# A Mathematical Framework for the problem of the combination of Prolog Programs

María Vargas-Vera[1], Dave Robertson and Robert Inder

*Department of Artificial Intelligence*
*University of Edinburgh,*
*Edinburgh, Scotland, EH1 1HN.*
*email: mariav@aisb.ed.ac.uk*

**Abstract**

The purpose of this paper is to describe a mathematical framework which supports our environment for the construction of complex Prolog programs by combining simpler Prolog programs described in [?]. This framework includes three groups of properties: correctness properties which hold after applying each composition method; properties for the type of program which is obtained in each stage of the combining process and properties for the definition of the join specification. Our motivation for providing this framework is to guarantee the soundness of the composition process at the three levels ie. on our methods, on the type of combined program and properties which are useful to the user level.

*Keywords*: skeletons, techniques, extensions, join specification, composition methods, unfold/fold transformations, classification of programs, method properties, extension properties, composition properties.

## 1 Introduction

The idea of stepwise enhancement (as described by Lakhotia et. al. [LS90]) consists of developing a program by finding a suitable basic control flow (called the 'skeleton'); once the skeleton has been determined, extra computations are included by applying appropriate *techniques* to produce an extension. Separate extensions can be combined to produce the desired program. The extensions can be regarded as another skeleton permitting us to repeat the process recursively until the final program has been developed. In this context, an extension is a program derived from a skeleton by applying consistently the following modifications: renaming the predicate, adding arguments, adding subgoals, reordering arguments and adding clauses which do not change the control flow. These modifications can be achieved by applying techniques. Techniques

[1]Supported by the National University of Mexico

are defined as programming practices which allow variables to be carried to the recursive subgoals in a recursive predicate or a data structure to be built by traversing the control flow of the skeleton and by adding appropriate elements to the new data structure. A description of a set of techniques is given in [KLS89].

The combination of programs is done by first defining a join specification which can be written by the user interactively. The join specification gives information about how the two programs are to be combined. The meaning is conveyed by sharing variables between the arguments of the resulting program and the two initial programs. Therefore a join specification is a formal expression in which we define the characteristics of the new program that can be generated using two other programs. The join specification $\mathcal{T}$ is defined as follows:

$\mathcal{T} :- \mathcal{R}, \mathcal{B}$ where $\mathcal{R}$ and $\mathcal{B}$ are predicates that will be joined; they are called join operands and $\mathcal{T}$ is the join target (the resulting combined program).

An example of join specification is shown as follows:

```
count_proof(Goal,Count,Proof) :- count(Goal,Count), proof(Goal,Proof).
```

The join specification by itself can sometimes be a very inefficient way of combining programs, for instance, programs belonging to the class of *meta-interpreters* cannot be combined efficiently by using only the join specification. For this class of programs, meta-interpreters, after applying the unfolding operation with respect to a join specification we obtain clauses that are not an instance of the join specification. Therefore it is not possible to fold any subgoal. A solution to this problem is the use of extra information which is kept in the history of the program development (more details are described in [**?**]). This information is used for getting efficiency in the combined program.

The second stage in the composition process, after the join specification definition, is the selection of a method for combining a pair of programs according with the features of each program. This selection is done automatically by the *selection procedure* which is described in [**?**].

The motivation for our research arose from the idea of having a framework which guarantees the soundness of each stage in the composition process. We define in our framework three different groups of properties which need to be guaranteed. Firstly, this soundness property needs to be guaranteed for each of our composition methods. Secondly we need to guarantee that the type of combined program at each stage of the composition process remains in one of the types defined in our classification of programs. For instance given a pair of programs with type A ie. $\langle (P1, typeA), (P2, typeA) \rangle$ the combined program has the same type A ie. $\langle (P1\_P2, typeA) \rangle$ where $P1\_P2$ is the name for the combined program. This information is passed to the *selection procedure* which selects the most suitable combination method for each class of programs (see section 2.1). This selection is done using the set of rules defined in [?]. For instance, if both programs are type A then the method is: join 1-1. Thirdly we have defined properties that will be useful at user level when he/she defines the join specification.

