Specialization of Recursive Predicates from Positive Examples Only

Moussa Demba
Department of Computer and Information Sciences,
Aljouf University Skaka, Saudi Arabia

Abstract: Problem statement: Given an overly general (definite) program \( P \) and its intended semantics \( \phi \) (the programmer’s intentions) where \( P \) does not satisfy \( \phi \), find out a new version \( P' \) of \( P \) such that \( P' \) satisfies \( \phi \). Approach: We proposed an approach for correcting overly general programs from positive examples by exploiting program synthesis techniques. The synthesized program, \( P' \), is a specialization of the original one, \( P \). In contrast to the previous approaches for logic program specialization, no negative examples were given as input but they will be discovered by the algorithm itself. The specialization process is performed according to the positive examples only. A method for refining logic programs into specialized version was then proposed. Results: The proposed approach was able to correct overly general programs using positive examples. We showed that positive examples can also be used for inducing finite-state machines, success sequences, that models the correct program. The failing sequences also exploited by theorem proved to produce counter-examples as in model checking, by composing substitutions used for inducing failing sequences. Conclusion: The contribution of the study was mainly the use of specification predicates to specialize an overly general logic program.

Key words: Program specialization, theorem proving, positive/negative examples, folding/unfolding rules, finite-state machine

INTRODUCTION

Our aim is to present a top-down approach for logic program specialization w.r.t the intended specification which is a first-order formula of the following form: 
\[
\forall \overline{x} \exists \overline{Y} \phi(\overline{x}, \overline{Y}) \leftarrow \forall \overline{x} \exists \overline{Y} \Gamma(\overline{x}, \overline{Y}) \quad (\text{or } \Gamma \leftarrow \Delta \text{ for short})
\]
where \( \Gamma \) and \( \Delta \) are conjunction of atoms. The problem we are interested in can be stated as follows:

Given: An overly general (definite) program \( P = (E^+ \cup C) \) where \( E^+ \) is a recursive sub-program defining positive examples, \( C \) is supposed to be the set of clauses defining the overly general predicates (i.e. the incorrect component of \( P \)) and the intended semantics \( \phi \) for \( P \) (the programmer’s intentions) with \( M(P) = \phi \) where \( M(P) \) denotes the least Herbrand model of \( P \).

Find: A definite program \( D \), called a specialization of \( C \), such that \( M(D) \subseteq M(C) \) and \( M(P') \models \phi \) where \( P' = (P/C) \cup D \).

The program \( P' \) is called a correct specialization of \( P' \) w.r.t \( E^+ \) if \( M(P') \subseteq M(P) \) and \( M(E^+) \subseteq M(P') \) and for any negative example \( e^- \), \( M(P') \models e^- \).

Roughly the approach takes an overly general program \( P \) and its intended semantics \( \phi \) and tries to produce a program \( P' \) that is guaranteed to satisfy the specification and therefore does not require verification. We outline a top-down method for synthesizing a correct and consistent logic program \( P' \) that satisfies the given specification. Moreover, the negative examples \( E^- \) correspond to ground atoms that are not deducible from \( P' \) and are automatically discovered during the specialization process.

For example, assume we are given the overly general program \( P = (E^+ \cup C) \) where:

- \( E^+ \): 
  \[
  \begin{align*}
  &\text{even}(0) \quad \leftarrow \\
  &\text{even}(s^2(n)) \quad \leftarrow \text{even}(n)
  \end{align*}
  \]

and

- \( C \): 
  \[
  \begin{align*}
  &\text{len}([], 0) \quad \leftarrow \\
  &\text{len}([a \mid x], s(n)) \quad \leftarrow \text{len}(x, n)
  \end{align*}
  \]

and its intended specification:

\[
\phi : \text{even}(n) \leftarrow \text{len}(x, n)
\]
supposed to establish the claim “if n is the length of list x, then n is even”.

