Kirschenbaum, M., Sterling, L., & Jain, A. (1993). Relating logic programs via program maps.

Originally published in *Annals of Mathematics and Artificial Intelligence, 8*(3–4), 229–416.
Available from: http://dx.doi.org/10.1007/BF01530791

Relating Logic Programs Via Program Maps

Marc Kirschenbaum[1], Leon Sterling and Ashish Jain

Department of Computer Engineering and Science &
Center for Automation and Intelligent Systems Research
Case Western Reserve University
Cleveland, Ohio, 44106, USA.

Abstract

This paper presents a mathematical theory underlying a systematic method for constructing Prolog programs called *stepwise enhancement*. Stepwise enhancement dictates building a program starting with a *skeleton* program which constitutes the basic control flow for the problem to be solved, and adding extra computations to the skeleton program by using well-understood programming techniques. Each extra computation can be developed independently, and the separate enhancements combined to produce the final program. While intuitions and motivation have focussed on Prolog, the methods are applicable to logic programming languages more generally. The central concept in our mathematical theory for stepwise enhancement is that of a program map between two logic programs. Our definition of a program map from an enhancement to its skeleton guarantees the lifting of computations, the essence of the enhancement methodology. In the paper, we give definitions of program map and extensions, show that the definitions preserve the property of computations lifting, give examples of extensions and programming techniques which generate them, and point to directions for future work.

[1] Mathematics and Computer Science Department, John Carroll University, Cleveland, OH, 44118

1 Introduction

This paper presents a mathematical theory underlying a systematic method for Prolog programmers to develop their programs called *stepwise enhancement*. Stepwise enhancement dictates building a program starting with a skeleton program which constitutes the basic control flow for the problem to be solved, and adding extra computations to the skeleton program by using well-understood programming techniques. Each extra computation can be developed independently, and the separate enhancements combined to produce the final program.

The theory has arisen from attempts to construct, explain and teach complicated Prolog programs, primarily in the area of Artificial Intelligence [Sterling 89]. While intuitions and motivation have focussed on Prolog, the methods are applicable to logic programming languages more generally. In this paper the theory is described in terms of logic programs. Indeed, at a sufficient level of abstraction, we believe that stepwise enhancement is applicable to conventional languages as argued in [Lakhotia 90], but that is beyond our current scope.

The central concept in our mathematical theory for stepwise enhancement is that of a program map between two logic programs, first discussed in [Power 90]. A program map between two logic programs should assure that both programs enjoy the same basic properties. ¿From our experience using and teaching stepwise enhancement, we have determined the property that computations lift from the skeleton to the enhancement is the essence of the enhancement methodology. [Power 90] developed a elegant theory for determining when computations lift but it was difficult to prove that a particular map met all of the conditions required of it.

Motivated by examples, we decided that the lifting of refutations, a stronger property than the lifting of computations was a central concept for stepwise enhancement. In this paper we, give definitions of program map and extension, show that the definitions preserve the property of computations lifting, give examples of extensions and programming techniques which generate them, define the composition of two extensions, show that the property of refutations lifting holds for the composed program.

2 Program Maps

A logic program P in a language L_P is a quadruple $P = (V_P, F_P, P_P, C_P)$ where V_P [2] is a set of variables, F_P is a set of function symbols, P_P is a set of predicate symbols explicitly appearing in the program P, and C_P is a (finite) set of definite clauses with variable, function and predicate symbols from V_P, F_P and P_P respectively. Constants are function symbols of arity 0. $\top$ denotes the empty goal.

Determining the definition of a program map has been the central issue of this work. The properties which were lifted in the majority of examples were taken as the requirements for a program map. Although this does give the appearance of a restrictive definition, it has been found to be appropriate for our applications. The definition we settled on is a compromise between generality and practicality.

Definition 2.1: $\Pi_{S_{m,n}}$ is a *projection* function if $\exists$ a sequence $S_{m,n} = x_1, x_2, \ldots, x_n$ with $x_i \in \{1, 2, \ldots, m\}$, each x_i distinct and $m \geq n$, with

$$\Pi_{S_{m,n}}(y_1, y_2, \ldots, y_m) = (y_{x_1}, y_{x_2}, \ldots, y_{x_n}).$$

If $S_{M,m} = 1, 2, \ldots, m$, then $\Pi_{S_{m,m}}$ is called an identity projection.

Definition 2.2: h is a program map from P to Q if:

1. h is a bijection from V_P to V_Q.

2. h is a bijection from F_P to F_Q, and for any term $f(t_1, t_2, \ldots, t_m)$ in the language L_P, there exists a sequence $S_{m,n}$ such that

   $$h(f(t_1, t_2, \ldots, t_m)) = h(f)(\Pi_{S_{m,n}}(h(t_1), h(t_2), \ldots, h(t_m))), \qquad m \geq 0.$$

   If $m = 0$ then h maps constants to constants.

3. There is a bi-partition of the set P_P into $\mathcal{T}$ and $P_P \setminus \mathcal{T}$ such that:

   (a) For all the atomic formulae $\mathcal{P}$ with $p \in \mathcal{T}$ as the predicate symbol $h(\mathcal{P}) = \top$

   (b) h is a bijection from $P_P \setminus \mathcal{T}$ to P_Q which satisfies the property that $\forall p \in P_P \setminus \mathcal{T}$ $\exists$ a sequence $S_{m,n}$, where m = arity of p and n = arity of $h(p)$, such that $\forall$ atomic formula of the form $p(t_1, t_2, \ldots, t_m)$ belonging to the language L_P,
   $h(p(t_1, t_2, \ldots, t_m)) = h(p)(\Pi_{S_{m,n}}(h(t_1), h(t_2), \ldots, h(t_m)))$.

