Higher-order Transformation of Logic Programs

Silvija Seres and Michael 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 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 method 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 general 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 functional programming 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 the functions *foldl* and *foldr* and parametric in the data structure. Algebraic laws regarding such higher-order functions prove to be highly versatile for transformation of functional programs.

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. Using a translation technique related to Clark completion [4], any logic program can be translated into 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 an embedding, 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 clause bodies. The combinators are implemented in such a way that they exactly mimic the SLD-resolution process.

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 transformation of logic programs. 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 many predicates are easily expressible using higher-order functions that accept more basic predicates as arguments. We can implement the general "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 particular 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, so also in logic programming it is the properties of generic recursion operators that yield 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 further research.

3 The Embedding

As discussed earlier, the standard approaches to logic program transformation, based purely on the declarative semantics, cannot answer questions of termination and efficiency, and consequently they can offer no guidance in deriving a more efficient program. At the other extreme, techniques that rely on operational semantics result in a tedious, low level treatment of program transformation, one that is unintuitive and difficult to generalise. We propose an intermediate approach, using a third kind of semantics for logic programs that captures the essentials of the operational behaviour of logic programs, but remains abstract enough for the techniques to be general and simple. We call this the *algebraic semantics* for logic programs.

The algebraic semantics arises from the Clark completion of the logic program. This completion translates any definite logic program to an equational, rather than implicational, form using only four basic operators: &, $\|$, $\exists$ and $\doteq$. The meaning that &, $\|$, $\exists$ are assigned by Clark is their standard interpretation in first order logic, so & is conjunction, $\|$ is disjunction, and $\exists$ is existential quantification. Clark defines $\doteq$ as equality, and gives an axiomatization of equality that is equivalent to unification. This is the declarative, or denotational, reading of the Clark completion of a logic program.

With a some minimal syntactic differences, the Clark completion of a predicate has essentially the same shape as the original one. For example, the standard predicate *append* is implemented in a logic program as:

$$\begin{array}{l} append([\ ], XS, XS). \\ append([X|XS], YS, [X|ZS]) :- \ append(XS, YS, ZS). \end{array}$$

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. Further steps of the completion consist of joining the clause bodies with the $\|$ operator and the literals in a clause with the & operator, existentially quantifying any variables that appear in the body but not in the head of a clause:

$$\begin{array}{l} append(p, q, r) = \\ \qquad (p \doteq nil \mathbin{\&} q \doteq r) \\ \quad \| \ (\exists x, xs, zs.\ p \doteq cons(x, xs) \mathbin{\&} r \doteq cons(x, zs) \mathbin{\&} \\ \qquad\qquad\qquad append(xs, q, zs)). \end{array}$$

Using Clark's form of a logic program allows us to perform equational reasoning, which simplifies program transformation. It also emphasises the similarities between logic and functional programming. In addition, it singles out &, $\|$, $\exists$ and $\doteq$ as the four basic operators which are necessary to achieve the expressiveness of logic programs. By focusing on each of these separately, we gain a *compositional* understanding of the computational behaviour of logic programming, as opposed to the usual monolithic view of the computation given by the standard procedural readings of a logic program such as SLD resolution.

The declarative reading ignores evaluation issues, but these four operators can also be given an operational reading, one that is correct with respect to SLD resolution, for example. This operational reading can be specified in many ways: state transition diagrams, rewrite rules, and others. We choose to specify the operational reading using an existing programming language – the lazy functional programming language Haskell – because it allows us to give compact and clear, yet executable, definitions to the four operators: see [11].

The Haskell evaluation of a function call in Clark's form of a logic program mimics SLD resolution of the corresponding goal in the original logic program. Even though it is evaluated in the functional setting of Haskell, the predicate *append* behaves like a relation, i.e., one can compute the answers to goals such as $append([1], y, [1, 2])$ or $append(x, y, [1, 2])$. 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 (see [12] and [13]).

The relationship between the abstract logical meaning of the four basic operators and their operational meaning that arises from SLD resolution is imperfect. In other words, in order for Clark's form of the logic program to have the same *computational* meaning as SLD resolution, the operators & and $||$ in the completed version of the logic program can not behave *exactly* like conjunction and disjunction in first order logic. Also, $\exists$ must introduce new variable names, rather than just existentially quantifying over a variable. The $\doteq$ needs to be understood as unification of terms. Therefore only some of the usual Boolean properties hold for these operators, and by recognising which properties of the operators hold both in first order logic and in an implementation which is correct with respect to SLD resolution, we can *distil* the algebraic semantics of a logic program.

