Fundamenta Informaticae XXI (2001) 1001–1031
IOS Press

A Multiple-Clause Folding Rule Using Instantiation and Generalization

David A. Rosenblueth
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. 20-726, 01000 México D. F.
México
drosenbl@servidor.unam.mx

Abstract. A program-transformation system is determined by a repertoire of correctness-preserving rules, such as folding and unfolding. Normally, we would like the folding rule to be in some sense the inverse of the unfolding rule. Typically, however, the folding rule of logic program transformation systems is an inverse of a limited kind of unfolding. In many cases this limited kind of folding suffices. We argue, nevertheless, that it is both important and possible to extend such a folding so as to be able to fold the clauses resulting from any unfolding of a positive literal. This extended folding rule allows us to derive some programs underivable by the existing version of this rule alone. In addition, our folding rule has applications to decompilation and reengineering, where we are interested in obtaining high-level program constructs from low-level program constructs. Moreover, we establish a connection between logic program transformation and inductive logic programming. This connection stems from viewing our folding rule as a common extension of the existing multiple-clause folding rule, on the one hand, and an operator devised in inductive logic programming, called "intra-construction," on the other hand. Hence, our folding rule can be regarded as a step towards incorporating inductive inference into logic program transformation. We prove correctness with respect to Dung and Kanchanasut's semantic kernel.

Keywords: Logic program transformation, folding rule, most specific generalization, semantic kernel, decompilation, inductive logic programming.

1. Introduction

The *unfolding* rule for logic programs resolves a clause with all the clauses constituting a definition of a predicate occurring in the body of the clause, thus deriving possibly several clauses. The *folding* rule would in some sense be the inverse of such an operation. Hence, the folding rule must in general take several clauses, having some subgoals matching the bodies of a definition, and derive a single clause. Although sufficient for some important examples, the usual folding rule for logic programs is limited in that the set of subgoals that do not match the bodies of the definition must be the same in each clause to be folded. Our objective will be to study a folding rule closer to an inverse of unfolding: A multiple-clause *unfolding* can be viewed as consisting of several independent single-clause unfoldings (i.e. resolution steps). Hence, our multiple-clause *folding* rule is composed of several independent single-clause foldings. We next combine the resulting clauses by first instantiating certain variables and then computing the most specific common generalization of the instantiated clauses. We prove that our folding rule preserves Dung and Kanchanasut's semantic kernel [8].

After the initial work of Clark and Sickel [7] and Hogger [13] on logic program transformation, Tamaki and Sato [35] gave a system preserving the least Herbrand model of a definite program. Tamaki and Sato used a folding rule limited to a single clause, but such a rule has the advantage of being able to take the definition from previous programs in the transformation sequence. Figure 1 [23] illustrates a transformation sequence, indicating that the semantic value is preserved: $V = Sem(P_0) = Sem(P_1) = \ldots = Sem(P_n)$.

$$\begin{tikzcd} P_0 \arrow[r] \arrow[drr, "Sem"'] & P_1 \arrow[r] \arrow[dr, "Sem"] & \cdots \arrow[r] & P_n \arrow[dl, "Sem"] \\ & & V & \end{tikzcd}$$

Figure 1. *A transformation sequence.*

Within single-clause folding, on the one hand, Seki extended the transformation system of Tamaki and Sato to stratified programs in [33] and investigated the preservation of the well-founded semantics in [34]. Gardner and Shepherdson [10] further studied unfold/fold transformations of logic programs, exhibiting minor errors of [35], but restricting themselves to a folding rule taking the definition from the current program. Generalizing the work of Seki [34] on the preservation of well-founded semantics, Aravindan and Dung [1] showed that Tamaki and Sato's folding and unfolding rules preserve the semantic kernel [8] of a program, thereby preserving also the least Herbrand model, the perfect model semantics, the stable models, the well-founded models, the preferred extension semantics, the regular model semantics, the stable-theory semantics, and the set of partial stable models.

Within multiple-clause folding, on the other hand, Burstall and Darlington [6] already mention a simple multiple-equation folding rule (suggested to them by Dana Scott) for their first-order equational language. In logic programming, Maher [17] was perhaps the first to fold several clauses, using a definition in the current program. Gergatsoulis and Katzouraki [11], by contrast, studied multiple-clause folding when the definition belongs possibly to a previous program in the transformation sequence. In [26] Pettorossi, Proietti, and Renault incorporated a multiple-clause folding in their $\mathcal{D}$ strategy, deriving string-search algorithms from naive specifications. Roychoudhury, Kumar, Ramakrishnan, and Ramakrishnan [31] proved the preservation of the semantic kernel for a multiple-clause folding rule having bookkeeping information similar to Kanamori and Fujita's counters [14]. Kanamori and Fujita basically count the number of unfoldings and foldings applied to a clause, and only allow a folding application if the number of unfoldings exceeds the number of foldings of such a clause. Essentially, Roychoudhury, Kumar, Ramakrishnan, and Ramakrishnan extended this mechanism to multiple-clause folding by bounding from below and from above the number of unfoldings minus the number of foldings with a pair of counters for each clause.

Like [8, 31], we use the semantic kernel of the program. However, we avoid dealing with bookkeeping information by assuming that each single-clause folding occurring in our rule is correct with respect to the semantic kernel. Our work, in that sense, is complementary to that of transformation systems having such a bookkeeping, like the "foldable" flag of [35], the counters of [14], or the measure structures of [32] (which generalizes all these methods), and the descent levels of [36].

Section 2 argues in favor of extending the existing multiple-clause folding rule [11, 17, 26, 31]. First we give examples of (forward) program transformation, where the ordinary folding rule has to be preceded by the introduction of equations (via the "generalization + equality introduction" rule [29]). This introduction of equations, however, is closely related to adding the standard equality theory to the program, thereby creating a combinatorial explosion. By contrast, we do not require such equations.

As another motivation for our folding rule we consider backward program transformation and decompilation. Interest in decompilation can be traced back to [12, Chapter 11] for a language similar to Algol. Recently there has been increasing activity in decompilation, as large programming systems are normally built by coupling existing programs written in different programming languages. Decompilation then serves to reduce redundancy.

Our final application relates logic program transformation with inductive logic programming. We argue that transformation rules with inductive capabilities are important for program transformation. Our folding rule can be regarded as a step towards a common formalism for both these areas.

In Section 3 we design a first version of a folding rule based on multiple single-clause foldings. Since we wish to derive a single clause, we must combine the resulting clauses of each independent single-clause folding. In this first version we compute the most specific common generalization of such clauses. We study situations in which it is correct to perform the combination by computing such a generalization: we give sufficient conditions for preserving the semantic kernel.

Section 4 first shows that such sufficient conditions are readily violated. For example, it is easy to find clauses resulting from an unfolding application which cannot be folded by this first version of a folding rule. Next we observe that it is possible to apply an arbitrary substitution to the variables of the definition clauses which occur in the head of such clauses but not in the bodies. This instantiation preserves the semantic kernel of the program.

A second version of our folding rule appears in Section 5. This version is based on an existing algorithm for computing a most specific common generalization of expressions. Our extension of such an algorithm instantiates the variables occurring only in the head of the definition clauses as it computes the most specific generalization of the instantiated clauses.

The paper concludes in Section 6. We assume familiarity with logic programming [16], and we refer the reader to [21, 23] for surveys of logic program transformation.

2. Reasons for extending factor folding

In this section, we first review unfolding and single-clause folding. Next, we consider how such a folding rule has been extended in the literature so as to deal with more than one clause. This extension cannot fold the clauses resulting from any unfolding. Hence, we turn our attention to situations where it would be advantageous to have a better folding rule.

2.1. Single-clause folding

Throughout, A and B denote atoms, Q and R denote tuples of literals, C, D, E, and F denote clauses, and P denotes a normal program. Lower-case Greek letters denote substitutions, and upper-case Roman letters near the end of the alphabet denote variables. In addition, $\mathrm{var}(e)$ denotes the set of variables occurring in an expression e, $\mathrm{msg}(e_1, e_2)$ a most specific generalization [27, 30] of expressions e_1 and e_2, and $\mathrm{mgu}(A, B)$ a most general unifier of A and B.

We consider a transformation sequence $P_0, \ldots, P_N$. (Later we will require that P_{k+1} be obtained from P_k by applying a program transformation rule preserving the semantic kernel.)

Since the position of the literal(s) selected for folding or unfolding play(s) no role in our discussion, we will simplify the notation by writing such literal(s) first.

Let P_k be a normal program and $C = A \leftarrow B, R$ a clause in P_k. Let $D_i = B_i \leftarrow Q_i$, $i = 1, \ldots, n$ be variants sharing no variables with C of all the clauses in P_k whose heads unify with B, and let θ_i be an mgu of B and B_i. Then the *result of unfolding C with respect to B* is the program obtained from P_k by replacing C by the n clauses $E_i = A\theta_i \leftarrow Q_i\theta_i, R\theta_i$ obtained by resolving C with D_i with respect to B.

Let $E = A \leftarrow Q\theta, R$ be a clause in P_k and $D = B \leftarrow Q$ a variant of a clause in P_j, $j \leq k$, such that E and D share no variables. Then *E can be folded using D deriving $F = A \leftarrow B\theta, R$ with single-clause folding*, which we represent as:

$$\frac{B \leftarrow Q \quad (D) \qquad\qquad A \leftarrow Q\theta, R \quad (E)}{A \leftarrow B\theta, R \quad (F)}$$

provided:

1. D is the only variant of a clause in P_j whose head, after standardizing apart, unifies with $B\theta$, and

2. (*a*) θ maps the variables which occur in Q but not in B into distinct variables that do not occur in F, and (*b*) θ only acts on the variables occurring in Q.

D will be called the *definition*, E the *clause to be folded*, and F the *folded clause*. The *result of folding E using D* is the program obtained by replacing E by F.

2.2. Factor folding

This single-clause folding has been extended to several clauses [11, 17, 26, 32]. Like the folding rule we study here, the existing multiple-clause folding rule can be viewed as composed of several single-clause folding steps. Unlike our folding rule, however, such existing folding rule only covers the case where all the individual folding steps derive the same clause.

In our notation, the ordinary extension to two clauses would be:

$$\frac{B_1 \leftarrow Q_1 \quad (D_1) \qquad A \leftarrow Q_1\theta_1, R \quad (E_1) \qquad B_2 \leftarrow Q_2 \quad (D_2) \qquad A \leftarrow Q_2\theta_2, R \quad (E_2)}{A \leftarrow B, R \quad (F)} \tag{1}$$

where $B = B_1\theta_1 = B_2\theta_2$. (Conditions corresponding to those of single-clause folding must also be satisfied.)

Our work can be viewed as investigating the combination of the resulting folded clauses in case such clauses are not identical to each other. The ordinary multiple-clause folding only replaces the distinct subgoals (or factors) of the clauses to be folded by a single, common subgoal, leaving intact the rest of each clause to be folded. We will thus refer to such a folding as *factor folding*.

