An Algorithm for Finite Approximations of Definite Programs and its Implementation in Prolog

Lunjin Lu*
School of Computer Science, The University of Birmingham
Birmingham, The United Kingdom

Peter Greenfield
School of Computer Science, The University of Birmingham
Birmingham, The United Kingdom

Abstract

In this paper, we first review a bottom-up abstract interpretation framework for finite approximations of definite programs that are characterised by abstraction functions. For each such abstraction function α, a finite approximation of the least fixpoint semantics is given as the least fixpoint of a function $\Phi_{P,\alpha}$ over an abstract domain in a similar way that the least fixpoint semantics is given as the least fixpoint of a function T_P over the concrete domain. Then the derivation of an algorithm for the computation of $lfp(\Phi_{P,\alpha})$ is detailed and its implementation in Prolog is presented. The final algorithm is efficient in both storage usage and time usage. The efficiency in its storage usage is obtained by formulating another function $\Psi_{P,\alpha}$ and establishing that $lfp(\Psi_{P,\alpha}) = lfp(\Phi_{P,\alpha})$. The time efficiency is obtained by using heuristic knowledge derived from program text and information accumulated during the computation of $lfp(\Phi_{P,\alpha})$. The time efficiency is exemplified through depth abstractions and stump abstractions.

1 Introduction

Let us assume that we are dealing with a definite program P, and that its Herbrand universe, Herbrand base and least fixpoint semantics are U_P, B_P and M_P respectively. Let $C_1, \ldots, C_N$ be the clauses of P. Let $C_i = A_{i,0} \leftarrow A_{i,1} \wedge \cdots \wedge A_{i,m_i}$. Let the set of ground instances of an atom A be denoted as $[A]$, i.e. $[A] = \{A\sigma | \exists\sigma.A\sigma \in B_P\}$.

The fixpoint semantics of definite programs given by van Emden and Kowalski [11] may be rewritten as follows.

Definition 1.1 (van Emden and Kowalski) The fixpoint semantics of a definite program P is the least fixpoint of function $T_P : \wp(B_P) \mapsto \wp(B_P)$ defined as follows.

$$T_P(I) = \{A_{i,0}\sigma \ |\exists\sigma.\exists C_i \in P.(A_{i,1}\sigma \wedge \ldots \wedge A_{i,m_i}\sigma \in I \wedge A_{i,0}\sigma \in B_P)\} \quad (1)$$

*Supported by the Sino-British Friendship Scholarship Scheme.

The size of $M_P = lfp(T_P)$ is usually infinite for a non-trivial program. This makes the computation infeasible. So, finite approximations of $lfp(T_P)$ must be used to derive program properties for program analyses.

Sato *et al.* [10] define a class of finite approximations of definite programs by means of depth abstractions. Xu *et al.* [12] present another class of finite approximations of definite programs by means of stump abstractions. Each of these abstractions is characterised by an abstraction function from a concrete domain to an abstract domain. Each element in the abstract domain represents a set of elements in the concrete domain.

Lu and Greenfield [5] generalise Sato's depth abstractions and Xu's stump abstractions and present a bottom-up abstract interpretation framework for finite approximations. In this paper, we derive an algorithm for the computation of such finite approximations and present its implementation as a Prolog meta-program. Section 2 reviews the bottom-up abstract interpretation framework. Section 3 presents the derivation of an algorithm for the computation of such finite approximations. Section 4 outlines an implementation of the algorithm as a Prolog meta-program. Section 5 exemplifies the algorithm developed through depth abstractions and stump abstractions. Section 6 concludes the paper.

2 A class of finite approximations

This section presents an abstract fixpoint semantics of a definite program for an abstraction function α.

Let $Terms$ be the set of terms, α be a function from $Terms$ to $\mathcal{T}erms$, the set of abstract terms. The concrete domain we consider is a sub-lattice of $(\wp(Terms), \subseteq)$ and the abstract domain a sub-lattice of $(\wp(\mathcal{T}erms), \subseteq)$. Throughout this paper, abstract objects are denoted by bold face lowercase letters or calligraphic uppercase letters.

