ELSEVIER Artificial Intelligence 98 (1998) 1–47

Artificial Intelligence

Schema induction for logic program synthesis

Nancy Lynn Tinkham[1]

Computer Science Department, Rowan University, 201 Mullica Hill Road, Glassboro, NJ 08028, USA

Received February 1993; revised August 1997

Abstract

Prolog program synthesis can be made more efficient by using schemata which capture similarities in previously-seen programs. Such schemata narrow the search involved in the synthesis of a new program. We define a generalization operator for forming schemata from programs and a downward refinement operator for constructing programs from schemata. These operators define schema-hierarchy graphs which can be used to aid in the synthesis of new programs. Algorithms are presented for efficiently obtaining least generalizations of schemata, for adding new schemata to a schema-hierarchy graph, and for using schemata to construct new programs. © 1998 Elsevier Science B.V.

Keywords: Inductive logic programming; Inductive inference; Automatic programming; Learning

1. Introduction

When writing computer programs, people often find it useful to draw on the knowledge of other programs that have been written before. An experienced programmer may, when presented with a new problem, recall solving a similar problem on an earlier occasion; a novice programmer may use examples from the classroom or textbook to guide problem-solving.

This observation motivates the following hypothesis: One way for machines to synthesize programs is (1) to see examples of programs, (2) to form generalizations which capture information about the forms of programs, and then (3) to use these generalizations in writing future programs. This paper will describe a language for expressing generalizations of programs, an algorithm for deriving generalizations, and an algorithm for synthesizing a program from a generalization.

[1] Email: nlt@rowan.edu.

As an illustration, consider the task of synthesizing the Prolog predicate *suffix(SuffixList, List)*, which succeeds if *SuffixList* is a suffix of *List*, given some examples of the intended behavior of the *suffix* predicate:

positive examples: *suffix*([*c*], [*a*, *b*, *c*]), *suffix*([*y*, *q*], [*z*, *y*, *q*])
negative examples: *suffix*([*a*, *b*], [*a*, *b*, *c*]), *suffix*([*z*], [*w*, *x*])

Our task becomes easier if we have knowledge of the similar predicates *prefix(PrefixList, List)*,

```
prefix( [ ], W).
prefix( [X|Y], [X|Z]) :- prefix(Y, Z).
```

which succeeds if *PrefixList* is a prefix of *List*, and *member(Element, List)*,

```
member(V, [V|W]).
member(X, [Y|Z]) :- member(X, Z).
```

which succeeds if *Element* is a member of *List*. The first step is to derive a schema which is a generalization of *prefix* and *member*. One such schema is the following:

```
Q(U, V).
Q(W, [X|Y]) :- Q(Z, Y).
```

This schema expresses the structure of a recursive clause together with a clause serving as the base of the recursion, a structure which is very common in Prolog programs. Further, we have the information that one of the arguments is a list, and that the recursive call involves the tail of that list.

Taking this schema as a starting point, we search for a program which is a specialization of the schema and which succeeds on all of the positive examples but none of the negative examples; eventually, the search finds

```
suffix(W, W).
suffix(X, [Y|Z]) :- suffix(X, Z).
```

The process, then, has two major components: (1) to derive a schema which is a generalization of a set of programs and captures some of the structural information about each of these programs; (2) to derive from a schema, making use of this structural information, a program which is consistent with a given set of positive and negative examples.

1.1. Background

Inductive inference, the process of learning from examples, covers a range of applications, including grammatical inference, inference of logic formulas, learning structures encoded in semantic nets, hypothesizing mathematical theorems, and automatic programming.

Gold [16] introduced identification in the limit as a model for the inference of a language from examples. Learning, in Gold's paper, is performed by enumerative algorithms—that is, algorithms that in some systematic way consider all machines in a given class until a machine for the target language is found. The main advantage of an enumerative algorithm is its thoroughness, which often makes it possible to prove which classes of languages can and cannot be identified by the algorithm. This thoroughness is also the main disadvantage of an enumerative approach, in that the vast number of possibilities examined makes the search extremely slow.

One of the improvements to enumerative algorithms is the introduction of refinement operators, which prune the search without sacrificing theoretical power. Refinement operators were introduced by Shapiro [30–32], who used them for refining discarded hypotheses. The mathematics of refinement operators in themselves were studied by Laird [19,20], who described both "downward" refinement (of which Shapiro's operators were examples) and "upward" refinement. We will return to refinement operators in Section 2.2, defining some operators for generalizing and specializing programs.

Concept learning has also been studied by Valiant [40], Angluin and Laird [1], Mitchell [25], Michalski [24], and Winston [41]. Early work on finding least generalizations of literals and clauses was done by Popplestone [28], Reynolds [29], and Plotkin [26,27]. A survey of inductive inference systems is given by Angluin and Smith [2], and work in grammatical inference is surveyed by Biermann and Feldman [6].

Automatic programming systems have been designed to work with LISP, Prolog, and other languages, and the input to these systems variously includes input/output examples (as in [37]), input/output specifications [4,7,23,34], and computation traces (e.g., [5]). The use of transformation rules to construct or improve programs has been studied by Burstall and Darlington [8] and Dershowitz [11,12]. Logic program synthesis has been studied by Flener and Deville [13], Gilbert and Hogger [15], Sterling and Kirschenbaum [35], Lau and Prestwich [21], Bergadano and Gunetti [3], Grobelnik [17], Johansson [18], and others.

1.2. Project overview

This paper describes a project which applies the idea of refinement operators to the problem of using known programs to aid in the synthesis of a new program to fit a set of positive and negative examples. It defines an upward refinement operator and an algorithm for using this operator efficiently to find generalizations of programs, called schemata. The paper then shows how these schemata can be used to narrow the search involved in program synthesis.

These ideas have been implemented as a Prolog system which repeatedly adds new schemata to its knowledge base so that the system becomes increasingly more efficient at synthesizing new programs. This implemented system will be described at the end of the paper.

2. Language and operator definitions

Prolog has been chosen in this paper as the basis for the language in which to express programs and schemata. Programs will be 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 multiset[2] 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:

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

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. This enables us to view programs more directly as logic expressions, without involving the extra-logical concept of order of computation. It is also more in keeping with a philosophy of Prolog programming which favors writing, where possible, programs which do not depend on the order of execution for correctness.

A schema will have a representation like that of a program, except that a schema may contain predicate variables and may contain the symbol $\Box$ (empty clause). The special symbols $\emptyset$ and $\{\Box\}$ 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.

For an overview of the Prolog programming language, see [10] and [36].

2.1. 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 $[R|S]$ are all terms. A *literal* is a predicate symbol with its arguments; $atom(R)$ is an example of a literal. A *clause* is either $\Box$ (representing the empty clause), a single literal, or an expression of the form

$$\lambda_1 \text{ :- } \{\lambda_2, \ldots, \lambda_n\},$$

where $\lambda_1, \ldots, \lambda_n$ are literals, and $\{\lambda_2, \ldots, \lambda_n\}$ 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 ($\lambda_2, \ldots, \lambda_n$) are *negative literals*. As an example, one of the clauses in the *flatten* program is

[2] 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 Appendix A and [22].

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

It contains a positive literal, $\mathit{flatten}([[U|V]|W], X)$, and three negative literals, $\mathit{flatten}([U|V], Y), \mathit{flatten}(W, Z)$, and $\mathit{append}(Y, Z, X)$. 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

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

However,

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

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 $\Box$ 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 $(X, Y, \ldots)$, individual-constants as lower-case letters $(a, b, \ldots)$, function symbols as lower-case letters $(f, g, \ldots)$, predicate-constants as lower-case letters $(p, q, \ldots)$, and predicate-variables as upper-case letters $(P, Q, \ldots)$. When needed for clarity, the arity of a function or predicate will be indicated by a superscript: f^2, p^4. 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, and 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 $\{\Box\}$.)

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) $\{p(X) \text{ :- } \{q(Y, Z), p(f(Y)), p(Z)\},$
$\quad p(b)\}$

(2) $\{R(X) \text{ :- } \{p(X)\}\}$

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

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

The choice to use multisets rather than ordinary sets (as might be more intuitive initially) in representing schemata was made for several reasons. First, if the program is viewed as a computation, observe that if two identical literals appear on the right hand side of a Prolog predicate, this duplication will indeed cause a repeated computation; in that sense, the program is not the same as one in which the duplicated literal is removed. Second, the properties studied in Section 3, degree of branching and degree of recursion, are much more well-behaved when a multiset representation is used.

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 $P^n(X_1, X_2, \ldots, X_n)$, where P^n, $n > 0$, is a predicate-variable and $X_1, \ldots, X_n$ 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 $P^n(X_1, X_2, \ldots, X_n)$, where $X_1, \ldots, X_n$ 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 σ is a term which is either an individual-constant or a term of form $f^n(X_1, X_2, \ldots, X_n)$, where f^n is a function symbol and $X_1, \ldots, X_n$ are individual-variables occurring exactly once in σ.

2.2. Operator definitions

This section introduces an ordering relation on schemata and a family of refinement operators. The next section will describe some of the properties of the relation and operators. In the presentation that follows, the substitution replacing all occurrences of V by t will be written as $\{V \backslash t\}$.

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 $\alpha \text{ :- } \{\lambda_1, \ldots, \lambda_n\}$, 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 $1 \leqslant i \leqslant n$.

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 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 $\{X\backslash a\}$ 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 $p(f(b))$.

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 $\lesssim$, 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 $\kappa_1 \lesssim \kappa_2$ exactly if

(1) κ_2 is $\Box$, 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 $\sigma_1 \lesssim \sigma_2$ exactly if

(1) σ_1 and σ_2 contain only $\Box$ clauses, and σ_1 contains at least as many clauses as σ_2; or
(2) σ_1 contains at least one non-$\Box$ 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 $\kappa_1 \lesssim \kappa_2$ with substitution θ.

Example. If σ_1 is

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

and σ_2 is

$\{p(Z)$,
$p(W)$:- $\{S(X, Y)\}$,
$p(c)$:- $\{p(d)\}\}$,

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

The ordering $\lesssim$ is easily seen to be reflexive (let θ be the empty substitution) and transitive (since we can compose substitutions). Tinkham [38] shows that $\lesssim$ is antisymmetric and thus that $\lesssim$ is a partial order.

Given an ordering on expressions such as $\lesssim$, Laird [20] defines an *upward refinement* γ to be a recursively enumerable relation on expressions such that γ^* is $\lesssim$, and a *downward refinement* ρ to be a recursively enumerable relation on expressions such that ρ^* is $\lesssim^{-1}$. 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; $\gamma^n(\sigma)$ is the set of all expressions which can be produced by n applications of γ to σ; and $\gamma^*(\sigma)$ is the set of all expressions which can be produced by 0 or more applications of γ to σ.

We introduce two refinement operators, one for upward refinement (generalization) and one for downward refinement (specialization). In order to make properties of the operators clearer to describe and study, 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.

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 holds:

(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 holds:

(1) $\sigma_2 \in \gamma_1(\sigma_1)$.
(2) *Adding clause*: σ_1 and σ_2 do not contain $\Box$, 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* $\Box$: 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 $\Box$.
(4) *Deleting duplicate occurrence of* $\Box$: σ_1 is a set containing $n+1$ occurrences of $\Box$ (and no other clauses), and σ_2 is a set containing n occurrences of $\Box$ (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 holds:
(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 $\Box$.
(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 a 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 holds:
(1) $\sigma_2 \in \rho_1(\sigma_1)$.
(2) *Deleting a clause*: σ_1 and σ_2 do not contain $\Box$, κ is a clause in σ_1, and σ_2 is derived from σ_1 by deleting κ.
(3) *Replacing* $\Box$ *by a most-general positive literal*: $\Box \in \sigma_1$, and σ_2 is derived from σ_1 by replacing $\Box$ 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* $\Box$: σ_1 is a set containing n occurrences of $\Box$ (and no other clauses), and σ_2 is a set containing $n+1$ occurrences of $\Box$ (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.

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 4 of ρ_2 to produce a 2-clause schema:

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

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

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

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

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

Then replace predicate-variables by predicate-constants:

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

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

$$\rightarrow \{max(X1, X2, X3) \text{ :- } \{X4 \geqslant X5\}, max(Y1, Y2, Y3) \text{ :- } \{Y4 > \boldsymbol{Y1}\}\}$$
$$\rightarrow \{max(X1, X2, X3) \text{ :- } \{X4 \geqslant X5\}, max(Y1, Y2, Y3) \text{ :- } \{\boldsymbol{Y2} > Y1\}\}$$
$$\rightarrow \{max(X1, X2, X3) \text{ :- } \{X4 \geqslant X5\}, max(Y1, Y2, \boldsymbol{Y2}) \text{ :- } \{Y2 > Y1\}\}$$
$$\rightarrow \{max(X1, X2, X3) \text{ :- } \{X4 \geqslant \boldsymbol{X2}\}, max(Y1, Y2, Y2) \text{ :- } \{Y2 > Y1\}\}$$
$$\rightarrow \{max(X1, X2, X3) \text{ :- } \{\boldsymbol{X1} \geqslant X2\}, max(Y1, Y2, Y2) \text{ :- } \{Y2 > Y1\}\}$$
$$\rightarrow \{max(X1, X2, \boldsymbol{X1}) \text{ :- } \{X1 \geqslant X2\}, max(Y1, Y2, Y2) \text{ :- } \{Y2 > Y1\}\}$$

2.3. *Basic properties*

Tinkham [38,39] proves several properties of γ_2 and ρ_2, listed here as Properties 1–8.

