Incremental Refinement of Success Patterns of Logic Programs

Lunjin Lu
Oakland University
lunjin@acm.org

Abstract

We propose a method for incrementally computing success patterns of logic programs with respect to a class of abstractions. The method is specialised for computing success patterns for depth and stump abstractions. Equational unification algorithms for these abstractions are presented.

1. INTRODUCTION

A program analysis is usually performed with respect to a fixed abstraction, and different analyses corresponding to different abstractions are performed separately even if there is a strong relationship between them. Take depth abstractions [9, 7, 6] for example, a depth 3 analysis will be performed separately from a depth 2 analysis even if the result of the depth 2 analysis can be used in the depth 3 analysis.

This paper is concerned with refining analyses whereby the result of a coarser analysis is used to obtain a finer one. In particular, we are concerned with obtaining finer success patterns of a logic program from coarser success patterns of the same program. A pattern describes a set of objects satisfying some properties. Patterns of the ground atoms that are provable from a program are called its success patterns.

We first show, for a class of abstractions, that the set of the success patterns of a logic program $\mathcal{P}$ with respect to an abstraction α is equal to the success set of the equational logic program $\mathcal{P} \cup E_\alpha$ where E_α is an equality theory induced by α. This leads to a method for incrementally refining success patterns. The set of coarser success patterns of $\mathcal{P}$ relative to a stronger abstraction α_1 can be obtained by computing the fixpoint semantics of $\mathcal{P} \cup E_{\alpha_1}$. If the success patterns are not fine enough, candidates for finer success patterns relative to a weaker abstraction α_2 are generated from the coarser success patterns and verified using the procedural semantics of $\mathcal{P} \cup E_{\alpha_2}$. We apply the method to compute success patterns with respect to depth [9, 7] and stump [12] abstractions. Equational unification algorithms for these abstractions are presented.

Section 2 presents a method for incrementally refining success patterns. Sections 3 and 4 devote to incremental refinement of success patterns for depth and stump abstractions. Section 5 concludes. Proofs are omitted due to space limit.

Let $\Sigma, \Pi, \mathcal{V}$ be respectively a set of function symbols, a set of predicate symbols and a denumerable set of variables. Let $\mathsf{V} \subseteq \mathcal{V}$. $\mathcal{T}(\Sigma, \mathsf{V})$ denotes the set of terms constructible from Σ and V, and $\mathcal{A}(\Pi, S)$ denotes the set of atoms constructible from Π and S where $S \subseteq \mathcal{T}(\Sigma, \mathcal{V})$. The Herbrand universe $\mathcal{HU}$ and the Herbrand base $\mathcal{HB}$ of a logic program $\mathcal{P}$ are $\mathcal{HU} = \mathcal{T}(\Sigma, \emptyset)$ and $\mathcal{HB} = \mathcal{A}(\Pi, \mathcal{HU})$ respectively. Let $\mathcal{T} = \mathcal{T}(\Sigma, \mathcal{V})$. A substitution θ is an almost identity function from $\mathcal{V}$ to $\mathcal{T}$. The set of substitutions is denoted by Sub. The application of θ to a term t is written as $t\theta$.

2. INCREMENTAL REFINEMENT

Let $\mathcal{T}^\alpha$ be a set of patterns of terms, called abstract terms, and α an abstraction (map) from $\mathcal{T}$ to $\mathcal{T}^\alpha$. The map α induces an equivalence relation $\approx_\alpha$ on $\mathcal{T}$, $(t_1 \approx_\alpha t_2) = (\alpha(t_1) \equiv \alpha(t_2))$. So, abstract terms are identified with the equivalence classes of $\approx_\alpha$, i.e., $\mathcal{T}^\alpha = \mathcal{T}_{/\approx_\alpha}$ where $\mathcal{T}_{/\approx_\alpha}$ is the quotient of $\mathcal{T}$ with respect to $\approx_\alpha$. An abstraction α is called stable if $\forall t, s \in \mathcal{T}.\forall \theta \in Sub.((t \approx_\alpha s) \rightarrow (t\theta \approx_\alpha s\theta))$. Let $E_\alpha = \{\approx_\alpha\}$ be the equality theory on $\mathcal{T}$ induced by α. We extend α to be an abstraction from $\mathcal{A}(\Pi, \mathcal{T})$ to $\mathcal{A}(\Pi, \mathcal{T}^\alpha)$: $\alpha(p(t_1, \cdots, t_n)) = p(\alpha(t_1), \cdots, \alpha(t_n))$. The $\approx_\alpha$ and E_α are extended accordingly. Elements of $\mathcal{A}(\Pi, \mathcal{T}^\alpha)$ are patterns of atoms and hence called abstract atoms.