For this specification P is false as we discover while attempting to prove it, for example there are particular values of the list x that generate negative examples: It is the case where the number of elements in x is odd and the negative examples even (\(s_k(0)\)) \(k = 0, \ldots, n\) will be generated (s is the successor function). But with the specialized version \(D\) of \(C\), the new program \(P' = (P \setminus D)\) satisfies \(\phi\) up to renaming the predicate \(len\) by \(len_2\) that is defined as follows:

\[
D' = \begin{cases} 
\text{len}_2([],0) 
\rightarrow \text{len}_2([a,b|x],s^2(n)) \rightarrow \text{len}_2(x,n) 
\end{cases}
\]

The new predicate \(\text{len}_2\) is called a specialization predicate of \(\phi\) w.r.t the predicate \(\text{len}\). The proposed method consists to synthesize \(D\).

Throughout the study, \(\Gamma\), \(\Delta\) and \(\Lambda\) denote conjunctions of atoms; \(\phi\) denotes the intended specification (a first-order formula); \(A\) and \(B\) denote atoms and \(\theta\) denotes substitution. In all formulas, existentially quantified variables are distinguished from universal variables by giving them upper-case letters. A program is a set of definite clauses, denoted by calligraphic letters: \(P, Q, \ldots\).

**MATERIALS AND METHODS**

Let \(C\) be a conjunction of atoms. Then \(\mu(C) = \emptyset\) if \(C\) is the predicate true and the multi-set of the atoms of \(C\) otherwise.

**Definition 1:** A conjunction of atoms \(C_1\) is a specialization of (or is syntactically less general than) a conjunction of atoms \(C_2\) (denoted \(C_1 \preceq C_2\)) with a substitution \(\theta\) iff \(\mu(C_2\theta) \subseteq \mu(C_1)\) (Flener and Deville, 1993).

For example, suppose \(C_1 = p(a, x) \land q(y), C_2 = p(v,w)\) and \(\theta = \{v/a, w/x\}\) we have \(C_1 \preceq C_2\). Indeed, \(\mu(C_2\theta) = \{p(a,x)\}\) and \(\mu(C_1) = \{p(a,x),q(y)\}\).

The following definition expresses the relation of generality between two horn clauses.

**Definition 2:** A definite clause \(A \leftarrow \Delta\) is represented by a couple of elements \((A, \Delta)\). A clause \((A_1, \Delta_1)\) is a specialization of a clause \((A_2, \Delta_2)\), denoted \((A_1, \Delta_1) \preceq (A_2, \Delta_2)\), with a substitution \(\theta\) iff (i) \(\mu(A_2\theta) \subseteq \mu(A_1)\) and (ii) \(\mu(\Delta_2\theta) \subseteq \mu(\Delta_1)\).

For example for \((A_1, \Delta_1) = (r(a,y), \{p(a,x) \land q(y)\})\) and \((A_2, \Delta_2) = (r(v,w,y), \{p(v,w)\})\), we have \((A_1, \Delta_1) \preceq (A_2, \Delta_2)\) with the substitution \(\theta = \{v/a, w/x\}\) as \(\mu(A_2\theta) = \{r(a,y)\}\) and \(\mu(\Delta_2\theta) = \{p(a,x)\}\).

The following definition expresses the relation of generality between two logic programs.

**Definition 3:** Let \(P_1\) and \(P_2\) be two definite logic programs, \(\{c_1, \ldots, c_n\}\) the set of clauses of \(P_1\) and \(\{d_1, \ldots, d_m\}\) the set of clauses of \(P_2\). \(P_1\) is a specialization of \(P_2\) (denoted \(P_1 \preceq P_2\)) iff for all \(1 \leq i \leq n\), there exists \(1 \leq j \leq m\) s.t. \(c_i \preceq d_j\).

**Example:** Let \(P_1\) and \(P_2\) be two definite programs:

\[
P_1 = \begin{cases} 
c_1 : p(s(0),0) \rightarrow c_2 : q(0,0) 
\end{cases}
\]

\[
P_2 = \begin{cases} 
d_1 : p(s(0),0) \rightarrow d_2 : p(s(0),0) 
\end{cases}
\]

Then \(P_1\) is a specialization of \(P_2\) as \(c_1 \preceq d_2\) and \(c_2 \preceq d_3\).