[2] The variable set $\{x_1, x_2, x_3, \ldots\}$ is countably infinite. Each of the variables explicitly appearing in the program is identified with a distinct x_i belonging to this set.

4. h maps C_P onto C_Q, where $h(H \leftarrow B_1, B_2, \ldots, B_n) =$

   $h(H) \leftarrow h(B_1), h(B_2), \ldots, h(B_n). \quad n \geq 0.$

   The mapping is not required to be bijective.

   Note: If $h(C') = C$ then by 3b above h is a bijection between the goals appearing in the body of C' which do not get mapped to $\top$ and the goals appearing in the body of C.

Definition 2.3: Program P is an *enhancement* of program Q if there exists a program map from P to Q.

Examples:

```
List::                      Length::
 list([ ]).                  length([ ], 0).
 list([X|Xs]):-              length([ ], 1).
    list(Xs).                length([X|Xs], N1):-
                                 length(Xs, N), plus1(N1,N).

   plus1(X,Y) is true if X equals Y + 1.
```

Define $h : Length \rightarrow List$ by

1. Let $V_{List} = V_{Length}$, and $h(x) = x, \forall x \in V_{Length}$,
2. Let $F_{List} = F_{Length}$, and $h(f) = f, \forall f \in F_{Length}$ with identity projection and
3. $\mathcal{T} = \{plus1\}$ and $h(length) = list$ with sequence $S_{2,1} = 1$.

h defined above satisfies properties 1-5 above and thus is a program map.

In the above example, it is true that despite the extra base case there exists a program map between *Length* and *List*. However, if another base case `length([X],1)` is introduced then there isn't a mapping as the base case `length([X],1)` cannot be mapped to any clause in *List*.

```
Lista::                 Listb::
 p([ ]).                 q([ ]).
 p([a|Xs]) :- p(Xs).      q([b|Xs]) :- q(Xs).
```

There is no program map between *Lista* and *List*. Since function symbols are only mapped to function symbols, there isn't an onto map from C_{Lista} to C_{List}. Similarly, there is no map between *List* and *Lista*. There does exist a program map from *Lista* to *Listb*.

1. Let $V_{Lista} = V_{List}$, and $h(x) = x, \forall x \in V_{Lista}$,

2. Let $F_{Lista} = F_{List}$, and $\{h(b) = a, h(a) = b\}$, and $\forall f \in F_{Lista} \setminus \{a, b\}$ h(f)=f with the identity projection.

3. $\mathcal{T} = \emptyset$ and $h(p) = q$ with the identity projection.

Definition 2.4: A *computation*[3] of a program P with goal G, where G is an atomic formula in the language L_P, consists of a sequence $S_0, S_1, \ldots, S_n, \ldots$ where S_i denotes the tuple $(A_i, C_i, \sigma_i, R_i)$ where A_i, C_i, and σ_i are defined recursively as follows:

1. $S_0 = (\top, \top, \{\}, G)$.

2. For $i > 0$,

   (a) $A_i \in R_{i-1}$.

   (b) $C_i = C\vartheta$. $C \in C_P$ and ϑ is a variable substitution $\{x_1/y_1, x_2/y_2, \ldots, x_n/y_n\}$ where $\{x_i \mid 0 \leq i \leq n\}$, are *all* the distinct variables appearing in the clause C, and $\{y_i \mid 0 \leq i \leq n\}$ is a set of new variables none of which appear in any $S_k, 0 \leq k \leq i-1$.

   (c) Let $C_i \equiv H \leftarrow B_1, B_2, \ldots, B_n.$, where A_i and H are unifiable then $\sigma_i = mgu(A_i, H)$.

   (d) $R_i = \{\{R_{i-1} \setminus A_i\} \bigcup^4 \{B_1, B_2, \ldots, B_n\}\}\sigma_i$.

The computation $S_0, S_1, \ldots, S_n$ is defined to have length n.

Definition 2.5: A computation $S_0, S_1, \ldots, S_n$ *extends* a computation $T_0, T_1, \ldots, T_m$ if $n \geq m$ and $S_i = T_i, 0 \leq i \leq m$.

Note that it only makes sense talking about extending computations of the same program.

Definition 2.6: A refutation (in program P with goal G) is a computation (in program P with goal G) of length n with $R_n = \top$.

Definition 2.7: Let h be a program map from program P to program Q. A computation $W' = S'_0, S'_1, \ldots, S'_n$ in P is over the computation $W = S_0, S_1, \ldots, S_n$ in Q if $\forall i, 1 \leq i \leq n, h(S'_i) = S_i$, where $h(S'_i) = S_i$ iff

1. $h(A'_i) = A_i$.

[3] We prefer this term to the more usual term *SLD-derivation* used in [Lloyd 84].

[4] Union for multi-sets

2. $h(C_i') = C_i$.

3. $\Pi_{S_{m,n}} h(\sigma_i') = \sigma_i$[5] where $S_{m,n}$ is the sequence associated with the predicate appearing in the head of the clause C_i.

