Functional Counterparts of some Logic Programming Techniques

Tatsuru Matsushita and Colin Runciman
Department of Computer Science, University of York
York, YO1 5DD, England
E-mail: {tatsuru,colin}@cs.york.ac.uk

ABSTRACT

Drawing on experience of translating a Prolog program into Haskell, a range of correspondences between logic and functional programs are discussed. Despite the differences of underlying paradigms, in many cases we can find close counterparts between the two programming schemes.

1. Introduction

To anyone surveying the entire range of programming languages and systems, logic programming and functional programming surely appear similar in many ways. Both have their origins in mathematical ideas rather than in the design of computing machines. Both are declarative, offering a programming style quite removed in principle from the low-level store-and-jump model of conventional imperative languages.

Yet in each paradigm there are quite distinctive programming techniques. At least, that is the impression often given by textbooks (though few discuss both paradigms in depth).

In this paper we look at some techniques often regarded as part of the distinctive repertoire of the logic programmer. We point out that there are counterparts of these techniques available to the functional programmer, using the tools of higher order functions and lazy evaluation. Our claims are modest. We are not offering a new translation principle. Not even the specific techniques are original. But few declarative programmers are well-versed in such correspondences, and their wider development and use could be a fruitful avenue of work.

Our paper has a specific context. It has been prompted by recent experience translating a moderately large logic program (about 13 k source lines) into a functional language. The application is a program transformation system, Starship [4]. We shall use this system from time to time as a source of illustration. Aside from our wish to study the correspondence between logic and functional programming, other motives for translating Starship included the hope of more direct and concise expression of transformation rules (through the use of higher order functions) and the potential for self-application (since the programs that Starship transforms are functional). The original system was programmed in SICStus Prolog [1], and the translation is written in Haskell [5]. In this paper, we adopt these languages throughout as the means of expression for logic and functional programs respectively.

The rest of the paper divides as follows. Section 2 discusses five different aspects of logic programming technique, and counterparts to them in a functional language. Section 3 briefly considers some aspects of logic programs that can pose difficulties for a functional translation. Section 4 concludes.

2. Techniques and counterparts

2.1. Backtracking and list of success technique

In logic programs, many predicate/procedure definitions involve several clauses, and for a specific call of such procedures more than one clause may match. Procedures can yield multiple results: there may be several ways of satisfying a goal, and each of them may result in different bindings of terms to variables. On the other hand, when the body-instance of an earlier clause fails, or failure occurs in the context from which the procedure was called, execution may need to backtrack to consider alternatives. A standard introductory example:

```
member(X,[X|_]).
member(X,[_|Xs]) :- member(X,Xs).
```

The goal `member(X,[1,2,3])` has three potential results, each binding `X` to a different element of the list. The first clause binds `X` to 1 without recursion. If this leads to failure in the calling context, backtracking leads to use of the recursive clause and hence further solutions.

In functional programs, though function definitions may involve several equations, at most *one* such equation applies for a given call. Reduction by this equation leads to *the* result; there is no backtracking. However, as Wadler points out [11], if we make the result of a function a *list*, and regard the items of this list as multiple results, lazy evaluation provides the counterpart of backtracking. There is no computation of the list beyond its first item unless and until the context demands it. An empty list represents failure, and list concatenation serves for the disjunction of alternatives. Returning to the `member` example, direct use of this idea leads to:

```
member [] = []
member (x:xs) = [x] ++
                member xs
```

For this example, we have ended up with an identity function. That is, `member` applications can be eliminated altogether using the equation:

```
member xs = xs
```

Starship Example This 'list of successes' technique finds many uses in the translation of Starship. For example, there is a predicate called `partially_rearrange` that extracts every possible form of an expression containing associative (and perhaps commutative) operators. So the expression

$$a + b + c$$

can be rearranged as:

$$a+(b+c) \quad \text{or} \quad (a+b)+c \quad \text{or} \quad b+(a+c) \quad \text{or} \quad \ldots$$

One of the main clauses defining `partially_rearrange` is:

```
partially_rearrange(E, R) :-
        chain(Op, E, C),
        (commutes(Op) ->
                aperm(C, P), associate(Op, P, R)
        ;
                associate(Op, C, R)).
```

Here `chain(Op, E, C)` derives a linear chain `C` of operands from an expression `E` whose outermost part is a tree of applications of the binary operator `Op`. The `aperm` goal generates permutations of operand in the case of a commutative operator, and `associate` generates alternative associations. This translates into the Haskell equation:

```
partially_rearrange e =
        let (op,c) = chain e in
        if commutes op then
                [r | p<-aperm c, r<-associate op p]
        else
                associate op c
```

While Prolog backtracks and returns each rearrangement as an alternative substitution for `R`, the Haskell solution yields a list of all possible rearrangements. Note the convenience of the list comprehension `[r | p<-aperm c, r<-associate op p]` in the commutative case.

