Unfold/Fold Transformations and Loop Optimization of Logic Programs

Saumya K. Debray

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract: Programs typically spend much of their execution time in loops. This makes the generation of efficient code for loops essential for good performance. Loop optimization of logic programming languages is complicated by the fact that such languages lack the iterative constructs of traditional languages, and instead use recursion to express loops. In this paper, we examine the application of unfold/fold transformations to three kinds of loop optimization for logic programming languages: recursion removal, loop fusion and code motion out of loops. We describe simple unfold/fold transformation sequences for these optimizations that can be automated relatively easily. In the process, we show that the properties of unification and logical variables can sometimes be used to generalize, from traditional languages, the conditions under which these optimizations may be carried out. Our experience suggests that such source-level transformations may be used as an effective tool for the optimization of logic programs.

1. Introduction

The focus of this paper is on the static optimization of logic programs. Specifically, we investigate loop optimization of logic programs. Since programs typically spend most of their time in loops, the generation of efficient code for loops is essential for good performance. In the context of logic programming languages, the situation is complicated by the fact that iterative constructs, such as *for* or *while*, are unavailable. Loops are usually expressed using recursive procedures, and loop optimizations have be considered within the general framework of inter-procedural optimization.

There are various levels at which loop optimization may be carried out for such languages: at the source language level, at the intermediate language level, or at the final code level. An advantage with performing the optimization at the intermediate language or final code level is that certain operations, which are not expressible at the source level, become explicit and hence amenable to optimization. However, because of the significantly larger number of statements, variables, etc., that must be manipulated at the intermediate or final code level, the analyses and transformations are usually more complex than at the source level. For the purposes of this paper, therefore, we consider optimizing transformations applicable at the source level.

We consider the application of source-to-source transformations – specifically, unfold/fold transformations – to improve the code for loops in Prolog. One potential problem with unfold/fold transformations is that in general, ‘‘eureka’’ steps – steps involving a significant amount of insight into the behavior of the program being transformed or the algorithm it implements – may be necessary to actually carry the transformation through and achieve real improvements. This can make the transformations difficult to automate. Since we are interested primarily in compiler optimizations, we restrict our attention to cases where the transformations can be automated relatively simply. We show how such classical loop optimizations as recursion removal (i.e. transformation of certain recursive programs to tail recursive form), loop fusion, and code motion out of loops can be handled using our approach. For some of these, we are able to exploit the properties of logical variables and unification to generalize, from traditional languages, the conditions under which the optimizations can be carried out.

There is a great deal of literature on loop optimization for traditional languages, see [1]. Unfold/fold transformations were introduced by Burstall and Darlington for functional languages [6], and have been applied to the improvement of recursive programs in functional languages [6, 11]. Cohen considers the application of source-to-source transformations to the improvement of recursive programs [Cohen Improvement Recursive]. Arsac and Kodratoff have studied the application of these techniques to recursion removal from functions [2]. The application of unfold/fold transformations to recursion removal in logic programs has been considered by

Bloch [4], Debray [9], and more recently by Azibi et al. [3]. Tamaki and Sato [18], and Kanamori and Horiuchi [12], have investigated the theory of unfold/fold transformation systems for logic programs. Other applications of unfold/fold transformation systems to logic programming include improvement of tree manipulation programs [15], and of generate-and-test programs [16].

We assume some acquaintance with the fundamentals of logic programming and Prolog. The remainder of this summary is organized as follows: Section 2 sketches some basic notions used later in the paper. Section 3 considers recursion removal, Section 4 discusses loop fusion, and Section 5 discusses code motion out of loops. Section 6 gives an overview of related work, and Section 7 concludes with a summary.

2. Preliminaries

2.1. Logic Programming Languages

A program in a logic programming language consists of a set of predicate definitions, corresponding to procedure declarations in traditional languages. A predicate definition consists of a set of clauses, each clause being of the form

$$p(\bar{T}_0) :- q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n).$$

Operationally, this can be thought of as a definition for the procedure p, with formal parameters $\bar{T}_0$, whose body consists of a set of procedure calls $\{q_1(\bar{T}_1), \ldots, q_n(\bar{T}_n)\}$, and where parameter passing is done via a generalized pattern-matching procedure called *unification*. Declaratively, this may also be read as the logical statement ''[$q_1(\bar{T}_1)$ and ... and $q_n(\bar{T}_n)$] implies $p(\bar{T}_0)$''. Usually, logic programming languages strengthen this one-way implication to be bidirectional, i.e. a predicate defined as

$$p :- Body_1.$$
$$\ldots$$
$$p :- Body_m.$$

is interpreted declaratively as ''p iff [$Body_1$ or ... or $Body_m$]''. This strengthening is referred to as the ''completion'' of the predicate [8].

Logic programming languages lack iterative constructs for loops such as *for* and *while*. Instead, loops in such languages are expressed using tail recursive procedures (though not necessarily linear, i.e. they are not limited to one recursive call in the body). A loop can, in most cases, be written schematically using two clauses: a nonrecursive clause giving the termination conditions, and a recursive clause containing the body of the loop:

$loop(\bar{X})$:– $loop\_term(\bar{X})$.
$loop(\bar{X})$:– $loop\_body(\bar{X}, \bar{Y})$, $loop(\bar{Y})$.

We will follow Edinburgh Prolog syntax and write variable names starting with upper case letters, and non-variable names (i.e. functor and predicate names) starting with lower case letters. In addition, ‘‘anonymous’’ variables will be written as underscores. We will adopt the following notation for lists: the empty list will be written as ‘[]’, while a list with head H and tail L will be written ‘[H|L]’.