**Definition 4 (Specialization predicate):** Let \(P_1\) and \(P_2\) be two definite programs defining the predicates \(P_1\) and \(P_2\) respectively. If \(P_1\) is a correct specialization of \(P\), i.e., \(P_1 \preceq P_2\) w.r.t the intended specification \(\phi\), then \(P_1\) is called a specialization predicate of \(P_2\) w.r.t \(\phi\).

**Proposition 1:** Let \(\phi: \Gamma \leftarrow \Delta\), \(P_2\) be the intended specification of the program \(P\). If \(P_1\) is a specialization predicate of \(\phi\) with respect to the predicate \(P_2\), then we have \(M(P_1 \cup P_2) = ((\Gamma, \Delta, P_2) \leftarrow P)\) where \(P_1\) defines the predicate \(P_1\).

**Proposition 2:** If \(P_1\) is a specialization predicate of \(\phi\) with respect to the predicate \(P_2\), then the two formulas (1) and (2) are equivalent:

\[
(\Gamma \leftarrow \Delta) \leftarrow P_2 \quad (1)
\]

\[
(\Gamma \leftarrow \Delta) \leftarrow P_1 \quad (2)
\]

**Proof:** It is easy to see that the formula (1) is equivalent to:

\[
(\Gamma \leftarrow \Delta) \leftarrow P_2 \quad (3)
\]
Moreover, if \( p_1 \not\subset p_2 \), \( p_2 \leftarrow p_1 \) is a theorem. Therefore, the formula (3) is equivalent to:

\[
\Gamma \leftarrow \Delta, p_i
\]  

(4)

For example, the formula \((\text{even} (n) \leftarrow \text{len}(x, n), \text{len}_2(x, n))\) is equivalent to \(\text{even}(n) \leftarrow \text{len}_2(x, n)\).

**Notation 1:** Hereafter and for the sake of simplicity, the notation \(\langle \phi \mid p \rangle\) stands for \(\langle \phi \leftarrow p \rangle\).

In the definitions 5-8, we define the semantic calculus that allows given \(P\) and its intended specification \(\phi\) such that \(P\) is faulty w.r.t \(\phi\), to find \(P'\) such that \(P'\) satisfies \(\phi\).

**Transformation rules:** The algorithm applies the transformation rules unfolding, folding and simplification (Sakurai and Motoda, 1988). Intuitively, unfolding is an extension of SLD-resolution and folding simplification (Sakurai and Motoda, 1988). The induction hypothesis by the corresponding conclusion. The application of an unfolding rule on a formula \(\phi_0\) generates a finite set of formulas \(\phi_i\), such that \(\phi_0\) follows from the \(\phi_i\)'s in the least Herbrand model of the program under consideration. Each formula \(\phi_i\) is associated with a specialization predicate, as it can be an overly general clause and \(\phi_i\) is obtained (directly or indirectly) from \(\phi_0\) by the rule of nfi, \(\phi_i\) is an existential one (\(\phi_i\) substitutes only existential variables of \(\Delta\)) in \(Q_{\text{dci}}\).

\[
\begin{align*}
\langle \phi_0 : (\Gamma \leftarrow \Delta, A) \mid IO_0 \rangle > & \quad \text{(nfi)} \\
\langle \phi_i : (\Gamma \leftarrow \Delta) \theta_i \leftarrow A \mid IO_i \rangle >_{i=1,...,k} & \quad \text{(dci)}
\end{align*}
\]

where, \(IO\) is the specialization predicate of \(\phi\) w.r.t the predicate \(\Delta\). Hence \(Q_{\text{dci}} = \{IO_0 \theta_i \leftarrow IO_i \}_{i=1,...,k}\).

**Definition 6:** (definite clause inference) Let \(P\) be a program, \(\phi_0\): \(\Gamma \leftarrow \Delta\) a formula and \(C = \{c_1,...,c_k\}\) the set of clauses of \(P\) such that \(c_i : B_i \leftarrow \Delta_i\) and suppose there is a substitution \(\theta_i\) s.t \(B_i \theta_i = A \theta_i\). Then the application of the rule of nfi on \(\phi_0\) w.r.t to the atom \(A\) yields a disjunction of \(k\) formulas:

\[
\langle \phi_0 : (\Gamma \leftarrow \Delta, A) \mid IO_0 \rangle > \quad \text{(nfi)}
\]

where, \(IO\) is the specialization predicate of \(\phi\) w.r.t the predicate \(A\). Hence \(Q_{\text{nfi}} = \{IO_0 \theta_i \leftarrow IO_i \}_{i=1,...,k}\).

