The Journal of Functional and Logic Programming

The MIT Press

Volume 2000, Article 4
31 March 2000

ISSN 1080–5230. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493, USA; (617)253-2889; *journals-orders@mit.edu, journals-info @mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/JFLP/*
- *gopher.mit.edu*
- *ftp://mitpress.mit.edu/pub/JFLP*

The Journal of Functional and Logic Programming is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.

Electronic Mail: *jflp.request@ls5.informatik.uni-dortmund.de*

Stepwise Enhancement and Higher-Order Programming in Prolog

Lee Naish
lee@cs.mu.oz.au

Leon Sterling
leon@cs.mu.oz.au

31 March 2000

Abstract

This paper presents two views of stepwise enhancement, one a pragmatic syntax-based approach and the other a semantic approach based on higher-order functions and relating to shape and polytypism. The approaches are outlined, and the perhaps surprisingly close relationship between the two described. By combining the advantages of both approaches, it is shown how more code in both functional and logic programming languages can be constructed in a systematic way. We describe a prototype system that allows higher-order predicate definitions to be produced automatically from type definitions or Horn clause skeletons and relate some experiences in using higher-order programming in Prolog.

1 Motivation

In the last few years, interest in systematic methods for the construction of Prolog programs has significantly increased. See, for example, Gegg-Harrison [GH95]; Power and Sterling [PS90]; Kirschenbaum, Michaylov, and Sterling [KMS96]; Sterling and Yalçinalp [SY96]; and Vasconcelos and Fuchs [VF95]. These papers have built on previous, less-systematic approaches to Prolog programming, implicit in text books and shared code. The progression of approaches was discussed in Naish and Sterling [NS98].

Two audiences can benefit from systematic approaches: novice programmers who need guidance to create code of reasonable quality and experienced

programmers who want to increase their productivity. The needs of these two groups are quite different. Novices can learn more easily from concrete examples and do not have the experience to recognise patterns and abstractions. Experienced programmers can benefit from using higher levels of abstraction in powerful tools.

We claim that both audiences can be addressed through the method of stepwise enhancement. This method can be explained directly in terms of programming techniques applied to simple programs and can be given a more theoretical basis in terms of higher-order functions. In this paper we review stepwise enhancement, sketch how to capture an enhancement as a higher-order function adapted from `foldr`, and then sketch how individual enhancements are specialisations of the particular `foldr` predicate. We show how this is closely related to the new ideas on shape and polytypism being discussed in the functional programming community (see Jay and Cockett [JC94]; Jay [Jay95]; Bellé, Jay, and Moggi [BJM96]; Jeuring and Jansson [JJ96]; Jansson and Jeuring [JJ97]). We go on to generalise this work in several ways, utilising key features of logic programming, such as nondeterminism and flexible modes, and we show how `foldl` can also be adapted. This leads to a proposal for a set of software tools to automatically produce Prolog procedures from type definitions. We describe a prototype implementation and relate our initial experiences using these tools and using the higher-order programming style with Prolog.

The current work was sparked by the challenge to use the higher-order approach to explain a "complicated" program, the rule interpreter described in Section 17.4 of the second edition of *The Art of Prolog* (Sterling and Shapiro [SS94]). In explaining the program, a new method was formulated: The final program is built around an output type rather than an input type. In this paper we give another example of this technique, using a different interpreter.

What has emerged is a better understanding of how types can drive program development, and how Naish and Sterling's views of systematic program construction via stepwise enhancement are complementary. The work relates to recent, exciting advancements in the functional programming community concerning shape, which sets our views on program development in a broader context. These insights have prompted the development of prototype software tools for partially automating coding based on types and skeletons. This paper describes our experiences with higher-order programming in Prolog and the use of these tools. The initial results show that a significant

increase in programming productivity is possible.

2 Program construction using stepwise enhancement

The method of stepwise enhancement (Lakhotia [Lak89]) was originally conceived as an adaptation of stepwise refinement to Prolog. It was advocated as a way to systematically construct Prolog programs, which exploits Prolog's high-level features. The key idea underlying stepwise enhancement is to visualise a program or solution in terms of its central control flow, or skeleton, and techniques that perform computations while the control flow of the skeleton is followed. Techniques can be developed independently and combined automatically using the method of composition.

The most common data structure for logic programs is the list, and many programs are based on skeletons for traversing lists. A tutorial example of using stepwise enhancement to develop a simple program is given in Chapter 13 of Sterling and Shapiro [SS94]. In this section we give the basic list-processing program as Program 1 for reference, and a (slightly) more elaborate example with binary trees.

Program 1 *A skeleton for list traversal or definition of lists:*

```
is_list([]).
is_list([X|Xs]) :- is_list(Xs).
```

Programs 2a and 2b are skeleton programs for traversing binary trees with values only at leaf nodes. (Later we also use M-way trees with data in internal nodes.) Program 2a, the left-hand program, does a complete traversal of the tree, while Program 2b, the right-hand program, traverses a single branch of the tree. Note that Program 2a can be viewed as a type definition of trees.

Program 2 *Skeletons for traversing a tree (a, b):*

```
is_tree(leaf(X)).
is_tree(tree(L,R)) :-
    is_tree(L),
    is_tree(R).
```

```
branch(leaf(X)).
branch(tree(L,R)) :- branch(L).
branch(tree(L,R)) :- branch(R).
```

Techniques capture basic Prolog programming practices, such as building a data structure or performing calculations in recursive code. 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 do one or more of the following: rename predicates, add arguments to predicates, add goals to clauses, and add clauses to programs. Unlike skeletons, techniques are not programs but can be conceived as a family of operations that can be applied to a program to produce a program.

A technique applied to a skeleton yields an *enhancement*. An enhancement that preserves the computational behaviour of the skeleton is called an *extension*.

We give examples of techniques. The two most commonly used techniques are calculate and build. They 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. Note that the terminology used in the stepwise enhancement literature tends to be procedural. Despite this, the resulting programs can be viewed in a declarative way and (in some cases) are reversible.

Two typical examples of the application of the calculate technique are given as Programs 3a and 3b. Both are extensions of Program 2a, which traverses a binary tree with values at its leaves. The left-hand program (3a) computes the product of the values of the leaves of the trees. The extra argument in the base case is the value of the leaf node. In the recursive case, the extra goal says that the product of a tree is the product of its left subtree and its right subtree. The predicate `is_tree/1` is renamed to `prod_leaves/2`. The right-hand program (3b), which computes the sum of the leaves, is very similar, the only difference being choice of names and the extra goal.

