Higher-order logic programming in Prolog

Lee Naish

(lee@cs.mu.OZ.AU, http://www.cs.mu.oz.au/~lee)

Technical Report 96/2

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia

Abstract

This is yet another paper which tells logic programmers what functional programmers have known and practiced for a long time: "higher order" programming is the way to go. How is this paper different from some of the others? First, we point out that `call/N` is not the way to go, despite its recent popularity as the primitive to use for higher order logic programming. Second, we use standard Prolog rather than a new language. Third, we compare higher order programming with the skeletons and techniques approach. Fourth, we present solutions to some (slightly) more challenging programming tasks. The interaction between higher order programming and some of the unique features of logic programming is illustrated and some important programming techniques are discussed. Finally, we examine the efficiency issue.
Keywords: call, apply, functions, skeletons and techniques, programming methodology, programming pearls

1 Introduction

Functional programmers have shown the outstanding benefits of the higher order style of programming. It is ubiquitous in modern functional programming languages. It enables greater code reuse, encourages more abstraction and alleviates the tedium of writing many similar long-winded recursive definitions. Over the years many people have also toyed with this style of programming in Prolog but to say it has not caught on would be a gross understatement. Higher order programming has been has been advocated more in newer logic programming languages such as HiLog [CKW93], Lambda Prolog [NM88][GH95], Mercury [SHC95] and the many combined logic and functional languages (see [Han94]). Some advocates of these languages are apparently unaware of the techniques available to Prolog programmers, despite the efforts of Richard O'Keefe and others.

One aim of this paper is to illustrate higher order programming techniques in Prolog. We give examples to show the interaction between higher order programming and other features of Prolog such as nondeterminism, logic variables, flexible modes, meta programming, Definite Clause Grammar notation and (in some systems) coroutining. Another aim is to point out a significant difference between two ways in which higher order features are supported in logic programming languages. The superior method was suggested by David H.D. Warren over a decade ago. Since then higher order logic programming has gone off the rails. The Prolog coding style advocated by O'Keefe and the HiLog and Mercury languages use a less flexible method which does not realise the full benefits of higher order programming. The final aim is to compare the higher order style with the "skeletons and techniques" approach to program development, one of the many approaches based on program patterns, schemata and cliches.

We assume familiarity with a modern functional programming language such as Haskell, Miranda[1] or Gofer [BW88]. The paper is structured as follows. After briefly discuss typical logic programming and functional programming styles we show how "higher order" predicates can be coded using the `call/N` primitive. Next we demonstrate that greater flexibility can be achieved using `apply/3` and discuss the implementation of both primitives. We then give some slightly larger examples of higher order programs and compare higher order coding with skeletons and techniques. Finally we discuss efficiency issues and some limitations of staying within the conventional Prolog framework.

2 Logic Programming Style

We use the following insertion sort program to illustrate how simple Prolog programs are typically constructed.

```
% insertion sort (simple version)
isort([], []).
isort(A.As, Bs) :-
    isort(As, Bs1),
    insert(A, Bs1, Bs).

% insert number into sorted list
insert(N, [], [N]).
```

[1] Miranda is a trademark of Research Software Limited

```
insert(N, H.L, N.H.L) :-
    N =< H.
insert(N, H.L0, H.L) :-
    N > H,
    insert(N, L0, L).
```

The `isort` predicate follows a typical pattern for list processing. There is a base case for `[]` and a case for `A.As` with a recursive call containing `As` and another call containing `A`. The structure of `insert` is similar, except the case dealing with non-empty lists has been split further into two sub-cases. The two basic techniques used are structural induction and case analysis.

More experienced Prolog programmers who are concerned about efficiency may use another typical pattern for the definition of `isort`: simply calling an auxiliary predicate which as an extra argument, an "accumulator". This tail-recursive coding typically uses less space and time than the simple version. Later we discuss the "skeletons and techniques" approach to developing code, which uses textual patterns in code more formally.

```
% insertion sort (accumulator version)
isort1(As, Bs) :-
    isort_a(As, [], Bs).

% second arg is sorted elements seen so far
isort_a([], Bs, Bs).
isort_a(A.As, Bs, Cs) :-
    insert(A, Bs, Bs1),
    isort_a(As, Bs1, Cs).
```

3 Functional Programming Style

A typical programmer using a modern functional language would code `insert` in a similar way to the Prolog version, using structural induction and case analysis. The code for `isort` may also be written in the same style. However, a good functional programmer should realise that the resulting code follows a familiar pattern. That pattern is generalised by the higher order function `foldr`, which can be used to code `isort` more simply as follows.

```
isort As = foldr insert [] As
```