2.2. Unfold/Fold Transformations

Unfold/fold transformations, introduced by Burstall and Darlington in the context of functional languages [6], are based on a very simple idea: that of replacing equals by equals. Unfolding refers to the replacement of a procedure call by the appropriate instance of the procedure body, and is essentially an inline expansion of the call. For example, given the program

$p(X, Y)$:– $q(X, Z)$, $r(Z, Y)$.
$q(U, V)$:– $s(U, U1, W)$, $t(W, U2, V)$.

we can unfold the literal for *q* in the clause for *p*, to obtain

$p(X, Y)$:– $s(X, X1, W1)$, $t(W1, X2, Y)$, $r(Z, Y)$.
$q(U, V)$:– $s(U, U1, W)$, $t(W, U2, V)$.

In general, given the definitions

$p(\bar{X})$:– $Lits_1$, $q(\bar{T})$, $Lits_2$.
$q(\bar{Y})$:– $Body_1$.
. . .
$q(\bar{Y})$:– $Body_m$.

let θ be a substitution such that $\bar{T} = \theta(\bar{Y})$ (after renaming the variables in the clauses, if necessary, so that no two clauses have any variables in common). Then, we can unfold the literal for *q* in the clause for *p* to obtain

$p(\theta(\bar{X}))$:– $\theta(Lits_1$, ($Body_1$; . . . ; $Body_m$), $Lits_2)$.

If the predicates called from $Lits_1$ are free of metalanguage constructs such as *var* and *nonvar*, and free of side effects, then this unfolded clause can in fact be used to replace the original clause for *p* that was unfolded.

One point to note here is that of variable renaming when clauses are invoked. For technical reasons beyond the scope of this paper (but see [13]), when a procedure is activated for a call in a logic program, i.e., when a literal is unified with the head of a clause, the variables in the invoked clause are renamed so that it does not contain any variable used upto that point in the computation. Since unfolding is essentially an inline expansion of a literal, this sort of variable renaming is necessary during unfolding as well. In the remainder of the paper, we will assume that variables in clauses are renamed whenever necessary to satisfy this requirement, and not state it explicitly.

Folding refers to the replacement of an instance of a procedure body by a call to the procedure. For example, given the predicate definitions

$$p(X, Y) :- p_1(X, Z), r(g(Z)), s(h(Z, Y)), p_2(Y, Z).$$
$$q(U, V) :- r(U), s(V).$$

we can fold the literals for r and s in the clause defining p, to yield

$$p(X, Y) :- p_1(X, Z), q(g(Z), h(Z, Y)), p_2(Y, Z).$$

In general, given a clause

$$p(\bar{X}) :- Lits_1, (L_1 ; \ldots ; L_m), Lits_2.$$

and a predicate q defined as

$$q(\bar{Y}) :- Body_1 ; \ldots ; Body_m.$$

if there is a substitution θ such that, after renaming variables as necessary so that the two clauses do not share variables, we have $\theta(Body_i) = L_i$, $1 \le i \le m$, then the clause for p can be folded to yield

$$p(\bar{X}) :- Lits_1, q(\theta(\bar{Y})), Lits_2.$$

That unfold/fold transformations preserve partial correctness in functional languages follows from the fact that they replace equals by equals; however, total correctness may have to be proved separately. Things are a little more complicated in logic programming languages because the underlying theory is one of implication rather than equality. It can be shown that unfold transformations still preserve partial correctness,[1] but fold transformations are applicable only if we assume that the definition of a predicate in a program is an *if-and-only-if* definition rather than a one-way implication. This does not pose a problem in practice, since logic programming languages usually make this assumption. It turns out, however, that further restrictions are necessary if the transformed programs are to have the same least-model semantics as the original ones [18], i.e. if total correctness is to be guaranteed with respect to SLD-resolution.

[1] Nonlogical features, such as Prolog's *cut*, can be handled by preprocessing the program.

The restrictions are essentially that (i) the substitution θ used in the folding should substitute distinct variables for the internal variables of the literals $Body_1, \ldots, Body_m$, and that these internal variables should not occur in the head $p(\bar{X})$ or in the other literals in the body, $Lits_1$ and $Lits_2$; and (ii) a clause is not used to fold itself. Our experience has been that these constraints, while necessary, are not overly restrictive in practice.

3. Recursion Removal

In languages lacking explicit constructs for iteration, loops are usually expressed as tail recursive procedures. It is well known that tail recursion can be replaced by iteration (see [5, 19, Kieburtz Schultis Transformation]). There are many problems, however, that are strongly iterative in flavor, but whose natural specifications are not tail recursive. This is exemplified by the factorial function:

$fact(x) = \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * fact(x-1).$

In Prolog, this might be coded as

```
fact(0, 1).
fact(N, F) :– N > 0, N1 is N–1, fact(N1, F1), F is N * F1.
```

Recursion removal refers to the transformation of such definitions to tail recursive form using operator properties such as associativity [2]. Once tail recursion has been achieved, iteration may be obtained in a relatively straightforward manner. This optimization can result in substantial savings in space and time. The problem has been studied in the case of functional languages by several researchers [Kieburtz Schultis Transformation, 6, 17]. In the case of logic programming languages, the situation is somewhat different in that there is no notion of function application and composition: rather, relations are computed using unification. Moreover, these relations may be computed in any order, in principle, and computations may be nondeterministic, making it difficult to predict statically the structures of expressions that might have to be evaluated.

