In:O. Danvy (Ed.) Proceedings of the 2nd ACM SIGPLAN Workshop on Continuations, January 14, 1997, ENS, Paris, France, 1997, pp. 9.1–9.22.

Flexible Continuations in Logic Programs via Unfold/Fold Transformations and Goal Generalization

Alberto Pettorossi
Department of Informatics, Systems, and Production, University of Roma II, Via della Ricerca Scientifica, 00133 Roma, Italy. adp@iasi.rm.cnr.it

Maurizio Proietti
IASI-CNR,
Viale Manzoni 30, 00185 Roma, Italy. proietti@iasi.rm.cnr.it

Abstract

We consider the use of continuations for deriving efficient logic programs. It is known that both in the case of functional and logic programming, the introduction of continuations is a valuable technique for transforming old programs into new, more efficient ones. However, in general, in order to derive programs with high levels of efficiency, one should introduce continuations according to suitable strategies. In particular, we show that it is preferable to introduce continuations in a flexible way, that is, during the process of program transformation itself, rather than at its beginning or at its end. We extend logic programs by allowing variables to range over goals, and we propose a set of transformation rules for this extended language. We propose a generalization strategy for the introduction of goal variables which may be viewed as continuations and they allow for the derivation of very efficient programs.

1 Introduction

Continuation-based program transformations [22] have been first proposed to improve the performance of functional programs. These transformations are based on a particular form of function generalization which consists in the introduction of an extra argument representing the 'future course of a computation'. The introduction of continuation arguments does not generate per se more efficient programs, but it may allow for subsequent, efficiency improving transformations. Moreover, the particular syntactic form of the derived programs may allow us to use sophisticated compilers, thereby producing very efficient code (see, for instance, [1]).

The transformation of logic programs into continuation passing style programs was also proposed in [20]. This particular transformation of logic programs is called *binarization* and the programs which are derived are called *binary* programs. Program binarization consists in adding to each predicate a continuation argument, which is a term representing the conjunction of goals which have to be evaluated before completing the evaluation of the given initial goal. Below we will address the problem due to this representation of a conjunction of goals via a term. The topmost goal to be evaluated by a given program has the continuation argument *true*, and indeed *true* is the neutral element w.r.t. a conjunction of goals.

Let us consider now the following logic program:

1. $maxlist([\,], 0) \leftarrow$
2. $maxlist([H|T], M) \leftarrow maxlist(T, MT),\ max(H, MT, M)$

and the corresponding binary program obtained by binarization:

3. $maxlist(L, S) \leftarrow maxlist\_cont(L, S, true)$
4. $maxlist\_cont([\,], 0, Cont) \leftarrow Cont$
5. $maxlist\_cont([H|T], M, Cont) \leftarrow$
   $maxlist\_cont(T, MT, max\_cont(H, MT, M, Cont))$

where the third argument of *maxlist_cont* is the continuation argument and $max\_cont(H, MT, M, Cont)$ is a new basic predicate such that for every goal *Cont* we have that $max\_cont(H, MT, M, Cont)$ holds iff the conjunction of $max(H, MT, M)$ and *Cont* holds.

Notice that in the binary program shown above, the variable *Cont* occurs both in an argument position and a goal position (see clause 4). Moreover, the predicate *max_cont* also occurs in an argument of the predicate *maxlist_cont*. Thus, strictly speaking, binary programs cannot be considered as *definite programs* [9] (that is, logic programs in the usual sense of Horn clause theories), because in binary programs goal variables, that is, variables ranging over goals, and predicate symbols occurring in argument positions, are not allowed. Moreover, the use of goal variables and the occurrence of predicate symbols in argument positions are allowed by most Prolog compilers which are currently available.

Binary programs allow for efficient implementations because they require only a reduced set of WAM instructions [4, 23]. However, naive binarizations may even decrease efficiency, because they introduce some extra arguments in which complex terms occur. It is known from the literature that in order to get efficient binary programs one should allow after the introduction of continuations, for some program specialization and simplification steps [3, 10].

In this paper we show that binarization can also be realized via an extension of the usual unfold/fold transformation rules [2, 12, 18], where we allow for the introduction and the manipulation of goal variables. By using this extended set of rules, we may derive very efficient programs because during binarization we may also perform efficiency improving program transformations, such as, the elimination of intermediate data structures [2, 15, 21].

The introduction of continuations performed within this transformational framework, is said to be *flexible* for indicating that continuations are introduced neither at the beginning nor at the end of the transformation process, and instead, they are introduced during the transformation process itself. In what follows we will feel free to refer to these continuations as *flexible continuations.* The introduction of our flexible continuations is interleaved with other transformation steps according to a particular strategy, called *goal generalization*, which depends on the syntactic form of the program at hand.

Through an example we show that the flexible introduction of continuations based on the extended unfold/fold transformation rules, is superior to the technique which first performs binarization and then applies the usual unfold/fold transformation rules without the introduction of goal variables. This is due to the fact that in order to perform folding steps and derive efficient programs, it may be necessary to perform generalization of the continuation arguments, and in general, it is very difficult to determine in advance, at the beginning of the transformation process, the generalized continuations which are suitable for folding.

We also prove that our technique for the flexible introduction of continuations is also superior to the other technique which first applies usual unfold/fold transformations and then performs binarization of the derived program.

2 Continuations via Unfold/Fold and Goal Variable Introduction

In this section we show that binarization can be realized via unfold/fold transformations, if we allow for a rule which introduces goal variables.