**Definition 7 (folding rule or cutr):** Let \(\phi_0: A \leftarrow \Pi\) and \(\theta_i; \Gamma \leftarrow \Delta_i\), \(\Delta_2\) be two formulas satisfying the following conditions: (i) \(\phi_0\) is obtained (directly or indirectly) from \(\phi_i\) by the rule of dci, (ii) \(\Delta_i\) is an instance of \(\Pi\), i.e., there is a substitution \(\theta\) such that \(\Pi \theta = \Delta_i\), (iii) for any local variable \(x\) in \(\Pi\), \(x\theta\) is a variable and does not occur other than in \(\Delta_i\) and (iii) \(\theta\) replaces different local variables of \(\Pi\) with different local variables of \(\Delta_i\). Then replace \(\phi_i\) by \(\phi_0: A \leftarrow \Pi\):

\[
\begin{align*}
\langle \phi_0 : (A \leftarrow \Pi) \mid IO_0 \rangle > & \quad \text{(cutr)} \\
\langle \phi_i : (\Gamma \leftarrow \Delta_i, A) \mid IO_i \rangle > & \quad \text{(cutr)} \\
\langle \phi_0 : (\Gamma \leftarrow \Lambda \theta, A) \mid IO_2 \rangle > & \quad \text{(cutr)}
\end{align*}
\]

in this case \(Q_{\text{cutr}} = \{IO_0 \leftarrow IO_0 \theta, IO_2\}\) defines the predicate \(IO_1\) in terms of \(IO_0\) and \(IO_2\) where \(IO_0\), \(IO_1\)
and \( \text{IO}_2 \) are the associated specification predicates of \( \phi_0 \), \( \phi_1 \) and \( \phi_2 \) respectively. \( \phi_0 \) thus plays the role of the induction hypothesis. This important rule allows us to synthesize recursive predicates.

**Definition 8 (Simplification or simp):** The rule of simplification (simp) simplifies the atoms A and B if there is an existent substitution \( \theta \) s.t \( A\theta = B \):

\[
\frac{\langle \phi : (A, \Gamma \leftarrow B, \Delta) \mid \text{IO} \rangle}{\langle \phi' : (\Gamma \leftarrow \Delta) \mid \text{IO}' \rangle} \quad \text{(simp)}
\]

and

\[
\text{(simp)}_{\text{comp}} = [\text{IO} \leftarrow \text{IO'}]
\]

All the rules are partially correct (Demba et al., 2005). Substitutions of variables during the specialization process are stored into the specialization predicates.

**RESULTS**

Let \( P \) be a logic program and \( \phi \) its intended specification (intended to be true). If \( M(P) \models \phi \), then \( P \) is buggy. Our goal is (i) to isolate the set of incorrect axioms, denoted by \( C \), of \( P \) w.r.t \( \phi \) and (ii) to synthesize a sub-program, denoted by \( D \), from the proof attempt of \( P \) w.r.t. to \( \phi \) and (iii) to determine the program \( P' = (P \setminus C) \cup D \) where \( D \) is a specialization of \( C \) and \( M(P') \models \phi \).

Suppose \( P = (E^+ \cup C) \) and \( D \) the synthesized program during the proof attempt of \( P \) w.r.t \( \phi \). Suppose \( C = \{c_1, \ldots, c_n\} \) and \( D = \{d_1, \ldots, d_m\} \), here is the specialization algorithm Fig. 1.

Hereafter, we will add clause numbers to incorporate into Fig. 2 and 3 of clause sequences in order to clarify which clauses have been resolved with.