Property 1. *Let σ_1 and σ_2 be schemata. $\sigma_1 \in \gamma_1(\sigma_2)$ iff $\sigma_2 \in \rho_1(\sigma_1)$. (That is, γ_1 and ρ_1 are inverse operations.)*

Property 2. *Let σ_1 and σ_2 be schemata. $\sigma_1 \in \gamma_2(\sigma_2)$ iff $\sigma_2 \in \rho_2(\sigma_1)$. (That is, γ_2 and ρ_2 are inverse operations.)*

Property 3. *Let K be a constant set for γ_2, and let L be the schema-definition language over K. Then $\gamma_2^*(\emptyset) = L$. (That is, γ_2^* is sufficiently powerful to generate all of a schema-definition language from its minimal element.)*

Property 4. *Let K be a constant set for ρ_2, and let L be the schema-definition language over K. Then $\rho_2^*(\{\square\}) = L$. (That is, ρ_2^* is sufficiently powerful to generate all of the schema-definition language from its maximal element.)*

Property 5. *Let σ_1 and σ_2 be schemata. Then $\sigma_1 \lesssim \sigma_2$ iff $\sigma_1 \in \rho_2^*(\sigma_2)$. (That is, the ordering induced by the specialization operator is the same as that of $\lesssim$; hence, generalization and specialization are indeed refinement operators.)*

The next two basic properties use a function ξ, which maps schemata into integers. (A similar function is used by Reynolds [29].)

Definition. Define $\xi(\sigma)$ to be

(the number of non-punctuation symbols in σ)

$-$ (the number of distinct variables occurring in σ)

$+$ (the number of literals in σ).

For example,

$$\xi(\{p(0), p(N) :\!-\ \{s(N,M), p(M)\}\}) = 9 - 2 + 4 = 11.$$

(Punctuation symbols are parentheses, braces, commas, and ":-". Symbols in expressions containing lists are counted as though the lists were represented as nested binary functions, rather than in abbreviated list notation; e.g., $[a,b]$, is analyzed as $.(a, .(b, [\ ]))$, containing 5 non-punctuation symbols.)

Property 6. *If σ_1 and σ_2 are schemata and $\sigma_2 \in \rho_1(\sigma_1)$, then $\xi(\sigma_1) + 1 \leqslant \xi(\sigma_2)$. (That is, an application of ρ_1 adds at least 1 to the ξ value of a schema.)*

Property 7. *If σ_1 and σ_2 are schemata and $\sigma_2 \in \rho_1^*(\sigma_1)$, then $\sigma_2 \in \rho_1^n(\sigma_1)$, where $n \leqslant \xi(\sigma_2) - \xi(\sigma_1)$. (That is, σ_2 can be derived from σ_1 in $\xi(\sigma_2) - \xi(\sigma_1)$ or fewer applications of ρ_1.)*

Property 8 shows that schemata related by $\lesssim$ are also related by the sets of goals covered:

Property 8. *Let σ_1 and σ_2 be schemata and I be an interpretation. If $\sigma_1 \lesssim \sigma_2$, then $C_I(\sigma_1) \subseteq C_I(\sigma_2)$. (That is, if σ_1 is a specialization of σ_2, then σ_1 covers a subset of the goals covered by σ_2.)*

3. Finding least generalizations

The main intuition being explored in this paper is that it ought to be easier to solve a new problem if one has seen problems with similar solutions before. Capturing the similarity of a collection of programs is the focus of this section. We want to find similarities that are as specific and thus as informative as possible; hence, we ask the question this way: *Given a set of programs, how can we find a least generalization of that set of programs?*

A least generalization will be defined as follows:

Definition. A schema σ is said to be a *least generalization* of a set of schemata Π if σ is a generalization of Π and there is no schema σ' such that σ' is a specialization of σ and such that σ is a generalization of Π.

Observe that a least generalization is not, in general, unique. For example, both

$$\{p(X) :\!-\ \{q(a,Y)\}\}$$

and

$$\{p(X) :\!-\ \{q(Y,d)\}\}$$

are least generalizations of the set

$$\{\{p(X) :\!-\ \{q(a,b),q(c,d)\}\},\{p(X) :\!-\ \{q(a,d)\}\}\},$$

but neither can be derived from the other using γ_2^*.

3.1. *A simple algorithm for finding least generalizations*

One method for finding a least generalization of a set of programs $\Pi = \{\pi_1,\ldots,\pi_n\}$ is to perform a breadth-first search on the space defined by γ_2: Beginning with the n sets $\{\pi_1\},\ldots,\{\pi_n\}$, add the schemata in $\gamma_2(\pi_i)$ to the ith set, for each i. Next, add the schemata in $\gamma_2(\gamma_2(\pi_i))$ to the ith set, for each i. Continue until the intersection of the generalizations of π_1, of π_2, and … of π_n is nonempty. Return this intersection as output.

While this procedure will find a least generalization (possibly several), the search will examine a large number of schemata, since the graph defined by γ_2 has a large branching factor. After examining some properties of our refinement operators, we will be able to describe a better algorithm which performs a much more constrained search.

3.2. *Properties of refinement operators*

In order to develop a more efficient algorithm for finding least generalizations, we will first explore some of the properties of the generalization and refinement operators γ_2 and ρ_2, so that these properties can be used to restrict the search space. Two properties of particular interest are *degree of branching* and *degree of recursion*.

Definition. The *degree of branching* of a schema σ, $bd(\sigma)$, is defined to be the number of clauses in σ.

Definition. Let σ be a schema such that the positive literals of σ contain the (variable or constant) predicate symbol ϕ. Define the *degree of recursion* of σ, $rd(\sigma)$, to be the maximum of $\{n \mid$ there is a clause κ in σ whose negative literals contain exactly n occurrences of $\phi\}$.

Example. If σ is

$$\begin{aligned}&\{q(4),\\ &\ q(X) \text{ :- } \{r(X)\},\\ &\ q(Y) \text{ :- } \{s(Y, Z, W), q(Z), q(W)\}\},\end{aligned}$$

then $bd(\sigma) = 3$ and $rd(\sigma) = 2$.

These are natural measures to consider, since two of the most obvious ways in which Prolog programs depart from a straight-line form are (1) by containing multiple clauses (allowing a conditional branch) and (2) by containing recursive calls (creating repetition).

Degree of branching and degree of recursion are, under certain conditions, well-behaved under application of γ_1 and ρ_1. This feature leads to an efficient algorithm for finding least generalizations.

We begin by noting some results which follow immediately from the definitions of γ_2 and ρ_2.

- A single application of γ_2 will *increase* or leave unchanged the degree of branching of a schema, and a single application of ρ_2 will *decrease* or leave unchanged the branching degree.
- If γ_2 or ρ_2 is applied to as to leave the degree of branching unchanged (that is, if a clause is not added, in the case of γ_2, or deleted, in the case of ρ_2), then the application of γ_2 will *decrease* or leave unchanged the degree of recursion, and the application of ρ_2 will *increase* or leave unchanged the degree of recursion.

It is possible for recursion degree to increase under application of γ_2 (if clauses are added) and to decrease under application of ρ_2 (if clauses are removed). However, we can show the existence of a generalization σ for a set of schemata Π with the property that the recursion degree of σ is the minimum of the recursion degrees of the schemata in the set, regardless of their degrees of branching. This is the task of the following lemmas and theorem.

Lemma 9 gives a generalization of any single-clause schema with degree of recursion r.

Lemma 9. *Let κ be a schema, other than $\{\square\}$, of arity a with $bd(\kappa) = 1$ and $rd(\kappa) = r$. Then $\{P(X_{1,0}, \ldots, X_{a,0}) \text{ :- } \{P(X_{1,1}, \ldots, X_{a,1}), \ldots, P(X_{1,r}, \ldots, X_{a,r})\}\}$, where P is a predicate-variable and $X_{1,0}, \ldots, X_{a,r}$ are distinct individual-variables, is a generalization of κ.*

Proof. Apply the following operations to κ:

(1) For each predicate-constant p occurring in κ, replace p by a predicate-variable not already occurring in κ.

(2) For each individual-constant c occurring in κ, replace c by an individual-variable not already occurring in κ.

(3) If κ contains more than one occurrence of any variable V, replace the first occurrence of V by a variable W not already occurring in κ. Repeat this step until no variable occurs more than once in κ.

(4) κ now consists entirely of most-general literals. Let P be the predicate-variable occurring in the positive literal of κ. If there is a literal λ in κ which contains a predicate-variable other than P, delete λ from κ. Repeat this step until no predicate-variable other than P appears in κ.

Call the result of this sequence of operations κ'. By the derivation, κ' is a generalization of the original schema κ. Since

$$\kappa' \approx \{P(X_{1,0},\ldots,X_{a,0}) \text{ :- } \{P(X_{1,1},\ldots,X_{a,1}),\ldots,P(X_{1,r},\ldots,X_{a,r})\}\},$$

the lemma follows. □

Lemma 10 gives a generalization of any single-clause schema, independent of its degree of recursion.

Lemma 10. *Let κ be a schema, other than $\{\square\}$, of arity a with $bd(\kappa) = 1$. Then $\{P(X_1,\ldots,X_a)\}$, where P is a predicate-variable and $X_1,\ldots,X_a$ are distinct individual-variables, is a generalization of κ.*

Proof. By Lemma 9,

$$\sigma = \{P(X_{1,0},\ldots,X_{a,0}) \text{ :- } \{P(X_{1,1},\ldots,X_{a,1}),\ldots,P(X_{1,r},\ldots,X_{a,r})\}\}$$

is a generalization of κ. Since $P(X_{1,0},\ldots,X_{a,0})$ can be derived from σ by deleting r negative literals, the result follows. □

These lemmas can be extended to give a generalization of multi-clause schemata, as described in the next definition and theorem.

Definition. Define $G(a,b,r)$ to be the schema containing the clause

$$P(X_{1,1,0},\ldots,X_{a,1,0}) \text{ :- } \{P(X_{1,1,1},\ldots,X_{a,1,1}),\ldots,P(X_{1,1,r},\ldots,X_{a,1,r})\}$$

and, for $2 \leqslant i \leqslant b$, the clauses $P(X_{1,i,0},\ldots,X_{a,i,0})$. For example, $G(3,4,2)$ is

$$\begin{aligned}&\{P(X_{1,1,0},X_{2,1,0},X_{3,1,0}) \text{ :- } \{P(X_{1,1,1},X_{2,1,1},X_{3,1,1}),P(X_{1,1,2},X_{2,1,2},X_{3,1,2})\},\\ &P(X_{1,2,0},X_{2,2,0},X_{3,2,0}),\\ &P(X_{1,3,0},X_{2,3,0},X_{3,3,0}),\\ &P(X_{1,4,0},X_{2,4,0},X_{3,4,0})\}\end{aligned}$$

Observe that $G(a, b, r)$ is a schema of branching-degree b and recursion-degree r and defines a (variable) predicate of arity a.

