

A theorem on refinement operators for logic program synthesis

Nancy Lynn Tinkham

Computer Science Department

Rowan University

Glassboro, NJ 08028

E-mail: nlt@rowan.edu

May 1997

Abstract

Upward and downward refinement operators can be used to order the space of program schemata; these operators make program generalization and synthesis more efficient. This paper presents a theorem relating a pair of refinement operators to the sets of goals covered by schemata.

1. Introduction

Refinement operators have several applications in the automatic synthesis of logic programs. Refinement operators were introduced by Shapiro [4, 5, 6], who used them for refining discarded hypotheses. The mathematics of refinement operators in themselves were studied by Laird [1, 2], who described both “downward” refinement (of which Shapiro’s operators were examples) and “upward” refinement. Tinkham [7] applies these ideas to allow patterns in previously-seen programs to guide the synthesis of new programs: a generalization (upward refinement) operator and a specialization (downward refinement) operator are used to form generalizations of known programs; these generalizations (called schemata), together with the specialization operator, are then used to synthesize new programs.

This paper presents a theorem showing that schemata which are related by generalization are also related by the sets of goals they cover. This revises an earlier result proved in Tinkham [7], which used a slightly different definition of “covers”.

2. Language and operator definitions

Prolog is used in this paper as the basis for the language in which to express programs and schemata. Programs are represented as multisets of Prolog-like clauses; the clauses will differ from the standard Prolog form in that the right hand side of a clause will be regarded as a multiset¹ rather than a sequence of literals. We will restrict our attention to programs defining only a single predicate. Hence, one example of a program is:

$$\begin{aligned} & \{ \textit{flatten} ([], []), \\ & \textit{flatten} ([R|S], [R|T]) :- \{ \textit{atom}(R), \backslash == (R, []), \textit{flatten}(S, T) \}, \\ & \textit{flatten} ([[U|V]|W], X) :- \{ \textit{flatten}([U|V], Y), \textit{flatten}(W, Z), \textit{append}(Y, Z, X) \} \end{aligned}$$

Observe that this program would continue to be a correct definition of *flatten* even if the sequence of literals or clauses were different; we are specifically choosing to study order-independent programs.

A schema has a representation like that of a program, except that a schema may contain predicate variables and may contain the symbol \square (empty clause). The special symbols \emptyset and $\{ \square \}$ will be used to represent the most specific and most general schemata, respectively. We will use “program” as the special case of “schema” in which no predicate variables or empty clauses occur; hence, a program is a schema, but a schema may or may not be a program.

¹ A multiset is a collection of objects in which repetition is significant, but, as in a set, order is not significant. The operations \cup (union), \cap (intersection), \subseteq (subset), \subset (proper subset), $+$ (sum), and $-$ (difference) on literals within clauses and on clauses within schemata will be multiset operations. For a definition of these multiset operators, see Liu [3].

3. Language definitions

Some terminology must be introduced here for describing programs and schemata. A *term* is an individual-variable, an individual-constant, or a function symbol with its arguments. In the *flatten* program, X , $[]$, and $[R \mid S]$ are all terms. A *literal* is a predicate symbol with its arguments; $atom(R)$ is an example of a literal. A *clause* is either \square (representing the empty clause), a single literal, or an expression of the form

$$\lambda_1 :- \{ \lambda_2, \dots, \lambda_n \} \quad ,$$

where $\lambda_1, \dots, \lambda_n$ are literals, and $\{ \lambda_2, \dots, \lambda_n \}$ is a multiset of literals. The literal in a single-literal clause and the literal on the left-hand side of a multi-literal clause (λ_1) are *positive literals*; the literals on the right hand side of a clause ($\lambda_2, \dots, \lambda_n$) are *negative literals*. As an example, one of the clauses in the *flatten* program is

$$flatten([[U|V]|W], X) :- \{ flatten([U|V], Y), flatten(W, Z), append(Y, Z, X) \}$$

It contains a positive literal, $flatten([[U|V]|W], X)$, and three negative literals, $flatten([U|V], Y)$, $flatten(W, Z)$, and $append(Y, Z, X)$. A *schema* is a multiset of clauses which contains only one predicate symbol in the clauses' positive literals. Thus, the *flatten* program above is a schema, as is

$$\begin{aligned} & \{ p(X) :- \{ R(X, Y) \}, \\ & p(W) :- \{ p(Z) \} \end{aligned}$$

However,

$$\begin{aligned} & \{ p(X) :- \{ R(X, Y) \}, \\ & q(Z) :- \{ p(Z) \} \end{aligned}$$

is not a schema, since it contains both p and q in its positive literals. A schema which contains no predicate-variables and does not contain \square is called a *program*. The definition of *flatten* above, for example, is a program.