Languages such as Haskell which use "curried" functions allow an even more concise and high level definition. The idea of currying it that applying a function f to arguments $a_1, a_2, ..., a_n$ is equivalent to applying f to a_1, which returns a function, and applying that function to $a_2, ..., a_n$. This allows greater flexibility in both the definition and use of functions and encourages more abstraction. Just considering the semantic relationship between the functions `isort`, `insert` and `foldr` we easily can arrive at the following definition:

```
isort = foldr insert []
```

The functional equivalent of the tail recursive `isort1` definition can obtained by replacing `foldr` by `foldl` and reversing the argument order of `insert`:

```
isort1 = foldl (converse insert) []
```

Experienced functional programmers use structural induction and case analysis where needed, but also use higher order functions to simplify the coding where common patterns of recursion would otherwise be used. In an introductory computer science course using Miranda at the University of Melbourne most students used `map` and `filter`, many used `until`, `foldl` and `foldr`, many defined their own higher order functions and some use compose (.), `converse`, `scan` and `iterate`.

4 Logic Programming with `call/N`

In standard Prolog `call(A)` treats `A` as a goal and calls it. For example, `A = append(Xs, Ys, [1,2]), call(A)` is equivalent to `append(Xs, Ys, [1,2])`. Richard O'Keefe and others have advocated the definition of `call/N` for larger `N`. `Call(A,` $\mathtt{B}_1, ..., \mathtt{B}_N$`)` calls `A` with additional arguments $\mathtt{B}_1, ..., \mathtt{B}_N$. For example, `A = append(Xs), call(A, Ys, [1,2])` is equivalent to the code above. Several Prolog systems support `call/N` as library predicates (up to some maximum `N`) or built in, and some newer logic programming languages such as Mercury [SHC95] have used `call/N` as the basis for their higher order features. Using `call/N` is it simple to define Prolog analogues of higher order functions:

```
% map(f, [e1, e2,..., en], [f e1, f e2,..., f en])
map(_F, [], []).
map(F, A0.As0, A.As) :-
    call(F, A0, A),
    map(F, As0, As).

% returns elements in list for which pred succeeds
filter(_P, [], []).
filter(P, A0.As0, As) :-
    (call(P, A0) ->
        As = A0.As1
    ;
        As = As1
    ),
    filter(P, As0, As1).

% foldr(op, b, [e1, e2,..., en], (e1 op (e2 op (...(en op b)...))))
foldr(F, B, [], B).
foldr(F, B, A.As, R) :-
    foldr(F, B, As, R1),
    call(F, A, R1, R).

% compose(f, g, x, f (g x))
compose(F, G, X, FGX) :-
    call(G, X, GX),
    call(F, GX, FGX).

% converse(f, x, y, f y x)
```

```
converse(F, X, Y, FYX) :-
    call(F, Y, X, FYX).
```

The following queries illustrate how these predicates can be used to in similar ways to higher order functions and how some additional features of logic programming interact with higher order predicates.

```
1. ?- filter(>(5), [3, 4, 5, 6, 7], As).
2. ?- map(plus(1), [2, 3, 4], As).
3. ?- map(between(1), [2, 3], As).
4. ?- map(plus(1), As, [3, 4, 5]).
5. ?- map(plus(X), [2, 3, 4], [3, 4, 5]).
6. ?- map(plus(X), [2, A, 4], [3, 4, B]).
7. ?- map(plus(X), [A, 3, 4], [3, 4, B]).
8. ?- foldr(append, [], [[2], [3, 4], [5]], As).
9. ?- foldr(converse(append), [], [[2], [3, 4], [5]], As).
10. ?- compose(map(plus(1)), foldr(append, []), [[2], [3, 4], [5]], As).
```

Query 1 binds `As` to the list of elements in the original list which are less than 5 (`>(5,X)` holds), that is, `[3,4]`. Query 2 adds 1 to each member of the list, returning `[3,4,5]`. These are typical simple examples used in higher order functional languages. The next four examples illustrate the added flexibility logic programs can have. Query 3 is nondeterministic. Given to integers `I` and `J`, `between(I,J,X)` nondeterministically binds `X` to an integer between `I` and `J`, inclusive. Thus query 3 returns six answers of the form `[X,Y]`, where `X` is 1 or 2 and `Y` is 1, 2 or 3. Query 4 computes the "input" of `map`, `[2,3,4]`, given the "output". This relies on `plus` being reversible. Query 5 computes the "function" `map` is applying, `plus(1)`, given the input and the output (of course it is necessary to partially specify the function for `map` to have any hope of working in this mode). Query 6 has no argument completely instantiated but still succeeds in the same way. First, `plus(X,2,3)` binds `X` to 1, then `plus(1,A,4)` binds `A` to 3, then `plus(1,4,B)` binds `B` to 5. Query 7 will not work with the standard left to right execution strategy but can if coroutining is used.