Let us consider the program made out of clauses 1 and 2 above. We can then introduce the following two clauses:

6. $maxlist\_cont(L, M, Cont) \leftarrow maxlist(L, M),\ \ Cont$
7. $max\_cont(X, Y, M, Cont) \leftarrow max(X, Y, M),\ \ Cont$

in which we have introduced the goal variable *Cont* as an argument of the predicates *maxlist_cont* and *max_cont*].

The transformational derivation of a new program for *maxlist_cont* proceeds by unfolding clause 6. We get two clauses:

8. $maxlist\_cont([\ ], 0, Cont) \leftarrow Cont$
9. $maxlist\_cont([H|T], M, Cont) \leftarrow maxlist(T, MT),$
$\qquad max(H, MT, M),\ \ Cont$

Now clause 9 can be folded using clauses 6 and 7, and we get:

10. $maxlist\_cont([H|T], MT, Cont) \leftarrow$
$\qquad maxlist\_cont(T, MT, max\_cont(H, MT, M, Cont))$

Clauses 8 and 10 are exactly clauses 4 and 5 which define the predicate *maxlist_cont* introduced by binarization.

The reader may realize that the program derivation from clauses $\{1, 2\}$ to clauses $\{8, 10\}$ can be promoted to a general technique in the sense that by performing analogous transformation steps and introducing goal variables we can derive the binary program corresponding to any given initial program. Thus, our transformation technique which uses the unfold/fold rules together with the introduction of goal variables, is at least as powerful as binarization. In Section 4 we show that our transformation technique is, in fact, more powerful and it allows for more flexibility as we will later specify.

3 Transformation Rules

In this section we formally present the set of rules we use for deriving from a given program P a new program P' by transformation. These transformation rules are an extension of those in [18] and as already anticipated, they allow for the introduction of goal variables (see the definition and goal abstraction rules below).

Let us first specify the syntax of our extended language, called *transformation language*, where we allow for occurrences of goals in argument positions and for occurrences of variables ranging over goals. We assume the following four disjoint sets: (i) *individual variables*, ranged over by X, (ii) *goal variables*, ranged over by G, (iii) *function symbols* with arities, ranged over by f, and (iv) *predicate symbols* with arities, ranged over by

p. As usual, arities may assume any non-negative integer value. Individual variables and goal variables will collectively be called *variables* and ranged over by V. The sets of terms, atoms, and goals are mutually defined as follows:

$$\begin{array}{lcl} \mathit{Terms} \ni t & ::= & X \ | \ f(t_1,\ldots,t_n) \ | \ g \\ \mathit{Atoms} \ni a & ::= & p(t_1,\ldots,t_m) \ | \ G \lhd g \\ \mathit{Goals} \ni g & ::= & G \ | \ \mathit{true} \ | \ a \ | \ (g,g) \end{array}$$

From these definitions we have that: (i) a goal may be considered as a term, (ii) the atom $G \lhd g$ is a special atom, called *goal binding*, which when evaluated, binds the goal variable G to the goal g, that is, the predicate $\lhd$ is defined by the clause: $G \lhd G \leftarrow$, and (iii) a goal is either a goal variable or an atom or a finite conjunction of goals, being the conjunction denoted by comma and the empty conjunction denoted by *true*.

We will also assume that conjunction of goals is associative and thus, we will feel free to omit the parentheses around goals when no confusion arises.

As usual, a clause is defined by: *head* $\leftarrow$ *body*, where *head* is an atom and *body* is a goal. We will discuss the semantics of our transformation language later in this section.

Here are the transformation rules we consider. Some more transformation rules could have been taken into account, such as the goal replacement rule or the case split rule [12, 13]. They would have allowed us to construct more powerful program transformation systems, but the following rules are indeed sufficient to illustrate the issues related to flexible continuations, which we want to discuss in this paper.

R1. Definition. It consists in adding to P a new clause, called *definition clause*, of the form:
$\mathit{newp}(V_1,\ldots,V_m) \leftarrow \mathit{def}$
where i) *newp* is a new predicate symbol, ii) *def* is a goal, and iii) $V_1,\ldots,V_m$ are variables occurring in *def*.

R2. Goal abstraction. It consists in replacing a clause in P of the form:
$p(V_1,\ldots,V_m) \leftarrow \mathit{body}[g]$
where $\mathit{body}[g]$ is a goal where we have singled out the context *body* and the occurrence of a goal g, by the new clause:
$p(V_1,\ldots,V_m) \leftarrow G \lhd g,\ \mathit{body}[G]$
where G is a new goal variable symbol.

By the properties of unification we get the following rule for replacing goal bindings. This rule can be considered as a particular case of the goal replacement rule [18].

R3. Replacement of goal bindings. An atom of the form: $G \lhd (g_1, g_2)$ for some goal variable G and some goals g_1 and g_2, can be replaced by the conjunction: $(G_2 \lhd g_2,\ G \lhd (g_1, G_2))$ where G_2 is a new goal variable symbol.

The reverse replacement is a particular case of the following unfolding rule.

Since goals may occur in argument positions, we modify the usual unfolding rule to take care of this case.