Program 3 *Extensions of Program 2a using the calculate technique (a, b):*

```
prod_leaves(leaf(X),X).
prod_leaves(tree(L,R),Prod) :-
```

```
sum_leaves(leaf(X),X).
sum_leaves(tree(L,R),Sum) :-
```

```
    prod_leaves(L,LProd),          sum_leaves(L,LSum),
    prod_leaves(R,RProd),          sum_leaves(R,RSum),
    Prod is LProd*RProd.           Sum is LSum+RSum.
```

Two enhancements of the same skeleton share computational behaviour. They can be combined into a single program that takes 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 the functionality of separate functions are combined into a single function. Program 4 is the result of the composition of Programs 3a and 3b.

Program 4 *The composition of two extensions:*

```
prod_sum_leaves(leaf(X),X,X).
prod_sum_leaves(tree(L,R),Prod,Sum) :-
     prod_sum_leaves(L,LProd,LSum),
     prod_sum_leaves(R,RProd,RSum),
     Prod is LProd*RProd,
     Sum is LSum+RSum.
```

A different programming technique uses accumulators. The accumulator-calculate technique adds two arguments to the defining predicate in the skeleton. 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, usually via unification. 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.

Program 5 shows the result of applying the accumulate-calculate technique to the tree traversal program, Program 2a. It computes the sum of the leaves of a binary tree and is comparable to Program 3a. In general, programs written with accumulator techniques run more efficiently than the equivalent program written with calculate and build techniques, due to the way tail recursion is implemented in Prolog.

Program 5 *Extension of Program 2a using the accumulate-calculate technique:*

```
sum_leaves(Tree,Sum) :- accum_sum_leaves(Tree,0,Sum).

accum_sum_leaves(leaf(X),Accum,Sum) :-
    Sum is Accum + X.
accum_sum_leaves(tree(L,R),Accum,Sum) :-
    accum_sum_leaves(L,Accum,Accum1),
    accum_sum_leaves(R,Accum1,Sum).
```

Program 6 is an example of the application of the accumulate-build technique, also applied to Program 2a. It builds an inorder traversal of the leaves of the tree. There is no explicit arithmetic calculation, rather lists built by unification in the base clause. There is one subtlety here. Accumulators build structures in reverse order, and hence the right subtree is traversed before the left subtree in order to have the final list in the correct order. With commutative operations such as addition for integers the order is immaterial.

Program 6 *Extension of Program 2a using accumulate-build:*

```
traversal(Tree,Xs) :- accum_leaves(Tree,[],Sum).

accum_leaves(leaf(X),Accum,[X|Accum]).
accum_leaves(tree(L,R),Accum,Sum) :-
    accum_leaves(R,Accum,Accum1),
    accum_leaves(L,Accum1,Sum),
```

The skeletons and techniques presented in this paper are all taken from Prolog, but stepwise enhancement is equally applicable to other logic programming languages, as discussed in Kirschenbaum, Michaylov, and Sterling [KMS96]. They claim that skeletons and techniques should be identified when a 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($\mathcal{R}$); concurrent logic programming languages such as flat concurrent Prolog and Strand; and higher-order logic program languages, in particular, λ-Prolog (Nadathur and Miller [NM88]).

3 A higher-order approach to programming

Naish [Nai96], and various references included therein, argue for a higher-order approach to programming in Prolog, based on similar techniques that are widely used in functional programming. One of the key steps in this approach is to develop suitable higher-order predicates, which can be used for a whole class of computations over a particular data structure. Modern functional languages have certain data types and higher-order functions built in, for example, the polymorphic type `list(T)` and higher-order function `foldr`, which generalises the common simple recursion used to compute a value from a list. Program 7 demonstrates the use of `foldr` using Prolog syntax in the style of Naish [Nai96].

Program 7 *Using foldr:*

```
:- type list(T) ---> [] ; [T|list(T)].

foldr(F, B, [], B).
foldr(F, B, [A|As], R) :-
    foldr(F, B, As, R1),
    call(F, A, R1, R).

sum(As, S) :- foldr(plus, 0, As, S).
product(As, P) :- foldr(times, 1, As, P).
length(As, L) :- foldr(add1, 0, As, L).
add1(_, TailLen, Len) :- Len is TailLen + 1.
```

In addition to the input list and result, `foldr` has two other arguments. One is the base case: what to return when the end of the list is reached. The other is a function: a predicate in the Prolog context. The predicate takes the head of a list and the result of folding the tail of a list to give the result of folding the whole list. The `call/N` predicates are available as built-ins or library predicates in several Prolog systems. The first argument (a predicate) is called with the additional arguments added. For example, `call(plus(A),R1,R)` is equivalent to `plus(A,R1,R)`, which is true if `A+R1=R`. In Naish [Nai96], an alternative higher-order primitive, `apply/3,` is recommended due to its greater flexibility. In this paper we simply use `call/N`, as it is more widely known.

Examples in Naish [Nai96] show how `foldr` can be used to compute both the sum and product in a single pass by using a pair of numbers for the base case, intermediate results, and final answer. These higher-order definitions can be optimised very effectively; see Sagonas and Warren [SW95], for example. Further examples are given to show how predicates that are analogous to `foldr` can be constructed.

4 Incorporating shape

Recent work on shape (Jay and Cockett [JC94]; Jay [Jay95]; Bellé, Jay, and Moggi [BJM96]) and polytypism (Jeuring and Jansson [JJ96]; Jansson and Jeuring [JJ96]) has formalised the notion that many data types have certain higher-order functions naturally associated with them. For example, `map/3` takes a list and produces another list of the same length. The shape of the output, the list structure, is the same as the shape of the input, and the elements of the lists are related by the function `map/3` applies. The idea of `map/3` can be applied to any algebraic type such as lists and trees, and also arrays and matrices. A generic version of `map/3` applied to a binary tree produces a binary tree of the same shape where the elements of the trees are related by the function `map/3` applies.