Theorem 11. *Let $\pi_1, \dots, \pi_n$ be schemata not containing $\Box$ which define predicates of arity a. Then $G(a, \max\{bd(\pi_1), \dots, bd(\pi_n)\}, \min\{rd(\pi_1), \dots, rd(\pi_n)\})$ is a generalization of $\{\pi_1, \dots, \pi_n\}$.*

Proof. For each i, $1 \leqslant i \leqslant n$, let κ_i be a clause in π_i such that $rd(\kappa_i) = rd(\pi_i)$. By Lemma 9,

$$\sigma_1 = P(X_{1,1,0}, \dots, X_{a,1,0}) \text{ :- } \{P(X_{1,1,1}, \dots, X_{a,1,1}), \dots, P(X_{1,1,r}, \dots, X_{a,1,r})\},$$

where $r = \min\{rd(\pi_1), \dots, rd(\pi_n)\}$, is a generalization of $\{\kappa_1, \dots, \kappa_n\}$. For each i, let π'_i be the result of removing κ_i from π_i, and let c_i be the number of clauses in π'_i. By repeated use of Lemma 10,

$$\{P(X_{1,1,0}, \dots, X_{a,1,0}), \dots, P(X_{1,c_i,0}, \dots, X_{a,c_i,0})\}$$

is a generalization of π'_i; hence

$$\sigma_2 = \{P(X_{1,1,0}, \dots, X_{a,1,0}), \dots, P(X_{1,c,0}, \dots, X_{a,c,0})\},$$

where $c = \max\{c_1, \dots, c_n\} = \max\{bd(\pi_1), \dots, bd(\pi_n)\} - 1$, is a generalization of $\{\pi'_1, \dots, \pi'_n\}$. Thus,

$$G(a, \max\{bd(\pi_1), \dots, bd(\pi_n)\}, \min\{rd(\pi_1), \dots, rd(\pi_n)\}) = \sigma_1 + \sigma_2$$

is a generalization of $\{\pi_1, \dots, \pi_n\}$. $\Box$

Example. A generalization of *pre_order*:

$\{pre\_order(nil, [\,]),$
$\quad pre\_order(tree(Node, Left, Right), [Node|T]) \text{ :- }$
$\qquad \{pre\_order(Left, LL), pre\_order(Right, RL), append(LL, RL, T)\}\}$

and *flatten*:

$\{flatten([\,], [\,]),$
$\quad flatten([H1|T1], [H1|T2]) \text{ :- } \{atom(H1), flatten(T1, T2)\},$
$\quad flatten([[A|B]|T3], L \text{ :- }$
$\qquad \{flatten([A|B], L1), flatten(T3, L2), append(L1, L2, L)\}\}$

is $G(2, 3, 2)$:

$\{P(X_{1,1,0}, X_{2,1,0}) \text{ :- } \{P(X_{1,1,1}, X_{2,1,1}), P(X_{1,1,2}, X_{2,1,2})\},$
$\quad P(X_{1,2,0}, X_{2,2,0}),$
$\quad P(X_{1,3,0}, X_{2,3,0})\}$

The importance of Theorem 11 is that a generalization σ of a set of schemata can be produced directly, without any search; further, since the branching degree of σ is the maximum of the branching degrees of the schemata in the set, there is a least generalization which has exactly as many clauses as σ, and this least generalization can be derived from σ by applying only ρ_1. This gives an efficient algorithm for finding least generalizations, as shown in the next section.

3.3. An improved algorithm for finding least generalization

The results of the preceding section suggest a procedure for finding least generalizations: first find a generalization of appropriate branching degree and recursion degree, and then apply downward refinement (ρ_1) to that initial approximation until a least generalization is found. Algorithm 1 uses this approach to find a least generalization of a set of programs. The computation begins by taking $G(a, \max\{bd(\pi_1), \ldots, bd(\pi_n)\}, \min\{rd(\pi_1), \ldots, rd(\pi_n)\})$ as a first approximation σ, since we have shown that this will be a generalization of $\{\pi_1, \ldots, \pi_n\}$. ρ_1 is then applied to σ, yielding σ'; if σ' is also a generalization of $\{\pi_1, \ldots, \pi_n\}$, then σ' becomes the new approximation; otherwise, we retain σ. This process is repeated until ρ_1 can no longer be applied, at which point we will, by definition, have found a least generalization. A set *MARKED* is used to record past applications of ρ_1, to prevent needless repetition.

Algorithm 1. Derive a schema from a set of programs
Input: A set of programs $\Pi = \{\pi_1, \ldots, \pi_n\}$, each of which has arity a.
Output: A schema σ such that σ is a *least generalization* of the programs in Π.

```
Data structures:
  A set MARKED, whose elements are representations of applications of $\rho_1$. An
    element of MARKED will have one of the following forms, where $r_i$ will record
    an application of rule $i$ of $\rho_1$:
    $r_2(\nu_1, \nu_2)$, where $\nu_1$ and $\nu_2$ are individual-variables, representing the unification
      of $\nu_1$ and $\nu_2$.
    $r_3(P_1, P_2)$, where $P_1$ and $P_2$ are predicate-variables, representing the unification
      of $P_1$ and $P_2$.
    $r_4(\nu, f)$, where $\nu$ is an individual-variable and $f$ is a function symbol,
      representing the replacement of $\nu$ by a most-general term with functor $f$.
    $r_5(P, p)$, where $P$ is a predicate-variable, and $p$ is a predicate-constant,
      representing the replacement of $P$ by $p$.
  A set CONST_SET, containing constant and function symbols.
Procedure:
  CONST_SET $\leftarrow$ the set of all individual-constants, function symbols, and
    predicate-constants occurring in $\Pi$.
  $\sigma \leftarrow G(a, \max\{bd(\pi_1), \ldots, bd(\pi_n)\}, \min\{rd(\pi_1), \ldots, rd(\pi_n)\})$.
  MARKED $\leftarrow \emptyset$.
  repeat
    Select an application $\alpha$ of $\rho_1$ to $\sigma$ such that
```

```
        $\alpha$ is not a member of MARKED;
        every constant or function symbol introduced by $\rho_1$ is an element of
          CONST_SET; and
        every predicate-variable introduced by $\rho_1$ has an arity $b$ such that there is some
          predicate-constant in CONST_SET with arity $b$.
      Apply $\rho_1$ to $\sigma$ as determined in the previous step, producing $\sigma'$.
      MARKED $\leftarrow$ MARKED $\cup\ \alpha$.
      If $\sigma'$ is a generalization of $\{\pi_1, \ldots, \pi_n\}$, then $\sigma \leftarrow \sigma'$ (else leave $\sigma$ unchanged).
    until $\rho_1$ can no longer be applied to $\sigma$.
    return $\sigma$.
```

In order to discuss the efficiency of Algorithm 1, we introduce the following notation: if σ is a schema, the *length* of σ, written $|\sigma|$, will denote the number of non-punctuation symbols in σ. Similarly, if $\Pi = \{\pi_1, \ldots, \pi_n\}$ is a set of schemata, $|\Pi|$ will denote $|\pi_1| + |\pi_2| + \cdots + |\pi_n|$ (that is, the total number of non-punctuation symbols in Π).

The most difficult portion of the computation in Algorithm 1 is the comparison of two schemata σ_1 and σ_2 to determine whether $\sigma_1 \lesssim \sigma_2$. Chandra and Merlin [9] show that the graph isomorphism problem is polynomially reducible to the problem of determining whether two sets of first-order atomic formulas are identical to within renaming of variables; the latter problem is trivially reducible to the problem of determining, for two clauses κ_1 and κ_2, whether $\kappa_1 \approx \kappa_2$. This problem, in turn, reduces to the problem of comparing two schemata to determine whether one is a generalization of the other, since for κ_1, κ_2 not containing $\Box$, $\kappa_1 \approx \kappa_2$ iff $\kappa_1 \lesssim \kappa_2$ and $\kappa_2 \lesssim \kappa_1$. Since no polynomial-time algorithm is known for graph isomorphism—Garey and Johnson [14, pp. 154–158 and 285] conjecture that it belongs to a class of problems intermediate in difficulty between P and NP-complete problems—it is unlikely that a polynomial-time algorithm exists to determine in general whether one schema is a generalization of another.

If we consider the search space itself, however, we find that Algorithm 1 is efficient in the number of schemata it examines, as the next theorem shows.

Theorem 12. *Algorithm* 1 *examines* $O(|\Pi|^3)$ *schemata.*

Proof. For brevity of notation, let $b = \max\{bd(\pi_1), \ldots, bd(\pi_n)\}$ and $r = \min\{rd(\pi_1), \ldots, rd(\pi_n)\}$.

We begin by examining the number of ways in which ρ_1 can be applied to σ.

(1) A literal can be added to σ in $bd(\sigma) * A$ ways, where A is the number of different arities occurring among the predicate-constants in Π.

(2) There are $V_I * (V_I - 1)$ ways to unify individual-variables in σ, where V_I is the number of individual-variables occurring in σ.

(3) There are no more than $V_P * (V_P - 1)$ ways to unify predicate-variables in σ, where V_P is the number of predicate-variables occurring in σ.

(4) There are no more than $V_P * C$ ways to replace a predicate-variable in σ by a predicate-constant, where C is the number of predicate-constants occurring in Π.

(5) There are $V_I * F$ ways to replace an individual-variable in σ by a most-general term, where F is the number of function symbols and individual-constants occurring in Π.

Hence, for each σ which is a generalization of Π, the number of immediate successors of σ examined is at most

$$\begin{aligned} & bd(\sigma) * A + V_I * (V_I - 1) + V_P * (V_P - 1) + V_P * C + V_I * F \\ & \quad < |\sigma| * A + (V_I + V_P)^2 + |\sigma| * |\Pi| + |\sigma| * |\Pi| \\ & \quad \leqslant |\sigma| * A + |\sigma|^2 + 2|\sigma| * |\Pi|. \end{aligned}$$

By the way in which $G(a,b,r)$ is defined, there must be a program $\hat{\pi} \in \Pi$ such that $bd(\hat{\pi}) = b$. Since σ is derived from $G(a,b,r)$ using ρ_1, $bd(\sigma) = b = bd(\hat{\pi})$, and $\hat{\pi} \in \rho_1(\sigma)$. Applying ρ_1 to a schema either leaves the length of the schema unchanged (in the cases of unifying variables and replacing variables by constants) or increases the length of the schema (in the cases of adding literals and replacing variables by terms of arity greater than 0). Hence, we know that $|\sigma| \leqslant |\hat{\pi}| \leqslant |\Pi|$. Substituting this into our previous formula, the number of successors of σ is at most

$$|\Pi| * A + |\Pi|^2 + 2|\Pi| * |\Pi| = |\Pi| * A + 3|\Pi|^2.$$

By Property 7, the number of applications of ρ_1 needed to derive a least generalization from $G(a,b,r)$ is at most

$$\max_i\{\xi(|\pi_i|)\} - \xi(G(a,b,r)) \leqslant 2|\Pi|.$$

Multiplying this by the bound on the number of applications of ρ_1, and noting that $A \leqslant |\Pi|$, we see that the algorithm examines at most

$$(|\Pi|^2 + 3|\Pi|^2) * 2|\Pi| = 8|\Pi|^3$$

successors of $G(a,b,r)$, and thus examines $O(|\Pi|^3)$ schemata in all. □

Algorithm 1 runs quickly in actual elapsed time as well: In the examples discussed in Section 5, the least generalization of *cube_root* and *reciprocal* (the most difficult example) was found in 9 seconds, and the other least generalizations in Section 5 were all found in less than 2 seconds each.

3.4. Summary: Finding least generalizations

In Section 3, we have defined the concepts of degree of branching and degree of recursion and have described how they vary when γ_2 and ρ_2 are applied. These observations allowed us to construct an algorithm that is able to find a least generalization of a set of schemata in a two-step process of finding a generalization $G(a,b,r)$ immediately, then specializing $G(a,b,r)$ until a least generalization is found. We then showed that the number of nodes examined by this algorithm is a polynomial in the length of the input.

4. Finding programs

Now that we can find generalizations of programs, we turn to look at ways to use these generalizations in program synthesis. The synthesis problem we are considering is the following: *Given a set of positive examples and a set of negative examples, find a program that covers all of the positive examples and none of the negative examples.* Examples will be ground atoms; positive examples are ground atoms that the predicate being synthesized should succeed on, and negative examples are ground atoms that the predicate should not succeed on. We might, for instance, describe the desired behavior of a *union* program in this way:

Positive examples:

$union([a,b,c],[b,c,d],[a,b,c,d])$

$union([a,b],[c],[a,b,c])$

Negative examples:

$union([a,b,c],[b,c,d],[a,b,c])$

$union([a,b],[a],[\ ])$

We would then want to find a program that covers $union([a,b,c],[b,c,d],[a,b,c,d])$ and $union([a,b],[c],[a,b,c])$ but not $union([a,b,c],[b,c,d],[a,b,c])$ or $union([a,b],[a],[\ ])$.

We want to model a situation in which some programs are already known and can be called as subroutines from new programs, and in which we want to synthesize a single new program described by some examples. We will assume, therefore, that synthesis algorithms are provided with Prolog definitions of predicate constants other than the predicate being synthesized. (Most often, these predicate constants represent utility predicates such as *member* and *append*.)

4.1. A search algorithm for finding programs

Our first synthesis algorithm is Algorithm 2, which takes as input a schema σ and a set of positive and negative examples and produces as output a program π that covers all of the positive and none of the negative examples, if such a $\pi \in \rho_2 * (\sigma)$ exists. It generates specializations of σ until one is found which covers the proper examples. An oracle *COVERS* is used to determine whether a program produced in this search covers a given example, since that question is, in general, undecidable. (An implementation of *COVERS* which is adequate for practical purposes is discussed in Section 5.2.)

