*J. Software Engineering & Applications*, **2010, 3: 198-207**
doi:10.4236/jsea.2010.33025 Published Online March 2010 (http://www.SciRP.org/journal/jsea)

# Incremental Computation of Success Patterns of Logic Programs

**Lunjin Lu**

Department of Computer Science and Engineering, Oakland University, Rochester, USA.
Email: lunjin@acm.org

Received October 28th, 2009; revised November 26th, 2009; accepted November 30th, 2009.

## ABSTRACT

*A method is presented for incrementally computing success patterns of logic programs. The set of success patterns of a logic program with respect to an abstraction is formulated as the success set of an equational logic program modulo an equality theory that is induced by the abstraction. The method is exemplified via depth and stump abstractions. Also presented are algorithms for computing most general unifiers modulo equality theories induced by depth and stump abstractions.*

***Keywords***: *Incremental Analysis*, *Success Patterns*, *Abstract Interpretation*, *Depth Abstract, Stump Abstraction*, *Logic Programs*

## 1. Introduction

In abstract interpretation, program analyses are viewed as program execution over non-standard data domains. Cousot and Cousot first laid solid logical foundations for abstract interpretations [1,2]. Their idea is to define a collecting semantics for a program which associates each program point with the set of all storage states that are possibly obtained when the execution reaches the point. In practice, an abstraction of the collecting semantics is calculated by simulating over a non-standard data domain the computation of the collecting semantics over the standard data domain. The standard data domain and the non-standard domain are called the concrete domain and the abstract domain respectively.

Abstract interpretation has been used to perform various analyses of logic programs such as occur check analysis [3], mode analysis [4–6], sharing analysis [7,8] and type analysis [6,9,10]. Further more, a number of abstract interpretation frameworks for logic programs have been brought about [5], Jones *et al.* [11], Bruynooghe [9] and Marriott *et al.* [12]. With an abstract interpretation framework, the design of a particular analysis reduces to the design of an abstract domain and a number of abstract operations on the abstract domain. The safeness of the analysis is verified by formalizing a correspondence between the concrete domain and the abstract domain and proving that the abstract operations safely simulate the concrete operations with respect to the correspondence. The correspondence between the abstract domain and the concrete domain can be formalized either as an abstraction (function) from the concrete domain to the abstract domain, or as a concretization (function) from the abstract domain to the concrete domain, or as a joined pair of abstraction and concretization, or as a relation between the concrete domain and the abstract domain. We assume that the correspondence is given as a surjective abstraction from the domain of concrete terms into a domain of abstract terms[1].

A program analysis is currently performed with respect to a fixed abstraction; and different analyses corresponding to different abstractions are performed separately even when there is a strong relationship between them. Take depth abstractions 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 to perform the depth 3 analysis, as we will show later in this paper. This paper is concerned with refining program analyses whereby the result of a coarser analysis corresponding to a stronger abstraction is used to obtain a finer analysis corresponding to a weaker abstraction. In particular, we are concerned with obtaining finer success patterns of a logic program from coarser success patterns of the same program. We introduce an ordering relation on abstractions of terms. We then argue, for a class of

[1]In case an abstraction is not surjective, we can always construct a new of abstract terms by eliminating those abstract terms that are not images of any concrete term under the abstraction. The abstraction is a surjective abstraction from the domain of concrete terms to the new domain of abstract terms.

abstractions, that the set of success patterns of a logic program $P$ with respect to an abstraction $\alpha$ is tantamount to the success set of the equational logic program $P \cup E_\alpha$ where $E_\alpha$ is an equality theory induced by $\alpha$. Therefore, either the fixpoint semantics or the procedural semantics defined for equational logic programs can be used to compute success patterns of logic programs. From this observation, the success patterns of a logic program $P$ can be computed by incremental refinement. A set of coarser success patterns of $P$ relative to a stronger abstraction $\alpha_1$ can be obtained by computing the fixpoint semantics of the equational logic program $P \cup E_{\alpha_1}$. If the success patterns are not fine enough for the application at hand, candidates for finer success patterns relative to a weaker abstraction $\alpha_2$ can be generated from the coarser success patterns and verified by using either the procedural or the fixpoint semantics of equational logic program $P \cup E_{\alpha_2}$. This refinement process is repeated until success patterns are fine enough for the application.

The remainder of this paper is organized as follows. Section 2 presents a fixed-point and a procedural abstract semantics of logic programs for a class of abstractions and lays a foundation for incremental refinement of success patterns of logic programs with respect to that class of abstractions. Sections 3 and 4 devote to incremental refinement of success patterns of logic programs for depth abstractions and stump abstractions respectively. In Section 5, we conclude this paper with a summary of the paper and some points to future work in analysis refinement.

## 2. A Foundation for Incremental Refinement

Let $\Sigma, \Pi, Vars$ be respectively a set of function symbols, a set of predicate symbols and a denumerable set of variables. $Term(\Sigma, V)$ denotes the set of terms constructible from $\Sigma$ and $V$, and $Atom(\Pi, S)$ denotes the set of atoms constructible from $\Pi$ and $S$ where $S$ is a set of terms. The Herbrand universe $\mathcal{HU}$ are theHerbrand base $\mathcal{HB}$ of a logic program $P$ are

$$\mathcal{HU} = Term(\Sigma, \varnothing)$$

and

$$\mathcal{HB} = Atom(\Pi, \mathcal{HU})$$

respectively. Let $Term = Term(\Sigma, Vars)$. Let $Term^\alpha$ be a set of abstract terms, and $\alpha$ be an abstraction from $Term$ to $Term^\alpha$. $\alpha$ induces an equivalence relation $\approx_\alpha$ on $Term$, $(t_1 \approx_\alpha t_2) = (\alpha(t_1) = \alpha(t_2))$. So, abstract terms in $Term^\alpha$ is identified with equivalence classes of $\approx_\alpha$. That is, $Term^\alpha = Term_{/\approx_\alpha}$. $\alpha$ is called stable if $\forall t, s \in Term \forall \theta \in Sub.((t \approx_\alpha s) \rightarrow (t\theta \approx_\alpha s\theta))$ where $Sub$ is the set of substitutions. Let $E_\alpha = \{\approx_\alpha\}$. $E_\alpha$ is an equality theory on $Term$. We extend $\alpha$ to an abstraction from $Atom(\Pi, Term)$ to $Atom(\Pi, Term^\alpha)$ as follows: $\alpha(p(t_1, \cdots, t_n)) = p(\alpha(t_1), \cdots, \alpha(t_n))$. $\approx_\alpha$ and $E_\alpha$ are extended accordingly.