R4. Unfolding of a clause C in a program P w.r.t. an atom a of its body. Let C be of the form: $h \leftarrow \mathit{body}[a]$, where $\mathit{body}[a]$ is a goal where we have singled out the context body and the occurrence of an atom a. Let $h_1 \leftarrow g_1,\ \ldots,\ h_n \leftarrow g_n$ be the clauses of P whose heads are unifiable with a. Let $\theta_1, \ldots, \theta_n$ be the corresponding most general unifiers. By unfolding C w.r.t. a we replace C by the clauses: $(h \leftarrow \mathit{body}[g_1])\theta_1,\ \ldots,\ (h \leftarrow \mathit{body}[g_n])\theta_n$.

Notice that, in particular, the unfolding of a clause w.r.t. an atom which is *not* unifiable with any head in P, amounts to the deletion of that clause.

R5. Folding of a goal g in the body of a clause C in a program P using a definition clause D.
Suppose that:
(i) C is of the form: $h \leftarrow \mathit{body}[g]$, where we have singled out the context body and the occurrence of a goal g,
(ii) D is a definition clause of the form: $\mathit{newp}(V_1, \ldots, V_m) \leftarrow \mathit{def}$, for some goal def,
(iii) $g = \mathit{def}\,\theta$, for some substitution θ, and
(iv) for every variable V in D and not in $\{V_1, \ldots, V_m\}$ we have that:
(iv.1) $V\theta$ is a variable which does not occur in (h, body) and
(iv.2) the variable $V\theta$ does not occur in the term $W\theta$, for each variable W occurring in def and different from V.
By folding we replace C by the clause: $h \leftarrow \mathit{body}[\mathit{newp}(V_1, \ldots, V_m)\,\theta]$.

The program transformation process and the introduction into programs of our flexible continuations is performed within the transformation language using the above rules R1–R5. This language is essentially the language of definite programs together with goal variables, goal bindings, and terms which may represent goals. We assume that the given initial program is written in the *source language* of definite programs and thus, it may be viewed as written in our transformation language which includes the source

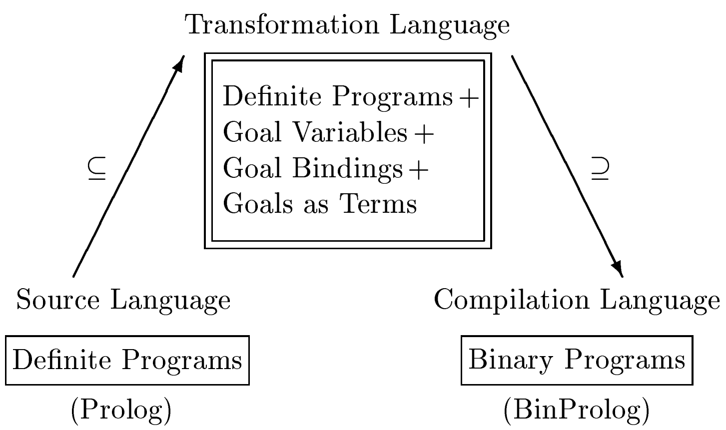

Figure 1: The three languages for logic programs with continuations. The transformation rules R1–R5 manipulate programs of the Transformation Language.

language as a sublanguage (see the up-arrow in Figure 1). The final program we want to derive, should be written in a *compilation language* which is a sublanguage of the transformation language (see the down-arrow in Figure 1), where clauses can be *binary clauses* only, that is, clauses without occurrences of goal conjunctions neither in the bodies nor in the heads (recall that goals may occur in heads of clauses because terms may represent goals). Binary clauses can be compiled using BinProlog [19] into efficient code for the so called *binary WAM* abstract machine, and then down to machine code.

The operational semantics of all these three languages is the usual SLD-resolution semantics. No real extension is needed for the operational semantics, because goal arguments are treated as terms, and goal variables when called at run-time are bound to non-variable goals.

For providing the denotational semantics of the transformation language (and the compilation language), we will resort to the usual semantics of definite programs by providing the following translation of programs written in the transformation language into definite programs (see [20] for a similar translation of binary programs into definite programs):
(i) for every predicate symbol p occurring in an argument position, that is,

in an argument of a function symbol or a predicate symbol, we replace it by a new function symbol, say f_p,
(ii.1) for every goal variable G occurring in an argument position, we replace it, consistently within each clause, by a new individual variable X,
(ii.2) for every goal variable G which does not occur in an argument position, we replace it, consistently within each clause, by the goal $demo(X)$, where X is a new variable and *demo* is a new predicate symbol defined by the clause:

$$demo(true) \leftarrow$$

together with one clause

$$demo((f_p(\ldots), X)) \leftarrow p(\ldots),\ demo(X)$$

for each predicate symbol p occurring in the program.

The concept of correctness of our transformations is is defined as follows. Consider a definite program P_1 and a program P_2 in our transformation language. We say that the transformation of P_1 into P_2 is correct iff for each atom $p(t_1, \ldots, t_m)$, where p occurs in P_1 and and no goal occurs in $t_1, \ldots, t_m$, we have that:
there exists a successful SLD-derivation of $p(t_1, \ldots, t_m)$ from P_1 iff there exists a successful SLD-derivation of $p(t_1, \ldots, t_m)$ from P_2 .

The formal proof of correctness properties of the transformations that can be realized by using our rules R1–R5 is beyond the scope of this paper. However, we believe that the results available in the literature [12] may easily be adapted to our case.

4 A Strategy for Avoiding Intermediate Data Structures during Binarization

We now turn to the problem of providing strategies that direct the application of the transformation rules so to improve program efficiency. It is impossible to define a universal strategy which improves program efficiency in all cases. Here we describe an enhancement of an existing strategy [15] for deriving binary programs without intermediate data structures. To this purpose we will apply a very useful technique which is goal generalization.