Algorithm 2. Find a program to fit a set of positive and negative examples, given a starting schema
Input:
A schema σ_0.
A set of positive examples $E^+ = \{e_1^+, \dots, e_n^+\}$ and a set of negative examples $E^- = \{e_1^-, \dots, e_m^-\}$.

```
    **Output:** A program which covers all examples in $E^+$ and none in $E^-$.
**Oracle used by algorithm:**
    $COVERS(\pi, e)$ returns $Y$ if $\pi$ is a program which covers $e$,
      and returns $N$ otherwise.
**Data structure:** A queue of schemata, $Q$.
**Procedure:**
    $Q \leftarrow [\sigma_0]$.
    while $Q$ is not empty
      Remove from $Q$ its first member $\sigma$.
      If $\sigma$ is not a program, then
        add to the end of $Q$ all members of $\rho_2(\sigma)$
      Else if $\sigma$ is a program and
          $COVERS(\sigma, e^+) = Y$ for all $e^+ \in E^+$ and
          $COVERS(\sigma, e^-) = N$ for all $e^- \in E^-$, then
        halt and return $\sigma$
      Else if $\sigma$ is a program and
          $COVERS(\sigma, e^+) = Y$ for all $e^+ \in E^+$ and
          $COVERS(\sigma, e^-) = Y$ for some $e^- \in E^-$, then
        add to the end of $Q$ all members of $\rho_2(\sigma)$.
```

Theorem 13. *Let E^+ be a set of positive examples, E^- be a set of negative examples, and σ_0 be a schema. Algorithm 2 with input E^+, E^-, and σ_0 will halt and return a program π covering all members of E^+ and no members of E^-, if such a $\pi \in \rho_2 * (\sigma_0)$ exists.*

Proof. Assume that there exists a program $\pi \in \rho_2 * (\sigma_0)$ which covers all members of E^+ and none of E^-. Any ancestor α of π either is not a program or is a program which covers all members of E^+, so if α is not a goal program and is selected from Q, all immediate descendants of α will be added to Q. Hence, if no goal program has yet been selected from Q, then there exists in Q an ancestor of π (possibly π itself). Because of this, if the algorithm halts, it will halt because it has found a goal program; that is, it will not halt because of exhausting Q without finding a goal.

Since the algorithm examines, in order of increasing n, the members of $\rho_2^n(\sigma_0)$, and since $\rho_2^n(\sigma_0)$ for any particular value of n is finite, there are a finite number of schemata which can be generated before π by this systematic application of ρ_2. Hence, either the algorithm will halt before finding π by selecting a goal program from Q earlier in the computation, or it will halt upon selecting π from Q. In either case, the computation halts and a program covering all members of E^+ and none of E^- is found. □

4.2. Schema hierarchy

In examining Prolog programs, we find that certain structural patterns recur: a program may consist of a recursive clause together with a base case, for instance, or it may contain a collection of non-recursive clauses which select one of several actions based on the truth of a condition. A programmer may go through the mental actions of selecting the basic form of a program (say, deciding that the program involves looping and hence requires a recursive clause and a non-recursive base clause) and then refine this form further as details of the required program behavior become clearer (for instance, deciding that the recursion will be on the tail of a list and that the base case is the empty list). Some of these structural patterns are described by schemata, with very general patterns being described by very general schemata and more specific patterns by more specific schemata.

The generalization operator γ_2 allows us to draw a directed acyclic graph of the schemata in a particular language, indicating which schemata are more general than others. Such a graph can be seen as classifying programs according to the schemata of which they are instances and as grouping programs as being relatively alike or dislike. γ_2 imposes a hierarchical structure on the entirety of a given schema-definition language. Very general schemata such as

$$\begin{aligned}&\{P(\_,\_,\_),\\ &\ P(\_,\_,\_),\\ &\ P(\_,\_,\_),\\ &\ P(\_,\_,\_)\}\end{aligned}$$

appear high up in the hierarchy. More specific schemata such as

$$\begin{aligned}&\{P([\,],[\,],[\,]),\\ &\ P([X|Z1],[X|Z2],[X|Z3]) :\!-\ \{P(Z1,Z2,Z3)\},\\ &\ P([Y|Z4],[W|Z5],[Y|Z6]) :\!-\ \{P(Z4,Z5,Z6)\},\\ &\ P([U|Z7],[V|Z8],[V|Z9]) :\!-\ \{P(Z7,Z8,Z9)\}\}\end{aligned}$$

occur lower in the hierarchy, programs such as

$$\begin{aligned}&\{\mathit{and}([\,],[\,],[\,]),\\ &\ \mathit{and}([1|Z1],[1|Z2],[1|Z3]) :\!-\ \{\mathit{and}(Z1,Z2,Z3)\},\\ &\ \mathit{and}([0|Z4],[W|Z5],[0|Z6]) :\!-\ \{\mathit{and}(Z4,Z5,Z6)\},\\ &\ \mathit{and}([1|Z7],[0|Z8],[0|Z9]) :\!-\ \{\mathit{and}(Z7,Z8,Z9)\}\}\end{aligned}$$

occur lower still, and programs consisting entirely of ground clauses such as

$$\begin{aligned}&\{\mathit{and}([\,],[\,],[\,]),\\ &\ \mathit{and}([1,1,1]],[1,0,1]],[1,0,1]]) :\!-\ \{\mathit{and}([1,1],[0,1],[0,1])\},\\ &\ \mathit{and}([0,0,1]],[0,1,0]],[0,0,0]]) :\!-\ \{\mathit{and}([0,1],[1,0],[0,0])\},\\ &\ \mathit{and}([1,1,0,1]],[0,1,1,1]],[0,1,0,1]]) :\!-\\ &\quad \{\mathit{and}([1,0,1],[1,1,1],[1,0,1])\}\}\end{aligned}$$

appear near the bottom.

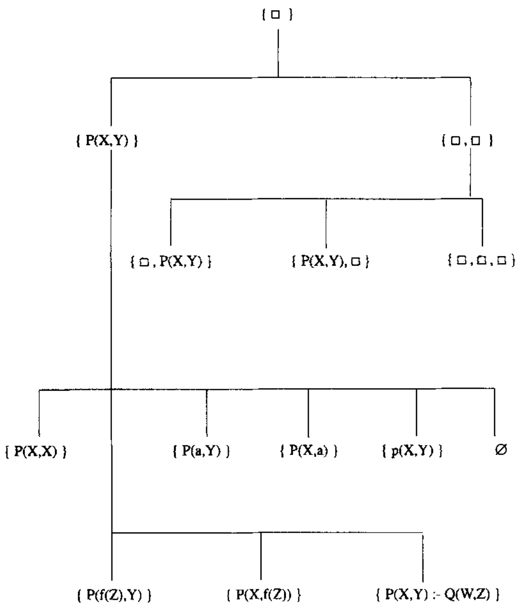


Fig. 1.

There are two types of hierarchical graphs we can construct using γ_2. The first is the complete graph containing as nodes all the schemata expressible in a given schema-definition language, with each arc representing a single application of γ_2. The first three levels of the complete graph for the language containing a function f^1, a constant a, and a predicate constant p^2, are shown in Fig. 1.

A second type is obtained by selecting nodes from the complete graph and allowing arcs to represent one or more applications of γ_2. These nodes may be selected according to aesthetic criteria (selecting schemata which represent "important" groupings of programs), computational criteria (giving a graph of a desired depth or branching factor), or for other reasons. This gives us a relatively small graph which classifies schemata according to similarity, using a generalization relation.

We see some of the more significant patterns for binary predicates in Fig. 2. The schema labelled *b*1 is the most general 4-clause schema of arity 2. *b*2 is more specific than *b*1 because one of its clauses contains a recursive call. Replacing a variable by a most-general term produces *b*3, and adding a second recursive call in the recursive clause produces *b*4, a schema which has as specializations doubly-recursive programs such as *flatten* and some tree-traversal programs (*pre_order*, *in_order*, *post_order*, and *leaf_list*). *b*5 represents recursion on the tail of a list, and two special cases of this are *b*7, with the empty list as a base case, and *b*6, with a singleton list as the base. *b*9 is a specialization of *b*7, containing two recursive clauses, and *b*8 describes two-clause list-traversal programs.

Fig. 3 shows a graph of schemata for predicates of arity 3. The most general 4-clause schema, *s*1, appears at the top. *s*2, a schema describing selection of one of two input

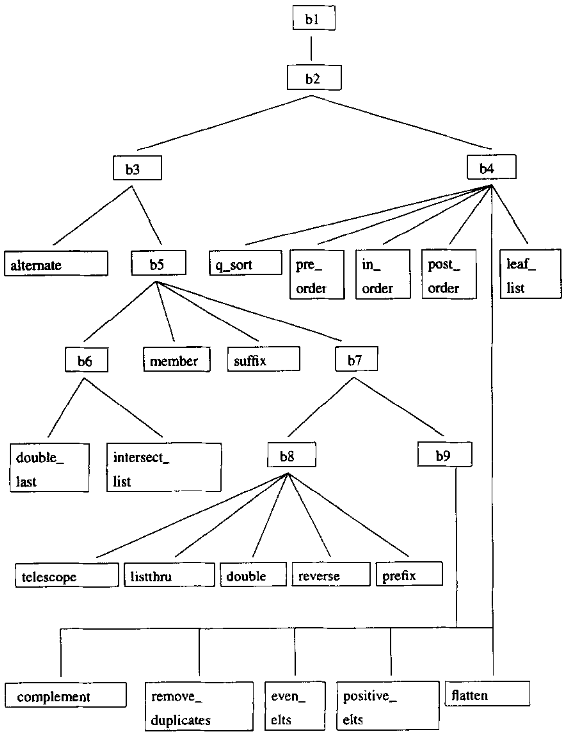


Fig. 2. Hierarchical graph for selected schemata of arity 2. The full definitions of the programs and schemata in this diagram appear in Appendix B.

parameters based on a test, is a generalization of *max* and *min*. *s*3, a schema containing a recursive call, has as descendants the programs *monus*, *int_div*, and *mod*, and schemata *s*4 and *s*5. *s*4 is a schema containing a clause with two recursive calls. *s*5, a schema representing recursion on the tail of a list, is a generalization of *s*7, in which recursion occurs on two lists; *s*7 is, in turn, a generalization of *s*9, with recursion on three lists. *s*5 is also a generalization of *s*6, in which one of the input parameters appears both on the left-hand side and the right-hand side of the clause. Unification of individual-variables and replacement of individual-variables by most-general terms produces *s*8 and *s*10, in turn, from *s*6. The diagram illustrates that *intersection* is more like *delete_all* in form than it is like *append*, and *intersection* resembles *append* more closely than it does *adjacent* or *and*.

4.3. Using a schema hierarchy for program synthesis

A schema-hierarchy graph makes synthesizing a new program easier, because it gives a collection of schemata from which to begin the search. By choosing a very general schema (such as $\{\Box\}$ or $\{P(\_,\_,\_), P(\_,\_,\_), P(\_,\_,\_)\}$) for the root of the graph, we retain enough generality in the search to ensure finding a solution. But by including

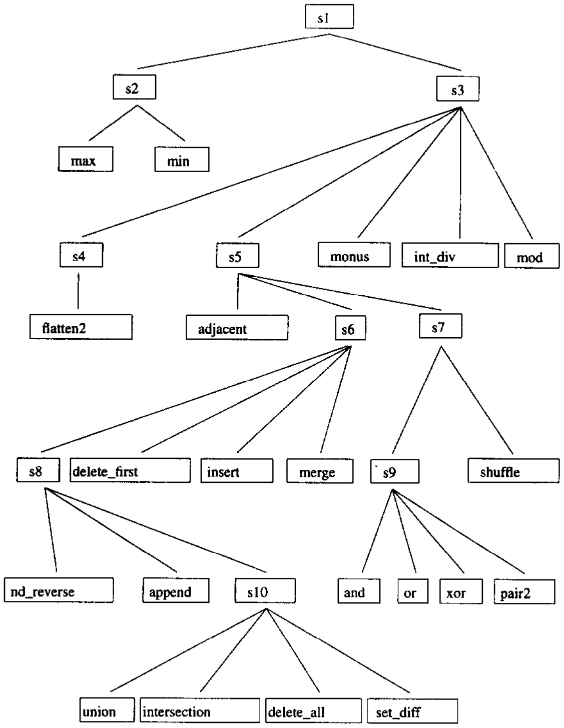


Fig. 3. Hierarchical graph for selected schemata of arity 3. The full definitions of the programs and schemata in this diagram appear in Appendix C.

other, more specific schemata in the graph, we concentrate the search around patterns known to be useful program generalizations. A well-selected set of schemata will thus greatly narrow the search.

Schema-hierarchy graphs are directed acyclic graphs consisting of nodes representing schemata and arcs representing one or more applications of ρ_2. While a schema-hierarchy graph is not necessarily a tree, we will consider ones that are sufficiently tree-like so as to contain exactly one node of in-degree 0, which we will call the root, and one or more nodes of out-degree 0, which we will call leaves. Conceivably, a graph could contain all the nodes intermediate in generality between the root and the set of leaves, but more often it will contain a small subset of these nodes which have been deemed to be "interesting".