Let $t, s \in Term(or\ Atom)$ and $\sigma, \theta \in Sub$. $\sigma$ is an $E_\alpha$-unifier of $t$ and $s$ if $t\sigma \approx_\alpha s\sigma$. $t$ and $s$ are $E_\alpha$-unifiable if they have one or more $E_\alpha$-unifiers. $\sigma$ is more general than $\theta$ with respect to $E_\alpha$, denoted as $\sigma \le_{E_\alpha} \theta$, iff there is an $\eta \in Sub$ such that $X\sigma\eta \approx_\alpha X\theta$ for all $X \in Vars$. An $E_\alpha$-unifier $\sigma$ of $t$ and $s$ is a maximally general $E_\alpha$-unifier ($E_\alpha$-*mgu*) of $t$ and $s$ iff, for any other $E_\alpha$-unifier $\theta$ of $t$ and $s$, $\theta \not\le_{E_\alpha} \sigma$.

### 2.1 Fixpoint and Procedural Abstract Semantics

This section presents a fixpoint and a procedural abstract semantics of a definite logic program $P$ with respect to a stable abstraction $\alpha$. It is well known that the success set of $P$ is tantamount to the least fixpoint of the following function $\mathbf{T}: \wp(\mathcal{HB}) \mapsto \wp(\mathcal{HB})$ by van Emden and Kowalski [13].

$$\begin{aligned}\mathbf{T}(I) = \{H\sigma : \exists\sigma.\exists H \leftarrow B_1, \cdots, B_m \in P.\\ B_1\sigma \in I \wedge \cdots \wedge B_m\sigma \in I\}\end{aligned} \quad (1)$$

For any logic program $P$ and any abstraction $\alpha$, $P \cup E_\alpha$ is an equational logic program. The fixpoint semantics of $P \cup E_\alpha$ given by Jaffar *et al.* [14] is

$$\mathbf{T}^\alpha \uparrow \omega \text{ with } \mathbf{T}^\alpha : \wp(\mathcal{HB}_{/\approx_\alpha}) \mapsto \wp(\mathcal{HB}_{/\approx_\alpha})$$

being defined as follows.

$$\begin{aligned}\mathbf{T}^\alpha(I^\#) = \{[H\sigma]_{\approx_\alpha} : \exists\sigma.\exists H \leftarrow B_1, \cdots, B_m \in P.\\ [B_1\sigma]_{\approx_\alpha} \in I^\# \wedge \cdots \wedge [B_m\sigma]_{\approx_\alpha} \in I^\#\}\end{aligned} \quad (2)$$

According to Jaffar *et al.* [14],

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

for any $A \in \mathcal{HB}$. We adopt $\mathbf{T}^\alpha \uparrow \omega$ as the fixpoint abstract semantics of $P$ relative to $\alpha$. The following lemma states the $\mathbf{T}^\alpha \uparrow \omega$ is a safe approximation of $\mathbf{T} \uparrow \omega$ with respect to $\alpha$.

**Lemma 1** If $\alpha$ is a stable abstraction then $\forall A \in \mathcal{HB}.(A \in \mathbf{T} \uparrow \omega \rightarrow [A]_{\approx_\alpha} \in \mathbf{T}^\alpha \uparrow \omega)$.

The procedural semantics of an equational logic program $P \cup E_\alpha$ is the equational *SLD* resolution with respect to the equality theory $E_\alpha$, denoted as $SLD_\alpha$. $SLD_\alpha$ plays same role for $P \cup E_\alpha$ as $SLD$ for $P$. $SLD_\alpha$ differs from $SLD$ in the sense that, in $SLD_\alpha$, $E_\alpha$-unification plays the role of normal unification in $SLD$. In the following, we adapt $SLD_\alpha$ so that it works on equivalence classes of $\approx_\alpha$ on $Atom$. 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 $\theta$ to an equivalence class $[t]_{\approx_\alpha}$ can accomplished by applying $\theta$ to any term $t'$ in $[t]_{\approx_\alpha}$ taking $[t'\theta]_{\approx_\alpha}$ as the result because of the stability of $\alpha$ which also allows us to define an $E_\alpha$*-mgu* of $[t]_{\approx_\alpha}$ and $[s]_{\approx_\alpha}$ as an $E_\alpha$*-mgu* of $t$ and $s$. The basic step in $SLD_\alpha$ can now be defined as follows.

**Definition 1** Let $G^\# \equiv \leftarrow A_1^\#, \cdots, A_j^\#, \cdots, A_p^\#$ and $C \equiv H \leftarrow B_1, \cdots, B_q$ be a variant of a clause of P. $W^\#$ is called $E_\alpha$-derived from $G^\#$ and $C$ using $E_\alpha$-mgu $\sigma$ if (1) $\sigma$ is an $E_\alpha$-mgu of $A_j^\#$ and $\alpha(H)$; and (2) $W^\# = \leftarrow A_1^\#\sigma, \cdots, A_{j-1}^\#\sigma, \alpha(B_1)\sigma, \cdots, \alpha(B_q)\sigma, A_{j+1}^\#\sigma, \cdots, A_p^\#\sigma.$

It is proven by Jaffar *et al.* [14] that

$$(P \cup E_\alpha \vdash A) \leftrightarrow (P \rightarrow_{SLD_\alpha} [A]_{\approx_\alpha})$$

where $P \rightarrow_{SLD_\alpha} [A]_{\approx_\alpha}$ denotes that $[A]_{\approx_\alpha}$ is provable from $P$ using $SLD_\alpha$. This implies that $\rightarrow_{SLD_\alpha}$ can be used to verify whether an abstract atom $[A]_{\approx_\alpha}$ is a success pattern of $P$ with respect to $\alpha$ according to lemma 1. In summary,