In what follows the absence of intermediate data structures is formalized as the absence of *unnecessary variables*, that is, variables which occur more than once in the body of a clause or variables which occur in the body and not in the head of a clause. Moreover, we assume that some *type information* is provided as an abstract interpretation of the program, so that we are able to identify unnecessary variables of a given type. For instance, the variable $L1$ in the body of clause 1 below, is an unnecessary variable of type list.

The presentation of our strategy will not be done in a formal way. We will describe it while presenting an example of its application. We believe that the reader, with the help of that example, which we now present, may convince himself of the generality of our approach.

Let us consider the following program which defines a predicate *gmr* such that $gmr(N, L2)$ holds iff the list $L2$ is generated from the integer N as follows. First, a list $L1$ of integers of length N is randomly generated, then its maximal element *Max* is computed, and finally the list $L2$ is constructed by replacing every element in $L1$ by 1 if it is greater or equal to $Max/2$, by 0 otherwise.

1. $gmr(N, L2) \leftarrow genlist(N, L1),\ maxlist(L1, Max),$
   $replace(Max, L1, L2)$
2. $genlist(0, [\,]) \leftarrow$
3. $genlist(N, [H|T]) \leftarrow greater(N, 0),\ minus(N, 1, N1),\ random(H),$
   $genlist(N1, T)$
4. $replace(A, [\,], [\,]) \leftarrow$
5. $replace(A, [H|T], [R|T1]) \leftarrow newelem(A, H, R),\ replace(A, T, T1)$
6. $newelem(A, H, 1) \leftarrow A \geq H/2$
7. $newelem(A, H, 0) \leftarrow A < H/2$

The predicate $minus(N, 1, N1)$ holds iff $N1$ is equal to $N-1$ and $random(H)$ randomly generates an integer and binds it to H. The predicate *maxlist* is defined by clauses 1 and 2 of Section 1.

In general, given a definite program P and an m-ary predicate p defined in P, we start off our transformation strategy by applying the definition rule and introducing a definition clause of the form:

$$p\_cont(X_1, \ldots, X_m, Cont) \leftarrow p(X_1, \ldots, X_m),\ Cont$$

Thus, in our example we start off by introducing the definition clause:

8. $gmr\_cont(N, L2, Cont) \leftarrow gmr(N, L2),\ Cont$

We then apply the unfolding rule once or more times and we derive a set of new clauses. For a fully automatic implementation of the transformation strategy, these unfolding steps should be guided by a subsidiary strategy, usually called *unfolding strategy.* Several unfolding strategies have been considered in the literature of logic program transformation (see, for instance, [5, 15]), and we refer to those papers for more details. In our case we simply perform one unfolding step and we get:

9. $gmr\_cont(N, L2, Cont) \leftarrow genlist(N, L1),\ maxlist(L1, Max),$
$\qquad replace(Max, L1, L2),\ Cont$

In the general case our strategy proceeds by transforming the clauses derived by unfolding into binary clauses where no unnecessary variables of the specified type occur. For this transformation we may need to perform folding steps using either the definition clause for p_cont introduced at the beginning of the transformation process or new definition clauses.

In our example, we have to derive binary clauses without unnecessary variables of type list and, since no folding is possible using clause 8 we introduce the following new clause:

10. $new1(N, L2, Cont) \leftarrow genlist(N, L1),\ maxlist(L1, Max),$
$\qquad replace(Max, L1, L2),\ Cont$

and by using clause 10 we may fold clause 9 and we may derive the binary clause:

11. $gmr\_cont(N, L2, Cont) \leftarrow new1(N, L2, Cont)$

Now we are left with the problem of transforming the definition clauses introduced for allowing the folding steps, because those clauses may be not binary and they may contain unnecessary variables. In our example, the clause used for folding is clause 10 and it is not binary and the unnecessary variable $L1$ of type list occurs in its body. Thus, starting from clause 10, we iterate a transformation process similar to the one we have now performed, starting from clause 8 and deriving clauses 10 and 11. We will stop iterating this transformation process consisting of unfolding, definition, and folding steps, when all clauses derived by the unfolding steps can be transformed into binary clauses without unnecessary variables by performing folding steps which use definition clauses which have been introduced in previous iterations, and thus, no new definition clauses have to be introduced.

To make the transformation process to terminate, the new definition clauses to be introduced should be chosen with care, because otherwise we run the risk of introducing an infinite number of them and the transformation process does not terminate. To avoid this problem, a generalization strategy may be useful. Indeed, generalization techniques have been widely used in program transformation because they allow for finite abstractions of infinite sets of program expressions.

Our generalization strategy uses goal generalization, that is, it may replace goals by goals variables, and it works as follows. Given two definition clauses D_1 and D_2 which are introduced during program transformation, we compute a most specific common generalization (modulo associativity and existence of neutral element), say *geng*, of the bodies of D_1 and D_2. Let the goal *geng* be of the form:

$$g_1,\ G_1,\ g_2,\ G_2,\ \ldots,\ g_n,\ G_n$$

where $g_1, \ldots, g_n$ are non-variable goals (possibly *true*) and $G_1, G_2, \ldots, G_n$ are goal variables. When computing this common generalization we may assume that the bodies of D_1 and D_2 do not share any variable, because variables in clauses are universally quantified and they may be freely renamed. Notice, however, that since goal conjunction is associative, the most specific common generalization is not necessarily unique. Notice also that, by our transformation strategy, the rightmost goal of *geng* is necessarily a goal variable (it is G_n in our case), because the last goal in the body of each new definition clause is a variable.