The abstraction function α induces an equivalence relation $\equiv_\alpha$ on $Terms$, $t_1 \equiv_\alpha t_2$ iff $\alpha(t_1) = \alpha(t_2)$. When $t_1 \equiv_\alpha t_2$, t_1 (or t_2) is called α-equivalent to t_2 (or t_1). It is possible that two or more instances of a term t belong to the same equivalent class, i.e., there exist some θ_1 and θ_2 such that $t\theta_1 \equiv_\alpha t\theta_2$. The fact that $t\theta_1 \equiv_\alpha t\theta_2$ is denoted as $\theta_1 \simeq_{\alpha,t} \theta_2$.

We generalise the definition of function α from $Terms \mapsto \mathcal{T}erms$ to $\wp(Terms) \mapsto \wp(\mathcal{T}erms)$ with definition $\alpha(S) = \{\alpha(s) | s \in S\}$. We define a concretion function $\gamma : \mathcal{T}erms \mapsto \wp(Terms)$ as $\gamma(\mathbf{t}) = \{t | t \in Terms, \mathbf{t} = \alpha(t)\}$, and generalise γ from $\mathcal{T}erms \mapsto \wp(Terms)$ to $\wp(\mathcal{T}erms) \mapsto \wp(Terms)$ with definition $\gamma(S) = \cup_{s \in S} \gamma(s)$.

$\alpha(M_P)$ is an ideal approximation of M_P with respect to α. That is, any other correct approximation of M_P is a superset of $\alpha(M_P)$ and hence less precise than $\alpha(M_P)$. However, it is impossible to obtain $\alpha(M_P)$ without actually calculating M_P. So, M_P must be approximated in another way. Lu and Greenfield [5] define an approximation of M_P for each abstraction function α as follows and establish its correctness.

Definition 2.1 The finite approximation of P characterised by the abstraction function α is the least fixpoint of function $\Phi_{P,\alpha} : \wp(\alpha(B_P)) \mapsto \wp(\alpha(B_P))$ defined as the following.

$$\Phi_{P,\alpha}(\mathcal{I}) = \left\{ \alpha(A_{i,0}\sigma) \,\middle|\, \begin{array}{l} \exists\sigma.\exists C_i \in P. \\ (A_{i,0}\sigma \in B_P \wedge \ldots \wedge A_{i,m_i}\sigma \in B_P \wedge \\ \alpha(A_{i,1}\sigma) \in \mathcal{I} \wedge \ldots \wedge \alpha(A_{i,m_i}\sigma) \in \mathcal{I}) \end{array} \right\} \tag{2}$$

$\Phi_{P,\alpha}$ is a monotone and $(\wp(\alpha(B_P)), \subseteq)$ is a complete lattice with set union $\cup$ as least upper bound operator. So, $\Phi_{P,\alpha}$ has the least fixpoint $lfp(\Phi_{P,\alpha}) = \cup_{i=0}^{i=\infty}\Phi^i_{P,\alpha}(\emptyset)$.

3 An Algorithm for Computing $lfp(\Phi_{P,\alpha})$

In this section we present in detail the derivation of an efficient algorithm for the computation $lfp(\Phi_{P,\alpha})$. In sub-section 3.1, an inefficient but intuitive algorithm is presented and several possible improvements in its storage usage and time usage are suggested. In sub-section 3.2, the improvement in storage usage is justified and an improved algorithm is formulated. In sub-section 3.3, the improvements in time usage are made by using heuristics derived from the program text resulting in a more efficient algorithm.

In order to compute $lfp(\Phi_{P,\alpha})$, the definition of $\Phi_{P,\alpha}$ is reformulated as follows.

$$\Phi_{P,\alpha}(\mathcal{I}) = \left\{ \mathcal{A} \,\middle|\, \begin{array}{l} \exists\sigma.\exists C_i \in P. \\ \alpha(A_{i,1}\sigma) \in \mathcal{I} \wedge \cdots \wedge \alpha(A_{i,m_i}\sigma) \in \mathcal{I} \wedge \mathcal{A} \in \alpha([A_{i,0}\sigma]) \end{array} \right\} \tag{3}$$

The equivalence between equation 3 and equation 2 may be established in a straightforward manner by mathematical induction.

3.1 An intuitive algorithm

We now present an inefficient but intuitive algorithm for the computation of $lfp(\Phi_{P,\alpha})$ from which the final efficient algorithm evolves.