3.1. An Example

We first illustrate our approach by a simple example. Consider a predicate to compute the length of a list:

```
(1)     len([ ], 0).
(2)     len([H|L], N) :– len(L, N1), N is N1 + 1.
```

While the computation essentially involves traversing a list and incrementing a counter at each step, the natural specification of the problem above is not tail recursive, and hence, for a list of length N, requires $O(N)$ stack space. We apply unfold/fold transformations, together with knowledge about the associativity of the operator ‘+’, to transform this specification to tail recursive form. The first step involves generalization. Here, an auxiliary predicate is generated, whose purpose is to make the recursive call and the *tail computation* ‘‘N is N1 + 1’’. Our aim will be to optimize this auxiliary predicate. The clauses resulting from the auxiliary predicate definition are

(3) len([ ],0).
(4) len([H|L], N) :– len_1(L, N, 1).

(5) len_1(L, N, M) :– len(L, N1), N is N1 + M.

The next step is to remove mutual recursion between these predicates by unfolding the call to *len* in clause (5):

(6) len_1([ ], N, M) :– N is 0 + M.
(7) len_1([H|L], N, M) :– len_1(L, N1, 1), N is N1 + M.

This (with the obvious simplification of (6)) is the *basic definition* of the auxiliary predicate. While this looks similar to the original definition, the crucial difference is that an extra argument (the generalization argument) is now available. The remainder of the transformation focuses on *collapsing* the basic definition to obtain tail recursion. Collapsing consists of an unfolding, followed by a simplification step, and finally a folding step. The first step is to unfold (7) using the initial definition (5), yielding

(8) len_1([H|L], N, M) :–
    len(L, N2), N1 is N2 + 1, N is N1 + M.

Using properties of the arithmetic predicate *is* and the associativity of +, (8) is transformed to

(9) len_1([H|L], N, M) :– len(L, N2), N is N2 + (1 + M).

The computation of the subexpression 1+M is independent of the call to *len*, and can be pulled forward to give

(10) len_1([H|L], N, M) :–
    K is 1 + M, len(L, N2), N is N is N2 + K.

The final step is to fold (10) using (5) (which had been used in the most recent unfolding step). This yields the clause

(11) len_1([H|L], N, M) :– K is 1 + M, len_1(L, N, K).

which is tail recursive. The final tail recursive definition, equivalent to the original program, is therefore

len([ ],0).
len([H|L], N) :– len_1(L, N, 1).

len_1([ ], M, M).
len_1([H|L], N, M) :– K is 1 + M, len_1(L, N, K).

3.2. The Transformation Strategy and its Applicability

We refer to the class of predicates we are interested in as *almost-tail-recursive*. Define a clause to be almost-tail-recursive if it the goals following the last recursive literal in the body involve only primitive computations. A predicate is almost-tail-recursive if every recursive clause for it is either tail-recursive or almost-tail-recursive, and there is at least one almost-tail-recursive clause. An almost-tail-recursive clause will be written as

$$p :- q_1, \ldots, q_n, p, eval(T_1 \circ T_2, X). \qquad n \geq 0.$$

where *eval* is a pseudo-evaluable-predicate, denoting that the result of evaluating $T_1 \circ T_2$ is unified with X. The ''goal'' $eval(T_1 \circ T_2, X)$ will be referred to as the tail computation.

The transformation strategy follows the example above quite closely. The procedure for carrying out the various steps of the transformation is described in [9]. The steps involved are the following:

(1) Define the auxiliary predicate. This is done as follows: consider an almost-tail-recursive predicate with tail computations $\{eval(T_{11} \circ T_{12}, X_1), \ldots, eval(T_{n1} \circ T_{n2}, X_n)\}$. Corresponding to each tail computation $eval(T_{j1} \circ T_{j2}, X_j)$, let $\bar{C}_j$ = $\$langlec_{j1}, \ldots, c_{jn_j} \$rangle$ be a tuple of all maximal ground subtrees of T_{j1} and T_{j2}. Let $N = \max_j \{length(\bar{C}_j)\}$. Then, if the arity of the original predicate is k, then that of the auxiliary predicate is $k + N$.

Consider the j^{th} tail computation, $eval(T_{j1} \circ T_{j2}, X_j)$, with $\bar{C}_j$ being $\$langlec_{j1}, \ldots, c_{jn_j} \$rangle$. Corresponding to each c_{jk}, let v_k be a new variable, and let $\bar{V}$ be the tuple $\$langlev_1, \ldots, v_{jn_j} \$rangle$. Let T^*_{j1}, T^*_{j2} be the expressions obtained by replacing each c_{jk} by v_k in T_{j1} and T_{j2}.

Replace each almost-tail-recursive clause by a pair of clauses, one for the original predicate and one for the auxiliary predicate, defined as follows: if the original clause is

$$p(\bar{X}_0) :- q(\bar{X}_1), p(\bar{X}_2), eval(T_1 \circ T_2, X).$$

and the auxiliary predicate is p_1, then the corresponding pair of clauses is

$$p(\bar{X}_0) :- q(\bar{X}_1), p_1(\bar{Z}, \bar{C}).$$
$$p_1(\bar{Z}, \bar{C}) :- p(\bar{X}_2), eval(T^*_1 \circ T^*_2, X).$$

If $\bar{Y}$ is the set of variables $\bar{X}_2$ together with the variables appearing in the tail computation *eval*$(T_1 \circ T_2, X)$, then $\bar{Z} = (\bar{X}_0 \cup \bar{X}_1) \cap \bar{Y}$ corresponds to the variables in the almost-tail-recursive call and tail computation that appear elsewhere before it in the original clause, and $\bar{C}$, $\bar{V}$, T_1^* and T_2^* are as defined above.