We have that, w.r.t. our operational semantics, *geng* is equivalent to the following goal *gengcont*:

$$K_{n-1} \lhd (g_n, G_n),\ \ldots,\ K_1 \lhd (g_2, G_2, K_2),\ g_1,\ G_1,\ K_1$$

whose intuitive meaning is: to solve the goal *gengcont* we have to solve the goal g_1 and the goal G_1 followed by the continuation K_1, *where* K_1 is the goal (g_2, G_2) followed by the continuation K_2, ..., *where* K_{n-1} is the goal g_n followed by the continuation G_n.

We then introduce a new clause *genD* of the form:

$$\begin{aligned} &genp(V_1, \ldots, V_k, J_1, K_1, \ldots, J_{n-1}, K_{n-1}, G_n) \leftarrow \\ &\qquad K_{n-1} \lhd (g_n, G_n),\ \ldots,\ K_1 \lhd (g_2, J_2),\ g_1,\ J_1 \end{aligned}$$

The body of *genD* is a generalization of *gengcont* where we have replaced the goals $(G_1, K_1), \ldots, (G_{n-1}, K_{n-1})$ by the goal variables $J_1, \ldots, J_{n-1}$, respectively. The variables $V_1, \ldots, V_k$ in the head of *genD* are distinct variables chosen as follows. Let θ_1 and θ_2 be the substitutions such that $geng\,\theta_1$ is the body of D_1 and $geng\,\theta_2$ is the body of D_2. We have that a variable V belongs to $\{V_1, \ldots, V_k\}$ iff V occurs in the goal $(g_1, g_2, \ldots, g_n)$ and, for some $i \in \{1, 2\}$, at least one of the following properties holds: 1) $V\theta_i$ is not a variable or 2) $V\theta_i$ is a variable occurring either in the head of D_i or in $W\theta_i$ for some W different from $V\theta_i$. (These technical conditions ensure that each clause that can be folded using D_1 or D_2, can also be folded using *genD*.)

When a new definition clause is introduced by goal generalization, it will contain goal bindings, and thus, during its transformation, unfolding and folding steps may be interleaved with replacements of goal bindings.

We now continue our derivation example where we will show an application of the goal generalization strategy. By unfolding steps from clause 10 we get:

12. $new1(0, [\,], Cont) \leftarrow Cont$
13. $new1(N, [E|T1], Cont) \leftarrow$
$$greater(N,0),\ minus(N,1,N1),\ random(H),\ genlist(N1,T),\ maxlist(T,MT),\ max(H,MT,Max),\ newelem(Max,H,E),\ replace(Max,T,T1), Cont$$

Clause 13 can be transformed into a binary clause without the unnecessary variable T of type list by introducing the following new definition clauses:

14. $greater\_cont(M, N, Cont) \leftarrow greater(M,N),\ Cont$
15. $minus\_cont(M, N, P, Cont) \leftarrow minus(M,N,P),\ Cont$
16. $random\_cont(H, Cont) \leftarrow random(H),\ Cont$
17. $new2(N1, H, E, T1, Cont) \leftarrow$
$$genlist(N1,T),\ maxlist(T,MT),\ max(H,MT,Max),\ newelem(Max,H,E),\ replace(Max,T,T1),\ Cont$$

Indeed, by folding clause 13 using clauses 14 through 17, we may derive the following binary clause:

18. $new1(N, [E|T1], Cont) \leftarrow$
$$greater\_cont(N, 0, minus\_cont(N, 1, N1, random\_cont(H, new2(N1, H, E, T1, Cont))))$$

The reader may notice that clause 17 is not a binary clause and it has the unnecessary variable T of type list. Now, instead of continuing our derivation by transforming clause 17, we apply the goal generalization strategy as follows. We consider clauses 10 and 17 and we compute the most specific generalization of their bodies (it is unique, in this case), which is:

$$genlist(N1,T),\ maxlist(T,MT),\ G,\ replace(Max,T,T1),\ Cont$$

Then we introduce the following new definition clause:

19. $new3(N, M1, J, K, M2, L2, Cont) \leftarrow$
$$K \lhd (replace(M2, L1, L2), Cont),\ genlist(N, L1),\ maxlist(L1, M1),\ J$$

which is an instance of clause *genD* which we have introduced above when presenting our generalization strategy. By applying goal abstraction to clause 9 and then folding using clause 19 we get:

20. $gmr\_cont(N, L2, Cont) \leftarrow new3(N, Max, G, G, Max, L2, Cont)$

By performing some unfolding steps, replacements of goal bindings, and folding steps, from clause 19 we get:

21. $new3(0, 0, J, Cont, M2, L2, Cont) \leftarrow J$
22. $new3(N, M1, J, newelem\_cont(M2,H,E,K), M2, [E|T1], Cont) \leftarrow$
$greater\_cont(N, 0, minus\_cont(N, 1, N1, random\_cont(H,$
$new3(N1, MT, max\_cont(H, MT, M1, J),$
$K, M2, T1, Cont))))$

The final program made is out of clauses 20, 21, and 22 together with the clause:

$gmr(N, L2) \leftarrow gmr\_cont(N, L2, true)$

This final program is a binary program without unnecessary variables of type list and, by using the BinProlog compiler, it runs 14 per cent faster and it uses 6 per cent less space than the initial program.