2.3. (Forward) program transformation

As an example where we would like to fold, but it is not possible to do so with factor folding, consider the following naive reverse program [21]:

$$\begin{aligned}
&rev([\,],[\,]) \leftarrow \\
&rev([X|Xs], Zs) \leftarrow rev(Xs, Ys),\ append(Ys, [X], Zs)
\end{aligned}$$

First, we introduce a new predicate ($revit$) by generalizing $[X]$ to Ys':

$$revit(Xs, Ys', Zs) \leftarrow rev(Xs, Ys),\ append(Ys, Ys', Zs)$$

Next, through standard unfolding and folding applications, as well as a property of $append$, it is possible to derive an iterative version of the reverse program:

$$rev([\,],[\,]) \leftarrow \tag{2}$$
$$rev([X|Xs], Zs) \leftarrow revit(Xs, [X], Zs) \tag{3}$$
$$revit([\,], Zs, Zs) \leftarrow \tag{4}$$
$$revit([X|Xs], Ys, Zs) \leftarrow revit(Xs, [X\,|\,Ys], Zs) \tag{5}$$

A more succinct program would be obtained by replacing (2) and (3) by:

$$rev(Xs, Zs) \leftarrow revit(Xs, [\,], Zs) \tag{6}$$

Factor folding, however, does not allow to derive (6). Note that if we unfold (6) using (4) and (5), we get (2) and (3). Hence, it seems legitimate to be able to fold (2) and (3) using (4) and (5), deriving (6).

Although such an extended folding step would not produce a more efficiently executable program, the more concise version of rev is preferable because of being less redundant, which makes it easier to modify.

It is interesting to note that already in [6, Section 6], Burstall and Darlington perceived a similar incompleteness in their transformation system (for a first-order equational language). Following a suggestion from Dana Scott, they added a rule (called "redefinition") [6, Section 7] replacing a multiple-equation definition by a single equation, using another multiple-equation definition.

In logic program transformation, when factor folding is not applicable, clauses are often massaged so as to obtain clauses conforming to D_i and E_i in (1). A rule sometimes called generalization + equality introduction [29] is then employed [22, 24]. Such a rule replaces a clause

$$A \leftarrow Q$$

by

$$A' \leftarrow X_1 = t_1, \ldots, X_m = t_m,\ Q'$$

where $(A', Q')\{X_1/t_1, \ldots, X_m/t_m\} = (A, Q)$.

To be able to apply factor folding in our example, we would have to replace (2)–(5) by:

$$rev(U, V) \leftarrow U = [\,],\ [\,] = V \tag{7}$$

$$rev(U, Zs) \leftarrow U = [X|Xs],\ revit(Xs, [X], Zs) \tag{8}$$

$$revit(U', V', Zs) \leftarrow U' = [\,],\ V' = Zs \tag{9}$$

$$revit(U', Ys, Zs) \leftarrow U' = [X|Xs],\ revit(Xs, [X\,|\,Ys], Zs) \tag{10}$$

Now both single-clause foldings: (7) using (9), and (8) using (10), produce the same clause, and factor folding can be used. However, (7) was obtained by applying the *symmetry axiom* ($X = Y \leftarrow Y = X$) of the standard equality theory after applying the generalization + equation introduction rule.

This use of the symmetry axiom suggests a drawback in employing equations. The application of the generalization + equation introduction rule has sometimes been referred to as not being trivial [22]. The reason is perhaps that such a rule is tantamount to successive applications of some standard-equality axioms, namely predicate substitutivity and function substitutivity. For a program P, these axioms are:

$$\{p(X_1, \ldots, X_{n_p}) \leftarrow X_1 = Y_1,\ \ldots,\ X_{n_p} = Y_{n_p},\ p(Y_1, \ldots, Y_{n_p}) : \ p \text{ is a predicate symbol occurring in } P\}$$

$$\{f(X_1, \ldots, X_{n_f}) = f(Y_1, \ldots, Y_{n_f}) \leftarrow X_1 = Y_1,\ \ldots,\ X_{n_f} = Y_{n_f} : \ f \text{ is a function symbol occurring in } P\}$$

Equality has played an important role in mechanized inference (e.g. [28]). In fact, before the advent of logic programming, it was common to include the standard equality axioms in resolution theorem provers. These axioms, however, are the source of a combinatorial explosion. Thus, in logic programming equality was avoided "like the plague" [9]. The standard equality axioms have been more recently introduced into logic programs to some degree as a means of incorporating functions. Such introduction is performed in [9], for example, by substantially restricting the order in which the axioms are applied. Moreover, [9] still avoids the symmetry axiom altogether.

Returning to our iterative example, we conclude that it is possible to apply the ordinary factor folding to (2) and (3) if we precede such a folding by generalization + equality introduction in addition to symmetry. However, the underlying search space may be intolerably large. (In other cases it is even necessary to add spurious equations.)

We now use an example studied by Pettorossi, Proietti, and Renault [25, 26], to further illustrate the limitations of factor folding. Consider the *string-matching problem*, consisting in finding all occurrences (or only one occurrence, if any) of a string called the *pattern* within another string called the *text*. The fact that the pattern *Pat* occurs in the text *Txt* can be expressed as the following specification using functional notation:

$$Txt = (X \bullet Pat) \bullet Z$$

where "$\bullet$" denotes string concatenation. These authors consider the case where we know *Pat* but not *Txt*. Hence, we wish to specialize the above specification to a known pattern. Let us assume, for clarity, a particular pattern, say 001. A translation of the above specification to a logic program using lists to represent strings would be:

$$s(T) \leftarrow append(X, [0,0,1], Y),\ append(Y, Z, T)$$

Pettorossi, Proietti, and Renault give an algorithm, called the $\mathcal{D}$ "strategy," indicating which transformation rule to apply at each step. Roughly speaking, the $\mathcal{D}$ strategy would derive, from this specification, a program corresponding to the finite-state automaton of the Knuth–Morris–Pratt algorithm [15]. Since we are not concerned here with the $\mathcal{D}$ strategy itself, we refer the interested reader to [25, 26], and give instead a use of the factor-folding rule. During the transformation process, we would fold the clauses:

$$\begin{aligned} &new_2([0|S]) \leftarrow append([0,1], \_, S) \\ &new_2([0|S]) \leftarrow append(\_, [0,0,1], X),\ append(X, \_, S) \end{aligned}$$

using variants of:

$$\begin{aligned} &new_3(S) \leftarrow append([0,1], \_, S) \\ &new_3(S) \leftarrow append(\_, [0,0,1], X),\ append(X, \_, S) \end{aligned}$$

deriving:

$$new_2([0|S]) \leftarrow new_3(S)$$

However, when we start from a right-associative specification, which is arguably equally naive:

$$Txt = X \bullet (Pat \bullet Z)$$

we may no longer be able to apply factor folding. A corresponding logic program would be:

$$s(T) \leftarrow append([0,0,1], Z, Y),\ append(X, Y, T)$$

The $\mathcal{D}$ strategy would prescribe first an unfolding step w.r.t. the left subgoal. Assuming the standard definition of $append$, we obtain:

$$s(T) \leftarrow append([0,1], X, Y'),\ append(Z, [0|Y'], T)$$

After three more unfolding steps, we get:

$$s(T) \leftarrow append(X, [0,0,1|Z], T)$$

The $\mathcal{D}$ strategy would indicate now another unfolding, obtaining:

$$s([0,0,1|T]) \leftarrow \tag{11}$$

$$s([C|T]) \leftarrow append(X, [0,0,1|Z], T) \tag{12}$$

Next we apply the "case-split" rule w.r.t. $C = 0$ and $C \neq 0$ to (12):

$$\begin{aligned} &s([0|T]) \leftarrow append(X, [0, 0, 1|Z], T) \\ &s([C|T]) \leftarrow C \neq 0,\ append(X, [0, 0, 1|Z], T) \end{aligned} \tag{13}$$

Finally, we would like to fold (11) and (13) to obtain:

$$s([0|T]) \leftarrow new'_3(T) \tag{14}$$

which is the clause we would need so as to derive a program corresponding to the desired finite-state automaton. However, we cannot perform such a folding step with factor folding because (11) and (13) do not have the same head (up to variable renaming).

As in the previous example, if we allow using the generalization + equality introduction rule, it is possible to employ factor folding. The price payed, however, may be a large search space.

2.4. Backward program transformation and decompilation

Normally, program transformation is used to obtain an implementation from a specification. The inverse process, however, can also be useful. Consider, for example, programming in the large. At present, large programming systems are often built from already-existing programs. Furthermore, such programs are frequently written in a variety of programming languages (say a commercial user interface or an off-the-shelf web-interaction module) of which we lack the source code. If such programs are coupled to each other without considering the possible duplication of code, redundancy may increase. This redundancy is undesirable, as it makes the whole system difficult to modify. The complexity of the whole system, in turn, may reach unmanageable levels. Hence, there has been a growing interest in reverse engineering and decompilation [3, 4, 5, 12, 20, 37].

A decompiler takes code written in assembly language and produces an equivalent program written in a high-level programming language. Often, however, strict decompilation is not enough, because *low-level* program constructs written in a high-level programming language may still be difficult to understand and modify. Therefore, decompilation is closely related to reengineering (where the target language is similar to or the same as the source language) [37]. Thus, reengineering involves being able to reverse ordinary program-transformation processes by obtaining high-level program constructs from low-level program constructs. An operation able to undo an arbitrary unfolding of a positive literal, like the one presented here, would contribute to a decompilation system being able to produce such high-level constructs.

2.5. Incorporating inductive logic programming into logic program transformation

Perhaps the most relevant application of an extended folding rule could be as a step towards incorporating inductive inference into logic program transformation. Program transformation can be considered a part of artificial intelligence [18]. Inductive inference, in turn, is a central topic of artificial intelligence. Therefore, having transformation rules equipped with inductive inference may benefit program transformation.

In fact, even among the earliest examples of program transformation, generalization is used as an operation [2, 6, 18, 38]. (Our reverse example uses generalization to suggest a definition of *revit* in terms of rev.) A case in point is that of Pettorossi and Proietti in [22, 24], where successive unfoldings produce a growing sequence of subgoals similar to each other. This sequence suggests defining a predicate having as argument an unbounded-length list of such subgoals. The recognition of the relationship between the pattern of growing subgoals and the definition of the new predicate is, however, performed with human intervention. A transformation system with inductive capabilities would, by contrast, detect not only this but other patterns.

At first sight, it might seem that inductive logic programming and logic program transformation have opposite objectives. While inductive inference is interested in generalizing from examples, program transformation is careful to preserve correctness. Two often used "operators" in inductive logic programming are "absorption" and "intra-construction." Absorption is reminiscent of single-clause folding, but does generalize, as expected. Intra-construction, by contrast, does not generalize. Moreover, intra-construction, like the multiple-clause folding, replaces several clauses by one, using a definition. Unlike folding, however, intra-construction produces itself the definition. In logic program transformation the definition us usually provided by a dedicated rule, "definition introduction."

Intra-construction is typically used to reorganize the program so far inferred. In the following example [19], we wish to infer a logic program for determining if a given number is the minimum of a list of numbers. After accepting several examples, the CIGOL system [19] produces (among others) the clauses:

$$min(D, [s(D)|E]) \leftarrow min(D, E) \tag{15}$$

$$min(F, [s(s(F))|G]) \leftarrow min(F, G) \tag{16}$$

to which intra-construction is applied, and both these clauses are replaced by:

$$\begin{aligned} &min(H, [I|J]) \leftarrow min(H, J),\ p(H, I) \\ &p(D, s(D)) \leftarrow \\ &p(F, s(s(F)) \leftarrow \end{aligned}$$

where p will denote the "less-than" relation.