Let $t, s \in \mathcal{T}(resp.\ \mathcal{A})$ and $\sigma, \theta \in Sub$. We call σ an E_α-unifier of t and s if $t\sigma \approx_\alpha s\sigma$. We say that t and s are E_α-unifiable if they have one or more E_α-unifiers. A substitution σ is more general than another substitution θ with respect to E_α, denoted as $\sigma \leq_{E_\alpha} \theta$, iff there is an $\eta \in Sub$ such that $X\sigma\eta \approx_\alpha X\theta$ for all $X \in \mathcal{V}$. An E_α-unifier σ of t and s is a maximally general E_α-unifier (E_α-mgu) of t and s iff, for any other E_α-unifier θ of t and s, $\theta \not\leq_{E_\alpha} \sigma$.

2.1 Fixpoint/Procedural Abstract Semantics

We now recall abstract semantics of a logic program $\mathcal{P}$ with respect to a stable abstraction α [5]. The success set of $\mathcal{P}$ equals to $\mathbf{T} \uparrow \omega$ where $\mathbf{T} : \wp(\mathcal{HB}) \mapsto \wp(\mathcal{HB})$ [10] is defined

$$\mathbf{T}(I) = \left\{ H\sigma \;\middle|\; \begin{array}{l} H \leftarrow B_1, \cdots, B_m \in \mathcal{P} \\ \sigma \in Sub \wedge \forall i \in [1..m].B_i\sigma \in I \end{array} \right\} \quad (1)$$

For any abstraction α, $\mathcal{P} \cup E_\alpha$ is an equational logic program. The fixpoint semantics of $\mathcal{P} \cup E_\alpha$ given by Jaffar et. al [4] is $\mathbf{T}^\alpha \uparrow \omega$ with $\mathbf{T}^\alpha : \wp(\mathcal{HB}_{/\approx_\alpha}) \mapsto \wp(\mathcal{HB}_{/\approx_\alpha})$ being defined

$$\mathbf{T}^\alpha(I^\flat) = \left\{ [H\sigma]_{\approx_\alpha} \;\middle|\; \begin{array}{l} H \leftarrow B_1, \cdots, B_m \in \mathcal{P} \\ \sigma \in Sub \wedge \forall i \in [1..m].[B_i\sigma]_{\approx_\alpha} \in I^\flat \end{array} \right\} \quad (2)$$

According to [4], $(\mathcal{P} \cup E_\alpha \models A) \leftrightarrow ([A]_{\approx_\alpha} \in \mathbf{T}^\alpha \uparrow \omega)$ for any $A \in \mathcal{HB}$. The following lemma states the $\mathbf{T}^\alpha \uparrow \omega$ is a safe approximation of $\mathbf{T} \uparrow \omega$ with respect to α and hence contains all the success patterns of $\mathcal{P}$ with respect to α.

Lemma 1. $(A \in \mathbf{T} \uparrow \omega \rightarrow [A]_{\approx_\alpha} \in \mathbf{T}^\alpha \uparrow \omega)$ *for any* $A \in \mathcal{HB}$ *if* α *is stable.*

The procedural semantics of an equational logic program $\mathcal{P} \cup E_\alpha$ is the equational SLD resolution with respect to the equality theory E_α, denoted as SLD_α. SLD_α plays the same role for $\mathcal{P} \cup E_\alpha$ as SLD for $\mathcal{P}$. SLD_α differs from SLD in that, in SLD_α, E_α-unification plays the role of normal unification in SLD. We now adapt SLD_α so that it works on equivalence classes of $\approx_\alpha$ on $\mathcal{A}$. Define $[t]_{\approx_\alpha}\theta = [t\theta]_{\approx_\alpha} = \alpha(t\theta)$. Notice that equivalence classes of terms (resp. atoms) are identified with abstract terms (resp. abstract atoms). The application of a substitution θ to an equivalence class $[t]_{\approx_\alpha}$ can be accomplished by applying θ to any term t' in $[t]_{\approx_\alpha}$ and taking $[t'\theta]_{\approx_\alpha}$ as the result because of the stability of α which also allows us to define an E_α-*mgu* of $[t]_{\approx_\alpha}$ and $[s]_{\approx_\alpha}$ as an E_α-*mgu* of t and s. The basic step in SLD_α is defined as follows: Let $G^\flat$ be $\leftarrow A_1^\flat, \cdots, A_p^\flat$ and C a variant $H \leftarrow B_1, \cdots, B_q$ of a clause in $\mathcal{P}$. If σ is an E_α-*mgu* of $A_1^\flat$ and $\alpha(H)$ then $\leftarrow \alpha(B_1)\sigma, \cdots, \alpha(B_q)\sigma, A_2^\flat\sigma, \cdots, A_p^\flat\sigma$ is called E_α-derived from $G^\flat$ and C using E_α-*mgu* σ.