As a final remark, we notice that during program derivation it was not necessary to rearrange the order of the goals in the body of the clauses. This is an important advantage of our technique which introduces continuations in a flexible way, because goal rearrangement may, in general, affect program termination. We will return to this point in the following section when we show that by the usual unfold/fold transformations followed by binarization it is not easy to derive, from the initial program made out of clauses 1 through 7 of this section, a more efficient binary program without unnecessary variables of type list.

5 Binarization + Unfold/Fold Transformations versus Flexible Continuations

In this section we show through an example that our technique based on the extended rules presented in Section 3 and on the strategy described in Section 4, is indeed more powerful than known techniques for binarization and unfold/fold transformations.

In particular, we will show that the derivation of a binary program without unnecessary variables of type list from the initial program presented in

Section 4 is problematic if we *first* perform the binarization transformation described in Section 1 and *then* we apply usual unfold/fold rules [18], instead of our rules R1–R5.

Later in this section, we will show that it is also problematic the reverse approach of applying first the usual unfold/fold transformations to avoid unnecessary variables and then transforming the resulting program into a binary program.

Let us consider the following program obtained by binarization from the initial program which defines the predicate *gmr* presented in Section 4:

1. $gmr\_cont(N, L2, Cont) \leftarrow genlist\_cont(N, L1,$
   $maxlist\_cont(L1, Max,$
   $replace\_cont(Max, L1, L2, Cont)))$
2. $genlist\_cont(0, [\,], Cont) \leftarrow Cont$
3. $genlist\_cont(N, [H|T], Cont) \leftarrow greater\_cont(N, 0,$
   $minus\_cont(N, 1, N1,$
   $random\_cont(H,$
   $genlist\_cont(N1, T, Cont))))$
4. $maxlist\_cont([\,], 0, Cont) \leftarrow Cont$
5. $maxlist\_cont([H|T], M, Cont) \leftarrow maxlist\_cont(T, MT,$
   $max\_cont(H, MT, M, Cont))$
6. $replace\_cont(A, [\,], [\,], Cont) \leftarrow Cont$
7. $replace\_cont(A, [H|T], [R|T1], Cont) \leftarrow newelem\_cont(A, H, R,$
   $replace\_cont(A, T, T1, Cont))$

This program has the unnecessary variable $L1$ of type list which occurs in clause 1. In order to avoid this unnecessary variable we may apply the strategy presented in [15]. In our example, this strategy consists of a sequence of unfolding steps, starting from clause 1, followed by some folding steps. We encounter here a first problem, that is, unfolding w.r.t. *maxlist_cont* and *replace_cont* which occur as arguments is not allowed by the usual unfolding rule of [18].

Thus, we need to use our extended rule R4 presented in Section 3. By applying this unfolding rule from clause 1 we get:

8. $gmr\_cont(0, [\,], Cont) \leftarrow Cont$

9. $gmr\_cont(N, [R|T1], Cont) \leftarrow greater\_cont(N, 0,$
$\quad minus\_cont(N, 1, N1,$
$\quad random\_cont(H,$
$\quad genlist\_cont(N1, T,$
$\quad maxlist\_cont(T, MT,$
$\quad max\_cont(H, MT, Max,$
$\quad newelem\_cont(Max, H, R,$
$\quad replace\_cont(Max, T, T1, Cont))))))))$

Unfortunately, no folding is possible, because the body of clause 9 is not an instance of the body of clause 1.

In order to make a folding step possible and transform clause 9 into a binary clause without unnecessary variables of type list we may introduce the following new predicate:

10. $gmmnr\_cont(N1, H, R, T1, Cont) \leftarrow genlist\_cont(N1, T,$
$\quad maxlist\_cont(T, MT,$
$\quad max\_cont(H, MT, Max,$
$\quad newelem\_cont(Max, H, R,$
$\quad replace\_cont(Max, T, T1, Cont)))))$

Then, by using our folding rule R5 which allows us to fold w.r.t. goals which occur in argument positions, we may derive:

11. $gmr\_cont(N, [R|T1], Cont) \leftarrow greater\_cont(N, 0,$
$\quad minus\_cont(N, 1, N1,$
$\quad random\_cont(H,$
$\quad gmmnr\_cont(N1, H, R, T1, Cont))))$

However, in order to avoid the unnecessary variable T of type list in the body of the new definition clause 10, we have to iterate the transformation process starting from clause 10, and these iterations may not terminate.

In order to terminate the transformation process we may apply the usual generalization strategy, that is, we may introduce the following new clause whose body is the most specific common generalization of clauses 1 and 10:

12. $gen(N1, T, MT, NewCont) \leftarrow genlist\_cont(N1, T,$
$\quad maxlist\_cont(T, MT, NewCont))$

Unfortunately, even if we are able to transform clause 12 into a set of clauses without unnecessary variables of type list, the introduction of the new predicate *gen* is still not sufficient to derive a program without unnecessary variables of type list, because when expressing *gmr_cont* in terms of *gen* (by folding clause 1 using clause 12), we have:

13. $gmr\_cont(N, L2, Cont) \leftarrow gen(N, L1, Max,$
$\qquad replace\_cont(Max, L1, L2, Cont))$

where $L1$ is an unnecessary variable of type list.