The intra-construction operator produces a definition (i.e. p) consisting of unit clauses only, and constructs a clause as follows. To the most specific generalization [27, 30] $min(H, [I|J]) \leftarrow min(H, J)$ of the input clauses (15), (16) a subgoal $p(H, I)$ is added, having as arguments exactly all variables that represent a strict generalization. The clauses of the definition (15), (16), on the other hand, have as arguments terms than undo the generalization. Hence, intra-construction performs no strict generalization.

Replacing the intra-construction operator by our folding rule would have several advantages. First, it would allow us to use a definition already existing in the program (as opposed to "inventing" a new predicate every time). Second, inductive logic programming would be benefited by being able to fold using definitions having nonunit clauses. Finally, well-known results of nonmonotonic semantics already studied in logic program transformation could be applied to inductive logic programming. This link between inductive logic programming and logic program transformation would be a step towards developing program-transformation rules with inductive capabilities.

3. Generalization

This section develops a first, unsatisfactory, method for combining the clauses resulting from independent single-clause folding steps. The combination is simply a common generalization of such clauses. It will not be until Section 4, where we incorporate instantiations, that we will arrive at a better method.

3.1. A most specific generalization

An *unfolding* with respect to an atom unifying with two clause heads can be viewed as composed of two independent resolution steps (i.e. single-clause unfoldings). In addition, each such resolution step can be viewed as an instantiation of *both* clauses being resolved with each other, followed by the construction of a clause consisting of instantiated literals.

Observe now that if we unfold the folded clause F of a two-clause factor folding (1) (w.r.t. D_1 and D_2), such a folded clause F does not receive a substitution other than renaming. The reason is that when the mgu of variants of $B_1\theta_1$ and B_1 is applied, the variables occurring in the variant of $B_1\theta_1$ are only renamed. As a consequence, factor folding only corresponds to an inverse of unfolding applications in which the unfolded clause is not properly instantiated. Moreover, this suggests an operation corresponding to the inverse of the instantiation of F, namely, generalization.

We thus arrive at a first, naive, multiple-clause folding, consisting of several independent single-clause foldings, followed by a common generalization of the clauses resulting from each such independent step. Clearly, it is incorrect to generalize too much. This suggests computing the *most specific generalization* (msg) [27, 30] of the folded clauses.

A two-clause version of this folding rule would be:

$$\dfrac{\dfrac{B_1 \leftarrow Q_1 \quad (D_1) \qquad A_1 \leftarrow Q_1\theta_1, R_1 \quad (E_1)}{A_1 \leftarrow B_1\theta_1, R_1 \quad (F_1)}\,\text{fold} \qquad \dfrac{B_2 \leftarrow Q_2 \quad (D_2) \qquad A_2 \leftarrow Q_2\theta_2, R_2 \quad (E_2)}{A_2 \leftarrow B_2\theta_2, R_2 \quad (F_2)}\,\text{fold}}{A \leftarrow B, R \quad (F)}\,\text{msg}$$

We will refer as *segment* i to the inference of F_i from D_i and E_i.

3.2. Sufficient conditions

As a motivation for sufficient conditions for the correctness of this kind of folding, let us temporarily ignore the right segment of a two-clause msg folding rule, and suppose that we are interested in preserving the least Herbrand model.

Consider first the next folding segment followed by a generalization.

$$\dfrac{\dfrac{b(0) \leftarrow q(0) \qquad\qquad a \leftarrow q(0),\ r(0)}{a \leftarrow b(0),\ r(0)}\,\text{fold}}{a \leftarrow b(U),\ r(U)}\,\text{gen}$$

Here, both occurrences of 0 are replaced with the same variable U. Assuming that $b(U)$ does not unify with any clause head but that of $b(0) \leftarrow q(0)$, then the generalized program preserves the least Herbrand model.

The same cannot be said of this other generalization, which substitutes distinct variables for both occurrences of 0:

$$\cfrac{\cfrac{b(0) \leftarrow q(0) \qquad\qquad a \leftarrow q(0),\ r(0)}{a \leftarrow b(0),\ r(0)}\ \text{fold}}{a \leftarrow b(U),\ r(V)}\ \text{gen}$$

The atom a may now be inferred via the generalized clause from an atom $r(v)$ for v different from 0.

Let us now develop generalization conditions. Recall that the unfolding rule preserves the least Herbrand model of a program by the correctness of resolution. Therefore, one way of proving that a folding rule also preserves the least Herbrand model would be to prove that an application of such a folding rule can be undone by unfolding (see [10]).

Intuitively, a sufficient condition for a generalization preserving the least Herbrand model would be such an invertibility property.

Let $E_i = A_i \leftarrow Q_i\theta_i, R_i$ be clauses in P_k and $D_i = B_i \leftarrow Q_i$ variants of clauses in P_j, $j \leq k$, such that E_i and D_i share no variables. Let $F_i = A_i \leftarrow B_i\theta_i, R_i$ and $F = A \leftarrow B, R$ be an msg of all the clauses F_i. Then the clauses E_i *can be msg-folded using* D_i *deriving* F, provided:

M1: $F \cdot D_i \simeq E_i$,

M2: The D_i are the only variants of clauses in P_j whose head, after standardizing apart, unify with B, and

M3: (*a*) θ_i maps the variables which occur in Q_i but not in B_i into distinct variables that do not occur in F_i, and (*b*) θ_i only acts on the variables occurring in Q_i.

We will call these the *M conditions* (for msg folding). The *result of msg-folding the clauses* E_i *using* D_i is the program obtained by replacing the clauses E_i by F.

Here, and throughout the sequel, $C_1 \cdot C_2$ denotes an application of resolution to standardized apart variants of C_1 and C_2 w.r.t. the leftmost subgoal of such a variant of C_1, and $C_1 \simeq C_2$ denotes equality up to variable renaming. In addition, we will use $\theta_1 \simeq \theta_2$ for denoting equality of substitutions, except for their variable-renaming subset of bindings.

3.3. The semantic kernel of a program

We have selected as semantics Dung and Kanchanasut's [8] "semantic kernel." The reason is that the semantic kernel of a program preserves not only the least Herbrand model, but numerous nonmonotonic semantics as well.

Intuitively, the semantic kernel of a program generalizes the least Herbrand model by having clauses instead of atoms. The head of each such clause is ground and the body consists only of negative ground literals.

A *quasi-interpretation* of a program P is a set of clauses of the form

$$A \leftarrow \text{ not } B_1, \ldots, \text{ not } B_n \qquad n \geq 0$$

where A, B_i are ground atoms constructible with symbols of P. The set of quasi-interpretations of a program P is denoted by QI_P. Clearly, QI_P is a complete lattice w.r.t. set inclusion.

Let C' be a ground clause of the form

$$A \leftarrow A_1, \ldots, A_m, \text{ not } B_1, \ldots, \text{ not } B_n \qquad m, n \geq 0$$

and let C_i be ground clauses of the form

$$A_i \leftarrow Q_i \qquad 1 \leq i \leq m$$

Then $C = T(C', \{C_1, \ldots, C_m\})$ is the clause

$$A \leftarrow Q_1, \ldots, Q_m, \text{ not } B_1, \ldots, \text{ not } B_n$$

and C' is said to be a *generating clause* for C.

The transformation $S_P : \mathit{QI}_P \rightarrow \mathit{QI}_P$ from and to quasi-interpretations is defined as:

$$S_P(I) = \{T(C', \{C_1, \ldots, C_m\}) \mid C' \in \text{gnd}(P) \text{ and } C_i \in I, 1 \leq i \leq m\}$$

Here and in the sequel $\text{gnd}(P)$ denotes the set of all ground instances of every clause in P.

Let $\mathit{SK}^n(P) = S_P^n(\emptyset)$. Note that S_P is continuous. Hence, the least fixed point of S_P is given by

$$\mathit{SK}(P) = \bigcup \{\mathit{SK}^n(P) \mid n \geq 1\}$$

$\mathit{SK}(P)$ will be called the *semantic kernel* of P.

3.4. Local correctness

The correctness of single-clause or factor folding is normally achieved through some bookkeeping machinery [14, 32, 35, 36]. For instance, in [14], such a machinery consists in counting the number of unfoldings and foldings applied to a clause, and allowing a folding application only if the number of unfoldings exceeds the number of foldings of such a clause.

We wish to make the correctness of our folding operations independent of the particular bookkeeping machinery used. It might seem at first sight that we could base the correctness of msg folding on that of single-clause folding. This is not possible, however. Observe that a segment (inferring F_i from D_i and E_i) of msg folding does not conform to the definition of single-clause folding. Condition 1 of single-clause folding, translated to a segment of msg folding, would be that the leftmost subgoal of the folded clause F_i unifies with only one head clause in the program from which the definition clause D_i is taken. However, segments of msg folding violate such a condition.

As an example, consider the following msg folding (which is also a factor folding), where $q_1(1)$ and $q_2(2)$ hold in the program before folding.

$$\cfrac{\dfrac{b(X) \leftarrow q_1(X) \qquad a(Y) \leftarrow q_1(Y)}{a(Y) \leftarrow b(Y)}\ \text{fold} \qquad \dfrac{b(X') \leftarrow q_2(X') \qquad a(Y') \leftarrow q_2(Y')}{a(Y') \leftarrow b(Y')}\ \text{fold}}{a(U) \leftarrow b(U)}\ \text{msg}$$

Here, $b(Y)$ unifies both with $b(X)$ and $b(X')$. Yet this folding is correct in the sense that the program after folding $q_1(1)$ and $q_2(2)$ also hold (and no additional ground atoms hold as a result of such a folding).

We are therefore interested in establishing a certain "local" correctness for each segment. This correctness intuitively states that exactly the ground atoms which hold via a clause to be folded E_i also hold via the appropriate instance of the folded clause F_i.

First we define the subset of the semantic kernel relative to a particular clause C':

$$SK_{C'}(P) = \{C \in SK(P) \mid C'\gamma \text{ is a generating clause for } C \text{ and } C' \in P\}$$

Hence, the subset of the semantic kernel relative to a clause to be folded E_i in a program P_k is $SK_{E_i}(P_k)$.

To define the same subset in the semantic kernel for the resulting program P_{k+1} we first extend $SK_{C'}$ to a set $\mathcal{C}$ of clauses:

$$SK_{\mathcal{C}}(P) = \bigcup_{C \in \mathcal{C}} SK_C(P)$$

Next, we have to construct the instances of the head of F_i limiting ourselves to instances of the leftmost subgoal of F_i which are also instances of the head of the definition D_i:

$$\mathcal{F}_i = \{(A_i \leftarrow B_i\theta_i,\ R_i) \cdot C \mid C \in SK_{D_i}(P_j)\}$$

We can now define a *locally correct segment* i as one satisfying:

$$SK_{E_i}(P_k) = SK_{\mathcal{F}_i}(P_{k+1})$$

In the case of the previous example we get:

$$\begin{aligned} E_1 &= a(Y) \leftarrow q_1(Y) \\ D_1 &= b(X) \leftarrow q_1(X) \\ SK_{E_1}(P_k) &= \{a(1) \leftarrow\} \\ SK_{D_1}(P_j) &= \{b(1) \leftarrow\} \\ \mathcal{F}_1 &= \{(a(Y) \leftarrow b(Y)) \cdot (b(1) \leftarrow)\} \\ SK_{\mathcal{F}_1}(P_{k+1}) &= \{a(1) \leftarrow\} \end{aligned}$$

3.5. Preservation of the semantic kernel

We are now in a position to prove that the M conditions, together with local correctness, are sufficient for preserving the semantic kernel of a program. We proceed in a manner similar to those of [35, Lemma 1] and [1, Theorem 4.1]. Before that, however, we need a slightly different and more useful form of *M1*. Intuitively, this other form asserts that the application of the unifying substitution of F and D_i undoes the generalization.