Definition 3.1 ($\phi_{C_i,\alpha} : \alpha(B_P) \mapsto \alpha(B_P)$)

$$\phi_{C_i,\alpha}(\mathcal{I}) = \left\{ \mathcal{A} \,\middle|\, \begin{array}{l} \exists\sigma.\alpha(A_{i,1}\sigma) \in \mathcal{I} \wedge \cdots \wedge \alpha(A_{i,m_i}\sigma) \in \mathcal{I} \wedge \\ \mathcal{A} \in \alpha([A_{i,0}\sigma]) \end{array} \right\} \tag{4}$$

The reading of $\phi_{C_i,\alpha}(\mathcal{I})$ is the set of α-abstractions of the ground atoms that may be derived in one step from $\gamma(\mathcal{I})$ and C_i. The following results from equation 3 and equation 4.

$$\Phi_{P,\alpha} = \phi_{C_1,\alpha} \cup \ldots \cup \phi_{C_N,\alpha} \tag{5}$$

Equation 5 can be utilised for the computation of $lfp(\Phi_{P,\alpha})$ resulting in the algorithm 3.1. $lfp(\Phi_{P,\alpha})$ is obtained by computing $\Phi^1_{P,\alpha}(\emptyset), \Phi^2_{P,\alpha}(\emptyset), \ldots$ until $\Phi^n_{P,\alpha}(\emptyset) = \Phi^{n+1}_{P,\alpha}(\emptyset)$ for some n.

Algorithm 3.1 An algorithm for computing $lfp(\Phi_{P,\alpha})$.

$\mathcal{J} \leftarrow \emptyset$
repeat
    $\mathcal{I} \leftarrow \mathcal{J}$
    $\mathcal{J} \leftarrow \Phi_{P,\alpha}(\mathcal{I}) = \phi_{C_1,\alpha}(\mathcal{I}) \cup \ldots \cup \phi_{C_N,\alpha}(\mathcal{I})$
until $\mathcal{J} = \mathcal{I}$
return $\mathcal{J}$

Before we discuss algorithm 3.1, let us consider the relation between the partial results in the algorithm.

Lemma 3.1 For any $n \geq 0$, $\Phi^n_{P,\alpha}(\emptyset) \subseteq \Phi^{n+1}_{P,\alpha}(\emptyset)$.

Proof The proof can be done in a straightforward manner by induction on n. $\Box$

Algorithm 3.1 is inefficient in its storage usage since it is necessary to store two partial results $\mathcal{I}$ and $\mathcal{J}$ when computing $\Phi^n_{P,\alpha}(\emptyset)$. We shall later show that only one partial result is needed.

Algorithm 3.1 is also inefficient in its time usage for the following reasons.

- It is possible that $\phi_{C_i,\alpha}(\Phi^n_{P,\alpha}(\emptyset))$ generates some elements that are already in $\Phi^n_{P,\alpha}(\emptyset)$. The time devoted to generating these elements is wasted since $\Phi^n_{P,\alpha}(\emptyset) \subseteq \Phi^{n+1}_{P,\alpha}(\emptyset)$ by lemma 3.1 and these elements need not to be generated.
- It is possible that $\phi_{C_i,\alpha}(\Phi^n_{P,\alpha}(\emptyset))$ does not generate any elements that are not in $\Phi^n_{P,\alpha}(\emptyset)$. The time devoted to the application of such $\phi_{C_i,\alpha}$ is wasted.

The above discussion suggests that improvements can be made to algorithm 3.1. This paper addresses the following.

1. Because each $\phi_{C_i,\alpha}$ is monotonic, the elements in each $\phi_{C_i,\alpha}(\Phi^n_{P,\alpha}(\emptyset))$ can be added to the previous partial result. Thus only one partial result needs to be stored. We show that the algorithm thus obtained is correct.
2. It is not necessary to apply every $\phi_{C_i,\alpha}$ in every application of $\Phi_{P,\alpha}$; only the $\phi_{C_i,\alpha}$ corresponding to those clauses whose bodies contain subgoals which might unify with some recently generated elements should be applied.
3. Regeneration of the elements in the previous partial result should be avoided whenever possible.

3.2 Improvement in storage usage

We now present an improved algorithm that incorporates improvement 1 outlined in sub-section 3.1. To justify this improvement, we formulate a new function $\Psi_{P,\alpha} : \alpha(B_P) \mapsto \alpha(B_P)$ and prove that $lfp(\Psi_{P,\alpha}) = lfp(\Phi_{P,\alpha})$.