Similarly, `foldr` can be generalised to any algebraic type. For lists, a call to `foldr` specifies two things: what should be returned for the empty list and what should be returned for a nonempty list, given the head and the result of folding the tail. For a general algebraic type, we need to specify what should be returned for each constructor in the type, given the arguments of the constructor corresponding to type parameters and the result of folding the arguments that correspond to a concrete type (generally the type being defined recursively).

Consider the `prod_leaves/2` example given earlier as Program 3a. The overall operation is to fold a tree into a single number. We need to define the results of folding terms of the form `leaf(X)` and `tree(L,R)`, given the folded versions of `L` and `R`.

Reconstructing the predicate `is_tree/1` as a definition of the type `bt(T)` and using the approach of Naish [Nai96], we arrive at Program 8: a version of `foldr` for this tree type and corresponding definitions of `prod_leaves/2` and `sum_leaves/2`. In Naish [Nai96] it is assumed that `foldrbt/4` are written by a programmer who has the required degree of insight. It is now clear that

this predicate can be generated *automatically* from a definition of the type. This is discussed in Section 6.

Program 8 *Extensions of Program 2a using foldr:*

```
:- type bt(T) ---> leaf(T) ; tree(bt(T),bt(T)).

foldrbt(TreeP, LeafP, leaf(X), Folded) :-
    call(LeafP, X, Folded).
foldrbt(TreeP, LeafP, tree(L, R), Folded) :-
    foldrbt(TreeP, LeafP, L, FoldedL),
    foldrbt(TreeP, LeafP, R, FoldedR),
    call(TreeP, FoldedL, FoldedR, Folded).

prod_leaves(T, P) :- foldrbt(times, =, T, P).
sum_leaves(T, P) :- foldrbt(plus, =, T, P).
```

5 You take the high road and I'll take the low road

The previous work of the authors sketched above can be seen as taking two different roads for program development, starting from the same place (a type definition) and arriving at the same destination (the enhanced program), as shown in Figure 1.

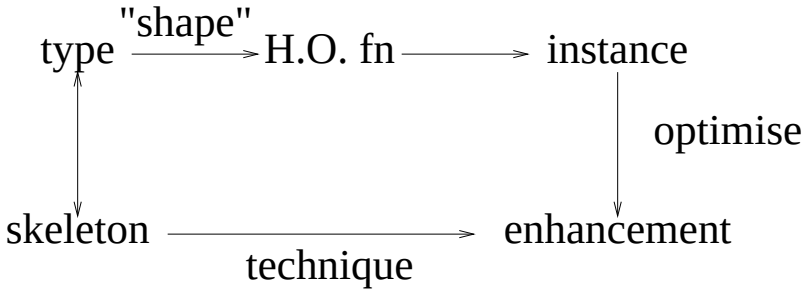

Figure 1: Two roads to enhancement

Sterling suggested the low road in Figure 1, going from the type to the enhancement via the skeleton. Explaining how to travel the low road is simple and does not require very abstract thinking or complex software tools. Such an approach to systematic program construction may be favoured by many programmers.

Naish suggested the high road in Figure 1, writing a version of `foldr` for the given type, using an instance of this `foldr` (a particular call to it) and then optimising, for example, using a partial evaluator. The additional abstraction and concentration on semantics rather than syntax may be favoured by more experienced programmers using more advanced programming environments. The work on shape allows us to automatically obtain the `foldr` definition from the definition of the type.

There is a very simple mapping between algebraic types and a class of logic programs called RUL programs (Yardeni and Shapiro [YS90]). RUL programs only have unary predicates. The argument of each clause head has the top level functor instantiated, but all subterms are unique variables. Clause heads are mutually nonunifiable (thus predicates are deterministic when their arguments are instantiated). The arguments of all calls are variables that occur in the head and nowhere else in the body. Examples of RUL programs are `is_list/1` (Program 1) and `is_tree/1` (Program 2a).

The high road taken by functional programmers is equivalent to starting with a skeleton that is an RUL program and enhancing it with predicates that behave as functions, that is, they are deterministic and always succeed. The theory of functional programming can be used to prove results concerning composition of enhancements, for example, and generally give a theoretical justification for the idea of enhancement.

6 Generalising both approaches

Having found a class of programs for which the high and low roads are equivalent, we can see several ways to generalise both. The low road can have improved handling of nonrecursive clauses and more flexible skeletons. The high road can be made more flexible by eliminating the type, mode, and determinism restrictions inherited from functional programming.

6.1 Goals in base cases

The first (very simple) observation is that enhancements should allow general goals to be added to the base cases of enhancements. For pedagogical reasons, the original presentation classifies enhancements into different categories according to the kind of goal added to recursive clauses and only allows additional unifications to be added to facts. The limited abstraction simplifies learning but also restricts the scope of enhancements. For example, to find the sum of the squares of the leaves of a tree using `foldrbt/4` we could use `foldrbt(plus, square, Tree, SumXX)`, where `square` takes a number and returns its square. The optimised program (or enhanced skeleton) would have a call to `square` in the base case.

6.2 Mutual recursion

The work on stepwise enhancement also does not adequately support mutually recursive skeletons. Similarly, mutually recursive types are not supported in all shape-based tools for functional languages. The *foldr* generalisation we propose supports mutual recursion by transforming a set of procedures rather than a single procedure. The details are provided in the next section and an example is given later in Programs 12 and 13.

6.3 Nondeterministic skeletons

Algebraic types correspond to RUL programs, in which predicates are deterministic and only have a single argument. The stepwise enhancement paradigm has no such restriction; hence nondeterministic skeletons such as `branch/1` (Program 2b) and `connected/2` (Program 9) can be used.

Program 9 *Transitive closure of the edge/2 relation:*

```
connected(A, A).
connected(A0, A) :-
    edge(A0, A1),
    connected(A1, A).
```

As noted in Naish [Nai96], higher-order logic programs can also be nondeterministic, and nondeterministic analogues of `foldr` can be constructed.

A version of `foldr` for paths in a graph was written (using considerable intellectual effort) based on the simple transitive closure procedure `connected/2`, above. The close relationship between "shape" and stepwise enhancement we have uncovered can be used to generalise the transformation from algebraic types (or RUL programs) to `foldr` functions. From an arbitrary skeleton (not necessarily an RUL program), we can generate an appropriate version of `foldr` as follows.