It is proven in [4] that $(\mathcal{P} \cup E_\alpha \models A) \leftrightarrow (\mathcal{P} \vdash_\alpha [A]_{\approx_\alpha})$ where $\mathcal{P} \vdash_\alpha [A]_{\approx_\alpha}$ denotes that $[A]_{\approx_\alpha}$ is provable from $\mathcal{P}$ using SLD_α. This and lemma 1 imply that $\vdash_\alpha$ can be used to verify whether an abstract atom $[A]_{\approx_\alpha}$ is a success pattern of $\mathcal{P}$ with respect to α. In summary,

$$(\mathcal{P} \cup E_\alpha \models A) \leftrightarrow ([A]_{\approx_\alpha} \in \mathbf{T}^\alpha \uparrow \omega) \leftrightarrow (\mathcal{P} \vdash_\alpha [A]_{\approx_\alpha}) \quad (3)$$

2.2 Incremental Refinement Method

Let α_1 and α_2 be two abstractions. Define $\alpha_1 \sqsubseteq \alpha_2$ iff $t \approx_{\alpha_1} s \rightarrow t \approx_{\alpha_2} s$ for all $t, s \in \mathcal{T}$. When $\alpha_1 \sqsubseteq \alpha_2$, we say that α_1 is weaker or finer than α_2 and that α_2 is stronger or coarser than α_1. Note that if $\alpha_1 \sqsubseteq \alpha_2$ then $[t]_{\approx_{\alpha_1}} \subseteq [t]_{\approx_{\alpha_2}}$ for any $t \in \mathcal{T}$. In other words, $\approx_{\alpha_1}$ is a finer partition on $\mathcal{T}$ (and $\mathcal{A}$) than $\approx_{\alpha_2}$. If $\alpha_1 \sqsubseteq \alpha_2$ then $E_{\alpha_1} \models E_{\alpha_2}$ and hence

$$(\alpha_1 \sqsubseteq \alpha_2) \rightarrow ((\mathcal{P} \cup E_{\alpha_1}) \models (\mathcal{P} \cup E_{\alpha_2})) \quad (4)$$

By equations 3 and 4, if $(\alpha_1 \sqsubseteq \alpha_2)$ then

$$\forall A \in \mathcal{HB}.(([A]_{\approx_{\alpha_1}} \in \mathbf{T}^{\alpha_1} \uparrow \omega) \rightarrow ([A]_{\approx_{\alpha_2}} \in \mathbf{T}^{\alpha_2} \uparrow \omega)) \quad (5)$$

Equation 5 lays the foundation for incremental refinement of success patterns. An initial set of success patterns can be obtained by computing $\mathbf{T}^\alpha \uparrow \omega$ which is a safe approximation of $\mathbf{T} \uparrow \omega$ relative to α. If the success patterns in $\mathbf{T}^\alpha \uparrow \omega$ are not finer enough for the application at hand then finer success patterns can be computed by a generate-and-test approach as follows. Firstly, a weaker abstraction α' is formed and candidate elements for $\mathbf{T}^{\alpha'} \uparrow \omega$ are generated from $\mathbf{T}^\alpha \uparrow \omega$. The formation of α' and generation of candidate elements for $\mathbf{T}^{\alpha'} \uparrow \omega$ can be done by splitting one or more equivalence classes of $\approx_\alpha$. Secondly, $SLD_{\alpha'}$ is used to verify if a particular candidate is in $\mathbf{T}^{\alpha'} \uparrow \omega$. This process is repeated until success patterns are fine enough.