Lemma 3.1. Suppose that F and D_i share no variables. Then *M1* and *M3* imply $(A \leftarrow B, R)\,\mathrm{mgu}(B, B_i) \simeq A_i \leftarrow B_i\theta_i, R_i$.

Proof:
M1 can be written as:

$$F \cdot D_i = (A \leftarrow Q_i, R)\mathrm{mgu}(B, B_i) \simeq E_i = A_i \leftarrow Q_i\theta_i, R_i \qquad \text{and} \tag{17}$$

$$B\,\mathrm{mgu}(B, B_i) = B_i\,\mathrm{mgu}(B, B_i) \tag{18}$$

By (17)

$$(A, R)\mathrm{mgu}(B, B_i) \simeq (A_i, R_i) \qquad \text{and}$$
$$Q_i\mathrm{mgu}(B, B_i) \simeq Q_i\theta_i \tag{19}$$

Hence, we need only prove $B\mathrm{mgu}(B, B_i) \simeq B_i\theta_i$.

By (19)

$$\mathrm{mgu}(B, B_i)|_{\mathrm{var}(Q_i)} \simeq \theta_i|_{\mathrm{var}(Q_i)} \tag{20}$$

Now $\mathrm{mgu}(B, B_i)$ does not act on $\mathrm{var}(Q_i) \setminus \mathrm{var}(B_i)$, and by condition *M3a* of msg folding, θ_i only renames $\mathrm{var}(Q_i) \setminus \mathrm{var}(B_i)$. Thus, we can strengthen (20) to:

$$\mathrm{mgu}(B, B_i)|_{\mathrm{var}(B_i)\cap\mathrm{var}(Q_i)} \simeq \theta_i|_{\mathrm{var}(B_i)\cap\mathrm{var}(Q_i)}$$

On the other hand, by condition *M3b*, θ_i does not act on $\mathrm{var}(B_i) \setminus \mathrm{var}(Q_i)$. In addition, $\mathrm{mgu}(B, B_i)$ either does not act on, or only renames, $\mathrm{var}(B_i) \setminus \mathrm{var}(Q_i)$. To see this, consider a term $t[X]$ occurring in B_i having an occurrence of a variable $X \in \mathrm{var}(B_i) \setminus \mathrm{var}(Q_i)$. If such an occurrence of $t[X]$ is generalized by msg to a variable, then X does not occur in B and $\mathrm{mgu}(B, B_i)$ does not act on X. If, on the other hand, such an occurrence of $t[X]$ is renamed by msg, then $\mathrm{mgu}(B, B_i)$ also renames X.

Therefore, $\mathrm{mgu}(B, B_i)|_{\mathrm{var}(B_i)} \simeq \theta_i|_{\mathrm{var}(B_i)}$. It follows that $B_i\mathrm{mgu}(B, B_i) \simeq B_i\theta_i$, which, together with (18) implies:

$$B\mathrm{mgu}(B, B_i) \simeq B_i\theta_i$$

We conclude that the lemma holds. □

Theorem 3.1. Let $P_0, \ldots, P_k$ be a correct transformation sequence w.r.t. the semantic kernel, i.e. $SK(P_0) = SK(P_d)$, $0 \le d \le k$. Let P_{k+1} be a program obtained by an msg folding in which the definition is in P_j, $j \le k$, and each segment of such a folding is locally correct. Then $SK(P_{k+1}) = SK(P_k)$.

Proof:
Since each segment is locally correct and the folded clause F is a generalization of the folded clause F_l of each segment, we need only prove $SK(P_{k+1}) \subseteq SK(P_k)$. We prove the following proposition by induction on i:

$$\forall i \text{ If } C \in SK^i(P_{k+1}), \text{ then } C \in SK(P_k)$$

Basis: If $i = 0$, then the proposition is obvious.
Induction Step: Assuming that the proposition is true for $i \le I$, we have to show that it holds for $i = I + 1$. That is, given $C \in SK^{I+1}(P_{k+1})$, we have to show that $C \in SK(P_k)$.

Let C' be a generating clause for C, i.e. $C = T(C', \{C_1, \ldots, C_n\})$. We need only consider the case in which $C' = F\phi = (A \leftarrow B, R)\phi$, where $F \in P_{k+1}$ and ϕ is a grounding substitution.

Because B and B_l unify for some l, there exists a grounding substitution ϕ_l, such that $(A \leftarrow B, R)\phi = (A \leftarrow B, R)\mathrm{mgu}(B, B_l)\phi_l$. By Lemma 3.1, $(A \leftarrow B, R)\mathrm{mgu}(B, B_l) \simeq (A_l \leftarrow B_l\theta_l, R_l)$. Hence, for some grounding substitution ϕ_l', $(A \leftarrow B, R)\mathrm{mgu}(B, B_l)\phi_l = (A_l \leftarrow B_l\theta_l, R_l)\phi_l'$. In addition, $\mathrm{head}(C_1) = B_l\theta_l\phi_l'$ and $\{\mathrm{head}(C_d) \mid d \ge 2\}$ is the set of positive literals in $R_l\phi_l'$.

By the induction hypothesis:

$$C_1, \ldots, C_n \in SK(P_k)$$

By the correctness of the transformation sequence:

$$C_1 \in SK(P_j)$$

Because B and B_l unify, $\mathrm{head}(C_1) = B_l\rho_l$, where ρ_l grounds $D_l = B_l \leftarrow Q_l$. Moreover, there exist

$$C_{1,1}, \ldots, C_{1,m} \in SK(P_j) \text{ s.t. } C_1 = T(D_l\rho_l, \{C_{1,1}, \ldots, C_{1,m}\})$$

By the correctness of the transformation,

$$C_{1,1}, \ldots, C_{1,m} \in SK(P_k)$$

where $\{\mathrm{head}(C_{1,d}) \mid d \geq 1\}$ is the set of positive literals in $Q_l\rho_l$.

We must now prove that $Q_l\rho_l$ is equal to $Q_l\theta_l\phi'_l$ for some ϕ'_l grounding $E_l = A_l \leftarrow Q_l\theta_l, R_l$. We distinguish three cases for a variable $X \in \mathrm{var}(D_l)$:

i. $X \in \mathrm{var}(Q_l) \setminus \mathrm{var}(B_l)$:

ϕ'_l must undo the application of θ_l and then apply ρ_l.

$$X\rho_l = X\theta_l\phi'_l \text{ if } \phi'_l \supseteq (\theta_l|_{\mathrm{var}(Q_l)\setminus\mathrm{var}(B_l)})^{-1}\rho_l$$

where $\sigma^{-1} = \{t/X \mid X/t \in \sigma\}$. By condition *M3a*, $(\theta_l|_{\mathrm{var}(Q_l)\setminus\mathrm{var}(B_l)})^{-1}$ is a substitution.

ii. $X \in \mathrm{var}(Q_l) \cap \mathrm{var}(B_l)$:

$B_l\rho_l = B_l\theta_l\phi'_l$ implies $X\rho_l = X\theta_l\phi'_l$.

iii. $X \in \mathrm{var}(B_l) \setminus \mathrm{var}(Q_l)$:

X may receive any substitution, as X is not necessary to make $Q_l\rho_l = Q_l\theta_l\phi'_l$.

Now,

$$C = T(E_l\phi'_l, \{C_{1,1}, \ldots, C_{1,m}, C_2, \ldots, C_n\})$$

Since all of $C_{1,1}, \ldots, C_{1,m}, C_2, \ldots, C_n$ are present in $SK(P_k)$, it follows that $C \in SK(P_k)$. We conclude that the theorem holds. □

An example illustrating cases *i* and *ii* would be:

$$\dfrac{\dfrac{b(X) \leftarrow q(X, X') \quad (D_1) \qquad\qquad a \leftarrow q(f(Y), Y'),\ r(Y'') \quad (E_1)}{a \leftarrow b(f(Y)),\ r(Y'') \quad (F_1)}\ \text{fold}}{a \leftarrow b(f(U)),\ r(V) \quad (F)}\ \text{msg}$$

$$\begin{aligned} \theta_1 &= \{X/f(Y), X'/Y'\} \\ \rho_1 &= \{X/f(0), X'/1\} \\ \phi'_1 &= \{Y/0, Y'/1, Y''/2\} \\ \phi_1 &= \{U/0, V/2\} \\ \phi &= \{U/0, V/2\} \end{aligned}$$

The folding rule given in this section is not able to fold the reverse or the string-matching examples of Section 2. Such a folding rule, however, allowed us to perform a separation of concerns by introducing first a simplified version of our folding rule, and the full-fledged version next.

4. Instantiation of head-only variables of the definition

Thus far we have given sufficient conditions for preserving the semantic kernel of a program when folding several clauses. We will note in this section, however, that in many cases in which we would like to fold, *M1* is not satisfied. Next, we will observe that it is possible to instantiate arbitrarily the variables occurring in the head and not in the body of a definition clause. This instantiation allows us to satisfy *M1* more often.

4.1. Motivation for the instantiation

We would like to be able to fold the clauses resulting from an arbitrary unfolding of a positive literal. Consider, as an example, an unfolding application of the clause:

$$a \leftarrow append(\_, [0], W),\ r(W) \tag{21}$$

assuming the standard definition of $append$:

$$append([\,], Y, Y) \leftarrow \tag{22}$$
$$append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z) \tag{23}$$

where, without danger of ambiguity, A denotes a variable. We derive:

$$a \leftarrow r([0]) \tag{24}$$
$$a \leftarrow append(\_, [0], Z),\ r([A|Z]) \tag{25}$$

Let us now try to fold (24) and (25) using variants of (22) and (23). The first segment would be:

$$\frac{append([\,], Y, Y) \leftarrow \qquad\qquad a \leftarrow r([0])}{a \leftarrow append([\,], Y, Y),\ r([0])}\ \text{fold}$$

and the second one:

$$\frac{append([A'|X'], Y', [A'|Z']) \leftarrow append(X', Y', Z') \qquad\qquad a \leftarrow append(\_, [0], Z),\ r([A|Z])}{a \leftarrow append([A'|\_], [0], [A'|Z]),\ r([A|Z])}\ \text{fold}$$

Let us now compute an msg of the folded clauses:

$$\frac{a \leftarrow append([\,], Y, Y),\ r([0]) \qquad\qquad a \leftarrow append([A'|\_], [0], [A'|Z]),\ r([A|Z])}{a \leftarrow append(\_, U, V),\ r([W_1|W_2]) \qquad (F)}\ \text{msg}$$

Note now that the generalized clause violates *M1* for both segments.

For the first segment we get:

$$F \cdot (22) = a \leftarrow r([W_1|W_2]) \\ \not\approx a \leftarrow r([0])$$

and for the second one:

$$F \cdot (23) = a \leftarrow \mathit{append}(X', Y', Z'),\ r([W_1|W_2]) \\ \not\approx a \leftarrow \mathit{append}(\_, [0], Z),\ r([A|Z])$$

Observe that when we resolve (21) with (22), the Y is instantiated to [0]. However, when we generalize, the Y is used without any instantiation. This suggests instantiating Y in the left clause of the generalization.

