Extensible Logic Program Schemata

Timothy S. Gegg-Harrison

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

Abstract. Schema-based transformational systems maintain a library of logic program schemata which capture large classes of logic programs. One of the shortcomings of schema-based transformation approaches is their reliance on a large (possibly incomplete) set of logic program schemata that is required in order to capture all of the minor syntactic differences between semantically similar logic programs. By defining a set of extensible logic program schemata and an associated set of logic program transformations, it is possible to reduce the size of the schema library while maintaining the robustness of the transformational system. In our transformational system, we have defined a set of extensible logic program schemata in λProlog. Because λProlog is a higher-order logic programming language, it can be used as the representation language for both the logic programs and the extensible logic program schemata. In addition to the instantiation of predicate variables, extensible logic program schemata can be extended by applying standard programming techniques (e.g., accumulating results), introducing additional arguments (e.g., a second list to append to the end of the primary list), combining logic program schemata which share a common primary input, and connecting logic program schemata which are connected via a result of one schema being an input to the other schema. These extensions increase the robustness of logic program schemata and enhance traditional schema-based transformational systems.

1 Introduction

Schema-based transformational systems maintain a library of logic program schemata which capture large classes of logic programs. One of the shortcomings of schema-based transformation approaches is their reliance on a large (possibly incomplete) set of logic program schemata that is required in order to capture all of the minor syntactic differences between semantically similar logic programs. By defining a set of extensible logic program schemata and an associated set of logic program transformations, it is possible to reduce the size of the schema library while maintaining the robustness of the transformational system. Our schema-based approach to logic program transformation is similar to the schema-based transformations of Fuchs and his colleagues [6,16]. The main difference is that their schema language which was developed for representing

Prolog schemata in a tutoring system [7,8] is not the same as their object language which is Prolog. We propose using a higher-order logic programming language to represent the logic programs and the set of extensible logic program schemata.

Logic program schemata have proven useful in teaching recursive logic programming to novices [8,9], debugging logic programs [10], transforming logic programs [5,6,16], and synthesizing logic programs [3,4]. A number of researchers have looked into various approaches to meta-languages which support logic program schemata, including our work on basic Prolog schemata [7,8], the work by Brna and his colleagues on Prolog programming techniques [2], Barker-Plummer's work on Prolog clichés [1], the work of Marakakis and Gallagher on program design schemata [14], Flener's work on logic algorithm schemata [3,4], and the work by Hamfelt and Fischer Nilsson on metalogic programming techniques [12]. An alternative approach to using a meta-language to represent program schemata is to have a set of general Prolog programs which can be extended by adding arguments and subgoals to produce other programs within the class. This is the approach taken by Sterling and his colleagues with their Prolog skeletons that are extended with programming techniques [13]. The present proposal attempts to merge both of these approaches by using a higher-order logic programming language as both the object language and the meta-language. Like Sterling's skeletons, extensible logic program schemata capture well-understood control flow patterns in logic programs and can be extended by applying programming techniques. Like traditional logic program schemata, extensible logic program schemata are higher-order logic programs that contain predicate variables which can be instantiated to produce logic programs with the same basic structure. Thus, extensible logic program schemata combine the strengths of both approaches.

2 Logic Program Schemata

λProlog is a higher-order logic programming language that extends Prolog by incorporating higher-order unification and λ-terms [15]. The syntactic conventions of λProlog are mostly the same as those of Prolog. In addition to λProlog's support of predicate variables and λ-terms, the most notable difference between the syntax of the Prolog and λProlog is that λProlog uses a curried notation. The Prolog program `sum/2`:

```
sum([],0).
sum([H|T],B) :- sum(T,R), B is H + R.
```

which finds the summation of all the elements in its input list would be written in λProlog's curried form as:

```
sum [] 0.
sum [H|T] B :- sum T R, B is H + R.
```

We can rewrite `sum/2` as a single clause using disjunction:

```
sum A B :- (A = [], B = 0);
           (A = [H|T], sum T R, B is H + R).
```

which enables us to write `sum/2` in λProlog is as a λ-term:

```
sum A B :- (X\Y\(sigma H\(sigma T\(sigma R\(
            (X = [], Y = 0);
            (X = [H|T], sum T R, Y is H + R)
           ))))) A B.
```