4. $h(R_i') = R_i$ [6].

The computation W is said to lift to W' if W' is over W.

The success or failure of satisfying property three depends upon how logic programs are represented. For example:

```
P::                                  Q::
 p(s(0),0).                           q(s(0)).
 p(s(s(X)),s(X)):- p(s(X),X).         q(s(s(X))):- q(s(X)).
```

Define

1. $h(x) = x$ for all $x \in V_P$,

2. $h(f) = f$ with the identity projection for all $f \in F_P$, and

3. $\mathcal{T} = \emptyset$ and $h(p) = q$ with sequence $S_{2,1} = 1$.

A computation for the goal $q(s(s(X)))$ is

$$
\begin{aligned}
S_0 &= (\top, \top, \{\}, q(s(s(X)))) \\
S_1 &= (q(s(s(X))), q(s(s(X_1))) : -q(s(X_1)), \{X/X_1\}, q(s(X_1))) \\
S_2 &= (q(s(X_1)), q(s(0)), \{X_1/0\}, \{\})
\end{aligned}
$$

$q(s(s(X)))$ lifts to $p(s(s(X)), s(X))$ with the computation:

$$
\begin{aligned}
T_0 &= (\top, \top, \{\}, p(s(s(X)), s(X))) \\
T_1 &= (p(s(s(X)), s(X)), p(s(s(X_1)), s(X_1)) : -p(s(X_1), X_1), \{X/X_1\}, p(s(X_1)), X_1) \\
T_2 &= (p(s(X_1), X_1), p(s(0), 0), \{X_1/0\}, \{\})
\end{aligned}
$$

Unfortunately, projecting out all substitutions from the second argument yields $\Pi_{S_{2,1}}(\{X_1/0\}) = \{\}$ instead of $\{X_1/0\}$. We want the computation from program P to be over the computation from program Q. One method to guarantee that the wrong substitution does not get projected out is to

[5] If $\sigma = \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ is any substitution then $h(\sigma_i) = \{h(x_1)/h(t_1), h(x_2)/h(t_2), \ldots, h(x_n)/h(t_n)\}$.

[6] If $\mathcal{P}$ is any set of logical expressions then $h(\mathcal{P}) = \{h(\mathcal{A}) \mid \text{for all A} \in \mathcal{P}\}$.

require all unifications to be explicit in the body of the clauses. Hence, every variable in the head of a clause will be unique. This does introduce another difficulty. Look at the following example:

```
Len::                                   List::
len(X,N):- X = [ ], N = 0.              list(X):- X = [ ].
len(Y,N):-                              list(Y):-
  Y = [X|Xs], len(Xs,M), plus1(N,M).      Y = [X|Xs], list(Xs).
```

It is necessary for both $h(X = [\,]) = (X = [\,])$ and $h(N = 0) = \top$ to be true an obvious inconsistency. Hence there does not exist a program map as defined above. In order to be able to distinguish between different calls of the same goal, each predicate, other than the defining predicate, appearing in the program will have a subscript attached to it. The exact ordering isn't important. The ordering chosen is from left to right and top to bottom. This makes it possible to have a consistent definition for a program map h that also satisfies property 3 above. Another way to overcome this problem would be to treat constraints differently than predicates. We believe restricting ourselves to idempotent substitutions can also overcome the inconsistency problem.

Definition 2.8: A logic program is in *acceptable* form if

1. All arguments at the head of each clause are distinct.
2. Predicates other than the defining predicate appears only once as a goal in the body of the program.

Logic Programs will be assumed to be in acceptable form for the remainder of this paper. Below is an example of a program in acceptable form.

List::
list(X):- eq_1(X,[ ]).
list(Y):-
  eq_2(Y,[X|Xs]),list(Xs).

eq_1(X,Y) :- X = Y.
eq_2(X,Y) :- X = Y.

Len::
len(X,N):- eq_3(X,[ ]), eq_4(N,0).
len(Y,N):-
  eq_5(Y,[X|Xs]),len(Xs,M),plus(N, M).

eq_3(X,Y) :- X = Y.
eq_4(X,Y) :- N is Y.
eq_5(X,Y) :- X = Y.

Lemma 2.9: If h is a program map from Program P to Program Q then $\forall$ atomic formulae $\mathcal{A}$ and substitution σ in the language L_P, $h(\mathcal{A}\sigma) = h(\mathcal{A})h(\sigma)$.

Proof: Let $\sigma = \{\frac{x_1}{t_1}, \frac{x_2}{t_2}, \ldots, \frac{x_m}{t_m}\}$. Then $h(p(x_1, x_2, \ldots, x_m)\sigma) =$

$$\begin{aligned}
&h(p(t_1, t_2, \ldots, t_m)) = \\
&h(p)(\Pi_{S_{m,n}}(h(t_1, t_2, \ldots, t_n))) = \\
&h(p)(\Pi_{S_{m,n}}(h(x_1), h(x_2), \ldots, h(x_m))\{\tfrac{h(x_1)}{h(t_1)}, \tfrac{h(x_2)}{h(t_2)}, \ldots, \tfrac{h(x_m)}{h(t_m)}\} = \\
&h(p)(\Pi_{S_{m,n}}(h(x_1), h(x_2), \ldots, h(x_m))h(\sigma) = \\
&h(p(x_1, x_2, \ldots, x_m))h(\sigma). \blacksquare
\end{aligned}$$

Lemma 2.10: If h is a program map from Program P to program Q then for any atomic formula $\mathcal{A} \in L_Q$ there exists an atomic formula $\mathcal{A}' \in L_P$ such that $h(\mathcal{A}') = \mathcal{A}$.