When candidates are tested in the above method, each candidate invokes an $SLD_{\alpha'}$ derivation, there might be many identical abstract subgoals (abstract atoms) to solve. Tabling techniques can be employed to improve efficiency [11]. Tabling also solves the problem of non-terminating $SLD_{\alpha'}$ derivations because there are only finite number of abstract atoms.

If $\alpha' \sqsubseteq \alpha$, $[A]_{\approx_{\alpha'}} \subseteq [A]_{\approx_\alpha}$ for any $A \in \mathcal{HB}$, i.e., the $\approx_{\alpha'}$ equivalence class including A is contained in the $\approx_\alpha$ equivalence class including A. Let $\mathcal{R}_{\alpha,\alpha'}$ be a refinement operator that splits an $\approx_\alpha$ equivalence class C into the set of $\approx_{\alpha'}$ equivalence classes contained in C.

$$\mathcal{R}_{\alpha,\alpha'}(C) = \{[A]_{\approx_{\alpha'}} | A \in \mathcal{HB} \wedge [A]_{\approx_\alpha} = C\}$$

Then candidate elements for $\mathbf{T}^{\alpha'} \uparrow \omega$ can be generated from $\mathbf{T}^\alpha \uparrow \omega$ by applying $\tilde{\mathcal{R}}_{\alpha,\alpha'}$ to $\mathbf{T}^\alpha \uparrow \omega$ where $\tilde{\mathcal{R}}_{\alpha,\alpha'}(S) = \bigcup_{C \in S} \mathcal{R}_{\alpha,\alpha'}(C)$.

The following two sections demonstrate the above refinement method via depth and stump abstractions.

3. DEPTH ABSTRACTIONS

The idea of enumerating success patterns to a certain depth is due to [9]. All terms identical to a certain depth are considered equivalent. For example, both $f(a, g(h(0), 1), b)$ and $f(a, g(2, h(h(0))), b)$ have main functor $f/3$ and the first and the third of their arguments are same. Both of their second arguments have $g/2$ as main functor. If this information is enough, then we can use either $f(a, g(h(0), 1), b)$ or $f(a, g(2, h(h(0))), b)$ as a representative of them. Since we are not interested in the arguments of $g/2$ we shall replace each argument of $g/2$ with a special symbol $\_$, denoting any term, i.e., we use $f(a, g(\_, \_), b)$ to represent both $f(a, g(h(0), 1), b)$ and $f(a, g(2, h(h(0))), b)$. $f(a, g(\_, \_), b)$ actually represents an infinite number of terms.

3.1 Depth Abstractions

Let $t = f(t_1, \cdots, t_m)$ be a term. Then t is a depth 0 sub-term of t, and a term s is a depth k sub-term of t if s is a depth $(k-1)$ sub-term of t_i for some $1 \leq i \leq m$. The depth k abstraction of a term t, denoted by $d_k(t)$, is obtained by replacing each depth k sub-term of t with an $\_$. Formally, $d_0(t) = \_$ and $d_k(f(t_1, \cdots, t_m)) = f(d_{k-1}(t_1), \cdots, d_{k-1}(t_m))$ for $k > 0$.

Lemma 2. *For any $k \geq 0$, d_k is stable.*

3.2 Refinement Operator

Let $t^\flat$ be an abstract term denoting an $\approx_{d_{k-1}}$ equivalence class. Let $\Sigma^\text{-} = \Sigma \cup \{\_\}$. Abstract terms for depth k abstraction are terms in $\mathcal{T}(\Sigma^\text{-}, \mathcal{V})$ whose depth k subterms are all $\_$. The operator $\hat{d} : \mathcal{T}(\Sigma^\text{-}, \emptyset) \mapsto \wp(\mathcal{T}(\Sigma^\text{-}, \emptyset))$ defined below splits $t^\flat$ by replacing each $\_$ in $t^\flat$ with an abstract term from $\mathcal{HU}_{/\approx_{d_1}}$ in all possible ways: $\hat{d}(\_) = \{f(\_, \cdots, \_) | f \in \Sigma\}$ and $\hat{d}(g(t_1, \cdots, t_m)) = \{g(r_1, \cdots, r_m) | \forall 1 \leq j \leq m.r_j \in \hat{d}(t_j)\}$. The following extension of $\hat{d}$ gives a refinement operator $\hat{d} : \mathcal{A}(\Pi, \mathcal{T}(\Sigma^\text{-}, \emptyset)) \mapsto \wp(\mathcal{A}(\Pi, \mathcal{T}(\Sigma^\text{-}, \emptyset)))$.

