Higher-order transformation of logic programs ⋆

Silvija Seres and Mike Spivey

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.
{Silvija.Seres, Mike.Spivey}@comlab.ox.ac.uk
http://www.comlab.ox.ac.uk/oucl/work/silvija.seres

Abstract. It has earlier been assumed that a compositional approach to algorithm design and program transformation is somehow unique to functional programming. Elegant theoretical results codify the basic laws of algorithmics within the functional paradigm and with this paper we hope to demonstrate that some of the same techniques and results are applicable to logic programming as well.

1 The problem

The Prolog predicates *rev1* and *rev2* are both true exactly if one argument list is the reverse of the other.

$$\begin{array}{ll}
\mathit{rev1}([\,],[\,]). & \mathit{rev2}(A,B) :- \ \mathit{revapp}(A,[\,],B). \\
\mathit{rev1}([X|A],C) :- & \mathit{revapp}([\,],B,B). \\
\quad \mathit{rev1}(A,B), \mathit{append}(B,[X],C). & \mathit{revapp}([X|A],B,C) :- \\
 & \quad \mathit{revapp}(A,[X|B],C).
\end{array}$$

These two predicates are equal according to their declarative interpretation, but they have a very different computational behaviour: the time complexity for *rev1* is quadratic while for *rev2* it is linear. The aim of this paper is to present a general technique for developing the efficient predicate from the clear but inefficient one, in this and similar examples.

Arguably the most general transformational technique in logic programming is the "rules and strategies" approach [7]. In this technique the *rules* perform operations such as an unfolding or folding of clause definitions, introduction of new clause definitions, deletion of irrelevant, failing or subsumed clauses, and certain rearrangements of goals or clauses. Subject to certain conditions, these rules can be proved correct relative to the most common declarative semantics of logic programs. The application of the transformation rules is guided by meta-rules called *strategies*, which prescribe suitable sequences of basic rule applications. The main strategies involve

⋆ Extended Abstracts of LOPSTR 2000, Tenth Int'l Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

tupling of goals that visit the same data structure in a similar way, generalisation of goals in a clause in order to fold them with some other clause, elimination of unnecessary variables, and fusion of predicates defined by two independent recursive predicates into a single predicate. These strategies are used as the building blocks of more complex transformation techniques, and for limited classes of predicates these complex strategies have been well understood and classified and can be seen as the backbone of a compositional method for transforming logic programs.

Our transformational example can indeed be solved by the rules and strategies approach, together with mathematical induction, needed to prove the associativity of *append* on which the transformation depends. The basic strategies involved are tupling and generalisation, and the derivation is simple and semantically correct relative to the least Herbrand model of the two programs. However, there are a few methodological problems in this approach: first, the declarative semantics does not quite capture the behaviour of logic programs when they are evaluated under the standard depth-first search strategy, and we have no clear measure of the reduction of the computation complexity. Second, the application of induction requires a separate form of reasoning. But maybe most importantly, if we did not know of this particular combination of strategies, there would be no systematic methodological approach to guide us in the derivation. As far as we know, there are no general results regarding what complex strategies can be applied for families of transformationaly similar predicates. Below we outline a generalised approach to logic program transformations, and argue that such an approach should be based on higher-order predicates and their properties.

2 The proposed solution

The problem described above has been recently explored and explained for functional programs in [2]. These results build on the ample heritage of program transformation in the functional programming community and are based on laws of algebra and category theory. According to this algebra of functional programming, the program transformation in the example above can be seen as an instance of a more general transformational strategy, valid for an entire family of programs based on functions *foldl* and *foldr* and parametric in the data structure. Algebraic laws regarding such higher-order functions prove to be highly versatile for functional program transformations.

With this paper we begin an investigation of how these results can be translated to logic programs, and present two examples where this technique has been successfully applied to derive efficient implementations of logic programs from their specifications.