Definition 3.2 ($\psi_{C_i,\alpha} : \alpha(B_P) \mapsto \alpha(B_P)$)

$$\psi_{C_i,\alpha}(\mathcal{I}) = \mathcal{I} \cup \phi_{C_i,\alpha}(\mathcal{I}) \tag{6}$$

Each $\psi_{C_i,\alpha}$ is monotonic.

Observation 3.1 The following hold.

$$\forall \mathcal{I}.\psi_{C_i,\alpha}(\mathcal{I}) \supseteq \mathcal{I} \tag{7}$$

$$\forall n \geq 0.(\psi_{C_i,\alpha}(\Phi^n_{P,\alpha}(\emptyset)) \subseteq \Phi_{P,\alpha}(\Phi^n_{P,\alpha}(\emptyset))) \tag{8}$$

Definition 3.3 ($\Psi_{P,\alpha} : \alpha(B_P) \mapsto \alpha(B_P)$) Let $\circ$ denote function composition.

$$\Psi_{P,\alpha} = \psi_{C_N,\alpha} \circ \psi_{C_{N-1},\alpha} \circ \cdots \circ \psi_{C_1,\alpha} \tag{9}$$

$\Psi_{P,\alpha}$ is monotonic since each $\psi_{C_i,\alpha}$ is monotonic. Before we use $\Psi_{P,\alpha}$ to compute $lfp(\Phi_{P,\alpha})$, we need to establish that $lfp(\Phi_{P,\alpha}) = lfp(\Psi_{P,\alpha})$

Lemma 3.2 For any $n \geq 0$, $\Phi^n_{P,\alpha}(\emptyset) \subseteq \Psi^n_{P,\alpha}(\emptyset)$.

Proof The proof is by induction on n. First, the result holds for $n = 0$ since $\Phi^0_{P,\alpha}(\emptyset) = \Psi^0_{P,\alpha}(\emptyset) = \emptyset$.

Now suppose that the result holds for $n-1$. Let $\mathcal{A} \in \Phi^n_{P,\alpha}(\emptyset)$. Then there exist some σ and some clause C_i such that $\mathcal{A} \in \alpha([A_{i,0}\sigma])$ and $\alpha(A_{i,j}\sigma) \in \Phi^{n-1}_{P,\alpha}(\emptyset)$ for $1 \leq j \leq m_i$. By the induction hypothesis, $\alpha(A_{i,j}\sigma) \in \Psi^{n-1}_{P,\alpha}(\emptyset)$ for $1 \leq j \leq m_i$. By equation 6, $\mathcal{A} \in \Psi^n_{C_i,\alpha}(\emptyset)$. This, together with the monotonicity of $\Psi_{P,\alpha}$ and equation 9, implies $\mathcal{A} \in \Psi^n_{P,\alpha}(\emptyset)$. This completes the proof of the lemma. □

Lemma 3.3 For any $n \geq 0$, letting N be the number of clauses, there is some $0 \leq m \leq n \times N$ such that $\Psi^n_{P,\alpha}(\emptyset) \subseteq \Phi^m_{P,\alpha}(\emptyset)$.

Proof The proof is by induction on n. The result holds for $n = 0$ since $\Psi^0_{P,\alpha}(\emptyset) = \emptyset$ with $m = 0$.

Suppose that $\Psi^{n-1}_{P,\alpha}(\emptyset) \subseteq \Phi^{m_1}_{P,\alpha}(\emptyset)$ and $m_1 \leq N \times (n-1)$. If suffices to show that, for any $\mathcal{A} \in \Psi^n_{P,\alpha}(\emptyset)$, $\mathcal{A} \in \Phi^m_{P,\alpha}(\emptyset)$ for some $0 \leq m \leq n \times N$. If $\mathcal{A} \in \Psi^{n-1}_{P,\alpha}(\emptyset)$, then by the induction hypothesis, $\mathcal{A} \in \Phi^{m_1}_{P,\alpha}(\emptyset)$. If $\mathcal{A} \notin \Psi^{n-1}_{P,\alpha}(\emptyset)$, then there is some clause C_i such that $\mathcal{A} \in \psi_{C_i,\alpha} \circ \cdots \circ \psi_{C_1,\alpha}(\Psi^{n-1}_{P,\alpha}(\emptyset))$. By the induction hypothesis and the monotonicity of $\psi_{C_j,\alpha}$ for $0 < j \leq i$, we have $\mathcal{A} \in \psi_{C_i,\alpha} \circ \cdots \circ \psi_{C_1,\alpha}(\Phi^{m_1}_{P,\alpha}(\emptyset))$. So, by equation 8, $\mathcal{A} \in \Phi^{m_1+i}_{P,\alpha}(\emptyset)$. Letting $m = m_1 + i$, we have $\mathcal{A} \in \Psi^n_{P,\alpha}(\emptyset) \Rightarrow \mathcal{A} \in \Phi^m_{P,\alpha}(\emptyset)$ and $0 \leq m \leq n \times N$ since $0 \leq i \leq n$ and $0 \leq m_1 \leq (n-1) \times N$. □

Theorem 3.1 $lfp(\Psi_{P,\alpha}) = lfp(\Phi_{P,\alpha})$.