Since the full power of ρ_2 can produce lengthy but relatively uninformative sequences of refinements (e.g., addition of a number of literals to a clause κ, followed by deletion of κ), we will restrict our attention in Sections 4.3.1 and 4.3.2 to graphs whose arcs correspond to one or more applications of ρ_1 only—hence, to graphs all of whose nodes have the same degree of branching. We will further impose the condition that for any σ_1 and σ_2 in a graph Γ, if σ_1 is a generalization of σ_2, then there is a path from σ_1 to σ_2 in Γ.

We will also need a definition for speaking about generalizations in a graph.

Definition. Let Σ be a set of schemata and Γ be a hierarchical graph of schemata. If a schema σ in Γ is a generalization of Σ, and if there is no specialization σ' of σ in Γ such that σ' is a generalization of Σ, then we say that σ is a *least generalization of Σ relative to Γ.*

Example. If Γ is the graph in Fig. 3, then schema $s10$ is the least generalization of *union* and *intersection* relative to Γ, and $s5$ is the least generalization of *adjacent, append,* and *xor* relative to Γ.

4.3.1. Expanding a schema hierarchy

In order to construct and maintain graphs with the properties we have described, we need to be able to add new programs and schemata to the graphs. To state this formally: given a schema σ and a hierarchical graph Γ which does not contain σ, we want to insert σ into Γ, preserving the property that for any nodes σ_1 and σ_2 in Γ, if σ_1 is a generalization of σ_2 then there is a path from σ_1 to σ_2 in Γ.

The algorithm below allows a set of schemata to be added to a schema-hierarchy graph. The ability to add a group of schemata at once is useful if, for instance, we have several new programs which are known to be closely related and which we would like to have grouped under a parent node representing their least generalization.

add_schema_set begins by finding a least generalization *LG* of the set of new schemata. *LG* is added to the graph, with the new schemata as *LG*'s children. We may have to establish a new root for the graph: if the root is a generalization of *LG* or vice versa, then the more general of the two is the root of the new graph; otherwise, a least generalization of *LG* and the root is added as the new root. To add individual schemata, *add_schema_set* calls *add_schema*, which finds all the relative least generalizations and most general specializations of the new schema in the graph and adds the appropriate arcs to the graph.

Algorithm 3. Add a set of new schemata to a schema-hierarchy graph
Input:

A schema-hierarchy graph *Graph*, rooted at *Root*
A set of schemata *Schema_Set* = $\{\sigma_1, \ldots, \sigma_n\}$, none of which are in *Graph*

Output:

A schema-hierarchy graph *New_Graph* which contains $\sigma_1, \ldots, \sigma_n$ and all the nodes of *Graph*

Procedure:

```
% add_schema_set(Root, Schema_Set, New_Graph):
% Add the schemata in Schema_Set to the graph rooted at Root, yielding New_Graph

  if Schema_Set is a singleton set $\{\sigma_1\}$, then
    add $\sigma_1$ to Graph, using add_schema
    return resulting graph as New_Graph
  else
```

```
    compute one least generalization $LG$ of $Schema\_Set$
    **if** $LG$ is a specialization of $Root$ **then**
      add $LG$ to $Graph$, using $add\_schema$
      add each member of $Schema\_Set$ to $Graph$, using $add\_schema$
    **else**
      split $Schema\_Set$ into
        $\Sigma_1$ = the set of schemata in $Schema\_Set$ which are specializations of $Root$
        $\Sigma_2$ = the set of schemata in $Schema\_Set$ which are not specializations
          of $Root$
      individually add to $Graph$ all members of $\Sigma_1$, using $add\_schema$
      compute a least generalization $LG_2$ of $\Sigma_2$
      compute a least generalization $New\_Root$ of $\{Root, LG_2\}$
      add $New\_Root$ to $Graph$ as the parent of $Root$
      **if** $New\_Root$ is identical to within renaming of variables to $LG_2$ **then**
        add the schemata in $\Sigma_2$ to $Graph$ as the children of $New\_Root$
      **else**
        add $LG_2$ to the graph as a child of $New\_Root$
        add the schemata in $\Sigma_2$ to $Graph$ as children of $LG_2$
      **return** resulting graph as $New\_Graph$

% $add\_schema(Root, Schema, New\_Graph)$:
% Add $Schema$ to the graph rooted at $Root$, yielding $New\_Graph$

  add the node $Schema$ to the set of nodes in $Graph$
  $find\_relative\_lgs(Root, Schema, ListG)$
  **for** each $\sigma \in ListG$
    establish arc from $\sigma$ to $Schema$ in $Graph$
  $find\_specializations(Root, Schema, ListS)$
  **for** each $\sigma \in ListS$
    establish arc from $Schema$ to $\sigma$ in $Graph$
  **if** $Root$ and $Schema$ are incomparable, **then**
    compute a least generalization $LG$ of $\{Root, Schema\}$
    add $LG$ to $Graph$ as the parent of $Root$ and $Schema$
    **return** resulting graph as $New\_Graph$

% $find\_relative\_lgs(Root, Schema, List)$:
% Find all least generalizations of $Schema$ relative to the graph rooted at $Root$;
% put these in $List$.

  **if** $Root$ is a generalization of $Schema$, **then**
    $L1 \leftarrow [\ ]$
    **for** every child $C$ of $Root$
      $find\_relative\_lgs(C, Schema, L2)$
      append $L2$ to $L1$
    **if** $L1 = [\ ]$ **then** $List \leftarrow [Root]$
    **else** $List \leftarrow L1$
  **else** $List \leftarrow [\ ]$
```

```
% find_specializations(Root, Schema, List):
% Find all most general specializations of Schema relative to the graph rooted at Root;
% put these in List.

  if Root is a specialization of Schema, then
    List ← [Root]
  else
    List ← [ ]
    for every child C of Root
      find_specializations(C, Schema, L2)
      append L2 to List
      remove from new List all schemata that have an ancestor from the graph
        appearing in List
```

4.3.2. Program synthesis

Algorithm 2 used breadth-first search from a single starting schema to find a program to fit a set of positive and negative examples. Algorithm 4 uses an alternative approach: it begins with a schema-hierarchy graph rather than a single schema, and it uses a bounded depth-first search to keep the search focused. As in Algorithm 2, we assume the existence of an oracle $COVERS(\pi, e)$, which, for a given program π and example e, returns Y if π covers e and N otherwise.

Algorithm 4. Find a program to fit a set of positive and negative examples, using a hierarchical graph of schemata.

Input:

A hierarchical graph of schemata, *Graph*, rooted at *Root*
A set of positive examples, E^+, and a set of negative examples E^-
A nonnegative integer D

Output:

A program within D steps of some node in *Graph* which covers all examples in E^+ and none in E^-, if such a program exists.

Data structure:

A stack of $(schema, depth)$ pairs, S

Procedure:

```
for each Node in Graph,
  $S \leftarrow [(Node, 0)]$
  while $S$ is not empty
    Pop from $S$ its top member $(\sigma, D_\sigma)$
    if $D_\sigma \leqslant D$ then
      if $\sigma$ is not a program, then
        for every schema $\sigma'$ in $\rho_2(\sigma)$,
          push $(\sigma', D_\sigma + 1)$ onto $S$
      else if $\sigma$ is a program and
          $COVERS(\sigma, e^+) = Y$ for all $e^+ \in E^+$ and
          $COVERS(\sigma, e^-) = N$ for all $e^- \in E^-$, then
        halt and return $\sigma$
    else if $\sigma$ is a program and
        $COVERS(\sigma, e^+) = Y$ for all $e^+ \in E^+$ and
        $COVERS(\sigma, e^-) = Y$ for some $e^+ \in E^-$, then
        for every schema $\sigma'$ in $\rho_2(\sigma)$,
            push $(\sigma', D_\sigma + 1)$ onto $S$
```

5. A system for learning and using schemata

This section describes a system implementing the main ideas of the preceding sections, illustrated by examples of schema inference and program synthesis as computed by this system.

5.1. Description of the system

The system is composed of three modules, each of which takes its input from and writes its output to one or more files: the modules communicate by means of these files.

(a) *Find least generalization*
Input:

A set of programs $\Pi = \{\pi_1, \ldots, \pi_n\}$

Output:

A least generalization σ of Π

(b) *Add program set*
Input:

A schema-hierarchy graph Γ_1

A set of programs $\Pi = \{\pi_1, \ldots, \pi_n\}$, none of which is in Γ_1

Output:

A new schema-hierarchy graph Γ_2 which contains $\pi_1, \ldots, \pi_n$

(c) *Find program*
Input:

A set of positive examples E^+ and a set of negative examples E^-

A schema-hierarchy graph Γ

A nonnegative integer D

Output:

A program which covers all of the examples in E^+ and none of the examples in E^- (if such a program exists)

The system makes long-term use of one or more files of schemata, structured into a hierarchical graph. In general, all known schemata defined with a given alphabet could be stored in a single file; more practically, the schemata could be grouped into a number of different files according to such features as arity, degree of branching, or type of problem, with a human user selecting the file that is most likely to contain a useful schema. As in Section 4.3, we will assume that all schema-hierarchy graphs have the property that for any σ_1 and σ_2 in the graph, if σ_1 is a generalization of σ_2, then there is a path from σ_1 to σ_2 in the graph. The algorithm used for creating and adding to schema-hierarchy graphs maintains this property.

The learning cycle consists of repeatedly adding new programs to the files of known schemata, in one of two ways:

(1) The system may be told directly about a program or a set of related programs. The new programs are added to the graph with the *add program set* module.

(2) The system may be asked to produce a program for a set of positive and negative examples. In this case, the *find program* module is used. Once the new program has been found, it can be added to the original hierarchical graph with *add program set*.

5.2. Implementation

All three modules have been implemented in SICStus Prolog and run on a Silicon Graphics Indy.

The *find least generalization* module is an implementation of Algorithm 1. The *add program set* module uses Algorithm 3 to add a set of new programs to the graph.

The *find program* module implements Algorithm 4. It takes as input a set of positive examples, a set of negative examples, a graph Γ, and an integer D, and it returns a program π within D ρ_2-steps of some node in Γ such that π covers all positive examples and no negative examples. Note that the special case of $D = 0$ asks us to find a π among the known schemata in Γ. In general, however, with $D > 0$, we will be looking for a program that the system has not seen before.

Since the problem of determining whether a program covers a particular example is, in general, undecidable, the oracle $COVERS(\pi, e)$ is approximated by a predicate which runs the program for a predetermined number of steps and then reports that the program has succeeded, (finitely) failed, or failed to halt in the allotted time. A program that does not halt on one of the examples (assuming it does not finitely fail on any positive example) will be treated as being too general, and will be refined to produce future candidate programs.

5.3. A sample problem

As an illustration, we will look at the generalization and synthesis of a sequence of list-processing problems as performed by the system.

We start with the two programs *double*:

$$\{double([\,],[\,]),$$
$$double([H|T1],[H,H|T2]) \text{ :- } \{double(T1,T2)\}\}$$

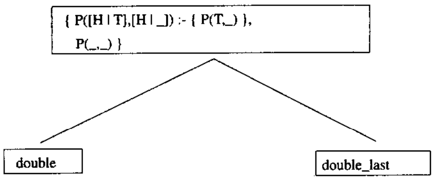
and *double_last*:

$\{double\_last([X],[X,X]).$
$double\_last([H|T1],[H|T2])$:- $\{\backslash{==}(T1,[\,]), double\_last(T1,T2)\}\}$

and ask for a least generalization. The system will find the schema

$\{P([H|T],[H|\_])$:- $\{P(T,\_)\},$
$P(\_,\_)\}$

From this we can form the graph



Suppose we then want to synthesize a program for finding prefixes of lists. Taking as input the positive examples

$prefix([a,b],[a,b,c]),\quad prefix([a],[a,b,c]),\quad prefix([\,],[a,b,c]),$
$prefix([a,b,c],[a,b,c]),\quad prefix([c],[c]),\quad prefix([b],[b,a]),$

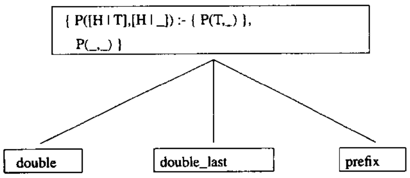
the negative examples

$prefix([b,c],[a,b,c]),\quad prefix([c],[a,b,c]),\quad prefix([a,b,c],[a,b]),$

and our first graph, with a search depth of 5, the system finds the program

$\{prefix([\,],\_),$
$prefix([H|T1],[H|T2])$:- $\{prefix(T1,T2)\}\}$

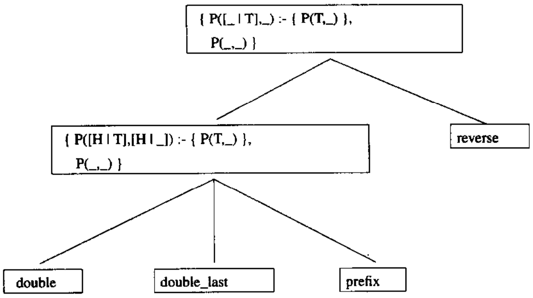
We can then add *prefix* to our graph, producing the new graph