We base our transformation methods on a translation of logic programs into lazy functional programs in Haskell that we have described elsewhere [11]. This embedding and a set of algebraic laws valid for the basic combinators of the embedding are sketched in section 3. There are two main advantages in using this functional embedding for logic program transformations. The first one is that it allows us to reason about logic programs in a simple calculational style, using rewriting and the algebraic laws of combinators. The second, and the more interesting reason, is that the embedding has the advantage that many higher-order predicates are easily expressible as functions, since Haskell functions can accept our predicates as arguments. We can implement the generalised "prepackaged recursion operators" *foldl*, *foldr*, *map* etc. as functions from predicates to predicates, and thereby get the opportunity to use their algebraic properties for program transformation. This approach avoids the problems related to higher-order unification, while it gives us the power of generic programming and provides the appropriate language and level of abstraction to reason about logic program transformation. Even though each derivation can be performed in a first-order setting, the *general* strategies guiding the program transformations depend essentially on the higher-order functions. We argue that, as in functional programming, also in logic programming it is the properties of the generic recursion operators that yield the generic transformation strategies.

In sections 4 and 5 we show two examples where the laws are used in logic program transformation, and in the final section we discuss related work and suggest directions for future research.

3 The embedding

Using a translation technique related to Clark's completion [4], any logic program can be translated to a set of functions in a lazy functional language in such a way that the declarative semantics of the logic program is preserved. In [11] we describe an implementation of such a embedding where a logic program is translated to a lazy functional program using a simple combinator library with only four combinators $\doteq$, $\exists$, & and $\|$; these combinators perform respectively unification, introduction of local variables, conjunction of literals in a clause body, and disjunction of clauses. The combinators are implemented in such a way that they exactly mimic the SLD-resolution process.

With a some minimal syntactic differences, the embedded predicates have essentially the same shape as the original ones. For example, the standard predicate *append* is implemented in a logic program as:

$$\mathit{append}([\ ], XS, XS).$$
$$\mathit{append}([X|XS], YS, [X|ZS]) :- \mathit{append}(XS, YS, ZS).$$

The patterns and repeated variables from the head of each clause can be replaced by explicit equations written at the start of the body. Then each head contains only a list of distinct variables, and renaming can ensure that these lists of variables are same. We translate such a logic predicate to a Haskell function by joining the clause bodies with the $\|$ operation and the literals in a clause with the & operation, by existentially quantifying any variables that appear in the body but not in the head of a clause, and by using $\doteq$ to compute unification:

$$\begin{aligned}
&\mathit{append}(p, q, r) = \\
&\qquad (p \doteq \mathit{nil} \;\&\; q \doteq r) \\
&\quad \| \; (\exists x, xs, zs \to p \doteq \mathit{cons}(x, xs) \;\&\; r \doteq \mathit{cons}(x, zs) \;\&\\
&\qquad\qquad\qquad \mathit{append}(xs, q, zs)).
\end{aligned}$$

The evaluation of this Haskell function mimics the computation of a corresponding goal in a logic program by SLD-resolution. The function *append* takes as input a tuple of terms and a substitution (representing the state of knowledge about the values of variables at the time the predicate is invoked), and produces a collection of substitutions, each corresponding to a solution of the predicate that is consistent with the input substitution. The collection of substitutions that is returned by *append* may be a lazy stream to model the depth-first execution model of Prolog, or it may be a search tree in a more general execution model. Other models of search may also be incorporated: for example, there is an implementation of a breadth-first traversal of the SLD-tree that uses lazy streams of finite lists of answers.

The embedded predicate *append* behaves like a relation, i.e., one can compute the answers to goals $append([1], y, [1, 2])$ or $append(x, y, [1, 2])$.

The implementation of each of the four combinators of the embedding is strikingly simple, and can be given a clear categorical description which yields nice computational rules: it can be easily proved that (irrespectively of the search model) the operators and the primitive predicates *true* and *false* enjoy some of the standard laws of predicate calculus, e.g. & is associative and has *true* as its left and right unit and *false* as its left zero, $\|$ is associative and has *false* as its left and right unit and & distributes through $\|$ from the right. Other properties that are satisfied by the connectives of propositional logic are not shared by our operators, because the answers are produced in a definite order and with definite multiplicity. These laws are, of course, valid in the declarative reading of logic programs. Since procedural equality is too strict when reasoning about predicates with different complexity behaviour, we will permit in our transformational proofs also the use of the laws that are only valid in the declarative semantics.