(2) Unfold the auxiliary predicate to remove mutual recursion with the original predicate. This yields the *basic definition* of the auxiliary predicate. Simplify the nonrecursive clauses of the basic definition using algebraic identities where possible.

(3) Use *collapsing* to transform the auxiliary predicate to tail recursive form. This involves an unfolding step, followed by some arithmetic manipulation using operator properties such as associativity and distributivity, and finally a folding step that achieves tail recursion.

The example above, illustrating the transformation, was extremely simple in that the predicate involved had only one recursive clause, was deterministic, and used only one kind of arithmetic operator. Our transformation is also applicable, however, to predicates that involve multiple recursive clauses, use more than one kind of operator (provided that the operators satisfy certain simple algebraic properties), or are nondeterministic. For example, our transformation can deal with the nondeterministic computation described by

$$f(x) = \begin{cases} 1 \text{ if } x = 0 \\ 2*f(x-1) \text{ if } x\ (>0) \textit{ is even} \\ 3*f(x-1)-1 \textit{ or } 2*f(x-1)+1 \text{ if } x\ (>0) \textit{ is odd} \end{cases}$$

Given an almost-tail-recursive clause

$p(\bar{X}_0)$:– $q_1, \ldots, q_n$, $p(\bar{X}_1)$, *eval*$(T_1 \circ T_2, X)$

the tail computation *eval*$(T_1 \circ T_2, X)$ will be referred to as *left-independent* (respectively, right-independent) if T_1 (respectively, T_2) is independent of the recursive literal $p(\bar{X}_1)$. A simple case of independence that is encountered relatively often is when either T_1 or T_2 is a ground term, as in the *len* example above. Independence may be inferred via static analysis methods described in [7, Debray Static Inference 1987]. We have the following sufficient conditions for this transformation to be applicable:

Proposition 3.1: The transformation is applicable if, for every tail computation *eval*$(T_1 \circ T_2, X)$ in the definition, $\circ$ is associative, and either every tail computation is left-independent, or every tail computation is right-independent. *$box*

If the operator $\circ$ is also commutative, the requirement can be relaxed somewhat:

Proposition 3.2: The transformation is applicable if, for every tail computation $eval(T_1 \circ T_2, X)$ in the definition, $\circ$ is associative and commutative, and every tail computation is either left-independent or right-independent. *$box*

If more than one kind of operator is involved, they must satisfy certain distributivity requirements:

Proposition 3.3: Let F be the set of operators appearing in the tail computations of an almost-tail-recursive predicate. A sufficient condition for the transformation to be applicable is that (*i*) each tail computation contains all the operators in F (or can be modified to incorporate them with the appropriate identity elements, which must exist); (*ii*) each operator in F is associative; (*iii*) F can be totally ordered so that given f_1, f_2 in F, $f_1 \leq f_2$ if and only if f_1 is right (left) distributive over f_2; and (*iv*) each such operator in each tail computation operates over a left (right) independent expression. *$box*

The modification of this proposition to incorporate operator commutativity, to relax the requirements given, is straightforward.

The benefits accruing from recursion removal include space savings (since the tail recursive predicates can usually execute in $O(1)$ space, compared to the $O(N)$ space requirements of the recursive definitions[2]), as well as improvements in speed resulting from the elimination of the recursive calls. The reduced space usage also improves locality of reference on the runtime stack. Preliminary experiments indicate that apart from the space savings, program speeds can improve from 12% to over 45%, even for code written by experienced Prolog programmers.

4. Loop Fusion

Loop fusion, also known as *loop jamming*, refers to the merging of the bodies of two loops. Amongst the benefits of loop fusion are (*i*) repeated traversals of data structures may be avoided; (*ii*) the tests of one loop disappear; and (*iii*) the construction of intermediate data structures may be avoided.

---

[2] Nondeterministic almost-tail-recursive predicates still require $O(N)$ space after transformation to tail recursive form, but the space requirements of the transformed program are smaller than those of the original almost-tail-recursive one.

In traditional languages, two loops may be fused only if both have the same number of iterations. A sufficient condition for loop fusion to be legal is that no quantity is computed by the second loop at iteration i if it is computed by the first loop at iteration $j \geq i$, and no value is used by the second loop at iteration i if it is computed by the first loop at iteration $j \geq i$. In our approach, the properties of unification and logical variables can be exploited to weaken these conditions for logic programming languages. For example, loop fusion can sometimes be effected even the two loops do not have the same number of iterations; because of unification and the once-only nature of ‘‘assignment’’ in logic programming languages, it is not necessary to require that no quantity be computed by the second loop at iteration i if it is computed by the first loop at iteration $j \geq i$; and logical variables can sometimes be used as placeholders to effect loop fusion even when a value is computed by the first loop at iteration j but "used" by the second loop at iteration $i \leq j$. On the other hand, because loops are typically written as recursive procedures, the detection of situations where two loops are candidates for fusion is somewhat more complicated than in traditional languages..