2.2. Definite clause grammars and parser combinators

Most programs have input. Many inputs are textual, and legal inputs form a small language which the program must parse. In Prolog, parsers can be built by giving the rules of a *definite clause grammar* (DCG). DCG rules are converted into ordinary

predicates, which include input/output lists, by a preprocessor in the Prolog system. For example, the DCG rule

```
command(com(Name,Count)) --> name(Name),count(Count).
```

defines a simple command consisting of a command-name and repetition count assuming separate rules for `name` and `count`. There may be alternative rules for the same construct, with parsing by backtracking if necessary.

In a functional language such as Haskell, parsers can be built using *parser combinators* [6]. Parsers are functions which take an input string as argument and return a list of pairs of parsed value and rest of the string. A parser of a value `t` has type:

```
parser :: String -> [(t,String)]
```

This is the lazy list of success technique again, with an empty list representing a failed parse. Parser combinators combine parsers to make another 'greater' parser. Two such combinators are:

```
p1 ... p2 = \inp -> [((v1,v2),out2) |
                     (v1,out1) <- p1 inp,
                     (v2,out2) <- p2 out1]
p `using` f = \inp -> [(f v,out)|(v,out) <- p inp]
```

The combinator `(...)` builds a parser for two values expressed in sequence – the resulting values are simply paired. The combinator `using` allows structures other than the default pair types to result from parsing. The above example of a DCG rule for a command translates to Haskell as:

```
command = name ... count `using` \(n,c) -> Com n c
```

Both LHS and RHS here are *functions*. The argument in each case is a list of tokens. There is no need for preprocessing in a functional language; the combinators yield parsers directly as functional values.

Starship Parser The Starship parser routines in both Prolog and Haskell versions follow grammatical structure. In each case, the input lists do not appear in the descriptions. For example, Starship accepts *laws* such as:

$$mapmap\ f\ g \rightarrow map\ f\ .\ map\ g = map\ (f\ .\ g)$$

The Prolog DCG rule for parsing the top level syntax of a *law* is:

```
law_form(law(Name, Vars_Etc, B)) -->
        law_head(Name, Vars_Etc), ['->'],
        law_body(B).
```

This becomes in Haskell:

```
law_form =
        law_head ... lit"->" *..
        law_body
        `using`
        \((Var name,v),b) -> Law name v b.
```

The `lit` function used here builds a parser for a literal string.

One advantage of Prolog is that its unification mechanism provides 'two way' matching on the left hand sides: i.e. 'inputs' can be decomposed and 'outputs' constructed implicitly. In the DCG term `law_form`, the result is represented by the pattern `law(Name, Vars_Etc, B)`, and constructed from the terms from `law_head` and `law_body`, through unification. In Haskell, the pair structure `((Var name,v),b)` is constructed by the standard parsers; the abstract syntax is derived from it by the function: `\((Var name,v),b) -> Law name v b`.

2.3. Non-ground computation and partial application

In Prolog, data terms containing uninstantiated variables are 'first-class citizens': they can be passed, for example, as arguments in a goal or bound to variables as a result of executing a goal. This possibility can be exploited to good effect. One standard technique is the so-called *difference-list* [10] of the form `Xs\Ys` where `Xs` is a partial list, with `Ys` as its tail beyond some point. For example, perhaps `Xs=[1,2,3|Ys]`. By unifying another list with `Ys`, concatenation is accomplished without recursively reconstructing the earlier part of the list.

```
append(Xs\Ys,Ys\Zs,Xs\Zs).
```

To obtain a normal list data structure `Xs` from a difference-list `Xs\Ys`, one simply unifies `Ys` with `[]` the empty list.