Definition 10 (foldr transformation) *A procedure with A arguments and C clauses leads to a higher-order procedure with $C + A + 1$ arguments. It has C "higher-order" arguments and one additional "result" argument. The recursive calls in the clause bodies have the same higher-order arguments as the head and new variables for their results. Each clause also has an additional* `call/N` *with the higher-order argument for that clause, the variables in the head that did not appear in any recursive body calls, result arguments of the body calls, and the result argument of the head. If this call has only two arguments, then* `call/2` *is replaced by* `=/2`*. (The higher-order argument is simply a term that is returned for the base case; the call to* `=/2` *can then be optimised away, as in Program 7.)*

Mutual recursion can be treated in the same way. Each procedure in a set of (mutually recursive) procedures is transformed as above, where C is the number of clauses in the set of procedures. In the description of the transformation above and other transformations in Section 8, we refer to "recursive" calls. This should be interpreted as including mutual recursion when transforming multiple procedures. Our implementation of the tools introduced in Section 7 has been made more flexible by using an even more liberal definition: A call to any procedure defined in the input to the tool is considered recursive and C is the number of clauses defined in procedures indirectly called by a specified procedure.

For `list/1` and `tree/1`, the results are the `foldr/4` and `foldrbt/4` definitions given in Programs 7 and 8. For `branch/1` and `connected/2` the results are in Program 11. The `foldrcon/5` procedure here is actually more general than the manually constructed version (which had a base case of `V=FB` instead of `call/3`) and can be used in the applications described in Naish [Nai96]. Section 6.1 gives an example where a call is needed in the base case.

Program 11 *Nondeterministic foldr for branch and connected:*

```
foldrb(FL, FR, FB, leaf(X), V) :-
    call(FB, X, V).
foldrb(FL, FR, FB, t(L,R), V) :-
    foldrb(FL, FR, FB, L, V1),
    call(FL, R, V1, V).
foldrb(FL, FR, FB, t(L,R), V) :-
    foldrb(FL, FR, FB, R, V2),
    call(FR, L, V2, V).

foldrcon(FE, FB, A, A, V) :-
    call(FB, A, V).
foldrcon(FE, FB, A0, A, V) :-
    edge(A0, A1),
    foldrcon(FE, FB, A1, A, V1),
    call(FE, A0, V1, V).
```

6.4 Polymorphic types and higher-order skeletons

Monomorphic types such as *list* correspond to first-order skeletons (RUL programs, as we have seen). The work on shape and polytypism uses polymorphic types such as *list*(T), where T is a type parameter. Polymorphic types correspond to higher-order skeletons with additional arguments. A type $t(T1, T2)$ can be mapped to a predicate `t(T1,T2,X)`, which succeeds if X is of type $t(T1, T2)$. If the definition of type t contains the constructor $c(E1, E2)$ (where $E1$ and $E2$ are type expressions), then `t/3` has the clause `t(T1, T2, c(X, Y)) :- call(`$E1$`, X), call(`$E2$`, Y).`

Instances of `call/N` can be specialised if their first argument is a nonvariable. For example, the type *list*(T) leads to the predicate `list/2` in Program 12. The type *rtree*(T), an M-way tree consisting of a term $rt(X, Y)$, where X is of type T and Y is of type *list*(*rtree*(T)), can be defined using the predicate `rtree/2`.

Program 12 *Higher-order skeletons for list*(T) *and rtree*(T)*:*

```
list(T, []).
list(T, [X|Xs]) :-
    call(T, X), list(T, Xs).
```

```
rtree(T, rt(X, RTs)) :-
    call(T, X), list(rtree(T), RTs).
```

higher-order skeletons go against the spirit of simplicity embodied in stepwise enhancement, and the control flow of the program above (mutual recursion through `call/N`) would certainly be confusing for a novice programmer. The advantage is that it saves having multiple copies of similar code. Rather than have separate skeletons for simple lists, lists of lists, lists of rtrees, and so on, a single higher-order definition can be given. A specialised definition of a type such as *rtree*(*any*) can be obtained by partial evaluation (eliminating all instances of `call/N`), and a version of `foldr` can be derived as described above. For `rtree/1`, the result is Program 13.

Program 13 *Specialised skeleton and version of foldr for rtree:*

```
rtree_any(rt(X, RTs)) :-
    list_rtree_any(RTs).

list_rtree_any([]).
list_rtree_any([RT|RTs]) :-
    rtree_any(RT),
    list_rtree_any(RTs).

foldrrt(FR, FC, B, rt(X, RTs), V) :-
    foldrlrt(FR, FC, B, RTs, V1),
    call(FR, X, V1, V).

foldrlrt(FR, FC, B, [], V) :-
    B = V.
foldrlrt(FR, FC, B, [RT|RTs], V) :-
    foldrrt(FR, FC, B, RT, V1),
    foldrlrt(FR, FC, B, RTs, V2),
    call(FC, V1, V2, V).
```

6.5 Flexible modes

As well as allowing flexibility with types and nondeterminism, logic programs allow flexibility with modes. Rather than having fixed inputs and

one output, as in functional programs, logic programs can potentially be run backward—computing what would normally be considered the input from a given output. This flexibility can extend to higher-order predicates, including those generated automatically from skeletons.

As an example, we can construct a metainterpreter for Prolog by using `foldrrt/4` backward. A Prolog proof tree is represented by an rtree, where each node contains (the representation of) a Prolog atom that succeeded. The `foldrrt/4` procedure can be used to check that an rtree of atoms is a valid proof tree for a particular program and goal. A proof tree is valid if the atom in the root is the goal and if, for each node in the tree containing atom A and children $B1, B2, \ldots$, there is a program clause instance $A :- B1, B2, \ldots$. The `proof_of/2` procedure in Program 14 represents clauses as a head plus a list of body atoms (procedure `lclause/2`) and can check that an rtree is a valid proof tree and return the atom that has been proved.