Proof: Let $\mathcal{A} \in L_Q$ be of the form $p(t_1, t_2, \ldots, t_n)$, $n \geq 1$. By definition of h, $\exists p' \in L_P$ such that $h(p') = p$ and $\exists$ a sequence $S_{m,n} = x_1, x_2, \ldots, x_n$ for this p. If $t_i \in V_Q$ then since h is a bijection from V_P to V_Q, $\exists t'_i \in V_P$ with $h(t'_i) = t_i$. If $t_i \in F_Q$ it is of the form $g(x_1, x_2, \ldots, x_r)$ and since h is a bijection from F_P to F_Q, $\exists t'_i \in F_P$ along with its projection map so that $h(t'_i) = g(x_1, x_2, \ldots, x_r)$. Construct $p'(y_1, y_2, \ldots, y_m)$ as follows:
$y_{x_i} = t'_i, \forall i, 1 \leq i \leq n$ and y_j is a new uninstantiated variable $\forall j \notin S_{m,n}, 1 \leq j \leq m$. This implies

$$\begin{aligned}
h(p'(y_1, y_2, \ldots, y_m)) &= h(p')(\Pi_{S_{m,n}}(h(y_1), h(y_2), \ldots, h(y_m))) \\
&= p(h(y_{x_1}), h(y_{x_2}), \ldots, h(y_{x_n})) \\
&= p(t_1, t_2, \ldots, t_n). \blacksquare
\end{aligned}$$

Theorem 2.11: If h is a program map from Program P to Program Q then for any computation W for a goal $G \in L_Q$ there exists a computation W' for a goal $G' \in L_P$ with $h(G') = G$ which is over W.

Proof: The proof proceeds by induction on the length of the computation. It will also be proven that the construction presented here has the following invariant:

$\forall i, 0 \leq i \leq n$, there is a *bijection* between the goals represented by the multi-set R'_i (excluding those which get mapped to $\top$) and goals represented by the multi-set R_i. This invariant property will be called *resolvent bijection*.

1. Base case: $n = 0$.
   Let $W = S_0 = (\top, \top, \{\}, G)$. From lemma 2.10 $\exists B$ such that $h(B) = G$. Let G' be the most general such goal. Define $W' = S'_0 = (\top, \top, \{\}, G')$. Since $R'_0 = G'$ and $R_0 = G$ it is true that W' is over W and the *Resolvent bijection* holds.

2. $n > 0$.

   Assume all computations (for any goal G) of length k in Q lift to a computation in P. Also assume that *resolvent bijection* holds.

   Let $W = S_0, S_1, \ldots, S_k, S_{k+1}$ be a computation (for any goal G) of length $k+1$ in Q. By the induction hypothesis there exist a computation. $W' = S'_0, S'_1, \ldots, S'_k$ in P such that $\forall i, 0 \leq i \leq k, h(S'_i) = S_i$. *Lift* S_{k+1} to S'_{k+1} by the following construction:

   (a) Construction for A'_{k+1}.
   By definition of A_{k+1} and the inductive hypothesis $A_{k+1} \in R_k = h(R'_k)$. Therefore, $\exists A'_{k+1} \in R'_k$ such that $h(A'_{k+1}) = A_{k+1}$.

   (b) Construction for C'_{k+1}.
   By definition, there exists $C \in C_Q$ and a variable substitution $\vartheta \equiv \{x_1/y_1, x_2/y_2, \ldots, x_m/y_m\}$ such that $C_{k+1} = C\vartheta$. $\{x_i \mid 1 \leq i \leq m\}$ are *all* the distinct variables appearing in the clause C. $\{y_i \mid 1 \leq i \leq m\}$ is a set of new variables none of which appear in any S_j, and $\{h^{-1}(y_i) \mid 1 \leq i \leq m\}$ is a set of variables none of which appear in any $S'_j, 0 \leq j \leq k$. Since h maps C_P onto C_Q, $\exists C' \in C_P$ such that $h(C') = C$. Let $C'_{k+1} = C'\vartheta'$ where $\vartheta' = \{h^{-1}(x_i)/h^{-1}(y_i) \mid 1 \leq i \leq m\}$[7] $\cup$ $\{v_i/w_i \mid 1 \leq i \leq n\}$. $\{v_i \mid 1 \leq i \leq n\}$ denotes the set of variables appearing in the clause C' such that each and every occurrence of any variable belonging to this set in clause C' gets projected out in $h(C')$. $\{w_i \mid 1 \leq i \leq n\}$ is a set of new variables none of which appear in any $S'_i, 0 \leq i \leq k$. The following calculation proves that the constructed clause C'_{k+1} lies over C_{k+1}. $h(C'_{k+1}) = h(C'\vartheta') = h(C')h(\vartheta')$[8] $= C\{\vartheta \cup \{h(v_i)/h(w_i) \mid 1 \leq i \leq n\}\} = C\vartheta$[9] $= C_{k+1}$.