In a functional language, data constructions containing variables are not by themselves well-formed expressions. It is not possible to pass as argument or obtain as result a data structure with some components unbound — although under lazy evaluation they may be bound to as yet unevaluated expressions. However, *functions* are 'first class citizens'. If we introduce a variable `v` under a lambda abstraction,

then the body of the function may of course build a data structure incorporating `v`. So a functional counterpart for the difference-list `[1,2,3|Ys]\Ys` is the lambda abstraction `\ys->([1,2,3]++ys)` or, using the section notation, simply the partial application `([1,2,3]++)`. By combining such partial applications using function composition, the functional programmer also achieves concatenation without recursively reconstructing the prefix of the list.

```
append f g = f . g
```

To obtain a normal list from a pre-section such as `(xs++)`, one simply applies it to `[]` as argument.

Starship Unparser In Starship we have to display transformed equations back to a screen, and must eventually write transformed source files. For this purpose `unparse` routines convert abstract syntax into lists of textual tokens. As a simple example, consider the unparser for type declarations that may precede a defining equations, such as the phrase from *newline* to *with* in:

$$\begin{array}{rrcl} Define & newline & :: & Char \\ and & tab & :: & Char \\ with & [tab, newline] & = & "\backslash t \backslash n" \end{array}$$

In the following (simplified) extract from the Prolog unparser, the difference-list is used to express textual representation of data structures.

```
unparse_types([], (Tail \ Tail)).
unparse_types([T], (Rest \ Tail)) :-
     unparse_type(T, (Rest \ [text('with ')|Tail])).
unparse_types([T|Ts], (Rest \ Tail)) :-
     unparse_type(T, (Rest \ [text('and ')|Tss])),
     unparse_types(Ts, (Tss \ Tail)).
```

This is translated into Haskell as:

```
unparseTypes [] = id
unparseTypes [t] =
        unparseType t . unparseText "with "
unparseTypes (t:ts) =
        unparseType t . unparseText "and " .
        unparseTypes ts
```

The results from `unparseTypes`, `unparseType` and `unparseText` are all functions of type `String->String`. By building a tree of function compositions, rather than a

tree of concatenations(++), repeated reconstruction of lists is avoided.

2.4. Meta-logical operators and higher order functions

Not only can Prolog terms include variables as first class citizens, there are metalogical operators that exploit the distinction between variables and non-variables. Other metalogical operators subvert the normal distinction between predicate terms and data. Both kinds of operator are used in the following definition:

```
map(_, [], []).
map(Term, [I|Is], [O|Os]) :-
        copy_term(Term, I^O^Instance),
        call(Instance),
        map(Term, Is, Os).
```

The technique here is to use metalogical operators to compensate for the lack of higher order programming. In a functional language, of course, defining map is rather more straightforward.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

So where have the metalevel operators gone? The purpose of `copy_term` in the logic program is to build a body appropriately instantiated for a given call. Performing this operation efficiently is a central concern of a functional language *compiler* [8]; it does not have to be explicit in a functional program. The purpose of the `call` operator is to coerce a data term into a predicate term. In the functional language there is no need for the data structure in the first place, as the functional value is passed directly.

Starship Example Several other procedures using similar metalogical techniques are defined in Starship. These routines are widely used throughout the program. To give just one example of the use of `map`, here is a simple procedure accessing the abstract syntax of definitions.

```
Cs define_ids Ns :-
        Cs define_vars Vars,
        map(I^O^(I has_id O), Vars, Ns).
```

In Haskell this becomes:

```
define_ids cs   = map has_id (define_vars cs)
```

2.5. Clause manipulation and continuation passing style

Another characteristic of predicate-procedures in Prolog is that their definitions can be *dynamic*. Clauses defining such predicates can be removed or added, during the execution of the program, using metalogical primitives such as `retract` and `assert`. Heavy use of dynamic predicates is poor style, but in many applications the best programming solution involves a small number of dynamic predicates. For example, options might be switched on or off by

```
switchOn(O)  :- assert(option(O)).
switchOff(O) :- retract(option(O)).
```

so that the success or failure of `option()` goals can be used to control various program components without any explicit mention of the structure in which options are held.