Proof By lemma 3.2 we have $lfp(\Phi_{P,\alpha}) \subseteq lfp(\Psi_{P,\alpha})$. By lemma 3.3 we have $lfp(\Psi_{P,\alpha}) \subseteq lfp(\Phi_{P,\alpha})$. So, $lfp(\Psi_{P,\alpha}) = lfp(\Phi_{P,\alpha})$. □

Using theorem 3.1, we can compute $lfp(\Psi_{P,\alpha})$ when $lfp(\Phi_{P,\alpha})$ is needed. Doing this results in algorithm 3.2 as an alternative for the computation of $lfp(\Phi_{P,\alpha})$.

Algorithm 3.2 An algorithm for the computation of $lfp(\Psi_{P,\alpha})$

$\mathcal{J} \leftarrow \emptyset$, $i \leftarrow 1$
repeat
  $\mathcal{J} \leftarrow \psi_{C_i,\alpha}(\mathcal{J})$
  $i \leftarrow i + 1$, **if** $i > N$ **then** $i \leftarrow 1$
until the last application of each $\psi_{C_i,\alpha}$ does not add any new elements.
return $\mathcal{J}$

Algorithm 3.2 needs only one partial result to be retained. $\psi_{C_i,\alpha}(\mathcal{J})$ adds to $\mathcal{J}$ those elements which can be derived from C_i and $\mathcal{J}$ in one step. The variable i is used to ensure that each $\psi_{C_i,\alpha}$ gets its turn to be applied. The correctness of algorithm 3.2 results from the fact that when no $\psi_{C_i,\alpha}$ generates any new elements, the least fixpoint is reached.

3.3 Improvements in time usage

In fact, the order in which $\psi_{C_i,\alpha}$ is applied has nothing to do with the correctness as long as each $\psi_{C_i,\alpha}$ gets its turn to be applied to the final result. It is possible that some applications of $\psi_{C_i,\alpha}$ in algorithm 3.2 do not contribute to $lfp(\Psi_{P,\alpha})$ and time devoted to such application is wasted. So, it is desirable to avoid such applications whenever possible. Whilst it is impossible to predict whether an application will contribute to $lfp(\Psi_{P,\alpha})$ without actually applying it, it is possible to predict whether an application is *likely* to contribute to $lfp(\Psi_{P,\alpha})$. These observations are utilised to effect improvements 2 and 3 outlined in sub-section 3.1 which result in algorithm 3.3.

Before we present algorithm 3.3, we give the following definition to simplify the presentation.

Definition 3.4 Let the *name/arity* of $A_{i,j}$ in $C_i = A_{i,0} \leftarrow A_{i,1} \wedge \cdots \wedge A_{i,m_i}$ be $p_{i,j}/n_{i,j}$. We define $C_r \prec C_s$ if $p_{r,0}/n_{r,0} = p_{s,k}/n_{s,k}$ for some $1 \leq k \leq m_s$. C_r is said to activate C_s.

The reading of $C_r \prec C_s$ is that the elements generated by $\psi_{C_r,\alpha}$ might be used by $\psi_{C_s,\alpha}$ to generate elements that are not already in the partial result. We now present the improved algorithm.

Algorithm 3.3 An improved algorithm for the computation of $lfp(\Psi_{P,\alpha})$ by intelligent selection of the order in which $\psi_{C_i,\alpha}$ is applied.