   (c) Construction for σ'_{k+1}. Let $C_{k+1} \equiv H \leftarrow B_1, B_2, \ldots, B_n$, $C'_{k+1} \equiv H' \leftarrow B'_1, B'_2, \ldots, B'_m$ with $m \geq n$, $\sigma_{k+1} = mgu(H, A_{k+1})$, and $\sigma'_{k+1} = mgu(H', A'_{k+1})$. Since, the heads of the clauses are in acceptable form, $h(H') = H$, and $h(A'_{k+1}) = A_{k+1}$ it follows that σ'_{k+1} would be a permutation of $\{h^{-1}(x_i)/h^{-1}(t_i) \mid 1 \leq i \leq n\} \cup \{y_i/t'_i \mid 1 \leq i \leq w\}$ where $\sigma_{k+1} = \{x_i/t_i \mid 1 \leq i \leq n\}$. The variables $y_i, 1 \leq i \leq m$, are the variables appearing in H'

[7] $h^{-1}(\vartheta)$
[8] follows from lemma 1.
[9] Since none of $h(v_i), 1 \leq i \leq n$ appear in C_{k+1}.

which get projected out according to the sequence $S_{m,n}$ associated with the predicate symbol p appearing in H'. This must be true for if $\{y_i/w_i\} \in \sigma'_{k+1}$ with h not projecting y_i out then since $\{h(y_i)/h(w_i)\} \notin \sigma_{k+1}$ it follows that
$h(A'_{k+1}(\sigma'_{k+1} \setminus \{y_i/w_i\})) = A_{k+1}(h(\sigma'_{k+1}) \setminus \{h(y_i)/h(w_i)\})) =$
$H(h(\sigma'_{k+1}) \setminus \{h(y_i)/h(w_i)\})) = h(H'(\sigma'_{k+1} \setminus \{y_i/w_i\}))$.
$\sigma'_{k+1} \setminus \{y_i/w_i\}$ must unify A'_{k+1} with H' since y_i is not projected out and h is a bijection on V_p. This contradicts $\sigma'_{k+1} = mgu(H', A'_{k+1})$. Thus, $\Pi_{S_{m,n}} h(\sigma'_{k+1}) = \sigma_{k+1}$.

(d) Construction for R'_{k+1}.
$R'_{k+1} = \{\{R'_k \setminus A'_{k+1}\} \bigcup \{B'_1, B'_2, \ldots, B'_m\}\}\sigma'_{k+1}$. By the Induction hypothesis the *resolvent bijection* hold for R'_k and R_k. Since $h(A'_{k+1}) = A_{k+1}$ and $h(C'_{k+1}) = C_{k+1}$ it is only necessary to show that $h(R'_k \setminus A'_{k+1}) = R_k \setminus A_{k+1}$. The bijection from the *resolvent bijection* for $n = k$ gives the required bijection from $R'_k \setminus A'_{k+1}$ to $R_k \setminus A_{k+1}$. ■

Hence $h(S'_{k+1}) = S_{k+1}$.

3 Extensions

Definition 3.1: A program P is an *extension* of a program Q if there is a program map from P to Q and for every refutation U in Q, there is a computation V and refutation W in P such that U lifts to V and W extends V.

¿From Theorem 2.11, every refutation lifts to a computation. The extra condition here is that there is a refutation that extends the computation.

Let us reconsider the acceptable programs Len and $List$ on page 6. Define h as follows:

1. Let $V_{Len} = V_{List}$, and $h(x) = x, \forall x \in V_{Len}$,
2. $h(f) = f$ for all $f \in F_{Len}$,
3. $\mathcal{T} = \{eq_4\}$, $h(len) = list$ with the sequence $S = 1$, $h(eq_3) = eq_1$ with the sequence $S = 1$ and $h(eq_5) = eq_2$ with the sequence $S = 1$.

h is a program map.

To show that Len is an extension of $List$ it is necessary to prove that every refutation lifts to a computation that can be extended to a refutation. For this simple example, an explicit construction can be given. A refutation of a goal *list(Xs)* where *Xs* has length n in program $List$ can be lifted to a computation where R'_n is the conjunctive goal, $eq_4(N_0, 0), plus1(N_1, N_0), \ldots, plus1(N_n, N_{n-1})$. Reducing them from left to right gives a refutation.

Being an extension is different from being an enhancement. All extensions are enhancements but the converse is not true.

Example: Redefine Len as:

```
Len1::
 len1(X,N):- eq1(X,[ ]), eq2(X,[1]), eq2(N,0).
 len1(Y,N):- eq1(Y, [X|Xs]), len1(Xs,M), plus1(N, M).
```

Extend the program map $h : Len \rightarrow List$ to include $\{eq2, plus1\}$ in $\mathcal{T}$. h is still a program map, but not an extension as none of the refutations in program $List$ lift to a computation in program $Len1$ which can be extended to a refutation.

Proposition 3.2: If P is an extension of Q and Q is an extension of R,then P is an extension of R.

Proof:

This proposition follows directly from the definition of extensions. ■

In general, what can be said about extensions? It is undecidable whether an arbitrary goal will succeed, and therefore it is undecidable whether a given program is an extension of another program. However, the definition allows us to designate classes of programs as extensions. This is the intent behind programming techniques to be discussed in the next section.