Program 14 *Interpreter constructed using rtree:*

```
% Checks Proof is a valid proof tree and returns proved Atom;
% run backwards its a meta interpreter returning a proof tree
proof_of(Proof, Atom) :-
    foldrrt(lclause2, cons, [], Proof, Atom).

% checks H :- B is a clause instance; returns H
lclause2(H, B, H) :- lclause(H, B).

% clause/2 where clause bodies are lists
lclause(append([],A,A), []).
lclause(append([A|B],C,[A|D]), [append(B,C,D)]).
lclause(append3(A,B,C,D), [append(A,B,E),append(E,C,D)]).
...

cons(H, T, [H|T]).
```

With a suitable evaluation order, the code can also be run backward. Given an atom, `foldrrt/4` acts as a metainterpreter, (nondeterministically) returning a proof tree for (a computed instance of) the atom. This is an example of constructing a program based on the type of its output, as discussed earlier. By utilising the natural association between a type and `foldr` and the flexible modes of logic programming, much of the process can be automated.

6.6 Foldl

In many cases, the higher-order function `foldl` is preferable to `foldr`, since it is tail recursive rather than left recursive (and thus may be more efficient, at least for strict evaluation). Note that the complexity of `foldl` can actually be worse in some cases (depending on the operation), and if the operation is not associative, the result of using `foldl` is generally different from that using `foldr`. It is not immediately obvious how to adapt `foldl` to general tree types rather than just lists. One possibility, suggested by Barry Jay, is to perform a breadth-first traversal (`foldr` uses a depth-first traversal). This can be coded in a tail-recursive fashion and is a familiar programming technique.

Another possibility, which we pursued initially and is used in Belleannie, Brisset, and Ridoux [BBR97], is to use `foldr` with more complex data flow, using logic variables. The result argument of `foldr` can be a pair of terms, one of which can be used as an input, and the accumulator style of programming can be used. If the accumulator is a list, we can think of `foldr` returning a difference list (see Sterling and Shapiro [SS94]) instead of a list. With this style of programming, the data dependencies are such that the instances of `call/N` in the `foldr` definitions can be executed before the recursive call(s), allowing tail recursion.

However, we believe the most elegant and natural generalisation of `foldl` is evident in the stepwise enhancement paradigm. We adapted stepwise enhancement to produce higher-order `foldr` procedures using a generalisation of the calculate and build techniques. By using *accumulator techniques*, we can produce a `foldl` procedure for any skeleton. Expert Prolog programmers use accumulators much more often than breadth-first traversals, and the code produced has simple data flow and can be translated into a functional language if the initial skeleton corresponds to an algebraic type.

Definition 15 (foldl transformation) *The transformation is similar to the one described for* `foldr`. *The same number of higher-order arguments is used, and there is one output argument, as before, but there is also an extra accumulator argument. The* `call/N` *is the leftmost atom in the body, and the accumulator and output arguments are "threaded" through this and the recursive calls in the clause body in the familiar way (refer to Sterling and Shapiro [SS94]).*

The accumulator and output arguments can be made implicit by using

the standard definite clause grammar (DCG) notation. The resulting version of `foldl` for lists is as follows.

Program 16 *Automatically derived foldl for lists:*

```
% Explicit accumulator version
foldl(FC, FB, [], A0, A) :-
    call(FB, A0, A).
foldl(FC, FB, [X|Xs], A0, A) :-
    call(FC, X, A0, A1),
    foldl(FC, FB, Xs, A1, A).

% DCG (implicit accumulator) version
foldl(FC, FB, []) -->
    call(FB).
foldl(FC, FB, [X|Xs]) -->
    call(FC, X),
    foldl(FC, FB, Xs).
```

There are two differences between this version of `foldl` and the standard `foldl` for lists defined in functional programming languages. The first is that the argument order for the call to the `FC` "function" is swapped. This is not essential but allows the accumulator and output arguments to be implicit using the DCG notation. It is also consistent with `foldr`. The second difference is the use of a function called in the base case. The standard version of `foldl` simply returns the accumulator when the end of the list is reached. This is equivalent to our version of `foldl` with the identity function (`=/2` in Prolog) as the function for the base case.

For data structure such as lists that are "linear" and have no data in the (only) leaf, calling a function when the base case is reached adds no real power. The function can always be called at the top level after `foldl` has returned, with the same effect. However, for tree structures or linear structures with data in the leaf, a function application at the base case is often essential. Below are the versions of `foldl` for the `bt` type and `connected/2` procedure. Note `prod_leaves/2` (`sum_leaves/2`) has the multiplication (addition) at the leaves, as in Program 5. These predicates are equivalent to the previous versions, in Program 8, if `plus/3` and `times/3` are associative.

Program 17 *Versions of foldl for* `is_tree/1` *and* `connected/2`*:*

```
foldlbt(F, B, leaf(X)) -->
    call(B, X).
foldlbt(F, B, t(L,R)) -->
    call(F),
    foldlbt(F, B, L),
    foldlbt(F, B, R).

prod_leaves(T, P) :-
    foldlbt(=, times, T, 1, P).

sum_leaves(T, P) :-
    foldlbt(=, plus, T, 0, P).

rev_traverse(Tree, Xs) :-
    foldlbt(=, cons, Tree, [], Xs).

foldlcon(F, B, A, A) -->
    call(B, A).
foldlcon(F, B, A0, A) -->
    call(F, A0),
    {edge(A0, A1)},
    foldlcon(F, B, A1, A).

% non-looping connected; returns path
con_no_loop(A0, A, As) :-
    foldlcon(cons_nm, cons, A0, A, [], As).

cons_nm(A0, As, [A0|As]) :-
    not member(A0, As).
```

For `foldlcon/6`, the call to `edge/2` is not recursive, and so accumulator arguments are not added (braces are used to indicate this in the DCG notation). From `foldlcon/2` it is simple to code `con_no_loop/3`, which finds connected nodes but avoids cycles. The accumulator is the list of nodes visited so far, in reverse order. The procedure that adds a new node to the accumulator, `cons_nm/3`, fails if the node is already on the path. The path is also returned at the top level.

Since the skeleton `is_tree/1` is an RUL program and hence equivalent to an algebraic type, `foldlbt/4` is deterministic and behaves as a higher-order function over that type. The threading of the accumulator and result arguments in the body of a clause is equivalent to nesting of functional expressions. For comparison, we give the equivalent Haskell code in Program 18.