These algebraic laws can be used to prove the equivalence of two logic programs with equivalent declarative reading. The basic idea is to embed both programs in this functional setting, and then use the laws to show that the two functions satisfy the same recursive equation. Further, a result exists that guarantees that all guarded recursive predicate definitions have a unique solution. The proof for the uniqueness of fixpoints is based on metric spaces and a certain contraction property of guarded predicates. We present this result elsewhere [10].

4 Example 1: reverse

The standard definition of the naive reverse predicate has quadratic time complexity:

$$\begin{aligned}
&rev1(l1, l2) = \\
&\qquad (l1 \doteq nil \;\&\; l2 \doteq nil) \\
&\quad \| \; (\exists x, xs, ys \rightarrow \; l1 \doteq cons(x, xs) \;\&\; rev1(xs, ys) \;\& \\
&\qquad\qquad append(ys, cons(x, nil), l2)).
\end{aligned}$$

A better definition of reverse uses accumulators and runs in linear time:

$$
\begin{aligned}
&\mathit{rev2}(\mathit{l1}, \mathit{l2}) = \mathit{revapp}(\mathit{l1}, \mathit{nil}, \mathit{l2})\\
&\mathit{revapp}(\mathit{l1}, \mathit{acc}, \mathit{l2}) =\\
&\qquad (\mathit{l1} \doteq \mathit{nil} \;\&\; \mathit{l2} \doteq \mathit{acc})\\
&\quad\; \| \; (\exists x, \mathit{xs} \to \mathit{l1} \doteq \mathit{cons}(x, \mathit{xs}) \;\&\; \mathit{revapp}(\mathit{xs}, \mathit{cons}(x, \mathit{acc}), \mathit{l2})).
\end{aligned}
$$

We can prove these two definitions equivalent by using the previously mentioned algebraic laws together with structural induction. This approach is similar to the rules and strategies approach for logic program transformation. However, there is a shorter and more elegant way of proving these predicates equal, by resorting to program derivation techniques based on higher-order *fold* predicates and their properties. Such fold operators have proved to be fundamental in functional programming, partly because they provide for a disciplined use of recursion, namely a recursive decomposition that follows the structure of the data type. They also satisfy a set of laws that are crucial in the functional program transformation proofs, and we will rely on one of those laws in our derivation. The outline of the proof is:

$$
\begin{aligned}
&\mathit{rev1}(\mathit{xs}, \mathit{ys}) && \\
&= \mathit{foldRList}\ (\mathit{snoc}, \mathit{nil})\ (\mathit{xs}, \mathit{ys}) && \text{by defn. of } \mathit{foldRList} \text{ and } \mathit{snoc}\\
&= \mathit{foldLList}\ (\mathit{flipapp}, \mathit{nil})\ (\mathit{xs}, \mathit{ys}) && \text{by duality law (1), see below}\\
&= \mathit{revapp}(\mathit{xs}, \mathit{nil}, \mathit{ys}) && \text{by defn. of } \mathit{foldLList}\\
&= \mathit{rev2}(\mathit{xs}, \mathit{ys}) && \text{by defn. of } \mathit{rev2}
\end{aligned}
$$

and we justify each of the steps below.