4.1. An Example

We first illustrate our basic approach to loop fusion via fold/unfold transformations using a simple example. Consider a program that computes the combined length of two lists:

```
(1)     len2(L1, L2, N) :–
            append(L1, L2, L3), length(L3, N).

(2)     append([ ], L, L).
(3)     append([H|L1], L2, [H|L3]) :– append(L1, L2, L3).

(4)     length([ ], 0).
(5)     length([H|L], N) :– length(L, N1), N is N1 + 1.
```

The predicate *len2* consists of two loops, each of which traverses a list once. The transformation begins by unfolding the call to *append* in (1):

```
(6)     len2([ ], L2, N) :– length(L2, N).
(7)     len2([H|L], L2, N) :–
            append(L, L2, L3), length([H|L3], N).
```

The second step is to unfold the call to *length* in (7):

```
(8)     len2([H|L1], L2, N) :–
            append(L1, L2, L3), length(L3, N1), N is N1 + 1.
```

Finally, (8) is folded using the original definition of *len2*, given by clause (1), to yield

```
(9)     len2([H|L1], L2, N) :–
```

len2(L1, L2, N1), N is N1 + 1.

The final definition of *len2*, therefore, is:

len2([ ], L2, N) :– length(L2, N).
len2([H|L1], L2, N) :– len2(L1, L2, N1), N is N1 + 1.

Since the transformed program avoids traversing the first input list twice, and also does not construct an intermediate list by concatenating the two input lists, it is both faster (by about 12%) and more space efficient than the original program. The reader can see also that this could be further optimized using the recursion removal techniques discussed in the previous section.

4.2. The Transformation Strategy and its Applicability

The transformation follows the example above. For simplicity of exposition, we assume that there is a predicate whose sole function is to execute the loops we intend to merge, i.e. which is defined by a single clause whose body consists simply of calls to the respective loops. We refer to this predicate as the *loop driver*. Assume we begin with definitions of the form

(1) loop_driver($\bar{X}$) :– $\text{loop}_1(\bar{Y})$, $\text{loop}_2(\bar{Z})$.

(2) $\text{loop}_1(\bar{X})$:– $\text{loop\_term}_1(\bar{X})$.

(3) $\text{loop}_1(\bar{X})$:– $\text{loop\_body}_1(\bar{X}, \bar{Y})$, $\text{loop}_1(\bar{Y})$.

(4) $\text{loop}_2(\bar{X})$:– $\text{loop\_term}_2(\bar{X})$.

(5) $\text{loop}_2(\bar{X})$:– $\text{loop\_body}_2(\bar{X}, \bar{Y})$, $\text{loop}_2(\bar{Y})$.

In clause (1), it is assumed that $\bar{X}$, $\bar{Y}$ and $\bar{Z}$ may overlap. The steps involved in the transformation are the following:

(1) Unfold the calls to the recursive predicates in the clause for the loop driver. This yields the clauses

(6) loop_driver($\bar{X}$) :– $\text{loop\_term}_1(\bar{Y})$, $\text{loop\_term}_2(\bar{Z})$.

(7) loop_driver($\bar{X}$) :–
$\text{loop\_body}_1(\bar{Y}, \bar{U})$, $\text{loop}_1(\bar{U})$, $\text{loop\_term}_2(\bar{Z})$.

(8) loop_driver($\bar{X}$) :–
$\text{loop\_term}_1(\bar{Y})$, $\text{loop\_body}_2(\bar{Z}, \bar{V})$, $\text{loop}_2(\bar{V})$.

(9) loop_driver($\bar{X}$) :–
$\text{loop\_body}_1(\bar{Y}, \bar{U})$, $\text{loop}_1(\bar{U})$,
$\text{loop\_body}_2(\bar{Z}, \bar{V})$, $\text{loop}_2(\bar{V})$.

(2) Rearrange independent computations in the resulting recursive clause (9), such that the calls to the loops being folded are adjacent. The clause resulting from this is:

(10) loop_driver($\bar{X}$) :–
loop_body$_1$($\bar{Y}$, $\bar{U}$), loop_body$_2$($\bar{Z}$, $\bar{V}$),
loop$_1$($\bar{U}$), loop$_2$($\bar{V}$).

(3) Fold the resulting clause using the original definition of the loop driver. The resulting clause is:

(11) loop_driver($\bar{X}$) :–
loop_body$_1$($\bar{Y}$, $\bar{U}$), loop_body$_2$($\bar{Z}$, $\bar{V}$),
loop_driver($\bar{W}$).

The transformed definition, where the loops have been fused, consists of clauses (6), (7), (8) and (11).

We next consider conditions under which the transformation is applicable. Our basic aim is to first try and transform the recursive clause (9), resulting from the unfolding, to the form in (10), where the literals for $loop_1$ and $loop_2$ are adjacent. Once this has been achieved, the folding step can be carried out. For the first step to be possible, it may be necessary to rearrange the literals in the clause, from

loop_driver($\bar{X}$) :–
loop_body$_1$($\bar{Y}$, $\bar{U}$), loop$_1$($\bar{U}$),
loop_body$_2$($\bar{Z}$, $\bar{V}$), loop$_2$($\bar{V}$).

to

loop_driver($\bar{X}$) :–
loop_body$_1$($\bar{Y}$, $\bar{U}$), loop_body$_2$($\bar{Z}$, $\bar{V}$),
loop$_1$($\bar{U}$), loop$_2$($\bar{V}$).

This requires that "loop$_1$($\bar{U}$)" and "loop_body$_2$($\bar{Z}$, $\bar{V}$)" be independent, i.e. not share variables at runtime (data dependence analysis may be used to determine this, see [7, Debray Static Inference 1987]). In general, however, the rearrangement of literals in a clause can affect computational aspects of programs such as termination, the order in which solutions are generated, etc. We must be able to guarantee that the rearrangement in this case will not adversely affect the computational behavior of the program. While this is difficult to determine in general, it is relatively straightforward in some cases, e.g. where, in addition to the independence criterion stated, (i) the loops are deterministic or functional (see [10, 14]), so that solution order is not an issue;

(ii) the termination of loop_body$_2$ is guaranteed (e.g. when it consists only of simple tests, or involves no recursion); and (iii) $loop_1$ is free of side-effects. (Notice that $loop_1$ may be nonterminating, but in this case delaying its execution by reordering literals cannot adversely affect the termination behavior of the program.)