Program 18 *Haskell version of foldl for is_tree/type bt:*

```
>data Bt a = Leaf a | Tree (Bt a) (Bt a)
>foldlbt :: (a->a)->(b->a->a)->(Bt b)->a->a
>foldlbt f b (Leaf x) a = b x a
>foldlbt f b (Tree l r) a =
>    foldlbt f b r (foldlbt f b l (f a))

>sum_leaves t = foldlbt (id) (+) t 0
```

There are actually two possible versions of `foldlbt/5`, depending on the order in which the two subtrees are visited. By swapping the two recursive calls in the DCG version, the argument threading is also changed, leading to a logically different procedure. The procedure `rev_traverse/2` in Program 17 returns the reverse of the traversal returned by Program 6. Using the other version of `foldlbt/5` would result in the same traversal order. The choice of traversal orders and additional argument in `foldl` are consistent with the intuition that programming with accumulators or `foldl` is more complicated than using simple recursion or `foldr`.

7 Tools for program development

A prototype tool to support the stepwise enhancement paradigm is discussed in Sterling and Sitt Sen [SS93]. Users interact with the tool to construct an enhancement from one of several predefined skeletons. The discussion in this paper on higher-order programming and shape suggests three tools to automate parts of the software development process. The first converts types (defined in some suitable syntax) into Horn clause definitions. The second converts Horn clause definitions into related higher-order predicate definitions. The third optimises code to eliminate the overheads of higher-order predicates (see Figure 2).

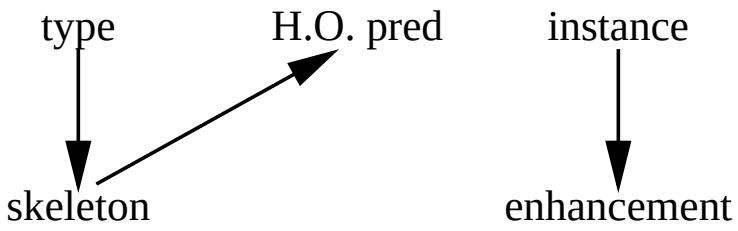

Figure 2: Three software tools

Considering only the work on shape and polytypism in functional languages, it seems more natural to go directly from types to higher-order functions. However, splitting this into two separate stages has significant advantages. Having Horn clause definitions of types is useful in itself for runtime type checks, test data generation, and so forth. These Horn clause definitions may also be refined by adding further constraints, for example, refining the definition of a binary tree to obtain the definition of a binary search tree. Most importantly, it allows higher-order definitions to be derived from arbitrary Horn clause programs, significantly increasing flexibility.

Program 19 *User input to tools:*

```
:- type tree(T) ---> leaf(T); t(tree(T),tree(T)).
:- skeleton tree_any/1 generates foldl, foldr.

sum_tree(T, N) :- foldl_tree_any(plus, =, T, 0, N).
```

We have implemented prototypes of the first two tools and experimented with a previously developed optimisation tool. Program 19 gives a small example of the use of our tools. It contains the definition of the polymorphic type `tree`, a directive which states that versions of $foldl$ and `foldr` should be generated for this type, and a use of $foldl$ for this type. In a more tightly integrated system, the need for explicit directives could be avoided by detecting such uses of higher-order predicates and generating predicates based on usage. The first tool takes the type definition and generates Horn clause definitions of `tree/2` (a higher-order predicate corrresponding to the polymorphic type) and `tree_any/1`. The second tool takes the definition of `tree_any/1` and produces Horn clause definitions of the higher-order predicates `foldl_tree_any/5` and `foldr_tree_any/4`. Existing tools/compilers

can transform `sum_tree` so that it is as efficient as hand-optimised code. We plan to implement a Web interface to the tools in the future.

Converting types into Horn clause definitions is conceptually straightforward. However, our implementation was made more complex by allowing type synonyms and undescriminated unions in type definitions and by seeking to make the Horn clause definitions as simple as possible. The simplicity of the output is important for the subsequent generation of higher-order predicates. Our prototype needs more work to ensure that the simplest output is produced for all possible inputs. Optimisation of higher-order code is also reasonably straightforward and has been done in several logic programming systems, such as the compilers for XSB-Prolog (Sagonas and Warren [SW95]) and Mercury (Somogyi, Henderson, and Conway [SHC95]). Unfortunately, most Prolog compilers do not perform such optimisations, and there are no available source-to-source optimisation tools that are guaranteed to work correctly for standard Prolog.

Converting Horn clause definitions into higher-order predicates requires more innovation. One challenge is taking a known transformation from regular types to higher-order functions and generalising it to Horn clause definitions, which may not correspond to regular types. We have shown one way to do this for `foldr` and developed a new generalisation of `foldl`, but each transformation requires careful thought and there is not necessarily a unique "best" solution. Another challenge is determining a *useful* set of transformations or abstractions. An obvious starting point is the functions over lists, which have been found useful in functional programming. However, the optimal set depends on the programming style and what programming patterns occur frequently. Although there are important programming patterns that lie in the intersection of what is coded in logic and functional languages, there are also patterns that are unique to each paradigm.

8 Higher-order coding in Prolog

Several years ago the first author implemented some higher-order predicates for his own use in programming. The initial set was a Prolog version of the Miranda standard library.[1] Some functions were not implemented because they rely on lazy evaluation. This was the first divergence between the higher-order primitives used in Prolog and functional programming. The

[1]Miranda is a trademark of Research Software Limited.

set of higher-order predicates was expanded as new code was developed and patterns were noted, resulting in further divergence. The usage pattern of higher-order primitives reported below is based on around 1400 lines of recently developed code (comments and blank lines excluded) for program analysis and transformation. The coding style was "pure," avoiding nonlogical primitives where possible, and efficiency was generally ignored. Analysis of a larger body of code would obviously be desirable.

More recently we implemented the prototype transformation tools for `foldr` and `foldl` and implemented extra transformations as it became evident they were useful. The main application to date is a program analysis system that manipulates systems of constraints. The task was to change the representation of constraints to allow greater flexibility (such as adding disjunctive constraints which were previously not supported). The transformation tools proved to be very useful.

The new data structure for constraints was quite complicated, one type having fifteen different constructors. In addition, the design took advantage of Prolog's flexibility with respect to types and used overloading of representations (multisets and conjunctions used the same representation) and subtypes (at different program points several different subtypes of the constraint type were used). Partly because of this, we edited the output of the transformation tools in several instances. Although the tools were designed with the higher-order coding style in mind, we used a mixture of the higher-order style and the stepwise enhancement style. With a more disciplined use of types, which would have required more explicit type definitions and type conversions, it is likely that fewer modifications to the output of the tools would have been required. However, editing the output of the tools would still have been useful in cases where the program pattern was similar but not identical to an instance of one of the abstractions supported by the tools. Around 500 lines of generated code were used in the final system (more than the number of lines of code used to implement the tools). We believe the tools significantly improved productivity.