$$\hat{d}(p(t_1, \cdots, t_n)) = \{p(r_1, \cdots, r_n) \mid \forall 1 \leq j \leq n.r_j \in \hat{d}(t_j)\}$$

Lemma 3. *$\mathcal{R}_{d_k, d_{k+1}} = \hat{d}$ for any $k \geq 0$ and $\Sigma \neq \emptyset$.*

3.3 E_{d_k}-Unification Algorithm

Now we present an E_{d_k}-unification algorithm. The following algorithm results from modifying Robinson's unification algorithm [8]. Function occur(k, X, t) is true iff X occurs in t at any depth $j < k$.

Algorithm 1. *This algorithm decides if t_1 and t_2 are E_{d_k}-unifiable and, if so, returns an E_{d_k}-mgu of t_1 and t_2.*

01 function *DU(k,t1,t2)* $\Rightarrow$ *(flag, σ)*
02 { if $k = 0$ then $(flag, \sigma) \Leftarrow (true, \emptyset)$
03 else if t_1 *or* t_2 *is a variable* then

```
04  { let X be the variable and t the other term
05   if X ≡ t then (flag, σ) ⇐ (true, ∅)
06   elseif occur(k,X,t) then (flag, σ)⇐DU(k, X, t{X ↦ t})
07   else (flag, σ) ⇐ (true, {X ↦ d_k(t)})
08  } else
09  { let t_1 ≡ f(x_1, ··· , x_n) and t_2 ≡ g(x_1, ··· , x_m)
10     if f ≠ g or m ≠ n then flag ⇐ false else
11     { j ⇐ 0,   (flag, σ_0) ⇐ (true, ∅)
12       while j < m and flag do
13         { j ⇐ j + 1
14           (flag, τ_j) ⇐ DU(k − 1, x_j σ_{j−1}, y_j σ_{j−1})
15           if flag then σ_j ⇐ σ_{j−1} τ_j   }
16       σ ⇐ σ_m   }   }
17  return (flag, σ)   }
```

Line 06 in algorithm 1 deals with the E_{d_k}-unification of X and t in the case X occurs in t at some depth $j < k$. This does not necessarily mean failure of the E_{d_k}-unification of X and t. For instance, $\{X \mapsto f(Y)\}$ is a E_{d_1}-*mgu* of X and $f(X)$. Algorithm 1 reduces the E_{d_k}-unification of X and t into that of X and $t\{X \mapsto t\}$.

Lemma 4. *If two terms t_1 and t_2 are E_{d_k}-unifiable then algorithm 1 terminates and gives a unique (module renaming) E_{d_k}-mgu of t_1 and t_2. Otherwise, it terminates and reports failure.*

3.4 Refinement of Success Patterns

All depth abstractions are comparable with respect to $\sqsubseteq$. Abstractions corresponding to bigger depths are finer than those corresponding to smaller depths.

Lemma 5. *For any $0 \leq j \leq k$, $d_k \sqsubseteq d_j$.*

Lemma 5 implies that, for any $A \in \mathcal{HB}$, if $[A]_{\approx_{d_k}} \in \mathbf{T}^{d_k} \uparrow \omega$ then $[A]_{\approx_{d_{k-1}}} \in \mathbf{T}^{d_{k-1}} \uparrow \omega$. This enables us to compute $\mathbf{T}^{d_k} \uparrow \omega$ by (i) applying $\tilde{\mathcal{R}}_{d_{k-1},d_k}$ to $\mathbf{T}^{d_{k-1}} \uparrow \omega$ to obtain candidate elements for $\mathbf{T}^{d_k} \uparrow \omega$; and (ii) applying SLD_{d_k} to eliminate false candidates.

Example 1. *This example illustrates incremental refinement of success patterns. Let $\alpha = d_1$ and*

$$\mathcal{P} = \{a(f(c)).\quad b(f(h(c))).\quad p(x) \leftarrow a(x), b(x).\}$$

We have $\mathbf{T}^{d_1} \uparrow \omega = \{a(f(\_)), b(f(\_)), p(f(\_))\}$.