The following notation conventions will be used for constant and variable symbols:

Individual-variables will be written as upper-case letters U, V, W, X, Y, Z .

Individual-constants will be written as lower-case letters a, b, c .

Function symbols will be written as lower-case letters f, g, h . When needed for clarity, the arity of a function or predicate will be indicated by a superscript: f^2, p^4 .

Predicate-constants will be written as lower-case letters p, q, r .

Predicate-variables will be written as upper-case letters P, Q, R .

Mnemonic names for constants and functions (such as *append* and numerals) will also be used. As in Prolog, when working with the list-forming functor “.”, we will usually use list notation, rather than explicitly nested functions, to represent the list. For example, $.(a, .(b, []))$ will be written $[a, b]$, and $.(a, .(b, X))$ will be written $[a, b|X]$.

For any particular application, we will define schemata in terms of a finite set of function symbols and predicate-constants; this models a setting in which a finite set of Prolog predicates is “known”, having been previously defined, and we are defining a single new predicate. A finite set K of function symbols, individual-constants, and predicate-constants will be called a *constant set*. If K is a constant set and A is the set of all integers a such that there is a predicate-constant in K of arity a , then L is defined to be the *schema-definition language over K* if L is the set of all schemata σ such that every function symbol, individual-constant, or predicate-constant occurring in σ is an element of K , and every predicate-variable P occurring in σ has an arity a_P such that $a_P \in A$. (Observe that, regardless of the choice of K , L will contain the elements \emptyset and $\{\square\}$.)

Example:

Let $K = \{p^1, q^2, f^1, b\}$, and let L be the schema-definition language over K . Some examples of schemata in L are:

- 1) $\{ p(X) :- \{ q(Y, Z), p(f(Y)), p(Z) \},$
 $p(b) \}$
- 2) $\{ R(X) :- \{ p(X) \} \}$

Example (1) is also a program, because it contains no predicate-variables. An example of an expression which is *not* a schema in L is:

- 3) $\{ p(X) :- \{ r(X) \},$
 $p(f(Z)) :- \{ p(Z) \} \}$

because it contains the predicate-constant r , which is not a member of K . •

Three final language-related definitions will make the discussion of operators in the next section easier. Let L be the schema-definition language over a constant set K . A *most-general positive literal* in a schema $\sigma \in L$ is a positive literal either of form P^0 , where P^0 is a predicate-variable, or of form $P^n(X_1, X_2, \dots, X_n)$, where P^n , $n > 0$, is a predicate-variable and X_1, \dots, X_n are individual-variables occurring exactly once in σ . (Recall that, from the definition of a schema-definition language, n must be an integer such that there is a predicate-constant of arity n among the constants in K .) A *most-general negative literal* in $\sigma \in L$ is similarly defined as a negative literal either of form P^0 , or of form $P^n(X_1, X_2, \dots, X_n)$, where X_1, \dots, X_n are individual-variables occurring exactly once in σ , with the additional constraint that the predicate-variable P^n must occur only once in σ . This constraint is omitted from the definition for positive literals because of the requirement that the predicate symbols appearing in the positive literals of a schema must be identical.

A *most-general term* in $\sigma \in L$ is a term which is either an individual-constant or a term of form $f^n(X_1, X_2, \dots, X_n)$, where f^n is a function symbol and X_1, \dots, X_n are individual-variables occurring exactly once in σ .

4. Operator definitions

This section describes an ordering relation on schemata and a family of refinement operators. In the presentation that follows, the substitution replacing all occurrences of V by t will be written as $\{V \setminus t\}$.

Definition:

Define an equivalence relation \approx on schemata: For schemata σ_1 and σ_2 , $\sigma_1 \approx \sigma_2$ exactly if σ_1 and σ_2 are identical except for, possibly, the naming of variables and the order of listing negative literals within a clause and clauses within a schema.

Examples:

$\{P(X), P(a)\} \approx \{Q(Y), Q(a)\}$, since P can be renamed Q and X can be renamed Y .

$\{P(X) :- \{q(X), r(X)\}, P(a)\} \approx \{P(a), P(X) :- \{r(X), q(X)\}\}$, since they differ only in order. •

For simplicity of presentation in the remainder of the paper, two schemata that are equivalent in the sense of \approx (that is, two schemata that differ only in variable names and in order of literals and clauses) will be considered to be the same schema.