In a functional program, all the defining equations are fixed throughout execution. There are no 'dynamic functions' and no metalevel operators to manipulate defining equations as values. Any part of the execution state that is dynamic can only be communicated to a section of the program by passing arguments. To represent 'change' of such dynamic state we can augment a function's result with a 'new state' value. For example, redefining a function f of type `a->b` to 'change' a state of type `s`, its type becomes `a->s->(b,s)`. In the degenerate case where the entire purpose of the function is state-change, it might be given the type `s->s`. For the options example, the state could be a list of active options:

```
switchOn  o os = o : os
switchOff o os = os \\ [o]

option o os = o `elem` os
```

But this approach leads to a global cluttering of the program with explicit mentions of the option list and expressions to construct or access tuples containing it. To avoid this problem, instead of passing the option list to the program at large, we pass parts of the program at large to the option functions, as *continuations*.

```
switchOn o k os  = k (o : os)
switchOff o k os = k (os \\ [o])

ifoption o ky kn os =
        (if o `elem` os then ky else kn) os
```

The argument order permits convenient composition with the state arguments left implicit. We may write, for example:

```
switchOn trace .
switchOn help .
ifoption logfile ( etc ) ( etc )
```

Starship object program In Starship, the most important use of dynamic clauses in the original Prolog-based version is to represent the *object program* being transformed. The effect of interpreting almost every command in Starship includes accessing one or more such clauses — possibly retracting some of them and asserting transformed replacements. In the Haskell translation, the object program constitutes the main part of a state for which continuation-passing operations are defined.

The type of the main function that interprets commands is:

```
exec    :: Command -> (ClauseDb -> a) -> ClauseDb -> a
```

where `ClauseDb` is the type of the 'database' of defining clauses for the program being tansformed. The polymorphic type `a` is usually text to be displayed to the Starship user.

Database manipulation, or, in general, state change, is a typical *impure* process. Wadler [12] describes *monadic* style programming – another way to model *impure* effects in a purely functional language.

3. Some problematic techniques

3.1. Cut operator

There is no single counterpart of the *cut* operator that is the most appropriate choice in all cases. *Cuts* are used in a variety of ways in Prolog programs, to prevent useless or inappropriate backtracking. For example, a predicate which computes the sum of positive integers from 1 up to a given limit `N`, might be defined as:

```
sum_to(N, 0) :- N =< 0, !.
sum_to(N, R) :-
        N1 is N-1,
        sum_to(N1, R1),
        R is N+R1.
```

The goal `sum_to(1,R)` succeeds with `R = 1` as the only solution. But without the *cut* operator at the end of the 1st line, backtracking would yield *again* `R = 1`, and then the spurious solutions `R = 0, R = -2, R = -5` and so on.

The `sum_to` example illustrates a common pattern: the initial goals of a body define a precondition for this clause to apply, and are separated by a cut from the remaining cut-free goals. This corresponds to the use of *guards* in Haskell. The `sum_to` predicate translates into the Haskell function:

```
sum_to n
        | n <= 0 = 0
        | otherwise = n + sum_to (n-1)
```

In this translation the `=` sign after the guard `n <= 0` corresponds to the cut operator. Once we get to the right hand side of this `=`, we are committed to use this equation.

We saw earlier the use of *list of successes* for backtracking. When this representation is used, a *cut* can be as simple as discarding the tail of a list. Suppose we define:

```
cut [] = []
cut (x:xs) = [x]
```

In a predicate clause such as

```
reduce(L,R) :-
        equation(E), match(L,E,R), !.
```

the *cut* is used to avoid multiple results when there could be several equations that match (each perhaps in several ways). One translation is

```
reduce l =
        cut [r|e <- equations, r <- match e l]
```

The *cut* operator can be especially important in parsers for efficiency reasons. If backtracking is possible, the input stream has to be kept. However, it is frequently the case that after reading a small number of tokens, we can recognise there is no possibility of backtracking: we reach a point where we can place a *cut*.

Rojemo [9] explains a *cut combinator* used in his *continuation-based* parsing sheme: instead of discarding the tail of a list he discards a *failure continuation* retaining only a *success continuation.*

3.2. Polymodality

Arguments of some Prolog predicates can be used as input in one case and as output in another case. This is *polymodality.* For example, the standard definition of `append` is:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

As its name suggests `append` can be used in a goal such as `append([1,2], [3,4], Zs)` with the result that `[1,2,3,4]` is bound to `Zs`. But in the case of

```
member(X,Ys) :- append(As,[X|Xs],Ys).
```

and a goal such as `member(X, [1,2,3])` the predicate `append` is given a whole list `Ys` and splits it into the list-pair `(As,[X|Xs])`.