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 need to permit in our transformational proofs also the use of some of the laws that are not valid in the procedural but only in the declarative semantics, for example, the commutativity of &.

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:

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

A better definition of reverse uses an accumulating parameter and runs in linear time:

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

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, where the recursive decomposition follows the structure of the data type. They also satisfy a set of laws that are crucial in functional program transformations, and we will rely on one of those laws in our derivation. The outline of the derivation is as follows:

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

We denote this derivation by $(*)$ and 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 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:

$$\begin{array}{l}
\mathit{foldRList}\ (p,e)\ (l,\mathit{res}) = \\
\qquad (l \doteq \mathit{nil}\ \&\ e \doteq \mathit{res}) \\
\quad \parallel (\exists x,\mathit{xs},r.\ l \doteq \mathit{cons}(x,\mathit{xs})\ \& \\
\qquad\qquad \mathit{foldRList}\ (p,e)\ (\mathit{xs},r)\ \&\ p(x,r,\mathit{res})),
\end{array}$$

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 things rolling'. For example, the function $\mathit{foldRList}\ (\mathit{add},0)$ 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 returns a substitution that maps *res* to 17.

In the first step of the derivation $(*)$, we use the following predicate *snoc*:

$$\mathit{snoc}(x,l,\mathit{res}) = \mathit{append}(l,\mathit{cons}(x,\mathit{nil}),\mathit{res}).$$

The pattern of recursion in the definition of *rev1* is the same as that captured by *foldRList*. Using a result that guarantees that recursive definitions have unique fixed points, we may conclude that *rev1* is equal to the instance of *foldRList* shown in the second line of our derivation $(*)$.

The next step in $(*)$ involves a transition from *foldRList* to another higher-order predicate, *foldLList*. This left-associative fold over lists could be defined as follows:

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

Roughly spaeaking, the function call $\mathit{foldLList}\ (\mathit{add}, 0)\ ([2, 7, 8], \mathit{res})$ would return the predicate $\mathit{add}(0, 2, r_1)\ \&\ \mathit{add}(r_1, 7, r_2)\ \&\ \mathit{add}(r_2, 8, r_3)\ \&\ (r_3 \doteq \mathit{res})$. Again, this predicate has the effect of setting *res* to 17, but this time the numbers are added from left to right.

The second step in $(*)$ is an instance of the duality law,

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

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 $f(x, e, \mathit{res})$ must equal $g(e, x, \mathit{res})$ for all x and *res*. The predicates f and g associate with each other iff the predicates $\exists t.\ (f(x, t, \mathit{res})\ \&\ g(y, z, t))$ and $\exists t.\ (g(t, z, \mathit{res})\ \&\ f(x, y, t))$ are equal. In functional notation this corresponds to $f(x, g(y, z)) = g(f(x, y), z)$. The proof of (1) requires the following auxilliary result:

$$\begin{aligned}
&\exists t.\ (f(x, t, \mathit{res})\ \&\ \mathit{foldLList}\ (g, y)\ (\mathit{xs}, t)) \\
&= \exists t.\ (f(x, y, t)\ \&\ \mathit{foldLList}\ (g, t)\ (\mathit{xs}, \mathit{res})).
\end{aligned}$$

This is proved by induction, using the associativity assumption about f and g. Then this equality, with y instantiated to e, is used together with the assumption about the equality of $f(x, e, \mathit{res})$ and $g(e, x, \mathit{res})$, in the induction proof for (1).

Returning to our derivation $(*)$, 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:

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

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

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

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

For the penultimate step in the our derivation $(*)$, we first prove that $\mathit{revapp}(l,\mathit{acc},\mathit{res})$ equals *foldLList* $(\mathit{flipapp},\mathit{acc})$ (l,res) by a simple induction proof. Then, instantiating the arbitrary term *acc* in *foldLList* to the term *nil*, we get exactly the *foldLList* $(\mathit{flipapp},\mathit{nil})$ $(\mathit{xs},\mathit{ys})$ from the third line of the proof, so we can rewrite this to a call to $\mathit{revapp}(\mathit{xs},\mathit{nil},\mathit{ys})$ in the fourth line. The final step follows directly from the definition of *rev2*.