Once this rearrangement of literals has been carried out, it is necessary to fold the resulting clause,

(A) loop_driver($\bar{X}$) :–
loop_body$_1$($\bar{Y}$, $\bar{U}$), loop_body$_2$($\bar{Z}$, $\bar{V}$),
loop$_1$($\bar{U}$), loop$_2$($\bar{V}$).

using the original definition of *loop_driver*, given by

(B) loop_driver($\bar{R}$) :– loop$_1$($\bar{S}$), loop$_2$($\bar{T}$).

For this to be possible, it is necessary that for some substitution θ,

(i) the literals "$loop_1(\bar{U})$, $loop_2(\bar{V})$" in (A) be an instance of the literals "$loop_1(\bar{S})$, $loop_2(\bar{T})$" in (B) via the substitution θ, i.e. $\bar{U} = \theta(\bar{S})$ and $\bar{V} = \theta(\bar{T})$; and

(ii) θ substitutes distinct variables for the internal variables of (B), i.e. the set of variables

$$\mathbf{V} = (vars(\bar{S}) \cup vars(\bar{T})) - vars(\bar{R})$$

and further, for no variable v in $\mathbf{V}$ is $\theta(v)$ a variable occurring in $\{\bar{U}, \bar{V}, \bar{X}\}$ in clause (A).

If these conditions are satisfied, the transformation can be carried through and loop fusion accomplished (condition (ii) is essentially that due to Tamaki and Sato for preserving equivalence, and referred to in Section 2.2).

In some cases these conditions can be weakened further, using the properties of unification and logical variables. This enables us to generalize the loop fusion conditions for traditional languages, by using logical variables as placeholders in some situations where a value is computed by the first loop at iteration j but "used" by the second loop at iteration $i \le j$. This is illustrated by the following example. Consider the following program, where the predicate *llast* is defined so that $llast(L_1, L_2)$ is true if and only if L_1 and L_2 are lists of the same length, with each element of L_2 equal to the last element of L_1:

```
llast([],[]).
llast([E|L1], L2) :– lastelt(L1, E, M), mklist([E|L1], M, L2).

lastelt([],X,X).
lastelt([E1|L], _ , X) :– lastelt(L, E1, X).
```

```
mklist([], _, []).
mklist([_|L1], E, [E|L2]) :– mklist(L1, E, L2).
```

Given a query "llast([1,4,2], X)", it can be seen that *lastelt* produces the value of the last element of the input list on its third iteration, but this value is used by *mklist* on its first iteration. Ordinarily, therefore, these two loops would not be fusible. Indeed, if the transformation steps are applied, it can be seen that the last folding step fails, and the transformation cannot be carried through. However, we can exploit logical variables to make loop fusion possible, by incorporating an extra generalization step. The idea is to introduce an additional logical variable whose value will be that of the last element of the list. This is used as a placeholder while building the output list, whose elements are not known until the last element of the input list is encountered in the iteration. At this point, all the elements of the output list are filled in at the same time via unification. Details of the transformation are as follows: first, we redefine *llast* in terms of an auxiliary predicate *llast_1*, incorporating a generalization variable:

```
llast(L1, L2) :– llast_1(L1, L2, _).

llast_1([], [], _).
llast_1([E|L1], L2, M) :–
        lastelt(L1, E, M), mklist([E|L1], M, L2).
```

From this point the transformation proceeds as before. After unfolding the literals for *lastelt* and *mklist* in *llast_1*, we get its definition as

```
llast_1([], [], _).
llast_1([E], [E|L], E) :– mklist([], E, L).
llast_1([E1, E2|L1], [E|L2], E) :–
        lastelt(L1, E2, E), mklist([E2|L1], E, L2).
```

The final folding step of the transformation, applied to the last clause, then yields the desired definition:

```
llast(L1, L2) :– llast_1(L1, L2, _).

llast_1([], [], _).
llast_1([E], [E|L], E) :– mklist([], E, L).
llast_1([E1, E2|L1], [E|L2], E) :– llast_1([E2|L1], L2, E).
```

In general, the requirement that the literals "$loop_1(\bar{U})$" and "$loop\_body_2(\bar{Z}, \bar{V})$" be independent can be relaxed, if $loop\_body_2$ consists entirely of unifications, as long as loop_1 is free of metalanguage constructs such as Prolog's *var*, *nonvar* and '==' predicates. The intuition here is that if $loop_1$ is free of side effects (as required earlier) and also free of metalanguage constructs such as these, then it consists of pure code, and in this case moving some unifications forward will not affect the semantics of the program. Moreover, as illustrated by the *llast* example above, the requirement that the literals

$$\text{loop}_1(\overline{U}),\ \text{loop}_2(\overline{V})$$

be an instance of the body of the original definition of *loop_driver* can also be weakened similarly in some cases by using an additional generalization step. Together, these show that the requirement that the loop fusion conditions for traditional languages can sometimes be generalized using logical variables and unification.