4 Techniques

Intuitively, a technique is a standard programming practice. ¿From a theoretical perspective a technique can be viewed as a method for selecting an extension of a particular skeleton. Unfortunately, a standard technique like summing numbers, will be syntactically different for different data structures, such as lists and trees. Thus, techniques depend upon the program to be enhanced. How to represent techniques is an important issue which

is currently under study. A description of one family of techniques called *calculate* will now be presented.

Calculate: One extra argument, $FinalVal$ say, is added to the predicate to be extended.

1. To each non recursive clause add an initialization predicate of the form eq_2(X,Y).

2. An extra argument say Val is added to the defining predicate in the body of a recursive clause to be extended. An extra arithmetic goal relating $FinalVal$ and Val is added at the end of the body of the clause.

As an example, Len is the *calculate* technique applied to $List$ with $eq_4(X, 0)$ as the initialization predicate, and the relation between $FinalVal$ and Val being $FinalVal\ is\ Val + 1$.

The argument showing that Len is an extension of $List$ can be generalized to show that any instance of a *calculate* technique where the arithmetic goal always succeeds if Val is a number will always generate an extension.

5 Program Equivalence

In this section the notion of program equivalence is defined. The logic programs P and Q are equivalent if P is an enhancement of Q and Q is an enhancement of P.

Theorem 5.1 Any two representatives P and Q from the same equivalence class, are extensions of each other.

Proof Let h_{pq} be the program map from P to Q and h_{qp} be the program map for Q to P.

1. P_P contains only the predicates explicitly appearing in P and Q_P contains only the predicates explicitly appearing in Q. By the definition of h_{pq}, there must be at least as many elements in P_P as there are in Q_P. Similarly, by the definition of h_{qp}, there must be at least as many elements in Q_P as there are in P_P. Hence the program maps are bijections between the predicate symbols.

2. Let $S_0, S_1, \ldots, S_n$ be any refutation in Q. h_{pq} is a program map hence the refutation lifts to a computation $S'_0, S'_1, \ldots, S'_n$. The resolvent

bijection states that h_{pq} is a bijection from the set R'_n (excluding those which get mapped to $\top$) to the set R_n. Since h_{pq} is a bijection from P_P onto Q_P it follows that h_{pq} is a bijection from R'_n to R_n. Hence $R'_n = \top$, and $S'_0, S'_1, \ldots, S'_n$ is in fact a refutation. Hence P is an extension of Q.

3. A similar argument shows that Q is an extension of P. ■

Intuitively, two programs are equivalent if you can get one program from the other by just renaming the predicate and function symbols and/or permuting the arguments. Note: $\forall q \in P_Q \quad \exists p \in P_P$ with $h(p) = q$ and a sequence $S_{m,n}$. It is possible rearrange the arguments in p to produce the predicate p' with the property that $h(p') = q$ and $S_{m,n} = (1, 2, \ldots, n)$. By the above it is clear that replacing p by p' produces an equivalent program. An enhancement where all sequences $S_{m,n} = (1, 2, \ldots, n)$ is said to be in standard form.

6 Composing Extensions

In logic programs the scope of a variable is the clause in which it appears. It is implicit in the semantics of the logic programs that the same variable name appearing in two different clauses stand for different entities, and a variable name appearing more than once in the same clause stand for the same entity. Making this scoping explicit is an implementation detail (e.g Prolog achieves this by using place holders like _365). However, composing two clauses is a meta-operation which brings up the issue of the variable naming used across the clauses. We take a conservative approach here by requiring the variable, function, and predicate names used in P to be distinct from those used in Q. If this requires renaming, the new program is equivalent to the original program. Hence, for this section the following is assumed: Program P and Q are extensions of program R in standard form. $V_P \cap V_Q = \emptyset$, $F_P \cap F_Q = \emptyset$, $P_P \cap P_Q = \emptyset$, and h_{pr}, h_{qr} denote the program map from P to R, and from Q to R respectively. The composite of P and Q is denoted by $P \times Q/R$. In order to define the composition of two extensions it is necessary to create a language for the composed program. The variables, function symbols, and the predicate symbols are created as follows:

1. Construction for $V_{P \times Q/R}$
   Define $V_{P \times Q/R} = \{A\_B \mid A = h_{pr}^{-1}(X), B = h_{qr}^{-1}(X) \quad \forall X \in V_R\}$

2. Construction for $F_{P \times Q/R}$
   Let $h_{pr}(f_p) = f$ and $h_{qr}(f_q) = f$ both with sequence $1, 2, \ldots, a$ where a is the arity of f. W.O.L.G assume m is the arity of f_p and n is the arity of f_q. The *merge* operation is defined by

   $f_p(X_1, X_2, \ldots, X_m) \bullet f_q(Y_1, Y_2, \ldots, Y_n) =$

   $f_p\_f_q$[10]$(X_1\_Y_1, X_2\_Y_2, \ldots, X_a\_Y_a, Z_{a+1}, \ldots, Z_m, W_{a+1}, \ldots, W_n)$

   Where

   $Z_i = X_i\_h_{qr}^{-1}(h_{pr}(X_i))$ and $W_i = Y_i\_h_{pr}^{-1}(h_{qr}(Y_i))$.

   Define $F_{P \times Q/R} = \{h_{pr}^{-1}(f) \bullet h_{qr}^{-1}(f) \mid \quad \forall f \in F_R\}$

3. Construction for $P_{P \times Q/R}$
   The construction is similar to $F_{P \times Q/R}$ except for the predicates which get mapped to $\top$.