5 Example 2: sort

Our second 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:

$$\begin{aligned}
&\mathit{naiveSort}(l1,l2) = \mathit{perm}(l1,l2)\ \&\ \mathit{isSorted}(l2)\\
&\mathit{isSorted}(l) =\\
&\qquad (l \doteq \mathit{nil})\\
&\qquad \|\ (\exists x.\ l \doteq \mathit{cons}(x,\mathit{nil}))\\
&\qquad \|\ (\exists x,y,l2.\ l \doteq \mathit{cons}(x,\mathit{cons}(y,l2))\ \&\\
&\qquad\qquad \mathit{le}(x,y)\ \&\ \mathit{isSorted}(\mathit{cons}(y,l2)))
\end{aligned}$$

where *perm* has the standard definition, using the auxiliary predicate *delete*:

$$\begin{aligned}
&\mathit{perm}(l1,l2) =\\
&\qquad (l1 \doteq \mathit{nil}\ \&\ l2 \doteq \mathit{nil})\\
&\qquad \|\ (\exists x,\mathit{xs},\mathit{zs}.\ l2 \doteq \mathit{cons}(x,\mathit{xs})\ \&\\
&\qquad\qquad \mathit{delete}(x,l1,\mathit{zs})\ \&\ \mathit{perm}(\mathit{zs},\mathit{xs}))\\
&\mathit{delete}(x,l1,l2) =\\
&\qquad (\exists \mathit{ys}.\ l1 \doteq \mathit{cons}(x,\mathit{ys})\ \&\ l2 \doteq \mathit{ys})\\
&\qquad \|\ (\exists y,\mathit{ys},\mathit{zs}.\ l1 \doteq \mathit{cons}(y,\mathit{ys})\ \&\ l2 \doteq \mathit{cons}(y,\mathit{zs})\ \&\\
&\qquad\qquad \mathit{delete}(x,\mathit{ys},\mathit{zs})).
\end{aligned}$$

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 as follows:

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

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:

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

In step 1, we simply unfold the definition of $naiveSort(l1, l2)$ and use the commutativity property of &. In the next step we argue that the predicate *perm* is an instance of $foldRList(add, nil)$, where the predicate *add* is as defined below. First, we use an auxiliary result stating that the relation *perm* is symmetric, i.e. that $perm(zs, xs)$ and $perm(xs, zs)$ are equivalent. Second, we define *add* to be the converse of *delete*, i.e. $delete(x, l1, zs) = add(x, zs, l1)$, and we can now rewrite *perm* as:

$$
\begin{aligned}
&perm(l1, l2) = \\
&\qquad (l1 \doteq nil \,\&\, l2 \doteq nil) \\
&\quad \| \ (\exists x, xs, zs.\ l2 \doteq cons(x, xs) \,\& \\
&\qquad\qquad add(x, zs, l1) \,\&\, perm(xs, zs)).
\end{aligned}
$$

Then, once again using the result about the symmetricity of perm, we swap $l1$ and $l2$ and obtain a recursive equation equivalent to the one defining $\mathit{foldRList}\ (\mathit{add}, \mathit{nil})\ (\mathit{l1}, \mathit{l2})$.

The third step is the major step in this derivation, and it is the one where the efficiency gain is achieved, i.e. the one that captures this transformation strategy. It involves the fusion law for $\mathit{foldRList}$, which can be proved by induction on the length of the input list. The assumptions for this law are as follows: let predicates f, g and h, and a term e, be such that $f(e)$ holds, and that $f(\mathit{res})\ \&\ g(x, y, \mathit{res})$ rewrites to the same recursive equation as $h(x, y, \mathit{res})\ \&\ f(y)$ for all terms x, y and res (in functional notation, $f(g\ x\ y) = h\ x\ (f\ y)$). Then, the fusion law states that:

$$f(\mathit{res})\ \&\ \mathit{foldRList}\ (g, e)\ (l, \mathit{res}) = \mathit{foldRList}\ (h, e)\ (l, \mathit{res}). \qquad (2)$$

If we now insert our predicate $\mathit{isSorted}$ for f, add for g, insert for h, and nil for e, the third step in the main proof is a straight-forward application of the fusion law. We only need to prove that our choices for f, g, h, and e satisfy the two fusion law requirements. The predicate call $\mathit{isSorted}(\mathit{nil})$ holds by definition, and the remaining condition for f, g and h is that:

$$\mathit{isSorted}(\mathit{res})\ \&\ \mathit{add}(x, l, \mathit{res}) = \mathit{insert}(x, l, \mathit{res})\ \&\ \mathit{isSorted}(l).$$

This equality can also be proved by an application of algebraic laws and induction on the lists l and res, using the lemma:

$$\begin{aligned}&\mathit{delete}(x, \mathit{zs}, \mathit{ys})\ \&\ \mathit{isSorted}(\mathit{cons}(y, \mathit{zs}))\\ &= \mathit{gt}(x, y)\ \&\ \mathit{delete}(x, \mathit{zs}, \mathit{ys})\ \&\ \mathit{isSorted}(\mathit{cons}(y, \mathit{zs}))\end{aligned}$$

which can also be proved by induction on the argument lists zs and ys.