The properties related to the first group are properties about each method for combining programs (called *method* properties). We define a method for each class of programs. These methods are based on:

- unfold/fold transformations which preserve the equivalence of programs (as was proved by Tamaki and Sato [TS83]).
- laws which are valid in an arithmetic domain (for instance commutativity in integer/real numbers)
- extra operations e.g. meta-join which is a modified version of the fold operation which uses knowledge about the program.

The properties in the second group are related to the type of the programs (denoted as *extension* properties). These properties are: reflexivity, anti-symmetry and transitivity in the relation called *extension* (denoted '$\preceq$'). Therefore the *extension* relation is a partial order relation. The properties in the partial order relation are useful for determining whether the combined program is an extension of the same skeleton as the initial programs. These properties are therefore useful in the selection of the method for combining a program. Some examples of properties belonging to this group are given below.

1. If $P$ is an extension of a skeleton $S$ then the control flow of the skeleton $S$ remains in program $P$, assuming that none of our techniques delete any subgoals which appear in the initial skeleton.
2. If each element of a given a pair of programs $< P1, P2 >$ is an extension of skeleton $S$ then $P1 \circ P2$ is an extension of skeleton $S$. We called this property the closure of the *composition* operation.

The third group of properties is related to the definition of the join specification (called *composition* properties) for instance, commutativity.

Our work is more general than that of Berzins in the sense that we can combine predicates with different numbers of arguments. Berzins gives a partial semantic rule for merging extensions for applicative programs [Ber86]. An applicative program consists of a set of recursive function definitions and an expression to be evaluated. Function definitions have a conventional form $\mathcal{F}\ (X_1, ..., X_n) = \mathcal{E}$ where $\mathcal{F}$ is the function name, each $X_i$ with $i \in \{1, \ldots, n\}$ is a parameter and $\mathcal{E}$ is a constant, variable, function application or conditional expression. Berzins limits his approach to program extensions and does not include program modification. A program extension extends the domain of the partial function without altering any of the initially defined values, while a modification redefines values that were initially defined. The merging of applicative programs can be described as follows: an applicative program consists of a set of named recursive definitions of functions and an expression. Therefore the result of merging two programs is a set containing a definition for each function appearing in either of them, where functions are matched by name. If a function appears in one of the programs but not in the other it will appear in the merged program unchanged. If a function appears in both programs with a different number of arguments a syntax error occurs. Otherwise it will appear with the formal parameters renamed (just in case it is necessary to make the formal parameters of both definitions coincide) and expressions in the bodies of the two definitions are merged to produce the body of the function in the merged program. Other researchers who also worked on formal aspects concerning the composition of two programs using program maps were Kirschenbaum et. al. [MKJ91].

This paper is structured as follows. Section 2 gives a brief discussion of the composition problem, giving a classification of pairs of programs which can be combined using our system and describing some of the properties of our combination methods. Section 3 introduces three groups of properties called method, extension and composition properties. Finally, Section 4 presents a summary of our achievements.

## 2 The Composition problem

People in several areas of computer science have been interested in the problem of merging two or more programs as a major issue of research. In the field of program transformation Burstall and Darlington built a system for transforming programs where the programs were expressed as recursion equations [BD77] and Tamaki and Sato developed unfold/fold transformations for logic programs proving that the unfold/fold transformations preserve equivalence of programs [TS83]. There are also solutions to the composition problem in procedural languages as described by Horwitz, for merging different enhancements of the same skeleton [HPR88], and Berzins gives a partial semantic rule for merging extensions for applicative programs [Ber86]. Other researchers who also worked in program composition were Sterling, Beer and Lakhotia. Their main interest was to compose meta-interpreters. Sterling and Lakhotia developed methods for combining programs such as procedural join, join 1-1 and meta-join [LS87].

The problem of combining programs together efficiently is extremely difficult in the general case. Problems arise when we try to combine programs with the same control flow and these become worse when we try to combine programs with different control flows. The partial solution which we have adopted for this is to classify Prolog programs into classes of programs and to propose a method for each class of program. Special classes of programs, for instance, meta-interpreters, mutants and trav-strav (see definition in section 2), within the composition system require extra information. This information is a history of program development which is constructed by a techniques editor (see [Rob91]). The kind of information that the combination system will require is a set of relations which give information about the skeleton used in the program development, how the subgoal in the clause of the initial skeleton was transformed in the program, plus information about extra clauses which do not appear in the initial skeleton.