The next three queries illustrate more complicated use of higher order features. Query 8 flattens a list of lists by appending them all together with `[]`. Query 9 flattens a list of lists but uses the converse of `append` (the first two arguments are swapped), so the resulting list is reversed. Query 10 flattens a list of lists then adds one to each element. After some practice the higher order style of programming becomes natural. It results in very concise high level code and the tedium of writing recursive definitions which are yet another instance of `map` or `foldr` becomes painfully obvious.

5 Logic Programming with `apply/3`

Unfortunately, the higher order definitions we have given no not always work in the desired way. The following two queries are the analogue of functional expressions which would be evaluated correctly in languages such as Haskell.

```
11. ?- foldr(compose(append, map(plus(1))), [], [[2], [3, 4], [5]], As).
12. ?- map(plus, [2, 3, 4], As).
```

Query 11 is intended to add one to each element in a list of lists then concatenate all the lists giving the same result as query 10. In practice, the following goal is executed.

```
call(compose(append, map(plus(1))), [5], [], X)
```

Since `compose/5` is not defined (though `compose/4` is), the execution terminates with an error or failure. Query 12 calls `call(plus,2,X)` which also results in an error or failure. The analogous expression in a functional language would return a list of functions: `[plus(2), plus(3), plus(4)]`. It is common for intermediate results in higher order functional computations to contain values which are functions and this allows great flexibility.

The relative inflexibility of the code we have presented stems from the fact that exactly the right number of arguments must be given for `call/N` to work correctly. By using a different primitive, `apply/3` [War82], we can achieve flexibility of modern higher order functional languages. We can think of `apply(F,X,FX)` as taking a function `F` and argument `X` and returning the result `FX`. In the case where `FX` is a first order term, `apply/3` and `call/3` behave identically. For example, `call(plus(1),2,X)` and `apply(plus(1),2,X)` both bind `X` to the number 3. However, whereas `call(plus,2,X)` results in an error or fails, `apply(plus,2,X)` binds `X` to a representation of the function which adds 2 to a number. For consistency and simplicity we just add the extra argument and return the term `plus(2)`, though other representations are possible and may be more efficient in some cases.

In general, `apply(F,X,FX)` calls `F` with two extra arguments if such a predicate exists. If there are not enough arguments it binds `FX` to `F` with one extra argument. Previous uses of `call/3` can simply be replaced by `apply/3`:

```
% version of map which can return a list of functions
%
map(_F, [], []).
map(F, A0.As0, A.As) :-
    apply(F, A0, A),
    map(F, As0, As).
```

Uses of `call/N` for larger `N` can be replaced by multiple calls to `apply/3` using the idea of currying. For example, `call(F,X1,X2,R)` can be replaced by the conjunction `apply(F,X1,FX1), apply(FX1,X2,R)`:

```
% version of foldr which supports more complicated functions
foldr(F, B, [], B).
foldr(F, B, A.As, R) :-
    foldr(F, B, As, R1),
    apply(F, A, FA),
    apply(FA, R1, R).
```

Looking at the execution of a simplified version of goal 11 above clarifies why two calls to `apply/3` are needed:

```
?- foldr(compose(f,g), [], [..., [5]], As).
...
   call(compose(f,g), [5], [], R), ... % call/4 version (doesn't work)
   apply(compose(f,g), [5], FA), apply(FA, [], R), ... % apply/3 version
```

Two `apply` computations are performed. The first call applies `g` to `[5]` obtaining some result `X` and binds `FA` to `f(X)`. The second call applies `f` to `X` and `[]`. In general, applying a function to N arguments may require N separate non-trivial computations. Using `call` rather than `apply` only works when the first $N-1$ computations are trivial, just returning the same function with an extra argument added.