The definitions of some families of higher-order predicates, for example the map and fold predicates over lists or other data structures, can be made without any extensions to the implementation of our embedding. They can be implemented using Haskell's higher-order functions on predicates, so we do not need to resort to the higher-order unification machinery of, say, λ-Prolog. For example, the predicate *foldRList*, which holds iff the predicate p applied right-associatively to all the elements of the list l yields the term res, could be defined as:

$$
\begin{aligned}
&\mathit{foldRList}\ (p, e)\ (l, \mathit{res}) =\\
&\qquad (l \doteq \mathit{nil} \;\&\; e \doteq \mathit{res})\\
&\quad\; \| \; (\exists x, \mathit{xs}, r \to l \doteq \mathit{cons}(x, \mathit{xs}) \;\&\\
&\qquad\qquad \mathit{foldRList}\ (p, e)\ (\mathit{xs}, r) \;\&\; p(x, r, \mathit{res}))
\end{aligned}
$$

where (p, e) are the higher-order parameters to the function *foldRList* and (l, res) are the arguments to the resulting predicate. The predicate p corresponds to a binary function to be applied to the consecutive list elements, and e denotes the initial element used to start p "rolling". For example, the function *foldRList* $(\mathit{add}, 0)$ applied to $([2,7,8], \mathit{res})$ produces the predicate $r_1 \doteq 0 \;\&\; \mathit{add}(8, r_1, r_2) \;\&\; \mathit{add}(7, r_2, r_3) \;\&\; \mathit{add}(2, r_3, \mathit{res})$; when invoked with the appropriate input substitution (say the empty one), this predicate has the effect that *res* unifies with the term denoting the numeral 17.

In the first step of the proof, we use the following predicate *snoc*:

$$snoc\ (x, l, res) = append(l, cons(x, nil), res)$$

By unfolding $foldRList\ (snoc, nil)\ (xs, ys)$ in the second line in the proof and using the standard algebraic laws, we arrive to the same guarded recursive definition as for *rev1*. By the uniqueness of the fixpoints we then conclude the equality of the two predicates involved in the initial step of the proof.

The next step in the proof involves a transition from *foldRList* to another higher-order predicate, *foldLList*. This left-associating fold over lists could be defined as:

$$\begin{aligned}
&foldLList\ (p, e)\ (l, res) = \\
&\qquad (l \doteq nil\ \&\ e \doteq res) \\
&\quad \| \ (\exists x, xs, r \rightarrow l \doteq cons(x, xs)\ \& \\
&\qquad\qquad p(e, x, r)\ \&\ foldLList\ (p, r)\ (xs, res))
\end{aligned}$$

Roughly, the function call $foldLList\ (add, 0)\ ([2, 7, 8], res)$ would return the predicate $add(0, 2, r_1)\ \&\ add(r_1, 7, r_2)\ \&\ add(r_2, 8, r_3)\ \&\ r_3 \doteq res$.

The second step in the main proof is an instance of the so-called duality law:

$$foldRList\ (f, e)\ (l, res) = foldLList\ (g, e)\ (l, res) \tag{1}$$

where f is replaced by *snoc*, g by *flipapp*, and e by *nil*. The law above holds if f, g and e satisfy the following requirements: f and g must associate with each other, and $f(x, e, res)$ must equal $g(e, x, res)$ for all x and res. The predicates f and g associate with each other iff the predicates $\exists t \rightarrow (f(x, t, res)\ \&\ g(y, z, t))$ and $\exists t \rightarrow (g(t, z, res)\ \&\ f(x, y, t))$ are equal. In functional notation this corresponds to $f(x, g(y, z)) = g(f(x, y), z)$. We can prove (1) in a simple rewriting proof, using the definitions of *foldRList* and *foldLList* and the assumptions about f, g and e.

Returning to our example, we need to check that the duality law really is applicable, so we now prove that the predicates *snoc* and *flipapp* and term *nil* satisfy the requirements for f, g and e. If *flipapp* is defined as:

$$flipapp\ (l, x, res) = append(cons(x, nil), l, res)$$

then we unfold the definition of both functions, and use the associativity of *append* in step marked with (*), to get:

$$\begin{aligned}
&(\exists t \rightarrow (snoc(x, t, res)\ \&\ flipapp(y, z, t))) \\
&= (\exists t \rightarrow (append(t, cons(x, nil), res)\ \&\ append(cons(z, nil), y, t))) \\
&= (\exists t \rightarrow (append(cons(z, nil), t, res)\ \&\ append(y, cons(x, nil), t))) \quad (*) \\
&= (\exists t \rightarrow (flipapp(t, z, res)\ \&\ snoc(x, y, t)))
\end{aligned}$$

and similarly for $snoc(x, nil, res)$ and $flipapp(nil, x, res)$. The associativity of *append* used in (*) can be shown by induction on the list argument *res*.