**Example 1:** Consider the program \( P = (E^+ \cup C) \) where \( C \) expresses that any natural number is odd:

\[
\begin{align*}
  c_1 &: \text{even}(s(0)) \leftarrow \text{IO}_1() \\
  c_2 &: \text{even}(s^2(n)) \leftarrow \text{odd}(n) \mid \text{IO}_2(n)
\end{align*}
\]

together with the intended specification

\[
\phi : \text{even}(s(n)) \leftarrow \text{odd}(n)
\]

It is clear that \( M(P) \models \phi \), for example we have even(0) and odd(0). Therefore, the program \( P \) and specially the sub-program \( C \), covers negative examples.

---

**Fig. 1: Program specialization algorithm**

**Fig. 2: Specialization of odd(n) w.r.t \( E^+ \)**

**Fig. 3: Specialization of plus(x, y, Z) w.r.t \( E^+ \)**

To fix this problem, we need to specialize the predicate odd. To do that, assume \( \text{IO}_\phi \) is a specialization predicate of odd associated to \( \phi \):

\[
\phi : \text{even}(s(n)) \leftarrow \text{odd}(n) \mid \text{IO}_\phi(n)
\]

Note that we need to synthesize a definite program \( D \) defining the predicate \( \text{IO}_\phi \) such that \( M(E^+ \cup D) \models \phi \) with \( \phi = \text{even}(s(n)) \leftarrow \text{IO}_\phi(n) \) and \( M(D) \subseteq M(C) \). \( D \) is initially empty.

The specialization process of \( C \) w.r.t \( E^+ \) is in the way depicted in Fig. 2. The first step consists to unfold \( \Gamma \) upon the atom odd(n) using the rule of nfi to obtain the following resultants:

\[
\begin{align*}
  c_1 &: \text{even}(s(0)) \leftarrow \text{IO}_1() \\
  c_2 &: \text{even}(s^2(n)) \leftarrow \text{odd}(n) \mid \text{IO}_2(n)
\end{align*}
\]
and IO₀ is defined in terms of IO₁ and IO₂ in Qₜₜ as follows:

\[
\begin{align*}
\text{IO}_0(0) & \leftarrow \text{IO}_1() \\
\text{IO}_0(s(n)) & \leftarrow \text{IO}_2(n)
\end{align*}
\]

The clause \(c_5\) corresponds to a negative example as \(M(E') \models \text{even}(s(0))\), then its associated specialization predicate IO₁ is set to false and the clause \(\text{IO}_0(0)\leftarrow\text{IO}_1()\) is not included in D:

\[
D = \emptyset \cup \{\text{IO}_0(s(n)) \leftarrow \text{IO}_2(n)\}
\]

We apply again the rule of nfi on \(c_6\) upon odd(n) to get:

\[
\begin{align*}
c_7 & : \text{even}(s^2(n)) \leftarrow \text{IO}_3() \\
c_6 & : \text{even}(s^1(n)) \leftarrow \text{odd}(n) \quad | \quad \text{IO}_4(n)
\end{align*}
\]

The clause \(c_7\) corresponds to a positive example, then IO₃ is set to true and:

\[
D = \{\text{IO}_0(s(n)) \leftarrow \text{IO}_2(n)\}
\]

Next, we unfold \(c_8\) using the rule of dci w.r.t \(c_2\), to obtain:

\[
c_9 : \text{even}(s(n)) \leftarrow \text{odd}(n) \quad | \quad \text{IO}_3(n)
\]

and

\[
D = \left\{\begin{array}{l}
\text{IO}_0(s(n)) \leftarrow \text{IO}_2(n) \\
\text{IO}_2(0) \\
\text{IO}_2(s(n)) \leftarrow \text{IO}_4(n) \\
\text{IO}_4(n) \leftarrow \text{IO}_3(n)
\end{array}\right.
\]

By eliminating the intermediate predicates a la Tamaki and Sato (1984), we get the final version:

\[
D = \left\{\begin{array}{l}
\text{IO}_0(s(0)) \\
\text{IO}_0(s^2(n)) \leftarrow \text{IO}_2(n)
\end{array}\right.
\]

and we have the correct specialization program \(P' = (P \cup C) \cup D\) of \(P\) w.r.t \(E'\) and \(M(P') \models \phi\) where \(\phi: \text{even}(s(n)) \leftarrow \text{IO}_6(n)\) (according to the Proposition 2).