Such an instantiation is not enough, however. An msg is:

$$\frac{a \leftarrow \mathit{append}([\,], [0], [0]),\ r([0]) \qquad\qquad a \leftarrow \mathit{append}([A'|\_], [0], [A'|Z]),\ r([A|Z])}{a \leftarrow \mathit{append}(\_, [0], [U|V]),\ r([W|V]) \qquad (F')}\ \text{msg}$$

which still violates *M1* in the first segment:

$$F' \cdot (22) = a \leftarrow r([W]) \\ \not\approx a \leftarrow r([0])$$

Similarly, when resolving (21) with (23), both occurrences of the W receive $[A|Z]$. This suggests instantiating A' to A in the clause resulting from the second segment.

After performing both these instantiations:

$$\frac{a \leftarrow \mathit{append}([\,], [0], [0]),\ r([0]) \qquad\qquad a \leftarrow \mathit{append}([A|\_], [0], [A|Z]),\ r([A|Z])}{a \leftarrow \mathit{append}(\_, [0], [U|V]),\ r([U|V])}\ \text{msg}$$

this new msg satisfies *M1*.

Now two questions arise: (1) Which variables is it possible to instantiate without compromising the total correctness? and (2) How do we instantiate such variables?

4.2. Head-only variables of the definition

As to the variables we can instantiate, consider the variables occurring in the head B_i and not in the body Q_i of a definition clause D_i (e.g. the Y in the first segment and the A' in the second segment). These variables occur in the folded clause $F_i = A_i \leftarrow B_i\theta_i, R_i$ of a single clause folding application because θ_i does not act on such variables. If we resolve F_i with a standardized apart variant of D_i, then these variables disappear. This suggests that we can apply an arbitrary substitution to the variables of D_i occurring in F_i and yet preserve the semantic kernel.

It is easy to see that the proof of Theorem 3.1 is also valid for this new kind of multiple-clause folding: Observe that the variables occurring in the head and not in the body of a definition do not play any role in such a proof (case *iii*). Hence, Theorem 3.1 also holds for such a folding. We therefore extend the current definition of msg folding with head-only substitutions and keep the same name for this new kind of folding.

Let $E_i = A_i \leftarrow Q_i\theta_i, R_i$ be clauses in P_k and $D_i = B_i \leftarrow Q_i$ variants of clauses in P_j, $j \leq k$, such that E_i and D_i share no variables. Let $F_i = A_i \leftarrow B_i\theta_i\tau_i, R_i$ and $F = A \leftarrow B, R$ be an msg of all the clauses F_i, where τ_i acts only on $\mathrm{var}(B_i) \setminus \mathrm{var}(Q_i)$. Then the clauses E_i *can be msg-folded using* D_i *deriving* F, provided the *M* conditions are satisfied.

5. Instantiation algorithm

We have identified the instantiation of head-only variables in a definition clause as a stage preceding the generalization. In this section we give an algorithm for such an instantiation.

5.1. An msg algorithm

We wish to instantiate head-only variables in such a way as to satisfy the *M1* condition (if possible) when computing an msg. This suggests reviewing algorithms for computing msgs: [27, 30]. Let us consider, with no loss of generality, an msg of only two terms. Intuitively, an msg of terms t_1 and t_2 is obtained by overlapping t_1 and t_2 and by taking the part which is common to both t_1 and t_2 starting from the root. For every occurrence of two corresponding parts which are not common to t_1 and t_2, $\mathrm{msg}(t_1, t_2)$ has the same variable.

Nondeterminism will play an important role in our algorithm. Thus, we found Prolog a convenient programming language for our purposes. For conciseness, however, we will add syntactic sugar. For example, it will be convenient to combine functional and relational notations, so that a call to a procedure named MSG will have as value an msg of two given terms t_1 and t_2. However, standard Prolog code of our main procedures can be found at the URL `http://leibniz.iimas.unam.mx/~drosenbl/fold`.

The MSG procedure is meant for computing not only $\mathrm{msg}(t_1, t_2)$, but also substitutions ρ_1 and ρ_2 such that $\mathrm{msg}(t_1, t_2)\rho_1 = t_1$, and $\mathrm{msg}(t_1, t_2)\rho_2 = t_2$. We will therefore add two "output" argument places for holding respectively the values of ρ_1, and ρ_2.

On the other hand, recall that for every occurrence of two corresponding parts which are not common to t_1 and t_2, $\mathrm{msg}(t_1, t_2)$ has the same variable. Hence, it will be advantageous to have two more additional argument places for substitutions recording previous occurrences of pairs of noncommon parts s_1 and s_2 of t_1 and t_2, respectively, and their corresponding variable U. Such substitutions will include, respectively, the bindings U/s_1 and U/s_2. Initially, these additional argument places will then have ϵ, the identity substitution: $\mathrm{MSG}(t_1, t_2, \epsilon, \epsilon, \rho_1, \rho_2)$.

We now give a procedure implementing the algorithm of [27, 30]. We assume that t_1, t_2 are either atoms or terms, with no variables in common. In case t_1 and t_2 are atoms, then they have the same predicate symbol. By convention, V is a variable not occurring in t_i or ϕ_i^1, and f is an arbitrary predicate or function symbol.

$\mathrm{MSG}(t_1, t_2, \phi_1^1, \phi_2^1, \rho_1, \rho_2) = t$ **where**
if $t_1 = f(r_1, \ldots, r_n)$ **and** $t_2 = f(s_1, \ldots, s_n)$
then $t := f(\mathrm{MSG}(r_1, s_1, \phi_1^1, \phi_2^1, \phi_1^2, \phi_2^2), \ldots, \mathrm{MSG}(r_n, s_n, \phi_1^n, \phi_2^n, \rho_1, \rho_2))$
else **if** $\exists U$ s.t. $U/t_1 \in \phi_1^1$ **and** $U/t_2 \in \phi_2^1$
then $\rho_1 := \phi_1^1;\ \rho_2 := \phi_2^1;\ t := U$

else $\rho_1 := \{V/t_1\} \cup \phi_1^1;\ \rho_2 := \{V/t_2\} \cup \phi_2^1;\ t := V$

The intended meaning would be:

$$\begin{aligned} \textsc{Msg}(t_1, t_2, \phi_1, \phi_2, \rho_1, \rho_2) &= \mathrm{msg}(t_1, t_2) \\ \mathrm{msg}(t_1, t_2)\rho_i &= t_i \\ U/s_i \in \phi_i \text{ and } V/s_i \in \rho_i \text{ implies } U &= V \end{aligned}$$

5.2. Coverage and matching

Consider now an msg folding with segments clauses E_1 and E_2, using D_1 and D_2, deriving $F_1 = A_1 \leftarrow B_1\theta_1, R_1$ and $F_2 = A_2 \leftarrow B_2\theta_2, R_2$, respectively. Let $F = \mathrm{msg}(F_1, F_2) = A \leftarrow B, R$. Observe that if a variable U representing a strict generalization occurs in (A, R) and not in B, then U will not be instantiated when resolving F with a standardized apart variant of D_i and the *M1* condition ($F \cdot D_i \simeq E_i$) might not be satisfied.

Consider as an example the following msg:

$$\frac{a \leftarrow b(X),\ r(e(f)) \qquad\qquad a \leftarrow b(h(g)),\ r(e(g))}{a \leftarrow b(V),\ r(e(U))}\ \text{msg}$$

Variable U represents a strict generalization and does not occur in $B = b(V)$. Even though we have not given the clauses to be folded E_i or the definition D_i, the resulting msg clearly violates condition *M1*. The reason is that the mgu of $F \cdot D_i$ will not act on U. Therefore, $R\mathrm{mgu}(B_i, B) \neq R_i$.

This suggests first computing $\textsc{Msg}((A_1, R_1), (A_2, R_2), \epsilon, \epsilon, \phi_1, \phi_2)$ (with a minor extension to handle pairs of atoms), and then using ϕ_i ($i = 1, 2$) for instantiating the variables of F_i which occur also in D_i (the head-only variables of D_i) in such a way that variables representing strict generalizations appear in B.

Now, not only locating the head-only variables in $B_i\theta_i$, but also computing $\mathrm{msg}(B_1\theta_1, B_2\theta_2)$ require traversing $B_i\theta_i$. Both these traversals can be incorporated into a single one by extending the Msg procedure.

If X in the above example is a head-only variable, we can instantiate such a variable. Which instantiation should we use? If we instantiate X (say) to $h(g)$:

$$\frac{a \leftarrow b(h(g)),\ r(e(f)) \qquad\qquad a \leftarrow b(h(g)),\ r(e(g))}{a \leftarrow b(h(g)),\ r(e(U))}\ \text{msg}$$

we will be in a similar (undesirable) situation as before.

By contrast, if we use the substitution $\{X/h(f)\}$:

$$\frac{a \leftarrow b(h(f)),\ r(e(f)) \qquad\qquad a \leftarrow b(h(g)),\ r(e(g))}{a \leftarrow b(h(U)),\ r(e(U))}\ \text{msg}$$

it is possible that we now satisfy condition *M1* simply because the variable introduced in R (i.e. U in $r(e(U))$) also occurs in B, and the mgu of the resolution step $F \cdot D_i$ may act on R.

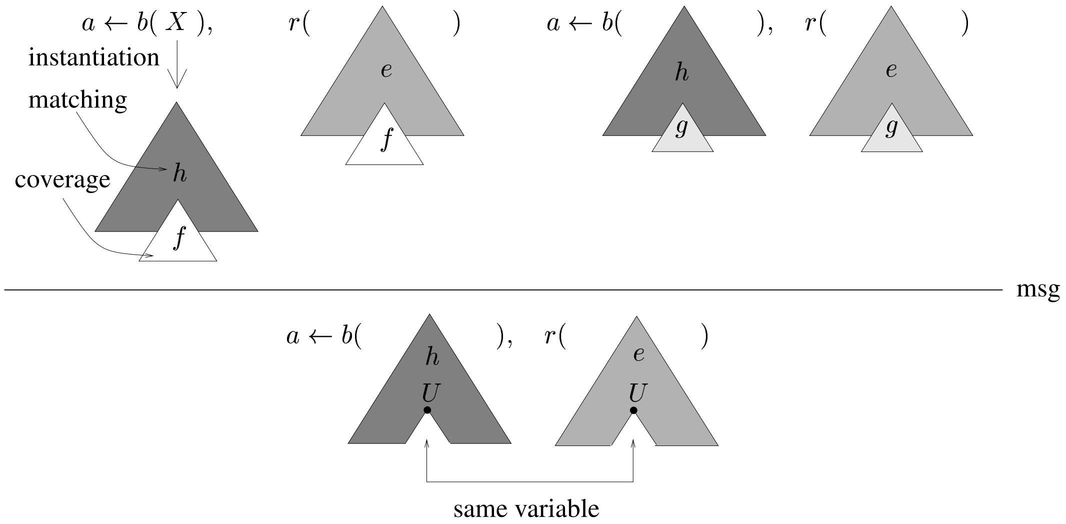

Figure 2. *Instantiation example.*

We observe that X must be instantiated to a term constructed both from function symbols of R_1 (i.e. f) and $B_2\theta_2$ (i.e. h). Figure 2 illustrates this situation. Note that any of the function symbols in this example could in fact represent a greater fragment of a term (or even be absent).

The f function symbol, taken from R_1, is used in the instance of X by a operation we will call *coverage*, as the pair (f, g) has already been "covered" when computing $\mathrm{msg}(R_1, R_2)$. The h function symbol, taken from $B_2\theta_2$, is used in the instance of X by a operation we will call *matching*, as it is essentially a one-way unification.