Supposing that we are now given the program *reverse*:

$\{reverse([\,],[\,]),$
$reverse([H|T],L)$:- $\{reverse(T,TR), append(TR,[H],L)\}\},$

we can add *reverse* to our graph, producing


{ P([_ | T],_) :- { P(T,_) },
P(_,_) }
{ P([H | T],[H | _]) :- { P(T,_) },
P(_,_) }
reverse
double
double_last
prefix

We can use this new graph to synthesize a program for finding suffixes of a list, using positive examples

$$\textit{suffix}([a,b,c],[b,c]),\quad \textit{suffix}([a,b,c],[c]),\quad \textit{suffix}([a,b,c],[\,]),$$
$$\textit{suffix}([a,b,c],[a,b,c]),\quad \textit{suffix}([c],[c]),\quad \textit{suffix}([b,a],[a]),$$

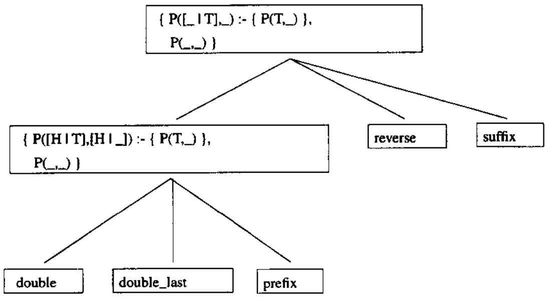
negative examples

$$\textit{suffix}([a,b,c],[a,b]),\quad \textit{suffix}([a,b,c],[b]),\quad \textit{suffix}([a,b],[a,b,c]),$$

and a depth of 3; the system will find the program

$$\{\textit{suffix}(X,X),$$
$$\textit{suffix}([\_|T],Y)\ :-\ \{\textit{suffix}(T,Y)\}\}$$

When the suffix program is added to the graph, our final graph becomes


{ P([_ | T],_) :- { P(T,_) },
P(_,_) }
{ P([H | T],[H | _]) :- { P(T,_) },
P(_,_) }
reverse
suffix
double
double_last
prefix

5.4. Numeric examples

This system is not designed with numeric predicates in mind, because numeric computation in Prolog is sensitive to the order in which literals appear. For example, in Prolog, the pair of literals X *is* $2+2$, $X > 0$ will succeed, but in the reverse order ($X > 0$, X *is* $2+2$) they will cause a run-time error if X is not already instantiated to a value from prior context. Since our formalism treats all programs and schemata as order-independent, it cannot guarantee that output programs will contain literals that are ordered properly for a Prolog compiler. However, some numeric programs can be synthesized.

5.4.1. Numeric list examples

For a simple example, the predicates

$\%\ sum\_squares(List, Ssq)$: Ssq is the sum of the squares of the elements in $List$

$\{sum\_squares([\ ], 0),$

$sum\_squares([H|T], Ssq)$:- $\{sum\_squares(T, TS), Ssq$ is $TS + H * H\}\}$

and

$\%\ prod\_list(List, Product)$: $Product$ is the product of the elements in $List$

$\{prod\_list([\ ], 1),$

$prod\_list([H|T], Product)$:-

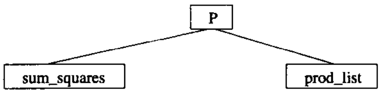
$\{prod\_list(T, TProduct), Product$ is $TProduct * H\}\}$

have the least generalization

$\{P([\ ], \_),$

$P([\_|T], X)$:- $\{P(T, \_), X$ is $\_\ \}\}$

The system constructs the graph



From this new schema, predicates *sum_list* and *list_length* can be synthesized. Using positive examples

$sum\_list([1,2,3,4], 10)$, $sum\_list([10,5], 15)$,

$sum\_list([1], 1)$, $sum\_list([\ ], 0)$,

and negative examples

$sum\_list([1,2,3,4], 11)$, $sum\_list([10,5], 10)$, $sum\_list([1], 0)])$,

the system synthesizes the program

$\{sum\_list([\ ], 0),$

$sum\_list([H|T], X)$:- $\{sum\_list(T, Y), X$ is $H + Y\}\}$

Starting from the same schema, and using positive examples

$list\_length([1,2,3,4],4)$, $list\_length([10,5],2)$,
$list\_length([1],1)$, $list\_length([\ ],0)$,

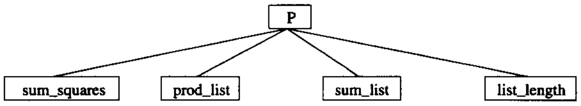
and negative examples

$list\_length([1,2,3,4],3)$, $list\_length([10,5],10)$,
$list\_length([10,5],5)$, $list\_length([1],0)$,

the system synthesizes the program

$$\{list\_length([\ ],0),$$
$$list\_length([\_|T],X) \text{ :- } \{list\_length(T,Y), X \text{ is } Y+1\}\}$$

The resulting graph is



5.4.2. Successive approximation example

A third example begins with a pair of predicates, *cube_root* and *reciprocal*, which are based on a successive approximation algorithm described in [12]. The *cube_root* program takes three parameters:

- N, the number we are taking the cube root of;
- R, an uninstantiated parameter which will hold the root at the end of the calculation;
- *Range*, a number indicating the desired precision in the answer. The root found will be such that N is between R^3 and $(R + Range)^3$.

For example, $cube\_root(8, 0.001, R)$ will find an R between 2.0 and 2.001.

The program proceeds by doubling *Range* in each recursive call until $Range > N$; thus, in the recursive calls, we have the sequence

$$Range,\quad 2*Range,\quad 4*Range,\quad 8*Range,\quad \ldots.$$

As execution backs out of the recursion, numbers in the sequence are either added to an accumulating sum or not, depending on whether the cube of the sum is less than or equal to N. For $cube\_root(3, 0.1, R)$, for example, the recursive calls are

$$\begin{gathered} cube\_root(3,0.1,R), \\ cube\_root(3,0.2,R), \\ cube\_root(3,0.4,R), \\ \vdots \\ cube\_root(3,3.2,R). \end{gathered}$$

The root found in this case will be $1.4 = 0.8 + 0.4 + 0.2$. Note that $1.4^3 \leqslant 3 < (1.4+0.1)^3$, so that 1.4 satisfies the precision requirements for the root.

This algorithm can also be used to find integer cube roots (that is, the largest integer less than or equal to N^{-3}) by using a *range* value of 1. *cube_root*$(9, 1, R)$, for example, finds the integer root 2.

```
% cube_root(N, Range, R)
% Find $R$ = cube root of $N$; more specifically, find $R$ such that
%   $R^3 \leqslant N < (R + Range)^3$
%   (thus $|cube\_root(N) - R| < Range$).
% Call with $N \geqslant 0$, $Range$ = desired precision interval (1 for ints),
%   R uninstantiated, initially.

{cube_root(N1, Range1, 0) :- {Range1 > N1},
 cube_root(N, Range, R) :- {Range =< N,
  Range2 = Range * 2,
  cube_root(N, Range2, R2),
  Square = (R2 + Range) * (R2 + Range) * (R2 + Range),
  inc_amount(Square =< N, Range, Inc),
  R = R2 + Inc}}
```

The *reciprocal* program uses the same approach as *cube_root*:

```
% reciprocal(N, Range, R)
% Find $R$ = reciprocal of $N$; more specifically, find $R$ such that
%   $1/R \leqslant N < 1/(R + Range)$
%   (thus $|reciprocal(N) - R| < Range$).
% Call with $N \geqslant 0$, $Range$ = desired precision interval (1 for ints),
%   R uninstantiated, initially.

{reciprocal(N1, Range1, 0) :- {Range1 > N1},
 reciprocal(N, Range, R) :- {Range =< N,
  Range2 = Range * 2,
  reciprocal(N, Range2, R2),
  Product = N * (R2 + Range),
  inc_amount(Product =< 1, Range, Inc),
  R = R2 + Inc}}
```

(The utility predicate *inc_amount*, called by both *cube_root* and *reciprocal*, is:

```
% inc_amount(+Condition, +Quantity; -Increment)
% Increment is Quantity if Condition is true, 0 if Condition is false.

inc_amount(Condition, Quantity, Quantity) :- Condition.
inc_amount(Condition, _, 0) :- \+Condition.   )
```

The system finds the least generalization

$\{P(N1, Range1, 0)$:- $\{Range1 > N1\},$
$P(N, Range, R)$:- $\{Range \leqslant N,$
$Range2 = Range * 2,$
$P(N, Range2, R2),$
$X = \_ * (R2 + Range),$
$inc\_amount(X \leqslant \_, Range, Inc),$
$R = R2 + Inc\}\}$

which captures the essence of the successive approximation algorithm used by both programs.

A square root program can now be synthesized from this schema. Using positive examples

$sq\_root(9,1,3),\quad sq\_root(25,1,5),\quad sq\_root(1,1,1),$
$sq\_root(0,0,0),\quad sq\_root(100,1,10),$

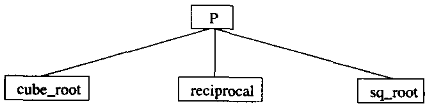
and negative examples

$sq\_root(9,1,0),\quad sq\_root(9,1,1),\quad sq\_root(9,1,2),\quad sq\_root(9,1,4),$

the system finds the program

$\{sq\_root(N1, Range1, 0)$:- $\{Range1 > N1\},$
$sq\_root(N, Range, R)$:- $\{Range \leqslant N,$
$Range2 = Range * 2,$
$sq\_root(N, Range2, R2),$
$X = (Range + R2) * (R2 + Range),$
$inc\_amount(X \leqslant N, Range, Y),$
$R = R2 + Y\}\}$

The resulting schema graph is



6. Conclusions

The enumerative search approach to automatic programming carries both benefits and liabilities. The chief benefit of enumerative search is the thoroughness with which it covers an easily-described space, allowing us to prove theorems about the circumstances under which it is, and is not, guaranteed to locate a target successfully. This thoroughness is also the main weakness of the approach, in that enumerative search generally examines an exponentially large number of possibilities. We have discussed two mechanisms for narrowing the search, (1) the use of upward and downward refinement operators, to avoid having to examine the entire space of syntactically correct programs, and (2) the use of schemata as a starting point for the search for a program, to avoid having to examine the entire space generated by the refinement operators.

This paper has described an algorithm for finding least generalizations of programs efficiently. It has also described a system which stores a collection of these least generalizations (or schemata) and uses them as a starting point for program synthesis. By starting with a particular schema, the system can derive a program to fit the input/output specifications much more quickly than it could have if it did not have the information provided by the schema. As the system derives more and more programs over its lifetime, it can add more and more schemata to its collection, thus enabling the efficient synthesis of an increasingly broad range of programs.

Acknowledgements

This paper summarizes doctoral research done at Duke University. I am grateful to my advisor, Alan Biermann, and to Pierre Flener, Donald Loveland, Gopalan Nadathur, David Plaisted, Bilge Karal Say, Joseph Shoenfield, and anonymous reviewers for their helpful comments on this research and on drafts of this paper.

Appendix A. Glossary

Numbers in parentheses indicate the section in which the term is introduced.

γ_1: An upward refinement operator. (2.2)
γ_2: An upward refinement operator. (2.2)
ρ_1: A downward refinement operator. (2.2)
ρ_2: A downward refinement operator. (2.2)
$\approx$: 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. (2.2)
$\lesssim$: A partial order on schemata. (2.2)
$C_I(\sigma)$: The set of goals covered by schema σ in interpretation I. (2.2)
$\xi(\sigma)$: The number of non-punctuation symbols in σ—the number of distinct variables in σ+ the number of literals in σ. (2.3)
$\{V\backslash t\}$: A substitution replacing all occurrences of variable V by term t. (2.2)
\==: A built-in Prolog predicate. X \== Y if X and Y are not identical.
$\uplus$: As a multiset operator, $\{a,b,b\} \uplus \{a,a,a,c\} = \{a,a,a,b,b,c\}$.
$\cap$: As a multiset operator, $\{a,a,b,b\} \cap \{a,a,a,c\} = \{a,a\}$.
$\subseteq$: As a multiset operator, $\{a,b,b\} \subseteq \{a,b,b,c\}$, and $\{a,b,b\} \subseteq \{a,b,b\}$, but $\{a,b,b\} \not\subseteq \{a,b,c\}$.

$\subset$: As a multiset operator, $\{a,b,b\} \subset \{a,b,b,c\}$, but $\{a,b,b\} \not\subset \{a,b,b\}$, and $\{a,b,b\} \not\subset \{a,b,c\}$.

$+$: As a multiset operator, $\{a,b,b\} + \{a,a,a,c\} = \{a,a,a,a,b,b,c\}$.

$-$: As a multiset operator, $\{a,a,a,a,b,b,c\} - \{a,b,b\} = \{a,a,a,c\}$.