## 2.1 Classification of pairs of programs

We divide the set of programs that we can write in Prolog into classes of programs according to stipulated structural features and we find methods which can be used for combining programs in each class. Our classification covers a large set of useful Prolog programs and we also show classes of programs that cannot be combined by any of our methods. The classification uses the traverse, short_traverse, search and solve skeletons (which are described in detail in [KLS89]). We classify the pairs of programs that will be combined into two types: programs with the same control flow and programs with different control flows. The programs with same control flow are programs which are extensions of the same skeleton. The programs belonging to the second type (programs with a different control) may have started by using the same skeleton but at some stage of their development extra clauses were added to the program. Alternatively they could be programs which have been built using different skeletons; for instance, a program constructed using the traverse skeleton and the second program using the short_traverse skeleton.

There are ten types of pairs of programs which are described below.

Pairs of type A are programs which come from the same skeleton with the same number of clauses, which have one condition in the recursive cases and which traverse just one data structure (for instance lists, trees).

Pairs of type B are programs which come from the same skeleton with a different number of clauses, which have more than one condition in the recursive cases and which traverse data structures of a given type.

Pairs of type C are programs which are extensions of type A or B working in an arithmetic domain.

Pairs of type D are programs which are pairs of type A or B, where the input for one program $P_2$ is the output of the other member of the pair $P_1$.

The meta-interpreters programs are those of type A built using the skeleton solve/1 defined below.

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- sys(A), call(A).
solve(A) :- clause(A,B), solve(B).
```

Pairs of type A or B which traverse different data structures are called *'different DS'*.

Mutant pairs of programs are pairs $< AB >$ (derived from pairs $< CD >$) in which the control flow is slightly modified with respect to the control flow in pair $< CD >$.

The *trav-strav* pairs of programs are those where one is constructed by using the traverse skeleton and the other is built by using the short_traverse skeleton.

The *trav-search* pairs of programs are those where one is constructed by using the traverse skeleton and the other is built by using the search skeleton.

The *strav-search* pairs of programs are those where one is constructed by using the short_traverse skeleton and the other is built by using the search skeleton.

Given the above classification of pairs of programs we then can select the most appropriate combination strategy from a library of methods. A full set is given in [VV94]. Each of our methods implemented in the composition system can be seen as a sequence of transformation operations. Thus the composition problem can be seen as a novel application of program transformation as opposed to its orthodox use in program synthesis. These methods are given to the user in an interactive system to help in the construction of a program that solves his/her application. Some of these methods are completely automatic while others are semi-automatic.

# 3 Mathematical foundations

## 3.1 Terminology and notation

We represent *extensions* with the symbol $\preceq$. An extension is a special type of enhancement restricted to add computations which do not affect the control flow of the skeleton.

$C_1 \preceq C_2$ should be read as $C_2$ *is an extension of* $C_1$ where $C_1$ and $C_2$ belong to a class of programs $C$.

$X \in Vp$ should be read as X is member of the set of variables of program P.

$X \in G_p$ should be read as X is a subgoal in program $P$.

$G_{ij}$ should be read as the ith-subgoal in the jth-clause in a program.

$P_i$ should be read as program number $i$.

$\mid P \mid$ should be read as the cardinality of a program ie. the number of clauses in program $P$.

$C_s$ is the set of clauses in program $S$.

$C_p$ is the set of clauses in program $P$.

**Definition 3.1** *$S \preceq P \longleftrightarrow$ diff (S,P) $= \emptyset \vee diff(S,P) = \bigcup_{i=1}^{n} diff(S_i, P_i)$ where $n$ is the number of clauses of $P$ and $S$.*

*$diff(S_i, P_i) = \{X \mid X \notin V_{s,i} \bigwedge X \in V_{p,i}\} \cup \{G_i \mid G_{j,i} \notin P_{s,i} \bigwedge G_{j,i} \in P_{p,i}\}$ with $j \in \{1, \ldots, m\}$*

The above definition says that program $P$ is an extension of program $S$ if there is either no difference between them (denoted by $diff$) or if the difference between program $S$ and program $P$, taken clause by clause from each program, is just variables that appears in the subgoals defined before and new subgoals which do not affect the control flow of the program.

The function $diff(S_i, P_i)$ is a function which takes as input two clauses from the pair $< S, P >$ of programs and produces the structural difference between the ith-clause in program S and the ith-clause in program P. Therefore we have two cases:

1. If $\Delta V_i \neq \emptyset$ and $\Delta G_i = \emptyset$ then we have the case that *diff*$(S_i, P_i)$= $\Delta V_i \bigcup \emptyset \equiv \Delta V_i$ where $\Delta V_i = V_{p,i} - V_{s,i}$ is the difference between the set of variables in the ith-clause of program $S$ and $P$. $\Delta G_i = G_{p,i} - G_{s,i}$ is the difference between the set of subgoals in the ith-clause of program $S$ and $P$. An example belonging to this case is obtained when the *context* technique is applied to the *traverse* skeleton. The result of applying *context* (which allows arguments to be carried to the recursive subgoals) to the skeleton *traverse* is the program which is defined below. In the resulting program just a new variable $New\_argument$ is added.