Because of the one-directional characteristic of functions, polymodal predicates do not have a direct counterpart in a functional language. A typical definition of `append` in Haskell is:

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

In contrast to Prolog, the related list-splitting function requires a separate definition with a more complex structure:

```
split :: [a] -> [([a],[a])]
split []    = [([],[])]
split (h:t) =
  ([],h:t) : [(h:l,r)|(l,r)<-split t]
```

The two functions have completely different types and it is impossible to use the ordinary `append` function in place of `split`. Darlington's *absolute set abstraction* [2], an extension of the functional language Hope, allows a definition of the form:

```
split zs = [(xs,ys) | append xs ys == zs]
```

But evaluation using such a definition takes us outside the realm of standard implementation methods – ie. compiled graph reduction.

In Starship, simple predicates to access data structures, such as `has_body`

```
func(_, B) has_body B
```

are very common. This predicate is used in two ways: in a goal with both instances of `B` instantiated, it is equivalent to the function

```
has_body :: Clause -> Bool
has_body (Func _ b) b1 = (b==b1)
```

When only the first `B` is instantiated, the equivalent function would be:

```
body_of :: Clause -> Expr
body_of (Func _ b) = b
```

In the deeper computational workings of Starship, however, there are not many places where the polymodal characteristics of predicates are exploited. By the time one has considered issues such as the order of solutions, and the placement of cuts for efficiency, there are few procedures in practice for which there are several useful modes of computation.

4. Discussion and conclusion

In the course of translating a large and structurally complex logic program to a functional language, we encountered a variety of programming techniques. In the majority of cases, it turns out that there are corresponding techniques in a functional language with higher order functions and lazy evaluation.

We have not made precise our notion of 'correspondence', though we are certainly interested in doing so. For example, Felleisen [3] studies the question of programming language power by asking about the extent to which program structure alters under translation. We have only accepted as counterparts those techniques that allow the translated programs to have similar structure to the originals. The question of execution efficiency is also important, though more difficult to address: implementation methods in both functional and logic programs remain active and rich fields of research.

In a recent paper, Marchiori [7] considers a limited class of Prolog programs, and defines a translation from these into functional programs. The translation not only preserves termination and computed answers, but also keeps computational complexity. Resulting functional programs *compute in a similar way* to the logic programs from which they are derived. Translation is carried out by exact rules suggesting

the possibility of an automatic translator. Although Marchiori's class of *functionally moded* predicates is rather limiting, it does include the predicate `split-append` that we mentioned in the section on *Polymodality*.

A topic we have only mentioned in passing is the *strong typing* of most functional languages, including Haskell, though the effects of working under a type system are all-pervasive in a program. For example, even if auxiliary state arguments are not named explicitly in defining equations, functions that potentially 'change state' can be clearly identified by their types; in Prolog, a thorough survey is required to detect all possibility of state changes. On the other hand, in Prolog clauses, a simple comma suffices to combine subgoals whatever type of computation they engage in; in Haskell, every expression has its type and even with the use of *type classes* there are limits to the overloading of combinator symbols.

Acknowledgements We would like to thank to the anonymous reviewers of the workshop for their instructive comments.

References

1. M. Carlsson and J. Widen. SICStus Prolog Users Manual. Technical Report R88007, Swedish Institute of Computer Science, 1988.
2. J. Darlington and Yi-Ke. Guo. Narrowing and unification in functional programming: an evaluation mechanism for absolute set abstraction. *LNCS*, 355, April 1989.
3. M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35–75, December 1991.
4. M.A. Firth. *A Fold/Unfold transformation system for a non-strict language.* PhD thesis, Department of Computer Science, University of York, September 1990. YCST.
5. P. Hudak *et. al.* Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.
6. G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–43, July 1992.
7. M. Marchiori. The Functional Side of Logic Programming. In *Proceedings FPCA'95 Conference on Functional Programming Language and Computer Architecture*, June 1995.
8. S. L. Peyton Jones and D. Lester. *Implementing Functional Languages.* Prentice Hall, 1992.
9. N Rojemo. *Garbage collection, and memory efficiency, in lazy functional languages.* PhD thesis, Department of Computing Science, Chalmers University of technology, 1995.
10. L. Sterling and E Shapiro. *The Art of Prolog.* The MIT Press, 1986.

11. P. Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannud, editor, *Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, France, September 1985. Springer-Verlag(LNCS201).
12. P. Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Tutorial text of the 1st International Spring School on Advanced Functional Programming Techniques*, pages 24–52, Sweden, May 1995. Springer-Verlag(LNCS925).