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

## 2.2 Foundation for Incremental Refinement

Let $\alpha_1$ and $\alpha_2$ be two abstractions. Define $\alpha_1 \subseteq \alpha_2$ iff $t \approx_{\alpha_1} s \rightarrow t \approx_{\alpha_2} s$ for all $t, s \in Term$. When $\alpha_1 \subseteq \alpha_2$, we say that $\alpha_1$ is weaker or finer than $\alpha_2$ and that $\alpha_2$ is stronger or coarser than $\alpha_1$. Note that if $\alpha_1 \subseteq \alpha_2$ then $[t]_{\approx_{\alpha_1}} \subseteq [t]_{\approx_{\alpha_2}}$ for any $t \in Term$. In other words, $\approx_{\alpha_1}$ is a finer partition on $Term$ (and $Atom$) than $\approx_{\alpha_2}$. If $\alpha_1 \subseteq \alpha_2$ then $E_{\alpha_1} \vdash E_{\alpha_2}$. Therefore, we have

$$(\alpha_1 \subseteq \alpha_2) \rightarrow ((P \cup E_{\alpha_1}) \vdash (P \cup E_{\alpha_2})) \quad (4)$$

By Equations (3) and (4),

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

Equation (5) lays a foundation for incremental refinement of success patterns of logic programs. An initial set of the success patterns of a logic program $P$ can be obtained by computing $\mathbf{T}^\alpha \uparrow \omega$ which is a safe approximation of $\mathbf{T} \uparrow \omega$ relative to $\alpha$. 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 $\alpha'$ is formed and candidates elements for $\mathbf{T}^{\alpha'} \uparrow \omega$ are generated from $\mathbf{T}^\alpha \uparrow \omega$. The formation of $\alpha'$ and generation of candidates 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 element is in $\mathbf{T}^{\alpha'} \uparrow \omega$. This process of refinement is repeated until success patterns are fine enough.

If $\alpha' \subseteq \alpha$, $[A]_{\approx_{\alpha'}} \subseteq [A]_{\approx_\alpha}$ 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 $R_{\alpha,\alpha'}$ be a refinement operator that splits an $\approx_\alpha$ equivalence class $C$ into the set of $\approx_{\alpha'}$ equivalence classes contained in $C$.

$$R_{\alpha,\alpha'}(C) = \{[A]_{\approx_{\alpha'}} : A \in \mathsf{HB} \wedge [A]_{\approx_\alpha} = C\}$$

Then candidates elements for $\mathbf{T}^{\alpha'} \uparrow \omega$ can be generated from $\mathbf{T}^\alpha \uparrow \omega$ by applying $R^*_{\alpha,\alpha'}$ to $\mathbf{T}^\alpha \uparrow \omega$ where $R^*_{\alpha,\alpha'}$ is defined $R^*_{\alpha,\alpha'}(S) = \bigcup_{c \in s} R_{\alpha,\alpha'}(C)$.

For a given set $S$ of $\approx_\alpha$ equivalence classes, $R^*_{\alpha,\alpha'}$ returns the union of the sets of $\approx_{\alpha'}$ equivalence classes resulting from applying $R_{\alpha,\alpha'}$ to $\approx_\alpha$ equivalence classes in $S$.

## 2.3 An Example of Incremental Refinement

We illustrate the idea of incremental refinement of success patterns of logic programs by means of depth abstractions proposed by Sato *et al.* [15]. A depth abstraction partitions $Term$ into a finite number of equivalent classes. Two terms belong to the same class iff their term trees are identical to a certain depth *n*, called the depth of abstraction. For example, $h(f(a), g(b))$ is equivalent to $h(f(b), g(a))$ to depth 2. Let $d_n$ denote depth $n$ abstraction. $d_n(t)$ replaces each sub-term of *t* at depth *n* with a _ that denotes any term. Letting $p \in \Pi$, we have

$$d_1(p(f(a), g(b)) = d_1(p(f(b), g(a)) = p(f(\_), g(\_)) .$$

Deferring a formal presentation of depth abstractions until Section 3, we now show how $SLD_\alpha$ can be used when it is necessary to increase the depth of abstraction.

**Example 1** Let $\alpha = d_1$ and *P*={*a*(*f*(*c*)). *b*(*f*(*h*(*c*))). *p*(*x*) ← *a*(*x*), *b*(*x*). }. We have $\mathbf{T}^{d_1} \uparrow 0 = \emptyset$,

$$\mathbf{T}^{d_1} \uparrow 1 = \{a(f(\_)), b(f(\_))\},$$

$$\mathbf{T}^{d_1} \uparrow 2 = \{a(f(\_)), b(f(\_)), p(f(\_))\},$$

$$\mathbf{T}^{d_1} \uparrow 3 = \{a(f(\_)), b(f(\_)), p(f(\_))\},$$

and

$$\mathbf{T}^{d_1} \uparrow \omega = \mathbf{T}^{d_1} \uparrow 3 = \{a(f(\_)), b(f(\_)), p(f(\_))\} .$$

Suppose now we want to be more precise and decide to compute $\mathbf{T}^{d_2} \uparrow \omega$. Note that the set of ground atoms that $\mathbf{T}^{d_2} \uparrow \omega$ approximates is a subset of the set of ground atoms that $\mathbf{T}^{d_1} \uparrow \omega$ approximates. Instead of computing the least fixpoint of $\mathbf{T}^{d_2}$, we compute $\mathbf{T}^{d_2} \uparrow \omega$ by a generate-and-test approach. 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 $R_{d_1,d_2}$ defined in Section 3 to elements in $\mathbf{T}^{d_1} \uparrow \omega$. For each element in $\mathbf{T}^{d_1} \uparrow \omega$, $R_{d_1,d_2}$ generates a set of candidates by substituting each occurrence of _ with every element from $\mathcal{HU}_{/\approx_{d_1}} = \{c, f(\_), h(\_)\}$. Thus, the set of candidates is

$$\{ a(f(c)), a(f(f(\_))), a(f(h(\_))), b(f(c)), b(f(f(\_))),$$
$$b(f(h(\_))), p(f(c)), p(f(f(\_))), p(f(h(\_))) \}.$$

After eliminating candidates that are not provable from *P* using $SLD_{d_2}$ ,we have

$$\mathbf{T}^{d_2} \uparrow \omega = \{a(f(c)), b(f(h(\_)))\} .$$

*p*(*f*(*c*)) has been eliminated as follows. First, ←*p*(*f*(*c*)) is resolved with the clause $p(x) \leftarrow a(x), b(x)$ resulting in $\leftarrow a(f(c)), b(f(c))$.Then goal $\leftarrow a(f(c))$ is resolved with the unit clause $a(f(c))$. However, $\leftarrow b(f(c))$ cannot be resolved with b(f(h(c))) because $d_2(b(f(c)))$= $d_2(b(f(c)))$ while $d_2(b(f(h(c)))) = b(f(h(\_)))$.