The versions of the higher order predicates which use `apply/3` behave in the same way as their functional counterparts. We believe that `apply/3` should be a standard part of Prolog. In retrospect, this is indicated by the implementation of NUE-Prolog [Nai91], a logic and functional language implemented by translating functions into Prolog via the "flattening" transformation. Higher order functions in NUE-Prolog are implemented using a specially coded version of `apply/3` which only works with functions. If `apply/3` had been provided in the Prolog system no additional predicate would have been needed to support NUE-Prolog and programmers could write equivalent higher order code using either functional or Prolog syntax.

Code using `call/N` is less flexible, cannot be reused as much and cannot be understood at such a high level as code using `apply/3`. Thus `call/N` does not achieve the full potential of higher order programming in Prolog. Mercury inherits this deficiency since its higher order features are based on `call/N`. The same is true of HiLog. The implementations of higher order code in HiLog are equivalent to using `call/N`, though much less direct [SW95][CKW93].

6 Implementation of `call/N` and `apply/3`

We will first discuss implementation of `call/N`. It is possible to implement `call/N` quite simply in Prolog if we put an upper bound on `N`. For each different arity we need a separate definition which will construct the term with additional arguments using `functor` and `arg` or `=..` then call the term using `call/1`. Quintus Prolog has supported `call/N` in this way. Three disadvantages of this technique are the number of definitions needed, the bound on `N` and the fact that an intermediate term is constructed, wasting heap space (if `=..` is used the intermediate list wastes even more space). In some systems `call/1` also has significant overheads, partly because it must detect some special cases such as conjunctions containing cut.

A significantly more efficient alternative is to build `call/N` into the Prolog system. The compiler can recognise `call/N` for unbounded `N` and at runtime no additional space needs to be used. In a WAM [War83] based system it is sufficient to load the argument registers then jump to the start of the procedure being called. NU-Prolog [TZ86] supports `call/N` in this way.

It is possible to achieve similar efficiency to the builtin implementation using pure Prolog by specialising the definition of `call/N` to a particular program. Rather than general code which supports every possible call, the definition can have a separate case for each procedure in the (given) program. For example, `call/3` could be defined with one clause for each procedure with arity greater than one as follows:

```
call(plus(A), B, C) :- plus(A,B,C).
call(append(A), B, C) :- append(A,B,C).
%... all other procedures missing 2 args
call(call, A, B) :- call(A, B).
```

```
call(call(A), B, C) :- call(A, B, C).
%... call/4 etc up to the bound on N
```

Other arities can be handled in the same way, including a version of `call/1` which doesn't support cut. It is easy to implement a tool which produces these definitions. To reduce the code size explosion these specialised definitions could be produced for only some arities. With a typical good Prolog implementation and these definitions the overhead of `call/N` is a switch statement and some register shuffling. In some systems this way of defining `call/1` is more efficient than the built in version. Since the definitions are pure Horn Clauses, they show that first order semantics can be given to "higher order" programs, though whether this is the most appropriate semantics is arguable.

The implementation issues concerning `apply/3` are similar to those for `call/N` but there is a question of definition we have glossed over so far concerning procedures with the same name but different arities. Suppose we have two procedures, `sum/2` and `sum/3` and a call of the form `apply(sum,A,B)`. It is not clear if the atom `sum` should be interpreted as `sum/2` missing two arguments (so `sum(A,B)` should be called) `sum/3` missing three arguments (so `B` should be unified with `sum(A)`), both (both answers could be returned on backtracking) or neither (`apply` could fail or cause an error). The choice is somewhat arbitrary but the bottom line is ambiguous procedure names do not mix well with the higher order style of programming and are best avoided.

Like `call/N`, `apply/3` can be implemented using `call/1`, as a built in or using Horn Clauses as follows (note this is a superset of the clauses in the definition of `call/3`):

```
apply(plus, A, plus(A)).
apply(plus(A), B, C) :- plus(A,B,C).
apply(append, A, append(A)).
apply(append(A), B, C) :- append(A,B,C).
%... all other procedures missing >= 2 args
apply(apply, A, apply(A)).
apply(apply(A), B, C) :- apply(A, B, C).
```

To define procedures such as `filter` a version of `call/2` (we use `applyp/2`) increases flexibility and for procedures such as `foldr/4` versions of `apply` with more arguments are convenient:

```
% same as call/2
applyp(plus(A,B), C) :- plus(A,B,C).
applyp(append(A,B), C) :- append(A,B,C).
%... all other procedures missing 1 arg
applyp(applyp(A), B) :- applyp(A, B).

% like call/4 but handles closures properly
apply4(F, A1, A2, R) :-
    apply(F, A1, FA1),
    apply(FA1, A2, R).
```