$\mathcal{J} \leftarrow \emptyset$, $F \leftarrow \{C_1, \ldots, C_N\}$
repeat
  $C \leftarrow SELECT(F)$
  $F \leftarrow F - \{C\}$
  $\mathcal{J} \leftarrow \psi_{C,\alpha}(\mathcal{J})$
  if $\psi_{C,\alpha}$ generates some new elements
  then $F \leftarrow F \cup \{C_j | C \prec C_j\}$
until $F = \emptyset$
return $\mathcal{J}$

There are two major differences between algorithm 3.2 and algorithm 3.3. First, the former applies $\psi_{C_i,\alpha}$ in a fixed order whilst the latter decides the order by means of a selection function. Second, the former applies all the functions $\psi_{C_1,\alpha}, \ldots, \psi_{C_N,\alpha}$ repeatedly whilst the latter applies only those which are *likely* to generate some new elements.

Algorithm 3.3 keeps an agenda F that contains all the clauses whose corresponding functions might generate some new elements. A selection function $SELECT$ selects a clause C from the agenda and this clause is then removed from the agenda. The algorithm applies $\psi_{C,\alpha}$ to the partial result. If and only if the application of $\psi_{C,\alpha}$ contributes some new elements to $lfp(\Psi_{P,\alpha})$, then each clause C_j satisfying $C \prec C_j$ is added into the agenda F because it is necessary to check if the application of $\psi_{C_j,\alpha}$ to the latest partial result generates any new element.

Whilst algorithm 3.3 avoids applying any $\psi_{C,\alpha}$ that does not generate new elements, the speed of the algorithm depends on the intelligence of $SELECT$, the function that selects clauses. It is desirable to select a clause C that generates more new elements than any other clause in F because of the monotonicity of $\psi_{C_i,\alpha}$.

We now formulate a selection function $SELECT$ that uses two heuristics to select clauses. For each clause C in F, letting m ($m \neq 0$) be the number of goals in the body of C, we add the following information.

- $n' = n/m$, where n is the number of clauses that have activated C since the last time that C was added into F. Intuitively, the more subgoals in a clause that are likely to be resatisfied, the more new elements is the clause likely to generate.

- $t' = t/m$, where t is the total number of new elements generated by the clauses that have activated C since the last time that C was added into F. It is also intuitively justifiable that the more likely each subgoal in a clause is to be resatisfied, the more new elements is the clause likely to generate.

- m, where m has been added so as to make the updating of the agenda easier.

The agenda thus augmented is denoted as $\mathcal{F}$ and each of its elements is a quadruple (n', t', m, C). The reason we use $n' = n/m$ and $t' = t/m$ instead of n and t is that intuitively, a clause with more conditions requires more resources to draw a new conclusion.

For a clause representing a fact ($m = 0$), n' and t' are given initial values to ensure that such a clause is considered before any clause representing an implication. This is accomplished by initialising $\mathcal{F}$ with $\mathcal{F}_0$ that has one entry $(n'_0, 0, m, C)$ for each clause C, where $n'_0 = 2$ when $m = 0$ or $n'_0 = 1/m$ when $m \neq 0$.

Definition 3.5 we define the operator $\uplus$ as follows.

$$\mathcal{F}_1 \uplus \mathcal{F}_2 = \left\{ (n', t', m, C) \;\middle|\; \begin{array}{l} ((C \notin F_1 \cap F_2) \wedge ((n', t', m, C) \in \mathcal{F}_1 \cup \mathcal{F}_2)) \vee \\ (C \in F_1 \cap F_2) \wedge (\exists (n'_1, t'_1, m, C) \in \mathcal{F}_1. \\ \exists (n'_2, t'_2, m, C) \in \mathcal{F}_2.(n' = n'_1 + n'_2, t' = t'_1 + t'_2)) \end{array} \right\}$$

Algorithm 3.4 This is a reformulation of algorithm 3.3 with heuristically based selection of the order in which $\psi_{C_i,\alpha}$ is applied.

$\mathcal{J} \leftarrow \emptyset$, $\mathcal{F} \leftarrow \mathcal{F}_0$
repeat
  Select clause C from $\mathcal{F}$ with the largest (n', t')
  $\mathcal{F} \leftarrow \mathcal{F} - \{(n', t', m, C)\}$
  $\mathcal{J} \leftarrow \psi_{C,\alpha}(\mathcal{J})$
  if $\psi_{C,\alpha}$ generates t_C new elements
  then $\mathcal{F} \leftarrow \mathcal{F} \biguplus \{(1/m_j, t_C/m_j, m_j, C_j) | C \prec C_j\}$
until $\mathcal{F} = \emptyset$
return $\mathcal{J}$

The selection can be accomplished by maintaining $\mathcal{F}$ as a sorted list. There are several small overheads introduced in algorithm 3.4 for deriving and storing the relation $\prec$, to remember how many new elements have been generated by each application of each $\psi_{C_i,\alpha}$ and to accomplish the operator $\biguplus$. The overheads are much smaller than the cost of the applications of those $\psi_{C_i,\alpha}$ that do not generate any new elements.