One way of overcoming this difficulty is to use our goal abstraction rule of Section 3 on clauses 1 and 10 whereby obtaining the following clauses, respectively:

1a. $gmr\_cont(N, L2, Cont) \leftarrow G \lhd replace\_cont(Max, L1, L2, Cont),$
$\qquad genlist\_cont(N, L1,$
$\qquad maxlist\_cont(L1, Max, G))$

10a. $gmmnr\_cont(N1, H, R, T1, Cont) \leftarrow$
$\qquad G \lhd replace\_cont(Max, T, T1, Cont),$
$\qquad genlist\_cont(N1, T,$
$\qquad maxlist\_cont(T, MT,$
$\qquad max\_cont(H, MT, Max,$
$\qquad newelem\_cont(Max, H, R, G))))$

Then by performing a generalization step, we may introduce the following predicate:

14. $gen\_abstr(N, M1, J, K, M2, L2, Cont) \leftarrow$
$\qquad K \lhd replace\_cont(M2, L1, L2, Cont),$
$\qquad genlist\_cont(N, L1, maxlist\_cont(L1, M1, J))$

which is very similar to the predicate $new3$ introduced in Section 4 by our goal generalization strategy.

We leave it to the reader to verify that, by using our extended rules R1–R5 of Section 3 we may now derive the same program we have derived in Section 4 (except that we now use the predicate name gen_abstr instead of $new3$).

In summary, we have been able to derive from our program in binary form an equivalent one without unnecessary variables of type list, by using our extended rules of Section 3 together with a generalization strategy very similar to the goal generalization strategy proposed in Section 4. We have also shown that by a straightforward application of the usual transformation rules of [18] and the usual generalization strategy we are not able to achieve the same result.

Now we show that the reverse approach of applying *first* the usual unfold/fold transformations for definite programs to avoid intermediate data structures and *then* transforming the resulting definite program into a binary program is also problematic.

Indeed, let us consider again the initial program given in Section 4 and let us consider the following derivation. We start off from clause 1 and by unfolding we derive:

2. $gmr(0, [\,]) \leftarrow$
3. $gmr(N, [E|T1]) \leftarrow greater(N, 0),\ minus(N, 1, N1),\ random(H),$
$genlist(N1, T),\ maxlist(T, MT),$
$max(H, MT, Max),\ newelem(Max, H, E),$
$replace(Max, T, T1)$

Clause 3 has the unnecessary variable T of type list and it cannot be folded using clause 1. In order to make a folding step possible we may apply the usual generalization strategy and introduce a new clause whose body is the most specific common generalization of the bodies of clauses 1 and 3. For this generalization step we assume that conjunction is associative and commutative (otherwise we would get a goal with an occurrence of a goal variable).

By generalization we introduce the following new clause:

4. $gen\_gmr(N1, M1, M2, T1) \leftarrow genlist(N1, T),\ maxlist(T, M1),$
$replace(M2, T, T1)$

We now use the unfolding and folding rules to derive from clause 4 a set of clauses without unnecessary variables of type list. By unfolding clause 4 we derive:

5. $gen\_gmr(0, 0, M2, [\,]) \leftarrow$
6. $gen\_gmr(N, M1, M2, [E|T1]) \leftarrow$
$greater(N, 0),\ minus(N, 1, N1),\ random(H),$
$genlist(N1, T),\ maxlist(T, MT),$
$max(H, MT, M1),\ newelem(M2, H, E),$
$replace(M2, T, T1)$

Now, in order to fold clause 6 using clause 4 and avoid the unnecessary variable T of type list, we should rearrange the order of the atoms in the body of clause 6 so that the atoms $genlist(N1, T)$, $maxlist(T, MT)$, and $replace(M2, T, T1)$ become adjacent. Unfortunately, the rearrangement of goals, although it preserves the least Herbrand model semantics, it may affect termination of Prolog programs and it may generate run-time errors due to the insufficient instantiation of the arguments of some built-in predicates. Indeed, in our case, if we move the atoms $max(H, MT, M1)$, $newelem(M2, H, E)$ to the right of $replace(M2, T, T1)$, then calls to $M2 \geq H/2$ with an unbound $M2$ are generated, and also, if we move the atoms $max(H, MT, M1)$, $newelem(M2, H, E)$ to the left of $genlist(N1, T)$, $maxlist(T, MT)$, then calls to $max(H, MT, M1)$ with an unbound MT are generated. In both cases run-time errors are generated.

In contrast, as already mentioned, by using our technique based on the extended transformation rules R1–R5 and the goal generalization strategy, there is no need for rearranging goals.

6 Related Work and Conclusions

We have proposed a technique for transforming logic programs into more efficient programs in continuation passing style, also called binary programs. Our technique is based on an extension of the definite program language [9] because we allow goals to occur as arguments of functions and predicate symbols, and we allow variables to range over goals. This language, called the transformation language, includes the language of binary programs. We have introduced a set of transformation rules which extend the usual unfold/fold rules for transforming definite programs [18]. By applications of our transformation rules from an initial definite program we may derive new program versions in our transformation language, and finally we may derive a binary program. As usual in the rule-based approach to program transformation [2], the applications of the transformation rules are guided by strategies. In particular we have proposed a strategy which allows us to avoid intermediate data structures. This strategy makes use of a new generalization technique, called goal generalization, which directs the introduction of new predicates whose arguments are goal variables.

Since our transformation technique derives programs with continuations by strategies which take into account the specific form of the programs at hand, we say that we use flexible continuations.