The following two sections demonstrate incremental refinement of success patterns of logic programs by considering two families of abstractions, namely depth abstractions and stump abstractions.

## 3. Depth Abstractions

The idea of enumerating success patterns of logic programs to a certain depth is due to Sato and Tamaki [15]. Depth abstraction has been used to ensure termination of an analysis, e.g. [10,16,17]. All terms (resp. atoms) 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, that is, 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.

This section defines depth abstractions, constructs a refinement operator and an equational unification algorithm for such abstractions, and exemplifies incremental refinement of success patterns with respect to depth abstractions.

### 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 $1 \le i \le m$ .

**Definition 2** Let *t* be a term. The depth *k* abstraction of *t*, denoted by $d_k(t)$, is obtained by replacing each depth *k* sub-term of *t* with an _.

$$\begin{array}{rcll} d_k(t) & = & \_ & k = 0 \\ d_k(f(t_1, \cdots, t_m)) & = & f(d_{k-1}(t_1), \cdots, d_{k-1}(t_m)) & k > 0 \end{array}$$

For instance, the depth 2 abstraction of $f(g(X,Y), g(h(Z)))$ is $f(g(\_,\_), g(\_))$, and its depth 3 abstraction is $f(g(X,Y), g(h(\_)))$ .

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

### 3.2 A Refinement Operator for Depth Abstractions

Let $t^\#$ be an abstract term denoting an $\approx_{d_{k-1}}$ equivalence class.

$$\tilde{d} : Term(\Sigma \cup \{\_\}, \emptyset) \mapsto \wp(Term(\Sigma \cup \{\_\}, \emptyset))$$

defined below splits $t^\#$ by replacing each _ in $t^\#$ with an abstract term from $\mathcal{HU}_{/\approx_{d_1}}$ in every possible way.

$$\widetilde{d}(\_) = \{f(\_,\cdots,\_) \mid f \in \Sigma\}$$

$$\widetilde{d}(g(t_1,\cdots,t_m)) = \{g(s_1,\cdots,s_m) \mid \forall 1 \le j \le m. s_j \in d(t_j)\}$$

Its extension yields a refinement operator

$$\widetilde{d} : Atom(\Pi, Term(\Sigma \cup \{\_\},\varnothing)))$$

$$\mapsto \wp(Atom(\Pi, Term(\Sigma \cup \{\_\},\varnothing))).$$

$$\widetilde{d}(p(t_1,\cdots,t_n)) = \{p(s_1,\cdots,s_n) \mid \forall 1 \le j \le n. s_j \in \widetilde{d}(t_j)\}$$

$$\widetilde{d}^* : \wp(Term(\Sigma \cup \{\_\},\varnothing)) \mapsto \wp(Term(\Sigma \cup \{\_\},\varnothing))$$

is the extension of $\widetilde{d}$ to sets of abstract atoms.

$$\widetilde{d}^*(S) = \bigcup_{A^{\#} \in S} \widetilde{d}(A^{\#})$$

**Lemma 3** If $\Sigma \neq \varnothing$ then $R_{d_k, d_{k+1}} = \widetilde{d}$ and $R^*_{d_k, d_{k+1}} = \widetilde{d}^*$ for any $k \ge 0$.

## 3.3 An $E_{d_k}$-Unification Algorithm

Now we present an $E_{d_k}$-unification algorithm and prove its correctness. The following algorithm for $E_{d_k}$-unification results from modifying Robinson's unification algorithm [18]. 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* $E_{d_k}$*-unifiable, returns an* $E_{d_k}$*-mgu of* $t_1$ *and* $t_2$.

```
01  function Dunify($k$, $t_1$, $t_2$) $\Rightarrow$ ($unifiable, \sigma$)
02  begin
03  if $k = 0$ then $(unifiable, \sigma) \leftarrow (true, \varnothing)$
04      else if $t_1$ or $t_2$ is a variablethen
05      begin let $X$ be the variable and $t$ the other term
06          if $X = t$ then $(unifiable, \sigma) \leftarrow (true, \varnothing)$
07          else if $occur(k, X, t)$ then $(unifiable, \sigma) \leftarrow Dunify(k, X, t\{X \mapsto t\})$
08          else $(unifiable, \sigma) \leftarrow (true, \{X \mapsto d_k(t)\})$
09      end else
10      begin let $t_1 = f(x_1,\ldots,x_n)$ and $t_2 = g(x_1,\ldots,x_m)$
11          if $f \neq g$ or $m \neq n$ then $unifiable \leftarrow false$
                else
12          begin $j \leftarrow 0$, $(unifiable, \sigma_0) \leftarrow (true, \varnothing)$
13              while $j < m$ and $unifiable$ do
14              begin $j \leftarrow j+1$
15                  $(unifuable, \tau_j) \leftarrow Dunify\ (k\text{-}1, x_j\sigma_{j\text{-}1}, y_j\sigma_{j\text{-}1})$
16                  if $unifiable$ then $\sigma_j \leftarrow \sigma_{j-1}\tau_j$
17              end
18          $\sigma \leftarrow \sigma_m$
19          end
20      end
21      return $(unifiable, \sigma)$
22  end
```

The line 07 in algorithm 1 deals with $E_{d_k}$-unification of $X$ and $t$ where $X$ 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, $\theta = \{X \mapsto f(Y)\}$ is a $E_{d_1}$*-mgu* of $X$ and $f(X)$. Algorithm 1 reduces the problem of $E_{d_k}$-unification of $X$ and $t$ into the problem of $E_{d_k}$-unification 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, the algorithm terminates and reports the fact.