So far, we have developed algorithm 3.4 for computing $lfp(\Phi_{P,\alpha})$. The algorithm is suitable for any abstraction function as long as $\alpha(B_P)$ is finite and an ancillary algorithm for the computation of $\psi_{C,\alpha}$ is provided.

4 The implementation

We first discuss the representations of the data structures used in the algorithm. There are four data structures: the program P, the approximation of the program $lfp(\Phi_{P,\alpha})$, the relation $\prec$ to be derived from the program text and the agenda $\mathcal{F}$. $\mathcal{F}$ is represented as a sorted list of quadruples, each clause C_i is represented by $(-n_i/m_i, -t_i/m_i, m_i, i)$. Negative numbers have been used to represent heuristics to take advantage of the built-in predicate *sort*.

Each clause C_i of P is represented as ':-'$(A_{i,0}, (A_{i,1}, \ldots, A_{i,m_i}))$. A static syntactic analysis procedure *syn_pg* translates the clause into (i, m_i, ':-'$(A_{i,0}, (A_{i,1}, \ldots, A_{i,m_i}))$). *syn_pg* also calculates abstraction dependent information and the relation $\prec$, representing $C_i \prec C_j$ as $act(i, j)$. Procedure *copy* renames the variables in a clause, procedure *member* enumerates the elements in a list and procedure *delete* deletes an element from a list.

The implementation of algorithm 3.4 is presented as program 4.1 in Edinburgh Prolog [4]. The function of each procedure is described by the comments just preceding it. An abstraction dependent predicate *psi* is to be provided to use the program. The reading of $psi(+\text{Info}, +C, +\mathcal{I}, -\mathcal{J}, \text{-NN})$ is that applying $\psi_{C,\alpha}$ to $\mathcal{I}$ results in $\mathcal{J}$ with NN being the number of new elements generated.

Program 4.1 Prolog code that implements algorithm 3.4.

```
/* lfp(+P, -LFP) computes the finite approximation LFP of P.  */
lfp(P, LFP) :-
        syn_pg(P, Info, PgN),       /* analyses and transforms P*/
        init_agenda(PgN, F0),
```

```
        sort(F0, F),
        compute(PgN, Info, F, [], LFP).

/* compute(+Program, +Info, +Agenda, +ParResB, -ParResA)      */
/* repeatedly calls procedure psi with the clause specified by*/
/* the first job of the Agenda and modifies the Agenda and    */
/* maintains it as a  sorted list.                            */
compute(_Pg, _Info, [], ParRes, ParRes).    /* agenda is empty*/
compute(Pg, Info, [(_N,_T,_M,I)|Ag], ParResB, ParResA) :-
        member((I,_,CLAUSE), Pg),        /*   find the clause */
        copy(CLAUSE, CLAUSECOPY),        /* rename the clause */
        psi(Info, CLAUSECOPY, ParResB, ParResI, NN),
        (  NN =\= 0
           ->   jobs_activated(Pg, Info, I, NN, Jobs),
                update_agenda(Ag, Jobs, NewAg),
                sort(NewAg, AgN),
                compute(Pg, Info, AgN, ParResI, ParResA)
            ;   compute(Pg, Info, Ag, ParResI, ParResA)
        ).

/* jobs_activated(+Pg, +Info, +I, +NN, -Jobs) generates the   */
/* Jobs for the clauses activated by clause I.                */
jobs_activated(Pg, info(Heu, _Oth), I, NN, Jobs) :-
        setof(Job, find_one_job(Pg,Heu,I,NN,Job),Jobs1)
        ->  Jobs = Jobs1
        ;   Jobs = [].
find_one_job(Pg, Heu, I, NN, (NJ,TJ,MJ,J)) :-
        member(act(I,J), Heu),
        member((J,MJ,_Clause), Pg),
        NJ is -1/MJ,                /* MJ can't be 0  otherwise*/
        TJ is -NN/MJ.               /* J cannot be activated   */

/* update_agenda(+F1,+F2,-F3) applies the operator  in         */
/* definition 3.5 to F1 and F2 to result in F3.                */
update_agenda(F1,[],F1).
update_agenda(F1,[(N,T,M,I)|F2],[(N,T,M,I)|F3]):-
        \+ member((_,_,_,I),F1),
        update_agenda(F1,F2,F3).
update_agenda(F1,[(N,T,M,I)|F2],[(N1,T1,M,I)|F3]) :-
        member((N0,T0,M,I), F1),
        N1 is N + N0,
        T1 is T + T0,
        delete((N0,T0,M,I),F1,F10),
        update_agenda(F10,F2,F3).

/* init_agenda(+Program,-F0) sets up the initial agenda.       */
init_agenda([],[]).
init_agenda([(I,MI,_C)|Pg],[(NI,0,MI,I)|F0]) :-
        (  MI=\=0
           ->   NI is -1/MI
```

```
    ;    NI is -2        /* give priority to unit clauses*/
),
init_agenda(Pg,F0).
```