λ-terms are used in λProlog to represent predicate application and anonymous predicates. Predicate application is denoted in λProlog by juxtaposition. Anonymous predicates are denoted with λ-abstractions which have the form `λx.ρ(x)` in λ-calculus and the form `(X\(ρ(X))` in λProlog and represents an anonymous predicate that has a single argument `X` which succeeds if `ρ(X)` succeeds where `ρ(X)` is an arbitrary set of λProlog subgoals. In addition to supporting λ-terms, λProlog also permits existential quantifiers. λProlog uses the keyword `sigma` to represent the existential quantifier ∃ so the λ-term `λx.λy.∃z.(p x y z)` would be coded in λProlog as `(X\Y\(sigma Z\(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 λProlog is that λProlog is a typed language. λProlog has several built-in types, including types for *int*, *bool*, *list*, and *o* (the type of propositions). If τ_1 and τ_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 τ_1 and τ_2, respectively. The application of T_1 to T_2 is represented as $(T_1\ T_2)$ and has the type τ_1 if T_1 is a term of type $(\tau_2 \rightarrow \tau_1)$ and T_2 is a term of type τ_2. If X is a variable and T is a term of type τ', then the abstraction $(X : \tau \backslash T)$ is a term of type $\tau \rightarrow \tau'$. λProlog 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 λProlog serves as an aid to the programmer rather than an added layer of syntax. Lists and integers are handled the same way in λProlog as they are in Prolog. Unlike Prolog, however, λProlog 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 λProlog subgoal `truth is X < Y` in place of the Prolog subgoal `X < Y`.

We have identified several logic program schemata that serve as prototype logic programs for list processing [11]. Each of these schemata has two arguments, an input list and a result. Although many logic programs can be used with various modes, we assume a given mode for each of our logic programs. In addition to recursive list processing schemata, it is also possible to define a set of recursive natural number programs which also have two arguments. One of the largest classes of list processing programs is the class of global list processing programs which includes all those list processing programs that process all elements of the input list (i.e., the entire input list is reduced). Global list processing programs are captured by the `reduceList/2` schema:

```
reduceList [] Result :-
    Base Result.
reduceList [H|T] Result :-
    reduceList T R, Constructor H R Result.
```

Global natural number processing programs are captured by the `reduce-Number/2` schema:

```
reduceNumber 0 Result :-
    Base Result.
reduceNumber N Result :-
    M is N - 1, reduceNumber M R,
    Constructor N R Result.
```

The `reduceList/2` and `reduceNumber/2` schemata can be generalized to include all singly-recursive reduction programs by incorporating the termination condition with the base case value computation and permitting an arbitrary destructor:

```
reduce Input Result :-
    Base Input Result.
reduce Input Result :-
    Destructor Input H T, reduce T R,
    Constructor H R Result.
```

Some explanation of the `reduce/2` schema is in order. It has two arguments and contains three predicate variables. The first argument is the primary input and the second argument is the primary output. The primary input and output can be either simple or structured terms, but they are both first-order terms. The three predicate variables represent arbitrary λProlog predicates. The predicate variable `Destructor` defines the process for destructing the input. The predicate variable `Constructor` defines the process for constructing the output. The other predicate variable, `Base`, is used to define the terminating condition, defining both the process to identify the terminating condition and the process which defines how to construct the output for the terminating condition. An example should help clarify `reduce/2`.

Consider the `factorial/2` program. For an arbitrary query `factorial A B`, the primary input is `A` and primary output is `B`. The destructor predicate decrements the input by one. This process can be defined with the anonymous predicate `(X\Y\Z\(Z is X - 1, Y = X))`. The constructor predicate for `factorial/2` multiplies the current input by the factorial of one less than the current input and can be defined with the anonymous predicate `(X\Y\Z\(Z is X * Y))`. As can be seen in the base case clause of the definition of `factorial/2`, the terminating condition occurs whenever the input becomes one and the terminating output value should be one in this case. This process can be defined with the anonymous predicate `(X\Y\(X = 0, Y = 1))`. Combining all this together, we can produce a program for `factorial/2` by instantiating the predicate variables in `reduce/2`:

```
factorial N Result :-
    (X\Y\(X = 0, Y = 1)) N Result.
factorial N Result :-
    (X\Y\Z\(Z is X - 1, Y = X)) N C M,
    factorial M R,
    (X\Y\Z\(Z is X * Y)) C R Result.
```

Furthermore, since `factorial/2` is a global natural number processing program, it is also possible to produce a program for it by instantiating the predicate variables in `reduceNumber/2`:

```
factorial 0 Result :-
    (X\(X = 1)) Result.
factorial N Result :-
    M is N - 1, factorial M R,
    (X\Y\Z\(Z is X * Y)) N R Result.
```

Now consider `sum/2` again. For an arbitrary query `sum A B`, the primary input is `A` and primary output is `B`. 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\Y\Z\(X = [Y|Z]))`. The constructor predicate for `sum/2` computes the summation by adding the current element to the sum of the rest of the list and can be defined with the anonymous predicate `(X\Y\Z\(Z is X + Y))`. As can be seen in the base case clause of the definition of `sum/2`, the terminating condition occurs whenever the input list becomes empty and the terminating output value should be `0`. This process can be defined with the anonymous predicate `(X\Y\(X = [], Y = 0))`. Combining all this together, we can produce a program for `sum/2` by instantiating the predicate variables in `reduce/2`:

```
sum List Result :-
    (X\Y\(X = [], Y = 0)) List Result.
sum List Result :-
    (X\Y\Z\(X = [Y|Z])) List H T, sum T R,
    (X\Y\Z\(Z is X + Y)) H R Result.
```