7 Skeletons and techniques

The "skeletons and techniques" view of Prolog programming has been proposed as a method of developing and understanding code [SK93]. We will describe this view then discuss how the higher order style of programming has leads to similar advantages in an (arguably) more elegant way. The example below illustrates how procedures for finding the length and sum of a list are similar to the procedure which defines or traverses a list. The only difference is they each have an additional argument and goal in the recursive clause. They are both *enhancements* of the `list` procedure and can be developed from it in a systematic way.

```
% definition/traversal of a list
list([]).
list(A.As) :-
    list(As).

% length of a list
length([], 0).
length(A.As, L) :-
    length(As, L1),
    L is L1 + 1.

% sum of a list of numbers
sum([], 0).
sum(A.As, S) :-
    sum(As, S1),
    S is S1 + A.
```

Under certain syntactic conditions [KSJ93], separate enhancements of the same procedure can be safely combined. For example, the enhancements associated with `length` and `sum` can be combined to create a procedure which computes both the sum and length of a list in one pass. It is easiest develop and understand `length` and `sum` separately but only by combining them into `sum_len` we avoid the overhead of traversing the list multiple times.

```
sum_len([], 0, 0).
sum_len(A.As, S, A) :-
    sum_len(As, S1, L1),
    S is S1 + A,
    L is L1 + 1.
```

Taking the higher order view of the two enhancements of `list`, we see they are both instances of `foldr`:

```
length(As, L) :- foldr(add1, 0, As, L).

sum(As, S) :- foldr(plus, 0, As, S).

add1(_, X, Y) :- plus(1, X, Y).
```

Higher order code can be seen as generalising a whole class of enhancements. One advantage over skeletons and techniques is that rather than just having a syntactic view of enhancements we can manipulate and reason about programs using techniques based on semantics.

For example, knowing `add1` and `plus` are associative and `plus` is symmetric we can arrive at the following definitions which uses the more efficient `foldl`.

```
length(As, L) :- foldl(converse(add1), 0, As, L).

sum(As, S) :- foldl(plus, 0, As, S).
```

There are several ways to arrive at a version of `sum_len` which uses a single traversal of the list.

Use program transformation: The first method is to start with a version which uses two traversals (just calling `sum` and `length` separately) and transform it using fold and unfold [TS84] et cetera. The transformations must combine two calls to `foldr` which use the same list argument. This is rather complicated and requires some knowledge of the properties of `plus` but can be done.

Use a different higher order predicate: The second method is to write a new higher order predicate which takes as arguments two predicates, two bases cases, one input list and two outputs. It can be coded directly or by generalising the program transformation steps above as follows. Given any two enhancements to `list`, the transformation steps used to combine them will be similar. The common structure in the transformations can be used to derive the higher order predicate below.

```
sum_len1(As, S, L) :-
    foldr_2(plus, 0, add1, 0, As, S, L).

% foldr_2(F1, B1, F2, B2, As, R1, R2) :-
%       foldr(F1, B1, As, R1), foldr(F2, B2, As, R2).
%
foldr_2(F1, B1, F2, B2, [], B1, B2).
foldr_2(F1, B1, F2, B2, A.As, R1, R2) :-
    foldr_2(F1, B1, F2, B2, As, R1a, R2a),
    apply4(F1, A, R1a, R1),
    apply4(F2, A, R2a, R2).
```

Use `foldr` with different arguments: The third method, which we believe is the *right* one, is to reuse the abstraction of `foldr`. By using `foldr` with a base case consisting of a pair of zeros and a function which maps pairs to pairs `sum_len` can be implemented quite simply:

```
sum_len2(As, S, L) :-
    foldr(plus_add1, 0 - 0, As, S - L).

plus_add1(H, S0 - L0, S - L) :-
    plus(H, S0, S),
    plus(1, L0, L).
```

By using a new abstraction `apply_pair(plus,add1)`, defined below, we can avoid defining `plus_add1`. `Apply_pair` can be used for combining *any* pair of enhancements and for defining `foldr_2` much more easily, using `foldr`:

```
apply_pair(F1, F2, A, X0 - Y0, X - Y) :-
    apply4(F1, A, X0, X),
    apply4(F2, A, Y0, Y).

foldr_2a(F1, B1, F2, B2, As, R1, R2) :-
    foldr(apply_pair(F1, F2), B1-B2, As, R1-R2),
```

One reason why the transformation approach is undesirable in general is that is can produce the wrong result for nondeterministic skeletons [SK93]. Consider the following example which traverses paths of a graph (later we use a more general version of this predicate).