```
traverse([],New_argument).
traverse([X|Xs],New_argument) :-
     traverse(Xs,New_argument).
```

2. $\Delta V_i \neq \emptyset$ and $\Delta G_i \neq \emptyset$ then
   If *diff*$(S_i, P_i) = \Delta V_i \bigcup \Delta G_i$ *with* $i \in \{1, \ldots, n\}$. For instance when the *calculate* technique is applied to the traverse skeleton we can build a program *sum/2* which computes the sum of the elements of a list. In this program a new variable and a new subgoal are included (the variable is $Sum$ and the subgoal is $Sum\ is\ S1+X$).

```
sum([],0).
sum([X|Xs],Sum) :-
     sum(Xs,S1), Sum is S1+X.
```

Extra conditions are imposed on the extensions. These are defined as follows: Programs $P_1$ and $P_2$ are extensions of skeleton $S$ if the following conditions hold:

- $V_{p1} \cap V_{p2} = \emptyset$ where $V_{p1}$ is the set of variables in program $P_1$ and $V_{p2}$ is the set of variables in program $P_2$, otherwise renaming is required.
- $P_{p1} \cap P_{p2} = \emptyset$ where $P_{p1}$ denotes the set of predicate symbols in program $P_1$ and $P_{p2}$ is the set of predicate symbols in program $P_2$, otherwise renaming is required.

## 3.2 Method properties

We have defined our composition system as a set of methods, for instance join 1-1 and procedural join, which are defined in terms of transformation operations such as the unfold and the fold operation. We know that these operations preserve the equivalence of programs as was proved in [TS83]. Therefore the correctness for the combined program is guaranteed.

The *'meta-interpreters' method* is based on unfold and meta-fold. This meta-fold operation is an extension to the standard definition of the fold operation. Meta-fold uses knowledge about the program's development to control how the fold operation is used. The *meta-fold* operation verifies before folding that the goals that are candidates for joining (from program A and from program B) are enhancements of the same goal in the skeleton. If that is the case the goals are folded, otherwise any goal is folded. This knowledge is provided as relations, which can be defined as follows:

$\mathcal{R}\ (\alpha, \beta, \gamma)$ where $\alpha$ is the subgoal in the skeleton, $\beta$ is the transformed subgoal in the skeleton and $\gamma$ is the program name.

The correctness of the *meta-interpreters* method is defined in terms of the proof of the following theorem and on the correctness of the history of the program. We start from a program $S_0$ and a set $D$ of clauses which define new predicates in terms of predicates in $S_0$. We let $S_1 = S_0 \bigcup D$ and transform $S_1$ into $S_2$, $S_2$ into $S_3$ and so on. The correctness of our method is proved by showing that every $S_i$ with $i \in \{2, \ldots, n\}$ is equivalent to $S_1$. In [TS83] this theorem was proved by using the following two lemmas:

**Lemma 3.1** *Let $S_0$ be a program, $D$ a definition set over $S_0$ program equivalent to $S_0 \bigcup D$. If $S'$ is a program obtained by applying one of the five transformations rules to $S$ (unfolding, folding, deletion, goal merging, case splitting), then $M(S') \subset M(S)$.*

**Lemma 3.2** *If $S$ is a D-extension of $S_0$ then $M(S) \supset M(S_0 \bigcup D)$*

where a D-extension is defined as follows:

Let $S_0$ and $S$ be programs and $D$ is a definition set over $S_0$. S is called a D-extension of $S_0$ if the following two conditions hold.

1. For any single proof $T_0$ in $S_0$, there is a proof $T$ of $\Gamma(T_0)$ in $S$ which is a D-simulation of $T_0$.
2. For any proof $T_0$ in $S_0$ such that $\Gamma(T_0)$ is a D-expansion of some goal $G$, there is a proof $T$ of {G} in $S$ which is a D-simulation of $T_0$.

where $\Gamma(T)$ is the multi-set of labels of all roots of $T$ and D-simulation is defined as follows:

Let D be a definition set over a program $S_0$, $T_0$ a proof in $S_0$, G a goal and T a proof of {G} in some program S. T is called a D-simulation of $T_0$ if the following two conditions hold.

1. $\Gamma(T_0) = \{G\}$ or $\Gamma(T_0)$ is a D-expansion of $G$.
2. Let $\Sigma(T) = T_1 \bigcup ... \bigcup T_n$ where each $T_i$ is a distinct single proof. Then there is a derivation $T'_0$ and proofs $T'_1, ..., T'_n$ in $S_0$ such that
   - $T_0$ is a concatenation of $T'_0$ with $T'_1 \bigcup ... \bigcup T'_n$, and
   - each $T_i$ is a D-simulation of $T'_i$ (i=1,...,n).

A D-expansion is an instance $\Gamma\theta$ of $\Gamma$ where $\theta$ is a substitution which maps each $x_i$ to $t_i$ and internal variables of the definition to arbitrary terms.

Therefore the extra condition that needs to be guaranteed is that the history of the program is correct. This history is generated automatically by the techniques editor and we are also assuming that the programs which will be combined are correct and well defined. In this context *correct* means that the recursive procedure is guaranteed to terminate and to be well defined. By *well defined* is meant that the procedures are not under-defined or over-defined [Bun88, BGB91]. Therefore the equivalence of programs is kept.

The *'particular'* method works over a specific algebraic system. In this method the join specification is the same as the user specification. The implementation of the *particular method* was done using unfold_B which is an extended version of *unfold* used when the join specification is formed just for two operands. This method was implemented by creating a catalogue of schema transformations tagged with conditions under which it is possible for the transformation to be performed. Each schema is represented in the system as a rule. Correctness is guaranteed because the method is based on unfold_B, laws which are valid for a specific arithmetic domain and the fold operation. The set of laws is for instance, commutativity, associativity in real or integer domain.

## 3.3 Extension properties

Firstly in this subsection we describe the set of rules used in the selection of the new type for the combined program and, secondly we describe the extension properties.

### 3.3.1 Types for the combined program

We need to determine the new type for the combined program at each stage of the composition process. This assignation of type is done by using a set of rules similar to the examples given below. In these rules the types of the programs are the preconditions and the conclusion is the type for the combined program.