Suppose we want to compute $\mathbf{T}^{d_2} \uparrow \omega$. We first generate a set of candidate elements for $\mathbf{T}^{d_2} \uparrow \omega$ and then use SLD_{d_2} resolution to eliminate false candidates. The generation of candidates is accomplished by applying the refinement operator $\mathcal{R}_{d_1,d_2}$ to elements in $\mathbf{T}^{d_1} \uparrow \omega$. For each element in $\mathbf{T}^{d_1} \uparrow \omega$, $\mathcal{R}_{d_1,d_2}$ generates a set of candidates by substituting each occurrence of $\_$ with elements from $\mathcal{HU}_{/\approx_{d_1}} = \{c, f(\_), h(\_)\}$. Thus, the set of candidates is

$$\left\{ \begin{array}{l} a(f(c)), a(f(f(\_))), a(f(h(\_))), b(f(c)), b(f(f(\_))), \\ b(f(h(\_))), p(f(c)), p(f(f(\_))), p(f(h(\_))) \end{array} \right\}$$

After eliminating false candidates that are not provable from $\mathcal{P}$ using SLD_{d_2}, we have $\mathbf{T}^{d_2} \uparrow \omega = \{a(f(c)), b(f(h(\_)))\}$. The atom $p(f(c))$ has been eliminated as follows. First, $\leftarrow p(f(c))$ is resolved with the clause $p(x) \leftarrow a(x), b(x)$ using E_{d_2}-mgu $\{X \mapsto f(c)\}$, deriving $\leftarrow a(f(c)), b(f(c))$. Then goal $\leftarrow a(f(c))$ is resolved with the unit clause $a(f(c))$ using E_{d_2}-mgu $\emptyset$, deriving $\leftarrow b(f(c))$. However, $\leftarrow b(f(c))$ cannot be resolved with $b(f(h(c))$ because $d_2(b(f(c))) = b(f(c))$ while $d_2(b(f(h(c)))) = b(f(h(\_)))$. There is no other $\vdash_\alpha$ derivations from $\leftarrow p(f(c))$. So, $p(f(c))$ is eliminated.

4. STUMP ABSTRACTIONS

Xu and Warren introduced a family of abstractions, called stump abstractions [12]. The idea is to detail each atom in $\mathbf{T} \uparrow \omega$ to the extent some function symbol has been repeated for a given number of times.

4.1 Stump Abstractions

Let t be a term and s a sub-term of t. We define $\mathsf{fc}(s,t)$ as a function which, for each function symbol g in Σ, counts the number of nodes labeled by g in the path from the root of the term tree of t to but excluding the root of the term tree of s. Let $w \in (\Sigma \mapsto \mathsf{N})$ where N is the set of natural numbers. Define $w \oplus f = w[f \mapsto (w(f) + 1)]$ and $w \ominus f = w[f \mapsto (w(f) - 1)]$. Formally, $\mathsf{fc} : \mathcal{T} \times \mathcal{T} \rightarrowtail (\Sigma \mapsto \mathsf{N})$ is defined as follows. If $s \equiv t$ then $\mathsf{fc}(s,t) = \lambda f.0$. If $t = f(t_1, \cdots, t_m)$, s is a sub-term of t_i and $\mathsf{fc}(s,t_i) = w$ then $\mathsf{fc}(s,t) = w \oplus f$. If $s = g(s_1, \cdots, s_k)$ then the repetition depth of s in t, denoted as $\mathsf{rd}(s,t)$ is defined as $\mathsf{fc}(s,t)(g)$. For instance, letting $t = f(g(h(1), g(1,2)), h(f(h(1), f(3,2))))$, $\mathsf{rd}(f(3,2),t) = 2$ and $\mathsf{rd}(g(1,2),t) = 1$.

Let $t \in \mathcal{T}$, and $w \in \Sigma \mapsto \mathsf{N}$. An abstract term $s_w(t)$ is obtained by replacing each sub-term $s = g(s_1, \cdots, s_k)$ of t satisfying $\mathsf{rd}(s,t) = w(g)$ with $g(\_, \cdots, \_)$. Formally, $s_w(f(t_1, \cdots, t_m)) = f(s_{w \ominus f}(t_1), \cdots, s_{w \ominus f}(t_m))$ if $w(f) \neq 0$ and otherwise, $s_w(f(t_1, \cdots, t_m)) = f(\_, \cdots, \_)$. For instance, $s_w(r(g(s(g(1))))) = r(g(\_))$ if $w = \{r \mapsto 1, g \mapsto 0, s \mapsto 1\}$.