Our modification of the MSG procedure must be able to perform both these operations. Before modifying such a procedure, however, it will be convenient to consider the clauses resulting from an unfolding application. The reason is that we wish to determine whether or not our operations suffice to fold such clauses.

5.3. Clauses resulting from an unfolding

Let us unfold $F' = A' \leftarrow B', R'$ assuming a definition with only two clauses $D_i = B_i \leftarrow Q_i$ $(i = 1, 2)$ having heads unifying with B'. We will represent this unfolding as:

$$\frac{B_1 \leftarrow Q_1 \quad (D_1) \qquad\qquad B_2 \leftarrow Q_2 \quad (D_2) \qquad\qquad A' \leftarrow B', R' \quad (F')}{(A' \leftarrow Q_1, R')\sigma_1 \quad (E_1) \qquad\qquad (A' \leftarrow Q_2, R')\sigma_2 \quad (E_2)}\ \text{unfold} \tag{26}$$

We are interested in studying the substitutions τ_1, τ_2 in the folding of E_1 and E_2 using variants of D_1 and D_2:

$$\dfrac{\dfrac{\dfrac{B_1 \leftarrow Q_1 \quad (D_1) \qquad A_1 \leftarrow Q_1\theta_1, R_1 \quad (E_1)}{A_1 \leftarrow B_1\theta_1, R_1}\,\text{fold}}{A_1 \leftarrow B_1\theta_1\tau_1, R_1}\,\text{instance} \qquad \dfrac{\dfrac{B_2 \leftarrow Q_2 \quad (D_2) \qquad A_2 \leftarrow Q_2\theta_2, R_2 \quad (E_2)}{A_2 \leftarrow B_2\theta_2, R_2}\,\text{fold}}{A_2 \leftarrow B_2\theta_2\tau_2, R_2}\,\text{instance}}{A \leftarrow B, R \quad (F)}\,\text{msg}$$

Observe that we are not interested in recovering a variant of F', which is impossible in general from E_i and D_i alone. (To see this, consider single-clause folding, which is a particular case of our folding.)

We will analyze all possible combinations of argument places of B', $B_1\theta_1$ and $B_2\theta_2$, and try to reduce such combinations to the fewest possible cases. For clarity, we will consider only the top-level (i.e. the argument places) of the terms occurring in such atoms. Later, however, we will explain why it is that it suffices to do so without having to study occurrences of terms at other levels.

An argument place in B' may be either a variable Z or a nonvariable term t. Note, however, that it is important in the variable case to distinguish between Z occurring in (A', R') or not. The reason is that only if Z occurs in (A', R') will the substitution received by Z, i.e. $Z\sigma_i$, appear in E_i. Moreover, if Z does not occur in (A', R'), then we can view Z as if it were a nonvariable term, because the substitution received by Z will not appear in E_i. (We give illustrations below.) Therefore, there are two possibilities for an argument place in B':

(a) a variable Z which occurs also in (A', R'), or

(b) a nonvariable term t or a variable Z which does not occur in (A', R').

At the same time, each argument place of $B_i\theta_i$ $(i = 1, 2)$ can be a variable, which may or may not occur also in $Q_i\theta_i$ (i.e. in E_i), or can be a nonvariable term. Note that we can only instantiate variables occurring in $B_i\theta_i$ and not occurring in $Q_i\theta_i$. Therefore, a variable occurring both in $B_i\theta_i$ and $Q_i\theta_i$ can be treated as a nonvariable term. In the case of a head-only variable, we will assume that such a variable occurs in $B_1\theta_1$. (The case of the head-only variable occurring in $B_2\theta_2$ is symmetrical.) This leaves two possibilities for argument places of $B_1\theta_1$ and $B_2\theta_2$, both of which have a head-only variable X in $B_1\theta_1$:

(i) X corresponds to a term t' in $B_2\theta_2$ which is either a variable also occurring in $Q_2\theta_2$ (i.e. in E_2) or a nonvariable term, or

(ii) X corresponds to another head-only variable Y in $B_2\theta_2$ (i.e. not occurring in E_2).

We are consequently left with four combinations.

5.4. Cases (i)

Cases (a)-(i) and (b)-(i) are more intuitive and hence we deal with them first. We give first an example of (a)-(i). Here, the head-only variable X corresponds to a nonvariable term 0:

$$\dfrac{b(X) \leftarrow \qquad b(0) \leftarrow \qquad a \leftarrow b(Z), r(Z)}{a \leftarrow r(X) \qquad a \leftarrow r(0)}\,\text{unfold}$$

Next we fold the resulting clauses with an instantiation of X that can be performed with coverage:

$$\dfrac{\dfrac{b(Y) \leftarrow \qquad a \leftarrow r(X)}{a \leftarrow b(Y), r(X)}\ \text{fold} \qquad \dfrac{b(0) \leftarrow \qquad a \leftarrow r(0)}{a \leftarrow b(0), r(0)}\ \text{fold}}{a \leftarrow b(U), r(U)}\ \text{msg with } \{Y/X\}$$

Observe that *M1* is satisfied: If F is the folded clause, then $F \cdot D_i \simeq E_i$.

We turn our attention to case (b)-(i). In the next example, the argument place of B' is a nonvariable term 0 and the corresponding argument place in B_2 is a variable Y occurring also in Q_2. An unfolding application would be:

$$\dfrac{b(X) \leftarrow \qquad a \leftarrow b(0), r \qquad b(Y) \leftarrow q(Y)}{a \leftarrow r \qquad a \leftarrow q(0), r}\ \text{unfold}$$

The following folding application instantiates X with matching:

$$\dfrac{\dfrac{b(X) \leftarrow \qquad a \leftarrow r}{a \leftarrow b(X), r}\ \text{fold} \qquad \dfrac{b(Y) \leftarrow q(Y) \qquad a \leftarrow q(0), r}{a \leftarrow b(0), r}\ \text{fold}}{a \leftarrow b(0), r}\ \text{msg with } \{X/0\}$$

and we can confirm that *M1* holds.

Remark at this point that the folding of clauses resulting from unfolding faces a nondeterministic situation. We are assuming that F', the clause that was unfolded, is not available. Because the clauses E_i and D_i by themselves are not enough to distinguish between (a)-(i) and (b)-(i), we must *nondeterministically* consider both possibilities. In case both coverage and matching are applicable, we should apply both operations. We could then test for condition *M1*, which may eliminate some candidates for the folded clause.

Recall that we seek an extension of the MSG procedure such that, given ϕ_1 and ϕ_2 (i.e. the substitutions constructed when computing $\text{msg}((A_1, R_1), (A_2, R_2))$), the extended algorithm instantiates head-only variables of $B_1\theta_1$ and $B_2\theta_2$, and computes an msg of the instantiated subgoals. Let us deal with coverage first.

The coverage operation uses the bindings in ϕ_i to determine an instantiation. Thus, such an operation can be incorporated at the point in which we inspect such bindings. This inspection occurs in line 3 of the MSG procedure. We can therefore establish the membership of a binding U/t_i *as we instantiate appropriate variables of* t_i.

We give an implementation using the following conventions: If x is a variable and s a term, then $x := s$ means that x is instantiated to s. HEADONLY(x) is intended to hold if x is a head-only variable.

MSG$'(t_1, t_2, \phi_1^1, \phi_2^1, \rho_1, \rho_2) = t$ **where**

1 **if** $t_1 = f(r_1, \ldots, r_n)$ **and** $t_2 = f(s_1, \ldots, s_n)$
2 **then** $t := f(\text{MSG}'(r_1, s_1, \phi_1^1, \phi_2^1, \phi_1^2, \phi_2^2), \ldots, \text{MSG}'(r_n, s_n, \phi_1^n, \phi_2^n, \rho_1, \rho_2))$
3 **else** **if** $\exists U$ s.t. $U/t_1' \in \phi_1^1$ **and** $U/t_2' \in \phi_2^1$ **and** EQ(t_1, t_1') **and** EQ(t_2, t_2')
4 **then** $\rho_1 := \phi_1^1;\ \rho_2 := \phi_2^1;\ t := U$
5 **else** $\rho_1 := \{V/t_1\} \cup \phi_1^1;\ \rho_2 := \{V/t_2\} \cup \phi_2^1;\ t := V$

The EQ procedure implements coverage. Given terms t and t', EQ instantiates head-only variables occurring in t in such a way that $t = t'$ holds, if possible.

$\text{EQ}(t, t') = p$ **where**
if $t = f(s_1, \ldots, s_n)$ **and** $t' = f(s'_1, \ldots, s'_n)$
then $p := \text{EQ}(s_1, s'_1)$ **and** $\cdots$ **and** $\text{EQ}(s_n, s'_n)$
else **if** $\text{VARIABLE}(t)$ **and** $t = t'$
  then $p :=$ **success**
  else **if** $\text{HEADONLY}(t)$
    then $t := t'$;   *% coverage*
      $p :=$ **success**
    else $p :=$ **failure**

Observe first that the coverage operation is by itself nondeterministic. The reason is that there may be more than one binding $U/t'_i \in \phi_i^1$ such that $\text{EQ}(t_i, t'_i)$ succeeds. We would like to have all possible answers resulting from such nondeterminism. Hence, we would need to be able to undo the instantiations. Therefore, the MSG' and EQ procedures can be readily implemented in a programming language such as Prolog, representing variables of the object level with metavariables.

Let us now consider the matching operation. Assume a head-only variable X corresponding to a nonvariable term t'. If we apply matching, we only wish to copy the (main) function symbol from t'. The reason is that we have to allow for the possibility of constructing subterms of the instance of X with *coverage*, as illustrated in Fig. 2. As a result, the new terms to which the head-only variables are instantiated happens incrementally, from top to bottom. Thus, when we determine the function symbol f of a new term, we have not yet determined the arguments of such a term. This invites instantiating a head-only variable to an "incomplete" term $f(V_1, \ldots, V_n)$, and treating $V_1, \ldots, V_n$ *as if they were head-only variables*. We recursively define a variable as *instantiable* if it is a head-only variable, or it occurs in a term to which an instantiable variable has been instantiated. We must therefore modify the EQ procedure by calling an auxiliary predicate $\text{INSTANTIABLE}(x)$, which holds if x is instantiable, instead of calling $\text{HEADONLY}(x)$.

Having incorporated coverage, we now add matching to MSG'. Below, $V_1, \ldots, V_n$ name distinct variables not occurring in either t_j or ϕ_j^1.

$\text{MSG}'(t_1, t_2, \phi_1^1, \phi_2^1, \rho_1, \rho_2) = t$ **where**
6 **if** $\text{INSTANTIABLE}(t_1)$ **and** $t_2 = f(s_1, \ldots, s_n)$
7 **then** $t_1 := f(V_1, \ldots, V_n)$;   *% matching*
8    $t := f(\text{MSG}'(V_1, s_1, \phi_1^1, \phi_2^1, \phi_1^2, \phi_2^2), \ldots, \text{MSG}'(V_n, s_n, \phi_1^n, \phi_2^n, \rho_1, \rho_2))$

$\text{MSG}'(t_1, t_2, \phi_1^1, \phi_2^1, \rho_1, \rho_2) = t$ **where**
9 **if** $t_1 = f(r_1, \ldots, r_n)$ **and** $\text{INSTANTIABLE}(t_2)$
10 **then** $t_2 := f(V_1, \ldots, V_n)$;   *% matching*
11    $t := f(\text{MSG}'(r_1, V_1, \phi_1^1, \phi_2^1, \phi_1^2, \phi_2^2), \ldots, \text{MSG}'(r_n, V_n, \phi_1^n, \phi_2^n, \rho_1, \rho_2))$

The cases of t_1 being an instantiable variable and t_2 another variable which is not instantiable, and vice versa, are already handled in line 5 of the MSG' procedure.