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

We may want to enhance this predicate by constructing a path or calculating the length of the path then combine the enhancements. If there are N paths between two nodes we want N answers, each consisting of a path and its length. Using a conjunction of the path and path length predicates would result in N^2 solutions and the paths would not be related to the lengths. However, the enhancements can be generalised by the higher order predicate `foldrp` defined below and can be combined using `foldrp` and `apply_pair`.

```
% Finds path A0,A1,...,An,A in a graph
% and returns "foldr(F,B,[A0,A1,...,An])"
% (a path with N edges corresponds to a list of N elements)
% Note that edge can be nondeterministic.
%
foldrp(F, B, A, A, B).
foldrp(F, B, A0, A, R) :-
    edge(A0, A1),
    foldrp(F, B, A1, A, R1),
    apply4(F, A0, R1, R).

% finds path between two nodes in graph
% (doesn't include last node; a trivial path from a
% node to itself is represented by the empty list)
path(A0, A, As) :-
    foldrp(cons, [], A0, A, As).

% constructs list from head and tail
cons(A, As, A.As).

% finds length of path between two nodes in graph
```

```
p_len(A0, A, L) :-
    foldrp(add1, 0, A0, A, L).

% finds path between two nodes in graph and its length
path_len(A0, A, As, L) :-
    foldrp(apply_pair(cons, add1), [] - 0, A0, A, As - L).
```

The next example we will consider is a higher order meta interpreter. `Solve_a(F,A,R)` interprets the atomic goal `A` and returns an additional result `R` which depends on the higher order argument `F`. It is assumed that the object program clauses are represented by non-ground facts of the `lclause/2` predicate, which is like `clause/2` except the second argument is a list of subgoals rather than the normal "defaulty" data structure [O'K90]. `Solve_a(F,A,R)` finds a matching clause using `lclause/2` and is called recursively on each subgoal in the body producing a list of results `Rs`. The predicate `F` is then called with the atomic goal `A` and the list of results `Rs` to produce the final result.

```
% enhanced meta interpreter
solve_a(F, A, R) :-
    lclause(A, As),
    map(solve_a(F), As, Rs),
    apply4(F, A, Rs, R).
```

If `lclause`, `map` and `apply` have appropriate control information `solve_a` will work with coroutining systems as well as those which use left to right execution. The extra arguments `F` and `R` allow us to implement various structural enhancements to the basic meta interpreter [SS86]. For example, a proof tree can be returned by simply pairing each atom with results of sub-computations and the size of the proof tree can be returned by adding one to the sum of the sizes of the subcomputations[2]:

```
% returns proof tree
proof(A, P) :-
    solve_a(pair, A, P).

% return pair given two terms
pair(A, B, A - B).

% returns proof tree size
size(A, P) :-
    solve_a(add1_sum, A, P).

% sums a list and adds 1
add1_sum(_A, Ns, N) :-
    foldr(plus, 1, Ns, N).
```

These two enhancements can be combined by calling `solve_a` with a predicate which is a combination of `pair` and `add1_sum`. As before, we could generalise the way these two predicates are combined using a higher order predicate like `apply_pair`.

[2] As an exercise, the reader may wish to generalise `solve_a` so redundant lists are not created in cases such as this. Hint: `map` is an instance of the fold paradigm.

```
% returns proof tree and its size
size_proof(A, S, P) :-
    solve_a(pairFst_add1SumSnd, A, S - P).

% a combination of pair and add1_sum...
pairFst_add1SumSnd(A, SPs, S - (A - Ps)) :-
    map(snd, SPs, Ps),
    foldr(compose(plus, fst), 1, SPs, S).
```

Our final example is a different style of meta interpreter. The structure of `solve_a` is based on proof trees. The more traditional way to describe the Prolog proof procedure is with SLD trees [Llo87]. Each node contains a goal. The children of a node are found by resolving the selected literal in the goal with matching program clauses. Prolog uses a depth first search for a path from the root of the SLD tree to a leaf node with an empty goal.