Definition:

Define a partial order on schemata, \leq , as follows: If $\kappa_1 = \alpha_1 :- S_1$ and $\kappa_2 = \alpha_2 :- S_2$ are clauses, where S_1 and S_2 are (possibly empty) multisets of literals, then $\kappa_1 \leq \kappa_2$ exactly if

- 1) κ_2 is \square , or
- 2) there is a substitution θ such that $\alpha_2\theta = \alpha_1$ and $S_2\theta \subseteq S_1$ (where \subseteq is the multiset subset relation).

If σ_1 and σ_2 are schemata, then $\sigma_1 \leq \sigma_2$ exactly if

- 1) σ_1 and σ_2 contain only \square clauses, and σ_1 contains at least as many clauses as σ_2 ; or
- 2) σ_1 contains at least one non- \square clause, and there is a one-to-one mapping ϕ from clauses in σ_1 to clauses in σ_2 and a substitution θ such that if $\kappa_1 \in \sigma_1$, $\kappa_2 \in \sigma_2$, and $\kappa_2 = \phi(\kappa_1)$, then $\kappa_1 \leq \kappa_2$ with substitution θ .

Example:

If σ_1 is

$$\{p(U) :- \{Q(a, V), r(b)\}, \\ p(a)\}$$

and σ_2 is

$$\begin{aligned} & \{p(Z) \text{ ,} \\ & p(W) :- \{S(X, Y)\} \text{ ,} \\ & p(c) :- \{p(d)\} \text{ } \} \text{ ,} \end{aligned}$$

then $\sigma_1 \leq \sigma_2$, with $\theta = \{Z \setminus a, W \setminus U, S \setminus Q, X \setminus a, Y \setminus V\}$. •

The ordering \leq is easily seen to be reflexive (let θ be the empty substitution) and transitive (since we can compose substitutions). Tinkham [7] shows that \leq is antisymmetric and thus that \leq is a partial order.

Given an ordering on expressions such as \leq , Laird [2] defines an *upward refinement* γ to be a recursively enumerable relation on expressions such that γ^* is \leq , and a *downward refinement* ρ to be a recursively enumerable relation on expressions such that ρ^* is \leq^{-1} . When viewed computationally, γ and ρ are referred to as (upward and downward) refinement operators. The notation $\gamma(\sigma)$ denotes the set of all expressions which can be produced by applying γ once to σ , and $\gamma^*(\sigma)$ is the set of all expressions which can be produced by 0 or more applications of γ to σ .

Two refinement operators are defined below, one for upward refinement (generalization) and one for downward refinement (specialization). Each of the operators has been divided into two parts; hence, we will define generalization operators γ_1 and γ_2 and specialization operators ρ_1 and ρ_2 . γ_1 and ρ_1 keep the number of clauses and \square s in a schema constant and thus have more easily described properties, as explored in Tinkham [7].

Definition of γ_1

Let K be a set of function symbols and predicate-constants. Let L be the schema-definition language over K , and let σ_1 and σ_2 be schemata in L . Then $\sigma_2 \in \gamma_1(\sigma_1)$ exactly if one of the following hold:

- 1) **Deleting negative literal:** σ_2 is derived from σ_1 by deleting a most-general negative literal λ from some clause κ in σ_1 .
- 2) **Separating individual-variables:** X is an individual-variable occurring more than once in σ_1 , and σ_2 is derived from σ_1 by replacing one or more, but not all, of the occurrences of X by an individual-variable Y not occurring in σ_1 .
- 3) **Separating predicate-variables:** P is a predicate-variable occurring more than once in σ_1 , and σ_2 is derived from σ_1 by replacing one or more, but not all, of the occurrences of P by a predicate-variable Q not occurring in σ_1 . This rule may only be applied when the result will be a schema — that is a set of clauses with only one predicate symbol in the positive literals.

- 4) **Generalizing predicate:** p is a predicate-constant occurring in a negative literal in σ_1 , P is a predicate-variable not occurring in σ_1 , and σ_2 is derived from σ_1 by replacing one or more occurrences of p in negative literals by P .
- 5) **Generalizing predicate:** p is a predicate-constant occurring in a positive literal in σ_1 , P is a predicate-variable not occurring in σ_1 , and σ_2 is derived from σ_1 by replacing *all* occurrences of p in positive literals and, optionally, one or more occurrences of p in negative literals, by P .
- 6) **Generalizing term:** σ_2 is derived from σ_1 by replacing one or more occurrences of a most-general term t in σ_1 by an individual-variable X not occurring in σ_1 .