Now that we have made the coverage and matching operations applicable to subterms other than the terms at the top level, we have justified our having only considered the argument places in our four unfolding cases.

5.5. Cases (ii)

Let us now focus on the cases in which an instantiable variable corresponds to another instantiable variable.

An example of case (a)-(ii), where the instantiable variables in $B_i\theta_i$ correspond to a variable Z which occurs also in (A', R'), would be:

$$\frac{b(X,X) \leftarrow \qquad b(Y,0) \leftarrow \qquad a \leftarrow b(Z,0), r(Z)}{a \leftarrow r(0) \qquad a \leftarrow r(Z)} \text{ unfold}$$

We can fold by *simultaneously* applying coverage to X and Y:

$$\frac{\dfrac{b(X,X) \leftarrow \qquad a \leftarrow r(0)}{a \leftarrow b(X,X), r(0)} \text{ fold} \qquad \dfrac{b(Y,0) \leftarrow \qquad a \leftarrow r(Z)}{a \leftarrow b(Y,0), r(Z)} \text{ fold}}{a \leftarrow b(U,0), r(U)} \text{ msg with } \{X/0, Y/Z\}$$

Note, however, that such a simultaneous application of coverage is already handled by the MSG$'$ algorithm, because in line 3 of MSG$'$ there are two calls to EQ.

It remains to treat case (b)-(ii), where the instantiable variables in $B_i\theta_i$ correspond to a nonvariable term t or a variable Z which does not occur in (A', R'). This case is more problematic because we could in principle fold by applying matching, but we do not know which instance each of the variables will receive, if any, later on (caused by other occurrences of such variables). An example would be:

$$\frac{b(X,X) \leftarrow \qquad b(Y,1) \leftarrow \qquad a \leftarrow b(0,Z), r(Z)}{a \leftarrow r(0) \qquad a \leftarrow r(1)} \text{ unfold}$$

which could be folded (if we knew the appropriate substitution) as follows:

$$\frac{\dfrac{b(X,X) \leftarrow \qquad a \leftarrow r(0)}{a \leftarrow b(X,X), r(0)} \text{ fold} \qquad \dfrac{b(Y,1) \leftarrow \qquad a \leftarrow r(1)}{a \leftarrow b(Y,1), r(1)} \text{ fold}}{a \leftarrow b(0,U), r(U)} \text{ msg with } \{X/0, Y/0\}$$

If we traversed the argument places of $B_i\theta_i$ from right to left, we could first use coverage in the right occurrence of X, which would apply the substitution $\{X/0\}$. Next, we would use matching in the Y, which would apply the substitution $\{Y/0\}$. However, because we would like to traverse the argument places only once, and from left to right, we would need to apply matching to two head-only variables (the left occurrences of X and Y), without knowing the instances that the variables will receive later on.

Notice that the matching operation copies a function symbol from one clause to the other, and to their common generalization as well. This suggests *unifying* both instantiable variables with each other, and with the corresponding variable in the generalization as well. As a result, the instances that one variable receives further on in the construction of the common generalization will automatically be applied to the other two variables. In the above example, once X and Y and a variable in the folded clause, say V, are unified with each other, the coverage applied to the right occurrence of X instantiates all three variables to 0 as if matching had been applied. Let us call this new operation *variable unification*.

Variable unification requires an important consideration. Because the terms t_1 and t_2 may now share variables, we must perform all instantiations with an occur check.

We are now ready to add variable unification to $\textsc{Msg}'$.

$\textsc{Msg}'(t_1, t_2, \phi_1^1, \phi_2^1, \phi_1^1, \phi_2^1) = t$ **where**
if $\textsc{Instantiable}(t_1)$ **and** $\textsc{Instantiable}(t_2)$
  then $t_1 := t_2;\ t := t_1$ *% variable unification*

The following table summarizes the behavior of the $\textsc{Msg}'$ algorithm on clauses resulting from an unfolding (26).

	$X \quad t'$	$X \quad Y$
$Z \in \mathrm{var}(A', R')$	coverage	(simultaneous) coverage
t or $Z \notin \mathrm{var}(A', R')$	matching	variable unification

Z is a variable occurring in B', t is a nonvariable term occurring in B'. X and Y are variables occurring in $B_i\theta_i$ and not occurring in $Q_i\theta_i$ (i.e. head-only variables). t' is a term occurring in $B_i\theta_i$, which is either a variable also occurring in $Q_i\theta_i$ or a nonvariable term.

With no loss of generality, we give the top-level program for folding only two clauses:

$\textsc{InstGen}(D_1, E_1, F_1, D_2, E_2, F_2, F) = p$ **where**
$F_1 = A_1 \leftarrow B_1, R_1;$
$F_2 = A_2 \leftarrow B_2, R_2;$
$(A, R) := \textsc{Msg}((A_1, R_1), (A_2, R_2), \epsilon, \epsilon, \phi_1, \phi_2);$
$B := \textsc{Msg}'(B_1, B_2, \phi_1, \phi_2, \_, \_);$
$F := A \leftarrow B, R;$
if $F \cdot D_1 \simeq E_1$ **and** $F \cdot D_2 \simeq E_2$
then $p :=$ **success**
else **if** $\textsc{PossibleBacktrack}$
  then $\textsc{Backtrack}$
  else $p :=$ **failure**

We have now proved:

Theorem 5.1. Let E_i be the clauses resulting from an unfolding application using $D_i, i = 1, \ldots, n$. The $\textsc{InstGen}$ algorithm folds E_i using D_i deriving a clause F such that $F \cdot D_i \simeq E_i$.

We finish this section with a few examples. Consider first the next unfold:

$$\frac{b(X, X', X) \leftarrow \qquad b(Y', Y, Y) \leftarrow \qquad a \leftarrow b(f, g, Z), r(Z)}{a \leftarrow r(f) \qquad a \leftarrow r(g)}\ \text{unfold}$$

We can then fold as follows:

$$\cfrac{\cfrac{b(X,X',X) \leftarrow \qquad a \leftarrow r(f)}{a \leftarrow b(X,X',X), r(f)}\ \text{fold} \qquad \cfrac{b(Y',Y,Y) \leftarrow \qquad a \leftarrow r(g)}{a \leftarrow b(Y',Y,Y), r(g)}\ \text{fold}}{a \leftarrow b(f,g,U), r(U)}\ \begin{array}{l}\text{msg with}\\ \{X/Y', X'/Y\}\{X/f, Y/g\}\end{array}$$

We first unified X with Y', then we unified X' with Y, and finally we applied coverage to the right occurrences of X and Y. The folded clause satisfies *M1*. From the clauses to be folded and the definition alone we cannot know that we have to apply variable unification twice followed by coverage. However, the MSG$'$ algorithm eventually tries such a combination.

Now we turn to the string-matching example of Section 2. We wished to fold:

$$\begin{array}{l} s([0,0,1|T]) \leftarrow \\ s([0|T]) \leftarrow \mathit{append}(X,[0,0,1|Z],T) \end{array}$$

using variants of:

$$\begin{array}{l} \mathit{new}_3'([0,1|T]) \leftarrow \\ \mathit{new}_3'(T) \leftarrow \mathit{append}(Z,[0,0,1|Z],T) \end{array}$$

Our folding rule computes:

$$\frac{\mathit{new}_3'([0,1|T']) \leftarrow \qquad s([0,0,1|T]) \leftarrow}{s([0,0,1|T]) \leftarrow \mathit{new}_3'([0,1|T'])}\ \text{fold}$$

$$\frac{\mathit{new}_3'(T) \leftarrow \mathit{append}(Z,[0,0,1|Z],T) \qquad s([0|T'']) \leftarrow \mathit{append}(X,[0,0,1|Z],T'')}{s([0|T'']) \leftarrow \mathit{new}_3'(T'')}\ \text{fold}$$

$$\frac{s([0,0,1|T]) \leftarrow \mathit{new}_3'([0,1|T']) \qquad s([0|T'']) \leftarrow \mathit{new}_3'(T'')}{s([0|U]) \leftarrow \mathit{new}_3'(U)}\ \text{msg with } \{T/T'\}$$

instantiating T to T' by coverage.

Finally, we return to the iterative reverse program, also in Section 2.

$$\begin{array}{l} \mathit{rev}([\,],[\,]) \leftarrow \\ \mathit{rev}([X|Xs], Zs) \leftarrow \mathit{revit}(Xs,[X],Zs) \\ \mathit{revit}([\,], Zs, Zs) \leftarrow \\ \mathit{revit}([X|Xs], Ys, Zs) \leftarrow \mathit{revit}(Xs,[X|Ys],Zs) \end{array}$$

Applying msg folding we obtain:

$$\frac{revit([\,], Zs, Zs) \leftarrow \qquad\qquad rev([\,],[\,]) \leftarrow}{rev([\,],[\,]) \leftarrow revit([\,], Zs, Zs)} \text{ fold}$$

$$\frac{revit([X|Xs], Ys, Zs) \leftarrow revit(Xs, [X|Ys], Zs) \qquad\qquad rev([X|Xs], Zs') \leftarrow revit(Xs, [X], Zs')}{rev([X|Xs], Zs') \leftarrow revit([X|Xs], [\,], Zs')} \text{ fold}$$

$$\frac{rev([\,],[\,]) \leftarrow revit([\,], Zs, Zs) \qquad\qquad rev([X|Xs], Zs') \leftarrow revit([X|Xs], [\,], Zs')}{rev(U, V) \leftarrow revit(U, [\,], V)} \text{ msg with } \{Zs/[\,]\}$$

by instantiating Zs to $[\,]$ by matching.

6. Concluding remarks

We conclude by first summarizing three possible applications to our folding rule. Next, we give a short description of msg folding. Finally, we suggest possible avenues for research.

6.1. Applications

A multiple-equation folding rule can be traced back to Burstall and Darlington seminal work [6]. Their folding rule is applicable when the right-hand sides of the definition of a function to be folded are pairwise identical to the right-hand sides of another definition. Such a rule allows these authors to derive some concise versions of iterative programs underivable otherwise. Although the application of their folding rule does not produce more efficiently executable programs, it does derive less redundant, and hence easier to modify, definitions.

Several researchers [11, 26, 31] have studied a multiple-clause folding rule for logic programs. This folding rule, however, is limited in that the set of subgoals that do not match the bodies of the definition must be the same in each clause to be folded. In spite of this limitation, such a folding rule has found applications when an additional rule is employed in conjunction. This additional rule, sometimes called generalization + equality introduction [29], can in many cases rewrite the clauses involved (the clauses to be folded as well as the definition used) so as to overcome this limitation. However, the generalization + equality introduction rule is difficult to use [22]. The reason is possibly that such a rule amounts to applications of axioms of the standard equality theory. This theory was avoided "like the plague" [9] in the advent of logic programming, as it is the source of a combinatorial explosion. By contrast, we presented a folding rule that does not require equality axioms.