   Define $P_{P \times Q/R} =$
   $\{h_{pr}^{-1}(r) \bullet h_{qr}^{-1}(r) \mid r \in P_R\} \bigcup \{p \mid h_{pr}(p) = \top\} \bigcup \{q \mid h_{qr}(q) = \top\}$.

Definition 6.1: Given clauses $C_1 \equiv H_1 \leftarrow B_1 \in L_P$, $C_2 \equiv H_2 \leftarrow B_2 \in L_Q$, and $C_3 \equiv H_3 \leftarrow B_3 \in L_R$ such that $h_{pr}(C_1) = h_{qr}(C_2) = C_3$. The *quotient* clause denoted by $C_1 \circ C_2/C_3$, is the clause $H \leftarrow B$ constructed as follow:

1. $H = H_1 \bullet H_2$.

2. Let $T_1 \quad = \{g_1 \bullet g_2 \mid \exists g \in B_3$ such that $h_{pr}(g_1) = g$ and $h_{qr}(g_2) = g\}$.
   Let $T_2 \quad = \{p(\alpha_1, \alpha_2, \ldots, \alpha_w) \mid p(X_1, X_2, \ldots, X_w) \in B_1, h_{pr}(p) = \top$
   and $\alpha_i = X_i\_h_{qr}^{-1}(h_{pr}(X_i))\}$
   Let $T_3 \quad = \{q(\beta_1, \beta_2, \ldots, \beta_v) \mid q(Y_1, Y_2, \ldots, Y_v) \in B_2, h_{qr}(q) = \top$
   and $\beta_i = Y_i\_h_{pr}^{-1}(h_{qr}(Y_i))\}$
   Define $B = T_1 \cup T_2 \cup T_3$

In order to define the *composition* operation for extensions it is necessary to require a bijection between the clauses of extensions and the clauses of the skeleton program. Note in the definition of program map it was only required that the mapping be onto.

Definition 6.2: The composition $P \times Q/R$ is a program whose set of clauses is given by $\{h_{pr}^{-1}(C) \circ h_{qr}^{-1}(C)/C \mid C \in C_R\}$.

[10] The function name $f_p\_f_q$ is assumed to be previously unused.

Theorem 6.3: $P \times Q/R$ is an enhancement of P.

Proof: In order to prove this result it is necessary to construct a program map, h_{pq}, from $P \times Q/R$ to program P.

1. By the construction of $V_{P \times Q/R}$, $\forall X \in V_R$ there corresponds exactly one variable $h_{pr}^{-1}(X) \in V_P$ and exactly one variable $h_{pr}^{-1}(X)\_h_{qr}^{-1}(X) \in V_{P \times Q/R}$. Define $h_{pq}(h_{pr}^{-1}(X)\_h_{qr}^{-1}(X)) = h_{pr}^{-1}(X) \quad \forall X \in V_R$. h is a bijection from $V_{P \times Q/R}$ to V_P.

2. By the construction of $F_{P \times Q/R}$, $\forall f \in F_R$ with arity say a there corresponds exactly one function $h_{pr}^{-1}(f) \in F_P$ with arity say m and exactly one new function $h_{pr}^{-1}(f)\_h_{qr}^{-1}(f) \in F_{P \times Q/R}$ with arity say n. Define $h_{pq}(h_{pr}^{-1}(f)\_h_{qr}^{-1}(f)) = h_{pr}^{-1}(f), \forall f \in F_R$ with sequence $1, 2, \ldots, m$.

   h is a bijection from $F_{P \times Q/R}$ to F_P.

3. By the construction of $P_{P \times Q/R}$, $\forall r \in P_R$ with arity say a there corresponds exactly one predicate $h_{pr}^{-1}(r) \in P_P$ with arity say m and exactly one new predicate $h_{pr}^{-1}(f)\_h_{qr}^{-1}(f) \in P_{P \times Q/R}$ with arity say n. Define

   $$\begin{aligned} h_{pq}(h_{pr}^{-1}(r)\_h_{qr}^{-1}(r)) &= h_{pr}^{-1}(r), \forall r \in P_R \text{ with sequence } 1, 2, \ldots, m. \\ h_{pq}(p) &= p \text{ with the identity sequence, } \forall p \text{ with } h_{pr}(p) = \top. \end{aligned}$$

   Define $\mathcal{T} = \{q \mid h_{qr}(q) = \top\}$ then h is a bijection from $P_{P \times Q/R} \setminus \mathcal{T}$ to P_P which satisfies condition 3b.

4. Define $h_{pq}(\top) = \top$.

5. By our construction it is clear that h_{pq} maps $C_{P \times Q/R}$ onto C_P.

Hence h_{pq} defined above satisfies the properties for being a program map from $P \times Q/R$ to P. ■

Lemma 6.4: $P \times Q/R$ is an enhancement of Q.

Proof: An argument similar to the above proves that $Q \times P/R$ is an enhancement of Q. The only differences between $Q \times P/R$ and $P \times Q/R$ is the naming of the predicate and function symbols and the argument order in the composition. Hence, $Q \times P/R$ is equivalent to $P \times Q/R$ proving $P \times Q/R$ is an enhancement of Q. ■

The following is due to the transitivity of enhancement.

Corollary 6.5: $P \times Q/R$ is an enhancement of R.