Furthermore, since `sum/2` is a global list processing program, it is also possible to produce a program for it by instantiating the predicate variables in `reduceList/2`:

```
sum [] Result :-
    (X\(X = 0)) Result.
sum [H|T] Result :-
    sum T R, (X\Y\Z\(Z is X + Y)) H R Result.
```

In order to capture programs like `position/3` which simultaneously reduce both a list and a natural number, we need to introduce another logic program schemata. The `reduceLN/3` schema captures programs which simultaneously reduce a list and a number. In addition to capturing `position/3`, the `reduceLN/3` schema also captures programs like `take/3` and `drop/3` which keep or remove the first n elements, respectively. The `reduceLN/3` schema looks like:

```
reduceLN [] N Result.
reduceLN L N Result :-
    L = [H|T], M is N - 1, reduceLN T M R,
    ((N = 0, Base L Result); Constructor H R Result).
```

An example of `reduceLN/3` schema is the `position/3` program:

```
position 0 [E|T] E.
position N [H|T] E :- M is N - 1, position M T E.
```

If we instantiate `Base` to `(X\Y\(sigma Z\(X = [Y|Z])))` and `Constructor` to `(X\Y\Z\(Z = Y))` then we can produce `position/3` from `reduceLN/3` assuming that the case of requesting the nth element from a list of less than n elements is a ill-posed query:

```
position [] N E.
position L N E :-
    L = [H|T], M is N - 1, position T M R,
    ((N = 0, (X\Y\(sigma Z\(X = [Y|Z]))) L E);
     (X\Y\Z\(Z = Y)) H R E).
```

The class of reduction programs presented so far share a common destructor. As such, we can refer to the schemata defined so far as *destructor-specific* schemata. It is also possible to have *constructor-specific* schemata. Two of the most popular higher-order programming programs are `map/3` and `filter/3`. Mapping and filtering programs are a subclass of reduction programs that also share a common constructor. 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 [] [] P.
map [H|T] Result P :-
    map T R P,
    (X\Y\Z\(sigma W\(P X W, Z = [W|Y]))) H R Result.
```

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

```
doubleAll List Result :-
     map List Result (X\Y\(Y is 2 * X)).
```

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` in λProlog as follows:

```
filter [] [] P.
filter [H|T] Result P :-
    filter T R P,
    (X\Y\Z\((P X, Z = [X|Y]); Z = Y)) H R Result.
```

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

```
positivesOnly List Result :-
    filter List Result (X\(truth is X > 0)).
```

It is possible to consider the mapping constructor and the filtering constructor as special cases of the following constructor:

```
(P X XX, Z = [XX|Y]); Z = Y
```

Notice that this constructor has the additional disjunctive subgoal `(Z = Y)` which is never invoked for mapping programs and it only captures filtering constructors if we rewrite the filtering constructor to add an additional argument to its filtering predicate:

```
(A\B\(P A, A = B))
```

which represents the mapped element. Now we can define the following special case of `reduceList/2` for mapping/filtering programs:

```
mapList [] [].
mapList [H|T] Result :-
    mapList T R,
    ((P H XX, Result = [XX|R]); Result = R).
```

It is important to note that the schemata presented in this section are very robust, capturing a large class of programs which also includes `reverse/2`, `insertion-sort/2`, `product/2`, `prefix/2`, and many others. We can extend each of these schemata to capture additional logic programs. For example, we can extend `reduce/2` to capture other programs like `append/3` and `count/3`. This is described in the next section.

3 Extensions to Logic Program Schemata

The `reduce/2` schema captures a large group of logic programs, but there is still a large group of logic programs that it is unable to capture. One of the major differences between logic programs is the number of arguments. In addition to instantiating predicate variables in logic program schemata to produce logic programs, it is also possible to extend program schemata to include additional arguments. Vasconcelos and Fuchs [16] handle this in their enhanced schema language by introducing argument vectors and having positional indicators to ensure specified arguments occur in the same position across terms. We propose handling varying number of arguments by extending our logic program schemata. There are several types of argument extension that can be applied to the `reduce/2` schema, corresponding to adding arguments to the predicate variables in `reduce/2`. We can extend the `reduce/2` schema to add an additional argument to the `Base` predicate:

```
reduceB Input Result ArgBase :-
    Base Input Result ArgBase.
reduceB Inputt Result ArgBase :-
    Destructor Input H T,
    reduceB T R ArgBase,
    Constructor H R Result.
```

An example of this type of extension is the creation of `append/3` from `prefix/2`:

```
prefix [] L.
prefix [H|T] [H|L] :- prefix T L.
```

The `prefix/2` predicate succeeds if its primary list is a prefix of its other list. The `prefix/2` predicate can be extended by the adding a new argument which represents the second list (i.e., the list that is to be appended to the primary list). If we make the new `Base` predicate unify its arguments then we get `append/3`:

```
append [] L List :- (X\Y\(X = Y)) L List.
append [H|T] [H|L] List :- append T L List.
```

Another argument extension that can be applied to the `reduce/2` schema is to add an argument to the `Constructor` predicate:

```
reduceC Input Result ArgCons :-
    Base Input Result.
reduceC Input Result ArgCons :-
    Destructor Input H T,
    reduceC T R ArgCons,
    Constructor H R Result ArgCons.
```

An example of this type of extension is the creation of `count/3` from `length/2`:

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

If we make the new `Constructor` predicate increment the count only when the head of the list satisfies a predicate given by the newly added argument then we can produce `count/3`:

```
count [] 0 P.
count [H|T] C P :-
    count T R P,
    (W\X\Y\Z\(P W, Y is X + 1); Y = X) H R C P.
```

Note that the additional argument on the `Constructor` for `count/3` serves as a “filter” which tests the appropriateness of the input element. For such programs, it would be possible incorporate the “filter” into the `Destructor` (i.e., it is possible to extend `reduce/2` by adding an additional argument to the `Destructor` predicate):

```
count [] 0 P.
count List C P :-
    (W\X\Y\Z\(remove W X Y Z)) List H T P,
    count T R P, C is R + 1.

remove [A|B] A B P :- P A.
remove [H|T] A B P :- remove T A B P.
```

which is an example of the use of `reduceD/3`:

```
reduceD Input Result ArgDest :-
    Base Input Result.
reduceD Input Result ArgDest :-
    Destructor Input H T ArgDest,
    reduceD T R ArgDest,
    Constructor H R Result.
```

The purpose of these semantics-altering extensions that enable the addition of arguments is to widen the applicability of the semantics-preserving schema transformations. Any transformation that is applicable to `reduce/2` is also applicable to `reduceB/3`, `reduceC/3`, and `reduceD/3`. There are two types of semantics-preserving extensions that can be applied to logic program schemata to produce equivalent logic program schemata: application of programming techniques and combination (or merging) and connection of logic program schemata. The first type of semantics-preserving extension is the application of programming techniques to logic program schemata. Programming techniques have been studied fairly extensively and a number of commonly occurring programming practices have been identified. One popular programming technique is the introduction of an accumulator, enabling the composition of the output from the right rather than from the left. Given that a program unifies with the `reduce/2` schema, we can transform the program by instantiating the following `reduceAcc/2` schema with the same `Base` and `Constructor` predicates:

```
    reduceAcc Input Result :-
        Base Dummy Acc,
        reduceAcc2 Input Result Acc.
    reduceAcc2 Input Result Result :-
        Base Input Dummy.
    reduceAcc2 Input Result Acc :-
        Destructor Input H T, Constructor Acc H A,
        reduceAcc2 T Result A.
```

as long as `Constructor` is an associative predicate. As an example, consider `sum/2` again. We can produce the more efficient (tail recursive) accumulator implementation of `sum/2` by instantiating this `reduceAcc/2` schema:

```
    sum List Result :-
        (X\(X = 0)) Acc, sum2 List Result Acc.
    sum2 [] Result Result.
    sum2 [H|T] Result Acc :-
        (X\Y\Z\(Z is X + Y)) Acc H A, sum2 T Result A.
```

A similar type of transformation is possible for programs captured by the `reduceLN/3` schema. The `reduceLN/3` is a forward processing schema which reduces its list from the front and reduces its integer from some maximum value down to 0. An equivalent backward processing schema which continues to reduce its list from the front but reduces its integer up from 0 to the maximum rather than from the maximum down to 0 looks like:

```
    reduceUpLN L N Result :-
        Max = (X\(X = N)),
        reduceUpLN2 L 0 Max Result.
    reduceUpLN2 [] N Max Result.
    reduceUpLN2 L N Max Result :-
        L = [H|T], M is N + 1, reduceUpLN2 T M Max R,
        ((Max N, Base L Result); Constructor H R Result).
```

As an example, consider `position/3` again. We can transform the standard forward processing program given in the previous section to a backward processing program using the `reduceLN/3` ⇒ `reduceUpLN/3` transformation producing the following implementation of `position/3`:

```
    position L N E :-
        Max = (X\(X = N)),
        position2 L 0 Max E.
```