We first rephrased as a logic program an example used by Burstall and Darlington [6] illustrating the use of their folding rule. The corresponding logic program cannot be made as concise as we would ideally wished if the ordinary folding rule alone is used. Our folding rule does derive the concise version.

Next, we provided another example by slightly modifying a program used by Pettorossi, Proietti, and Renault [25, 26] exhibiting the ordinary multiple-clause folding. To be sure, our modification of this example can be transformed with the ordinary folding rule, but equality axioms must be employed as well, possibly creating a large search space.

Having mentioned two applications of (forward) program transformation, we then turned our attention to decompilation and backward program transformation. Decompilation has had a long history [3, 12]. Moreover, recently there has been an increased interest [4, 5, 20, 37] because large systems are often built by coupling several off-the-shelf programs. If coupled without considering the possible duplication of code, the resulting system may be highly redundant and therefore difficult to modify. Decompilation and backward program transformation allow the programmer to detect and reduce this redundancy.

As another application of our folding rule, we argued for the development of transformation rules equipped with inductive inference. Program transformation is often regarded as a part of artificial intelligence [18], where inductive inference plays a prominent role. Generalizations occur frequently in the program transformation literature [2, 6, 18, 22, 24, 38]. As a step towards incorporating inference into program transformation rules, we considered an operator called intra-construction [19], developed for inductive logic programming. This operator does not perform strict generalizations, but rather reorganizes the program inferred so far. Replacing intra-construction by our folding rule would facilitate the development of a formalism common to logic program transformation and inductive logic programming.

6.2. Msg folding

The ordinary multiple-clause folding rule can be regarded as the application of several independent single-clause foldings, all of which derive the same clause. We have extended this folding rule in two steps.

In the first step, we observed that just as in the unfolding rule the clause to be unfolded is instantiated, we should construct (for an inverse rule) a *common generalization* of the clauses resulting from individual folding applications. Next, we indicated that in general it is incorrect (under the least Herbrand model semantics, say) to compute such a common generalization.

Consider for example an unfolding of $F' = A' \leftarrow B', R'$ assuming that the only clause whose head unifies with B' is the definition clause $D = B \leftarrow Q$. The resulting clause is $E = (A' \leftarrow Q, R')\sigma$, where σ is an mgu of B' and B. Assume now that we do not have access to F', and that we fold $E = A \leftarrow Q\theta, R$ using D. The folded clause is then $F = A \leftarrow B\theta, R$. However, θ only acts on the variables occurring in Q. Observe that it is possible for σ to act on the variables occurring in B and not in Q. Therefore, the folding application does not have access to such bindings.

The second step of our work can be viewed as a study of possible bindings (missing in θ) that would make the folded clause satisfy sufficiency conditions for folding. Our instantiation algorithm is an elaboration of an existing algorithm for computing a most specific generalization (msg) [27, 30], where we instantiate the variables occurring in the head and not occurring in the body of the definition clauses. Our algorithm employs essentially two operations: (1) coverage, using terms already “covered” in the partial construction of the common generalization, and (2) matching, copying a function symbol from the clauses obtained with the other single-clause foldings.

We proved that our folding rule is able to fold the clauses resulting from an arbitrary unfolding of a positive literal. Moreover, our folding rule assumes that the definition is possibly drawn from a previous program in the transformation sequence. If each single-clause folding application of a multiple-clause folding preserves the semantic kernel of Dung and Kanachanasut [8], then so does the clause computed with our instantiation/generalization algorithm.

6.3. Future work

In a transformation sequence, whether a folding rule is applicable or not is highly dependent on the definition used. Sometimes such a definition is taken from either the current or a previous program, but other times it is added afresh. In general, it is not easy to know which of the previous definitions is appropriate or when it is preferable to introduce a new definition. Just as our folding rule suggests how to instantiate head-only variables of the definition, it might be possible to devise a better folding rule that would propose other characteristics of a usable definition. Thus, a possible line of research could be to extend msg folding so as to offer a definition for folding.

Another direction for research would be to establish a link between logic program transformation and inductive logic programming. We have here observed similarities between our folding rule and the existing intra-construction operator. A transformation system with inductive capabilities would be a significant contribution.

Acknowledgments

I am very grateful to Julio C. Peralta. In addition to giving me continuous moral support, he spent numerous hours listening to me and giving me intelligent suggestions. The referees' insightful comments substantially improved a draft of this paper. We also acknowledge the facilities provided by IIMAS, UNAM.

References

[1] Aravindan, C., Dung, P. M.: On the correctness of unfold/fold transformation of normal and extended logic programs, *The Journal of Logic Programming*, **24**(3), 1995, 201–217.

[2] Arsac, J., Kodratoff, Y.: Some techniques for recursion removal from recursive functions, *ACM Transactions on Programming Languages and Systems*, **4**(2), 1982, 295–322.

[3] Baker, B. S.: An algorithm for structuring flowgraphs, *Journal of the ACM*, **24**(1), 1977, 98–120.

[4] Bowen, J. P.: From programs to object code and back again using logic programming: compilation and decompilation, *Journal of Software Maintenance: Research and Practice*, **5**(4), 1993, 205–234.

[5] Breuer, P. T., Bowen, J. P.: Decompilation: the enumeration of types and grammars, *ACM Transactions on Programming Languages and Systems*, **16**(5), 1994, 1613–1647.

[6] Burstall, R. M., Darlington, J.: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, **24**(1), 1977, 44–67.

[7] Clark, K., Sickel, S.: Predicate logic: a calculus for deriving programs, *Proc. IJCAI-77*, 1977.

[8] Dung, P. M., Kanchanasut, K.: A fixpoint approach to declarative semantics of logic programs, *North American Logic Programming Conference* (E. L. Lusk, R. A. Overbeek, Eds.), 1989.

[9] van Emden, M. H., Yukawa, K.: Logic programming with equations, *The Journal of Logic Programming*, **4**, 1987, 265–288.

[10] Gardner, P. A., Shepherdson, J. C.: Unfold/Fold Transformations of Logic Programs, in: *Computational Logic. Essays in Honor of Alan Robinson* (J.-L. Lassez, G. Plotkin, Eds.), The MIT Press, 1992, 565–583.

[11] Gergatsoulis, M., Katzouraki, M.: Unfold/fold transformations for definite clause programs, *Proc. 6th International Symposium, Programming Language Implementation and Logic Programming* (M. Hermenegildo, J. Penjam, Eds.), Springer-Verlag, Madrid, Spain, 1994, Lecture Notes in Computer Science No. 844.

[12] Halstead, M. H.: *Machine-independent computer programming*, Spartan Books, 1962.

[13] Hogger, C. J.: Derivation of Logic Programs, *Journal of the ACM*, **28**(2), 1981, 372–392.

[14] Kanamori, T., Fujita, H.: *Unfold/fold transfomation of logic programs with counters*, Technical report, ICOT, 1986.

[15] Knuth, D. E., Morris, J. H., Pratt, V. R.: Fast Pattern Matching in Strings, *SIAM Journal on Computing*, **6**(2), 1977, 323–350.

[16] Lloyd, J. W.: *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, 1987.

[17] Maher, M. J.: A Transformation System for Deductive Database Modules with Perfect Model Semantics, *Theoretical Computer Science*, **110**, 1993, 377–403.

[18] Manna, Z., Waldinger, R.: Synthesis: Dreams ⇒ Programs, *IEEE Transactions on Software Engineering*, **SE-5**(4), 1979, 294–328.

[19] Muggleton, S., Buntine, W.: Machine Invention of First-order Predicates by Inverting Resolution, *Proc. Fifth International Conference on Machine Learning* (J. Laird, Ed.), Morgan Kaufmann, 1988, Also in S. Muggleton (ed), *Inductive Logic Programming*, Academic Press, 1992, APIC-38, pp 261–280.

[20] Mycroft, A.: Type-based decompilation, *Proc. Eighth European Symposium on Programming* (S. D. Swierstra, Ed.), 1999, Lecture Notes in Computer Science No. 1576.

[21] Pettorossi, A., Proietti, M.: Transformation of Logic Programs: Foundations and Techniques, *The Journal of Logic Programming*, **19, 20**, 1994, 261–320.

[22] Pettorossi, A., Proietti, M.: Program Derivation via List Introduction, *Proc. IFIP TC2 Working Conference on Algorithmic Languages and Calculi* (R. Bird, L. Meertens, Eds.), Chapman & Hall, 1997.

[23] Pettorossi, A., Proietti, M.: Transformation of Logic Programs, in: *Handbook of Logic in Artificial Intelligence and Logic Programming* (D. M. Gabbay, C. J. Hogger, J. A. Robinson, Eds.), vol. 5, Oxford University Press, 1998, 697–787.

[24] Pettorossi, A., Proietti, M.: The List Introduction Strategy for the Derivation of Logic Programs, *Formal Aspects of Computing*, **13**, 2002, 233–251.

[25] Pettorossi, A., Proietti, M., Renault, S.: Derivation of Efficient Logic Programs by Specialization and Reduction of Nondeterminism, *to appear in: Higher-Order and Symbolic Computation*.

[26] Pettorossi, A., Proietti, M., Renault, S.: Enhancing Partial Deduction via Unfold/Fold Rules, *Proc. 6th Int. Workshop on Logic Program Synthesis and Transformation*, Springer-Verlag, Stockholm, Sweden, 1996, Lecture Notes in Computer Science No. 1207.

[27] Plotkin, G. D.: A note on inductive generalization, in: *Machine Intelligence 5* (B. Meltzer, D. Michie, Eds.), Edinburgh University Press, 1970, 153–163.

[28] Plotkin, G. D.: Building-in equational theories, in: *Machine Intelligence 7* (B. Meltzer, D. Michie, Eds.), Edinburgh University Press, 1972, 73–90.

[29] Proietti, M., Pettorossi, A.: Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs, *Theoretical Computer Science*, **142**, 1995, 89–124.

[30] Reynolds, J. C.: Transformational Systems and the Algebraic Structure of Atomic Formulas, in: *Machine Intelligence 5* (B. Meltzer, D. Michie, Eds.), Edinburgh University Press, 1970, 135–151.

[31] Roychoudhury, A., Kumar, K. N., Ramakrishnan, C. R., Ramakrishnan, I.: Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs, *International Journal of Foundations of Computer Science*, **13**(3), 2002, 387–403.

[32] Roychoudhury, A., Kumar, K. N., Ramakrishnan, C. R., Ramakrishnan, I. V.: A parameterized unfold/fold transformation framework for definite logic programs, *Proc. Principles and Practice of Declarative Programming* (G. Nadathur, Ed.), Springer-Verlag, 1999, Lecture Notes in Computer Science No. 1702.

[33] Seki, H.: Unfold/fold transformation of stratified programs, *Theoretical Computer Science*, **86**, 1991, 107–139.

[34] Seki, H.: Unfold/fold transformation of general logic programs for the well-founded semantics, *The Journal of Logic Programming*, **16**, 1993, 5–23.

[35] Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs, *Proc. Second International Logic Programming Conference*, 1984.

[36] Tamaki, H., Sato, T.: *A generalized correctness proof of the unfold/fold logic program transformation*, Technical report, Ibaraki University, 1986.

[37] Ward, M. P.: Pigs from sausages? reengineering from assembler to C via FermaT transformation, *Science of Computer Programming (special issue on program transformation)*, **52**(1–3), 2004, 213–255.

[38] Wegbreit, B.: *Goal-Directed Program Transformation*, Technical Report CSL-75-8, XEROX Palo Alto Research Center, September 1975.