## 3.4 Refinement of Success Patterns for Depth Abstractions

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

**Lemma 5** For any $0 \le j \le k$, $d_k \subseteq d_j$.

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 refine success patterns of $P$ by increasing abstraction depth. Suppose that success patterns in $\mathbf{T}^{d_{k-1}} \uparrow \omega$ are not fine enough and it is necessary to compute $\mathbf{T}^{d_k} \uparrow \omega$. Rather than throwing away $\mathbf{T}^{d_{k-1}} \uparrow \omega$ and computing $\mathbf{T}^{d_k} \uparrow \omega$ from scratch, we compute $\mathbf{T}^{d_k} \uparrow \omega$ by

1) applying $\widetilde{d}^*$ to $\mathbf{T}^{d_{k-1}} \uparrow \omega$ resulting in a set of candidate elements for $\mathbf{T}^{d_k} \uparrow \omega$ since $\widetilde{d}^*(\mathbf{T}^{d_{k-1}} \uparrow \omega) \supseteq \mathbf{T}^{d_k} \uparrow \omega$;

2) applying $SLD_{d_k}$ to eliminate those candidate elements that are not provable from $P$ using $SLD_{d_k}$.

The following two examples illustrate incremental refinement of success patterns of logic programs with respect to depth abstractions.

**Example 2** Let

$$P = \{p(a,b), p(X,Y) \leftarrow q(X,Y), q(a,b), q(r(X), s(Y)) \leftarrow q(X,Y)\}$$

We have

$\mathbf{T}^{d_1} \uparrow 0 = \quad \varnothing$
$\mathbf{T}^{d_1} \uparrow 1 = \quad \{p(a,b), q(a,b)\}$
$\mathbf{T}^{d_1} \uparrow 2 = \quad \{p(a,b), q(a,b), q(r(\_), s(\_))\}$
$\mathbf{T}^{d_1} \uparrow 3 = \quad \{p(a,b), q(a,b), q(r(\_), s(\_)), p(r(\_), s(\_))\}$
$\mathbf{T}^{d_1} \uparrow 4 = \quad \{p(a,b), q(a,b), q(r(\_), s(\_)), p(r(\_), s(\_))\}$

So,

$\mathbf{T}^{d_1} \uparrow \omega = \mathbf{T}^{d_1} \uparrow 4 = \{p(a,b), q(a,b), q(r(\_), s(\_)), p(r(\_), s(\_))\}$

**Example 3** Let *P* be the same as example 2 and suppose that success patterns in $\mathbf{T}^{d_1} \uparrow \omega$ are not fine enough. We compute $\mathbf{T}^{d_2} \uparrow \omega$ as follows. We first apply $\widetilde{d}^*$ to $\mathbf{T}^{d_1} \uparrow \omega$ resulting in the following candidate elements for $\mathbf{T}^{d_2} \uparrow \omega$.

$$
\begin{aligned}
&p(a,b), q(a,b), q(r(a),s(a)), q(r(a),s(b)), q(r(a),s(r(\_))),\\
&q(r(a),s(s(\_))), q(r(b),s(a)), q(r(b),s(b)), q(r(b),s(r(\_))),\\
&q(r(b),s(s(\_))), q(r(r(\_)),s(a)), q(r(r(\_)),s(b)), q(r(r(\_)),s(r(\_))),\\
&q(r(r(\_)),s(s(\_))), q(r(s(\_)),s(a)), q(r(s(\_)),s(b)), q(r(s(\_)),s(r(\_))),\\
&q(r(s(\_)),s(s(\_))), p(r(a),s(a)), p(r(a),s(b)), p(r(a),s(r(\_))),\\
&p(r(a),s(s(\_))), p(r(b),s(a)), p(r(b),s(b)), p(r(b),s(r(\_))), p(r(b),s(s(\_))),\\
&p(r(r(\_)),s(a)), p(r(r(\_)),s(b)), p(r(r(\_)),s(r(\_))), p(r(r(\_)),s(s(\_))),\\
&p(r(s(\_)),s(a)), p(r(s(\_)),s(b)), p(r(s(\_)),s(r(\_))), p(r(s(\_)),s(s(\_)))
\end{aligned}
$$

We then apply $SLD_{d_2}$ to eliminate those candidate elements that are not provable from *P* by using $SLD_{d_2}$, we have

$\mathbf{T}^{d_2} \uparrow \omega$ = p(a,b),q(a,b),q(r(a)),s(b)),q(r(r(_)),s(s(_))),
p(r(a),s(b)),p(r(r(_)),s(s(_)))

$q(r(r(\_)),s(s(\_)))$ has not been removed because it is provable from *P* by using $SLD_{d_2}$. The $SLD_{d_2}$-refutation process is as follows.

$G_0 = \leftarrow q(r(r(\_)),s(s(\_)))$

$\begin{cases} C_0 = q(r(X1),s(Y1)) \leftarrow q(X1,Y1) \\ \sigma_0 = \{X1/r(X2), Y1/s(Y2)\} \end{cases}$

$G_1 = \leftarrow q(r(X2),s(Y2))$

$\begin{cases} C_1 = q(r(X3),s(Y3)) \leftarrow q(X3,Y3) \\ \sigma_1 = \{X3/X3, Y3/Y2\} \end{cases}$

$G_2 = \leftarrow q(X2,Y2)$

$\begin{cases} C_2 = q(a,b) \\ \sigma_2 = \{X2/a, Y2/b\} \end{cases}$

$G_3 = \varepsilon$

Variables $X2$ and $Y2$ in

$$\sigma_0 = \{X1/r(X2), Y1/s(Y2)\},$$

occur neither in $G_0$ nor in the head of $C_0$. They are introduced by $E_{d_2}$-unification to indicate that they can be replaced by any other terms.

$p(r(s(\_)),s(r(\_)))$ has been eliminated because it is not a provable from *P* by using $SLD_{d_2}$. The $E_{d_2}$-refutation process is as follows.

$G_0 = \leftarrow p(r(s(\_)),s(r(\_))))$
$C_0 = p(X1,Y1) \leftarrow q(X1,Y1)$
$\sigma_0 = \{X1/r(s(X2)), Y1/s(r(Y2))\}$
$G_1 = \leftarrow q(r(r(X2)),s(s(Y2)))$
$C_1 = q(r(X3),s(Y3)) \leftarrow q(X3,Y3)$
$\sigma_1 = \{X3/r(X4), Y3/s(Y4)\}$
$G_2 = \leftarrow q(r(X4),s(Y4))$

The $E_{d_2}$-refutation fails because no clause head $E_{d_2}$-unifies with $q(r(X4),s(Y4))$.

## 4. Stump Abstractions

Xu and Warren have introduced a family of abstractions, called stump abstractions, that reflect recursiveness [19]. The idea is to detail each atom in $\mathbf{T} \uparrow \omega$ to the extent in which some function symbol has been repeated for a given times.