```
RULE 1
if
both program are of type A
then
the type for the combined program is A
RULE 2
if
both program are of type B
then
the type for the combined program is B
RULE 3
if
one of the programs is type A and the other
program is type mutant
then
the type for the combined program is mutant
```

**Lemma 3.3** $S \preceq P$ *then* $S \subseteq P$

In the lemma the relation $\subseteq$ is a special kind of subset compared with the standard definition of subset. $S \subseteq P$ states that all the variables in S appear in program $P$ and all the subgoals appearing in program $S$ appear in program $P$.

Formally: $S \subseteq P \Leftrightarrow \forall\ x, x \in V_s\ and\ \forall G_{i,j} \in S \implies x \in V_P\ and\ G_{i,j} \in P$ with $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$

The subgoals $G_i \in P$ can be found in two ways: as an enhancement (more arguments and/or different name) or as the same subgoal (same name and same number of arguments).

Let the set of variables which occur in program $P$ be $Vp = \{X_1, \ldots, X_n\}$. Let the set of the program variables in program $S$ be $Vs = \{Y1, \ldots, Ym\}$. We assume that $Vp \cap Vs = \emptyset$.

**Proof:**

Case 1. diff(S,P)= $\emptyset$

Assuming that we are taking a program with k-clauses. The case $K < 2$ it is not possible because the most simple skeleton has at least one base case and one recursive case.

when k=2 then

$\Delta V_{s,1} = \Delta V_{p,1}$ and $(\Delta G_{s,1} = \Delta G_{p,1})$ and $\Delta V_{s,2} = \Delta V_{p,2}$ and $(\Delta G_{s,2} = \Delta G_{p,2}) \implies$
$S \subseteq P$ and $P \subseteq S$
in particular $\implies S \subseteq P$.

when $k > 2$

S $\preceq$ P then diff(S,P) $= \emptyset \stackrel{by\ def}{\implies} \Delta V_{s,k} = \Delta V_{p,k}\ and\ \Delta G_{s,k} = \Delta G_{p,k}\ \forall k \in S \implies$
$S \subseteq P\ and\ P \subseteq S$

$\Longrightarrow S \subseteq P$

Case 2. diff(S,P) $\neq \emptyset$

subcase $\Delta V_{s,k} \neq \Delta V_{p,k}$ and $\Delta G_{s,k} = \Delta G_{p,k}$ and $k \in Cs$ and $C_s = C_p$
$\Longrightarrow S_k \subseteq P_k$ Therefore $S \subseteq P$

subcase $\Delta V_{s,k} \neq \Delta V_{p,k}$ and $\Delta G_{s,k} \neq \Delta G_{p,k}$

$S \preceq P \overset{by\ def}{\Longrightarrow} \bigcup_{k=1}^{n} diff(S_k, P_k)$ where

$diff(S_k, P_k) = \{X \mid X \notin V_{s,k} \bigwedge X \in V_{p,k}\} \cup \{G_i \mid G_{i,k} \notin P_{s,k} \bigwedge G_{i,k} \in P_{p,k}\}$

$\Longrightarrow$

$\exists B_1, \ldots, B_q$ new subgoals in clause $k$ with the following structure
$B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \in P$ and $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \notin S$ with $0 < j \leq q \Longrightarrow$
$S_k \subseteq S_k \cup \{B_1, \ldots, B_q\}$

We know that $S_k \cup \{B_1, \ldots, B_q\} = P_k \ \forall k \in C_s$ where $C_s$ is the set of clauses in program $S$.

Therefore

$\bigcup_{k=1}^{n} S_k \subseteq \bigcup_{k=1}^{n} (S_k \bigcup \{B_1, \ldots B_n\})$

$\Longrightarrow S \subseteq P$

**Theorem 3.1** *The relation $\preceq$ is reflexive, anti-symmetric and transitive.*

**Proof:**

**Reflexivity** : $P_1 \preceq P_1$

$P_1 \preceq P_1$ by using definition 3.1

**Symmetry** : $P_1 \preceq P_2$ then $P_2 \preceq P_1$

case $\Delta V_{p1,k} \neq \Delta V_{p2,k}$ and $\Delta G_{p1,k} = \Delta G_{p2,k}$ and $k \in Cs$ and $C_{p1} = C_{p2}$ and $P_1 \not\equiv P_2$
$\Longrightarrow P_{1,k} \subseteq P_{2,k}$

$\not\Longrightarrow P_{2,k} \subseteq P_{1,k}$

therefore

$P_2 \not\preceq P_1$

case $\Delta V_{p1,k} \neq \Delta V_{p2,k}$ and $\Delta G_{p1,k} \neq \Delta G_{p2,k}$ and $P_1 \not\equiv P_2$

$P_1 \preceq P_2 \Longrightarrow$

$P_1 \subseteq P_2$ *then* $\exists B_1, \ldots, B_q$ new subgoals in clause $k$ with the following structure: $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \in P$ and $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \notin S$ with $0 < j \leq q$

$P_{1,k} \subseteq P_{2,k}$

$\not\Longrightarrow P_{2,k} \subseteq P_{1,k}$

therefore

$P_2 \not\preceq P_1$

therefore $\preceq$ is anti-symmetric.