The `connected` predicate defined earlier can be elegantly generalised by a higher order transitive closure relation. By using a different "edge" relation we can search any graph, including an SLD tree. Resolution theorem provers can be thought of as finding a path in a graph where each edge corresponds to a single resolution step. The Prolog strategy corresponds to the simple transitive closure algorithm and a simple selection rule for the resolution step.

```
% transitive closure of "edge" relation E
% simple depth first algorithm; uses DCG notation
tc(E) --> [].
tc(E) --> apply(E), tc(E).

% connected is the transitive closure of edge
connected --> tc(edge).

% meta interpreter for goal (a list of atoms)
% Finds sequence of resolution steps ending in empty goal
solve(G) :-
    tc(resolve, G, []).

% resolves first atom in goal with a clause
% (append could be avoided using difference list
% for clause bodies)
resolve(A.Cont, R) :-
    lclause(A, Body),
    append(Body, Cont, R).
```

A bounded depth interpreter can be constructed simply by attaching a depth bound to each node and preventing nodes with a depth bound of zero having any children. The depth of the solution in the SLD tree can also be returned:

```
% meta interpreter with depth bound DB (on SLD tree)
% Also returns depth of solution
solve_d(DB, G, DS) :-
```

```
    tc(apply_pair0(dec_depth, resolve), DB - G, D - []),
    DS is DB - D.

% decrements depth; fails if depth < 1
dec_depth(D0, D) :-
    D0 > 0,
    D is D0 - 1.

% like apply_pair/5 but without the extra argument
apply_pair0(F, G, X0 - Y0, X - Y) :-
    apply(F, X0, X),
    apply(G, Y0, Y).
```

We feel this example illustrates how the higher order approach allows more abstraction than skeletons and techniques. Using the semantics of `tc` allows us to reason about `solve` and `solve_d` at a very high level. The syntax and indeed the algorithm which implements `tc` can be changed without affecting the correctness of the meta interpreters. By using skeletons and techniques we get stuck at the level of syntax and the abstraction that declarative semantics can provide is difficult to achieve.

8 Efficiency

There are two disadvantages that the higher order approach has compared with skeletons and techniques. The first is that it requires more abstract thinking and hence is more difficult for novice programmers. However, abstraction is a good thing in general and for more experienced programmers the additional abstraction should be a great advantage. The second disadvantage is efficiency. The overheads of executing higher order code can be significant, especially if the primitives such as `apply` are implemented poorly. Fortunately, currently available partial evaluators for Prolog do an excellent job at eliminating the overheads. The sample session below shows how Mixtus [Sah93] optimises the version of `sum_len` which used `foldr` and `apply_pair`. Since Mixtus does not understand `apply/3` or `call/N` a simple version of `call/N` which used `=..` was supplied. The efficiency difference between the Mixtus output and a hand coded version is two jumps, which may be eliminated by peephole optimisation.

```
Mixtus 0.3.6

| ?- pe(sum_len(As, S, L)).
sum_len(A, B, C) :-
    sum_len1(A, B, C).

% sum_len1(A,B,C):-sum_len(A,B,C)
sum_len1(A, B, C) :-
    foldrapply_pairplus1(A, B, C).

% foldrapply_pairplus1(A,B,C):-
%  foldr(apply_pair(plus,add1),0-0,A,B-C)
```

```
foldrapply_pairplus1([], 0, 0).
foldrapply_pairplus1([D|F], C, A) :-
    foldrapply_pairplus1(F, E, B),
    C is D+E,
    A is B+1.
```

We have also experimented with using Mixtus to partially evaluate more complex procedures such as `solve_a`. In all cases the higher order overheads were eliminated completely. Some algorithmic inefficiencies, such as creating redundant lists, were avoided but only in some cases. Mercury transforms code specifically to eliminate higher order overheads and other systems do the same [SW95][Tra94]. Efficiency considerations should not deter us from using higher order code in Prolog.

9 Moving away from Prolog

Although we have argued that the higher order style of programming can be supported in Prolog, it must be conceded that additional benefits can be gained by moving to a richer framework. Here we mention three areas.

9.1 Syntax

Definitions of many simple functions such as `cons` and `pair` can be avoided if the langauge supports *lambda expressions* or similar constructs, as is done in Lambda Prolog, Mercury and many functional languages. This can make programs shorter and the programmer has fewer names to invent. It typically does not increase expressive power since each lambda expression in a program can be statically replaced by a call to a new function or procedure. However, it does have a significant impact on all source level tools and meta programming, since the structure of terms is made more complex. Lambda terms are supported in Qu-Prolog [CRS91] *because* they make terms more expressive, not for higher order programming. Semantics and implementation are also significantly affected due to the possibility of unifying two lambda terms.

9.2 Type checking

All too often when developing complex higher order Prolog code the result of a query is simply: `fail`. For equivalent code in another language the compiler would inform the programmer of a type error at a particular line in the program. The types in higher order code tend to be much more complex than types in first order code. For example, the types of the functions in Query 11 are: `(*->**)->(***->*)->***->**`, `(*->**->**)->**->[*]->**`, `(*->**)->[*]->[**]`, `[*]->[*]->[*]` and `num->num->num`. Type checking, either as an essential part of the language and compiler or as a separate optional tool, is extremely valuable for debugging higher order code.