Transformation of definite programs into binary programs, also called binarization, was proposed in [20] and an efficient compiler, called BinProlog, based on this transformation and on a specialized WAM abstract machine, is under development [19]. Some optimizations of binary programs have been proposed in [3, 4, 10]. Through an example we have argued that our approach is more powerful than the ones which apply optimization techniques after binarization. However, in principle the two approaches are complementary to one another, because our transformation technique produces binary programs which, in turn, may be optimized by further transformations.

Continuation-based transformations in the case of functional languages has been first proposed by Wand [22], and since then many papers have been written on this topic. The use of continuation-based transformations has been recognized, also in functional languages, as a valuable technique for very efficient implementation methods [1]. Apart from the difference due to the language we consider, our approach is different from Wand's approach, because we derive the new programs from the old ones by using transformation rules while Wand applies schema equivalences. Again, we believe that the use of transformation rules gives us more flexibility, although it requires a greater effort for mechanization. Moreover, we have to provide correctness proof for the transformation rules only, whereas an individual correctness proof for each schema equivalence is needed in Wand's approach.

Our transformation strategy outlined in Section 4 is aimed at improving program efficiency by deriving binary programs which also avoid intermediate data structures. Thus, it may be seen as an extension of the technique presented in [15] for eliminating the so called unnecessary variables from definite programs. An advantage of using flexible continuations over the technique presented in [15] is that we avoid to use the goal rearrangement transformation, which may affect the operational semantics when using existing Prolog compilers.

The transformational elimination of intermediate data structures via function composition, is an old idea (see, for instance, [2]). However, the design of effective transformation algorithms, such as the *deforestation algorithm* [21] and *positive supercompilation* [16], which perform this elimination in an automatic way, needs further study.

Our transformation technique also realizes what in a functional setting is called the *tupling strategy* [2, 11], that is, it may be used to avoid multiple traversals of data structures by defining new predicates which collectively perform all required computations during one pass over a given data structure. This is due to the fact that in logic programming both composition and tupling are expressed in terms of the conjunction operator, and thus, we do not need separate strategies for composition and tupling.

The goal generalization strategy presented in this paper is also related to techniques for higher-order generalization in functional programming, such as the *supercombinators* technique [6], *lambda lifting* [7], *lambda hoisting* [17], and the *lambda abstraction strategy* [14], because the introduction of goal variables is analogous to the introduction of a higher-order object. However, we insist on the fact that programs of our transformation language which allows for the use of goal variables, can be translated in a straightforward way into definite programs which are first order logic formulas.

Finally, we would like to indicate as a topic for further study, that our goal generalization strategy for the introduction of flexible continuations could also be used to increase the power of other transformation techniques, like for instance, those for program specialization presented in [5, 8, 13].

7 Acknowledgments

We thank O. Danvy, J. Gallagher, R. Glück, M. Leuschel, B. Martens, and the participants in the Dagstuhl Seminar on Partial Evaluation for many stimulating conversations. We also thank the anonymous referees for their suggestions. A transformation system based on the rules and strategies presented in this paper is currently under development at Roma Tor Vergata University by S. Renault. Special thanks go to P. Tarau for helping us in using the BinProlog system.

This work has been partly supported by the INTAS Project, Murst 40% and 60% (Italy), the Progetto Coordinato 'Programmazione Logica' of the CNR (Italy), and the MAP Research Center of Finsiel and Roma Tor Vergata University (Italy).

References

[1] A. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of Sixteenth ACM Symposium on Principles of Programming Languages (POPL'89) Austin, Texas, USA*, pages 293–302. ACM Press, 1989.

[2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[3] B. Demoen. On the transformation of a Prolog program to a more efficient binary program. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '92, Manchester, U.K.*, Workshops in Computing, pages 242–252. Springer-Verlag, 1993.

[4] B. Demoen and A. Mariën. Implementation of Prolog as binary definite programs. In A. Voronkov, editor, *Proceedings of the Russian Confer-*

ence on Logic Programming, Lecture Notes in Artificial Intelligence 592, pages 165–176. Springer-Verlag, 1992.

[5] J. P. Gallagher. Tutorial on specialization of logic programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

[6] R. J. M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proc. ACM Symposium on Lisp and Functional Programming, Pittsburgh, PA, USA*, pages 1–10. ACM Press, 1982.

[7] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *International Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, pages 190–203. Springer-Verlag, 1985.

[8] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany*, pages 319–332. MIT Press, 1996.

[9] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.

[10] U. W. Neumerkel. A transformation based on the equality between terms. In Y. Deville, editor, *Logic Program Synthesis and Transformation, Proceedings of LOPSTR '93, Louvain-la-Neuve, Belgium*, Workshops in Computing, pages 162–176. Springer-Verlag, 1994.

[11] A. Pettorossi. Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia*, pages 1–6, 1977.

[12] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[13] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic program. In *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France.* ACM Press, 1997. (To appear).

[14] A. Pettorossi and A. Skowron. Higher order generalization in program derivation. In *International Joint Conference on Theory and Practice of Software Development, TAPSOFT '87*, Lecture Notes in Computer Science 250, pages 182–196. Springer-Verlag, 1987.

[15] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[16] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 465–479. MIT Press, 1995.

[17] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[18] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

[19] P. Tarau. BinProlog 5.25. User Guide. Technical report, University of Moncton, Moncton, Canada, E1A 3E9, 1996.

[20] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Małuszynski, editors, *Proceedings PLILP '90*, pages 159–173. Springer-Verlag, 1990.

[21] P. L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[22] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

[23] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.