Definition of γ_2

Let K be a set of function symbols and predicate-constants. Let L be the schema-definition language over K , and let σ_1 and σ_2 be schemata in L . Then $\sigma_2 \in \gamma_2(\sigma_1)$ exactly if one of the following hold:

- 1) $\sigma_2 \in \gamma_1(\sigma_1)$.
- 2) **Adding clause:** σ_1 and σ_2 do not contain \square , and σ_2 is derived from σ_1 by adding one clause κ to the set of clauses in σ_1 .
- 3) **Replacing most-general positive literal by \square :** Clause κ in σ_1 is a set containing a single most-general positive literal and no negative literals, and σ_2 is derived from σ_1 by replacing κ by \square .
- 4) **Deleting duplicate occurrence of \square :** σ_1 is a set containing $n + 1$ occurrences of \square (and no other clauses), and σ_2 is a set containing n occurrences of \square (and no other clauses), for some $n > 0$.

Definition of ρ_1

Let K be a set of function symbols and predicate-constants. Let L be the schema-definition language over K , and let σ_1 and σ_2 be schemata in L . Then $\sigma_2 \in \rho_1(\sigma_1)$ exactly if one of the following hold:

- 1) **Adding negative literal:** σ_2 is derived from σ_1 by adding a most-general negative literal λ to some clause κ in σ_1 , where κ is not \square .
- 2) **Unifying individual-variables:** X and Y are distinct individual-variables occurring in σ_1 , and σ_2 is derived from σ_1 by replacing all occurrences of Y by X .
- 3) **Unifying predicate-variables:** P and Q are distinct predicate-variables occurring in σ_1 , and σ_2 is derived from σ_1 by replacing all occurrences of Q by P .

- 4) **Replacing predicate-variable by predicate-constant:** P is a predicate-variable occurring in σ_1 , p is a predicate-constant, and σ_2 is derived from σ_1 by replacing all occurrences of P by p .
- 5) **Replacing individual-variable by most-general term:** X is an individual-variable occurring in σ_1 , t is a most-general term, and σ_2 is derived from σ_1 by replacing all occurrences of X by t .

Definition of ρ_2

Let K be a set of function symbols and predicate-constants. Let L be the schema-definition language over K , and let σ_1 and σ_2 be schemata in L . Then $\sigma_2 \in \rho_2(\sigma_1)$ exactly if one of the following hold:

- 1) $\sigma_2 \in \rho_1(\sigma_1)$.
- 2) **Deleting a clause:** σ_1 and σ_2 do not contain \square , κ is a clause in σ_1 , and σ_2 is derived from σ_1 by deleting κ .
- 3) **Replacing \square by a most-general positive literal.** $\square \in \sigma_1$, and σ_2 is derived from σ_1 by replacing \square by a most-general positive literal. This rule may only be applied when the result will be a schema — that is a set of clauses with only one predicate symbol in the positive literals.
- 4) **Duplicating \square :** σ_1 is a set containing n occurrences of \square (and no other clauses), and σ_2 is a set containing $n + 1$ occurrences of \square (and no other clauses), for some $n > 0$.

We add two definitions for discussing these operators:

Definition:

Let σ_1 and σ_2 be schemata. If $\sigma_1 \in \gamma_2^*(\sigma_2)$, we will say that σ_1 is a *generalization* of σ_2 . We will also say that σ_1 is *more general than* σ_2 .

A schema σ is said to be a *generalization* of a set of schemata Π if σ is a generalization of every schema in Π .

Definition:

Let σ_1 and σ_2 be schemata. If $\sigma_1 \in \rho_2^*(\sigma_2)$, we will say that σ_1 is a *specialization* of σ_2 . We will also say that σ_1 is *more specific than* σ_2 .

A schema σ is said to be a *specialization* of a set of schemata Π if σ is a specialization of every schema in Π .