9.3 Semantics

Although "higher order" Prolog code can be given first order semantics using the Horn clause definition of `apply/3`, this semantics is not intuitive or high level. Ideally, in the programmer's intended interpretation the term `plus` should be the addition predicate/function and

the term `converse(plus)` should have the same interpretation. However, there is nothing to prevent Prolog from attempting to unify these two terms and failing, even though they are equal in this interpretation. In general, a call with `plus` as an argument will have different answers to a call with `converse(plus)`:

```
surprise(plus, A, B, C) :- plus(A, B, C).
surprise(converse(plus), A, B, C) :- append(A, B, C).
```

By insisting that syntactic and semantic equality are equivalent, the Clark equality axioms (see [Llo87]) prevent a high level interpretation. Instead we must think at a lower level and interpret `plus` as *a representation* of the addition relation. The more abstract view of programs requires a different approach to semantics and restrictions on programs so predicates such as `surprise` cannot be used. It seems that types are essential and modes are desirable for detecting errors at compile time.

10 Conclusion

The logic programming community should take advantage of the higher order style of programming developed by the functional programming community. This style encourages more abstraction, more reuse of code and more concise programs. We have shown by example how higher order programming also fits well with the additional strengths of logic programming such as nondeterminism, multiple modes, logic variables, coroutining, meta programming, Definite Clause Grammars et cetera. It seems a good alternative to skeletons and techniques for developing code. The main disadvantage of higher order programming, loss of efficiency, can be overcome with current techniques.

To get all these benefits it is not necessary to change Prolog in any significant way. The higher order style of programming can be done using just Horn clauses. However, the popular approach of using `call/N` as the basic primitive is not ideal. To achieve as much flexibility as modern functional languages we must abandon `call/N` and use primitives such as `apply/3`. This lesson is also applicable to other logic languages. Other programming languages arguably have advantages over Prolog in areas such as syntax, debugging support, semantics and numerous language features. However, the ability to support higher order programming in itself should not be considered an advantage over Prolog.

Acknowledgements

Thanks to Leon Sterling for encouraging me to present this work.

References

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, Hempel Hemsted, UK, 1988.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CRS91] Anthony S.K. Cheng, Peter J. Robinson, and John Staples. Higher level meta programming in Qu-Prolog 3.0. In Koichi Furukawa, editor, *Proceedings of the*

Eighth International Conference on Logic Programming, pages 285–300, Paris, France, June 1991.

[GH95] T.S. Gegg-Harrison. Representing logic program schemata in λProlog. In Leon Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 467–481, Kanagawa, Japan, June 1995.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[KSJ93] M. Kirschenbaum, L. Sterling, and A. Jain. Relating logic programs via program maps. *Annals of Mathematics and Artificial Intelligence*, 8:229–245, 1993.

[Llo87] John W. Lloyd. *Foundations of logic programming (second, extended edition).* Springer series in symbolic computation. Springer-Verlag, New York, 1987.

[Nai91] Lee Naish. Adding equations to NU-Prolog. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture notes in computer science, pages 15–26, Passau, Germany, August, 1991. Springer-Verlag.

[NM88] G. Nadathur and D. Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 810–827, Seattle, Washington, August 1988.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* Logic Programming. MIT Press, Cambridge, Massachusetts, 1990.

[Sah93] Dan Sahlin. Mixtus: An automatic partial avaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[SHC95] Zoltan Somogyi, Fergus J. Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.

[SK93] L. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In Jean-Marie Jacquet, editor, *Constructing logic programs*, pages 127–140. Wiley, Chichester, England, 1993.

[SS86] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques.* Logic Programming series. MIT Press, Cambridge, Massachusetts, 1986.

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

[Tra94] T. Traill. *Transformation of logic programming.* Honours thesis, Department of Computer Science, University of Melbourne, Australia, November 1994.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Proceedings of the Second International Logic Programming Conference*, pages 127–138, Uppsala, Sweden, July 1984.

[TZ86] James Thom and Justin Zobel. NU-Prolog reference manual, version 1.0. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[War82] David H.D. Warren. Higher-order extensions to prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[War83] David H.D. Warren. An abstract Prolog instruction set. Tecnical Note 309, SRI International, Menlo Park, California, October 1983.