8.1 Map

Prior to the development of our transformation tools, we noted that the most commonly used higher-order predicates were those in the map family (over seventy-five percent of higher-order calls). However, we had adapted these predicates to suit the logic programming style. The function `map2` takes a

function f and two lists [$x_1, x_2, \ldots$] and [$y_1, y_2, \ldots$] and returns the list [$f\ x_1\ y_1, f\ x_2\ y_2, \ldots$]. If the two input lists differ in length, the additional elements in the longer list are ignored, increasing flexibility. The analogous Prolog predicate, `map2(F,As,Bs,Cs)`, is (ironically) less flexible if we allow this freedom. A version that only succeeds for lists of the same length can be used in more modes, for example, producing `As` and `Bs` from `Cs`. Only a minority of our uses of `map2` were in the mode corresponding to the functional program.

The second most commonly used predicate in the `foldl` family (almost as common as `map/3`) was a predicate we named `map0/2`. It takes a predicate and applies it to each member of a list, naturally fitting into the map family but without a functional programming counterpart (though it is similar to `all`, which takes a boolean function and a list and returns a boolean). Its uses in Prolog are for several programming techniques that are not available in languages such as Miranda. First, predicates can be used as tests, implicitly succeeding or failing rather than explicitly returning a boolean. Second, because list elements can contain logic variables, calling a predicate for each list element may further instantiate the list. Third, Prolog procedures can have side effects.

Our tool supports transformation of a skeleton into a version of `map`N for all natural numbers N. As well as higher-order arguments, the transformed code has N additional copies of the original set of arguments and corresponding copies of all nonrecursive calls. The principle is that `map`N should succeed only if the original skeleton succeeds for each of the sets of arguments. Higher-order arguments are added to relate variables that do not occur in any recursive calls. Program 20 shows the result of transforming `tree_any/1` with `map(2)` and `connected` with `map(1)`. `Map2_tree_any(P,T1,T2,T3)` is true if the last tree arguments are trees of the same shape with their elements related by predicate `P`. `Map_connected(P1,P2,X0,X,Y0,Y)` is true if there are two paths of the same length, from `X0` to `X` and from `Y0` to `Y`, where the end nodes are related by `P2` and the other nodes are related by `P1`.

Program 20 *Map transformations for* `tree_any/1` *and* `connected/2`*:*

```
map2_tree_any(A, leaf(B), leaf(C), leaf(D)) :-
    call(A, B, C, D).
map2_tree_any(A, tree(B, C), tree(D, E), tree(F, G)) :-
    map2_tree_any(A, B, D, F),
```

```
    map2_tree_any(A, C, E, G).

map_connected(A, B, C, C, D, D) :-
    call(A, C, D).
map_connected(A, B, C, D, E, F) :-
    call(B, C, E),
    edge(C, G),
    edge(E, H),
    map_connected(A, B, G, D, H, F).
```

8.2 Member

The predicate `member/2` is commonly used for either searching a list for a particular element or generating all elements of a list using backtracking. Analogous operations are also useful for other data structures, though searching is often best treated separately for efficiency reasons. To generalise `member/2`, we noted that it can succeed with nonlists, since solutions can be returned without the whole data structure being traversed. In our transformation, all nonrecursive calls are simply deleted. We add one extra argument, which is bound to successive members of the data structure; there are no "higher-order" arguments added. Conjunctions of calls are transformed into disjunctions, with the same variable used as the extra argument in the head and each call. Additional disjuncts are added, which unify this variable with each variable that occurs in the clause head but not in a recursive call. Clauses with no recursive calls or head variables have `fail` (an empty disjunction) as the body; such clauses could also be deleted. Program 21 shows the result of transforming `rtree_any/1`.

Program 21 *Member transformation for* `rtree_any/1`*:*

```
member_rtree_any(rt(A, B), C) :-
    (    C = A
    ;    member_list_rtree_any(B, C)
    ).

member_list_rtree_any([], A) :-
    fail.
member_list_rtree_any([A|B], C) :-
```

```
(   member_rtree_any(A, C)
;   member_list_rtree_any(B, C)
).
```

8.3 Other predicates

Before implementing our transformation tools we recognised several programming patterns and implemented the corresponding higher-order predicates. A version of `map` that also has an accumulator pair of arguments, `map_acc`, has been of use and could be implemented in functional languages as well. A variation of `map0` that calls a predicate for each consecutive pair of elements in a list, `map0_consec`, can be used to check sortedness, for example. This could be adapted to functional programming by applying functions that return booleans. We have also used a combination of `map` and `filter` that relies on the ability of a predicate to return either a result or fail: `split(P,Xs,Ys,Zs)` applies `P/2` to each member of `Xs` and returns the list of results of successful calls to `P` and the list of elements for which `P` failed.

We have also found it useful to extend our tools to implement transformations that are special cases of other higher-order predicates. Instead of using a complicated instance of a very general higher-order predicate, we can use a relatively simple predicate. This is appropriate if the instance of the higher-order predicate is common. We have already seen an example of this: *map* is an instance of *foldr*, at least in the case of algebraic types. It is somewhat special instance since it is particularly useful and can be coded in a tail-recursive way due to the nature of unification. We have implemented another instance of *map*/*foldr* that preserves the shape of the data structure without relating the elements: *same_shape*. It is a generalisation of `same_length` for lists. We have also implemented an instance of *foldl* that converts any data structure into a list using an accumulator, *list_from*, which generalises a reverse postorder traversal. The transformed code contains calls to `append` where the first argument is the list of variables that do not occur in recursive calls. The `append` calls are optimised away by the DCG expansion in NU-Prolog. Program 22 shows the result of transforming `rtree_any/1`.