```
position2 [] N Max Result.
position2 L N Max E :-
    L = [H|T], M is N + 1, position2 T M Max R,
    ((Max N, (X\Y\(sigma Z\(X = [Y|Z]))) L E);
     (X\Y\Z\(Z = Y)) H R E).
```

The second type of semantics-preserving logic program extension is the combination (or merging) of logic program schemata. The idea is to merge two logic program schemata whenever they have a common argument. Combination schema transformations are listed in the following table.

Initial Schemata		**Combination Schema**
`mapList/2`	`reduceList/2`	`mapReduceList/2`
`reduceList/2`	`reduceList/2`	`reduceListList/3`
`reduceLN/3`	`reduceLN/3`	`reduceLNLN/4`
`reduceListAcc/2`	`reduceUpLN/3`	`reduceConnect/2`
`reduceListAcc/2`	`reduceUpLNLN/4`	`reduceConnect/3`

Probably the most obvious combination schema transformation is to combine logic program schemata which have a common primary input. The `reduceList/2 + reduceList/2` ⇒ `reduceListList/3` transformation combines two `reduce-List/2` schemata that have a common primary input.

```
reduceList [] Result1 :-
 Base1 Result1.
reduceList [H|T] Result1 :-
 reduceList T R,
 Constructor1 H R Result1.
```

```
reduceList [] Result2 :-
 Base2 Result2.
reduceList [H|T] Result2 :-
 reduceList T R,
 Constructor2 H R Result2.
```

$$\Downarrow$$

```
reduceListList [] Result1 Result2 :-
 Base1 Result1, Base2 Result2.
reduceListList [H|T] Result1 Result2 :-
 reduceListList T R1 R2,
 Constructor1 H R1 Result1,
 Constructor2 H R2 Result2.
```

An example of the use of the `reduceList/2 + reduceList/2` ⇒ `reduceListList/3` transformation is the creation of a singly-recursive implementation of the `average/2` predicate from the following straightforward solution:

```
average List Average :-
    length List Length,
    sum List Sum,
    Average is Sum / Length.

length [] 0.
length [H|T] Length :- length T R, Length is R + 1.

sum [] 0.
sum [H|T] Sum :- sum T R, Sum is R + H.
```

Applying the `reduceList/2 + reduceList/2 ⇒ reduceListList/3` transformation to this program produces the following implementation of `average/2`:

```
average List Average :-
    average2 List Length Sum,
    Average is Sum / Length.
average2 [] 0 0.
average2 [H|T] Length Sum :-
    average2 T L S, Length is L + 1, Sum is S + H.
```

Because the `reduceListList/3` schema was created by combining two global list processing schemata which share a common primary input and have distinct outputs, the same process can also be used to combine two accumulated implementations of global list processing schemata (or even one of each). It is also possible to combine two `reduceLN/3` schemata with the `reduceLN/3 + reduceLN/3 ⇒ reduce-LNLN/4` transformation.

```
reduceLN [] N Result1.
reduceLN L N Result1 :-
 L = [H|T], M is N - 1,
 reduceLN T M R,
 ((N = 0, Base1 L Result1);
 Constructor1 H R Result1).
```

```
reduceLN [] N Result2.
reduceLN L N Result2 :-
 L = [H|T], M is N - 1,
 reduceLN T M R,
 ((N = 0, Base2 L Result2);
 Constructor2 H R Result2).
```

$$\Downarrow$$

```
reduceLNLN [] N Result1 Result2.
reduceLNLN L N Result1 Result2 :-
 L = [H|T], M is N - 1,
 reduceLNLN T M R1 R2,
 ((N = 0, Base1 L Result1, Base2 L Result2);
  (Constructor1 H R1 Result1),
   Constructor2 H R2 Result2)).
```

An example of the use of the `reduceLN/3 + reduceLN/3 ⇒ reduce-LNLN/4` transformation is the splitting of a list into two sublists, one sublist which contains the first n elements of the list and a second sublist which contains all but the first n elements of the list. This task can be accomplished using the well-known `take/3` and `drop/3` predicates:

```
    takedrop List N FirstPart ButFirstPart :-
        take List N FirstPart, drop List N ButFirstPart.

    take [] N [].
    take L N Result :-
        L = [H|T], M is N - 1, take T M R,
        ((N = 0, (X\Y\(Y = [])) L Result);
         (X\Y\Z\(Z = [X|Y])) H R Result).

    drop [] N [].
    drop L N Result :-
        L = [H|T], M is N - 1, drop T M R,
        ((N = 0, (X\Y\(Y = X)) L Result);
         (X\Y\Z\(Z = Y)) H R Result).
```

Applying the `reduceLN/3 + reduceLN/3 ⇒ reduceLNLN/4` transformation to `takedrop/4` produces the following implementation:

```
    takedrop [] N [] [].
    takedrop L N Res1 Res2 :-
        L = [H|T], M is N - 1, takedrop T M R1 R2,
        ((N = 0,
          (X\Y\(Y = [])) L Res1, (X\Y\(Y = X)) L Res2);
         (X\Y\Z\(Z = [X|Y])) H R1 Res1),
          (X\Y\Z\(Z = Y)) H R2 Res2)).
```

The `reduceLN/3 + reduceLN/3 ⇒ reduceLNLN/4` transformation has a corresponding `reduceUpLN/3 + reduceUpLN/3 ⇒ reduceUpLNLN/4` transformation which enables the combination of two backward processing `reduce-LN/3` programs. Although the transformational system of Vasconcelos and Fuchs [16] supports the combination of logic program schemata which share a common input, their system currently does not support any transformations which connect two logic programs where the output of one schema is the input to another schema. We can support such transformations by connecting `mapList/2` and `reduceList/2` where the mapping/filtering program maps a predicate across the elements of the input list and this mapped list is then reduced. The `mapList/2 + reduceList/2 ⇒ mapReduce-List/2` transformation combines the `mapList/2` schema with the `reduceList/2` schema where the list that is produced by the mapping/filtering program is the input to the reduction program.

```
mapList [] [].
mapList [H|T] TempRes :-
 mapList T R,
 ((P H XX, TempRes = [XX|R]);
  TempRes = R).
```

```
reduceList [] Result :-
 Base Result.
reduceList [H|T] Result :-
 reduceList T R,
 Constructor H R Result.
```

$$\Downarrow$$

```
mapReduceList [] Result :-
 Base Result.
mapReduceList [H|T] Result :-
 mapReduceList T R,
 ((P H XX, Constructor XX R Result); Result = R).
```

As an example, consider the following straightforward solution to counting the number of positive elements in a list by filtering out the non-positive elements (using `positivesOnly/2`) and then counting the number of elements in the filtered list (using `length/2`):

```
positiveCount List Result :-
    positivesOnly List X, length X Result.

positivesOnly [] [].
positivesOnly [H|T] Result :-
    positivesOnly T R,
    ((truth is H > 0, XX = H, Result = [XX|R]);
     Result = R).

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

Applying the `mapList/2 + reduceList/2 ⇒ mapReduceList/2` transformation to this `positiveCount/2` program produces the following implementation:

```
positiveCount [] 0.
positiveCount [H|T] Result :-
    positiveCount T R,
    (((truth is H > 0, XX = H), Result is R + 1);
     Result = R).
```

Another class of combination schema transformations enable the connection of two logic program schemata that share a common primary input and the result of one schema is an additional input to the other schema. We have identified two schema transformations for this type of combination schema transformation: `reduceListAcc/2 + reduceUpLN/2 ⇒ reduceConnect/2` and `reduceListAcc/2 + reduce-UpLNLN/3 ⇒ reduceConnect/3.`

```
reduceListAcc L N :-
 Base1 A,
 reduceListAcc2 L N A.
reduceListAcc2 [] N N.
reduceListAcc2 L N A :-
 L = [H|T],
 Constructor1 A H B,
 reduceListAcc2 T N B.
```

```
reduceUpLN L N Result :-
 Max = (X\(X = N)),
 reduceUpLN2 L 0 Max Result.
reduceUpLN2 [] N Max Result.
reduceUpLN2 L N Max Result :-
 L = [H|T], M is N + 1,
 reduceUpLN2 T M Max R,
 ((Max N, Base2 L Result);
  Constructor2 H R Result).
```

$$\Downarrow$$

```
reduceConnect List Result :-
 Base1 Acc, Max = (X\(X = C)),
 reduceC2 List Acc A 0 Max Result.
reduceC2 [] Acc Acc N Max Result :-
 Connect Acc C, Max = (X\(X = C)).
reduceC2 List Acc RL N Max Result :-
 List = [H|T], M is N + 1, Constructor1 Acc H A,
 reduceC2 T A RL M Max R,
 ((Max N, Base2 List Result); Constructor2 H R Result).
```

As an example of the `reduceAccList/2 + reduceUpLN/2 ⇒ reduce-Connect/2` transformation, consider finding the middle element in an arbitrary list. A straightforward solution to this problem is to count the number of elements in the list (using `length/2`), divide this count by two, and then use this value to find the middle element (using `position/3`):

```
middle List Middle :-
    length List Length, Half is Length div 2,
    position List Half Middle.

length [] 0.
length [H|T] Length :- length T R, Length is R + 1.

position [] N E.
position L N E :-
    L = [H|T], M is N - 1, position T M R,
    ((N = 0, (X\Y\(sigma Z\(X = [Y|Z]))) L E);
     (X\Y\Z\(Z = Y)) H R E).
```