Example:

To illustrate the use of the refinement operator ρ_2 , here is an example derivation of a program *max* from the most general schema, $\{\square\}$. (Changes at each step are indicated in **bold**.) First, apply rule 3 of ρ_2 to produce a 2-clause schema:

$\{\square\}$
 $\rightarrow \{\square, \square\}$

Then replace each \square with a most-general literal:

$\rightarrow \{\mathbf{P(X1, X2, X3)}, \square\}$
 $\rightarrow \{P(X1, X2, X3), \mathbf{P(Y1, Y2, Y3)}\}$

Next, add some most-general negative literals to the clauses:

$\rightarrow \{P(X1, X2, X3), P(Y1, Y2, Y3) :- \mathbf{R(Y4, Y5)}\}$
 $\rightarrow \{P(X1, X2, X3) :- \mathbf{Q(X4, X5)}, P(Y1, Y2, Y3) :- \{R(Y4, Y5)\}\}$

Then replace predicate-variables by predicate-constants:

$\rightarrow \{\mathbf{max(X1, X2, X3)} :- \{Q(X4, X5)\}, \mathbf{max(Y1, Y2, Y3)} :- \{R(Y4, Y5)\}\}$
 $\rightarrow \{\max(X1, X2, X3) :- \{Q(X4, X5)\}, \max(Y1, Y2, Y3) :- \{Y4 > Y5\}\}$
 $\rightarrow \{\max(X1, X2, X3) :- \{X4 \geq X5\}, \max(Y1, Y2, Y3) :- \{Y4 > Y5\}\}$

Finally, unify the individual-variables until the goal program is produced:

$\rightarrow \{\max(X1, X2, X3) :- \{X4 \geq X5\}, \max(Y1, Y2, Y3) :- \{Y4 > \mathbf{Y1}\}\}$
 $\rightarrow \{\max(X1, X2, X3) :- \{X4 \geq X5\}, \max(Y1, Y2, Y3) :- \{\mathbf{Y2} > Y1\}\}$
 $\rightarrow \{\max(X1, X2, X3) :- \{X4 \geq X5\}, \max(Y1, Y2, \mathbf{Y2}) :- \{Y2 > Y1\}\}$
 $\rightarrow \{\max(X1, X2, X3) :- \{X4 \geq \mathbf{X2}\}, \max(Y1, Y2, Y2) :- \{Y2 > Y1\}\}$
 $\rightarrow \{\max(X1, X2, X3) :- \{\mathbf{X1} \geq X2\}, \max(Y1, Y2, Y2) :- \{Y2 > Y1\}\}$
 $\rightarrow \{\max(X1, X2, \mathbf{X1}) :- \{X1 \geq X2\}, \max(Y1, Y2, Y2) :- \{Y2 > Y1\}\}$ ●

Tinkham [7] proves several properties of γ_2 and ρ_2 . Among these are 1) that γ_1 and ρ_1 are inverse operations; 2) that γ_2 and ρ_2 are inverse operations; and 3) for schemata σ_1 and σ_2 , $\sigma_1 \leq \sigma_2$ iff $\sigma_1 \in \rho_2^*(\sigma_2)$ (that is, the ordering induced by the specialization operator is the same as that of \leq ; hence, generalization and specialization are indeed refinement operators).

5. Operator properties

To relate the refinement operators to the set of goals which can be “computed” by a schema, we define *the set of goals covered by a schema* and then present a theorem relating this set of goals to the refinement relation \leq .

Definitions:

An *interpretation* is a set I of ground atoms. A *goal* is a single ground atom. A schema σ is said to *cover goal g in I* if

- 1) one of the clauses in σ is the symbol \square ; or
- 2) σ contains a clause κ of the form $\alpha :- \{\lambda_1, \dots, \lambda_n\}$, and there is a substitution θ such that $\alpha\theta = g$ and either $\lambda_i\theta$ is in I or σ covers $\lambda_i\theta$, for $1 \leq i \leq n$.

The set of goals covered by a schema σ in an interpretation I will be denoted by $C_I(\sigma)$.

Example:

Let I be $\{q(a), q(b), r(a)\}$, and let σ be $\{p(a), p(f(X)) :- \{p(X), q(X)\}\}$. σ covers the goal $p(a)$, because σ contains the clause $p(a)$, and $p(a) \{\} = p(a)$. σ also covers the goal $p(f(a))$, because σ contains the clause $p(f(X)) :- \{p(X), q(X)\}$: if we apply the substitution $\{X \setminus a\}$ to the literals of this clause, obtaining $p(f(a)) :- \{p(a), q(a)\}$, we see that $p(f(a))$ matches our goal, $p(a)$ is covered by σ , and $q(a)$ is in I . σ does not cover $p(b)$ or $p(f(b))$. •

We can now prove a theorem which shows that schemata related by \leq are also related by the sets of goals covered: If σ_1 is a specialization of σ_2 , then σ_1 covers a subset of the goals covered by σ_2 .