The transformation described here is not limited to linear tail recursive predicates. For example, the predicate *tmaxmin* (defined below), which computes the maximum and minimum leaf values in a nonempty tree, can be transformed from

```
tmaxmin(T, Max, Min) :– tmax(T, Max), tmin(T, Min).

tmax(leaf(X), X).
tmax(tree(X,Y), Z) :–
        tmax(X, Z1), tmax(Y, Z2), max(Z1, Z2, Z).

tmin(leaf(X), X).
tmin(tree(X,Y), Z) :–
        tmin(X, Z1), tmin(Y, Z2), min(Z1, Z2, Z).
```

to the following, which avoids repeated traversals of the same data structure:

```
tmaxmin(leaf(X), X, X).
tmaxmin(tree(X,Y), Max, Min) :–
    tmaxmin(X, Max1, Min1), tmaxmin(Y, Max2, Min2),
    max(Max1, Max2, Max), min(Min1, Min2, Min).
```

Notice that the associativity of *max* and *min* can be used to transform this to a nonlinear tail recursive definition using the recursion removal techniques discussed earlier, resulting in a more space-efficient program.

5. Code Motion out of Loops

There may be computations in a loop whose results are independent of the number of times the loop is executed. Since these computations are loop invariant, there is no need to perform them repeatedly. Instead, they may be moved to a point just before the start of the loop. This is referred to as code motion out of loops.

As mentioned in Section 2.2, it is necessary to rename variables in clauses when carrying out unfold/fold transformations. Sometimes, however, it is also necessary to keep track of the identities of variables across such renamings. To this end, we have the following proposition:

Proposition 5.1: Consider a clause

$p(\bar{X})$:– $Lits_1$, $X = T_1$, $Lits_2$, $X = T_2$, $Lits_3$.

where T_1 and T_2 are alphabetic variants. Then, the clause is equivalent to

$p(\bar{X})$:– $Lits_1$, $X = T_1$, $Lits_2$, $X = T_1$, $\theta(Lits_3)$.

where θ is a substitution that renames T_2 to T_1. *$box*

It is, in fact, possible to strengthen this proposition significantly, e.g., by eliminating the requirement that T_1 and T_2 be alphabetic variants. In this case, θ becomes the most general unifier of T_1 and T_2, and the clause we obtain is

$p(\bar{X})$:– $Lits_1$, $X = \theta(T_1)$, $Lits_2$, $X = \theta(T_2)$, $Lits_3$.

In this case, however, it is more difficult to argue that the two clauses are necessarily equivalent operationally, since the unification of X with $\theta(T_1)$ instead of T_1 may change the behavior of the program if $Lits_2$ has side effects or contains metalanguage constructs. It can also affect the termination behavior of $Lits_2$. For our purposes, the weak version given in Proposition 5.1 is adequate, so we choose to avoid the complications of the more general statement.

In logic programs, loop invariant computations can often be detected simply by using unfold/fold transformations at the source level. This is illustrated by the following example. Consider the program

```
process_vars([ ], _).
process_vars([X|L], Table) :–
    Table = table(VarTab, CTab),
    proc_var(X, VarTab), process_vars(L, Table).
```

The first step in detecting loop invariant computations is to unfold the tail recursive literal. This gives the clause

```
process_vars([ ], _).
process_vars([X], Table) :–
    Table = table(VarTab, CTab), proc_var(X, VarTab).
process_vars([X|L], Table) :–
    Table = table(VarTab, CTab), proc_var(X, VarTab),
    L = [Y|L1], Table = table(VarTab1, CTab1),
```

```
        proc_var(Y, VarTab1), process_vars(L1, Table).
```

Notice that the unfolding step, after the application of Proposition 5.1, results in the duplication of the literal "*Table* = *table*(*VarTab*, *CTab*)" in the recursive clause. It follows, from a straightforward inductive argument, that this computation is invariant in the loop and hence may be moved out of the body of the loop. The resulting code is

```
    process_vars(L, Table) :–
        Table = table(VarTab, CTab), process_vars1(L, VarTab).

    process_vars1([ ], _).
    process_vars1([X|L], VarTab) :–
        proc_var(X, VarTab), process_vars1(L, VarTab).
```

The transformation strategy follows this example. Consider a loop written as

$\text{loop}(\bar{X})$:– $\text{loop\_term}(\bar{X})$.
$\text{loop}(\bar{X})$:– $\text{proc}_1(\bar{U})$, $\text{q}(\bar{Y})$, $\text{proc}_2(\bar{V})$, $\text{loop}(\bar{Z})$.

The first step is to unfold the tail recursive clause and examine the recursive clause that results for duplicate literals introduced as a result of unfolding. Suppose the clause resulting from unfolding and applying Proposition 5.1 is

$\text{loop}(\bar{X})$:–
$\quad\quad \text{proc}_1(\bar{U})$, $\text{q}(\bar{Y})$, $\text{proc}_2(\bar{V})$,
$\quad\quad \text{proc}_1(\bar{U}')$, $\text{q}(\bar{Y})$, $\text{proc}_2(\bar{V}')$, $\text{loop}(\bar{Z}')$.

The literal "$\text{q}(\bar{Y})$" is duplicated by the unfolding step, and hence is a candidate for motion out of the loop. Before it can actually be moved, however, we have to ensure that the reordering of literals this entails will not adversely affect the computational behavior of the program. The conditions under which the reordering can be carried out are similar to those for loop fusion: (*i*) $proc_1$ and q are deterministic or functional, so that solution order is not an issue; (*ii*) "$proc_1(\bar{U})$" and "$q(\bar{Y})$" are independent; (*iii*) q is guaranteed to terminate (e.g. when it consists of simple tests or arithmetic computations); and (*iv*) $proc_1$ and q are free of metalanguage predicates and side effects.