The first step in the transformation of `middle/2` is to transform `length/2` to an accumulated implementation using the `reduce/2 ⇒ reduceAcc/2` tranformation producing the following implementation of `length/2`:

```
length List Length :- Acc = 0, length2 List Length Acc.
length2 [] Len Len.
length2 [H|T] Len Acc :- A is Acc + 1, length2 T Len A.
```

The next step is to transform `position/3` from a forward processing program to a backward processing program using the `reduceLN/3` ⇒ `reduceUpLN/3` transformation producing the following implementation of `position/3`:

```
position L N E :-
    Max = (X\(X = N)),
    position2 L 0 Max E.
position2 [] N Max Result.
position2 L N Max E :-
    L = [H|T], M is N + 1, position2 T M Max R,
    ((Max N, (X\Y\(sigma Z\(X = [Y|Z]))) L E);
     (X\Y\Z\(Z = Y)) H R E).
```

The final step in the transformation is to apply the `reduceListAcc/2 + reduceUpLN/3` ⇒ `reduceConnect/2` transformation to combine `length/2` and `position/3` to produce the following implementation of `middle/2`:

```
middle List Middle :-
    Acc = 0, Max = (X\(X = Half)),
    middle2 List Acc Length 0 Max Middle.
middle2 [] Length Length AH Max Middle :-
    Half is Length div 2, Max = (X\(X = Half)).
middle2 [H|T] AL Length AH Max Middle :-
    NL is AL + 1, NH is AH + 1,
    middle2 T NL Length NH Max Mid,
    ((Max AH, Middle = H); Middle = Mid).
```

The final combination schema transformation that we consider is the `reduce-ListAcc/2 + reduceUpLNLN/4` ⇒ `reduceConnect/3` transformation.

```
reduceListAcc L N :-
 Base1 A,
 reduceListAcc2 L N A.
reduceListAcc2 [] N N.
reduceListAcc2 L N A :-
 L = [H|T],
 Constructor1 A H B,
 reduceListAcc2 T N B.
```

```
reduceUpLNLN L N R1 R2 :-
 Max = (X\(X = N)),
 reduceUpLNLN2 L 0 Max R1 R2.
reduceUpLNLN2 [] N Max R1 R2.
reduceUpLNLN2 L N Max R1 R2 :-
 L = [H|T], M is N + 1,
 reduceUpLNLN2 T M Max S1 S2,
 ((Max N, Base21 L R1, Base22 L R2);
  (Constructor21 H S1 R1),
   Constructor22 H S2 R2)).
```

$$\Downarrow$$

```
reduceConnect List R1 R2 :-
 Base1 Acc, Max = (X\(X = C)),
 reduceC2 List Acc A 0 Max R1 R2.
reduceC2 [] Acc Acc N Max R1 R2 :-
 Connect Acc C, Max = (X\(X = C)).
reduceC2 List Acc RL N Max R1 R2 :-
 List = [H|T], M is N + 1, Constructor1 Acc H A,
 reduceC2 T A RL M1 M2 Max S1 S2,
 ((Max N, Base21 List R1, Base22 List R2);
  (Constructor21 H S1 R1), Constructor22 H S2 R2)).
```

As an example of the `reduceListAcc/2 + reduceUpLNLN/4 ⇒ reduceConnect/3` transformation, consider the task of splitting a list of elements in half. We can do this in λProlog by invoking three reduction programs:

```
splitlist List FirstHalf LastHalf :-
    length List N, M is N div 2,
    take List M FirstHalf, drop List M LastHalf.

length [] 0.
length [H|T] Length :- length T R, Length is R + 1.

take [] N [].
take L N Result :-
    L = [H|T], M is N - 1, take T M R,
    ((N = 0, (X\Y\(Y = [])) L Result);
     (X\Y\Z\(Z = [X|Y])) H R Result).

drop [] N [].
drop L N Result :-
    L = [H|T], M is N - 1, drop T M R,
    ((N = 0, (X\Y\(Y = X)) L Result);
     (X\Y\Z\(Z = Y)) H R Result).
```

In order to apply the `reduceUpLN/3 + reduceUpLN/3 ⇒ reduceUp-LNLN/4` transformation to `take/3` and `drop/3` we must first transform them from forward processing programs to backward processing programs using the `reduceLN/3 ⇒ reduceUpLN/3` transformation:

```
take L N Result :-
    Max = (X\(X = N)),
    take2 L 0 Max Result.
take2 [] N Max [].
take2 L N Max Result :-
    L = [H|T], M is N + 1, take2 T M Max R,
    ((Max N, (X\Y\(Y = [])) L Result);
     (X\Y\Z\(Z = [X|Y])) H R Result).
```

```
drop L N Result :-
    Max = (X\(X = N)),
    drop2 L 0 Max Result.
drop2 [] N Max [].
drop2 L N Max Result :-
    L = [H|T], M is N + 1, drop2 T M Max R,
    ((Max N, (X\Y\(Y = X)) L Result);
     (X\Y\Z\(Z = Y)) H R Result).
```