**Transitivity**: $P_1 \preceq P_2$ and $P_2 \preceq P_3$ then $P_1 \preceq P_3$.

$P_1 \preceq P_2 \stackrel{by\ using\ lemma\ 3.3}{\Longrightarrow} P_1 \subseteq P_2$

and

$P_2 \preceq P_3 \stackrel{by\ using\ lemma\ 3.3}{\Longrightarrow} P_2 \subseteq P_3 \Longrightarrow$

$P_1 \subseteq P_3$ by using lemma 3.4.

Therefore the relation $\preceq$ is reflexive, anti-symmetric and transitive. $\square$

**Lemma 3.4** *If* $P_1 \subseteq P_2$ *and* $P_2 \subseteq P_3$ *then* $P_1 \subseteq P_3$

**Proof:**

case $\Delta V_{p1,k} \neq \Delta V_{p2,k}$ and $\Delta G_{p1,k} = \Delta G_{p2,k}$ and $k \in Cs$ and $C_{p1} = C_{p2}$
$\Longrightarrow P_{1,k} \subseteq P_{2,k}$

On the other hand we have the case in which

$\Delta V_{p1,k} \neq \Delta V_{p3,k}$ and $\Delta G_{p2,k} \neq \Delta G_{p3,k}$ and $k \in Cs$ and $C_{p2} = C_{p3}$
$\Longrightarrow P_{2,k} \subseteq P_{3,k}$

$\Longrightarrow P_{1,k} \subseteq P_{3,k}$ by using the transitivity in sets.

$\Longrightarrow P_1 \subseteq P_3$

case $\Delta V_{p1,k} \neq \Delta V_{p2,k}$ and $\Delta G_{p1,k} \neq \Delta G_{p2,k}$

$P_1 \subseteq P_2$ *then* $\exists B_1, \ldots, B_q$ new subgoals in clause $k$ with the following structure: $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \in P$ and $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \notin S$ with $0 < j \leq q$

Therefore

$P_{1,k} \subseteq P_{2,k}$

On the other hand we have the case in which

$\Delta V_{p2,k} \neq \Delta V_{p3,k}$ and $\Delta G_{p2,k} \neq \Delta G_{p3,k}$

$P_2 \subseteq P_3$ *then* $\exists B_1, \ldots, B_q$ new subgoals in clause $k$ with the following structure: $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \in P$ and $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_r) \notin S$ with $0 < j \leq q$

$P_{2,k} \subseteq P_{3,k} \Longrightarrow P_{1,k} \subseteq P_{3,k}$

Therefore

$P_1 \subseteq P_3$

**Theorem 3.2** *$P_1$ is an extension of skeleton $S$ with type $\alpha$ and $P_2$ is an extension of skeleton $S$ with type $\alpha$ then $P_1 \circ P_2$ is an extension of $S$ of type $\alpha$ where $\alpha \in \{A, B, C, D, Mutant\}$.*

The previous theorem states that the composition (denoted by the symbol $\circ$) of two extensions of the same skeleton and with the same type $\alpha$ is another program which is an extension of the same skeleton with the same type $\alpha$. This property is important for performing further composition stages.

**Proof:**

$S \preceq P_1$ *and* $S \preceq P_2$

$\overset{by\ lemma\ 3.3}{\Longrightarrow}$

$S \subseteq P_1$ *and* $S \subseteq P_2$

case $\Delta V_{s,k} \neq \Delta V_{p,k}$ and $\Delta G_{s,k} \neq \Delta G_{p,k}$

$\Longrightarrow$

$\exists B_1, \ldots, B_q$ new subgoals in clause $k$ with the following structure $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \in P$ and $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \notin S$ with $0 < j \leq q, 0 \leq l \leq m$

$P1 = S_j \cup \{B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \mid 0 < j \leq n\} \; \forall\, j \in C_{p1}$

in analogous way we have

$P2 = S_k \cup \{B_k(X_1, \ldots, X_n, Y_1, \ldots, Y_w) \mid 0 < k \leq m\} \; \forall\, k \in C_{p2}$

therefore

$\exists B_1, \ldots, B_q$ new subgoals in clause $k$ with the following structure $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_t) \in P$ and $B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_t) \notin S$ with $0 < j \leq q$

$P1 \circ P2 = S_k \cup \{B_j(X_1, \ldots, X_n, Y_1, \ldots, Y_k) \mid 0 < j \leq q\}$

$\cup \{B_k(X_1, \ldots, X_n, Y_1, \ldots, Y_k) \mid 0 < k \leq m\}$

$\Longrightarrow S_k \subseteq P_1 \circ P_2$

$S \subseteq (P_1 \circ P_2)$

Therefore $S \preceq (P_1 \circ P_2)$.

$P_1 \circ P_2$ is an extension of S.

**Theorem 3.3** *If $Q \preceq P$ and $P \preceq Q$ then $P \equiv Q$.*

**Proof:**

by using lemma 3.3 we obtain:

$Q \subseteq P$ and $P \subseteq Q$

Therefore

$P \equiv Q$

## 3.4 Composition properties

**Theorem 3.4** *The relation $\circ$ is commutative.*

**Proof:**

Symmetry: $P_1 \circ P_2$ then $P_2 \circ P_1$

