representation language.

Previous approaches to representing program schemata and programming techniques have relied on the introduction of an abstract meta-language. Higher-order logic programming languages like  $\lambda$ Prolog provide an alternative to the meta-language approach. In addition to providing an alternative to an abstract meta-language,  $\lambda$ Prolog's ability to represent  $\lambda$ Prolog program schemata and programming techniques as  $\lambda$ Prolog programs enables  $\lambda$ Prolog to support conditional recursion which promotes a more structured style of logic programming.

Schemata serve a fundamental role in most human cognitive processes. It has been shown that schemata enable the organization of meaningful information for complex domains like computer programming. In addition to being essential to expert programmers, program schemata have also been shown to be useful in teaching recursive Prolog programming to novices. Thus, the incorporation of structured constructs into logic programming made possible with logic program schemata and programming techniques enhances the logic programming paradigm for programmers of all levels.

## References

- [1] B. Adelson. Problem Solving and the Development of Abstract Categories in Programming Languages. *Memory & Cognition*, 9: 422-433, 1981.
- [2] D. Barker-Plummer. Cliché Programming in Prolog. In M. Bruynooghe, editor, *Proceedings of the 2<sup>nd</sup> Workshop on Meta-Programming in Logic*, Leuven, Belgium, pages 247-256, 1990.
- [3] A. Bowles and P. Brna. Programming Plans and Programming Techniques. In P. Brna, S. Ohlsson, and H. Pain, editors, *Proceedings of the 6<sup>th</sup> World Conference on Artificial Intelligence in Education*, Edinburgh, Scotland, pages 378-385, AACE Press, 1993.
- [4] P. Brna, A. Bundy, A. Dodd, M. Eisenstadt, C. Looi, H. Pain, D. Robertson, B. Smith, and M. van Someren. Prolog Programming Techniques. *Instructional Science*, 20: 111-133, 1991.
- [5] P. Flener and Y. Deville. Towards Stepwise, Schema-Guided Synthesis of Logic Programs. In T.P. Clement and K. Lau, editors, *Proceedings of the 1<sup>st</sup> International Workshop on Logic Program Synthesis and Transformation*, Manchester, England, pages 46-64, Springer-Verlag, 1991.

- [6] N.E. Fuchs and M.P.J. Fromherz. Schema-Based Transformations of Logic Programs. In T.P. Clement and K. Lau, editors, *Proceedings of the 1<sup>st</sup> International Workshop on Logic Program Synthesis and Transformation*, Manchester, England, pages 111-125, Springer-Verlag, 1991.
- [7] T.S. Gegg-Harrison. *Basic Prolog Schemata*. Technical Report CS-1989-20, Department of Computer Science, Duke University, Durham, North Carolina, 1989.
- [8] T.S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20: 173-192, 1991.
- [9] T.S. Gegg-Harrison. Adapting Instruction to the Student's Capabilities. *Journal of Artificial Intelligence in Education*, 3: 169-181, 1992.
- [10] T.S. Gegg-Harrison. *Exploiting Program Schemata in a Prolog Tutoring System*. Ph.D. Dissertation, Department of Computer Science, Duke University, Durham, North Carolina, 1993.
- [11] T.S. Gegg-Harrison. Exploiting Program Schemata in an Automated Program Debugger. *Journal of Artificial Intelligence in Education*, 5: 255-278, 1994.
- [12] M. Kirschenbaum and L.S. Sterling. *Prolog Programming using Skeletons and Techniques*. Technical Report TR-90-109, Department of Computer Science, Case Western Reserve University, Cleveland, Ohio, 1990.
- [13] M. Kirschenbaum and L.S. Sterling. Applying Techniques to Skeletons. In J. Jacquet, editor, *Constructing Logic Programs*, pages 127-140, MIT Press, 1993.
- [14] G. Nadathur and D. Miller. An Overview of  $\lambda$ Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5<sup>th</sup> International Conference and Symposium on Logic Programming*, Seattle, Washington, pages 810-827, MIT Press, 1988.
- [15] G. Nadathur and D. Miller. Higher-Order Horn Clauses. *Journal of the ACM*, 37: 777-814, 1990.
- [16] M.W. van Someren. What's Wrong? Beginners' Problems with Prolog. *Instructional Science*, 19: 257-282, 1990.

## Appendix - $\lambda$ Prolog Program Schemata

Global List Processing Schema:

```
reduceAll List Result Constructor Base :-  
  (List = [], Result = Base);  
  (List = [A|B],  
   reduceAll B C Constructor Base,  
   Constructor A C Result).
```

Global List Processing Accumulator Schema:

```
reduceAllAcc List Acc Constructor Result :-  
  (List = [], Result = Acc);  
  (List = [A|B],  
   Constructor A Acc C,  
   reduceAllAcc B C Constructor Result).
```

General List Processing Schema:

```
reduce List Result Destructor Constructor Stop Base :-  
  (Stop List, Base List Result);  
  (Destructor List A B,  
   reduce B C Destructor Constructor Stop Base,  
   Constructor A C Result).
```

General List Processing Accumulator Schema:

```
reduceAcc List Acc Destructor Constructor Stop Result :-  
  (Stop List, Res = Acc);  
  (Destructor List A B,  
   Constructor A Acc C,  
   reduceAcc B C Destructor Constructor Stop Result).
```

Divide and Conquer Schema:

```
reduceDouble List Result Divide Constructor Stop Base :-  
  (Stop List, Base List Result);  
  (Divide List A B,  
   reduceDouble A AR Divide Constructor Stop Base,  
   reduceDouble B BR Divide Constructor Stop Base,  
   Constructor AR BR Result).
```