This section defines stump abstractions, constructs a refinement operator and an equational unification algorithm for such abstractions, and exemplifies incremental refinement of success patterns of logic programs with respect to stump abstractions.

### 4.1 Stump Abstractions

Let *t* be a term and *s* a sub-term of *t*. We define $fc(s,t)$ as a function which, for each function symbol *g* in $\Sigma$, registers the number of nodes labelled 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 $w \in (\Sigma \mapsto N)$ where *N* is the set of natural numbers. Define $w \oplus f = w[f \mapsto w(f)+1]$ and if $w(f) > 0$ then $w!f =$

$w[f \mapsto w(f)-1]$ .Define $fc : Term \times Term \to (\Sigma \mapsto N)$ as follows. If $s \equiv t$ then $fc(s,t) = \lambda f.0$ . If $t = f(t_1,\cdots,t_m)$ and $fc(s,t_i) = w$ for some $1 \le i \le m$ then $fc(s,t) = w \oplus f$ .Otherwise, $fc(s,t)$ is undefined. If $s = g(s_1,\cdots,s_k)$ then the repetition depth of $s$ in $t$, denoted as $rd(s,t)$ is defined as $fc(s,t)(g)$ .For instance, letting $t = f(g(h(1),g(1,2)),h(f(h(1),f(3,2))))$ , $rd(f(3,2),t) = 2$ , and $rd(g(1,2), \quad t) = 1$ .

**Definition 3** Let $t \in Term$ , and $w \in \Sigma \mapsto N$ . $s_w(t)$ is obtained by replacing each sub-term $s = g(s_1,\cdots,s_k)$ of $t$ satisfying $rd(s,t) = w(g)$ with $g(\_,\cdots,\_)$ . Formally,

$$s_w(f(t_1,\cdots,t_m)) = \begin{cases} f(s_{w!f}(t_1),\cdots,s_{w!f}(t_m)) & w(f) \ne 0 \\ f(\_,\cdots,\_) & w(f) = 0 \end{cases}$$

For instance, letting $w = \{r \mapsto 1, g \mapsto 0, s \mapsto 1\}$ , $s_w(r(g(s(g(1))))) = r(g(\_))$ .

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

## 4.2 A Refinement Operator for Stump Abstractions

Let $x, y \in (\Sigma \mapsto N)$ and define

$$x\langle\langle y = \forall f \in \Sigma . x(f) \le y(f) .$$

As shown later, $x\langle\langle y \leftrightarrow s_y \subseteq s_x$ . Intuitively, the bigger the limit for each function symbol, the weaker the abstraction.

**Definition 4** Define

$$\bar{s} : (\Sigma \mapsto N) \times \Sigma \mapsto \wp(Term(\Sigma \cup \{\_\},\emptyset))$$

as follows.

$$\begin{array}{lcll} \bar{s}(w,f) & = & \{f(\_,\cdots,\_)\} & w(f) = 0 \\ \bar{s}(w,f) & = & \{f(t_1,\cdots,t_m) \mid t_j \in \bigcup_{g\in\Sigma} \bar{s}(w!f,g)\} & w(f) \ne 0 \end{array}$$

For given $w \in \Sigma \mapsto N$ and $f \in \Sigma$ , $\bar{s}(w,f)$ is the set of the abstract terms identifying the $\approx_{s_w}$ equivalence classes of those ground terms whose main functors is $f$.

The following defined function

$$\tilde{s} : (\Sigma \mapsto N) \times Term(\Sigma \cup \{\_\},\emptyset) \mapsto \wp(Term(\Sigma \cup \{\_\},\emptyset))$$

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.

$$\tilde{s}(w,\_) = \bigcup_{f\in\Sigma} \bar{s}(w,f)$$

$$\tilde{s}(w, g(t_1,\cdots,t_m)) = \{g(s_1,\cdots,s_m) \mid \forall 1 \le j \le m . s_j \in \tilde{s}(w!g,t_j)\}$$

Its extension as in the following gives rise to a refinement operator for stump abstractions

$$\tilde{s} : (\Sigma \mapsto N) \times Atom(\Pi, Term(\Sigma \cup \{\_\}, \emptyset)) \mapsto \wp(Atom(\Pi, Term(\Sigma \cup \{\_\},\emptyset)))$$

$$\tilde{s}(w, p(t_1,\cdots,t_m)) = \{p(s_1,\cdots,s_m) \mid \forall 1 \le j \le m . s_j \in \tilde{s}(w,t_j)\}$$

$$\tilde{s}^* : (\Sigma \mapsto N) \times \wp(Atom(\Pi, Term(\Sigma \cup \{\_\},\emptyset))) \mapsto \wp(Atom(\Pi, Term(\Sigma \cup \{\_\},\emptyset)))$$

is the extension of $\tilde{s}$ to sets of abstract atoms.

$$\tilde{s}^*(w,S) = \bigcup_{A^\# \in S} \tilde{s}(w, A^\#)$$

**Lemma 7** For any $x\langle\langle y$ , $R_{s_x,s_y} = \tilde{s}(y,\cdot)$ and $R^*_{s_x,s_y} = \tilde{s}^*(y,\cdot)$ .

## 4.3 An $E_{s_w}$ -Unification Algorithm

The $E_{s_w}$ -unification algorithm is given in algorithm 2. The function *Sunif* 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*, $occur(w,X,t)$ 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 $t_2$.

```
function Sunify( $w$ , $t_1, t_2$ ) $\Rightarrow$ ( $unifiable, \sigma$ )
begin
 if $t_1$ or $t_2$ is a variable then
 begin let $X$ be the variable and $t$ the other term
   if $X = t$ then $(unifiable,\sigma) \leftarrow (true,\emptyset)$
   else if $occur(w,X,t)$ then $(unifiable,\sigma)) \leftarrow Sunify(w,X,t\{X \mapsto t\}$
   else $(unifiable,\sigma) \leftarrow (true,\{X \mapsto s_w(t)\})$
 end else
 begin let $t_1 = f(x_1,\cdots,x_n)$ and $t_2 = g(x_1,\cdots,x_m)$
   if $f \ne g$ or $m \ne n$ then $unifiable \leftarrow false$ else
   if $w(f) = 0$ then $(unifiable,\sigma) \leftarrow (true,\emptyset)$
     else
     begin $j \leftarrow 0$, $(unifiable,\sigma_0) \leftarrow (true,\emptyset)$
       while $j < m$ and $unifiable$ do
         begin $j \leftarrow j+1$
           $(unifiable, \tau_j) \leftarrow Sunify(w!f, x_j\sigma_{j-1}, y_j\sigma_{j-1})$
            if $unifiable$ then $\sigma_j \leftarrow \sigma_{j-1}\tau_j$
         end
       $\sigma \leftarrow \sigma_m$
     end
 end
```

```
21    return $(unifiable, \sigma)$
22  end
```