Program 22 *List-from transformation for* `rtree_any/1`*:*

```
list_from_rtree_any(rt(A, B)) -->
    append([A]),
```

```
    list_from_list_rtree_any(B).

list_from_list_rtree_any([]) -->
    append([]).
list_from_list_rtree_any([A|B]) -->
    append([]),
    list_from_rtree_any(A),
    list_from_list_rtree_any(B).
```

9 Further work

A category-theoretic reconstruction of our new transformations such as that for `foldl` (restricted to RUL programs) may produce some deeper insights and should extend the understanding of shape and polytypism for functional languages. A more theoretical treatment of the higher-order logic programs we derive may also be worthwhile. A language such as λ-Prolog (Nadathur and Miller [NM88]) that has higher-order constructs with well-defined semantics is a potential basis and has been used for characterisation of schemas (Gegg-Harrison [GH95, GH96]). Incorporating transformations such as those we have suggested into logic programming languages that support some notion of types would also enrich these languages.

The tools we have implemented could also be improved and extended. The tool that converts from type definitions to Horn clauses should ideally support an expressive type language but always output the simplest possible Horn clause definitions. The tool that generates higher-order definitions could also be enhanced. Instead of always producing code cotaining `call/N` there could be optional support for `apply/3`. Alternatively, "`??`" could be output instead of `call` and the extra higher-order arguments could be avoided, supporting the stepwise enhancement paradigm. Other "shapely" operations such as `zip2/3` (which takes two lists and returns a list of pairs) could also be generalised, as suggested by Jay [Jay95]. Further generalisations of `foldr` and `foldl` could also be devised (see Belleannie, Brisset, and Ridoux [BBR97]). For example, we could add higher-order calls to the start *and* end of each clause body, or even between each call as well. We have identified one higher-order predicate for lists, `map_acc`, which is a combination of `foldr` (map) and `foldl` (accumulators) and could be generalised to other data types.

We note that while the quest for more expressive constructs is alluring, the "Holy Grail" is not a single construct with ultimate expressive power; we started with just that—a general purpose programming language. If each use is twice as complicated, there is no benefit in replacing `foldr` and `foldl` by a more general predicate that is applicable in twenty percent more situations. The ideal situation is to have a collection of higher-order predicates or functions with a good tradeoff between applicability and complexity. Such sets can be developed over time, based on coding patterns that occur in practice, and properties of different (sets of) primitives can be established.

10 Conclusions

Research into systematic program construction has the important aim of elevating coding from the realm of arts and entertainment to science and engineering. In this paper we have built a bridge between the pragmatic syntax-based approach of stepwise enhancement and the very theoretical semantic approach of shape and polytypism. Despite the gulf between the research methodologies behind these two approaches, there is a very close relationship between them. This is pleasing in itself but also allows us to see ways in which both approaches can be generalised.

From the work on shape and polytypism in functional languages, we have the generality of arbitrary functions as parameters, polymorphic types, and the automatic synthesis of certain higher-order functions from algebraic types. From the work on stepwise enhancement in logic programming, we have the generality of nondeterminism, additional arguments, flexible modes, and use of accumulators. By combining the advantages of both approaches, we have shown how more code in both functional and logic programming languages can be constructed in a systematic and partially automated way. Initial experiences with prototype tools based on these ideas have been very encouraging.

References

[BBR97] C. Belleannie, P. Brisset, and O. Ridoux. A pragmatic reconstruction of λ-Prolog. Technical Report Publication Interne no. 877, IRISA, October 1994 (revised 1997).

[BJM96] G. Bellé, C. B. Jay, and E. Moggi. Functorial ml. In *Proceedings of PLILP '96*, volume 1140 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1996.

[GH95] T. Gegg-Harrison. Representing logic program schemata in λ-Prolog. In L. Sterling, editor, *Proceedings of the Twelfth ICLP*, pages 467–481. MIT Press, 1995.

[GH96] T. Gegg-Harrison. Extensible logic program schemata. In J. P. Gallagher, editor, *Logic Programming Synthesis and Transformation: Sixth International Workshop, LOPSTR'96*, volume 1207 of *Lecture Notes in Computer Science*, pages 256–274. Springer-Verlag, 1996.

[Jay95] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[JC94] C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Proceedings of Programming Languages and Systems—ESOP '94: Fifth European Symposium on Programming*, pages 302–316. Springer-Verlag, April 1994.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[JJ97] P. Jansson and J. Jeuring. PolyP—A polytypic programming language extension. In *Conference Record of POPL '97: The Twenty-fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM, 1997.

[KMS96] M. Kirschenbaum, S. Michaylov, and L. S. Sterling. Skeletons and techniques as a normative approach to program development in logic-based languages. In *Proceedings of ACSC'96, Australian Computer Science Communications, 18(1)*, pages 516–524, 1996.

[Lak89] A. Lakhotia. *A Workbench for Developing Logic Programs by Stepwise Enhancement*. Ph.D. thesis, Case Western Reserve University, 1989.

[Nai96] L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996.

[NM88] G. Nadathur and D. Miller. An overview of λ-Prolog. In K. Bowen and R. Kowlaski, editors, *Proceedings of JICSLP*, pages 810–827. MIT Press, 1988.

[NS98] L. Naish and L. Sterling. A higher order reconstruction of stepwise enhancement. In N. E. Fuchs, editor, *Logic Programming Synthesis and Transformation: Seventh International Workshop, LOPSTR'97*, volume 1463 of *Lecture Notes in Computer Science*, pages 245–262. Springer-Verlag, October 1998.

[PS90] A. J. Power and L. S. Sterling. A notion of map between logic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh ICLP*, pages 390–404. MIT Press, 1990.

[SHC95] Z. Somogyi, F. J. Henderson, and T. Conway. Mercury: An efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995. Australian Computer Science Association.

[SS93] L. S. Sterling and C. Sitt Sen. A tool to support stepwise enhancement in prolog. In *Workshop on Logic Programming Environments*, pages 21–26, Vancouver, October 1993.

[SS94] L. Sterling and E. Shapiro. *The art of Prolog, 2d ed.* Logic Programming Series. MIT Press, Cambridge, 1994.

[SW95] K.s Sagonas and D. S. Warren. Efficient execution of HiLog in WAM-based Prolog implementations. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 349–363, Japan, June 1995. MIT Press.

[SY96] L. S. Sterling and U. Yalçinalp. Logic programming and software engineering—Implications for software design. *Knowledge Engineering Review*, 11(4):333–345, 1996.

[VF95] 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 LOPSTR'95*, pages 174–188. Springer-Verlag, 1995.

[YS90] E. Yardeni and E. Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.