Let $JS_1$ the join specification be defined as:

$P_1\_P_2 :- P_1, P_2$

$P_1 \circ P_2$ was joined using join specification $JS_1$

If we change the join specification from $JS_1$ to $JS_2$ defined as:

$P2\_P_1 :- P_2, P_1$

Then the joined program using the join specification $JS_2$ accomplishes that:

- No new variables are introduced by the composed program due to the join specification restriction. The same variables appear in $P_1 \circ P_2$ appears in $P_2 \circ P_1$
- Goals in $G_{i,k}$ which are not operands in the join specification appearing in $P_1 \circ P_2$ are in $P_2 \circ P_1$ just in a different order.
- Goals which appear as operands in the join specification can be rewritten in the combined program. The only difference between $P_1 \circ P_2$ and $P_2 \circ P_1$ is the predicate name.

therefore

$P_1 \circ P_2 = P_2 \circ P_1$

The theorem 3.4 is important because it guarantees that the order in which the programs are combined is irrelevant (ie. $\mathcal{T}$ :– $\mathcal{P}, \mathcal{Q} \equiv \mathcal{T}$ :– $\mathcal{Q}, \mathcal{P}$).

# 4 Conclusions

We have developed a mathematical framework which guarantees the correctness of the composition process at three different levels. The first group of properties guarantees correctness of each method for combining programs. The combined program is correct because it is constructed by applying transformational operations (proofs that the unfold and fold operation preserve equivalence in programs are in [TS83]). The second group of properties guarantees that the type for the combined program is one of the types in our classification. This information helps in the selection of each method at different stages in the combination process and also in the definition of the combined program. Finally in the third group we define properties for the user level (only one example was presented in this paper). This set of properties is of particular importance in the definition of the join specification, for instance, the commutativity in the operands of the join specification. Our result is that by having this mathematical framework it is possible to guarantee that the composition process is correct at the three levels described above.

**Acknowledgements**

I would like to acknowledge to Ian Lewin, Siani Baker and Susan Bull for their comments on this paper.

# References

[BD77] Rod Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal ACM*, 24(1):44–67, 1977.

[Ber86] Valdis Berzins. On Merging Software Extensions. *Acta Informatica*, 23:607–619, 1986.

[BGB91] Alan Bundy, G. Grosse, and P. Brna. A Recursive Techniques Editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[Bun88] Alan Bundy. Proposal for a Recursive Techniques Editor for Prolog. Technical Report 394, Department of Artificial Intelligence, University of Edinburgh, 1988.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[KLS89] Marc Kirschenbaum, Arun Lakhotia, and Leon S. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Case Western Reserve University, 1989.

[LS87] Arun Lakhotia and Leon Sterling. Composing Logic Programs with Clausal Join. Tr 87-25, Computer Engineering and Science Department, Case Western Reserve University, 1987.

[LS90] Arun Lakhotia and Leon S. Sterling. Stepwise Enhancement: A Method of Incremental Programming. Tr 90-145, Case Western Reserve University, 1990.

[MKJ91] Leon S. Sterling Marc Kirschenbaum and Ashish Jain. Relating Logic Programs Via Program Maps. Tr 91-176, Case Western Reserve University, 1991.

[Rob91] Dave Robertson. A Simple Prolog Techniques Editor for Novice Users. In G. A. Wiggins, C. Mellish, and T. Duncan, editors, *3rd UK Annual Conference on Logic Programming*, pages 190–205. Springer Verlag, April 1991.

[TS83] Hisao Tamaki and Taisuke Sato. A Transformation System for Logic Programs which Preserves Equivalence. Tr 83-18, ICOT Research Center, 1983.

[VV94] Maria Vargas-Vera. *Guidance during Program Composition in a Prolog Techniques Editor*. PhD thesis, Department of Artificial Intelligence, Edinburgh University, 1994. Submitted.