The line 06 in algorithm 2 deals with $E_{s_w}$-unification of $X$ and $t$ where $X$ occurs in $s_w(t)$ by reducing the problem of $E_{s_w}$-unification of $X$ and $t$ into the problem of $E_{s_w}$-unification 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 an unique (module renaming) $E_{s_w}$-mgu of $t_1$ and $t_2$.Otherwise, the algorithm terminates and reports the fact.

## 4.4 Refinement of Success Patterns for Stump Abstractions

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

**Lemma 9** For any $x, y \in (\Sigma \mapsto N)$, $x\langle\langle y \leftrightarrow s_y \subseteq s_x$.

**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 $x\langle\langle y$. This enables us to refine success patterns of $P$ by increasing repetition depths for some function symbols. Suppose that success patterns in $\mathbf{T}^{s_x} \uparrow \omega$ are not fine enough and it is necessary to compute $\mathbf{T}^{s_y} \uparrow \omega$ for some $y$ such that $x\langle\langle y$. Rather than throwing away $\mathbf{T}^{s_x} \uparrow \omega$ and computing $\mathbf{T}^{s_y} \uparrow \omega$ from scratch, we compute $\mathbf{T}^{s_y} \uparrow \omega$ by

1) applying $\tilde{s}^*(y,\cdot)$ to $\mathbf{T}^{s_x} \uparrow \omega$ resulting in a set of candidate elements for $\mathbf{T}^{s_y} \uparrow \omega$ since $\mathbf{T}^{s_y} \uparrow \omega \subseteq \tilde{s}^*(y, \mathbf{T}^{s_x} \uparrow \omega)$;

2) applying $SLD_{s_y}$ to eliminate those candidate elements that are not from $P$ using $SLD_{s_y}$.

The following two examples illustrate incremental refinement of success patterns for stump abstractions.

**Example 4** Let

$P = \{p(a,b), p(X,Y) \leftarrow q(X,Y), q(a,b), q(r(X),s(Y)) \leftarrow q(X,Y)\}$

We have

$$\mathbf{T}^{s_{\lambda f.1}} \uparrow 0 = \varnothing$$

$$\mathbf{T}^{s_{\lambda f.1}} \uparrow 1 = \{p(a,b), q(a,b)\}$$

$$\mathbf{T}^{s_{\lambda f.1}} \uparrow 2 = \{p(a,b), q(a,b), q(r(a),s(b))\}$$

$$\mathbf{T}^{s_{\lambda f.1}} \uparrow 3 = \begin{Bmatrix} p(a,b), q(a,b), q(r(a),s(b)), \\ p(r(a),s(b)), q(r(r(\_)),s(s(\_))) \end{Bmatrix}$$

$$\mathbf{T}^{s_{\lambda f.1}} \uparrow 4 = \begin{Bmatrix} p(a,b), q(a,b), q(r(a),s(b)), p(r(a),s(b)), \\ q(r(r(\_)),s(s(\_))), p(r(r(\_)),s(s(\_))) \end{Bmatrix}$$

$$\mathbf{T}^{s_{\lambda f.1}} \uparrow 5 = \mathbf{T}^{s_{\lambda f.1}} \uparrow 4$$

So, $\mathbf{T}^{s_{\lambda f.1}} \uparrow \omega = \mathbf{T}^{s_{\lambda f.1}} \uparrow 5 =$

$$\begin{Bmatrix} p(a,b), q(a,b), q(r(a),s(b)), p(r(a),s(b)), \\ q(r(r(\_)),s(s(\_))), p(r(r(\_)),s(s(\_))) \end{Bmatrix}$$

**Example 5** Let $P$ be the same as example 4. Suppose that success patterns in $\mathbf{T}^{s_{\lambda f.1}} \uparrow \omega$ are not fine enough. We compute $\mathbf{T}^{s_{\lambda f.2}} \uparrow \omega$ as follows. We first compute $\tilde{s}^*(\lambda f.2, \mathbf{T}^{s_{\lambda f.1}} \uparrow \omega)$ and then use $SLD_{s_{\lambda f.2}}$ to eliminate those candidates in $\tilde{s}^*(\lambda f.2, \mathbf{T}^{s_{\lambda f.1}} \uparrow \omega)$ that are not provable from $P$ using $SLD_{s_{\lambda f.2}}$. The result is

$$\mathbf{T}^{s_{\lambda f.2}} \uparrow \omega = \begin{Bmatrix} p(a,b), q(a,b), q(r(a),s(b)), p(r(a),s(b)), \\ q(r(r(a)),s(s(b))), p(r(r(a)),s(s(b))), \\ q(r(r(r(\_))),s(s(s(\_)))), p(r(r(r(\_))),s(s(s(\_)))) \end{Bmatrix}$$

$q(r(r(r(\_))),s(s(s(\_))))$ has not been removed because it is provable from $P$ using $SLD_{s_{\lambda f.2}}$ as follows.

$$\begin{aligned}
G_0 &= \leftarrow q(r(r(r(\_))),s(s(s(\_)))) \\
&\quad C_0 = q(r(X1),s(Y1)) \leftarrow q(X1,Y1) \\
&\quad \sigma_0 = \{X1/r(r(X2)), Y1/s(s(Y2))\} \\
G_1 &= \leftarrow q(r(r(X2)),s(s(Y2))) \\
&\quad C_1 = q(r(X3),s(Y3)) \leftarrow q(X3,Y3) \\
&\quad \sigma_1 = \{X3/r(X2), Y3/s(Y2)\} \\
G_2 &= \leftarrow q(r(X2),s(Y2)) \\
&\quad C_2 = q(r(X4),s(Y4)) \leftarrow q(X4,Y4) \\
&\quad \sigma_2 = \{X4/X2, Y4/Y2\} \\
G_3 &= \leftarrow q(X2,Y2) \\
&\quad C_3 = q(a,b) \\
&\quad \sigma_3 = \{X2/a, Y2/b\} \\
G_4 &= \varepsilon
\end{aligned}$$