Lemma 6. *For any $w \in \Sigma \mapsto \mathsf{N}$, s_w is stable.*

4.2 Refinement Operator

Let $x, y \in (\Sigma \mapsto \mathsf{N})$ and define $x \trianglelefteq y = \forall f \in \Sigma. x(f) \leq y(f)$. As will be shown later, $x \trianglelefteq y \leftrightarrow s_y \sqsubseteq s_x$. Intuitively, the bigger the limit for each function symbol, the weaker the abstraction. Define $\bar{s} : (\Sigma \mapsto \mathsf{N}) \times \Sigma \mapsto \wp(\mathcal{T}(\Sigma^{\text{-}}, \emptyset))$ as follows: $\bar{s}(w,f) = \{f(\_, \cdots, \_)\}$ if $w(f) = 0$ and, otherwise, $\bar{s}(w,f) = \{f(t_1, \cdots, t_m) | t_j \in \bigcup_{g \in \Sigma} \bar{s}(w \ominus f, g)\}$. Given $w \in \Sigma \mapsto \mathsf{N}$ and $f \in \Sigma$, $\bar{s}(w,f)$ is the set of the abstract terms identifying the $\approx_{s_w}$ equivalence classes of the ground terms whose main functors is f. The following function $\hat{s} : (\Sigma \mapsto \mathsf{N}) \times \mathcal{T}(\Sigma^{\text{-}}, \emptyset) \mapsto \wp(\mathcal{T}(\Sigma^{\text{-}}, \emptyset))$ splits an equivalence class of ground terms for a coarser stump abstraction into the set of equivalence classes of ground terms for a finer stump abstraction: $\hat{s}(w, \_) = \bigcup_{f \in \Sigma} \bar{s}(w,f)$ and $\hat{s}(w, g(t_1, \cdots, t_m)) = \{g(r_1, \cdots, r_m) | \forall 1 \leq j \leq m. r_j \in \hat{s}(w \ominus g, t_j)\}$. A refinement operator is obtained as this extension of $\hat{s}$: $\hat{s}(w, p(t_1, \cdots, t_m)) = \{p(r_1, \cdots, r_m) | \forall 1 \leq j \leq m. r_j \in \hat{s}(w, t_j)\}$.

Lemma 7. *For any $x \trianglelefteq y$, $\mathcal{R}_{s_x,s_y} = \hat{s}(y, \cdot)$.*

4.3 E_{s_w}-Unification Algorithm

The E_{s_w}-unification algorithm is given in algorithm 2. The function *SU* has three parameters. The first parameter w maps each function symbol into the limit of its repetition depth. The second and third parameters are terms to be unified. For any variable X and term t, $\mathrm{occur}(w, X, t)$ is true iff X occurs in $s_w(t)$.

Algorithm 2. *This algorithm decides if t_1 and t_2 are E_{s_w}-unifiable and, if so, returns an E_{s_w}-mgu of t_1 and t_2.*

```
01 function SU(w, t1,t2) ⇒ (flag, σ)
02 { if t1 or t2 is a variable then
03   { let X be the variable and t the other term
04    if X ≡ t then (flag, σ) ⇐ (true, ∅)
05    elseif occur(w,X,t) then (flag, σ)⇐SU(w, X, t{X ↦ t})
06    else (flag, σ) ⇐ (true, {X ↦ s_w(t)})
07   } else
08   { let t1 ≡ f(x1, ··· , xn) and t2 ≡ g(x1, ··· , xm)
09      if f ≠ g or m ≠ n then flag ⇐ false else
10      if w(f) = 0 then (flag, σ) ⇐ (true, ∅) else
11      { j ⇐ 0,   (flag, σ0) ⇐ (true, ∅)
12        while j < m and flag do
13          { j ⇐ j + 1
14            (flag, τj) ⇐ SU(w ⊖ f, xjσj−1, yjσj−1)
15            if flag then σj ⇐ σj−1τj   }
16        σ ⇐ σm   }   }
17 return (flag, σ)   }
```

Line 05 in algorithm 2 deals with the E_{s_w}-unification of X and t in the case X occurs in $s_w(t)$ by reducing the E_{s_w}-unification of X and t into that of X and $t\{X \mapsto t\}$.