Now we can combine `take/3` and `drop/3` by applying the `reduceUpLN/3 + reduceUpLN/3 ⇒ reduceUpLNLN/4` transformation:

```
takedrop L N Res1 Res2 :-
    Max = (X\(X = N)),
    takedrop2 L 0 Max Res1 Res2.
takedrop2 [] N Max [] [].
takedrop2 L N Max Res1 Res2 :-
    L = [H|T], M is N + 1,
    takedrop2 T M Max R1 R2,
    ((Max N,
      (X\Y\(Y = [])) L Res1, (X\Y\(Y = X)) L Res2));
     (X\Y\Z\(Z = [X|Y])) H R1 Res1),
      (X\Y\Z\(Z = Y)) H R2 Res2)).
```

After transforming `length/2` to its accumulated implementation as we did in the previous example, we can apply the `reduceListAcc/2 + reduceUpLNLN/4 ⇒ reduceConnect/3` transformation producing the following implementation of `splitlist/3`:

```
splitlist L Res1 Res2 :-
    Acc = 0, Max = (X\(X = Half)),
    splitlist2 L Acc N 0 Max Res1 Res2.
splitlist2 [] Length Length N Max [] [] :-
    Half is Length div 2, Max = (X\(X = Half)).
splitlist2 L AL Length AH Max Res1 Res2 :-
    L = [H|T], NL is AL + 1, NH is AH + 1,
    splitlist2 T NL Length NH Max R1 R2,
    ((Max AH, (X\Y\(Y = [])) L Res1,
      (X\Y\(Y = X)) L Res2);
     ((X\Y\Z\(Z = [X|Y])) H R1 Res1,
      (X\Y\Z\(Z = Y)) H R2 Res2)).
```

Logic program schemata and logic program schema transformations can be used to help in program development by enabling the programmer to produce a simple straightforward solution to the problem and then transform that solution into an efficient one by applying a set of program transformations.

4 Conclusion

We have proposed an extensible schema-based logic program transformation system as an improvement to the traditional schema-based meta-language approaches. In this system, we have defined a set of extensible logic program schemata in λProlog. Because λProlog is a higher-order logic programming language, it can be used as the representation language for both the logic programs and the extensible logic program schemata. In addition to the instantiation of predicate variables, extensible logic program schemata can be extended by applying standard programming techniques (e.g., accumulating results), introducing additional arguments (e.g., a second list to append to the end of the primary list), combining logic program schemata which share a common primary input, and connecting logic program schemata which are connected via a result of one schema being an input to the other schema. These extensions increase the robustness of logic program schemata and enhance traditional schema-based transformational systems.

References

[1] D. Barker-Plummer. Cliché Programming in Prolog. In M. Bruynooghe, editor, *Proceedings of the 2nd Workshop on Meta-Programming in Logic*, Leuven, Belgium, pages 247-256, 1990.

[2] 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.

[3] P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.

[4] P. Flener and Y. Deville. Logic Program Synthesis from Incomplete Specifications. *Journal of Symbolic Computation*, 15: 775-805, 1993.

[5] P. Flener and Y. Deville. Logic Program Transformation Through Generalization Schemata. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, Utrecht, The Netherlands, pages 171-173, Springer-Verlag, 1995.

[6] N.E. Fuchs and M.P.J. Fromhertz. Schema-Based Transformations of Logic Programs. In T.P. Clement and K. Lau, editors, *Proceedings of the 1st 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: 173-190, 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 an Automated Program Debugger. *Journal of Artificial Intelligence in Education*, 5: 255-278, 1994.

[11] T.S. Gegg-Harrison. Representing Logic Program Schemata in λProlog. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, Kanagawa, Japan, pages 467-481, MIT Press, 1995.

[12] A. Hamfelt and J. Fischer Nilsson. Declarative Logic Programming with Primitive Recursive Relations on Lists. In M. Maher, editor, *Proceedings of the 13th Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, pages 230-243, MIT Press, 1996.

[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] E. Marakakis and J.P. Gallagher. Schema-Based Top-Down Design of Logic Programs using Abstract Data Types. In L. Fribourg and F. Turini, editors, *Proceedings of the 4th International Workshops on Logic Program Synthesis and Transformation and Meta-Programming in Logic*, Pisa, Italy, pages 138-153, Springer-Verlag, 1994.

[15] G. Nadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, Washington, pages 810-827, MIT Press, 1988.

[16] W.W. Vasconcelos and N.E. Fuchs. An Opportunistic Approach for Logic Program Analysis and Optimisation Using Enhanced Schema-Based Transformations. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, Utrecht, The Netherlands, pages 174-188, Springer-Verlag, 1995.