For the penultimate step in the proof, we first prove that $revapp(l, acc, res)$ equals $foldLList\ (flipapp, acc)\ (l, res)$. We can prove this by a simple induction proof, in which we do induction on the argument l to show that the two predicates rewrite to the same guarded recursive definition, and then by referring to the fixpoint theorem once again to conclude that they are equal. Then, instantiating the arbitrary term acc in $foldLList$ to the term nil, we get exactly the $foldLList\ (flipapp, nil)\ (xs, ys)$ from the third line of the proof, so we can rewrite this to a call to $revapp(xs, nil, ys)$ in the fourth line. The final step follows directly from the definition of $rev2$.

The accumulation program transformation used in this example is closely related to the continuation passing programming style. It is argued informally in [12] that accumulators are often just a data structure representing a continuation function, while [3] gives a survey of many ways of representing accumulators and presents continuations as one of them. From a purely algebraic view, both techniques exploit the associativity property and an existence of a neutral element of some monoid. The *reverse* logic predicate can easily be expressed in terms of continuations in our embedding; we can implement the continuations as λ-abstractions over predicates:

$$
\begin{aligned}
&rev3(l1, l2) = revc\ (\lambda x.x \doteq l2)\ (l1, l2)\\
&revc\ (f)\ (l1, l2) =\\
&\qquad (l1 \doteq nil \mathbin{\&} f(nil))\\
&\quad \parallel (\exists x, xs \to l1 \doteq cons(x, xs)\\
&\qquad\qquad revc\ (\lambda y.(\exists r \to append(y, cons(x, nil), r) \mathbin{\&} f(r)))\ (xs, l2))
\end{aligned}
$$

This definition of $rev3$ can also be proved equal to $rev2$ by techniques similar to those used in functional programming as described in [12].

5 Example 2: sort

The following example is inspired by [2]. We start with the standard implementation of the *naiveSort* predicate that uses the 'generate-and-test' method to sort a list:

$$
\begin{aligned}
&naiveSort(l1, l2) = perm(l1, l2) \mathbin{\&} isSorted(l2)\\
&isSorted(l) =\\
&\qquad (l \doteq nil)\\
&\quad \parallel (\exists x \to l \doteq cons(x, nil))\\
&\quad \parallel (\exists x, y, l2 \to l \doteq cons(x, cons(y, l2)) \mathbin{\&} le(x, y) \mathbin{\&}\\
&\qquad\qquad isSorted(cons(y, l2)))
\end{aligned}
$$

where *perm* has the standard definition, using the auxiliary predicate *delete*. We now wish to show that *naiveSort* is equivalent to its more efficient variant *iSort*, which performs insertion sort. Given a predicate $insert(x, zs, l2)$ which is true if the sorted list $l2$ is the result of inserting the element x in the appropriate position in the sorted list zs, the usual implementation of the $iSort$ predicate is:

$$
\begin{aligned}
& iSort(l1, l2) = \\
& \qquad (l1 \doteq nil \;\&\; l2 \doteq nil) \\
& \quad \| \; (\exists x, ys \rightarrow l1 \doteq cons(x, ys) \;\& \\
& \qquad\qquad iSort(ys, zs) \;\&\; insert(x, zs, l2))
\end{aligned}
$$

The outline of this derivation is similar to the previous example, except that the essential step this time uses the fusion law for fold instead of the duality law:

$$
\begin{aligned}
& naiveSort(l1, l2) \\
& = isSorted(l2) \;\&\; perm(l1, l2) && \text{by defn. of } naiveSort \\
& = isSorted(l2) \;\&\; foldRList\ (add, nil)\ (l1, l2) && \text{by defn. of } foldRList \\
& = foldRList\ (insert, nil)\ (l1, l2) && \text{by fusion (2), see below} \\
& = iSort(l1, l2). && \text{by defn. of } iSort
\end{aligned}
$$