In the final step, we simply recognise that $\mathit{isort}(\mathit{l1}, \mathit{l2})$ is equivalent to $\mathit{foldRList}\ (\mathit{insert}, \mathit{nil})\ (\mathit{l1}, \mathit{l2})$.

Following a similar approach, we can also derive the equivalence of the naive sort and, for example, $\mathit{quickSort}$ or $\mathit{selectionSort}$. 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 $\mathit{quickSort}$ uses fold and unfold predicates on trees. The reason for this is that even though $\mathit{quickSort}$ 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. Essentially, the derivation of $\mathit{quickSort}$ involves proving the equality:

$$\begin{aligned}&\mathit{isSorted}(\mathit{l2})\ \&\ \mathit{perm}(\mathit{l1}, \mathit{l2})\\ &= \mathit{mkTree}(\mathit{l1}, t)\ \&\ \mathit{flatten}(t, \mathit{l2})\end{aligned}$$

where the predicate $mkTree(l1, t)$ holds if t is an ordered tree:

$$\begin{aligned}
&mkTree(l, t) = \\
&\qquad (l \doteq nil \mathbin{\&} t \doteq null) \\
&\quad\ \| \ (\exists x, xs, t1, t2, l1, l2, a.\ l \doteq cons(x, xs) \mathbin{\&} t \doteq fork(t1, a, t2) \mathbin{\&} \\
&\qquad\qquad split(l, l1, a, l2) \mathbin{\&} mkTree(l1, t1) \mathbin{\&} mkTree(l2, t2)) \\
&split(l, l1, a, l2) = \\
&\qquad \exists y, ys.\ l \doteq cons(y, ys) \mathbin{\&} a \doteq y \mathbin{\&} \\
&\qquad\qquad filter(\lambda x.le(x, y))(l, l2) \mathbin{\&} filter((\lambda x.gt(x, y)(l, l1)
\end{aligned}$$

where *filter* (g) $(l, l1)$ is a higher order predicate that holds if $l1$ contains all the elements of l that satisfy g, and $flatten(t, l)$ holds if the list l corresponds to the flattened tree t. The terms representing trees and fold functions on trees are defined similarly to the corresponding definitions for lists.

6 Related and Further Work

An embedding of logic programs to a functional setting has been explored by Wand [15], 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 transformation 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 a functional setting includes, among others, Wand's work [14] 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 directions 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*, pages 116–149, Springer LNCS 1608, 1998.
6. R. Hinze. Prological features in a functional setting - axioms and implementations. In *Proc. of FLOPS'98*, pages 98–122, World Scientific, Japan, 1998.
7. A. Pettorossi and M. Proietti. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter Transformation of Logic Programs, pages 697–787. Oxford University Press, 1998.
8. A. Pettorossi and M. Proietti. Program derivation via list introduction. In *Proceedings of IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, pages 296–323. Chapman and Hall, Le bischenberg, France, 1997.
9. B.J. Ross. Using algebraic semantics for proving Prolog termination and transformation. In *Proceedings of the ALPUK 1991*, pages 135–155. Edinburgh, Springer, 1991.
10. S. Seres. *The Algebra of Logic Programming.* PhD thesis, Oxford University, 2001 (to appear).
11. S. Seres and J.M. Spivey. Functional Reading of Logic Programs. In *Journal of Universal Computer Science*, volume 6(4), pages 433–446, 1999.
12. S. Seres, J.M. Spivey, and C.A.R. Hoare, Algrebra of Logic Programming *Proceedings of ICLP'99*, pages 184–199, Las Cruces, USA, The MIT Press, 1999.
13. J.M. Spivey. The monad of breadth-first search. *Journal of Functional Programming*, volume 10(4), pages 397–408, 2000.
14. M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, volume 27(1), pages 164–180, 1980.
15. M. Wand. A semantic algebra for logic programming. Technical Report 148, Indiana University Computer Science Department, 1983.