Note that the clause \(c_3\) is automatically removed and \(c_4\) is refined by specialization. The success sequences of clauses that cover only the positive examples, \(E'\), are of the following form \(c_4(c_2c_4)^*c_3\) represented by the Fig. 2. Any other combination of clauses will cover negative examples, then leads to failure. From \(E'\), we can induce the finite-state machine of Fig. 2 that corresponds to the sub-program D.

The Fig. 2 can be interpreted as follows: the transition \(c_4\) corresponds to the application of the rule of nfi while the transition \(c_2\) corresponds to the application of the rule of dci. The loop means that recursive specialization is necessary, that is the application of the folding rule (cutr) is needed to complete the process.

The approach can also be applied to more complex specification. For example a specification with existential variables or an original program where the positive examples consist of different predicates as in the following example.

**Example 2:** Consider the specification:

\[
\phi: \text{plus}(x, y, Z), \text{sup}(Z, x) \leftarrow \text{nat}(x), \text{nat}(y)
\]

where the predicates sup, nat and plus are initially defined as usual by the program, say \(P = (E' \cup C)\):

\[
\begin{align*}
c_1 & : \text{sup}(s(x),0) \\
c_2 & : \text{sup}(s(x),s(y)) \\
c_3 & : \text{nat}(0) \\
c_4 & : \text{nat}(s(x))
\end{align*}
\]

and the clause \(\text{IO}_3(n) \leftarrow \text{IO}_4(n)\); \(\text{IO}_6(n)\) is generated. The formula \(c_{10}\) can be simplified to true, \(\text{IO}_6\) is then set to true. The final program D is then:

\[
D = \left\{\begin{array}{l}
\text{IO}_0(s(0)) \\
\text{IO}_0(s^2(n)) \leftarrow \text{IO}_2(n)
\end{array}\right.
\]
surprisingly, D is a specialization of C and M(K∪D) |= (sup(z, x) ← IO₀(x, y, z)). Comparing C and D, we can say that the error was in the first clause of C, i.e., the underlined arguments. The correct program P’ is then:

\[
\begin{align*}
\text{sup}(x, y) & \iff plus(x, y, z) \\
\text{nat}(x) & \iff \text{sup}(x, y)
\end{align*}
\]

The success sequences of clauses that cover only the positive examples, E’, are depicted by the Fig. 3. Any other combination of clauses will cover negative examples, then leads to failure. From E’, we can induce the finite-state machine of Fig. 3 that corresponds to the sub-program D.

From E’, we can induce the finite-state machine of Fig. 3. The transitions \( c_3 \) and \( c_4 \) correspond Fig. 3. Specialization of \( plus(x, y, z) \) w.r.t \( E’ \) to the application of the rule of \( \text{nfi} \) and the transitions \( c_5 \) and \( c_6 \) correspond to the application of the rule \( \text{dci} \). Again to complete the process, the application of the folding rule is needed.

**DISCUSSION**

Bostrom and Idestam-Alquist (1994; 1999) presented top down approaches such as the divide-and-conquer, covering and SPECTRE algorithms for logic program specialization using unfolding and clause deletion rules. One of the limitations of those algorithms is that the divide-and-conquer algorithm does not work when specializing clauses that define recursive predicates and the SPECTRE algorithm cannot synthesize recursive specifications. A bottom-up approach has been proposed in (Kanamori and Seki, 1986; Ferri et al., 2001; Leuschel and Massart, 2003). Ballis (2005) claimed that his approach can be applied as a top down or a bottom up approach. All those approaches are driven by (a finite set) positive and negative examples. It is not also clear how they handle cases when some positive examples are not included in the specification. Other works have been proposed to correct faulty specification (Protzen, 1996 Monroy, 2000) and all deal with faulty universally quantified equations.

To guarantee that all positive examples are included in the original program, we have proposed to represent them not as a set of ground terms but a recursive program denoted \( E’ \). The intended specification we consider is not limited to Horn clauses but a first-order formula with universal and existential variables. The negative examples are not given as input but discovered during the proof process. Recursive predicates are synthesized, if needed.