$q(r(r(s(a))),s(s(\_)))$ has been eliminated because it can not proved from $P$ using $SLD_{s_{\lambda f.2}}$ as shown in the following.

$$\begin{aligned}
G_0 &= \leftarrow q(r(r(s(a))),s(s(\_))) \\
&\quad C_0 = q(r(X1),s(Y1)) \leftarrow q(X1,Y1) \\
&\quad \sigma_0 = \{X1/r(s(a)), Y1/s(s(Y2))\} \\
G_1 &= \leftarrow q(r(s(a)),s(s(Y2))) \\
&\quad C_1 = q(r(X3),s(Y3)) \leftarrow q(X3,Y3) \\
&\quad \sigma_1 = \{X3/s(a), Y3/s(Y2)\} \\
G_2 &= \leftarrow q(s(a),s(Y2))
\end{aligned}$$

The refutation process fails because there is no clause of $P$ whose head $E_{s_{\lambda f.2}}$-unifies with $q(s(a),s(Y2))$.

## 5. Conclusions and Future Work

We have proposed a method for incrementally computing success patterns of logic programs for stable abstractions. We have introduced a partial order on abstractions to reflect relative strength of 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. The refinement operator is specified.

We have applied the method for incremental refinement of success patterns of logic programs for depth abstractions 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 would be better to be able to split some equivalence classes and keep others intact. However, it is not clear if such a fine-tuning approach will guarantee the stability of the resulting abstraction $\alpha$ which is a prerequisite of using $SLD_{\alpha}$ to eliminate false candidates.

Another interesting topic on incremental refinement of success patterns of logic programs is to study the possibility of applying $\mathbf{T}^{\alpha'}$ to eliminate false candidates where $\alpha'$ is the abstraction resulting from refinement. Yet another interesting topic on incremental refinement of success patterns of logic programs is to combine domain refinement such as that proposed in this paper with compositional approach towards logic program analysis proposed by Codish *et al.* [3] since compositional approach is the only feasible way to analyze large programs. It is necessary to study the interaction between the refinement of analyses of program modules and the composition of analyses of program modules.

## 6. Acknowledgements

This work was supported, in part, by NSF grant CCR-0131862.

## REFERENCES

[1] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, The ACM Press, New York, pp. 269–282, 1979.

[2] P. Cousot and R. Cousot, "Abstract interpretation and application to logic programs," The Journal of Logic Programming, Vol. 13, No. 2–3, pp. 103–179, 1992.

[3] H. Søndergaard, "An application of abstract interpretation of logic programs: Occur check problem," In: B. Robinet and R. Wilhelm, Ed., European Symposium on Programming, Lecture Notes in Computer Science, Springer, Vol. 213, pp. 324–338, 1986.

[4] C. Mellish, "Some global optimizations for a Prolog compiler," Journal of Logic Programming, Vol. 2, No. 1, pp. 43–66, 1985.

[5] C. Mellish, "Abstract interpretation of Prolog programs," In: S. Abramsky and C. Hankin, Ed., Abstract interpretation of declarative languages, Ellis Horwood Ltd., pp. 181–198, 1987.

[6] M. Bruynooghe and G. Janssens, "An instance of abstract interpretation integrating type and mode inferencing," Proceedings of the Fifth International Conference and Symposium on Logic Programming, The MIT Press, Seattle, pp. 669–683, 15–19 August 1988.

[7] D. Jacobs and A. Langen, "Static analysis of logic programs for independent and parallelism," Journal of Logic Programming, Vol. 13, No. 2–3, pp. 291–314, 1992.

[8] X. Li, A. King, and L. Lu, "Collapsing closures," In: S. Etalle and M. Truszczynski, Ed., Proceedings of the Twenty Second International Conference on Logic Programming, Lecture Notes in Computer Science, Vol. 4079, pp. 148–162, 2006.

[9] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen, "Abstract interpretation: Towards the global optimisation of Prolog programs," Proceedings of the 1987 Symposium on Logic Programming, The IEEE Computer Society Press, San Francisco, pp. 192–204, 31 August–4 September 1987.

[10] L. Lu, "Improving precision of type analysis using non-discriminative union," Theory and Practice of Logic Programming, Vol. 8, pp. 33–80, 2008.

[11] K. Marriott, H. Søndergaard, and N. D. Jones, "Denotational abstract interpretation of logic programs," ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, pp. 607–648, 1994.

[12] K. Marriott and H. Søndergaard, "Bottom-up abstract interpretation of logic programs," Proceedings of the Fifth International Conference and Symposium on Logic Pro- gramming, The MIT Press, Seattle, pp. 733–748, 15–19 August 1988.

[13] M. H. van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," Artificial Intelligence, Vol. 23, No. 10, pp. 733–742, 1976.

[14] J. Jaffar, J. L. Lassez, and M. J. Maher, "Theory of complete logic programs with equality," Journal of Logic Programming, Vol. 1, No. 3, pp. 211–23, October 1984.

[15] T. Sato and H. Tamaki, "Enumeration of success patterns in logic programs," Theoretical Computer Science, Vol. 34, No. 1, pp. 227–240, 1984.

[16] P. M. Hill and F. Spoto, "Generalizing Def and Pos to type analysis," Journal of Logic and Computation, Vol.

12, No. 3, pp. 497–542, 2002.

[17] M. Li, Z. Li, H. Chen, and T. Zhou, "A novel derivation framework for definite logic program," Electronic Notes in Theoretical Computer Science, Vol. 212, pp. 71–85, 2008.

[18] J. A. Robinson, "A machine-oriented logic based on the resolution principle," Journal of the ACM, Vol. 12, No. 1, pp. 23–41, 1965.

[19] J. Xu and D. S. Warren, "A type inference system for Prolog," Proceedings of the 5th International Conference and Symposium on Logic Programming, The MIT Press, Seattle, pp. 604–619, 15–19 August 1988.

[20] M. Codish, S. K. Debray, and R. Giacobazzi, "Compositional analysis of modular logic programs," Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, The ACM Press, New York, USA, pp. 451–464, January 1993.