$bd(\sigma)$: The *degree of branching* of schema σ. (3.2)

Constant set: A finite set of functions symbols, individual-constants, and predicate constants. (2.1)

Degree of branching: The number of clauses in a schema. (3.2)

Degree of recursion: If σ is a schema such that every positive literal in σ contains the (variable or constant) predicate symbol ϕ, then the degree of recursion of σ is the maximum of $\{n \mid$ there is a clause κ in σ whose negative literals contain exactly n occurrences of $\phi\}$. (3.2)

Downward refinement operator: Given an ordering relation $\leqslant$ on expressions, an operator ρ is a downward refinement operator if ρ is a recursively enumerable relation on expressions such that ρ^* is $\leqslant^{-1}$. (2.2)

Generalization of a schema: Schema σ_1 is a generalization of schema σ_2 if there is a schema σ' such that $\sigma_1 \approx \sigma'$ and $\sigma' \in \gamma_2 * (\sigma_2)$. (2.2)

Generalization of a set: A schema σ is a generalization of a set of schemata Π if σ is a generalization of every schema in Π. (2.2)

Least generalization: A schema σ is a least generalization of a set of schemata Π if σ is a generalization of Π and there is no schema σ' such that σ' is a specialization of σ and such that σ' is a generalization of Π.

Multiset: A collection of objects in which repetition is significant, but, as in a set, order is not significant. See also the definitions for $\cup$, $\cap$, $\subseteq$, $\subset$, $+$, and $-$, and [22]. (2)

Program: A schema containing no predicate-variables and not containing $\Box$. (2.1)

$rd(\sigma)$: The *degree of recursion* of schema σ. (3.2)

Refinement operator: See *upward refinement operator*, *downward refinement operator*.

Schema: A multiset of clauses such that there is only one predicate symbol appearing in the positive literals of the clauses. (2.1)

Schema-definition language over (a constant set) K: If A is the set of all integers a such that there is a predicate constant in K of arity a, then L is 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 s occurring in σ is an element of K, and every predicate variable P occurring in σ has an arity a_P such that $a_P \in A$. (2.1)

Specialization of a schema: Schema σ_1 is a specialization of schema σ_2 if there is a schema σ' such that $\sigma_1 \approx \sigma'$ and $\sigma' \in \rho_2 * (\sigma_2)$. (2.2)

Upward refinement operator: Given an ordering relation $\leqslant$ on expressions, an operator γ is an upward refinement operator if γ is a recursively enumerable relation on expressions such that γ^* is $\leqslant$. (2.2)

Appendix B. Schemata of arity 2

Listed below are the schemata and programs mentioned in Fig. 2. To make the programs easier to read, a few lines of comments are included. The symbols "+",

"–", and "?" preceding parameter names in a comment indicate, respectively, an input parameter, an output parameter, and a parameter that can be either. In many places mnemonic constant and variable names are used for readability, rather than adhering strictly to the single-letter names used elsewhere.

```
% b1
{P(_,_),
 P(_,_),
 P(_,_),
 P(_,_)}

% b2
{P(_,_),
 P(_,_) :- {P(_,_)},
 P(_,_),
 P(_,_)}

% b3
{P(_,_),
 P([_|_]) :- {P(_,_)},
 P(_,_),
 P(_,_)}

% b4
{P(_,_),
 P(_,_) :- {P(_,_),P(_,_)},
 P(_,_),
 P(_,_)}

% b5
{P(_,_),
 P([_|T],_) :- {P(T,_)},
 P(_,_),
 P(_,_)}

% b6
{P([_],_),
 P([_|T],_) :- {P(T,_)}}

% b7
{P([ ],_),
 P([_|T],_) :- {P(T,_)},
 P(_,_),
 P(_,_)}

% b8
{P([ ],_),
 P([_|T],_) :- {P(T,_)}}

% b9
{P([ ],_),
P([_ T1],_) :- {P(T1,_)},
P([_ T2],_) :- {P(T2,_)}}

% alternate(+List1, –List2)
% List2 contains the first, third, ..., elements of List1.
% For example, alternate([a,b,c,d,e],[a,c,e]).
% Adapted from [33, p.266].
{alternate([ ],[ ]),
 alternate([X,_|T1],[X|T2]) :- {alternate(T1,T2)}}

% member(?L,?X): X is a member of list L.
{member([X|_],X),
 member([_|T],X) :- {member(T,X)}}

% suffix(?S,?L): S is a suffix of list L.
```

```
{suffix(L, L),
 suffix([_|T], L) :- {suffix(T, L)}}

% double_last(+List1, -List2)
% Double the last element in List1, producing List2.
% E.g.: double_last([a, b, c], [a, b, c, ]).
{double_last([X], [X, X]),
 double_last([H|T1], [H|T2]) :-
   {T1 \== [ ], double_last(T1, T2)}}

% intersect_list(+LL1, -List2)
% List2 is the intersection of all the lists in LL1.
{intersect_list([X], X),
 intersect_list([H|T], L) :-
   {intersect_list(T, L1), intersect(H, L1, L)}}

% telescope(+List1, -List2)
% For example, telescope([a, b, c, d], [a, b, c, d, b, c, d, c, d, d]).
% Adapted from [33, pp. 264 and 266].
{telescope([ ], [ ]),
 telescope([H|T], L) :-
   {telescope(T, L2), append([H|T], L2, L)}}

% listthru(+List1, -List2)
% Convert a list of elements to a list of singleton lists.
% For example, listthru([a, b, c], [[a], [b], [c]]).
% Adapted from [33, p. 266].
{listthru([ ], [ ]),
 listthru([H|T1], [[H]|T2]) :- {listthru(T1, T2)}}

% double(+List1, -List2)
% Double each element in List1, producing List2.
% E.g.: double([a, b, c], [a, a, b, b, c, c]).
{double([ ], [ ]),
 double([H|T1], [H, H|T2]) :- {double(T1, T2)}}

% reverse(+List1, -List2)
% List2 is List1 reversed.
{reverse([ ], [ ]),
 reverse([H|T], L) :- {reverse(T, TR), append(TR, [H], L)}}

% prefix(-P, +L): P is a prefix of list L.
{prefix([ ], _),
 prefix([H|T1], [H|T2]) :- {prefix(T1, T2)}}

% complement(+V1, -V2).
% V2 is the complement of V1 (bitwise not).
{complement([ ],[ ]),
 complement([0|T1],[1|T2]) :- {complement(T1,T2)},
 complement([1|T3],[0|T4]) :- {complement(T3,T4)}}

% remove_duplicates(+List1,List2)
% Remove all duplicate elements in List1, producing List2.
{remove_duplicates([ ],[ ]),
 remove_duplicates([H|T],L) :-
   {member(H,T),remove_duplicates(T,L)},
 remove_duplicates([H1|T1],[H1|T2]) :-
   {non_member(H1,T1),
    remove_duplicates(T1,T2)}}

% even_elts(+List1,-EvenList)
% EvenList contains the even elements of List1.
% For example, even_elts([4,5,-3,0,2,-1],[4,0,2]).
{even_elts([ ],[ ]),
 even_elts([H1|T1],[H1|T2]) :-
   {0 is H1 mod 2,even_elts(T1,T2)},
 even_elts([H2|T3]L) :-
   {1 is H2 mod 2,even_elts(T3,L)}}

% positive_elts(+List1,-PosList)
% PosList contains the positive elements of List1.
% For example, positive_elts([4,5,-3,0,2,-1],[4,5,2]).
{positive_elts([ ],[ ]),
 positive_elts([H1|T1],[H1|T2]) :-
   {H1 > 0,positive_elts(T1,T2)},
 positive_elts([H2|T3],L) :-
   {H2 =< 0,positive_elts(T3,L)}}

% flatten(+LL,-FlatList)
% Flatten list LL to produce FlatList.
{flatten([ ],[ ]),
 flatten([H1|T1],[H1|T2]) :-
   {atom(H1),flatten(T1,T2)}
 flatten([[A|B]|T3],L) :-
   {flatten([A|B],L1),flatten(T3,L2),append(L1,L2,L)}}

% q_sort(+Unsorted,-Sorted): Quicksort.
{q_sort([ ],[ ]),
 q_sort([Pivot|T],Sorted) :-
   {split(T,Pivot,L1,L2),
    q_sort(L1,Sorted1),
    q_sort(L2, Sorted2),
    append(Sorted1, [Pivot|Sorted2], Sorted)}}

% pre_order(+Tree, -Nodes)
% Nodes is a pre-order list of the nodes in Tree.
{pre_order(nil, [ ]),
 pre_order(tree(Node, Left, Right), [Node|T]) :-
   {pre_order(Left, LL),
    pre_order(Right, RL),
    append(LL, RL, T)}}

% in_order(+Tree, -Nodes)
% Nodes is an in-order list of the nodes in Tree.
{in_order(nil, [ ]).
 in_order(tree(Node, Left, Right), List) :-
   {in_order(Left, LL),
    in_order(Right, RL),
    append([Node], RL, L1),
    append(LL, L1, List)}}

% post_order(+Tree, -Nodes)
% Nodes is a post-order list of the nodes in Tree.
post_order(nil, [ ]),
post_order(tree(Node, Left, Right), List) :-
  {post_order(Left, LL),
   post_order(Right, RL),
   append(RL, [Node], L1),
   append(LL, L1, List)}}

% leaf_list(+Tree, -List)
% List is the list of leaves in Tree.
leaf_list(nil, [ ]),
leaf_list(tree(_, Left, Right), List) :-
  {leaf_list(Left, LL),
   leaf_list(Right, RL),
   append(LL, RL, List)}}
```