Theorem 6.6: If P is an extension of R and Q is an extension of R then $P \times Q/R$ is an extension of R.

Proof: By theorem 2.11, every refutation in R lifts to a computation in $P \times Q/R$. It remain to prove that the computation can be lifted to a refutation. By the resolvent bijection, the resolvent from the lifted refutation contains only those goals which get mapped to $\top$. Those goals come from the program P which get mapped to $\top$ by h_{pr} and from the program Q which get mapped to $\top$ by h_{qr}. Since P is an extension of R and h_{pq} is the identity on all goals which get mapped to $\top$ by h_{pr}, it is possible to find a refutation for the goals which get mapped to $\top$ by h_{pr}. Similarly, it is possible to find a refutation for the goals which get mapped to $\top$ by h_{qr}. Hence $P \times Q/R$ is an extension of R. ■

We conclude this section with an example. By applying the *calculate* technique to the program *List* the following two programs are produced:

```
sum::                                  Len::
sum(X,S):- eq1(X,[ ]),eq2(S,0).        len(Z,N):- eq4(Z,[ ]), eq5(N,0).
sum(Y,S):-                             len(Z,N):-
  eq3(Y,[X|Xs]),sum(Xs,S1),              eq6(Z,[W|Ws]),len(Ws,N1),
  plus(S1,X,S).                          plus1(N, N1).

eq1(X,Y) :- X = Y.                     eq4(A,B) :- A = B.
eq2(X,Y) :- X is Y.                    eq5(A,B) :- A is B.
eq3(X,Y) :- X = Y.                     eq6(A,B) :- A = B.
plus(X,Y,Z) is true if Z equals X + Y.
plus1(X,Y) is true if X equals Y + 1.
```

The composition of *Sum* and *Len* is :

```
Sum_Len::
  sum_len(C,D,E):- eq1_eq4(C,[   ]),eq2(D,0),eq5(E,0).
  sum_len(F,D,E):- eq3_eq6(F,[G|Gs]),
    sum_len(Gs,D1,E1),plus(D1,F,D),plus1(E,E1).
```

The variables were chosen for brevity. The variable C for example, is used instead of X_Z.

7 Conclusion

We claim that the language of skeletons and techniques simplifies program development. In particular, it provides a powerful framework for presenting useful topics in an introductory course in Prolog. Once the idea of skeletons and extensions has been accepted, new topics can be presented via skeletons with the details of the extensions and compositions left for the students as a simple exercise. We have successfully introduced meta-programming via this methodology by using the solve skeleton and then extending it for different applications.

A consequence of constructing Prolog programs by applying techniques to skeletons is that this imposes some standardization for Prolog programs. This could make Prolog a more inviting tool for software engineering projects since each module of code could start with an underlying skeleton which is extended and composed. This approach to program development is, in our experience, accessible both for the average and neophyte Prolog programmer.

Another reason for our optimism is the great potential for making program development automatic or at least semi-automatic. With a given set of skeletons and techniques to work with, it is possible to automate the creation of enhancements and their composition to produce programs. If a module contains only those techniques that have been proven to produce extensions, it is possible to make conclusions about the composed program. This means if the skeleton meets the specifications of a problem we can conclude an enhancement will also meet the necessary requirements. Programs developed in this manner have correctness built into the construction thus minimizing program testing. Therefore, the programmer would be responsible for only the problem solving i.e. finding the correct flow of the program and the technique which would best suit the given problem but not the details of putting it together.

We have applied the method of skeletons and techniques to machine learning with some success. Skeletons and techniques were used to generate new Prolog clauses for Shapiro's Model Inference System [Kirsch 91]. Some Prolog programs which could not be synthesized using MIS, were learned with this methodology.

In conclusion, this paper presents the theory underlying stepwise enhancement. We believe stepwise enhancement is applicable to various areas

of computer science. Besides machine learning and software engineering which were mentioned above, the second author used skeletons and techniques to help develop expert systems. Hence, applying stepwise enhancement to artificial intelligence type problems shows great promise.

Acknowledgements

We acknowledge stimulating discussions among the ProSE group at Case Western Reserve University. This work was supported by NSF grant CCR-90-00387.

References

[Kirsch 91] Marc Kirschenbaum and Leon Sterling. *Refienemtn Strategies for Inductive Learning Of Simple Prolog Programs.* Proceedings of 12th IJCAI, Australia, 1991.

[Lakhotia 90] Arun Lakhotia and Leon Sterling. *A Method for Incremental Programming.* Proceedings of Skokie Conference on Software Engineering, Skokie IL., June 1990.

[Lloyd 84] J.W.Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Power 90] A.J.Power and Leon Sterling. *A Notion of Map between Logic Programs*, Proceedings of the Seventh International Conference on Logic Programming, (eds. Warren, D.H.D and Szeredi, P.) pages 390-404, MIT Press, 1990.

[Sterling 88] Leon Sterling and Arun Lakhotia. *Composing Prolog Meta-interpreters.* Proceedings of the Fifth International Conference on Logic Programming, (eds. Bowen, K. and Kowalski, R.), Seattle, pages 386-403, MIT Press, August 1988.

[Sterling 89] Leon Sterling and Randall Beer. *Meta-interpreters for Expert Systems.* Journal of Logic Programming, vol. 6, no. 1 and 2, pages 163-178, 1989.