Theorem:

Let σ_1 and σ_2 be schemata and I be an interpretation. If $\sigma_1 \leq \sigma_2$, then $C_I(\sigma_1) \subseteq C_I(\sigma_2)$.

Proof:

Since $\sigma_1 \leq \sigma_2$, $\sigma_1 \in \rho_2^*(\sigma_2)$.

Case 1: σ_1 is derived from σ_2 by adding a literal λ to clause $\kappa_2 = \alpha :- \{\lambda_1, \dots, \lambda_n\}$ of σ_2 to produce clause $\kappa_1 = \alpha :- \{\lambda_1, \dots, \lambda_n, \lambda\}$ in σ_1 : Let $\sigma' = \sigma_2 - \{\kappa_2\}$. Any goal that is added to $C_I(\sigma')$ by adding clause κ_1 would also be added by clause κ_2 , since $\{\lambda_1\theta, \dots, \lambda_n\theta\} \subseteq \{\lambda_1\theta, \dots, \lambda_n\theta, \lambda\theta\}$. Thus $C_I(\sigma' + \{\kappa_1\}) \subseteq C_I(\sigma' + \{\kappa_2\})$; that is, $C_I(\sigma_1) \subseteq C_I(\sigma_2)$.

Case 2: σ_1 is derived from σ_2 by a substitution ϕ which a) unifies two individual-variables or two predicate-variables, b) replaces all occurrences of an individual-variable by a term, or c) replaces one or more occurrences

of a predicate variable by a predicate constant:

If σ_1 contains \square , then σ_2 also contains \square , so both schemata cover all possible goals.

Otherwise, consider the effect of replacing a single clause $\kappa\phi = \alpha\phi : -\{\lambda_1\phi, \dots, \lambda_n\phi\}$ in σ_1 by $\kappa = \alpha : -\{\lambda_1, \dots, \lambda_n\}$. Let $\sigma' = \sigma_1 - \{\kappa\phi\}$; the new schema is thus $\sigma' + \{\kappa\}$. Any goal that is added to $C_I(\sigma')$ by $\kappa\phi$ will also be added to $C_I(\sigma')$ by κ , since ϕ can be applied to the literals in κ as the first step in matching the positive literal to the goal. That is, if g is a goal that is covered using clause $\kappa\phi$ and substitution θ in σ_1 , then g will be covered using clause κ and substitution $\phi\theta$ in $\sigma' + \{\kappa\}$. By repeating this argument for each clause in σ_1 , we see that the set of goals covered by σ_1 is a subset of those covered by the more general schema σ_2 .

Case 3: σ_1 is derived from σ_2 by deleting a clause : Let g be a goal covered by σ_1 . Since every clause in σ_1 occurs in σ_2 as well, g is covered by σ_2 .

Case 4: σ_1 is derived from σ_2 by replacing one occurrence of \square by a clause consisting of a single most-general positive literal : Let g be a goal covered by σ_1 . Since σ_2 contains \square , g is covered by σ_2 .

Case 5: σ_1 consists of n occurrences of \square , and σ_2 consists of $n-1$ occurrences of \square : Since a schema containing \square covers any goal, σ_1 and σ_2 are both schemata which cover all goals; hence, any goal covered by σ_1 is also covered by σ_2 . •

In summary, we see that operators γ_2 and ρ_2 are refinement operators, corresponding to the relation \leq ; and that this relation also corresponds to a subset relation on the set of goals covered by a schema.

References

- [1] Laird, P. D., *Inductive Inference by Refinement* (Yale University Computer Science Department, New Haven, TR-376, 1986).
- [2] Laird, Philip, *Learning from Good Data and Bad* (Yale University, New Haven, Ph.D. dissertation, TR-551, 1987).
- [3] Liu, C. L., *Elements of Discrete Mathematics* (McGraw-Hill, New York, 1977).
- [4] Shapiro, Ehud Y., *Inductive Inference of Theories from Facts* (Yale University Computer Science Department, New Haven, Research Report 192, 1981).
- [5] Shapiro, Ehud Y., An algorithm that infers theories from facts, in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (1981) 446-451.
- [6] Shapiro, Ehud Y., *Automatic Program Debugging* (MIT Press, Cambridge, MA, 1982).

- [7] Tinkham, Nancy L., Induction of schemata for program synthesis (Duke University technical report, Durham, NC, CS-1990-14, 1990).