Appendix C. Schemata of arity 3

Listed below are the schemata and programs mentioned in Fig. 3. As in Appendix B, some comments and mnemonic variable names are used.

```
% s1
{P(_,_,_),
 P(_,_,_),
 P(_,_,_),
 P(_,_,_)}

% s2
{P(W,X,W) :- {Q(W,X)},
 P(Y,Z,Z) :- {R(Y,Z)}}

% s3
{P(_,_,_),
 P(_,_,_) :- {P(_,_,_)},
 P(_,_,_),
 P(_,_,_)}

% s4
{P(_,_,_),
 P(_,_,_) :- {P(_,_,_),P(_,_,_)},
 P(_,_,_),
 P(_,_,_)}

% s5
{P(_,_,_),
 P([_|T],_,_) :- {P(T,_,_)},
 P(_,_,_),
 P(_,_,_)}

% s6
{P([],_,_),
 P([_|T],X,_) :- {P(T,X,_)},
 P(_,_,_),
 P(_,_,_)}

% s7
{P([],[],[]),
 P([_|X1],[_|X2],_) :- {P(X1,X2,_)},
 P([_|Y1],[_|Y2],_) :- {P(Y1,Y2,_)},
 P([_|Z1],[_|Z2],_) :- {P(Z1,Z2,_)}}

% s8
{P([],_,_),
 P([_|T1],X,_) :- {P(T1,X,_)},
 P([_|T2],_,_) :- {P(T2,_,_)}}

% s9
{P([],[],[]),
 P([_|X1],[_|X2],[_|X3]) :- {P(X1,X2,X3)},
 P([_|Y1],[_|Y2],[_|Y3]) :- {P(Y1,Y2,Y3)},
 P([_|Z1],[_|Z2],[_|Z3]) :- {P(Z1,Z2,Z3)}}

% s10
{P([],_,_),
 P([_|T1],X,[_|T3]) :- {P(T1,X,T3)},
 P([_|T2],Y,Z) :- {P(T2,Y,Z)}}

% max(+X,+Y,?Z): Z is the larger of X and Y.
{max(A1,B1,A1) :- {>= (A1,B1)},
 max(A2,B2,B2) :- {> (B2,A2)}}

% min(+X,+Y,?Z): Z is the smaller of X and Y.
{min(A1,B1,A1) :- {=< (A1,B1)},
 min(A2, B2, B2) :- {< (B2, A2)}}

% monus(+X, +Y, ?Z)
% if X > Y, then Z is X − Y; otherwise Z is 0.
% Numbers are represented in successor notation.
{monus(N, 0, N),
 monus(0, _, 0),
 monus(s(X), s(Y), Z) :- {monus(X, Y, Z)}}

% int_div(+X, +Y, ?Z)
% Z is X divided by Y, with remainder ignored.
% Numbers are represented in successor notation.
{int_div(N, N, s(0)),
 int_div(X, Y, s(Z)) :-
    {greater_than(X, Y), monus(X, Y, X2), int_div(X2, Y, Z)},
 int_div(A, B, 0) :- {less_than(A, B)}}

% mod(+X, +Y, ?Z): Z is X mod Y.
% Numbers are represented in successor notation.
{mod(N, N, 0),
 mod(X, Y, Z) :-
    {greater_than(X, Y), monus(X, Y, X2), mod(X2, Y, Z)},
 mod(A, B, A) :- {less_than(A, B)}}

% flatten2(+X, +Y, −Z)
% Flatten list X to produce list Z, using Y for work space.
% Adapted from [36, p.286].
{flatten2([ ], Ws, Ws),
 flatten2([X|Xs], Zs, Ys) :-
    {flatten2(Xs, Zs, Ys1), flatten2(X, Ys1, Ys)},
 flatten2(V, Vs, [V|Vs]) :- {constant(V), \==(V, [ ])}}

% adjacent(+X, ?Y, ?Z)
% Y and Z are adjacent elements of list X.
{adjacent([A, B|_], A, B),
 adjacent([_|T], X, Y) :- {adjacent(T, X, Y)}}

% insert(+List1, +Elt, −List2)
% Insert Elt into sorted List1 to produce sorted List2.
{insert([ ], X, [X]),
 insert([H|T], X, [X, H|T]) :- {H >= X},
 insert([H|T1], X, [H|T2]) :-
    {H < X, insert(T1, X, T2)}}

% merge(+List1, +List2, −List3)
% Merge sorted List1 and sorted List2, producing List3.
{merge( [ ], L, L),
 merge(L, [ ], L) :- {L \== [ ]},
 merge( [H1|T1], [H2|T2], [H1|T3]) :-
    {H1 =< H2, merge(T1, [H2|T2], T3)},
 merge( [H1|T1], [H2|T2], [H2|T3]) :-
    {H1 > H2, merge( [H1|T1], T2, T3)}}

% shuffle( +List1, +List2, -List3)
% For example, shuffle( [a, b, c], [d, e, f], [a, d, b, e, c, f]).
% Adapted from [33, p.266].
{shuffle( [ ], [ ], [ ]),
 shuffle( [H1|T1], [H2|T2], [H1, H2|T3]) :-
    {shuffle(T1, T2, T3)}}

% nd_reverse( +X, +Y, -Z)
% Z is X reversed, with duplicate elements removed.
% Y, a scratch list, is initially [ ].
% From [36, p.146].
{nd_reverse( [ ], Es, Es),
 nd_reverse( [A|As], Revs, Bs) :-
    {member(A, Revs), nd_reverse(As, Revs, Bs)},
 nd_reverse( [C|Cs], Revs2, Ds) :-
    {non_member(C, Revs2),
     nd_reverse(Cs, [C|Revs2], Ds)}}

% append( +X, +Y, -Z)
{append( [ ], L, L),
append( [H|T], L2, [H|NewT]) :- {append(T, L2, NewT)}}

% and( +X, +Y, -Z): Z is bitwise "and" of X, Y.
{and( [ ], [ ], [ ]),
 {and( [1|T1], [1|T2], [1|T3]) :- {and(T1, T2, T3)},
 {and( [0|T4], [_|T5], [0|T6]) :- {and(T4, T5, T6)},
 {and( [1|T7], [0|T8], [0|T9]) :- {and(T7, T8, T9)}}

% or( +X, +Y, -Z): Z is bitwise "or" of X, Y.
{or( [ ], [ ], [ ]),
 or( [0|T1], [0|T2], [0|T3]) :- {or(T1, T2, T3)},
 or( [1|T4], [_|T5], [1|T6]) :- {or(T4, T5, T6)},
 or( [0|T7], [1|T8], [1|T9]) :- {or(T7, T8, T9)}}

% xor( +X, +Y, -Z): Z is bitwise "xor" of X, Y.
{xor( [ ], [ ], [ ]),
 xor( [H|T4], [H|T5], [0|T6]) :- {xor(T4, T5, T6)},
 xor([H1|T1],[H2|T2],[1|T3]) :-
   {\==(H1,H2),xor(T1,T2,T3)}}


% pair2(+List1, +List2, -List3)
% For example, pair2([a,b,c],[d,e,f],[[a,d],[b,e],[c,f]]).
% Adapted from [33, p.266].
{pair2([ ],[ ],[ ]),
 pair2([H1|T1],[H2|T2],[[H1,H2]|T3]) :- {pair2(T1,T2,T3)}}


% union(+X, +Y, -Z)
{union([ ],X4,X4),
 union([X5|R5],Y5,Z5) :-
   {member(X5,Y5),union(R5,Y5,Z5)}}
 union([X6|R6],Y6,[X6|Z6]) :-
   {non_member(X6,Y6),union(R6,Y6,Z6)}}


% intersection(+X, +Y, -Z)
{intersection([ ],_,[ ]),
 intersection([X2|R2],Y2,[X2|Z2]) :-
   {member(X2,Y2),intersection(R2,Y2,Z2)}}
 intersection([X3|R3],Y3,Z3) :-
   {non_member(X3,Y3),intersection(R3,Y3,Z3)}}


% delete_first(+X, +Y, -Z)
% Delete the first occurrence of Y from list X, producing list Z.
{delete_first([ ],_,[ ]),
 delete_first([E|T],E,T),
 delete_first([H|T1],Elt,[H|T2]) :-
   {\==(H,Elt),delete_first(T1,Elt,T2)}}


% delete_all(+X, +Y, -Z)
% Delete all occurrences of Y from list X, producing list Z.
{delete_all([ ],_,[ ]),
 delete_all([E|T],E,L) :- {delete_all(T,E,L)},
 delete_all([H|T1],Elt,[H|T2]) :-
   {\==(H,Elt),delete_all(T1,Elt,T2)}}


% set_diff(+X, +Y, -Z)
{set_diff([ ],_,[ ]),
 set_diff([H|T1],Set2,[H|T3]) :-
   {non_member(H,Set2),set_diff(T1,Set2,T3)},
 set_diff([H1|T],S2,Diff) :-
   {member(H1,S2),set_diff(T,S2,Diff)}}
```

References

[1] D. Angluin and P.D. Laird, Identifying k-CNF formulas from noisy examples, Technical Report TR-478, Computer Science Department, Yale University, New Haven, CT (1986).

[2] D. Angluin and C.H. Smith, Inductive inference: theory and methods, *Computing Surveys* 15 (3) (1983) 237–269.

[3] F. Bergadano and D. Gunetti, Inductive synthesis of logic programs and inductive logic programming, in: Y. Deville, ed., *Logic Program Synthesis and Transformation: Proceedings of LOPSTR 93, International Workshop of Logic Program Synthesis and Transformation* (Springer, Berlin, 1994) 45–56.

[4] W. Bibel and K.M. Hörnig, LOPS—a system based on a strategical approach to program synthesis, in: A.W. Biermann, G. Guiho and Y. Kodratoff, eds., *Automatic Program Construction Techniques* (Macmillan, New York, 1984) 69–89.

[5] A.W. Biermann, The inference of regular LISP programs from examples, *IEEE Trans. Systems Man Cybernet.* 8 (8) (1978) 585–600.

[6] A.W. Biermann and J.A. Feldman, A survey of results in grammatical inference, in: Y.-H. Pao and G.W. Ernst, eds., *Tutorial: Context-Directed Pattern Recognition and Machine Intelligence Techniques for Information Processing* (IEEE, Silver Spring, MD, 1982) 113–136; reprinted from: S. Watanabe, ed., *Proceedings of the International Conference of Frontiers of Pattern Recognition* (Academic Press, New York, 1972) 36–54.

[7] M. Broy, Program construction by transformations: a family tree of sorting programs, A.W. Biermann and G. Guiho, eds., *Computer Program Synthesis Methodologies* (Reidel, Dordrecht, Netherlands, 1983) 1–49.

[8] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *J. ACM* 24 (1) (1977) 44–67.

[9] A.K. Chandra and P.M. Merlin, Optimal implementation of conjunctive queries in relational data bases, in: *Proceedings 9th ACM Symposium on Theory of Computing* (ACM, New York, 1977) 77–90.

[10] W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer, Berlin, 4th ed., 1994).

[11] N. Dershowitz, *The Evolution of Programs* (Birkhäuser, Boston, MA, 1983).

[12] N. Dershowitz, Programming by Analogy, in: *Machine Learning*, Vol. II (Morgan Kaufmann, Los Altos, CA, 1986) 393–421.

[13] P. Flener and Y. Deville, Synthesis of composition and discrimination operators for divide-and-conquer logic programs, in: J.-M. Jacquet, ed., *Constructing Logic Programs* (Wiley, New York, 1993) 67–96.

[14] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W.H. Freeman, New York, 1979).

[15] D. Gilbert and C. Hogger, Deriving logic programs from observations, in: J.-M. Jacquet, ed., *Constructing Logic Programs* (Wiley, New York, 1993) 113–126.

[16] E.M. Gold, Language identification in the limit, *Inform. and Control* 10 (5) (1967) 447–474.

[17] M. Grobelnik, Induction of Prolog programs with Markus, in: Y. Deville, ed., *Logic Program Synthesis and Transformation: Proceedings of LOPSTR 93, International Workshop of Logic Program Synthesis and Transformation* (Springer, Berlin, 1994) 57–63.

[18] A.-L. Johansson, Interactive program derivation using program schemata and incrementally generated strategies, in: Y. Deville, ed., *Logic Program Synthesis and Transformation: Proceedings of LOPSTR 93, International Workshop of Logic Program Synthesis and Transformation* (Springer, Berlin, 1994) 100–112.

[19] P.D. Laird, Inductive inference by refinement, Technical Report TR-376, Computer Science Department, Yale University, New Haven, CT (1986).

[20] P. Laird, Learning from good data and bad, Ph.D. Dissertation, TR-551, Yale University, New Haven, CT (1987).

[21] K.K. Lau and S.D. Prestwich, Top-down synthesis of recursive procedures from first-order logic specifications, in: D.H.D. Warren and P. Szeredi, eds., *Logic Programming: Proceedings of the Seventh International Conference* (MIT Press, Cambridge, MA, 1990) 667–684.

[22] C.L. Liu, *Elements of Discrete Mathematics* (McGraw-Hill, New York, 1977).

[23] Z. Manna and R. Waldinger, A deductive approach to program synthesis, in: B.L. Webber and N.J. Nilsson, eds., *Readings in Artificial Intelligence* (Tioga, Palo Alto, CA, 1981) 141–172; reprinted from: *ACM Trans. Programming Languages and Systems* 2 (1) (1980) 120–151.

[24] R.S. Michalski, Inductive learning as rule-guided generalization of symbolic descriptions: a theory and implementation, in: A.W. Biermann, G. Guiho and Y. Kodratoff, eds., *Automatic Program Construction Techniques* (Macmillan, New York, 1984) 517–552.

[25] T.M. Mitchell, Generalization as search, *Artificial Intelligence* 18 (2) (1982) 203–226.

[26] G.D. Plotkin, A note on inductive generalization, in: B. Meltzer and D. Mitchie, eds., *Machine Intelligence*, Vol. 5 (Halsted Press, New York, 1970) 153–163.

[27] G.D. Plotkin, A further note on inductive generalization, in: B. Meltzer and D. Mitchie, eds., *Machine Intelligence*, Vol. 6 (Halsted Press, New York, 1971) 101–124.

[28] R.J. Popplestone, An experiment in automatic deduction, in: B. Meltzer and D. Mitchie, eds., *Machine Intelligence*, Vol. 5 (Halsted Press, New York, 1970) 101–124.

[29] J.C. Reynolds, Transformational systems and the algebraic structure of atomic formulas, in: B. Meltzer and D. Mitchie, eds., *Machine Intelligence*, Vol. 5 (Halsted Press, New York, 1970) 135–151.

[30] E.Y. Shapiro, Inductive inference of theories from facts, Research Report 192, Computer Science Department, Yale University, New Haven, CT (1981).

[31] E.Y. Shapiro, An algorithm that infers theories from facts, in: *Proceedings IJCAI-81*, Vancouver, BC (1981) 446–451.

[32] E.Y. Shapiro, *Automatic Program Debugging* (MIT Press, Cambridge, MA, 1982)

[33] D. Shaw, W. Swartout and C. Green, Inferring LISP programs from examples, in: *Proceedings IJCAI-75*, Tblisi, Georgia (1975) 260–267.

[34] D.R. Smith, The Structure and Design of Global Search Algorithms, KES.U.87.12, Kestrel Institute, Palo Alto, CA (1988).

[35] L. Sterling and M. Kirschenbaum, Applying techniques to skeletons, in: J.-M. Jacquet, ed., *Constructing Logic Programs* (Wiley, New York, 1993) 127–140.

[36] L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, Cambridge, MA, 2nd ed., 1994).

[37] P.D. Summers, A methodology for LISP program construction from examples, *J. ACM* 24 (1) (1977) 161–175.

[38] N.L. Tinkham, Induction of schemata for program synthesis, Technical Report CS-1990-14, Duke University, Durham, NC (1990).

[39] N.L. Tinkham, A theorem on refinement operators for logic program synthesis, Technical Report TR97-1, Rowan University, Glassboro, NJ (1997).

[40] L.G. Valiant, A theory of the learnable, *Comm. ACM* 27 (11) (1984) 1134–1142.

[41] P.H. Winston, Learning structural descriptions from examples, in: P.H. Winston, ed., *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975) 157–209.