**CONCLUSION**

We have presented a new way to specialize logic programs from positive examples only. With this approach recursive predicates can be obtained. We have shown that positive examples can be used for inducing finite-state machines (success sequences). The failing sequences could also be exploited by theorem proves to produce counter-examples as in model checking, by composing substitutions used for inducing failing sequences. The presented approach is implemented in Ocaml and integrated into the interactive proof assistant SPES (Demba et al., 2005). The contribution of the study is mainly the use of specification predicates to specialize an overly general logic program.

The framework presented here has two major advantages: (i) The positive examples defined in \( E’ \) are guaranteed to be included both in the meaning of the original program and of the specialized version. Note that in (Ballis, 2005; Alpuente et al., 2001; Bostrom and Idestam-Alquist, 1999), \( E’ \) consists of a finite set of ground atoms and it is not clear how they handle cases when some positive examples are not included in the original program. (ii) The specialization process is performed according to the positive examples only, no need to negative examples. (iii) It supports reasoning about specifications whose stat-spaces may be infinite.

But more works are needed to guarantee the termination of the procedure. This problem is due by the fact that the procedure is based on theorem proving techniques.
REFERENCES

Alpuente, M., F.J. Correa and M. Falaschi, 2001. Declarative debugging of functional logic programs. Elect. Notes Theor. Comput. Sci., 57: 17-40. DOI: 10.1016/S1571-0661(04)00266-X

Ballis, D., 2005. Rule based software verification and correction. Ph.D. Thesis, Universidad Politécnica de Valencia, Spain. http://hdl.handle.net/10251/1948

Bostrom, H. and P. Idestam-Alquist, 1994. Specialization of logic programs by pruning SLD-tree. Proceeding of the 4th International Workshop on Inductive Logic Programming, Sept. 1994, Bonn Germany, pp: 12-14. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3200

Bostrom, H. and P. Idestam-Alquist, 1991. Induction of logic programs by example-guided unfolding. J. Logic Program, 40: 159-183. DOI: 10.1016/S0743-1066(99)00017-5

Demba, M., F. Alexandre and K. Bsaies, 2005. Correction of faulty conjectures and programs extraction. Proceeding of the 20th International Workshop on Disproving Non-Theorems, Non-Validity, Non-Provability (CADE 05), July, 2005, pp: 17-27.

Ferri, C., J. Hernandez and M.J. Ramirez, 2001. Incremental learning of functional logic pro-grams. Lecturer Notes Comput. Sci., 2024: 233-247.

Flener, P. and Y. Deville, 1993. Logic program synthesis from incomplete specifications. J. Symb. Comput., 15: 775-805. DOI: 10.1016/S0747-7171(06)80012-X

Kanamori, T. and H. Seki, 1986. Verification of prolog programs using an extension of execution. Proceeding of the 3rd International Conference on Logic Programming, (LG’86), ACM Press, London, United Kingdom, pp: 475-489. http://portal.acm.org/citation.cfm?id=12105

Leuschel, M. and T. Massart, 2003. Inductive theorem proving by program specialization: Generating Proofs for Isabelle using Ecce (invited talk). Proceeding of the Symposium on Logic Based Program Synthesis and Transformation, Aug. 2003, Uppsala, Sweden, pp: 1-18. http://eprints.ecs.soton.ac.uk/8342/

Monroy, R., 2000. The use of abduction and recursion editor techniques for the correction of faulty conjectures. Proceeding of the 15th IEEE International Conference on Automated Software Engineering, Sept. 11-15, IEEE Computer Society Press, USA., pp: 91-99. http://portal.acm.org/citation.cfm?id=786768.7869

Protzen, M., 1996. Patching faulty conjectures. Lecturer Notes Comput. Sci., 1104: 77-91. DOI: 10.1007/3-540-61511-3_70

Sakurai, A. and H. Motoda, 1988. Proving definite clauses without explicit use of inductions. Lecturer Notes Comput. Sci., 383: 11-26. DOI: 10.1007/3-540-51564-X_52

Tamaki, H. and T. Sato, 1984. Unfold/fold transformation on logic programs. Proceeding of the 2nd International Conference on Logic Programming, July 1984, IEEE Computer Society Press, USA., pp: 127-138.