Lemma 8. *Let t_1 and t_2 be terms. If t_1 and t_2 are E_{s_w}-unifiable then algorithm 2 terminates and gives a unique (module renaming) E_{s_w}-mgu of t_1 and t_2. Otherwise, it terminates and reports failure.*

4.4 Refinement of Success Patterns

The following lemma establishes the applicability of the incremental refinement method for stump abstractions.

Lemma 9. *For any $x, y \in (\Sigma \mapsto \mathsf{N})$, $x \trianglelefteq y \leftrightarrow s_y \sqsubseteq s_x$.*

Lemma 9 implies that if $[A]_{\approx_{s_y}} \in \mathbf{T}^{s_y} \uparrow \omega$ then $[A]_{\approx_{s_x}} \in \mathbf{T}^{s_x} \uparrow \omega$ for any $x \trianglelefteq y$. This enables us to refine success patterns of $\mathcal{P}$ by increasing repetition depths for some function symbols. The details of the refinement process is omitted because it is the same as the refinement process for depth abstraction with the exception that stump abstractions are used in place of depth abstractions.

5. CONCLUSION AND FUTURE WORK

We have presented a method for incrementally computing success patterns of logic programs for stable abstractions. The method makes use of a fixed-point and a procedural abstract semantics of logic programs with respect to stable abstractions, a refinement operator that splits an equivalence class induced by a coarser abstraction into a set of equivalence classes induced by a finer abstraction, and equational unification. Incremental program analysis has been studied before [2, 3]. In [3], the authors present a method for incrementally analyzing the program after it is modified. The new program is analyzed with respect to the same abstraction as the old one. This paper addresses a different issue, it presents a method for computing a finer analysis of a program from a coarser analysis of the same program. In [2], a method for refining success patterns is proposed. The program is first transformed using coarser success patterns and the transformed program is analyzed to finer success patterns. The method proposed in this paper doesn't transform the program. Instead, it generates candidate finer success patterns from coarser success patterns and then applies abstract procedural semantics to verify the generated candidates.

We have applied the method for depth and stump abstractions by constructing suitable refinement operators and equational unification algorithms. For depth abstractions, abstraction depth can be increased uniformly while for stump abstractions, repetition depth for each function symbol can be increased independently.

For depth abstractions, abstraction depth can only be increased uniformly. That means that every equivalence class has to be split when analysis is refined. It might be better to be able to split some equivalence classes and keep others intact. It is interesting to find out if such a fine-tuning approach will guarantee the stability of the resulting abstraction α which is a prerequisite for using SLD_α to eliminate false candidates. Another interesting topic on incremental refinement is to study the possibility of applying $\mathbf{T}^\alpha$ to eliminate some false candidates before SLD_α is applied. Yet another interesting topic on incremental refinement of success patterns is to combine domain refinement such as that proposed in this paper with compositional approach towards logic program analysis proposed by Codish et. al [1] since compositional approach is the only feasible way to analyse large programs.

6. REFERENCES

[1] M. Codish, S.K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *POPL'93*, pages 451–464. The ACM Press, 1993.

[2] S. Genaim and M. Codish. Incremental refinement of semantic based program analysis for logic programs. In *ACSC'99*. Springer, 1999.

[3] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of constraint logic programs. *ACM TOPLAS*, 22(2):187–223, 2000.

[4] J. Jaffar, J. L. Lassez, and M. J. Maher. A theory of complete logic programs with equality. *Journal of Logic Programming*, 1(3):211–223, 1984.

[5] L. Lu and P. Greenfield. Abstract fixpoint semantics and abstract procedural semantics of definite logic programs. In *ICCL'92*, pages 147–154. IEEE Computer Society Press, 1992.

[6] L. Lu and A. King. Determinacy inference for logic programs. *Lecture Notes in Computer Science*, 3444:108–123. 2005.

[7] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(1–4):181–204, 1992.

[8] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[9] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34(1):227–240, 1984.

[10] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Artificial Intelligence*, 23(10):733–742, 1976.

[11] D. S. Warren. Memoing for logic programs. *CACM*, 35(3):93–111, 1992.

[12] J. Xu and D.S. Warren. A type inference system for Prolog. In *JICSLP'88*, pages 604–619. The MIT Press, 1988.