Here add is defined as the converse of $delete$. The most interesting step in the derivation involves the fusion law for $foldRList$. Assume that the predicates f, g and h, and term e, are such that $f(e)$ holds and that $f(res) \& g(x, y, res)$ equals $h(x, y, res) \& f(y)$ for all terms x, y and res (in functional notation, $f(g\ x\ y) = h\ x\ (f\ y))$. Then, the fusion law states that:

$$
f(res) \;\&\; foldRList\ (g, e)\ (l, res) = foldRList\ (h, e)\ (l, res) \tag{2}
$$

This law can be proved by induction. If we now insert our predicate $isSorted$ for f, add for g, $insert$ for h, and nil for e, the third step in the main proof is a direct application of this law. It can be easily shown that the chosen predicates satisfy the conditions for f, g, h and e.

Following a similar approach, we can also derive the equivalence of the naive sort and, for example, $quickSort$ or $selectionSort$. Both of these derivations rely on the fusion law, but they are algebraically slightly more advanced than the above derivation of $iSort$ because they also involve properties of unfold predicates. The derivation of $quickSort$ uses fold and unfold predicates on trees. The reason for this is that even though $quickSort$ is usually represented as a flat recursive predicate, it has a compositional form which is basically a sort on trees where the intermediate tree data type has been eliminated.

6 Related and further work

An embedding of logic programs to a functional setting has been explored by Wand [13], Baudinet [1], Ross [9] and Hinze [6], but with different motives. They all pursue an algebraic semantics for logic programming, but they do not attempt to generalise their techniques to transformational strategies. The examples presented here are mostly inspired by Bird and de Moor's work [2] on similar program synthesis and transformation techniques for functional programming. The contribution of this paper is in a translation of these techniques to logic programming. Related work in functional setting includes, among others, Wand's work [12] on continuation based program transformation techniques. In a logic programming setting, Pettorossi and Proietti present in [8] a particular transformation strategy based on an of introduction of lists and higher-order predicates on lists.

This approach to logic program transformation opens several areas for further research. One is to apply these transformational techniques to constraint programming, which can also be translated to a functional setting by means of our embedding. Another direction is to examine what other results from [2] we can transfer to logic programming. Yet another direction is to build automatic tools for logic program transformation based on the algebraic approach described here; this has been successfully done for functional programs in [5]. All of these direction motivate a further cross-fertilisation of methods for program transformation between the two declarative paradigms.

References

1. M. Baudinet. *Logic Programming Semantics Techniques and Applications.* PhD thesis, Stanford Univeristy, 1989.
2. R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.
3. E. Boiten. The many disguises of accumulation. Technical Report 91-26, University of Nijmegen, 1991.
4. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
5. O. de Moor and G. Sittampalam. Generic program transformation. In *Procs. 3rd International Summer School on Advanced Functional Programming*, 1998.
6. R. Hinze. Prological features in a functional setting - axioms and implementations. In *Proc. of FLOPS'98*, Fuji, 1998.
7. A. Pettorossi and M. Proietti. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter Transformation of Logic Programs, pages 697–787. Oxford University Press, 1998.
8. M. Proietti and A. Pettorossi. Program derivation via list introduction. In *Proceedings of IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, Le bischenberg, France, 1997.
9. B.J. Ross. Using algebraic semantics for proving Prolog termination and transformation. *Proceedings of the UKALP 1991*, 1991.
10. S. Seres. *Algebraic Techniques for Logic Programming.* PhD thesis, University of Oxford, 2000.
11. J.M. Spivey and S. Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell'99*, Paris, France, 1999.
12. M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1), 1980.
13. M. Wand. A semantic algebra for logic programming. Technical Report 148, Indiana University, 1983.