Once the loop-invariant computation has been identified and the conditions above have been checked, the transformation proceeds as follows: first, literals in the original definition are reordered so that the loop-invariant computation is at the head of the clause:

$\text{loop}(\bar{X})$:– $\text{loop\_term}(\bar{X})$.
$\text{loop}(\bar{X})$:– $\text{q}(\bar{Y})$, $\text{proc}_1(\bar{U})$, $\text{proc}_2(\bar{V})$, $\text{loop}(\bar{Z})$.

An auxiliary predicate loop_1 is now introduced, whose definition resembles that of the original predicate, except that the loop-invariant computation has been deleted. The definition of the loop is modified to refer to this auxiliary predicate:

$\text{loop}(\bar{X})$:– $\text{loop\_term}(\bar{X})$.
$\text{loop}(\bar{X})$:– $\text{q}(\bar{Y})$, $\text{proc}_1(\bar{U})$, $\text{proc}_2(\bar{V})$, $\text{loop\_1}(\bar{Z})$.

$\text{loop\_1}(\bar{X})$:– $\text{loop\_term}(\bar{X})$.
$\text{loop\_1}(\bar{X})$:– $\text{proc}_1(\bar{U})$, $\text{proc}_2(\bar{V})$, $\text{loop\_1}(\bar{Z})$.

In order for this program to be equivalent to the original one, it is also necessary that *q* be free of side effects. This is easy to guarantee if *q* consists of simple tests and arithmetic computations.

In effect, what this does is to move the loop-invariant computation to the preheader for the loop. Traditionally, the plausibility of code motion out of loops has rested on the assumption that the loop will be executed at least once on the average. We do not have to make this assumption here (though we expect that loops will typically be executed at least once): notice that the code for loop termination is repeated, once in the predicate *loop* and once in *loop_1*. What this achieves is that the loop-invariant computation $\text{q}(\bar{Y})$ is performed only if the loop body is executed at least once; and further, it is performed at most once for any call to that loop. In this sense, therefore, the code achieved above is optimal. Of course, not all loop invariant computations may be detected using this scheme.

6. Conclusions

Since programs typically spend much of their time in loops, the generation of efficient code for loops is essential for good performance. Loop optimization in logic programming languages is complicated by the fact that such languages lack iterative constructs, such as *for* and *while*, that are available in traditional languages. Instead, logic programming languages express loops using recursion. In this paper, we examine the application of unfold/fold transformations to the optimization of loops in logic programming languages. Specifically, we examine three loop optimizations: recursion removal, loop fusion, and code motion out of loops. We give simple unfold/fold transformation sequences to carry out the loop optimizations in these cases, and describe the conditions under which the transformations are applicable. In the process, we show how the properties of unification and logical variables can be used to generalize the conditions of applicability of these optimizations from traditional languages.

References

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers - Principles, Techniques and Tools*, Addison-Wesley, 1986.

2. J. Arsac and Y. Kodratoff, Some Techniques for Recursion Removal from Recursive Functions, *ACM Trans. Prog. Lang. and Systems 4*, 2 (Apr. 1982), 295-322.

3. N. Azibi, E. J. Costa and Y. Kodratoff, Methode de Transformation de Programmes de Burstall-Darlington Appliquee a la Programmation Logique, Research Report No. 268, Universite de Paris-Sud, Orsay, France, Mar. 1986.

4. C. Bloch, Source-to-Source Transformations of Logic Programs, CS84-22, Dept. of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, Dec. 1984.

5. M. Bruynooghe, The Memory Management of PROLOG Implementations, in *Logic Programming*, K. L. Clark and S. Tarnlund (ed.), Academic Press, London, 1982. A.P.I.C. Studies in Data Processing No. 16.

6. R. M. Burstall and J. Darlington, A Transformation System for Developing Recursive Programs, *J. ACM 24*, 1 (January 1977), 44-67.

7. J. Chang, A. M. Despain and D. DeGroot, AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis, in *Digest of Papers, Compcon 85*, IEEE Computer Society, Feb. 1985.

8. K. L. Clark, Negation as Failure, in *Logic and Data Bases*, H. Gallaire and J. Minker (ed.), Plenum Press, New York, 1978.

9. S. K. Debray, Optimizing Almost-Tail-Recursive Prolog Programs, in *Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, Sept. 1985.

10. S. K. Debray and D. S. Warren, Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), pp. 451-481.

11. M. S. Feather, A System for Assisting Program Transformation, *ACM Trans. Prog. Lang. and Systems 4*, 1 (Jan. 1982), 1-20.

12. T. Kanamori and K. Horiuchi, Construction of Logic Programs Based on Generalized Unfold/Fold Rules, in *Proc. Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 744-768.

13. J. W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, 1984.

14. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.

15. H. Nakagawa, Prolog Program Transformations and Tree Manipulation Algorithms, *J. Logic Programming 2*, 2 (July 1985), 77-91.

16. H. Seki and K. Furukawa, Notes on Transformation Techniques for Generate and Test Logic Programs, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987, pp. 215-223.

17. E. St.-James, Recursion is More Efficient than Iteration, in *Proc. 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, Aug. 1984.

18. H. Tamaki and T. Sato, Unfold/Fold Transformations of Logic Programs, in *Proc. 2nd. Logic Programming Conference*, Uppsala, Sweden, 1984.

19. D. H. D. Warren, An improved Prolog implementation which optimises tail recursion, Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, 1980. Presented at the 1980 Logic Programming Workshop, Debrecen, Hungary.