5 Two examples

In this section, we show how to use program 4.1 for the computations of finite approximations through an example program and depth abstractions and stump abstractions.

Definition 5.1 (depth k abstraction) 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 a newly created variable. Each such variable is denoted by an '_'. A term thus obtained is called a depth k abstract term.

Any depth k abstract term in $d_k(B_P)$, $d_k(U_P)$ or $d_k(M_P)$ does not have any variable at any level less than k. Such a term is called a canonical depth k abstract term.

Definition 5.2 (n-stump abstraction) Let t be a term and t' be a sub-term of t with f being its primary functor. t' has repetition depth (RD) 0 if the primary functor of any super-term of t' is different from f. t' has RD n if (a) t' has a super-term t'' with f as its primary functor and the RD of t'' is $n-1$ *and (b)* no super-term of t' with f as its primary functor has RD more than $n-1$.

The n-stump of a term results from replacing each argument of each of its sub-terms with RD n with a newly created variable. The function that maps a term to its n-stump is called an n-stump abstraction mapping, denoted as s_n.

We have implemented one abstraction-dependent predicate *psi* for any depth abstraction and another for any stump abstraction. Consider the following small program.

```
1    q(g(X,Y),g(Y,X)).
2    r(f(g(a,a))).
3    r(f(X)) :- r(X).
4    r(g(X,X)) :- r(X).
5    p(X):- q(X,Y),r(Y).
```

Figure 1 shows the difference between the performance of the heuristic algorithm 3.4 and that of the non-heuristic algorithm 3.2 for $lfp(\Phi_{P,d_3})$. It lists the clause number and the new elements generated for each application of $\psi_{C_i,\alpha}$ in the two algorithms. There are some empty lines for the heuristic algorithm because it avoids some non-productive applications. The heuristic algorithm invokes *three* non-productive calls to *psi* while the non-heuristic algorithm invokes *eight* such calls.

Figure 2 shows the difference between the performance of the heuristic algorithm 3.4 and that of the non-heuristic algorithm 3.2 for $lfp(\Phi_{P,s_1})$. The format of this figure is the same as figure 1. The heuristic algorithm invokes *two* non-productive calls to *psi* while the non-heuristic algorithm invokes *nine* such calls.

In both figure 1 and figure 2, q(g(X,Y),g(Y,X)) has been chosen to represent the set of abstract atoms that are abstractions of the ground instances of $q(g(X, Y), g(Y, X))$. For the depth abstraction d_3, q(g(X,Y), g(Y,X)) represents the following nine abstract atoms.

$$\left\{\begin{array}{c} q(g(a,a),g(a,a)),\ q(g(a,f(\_)),g(f(\_),a)),\ q(g(a,g(\_,\_)),g(g(\_,\_),a)), \\ q(g(f(\_),a),g(a,f(\_))),\ q(g(f(\_),f(\_)),g(f(\_),f(\_))), \\ q(g(f(\_),g(\_,\_)),g(g(\_,\_),f(\_))),\ q(g(g(\_,\_),a),g(a,g(\_,\_))), \\ q(g(g(\_,\_),f(\_)),g(f(\_),g(\_,\_))),\ q(g(g(\_,\_),g(\_,\_)),g(g(\_,\_),g(\_,\_))) \end{array}\right\}$$

For the stump abstraction s_1, (g(X,Y), g(Y,X)) represents the following 25 abstract atoms.

$$\left\{\begin{array}{c} q(g(a,f(a)),g(f(a),a)),\ q(g(f(f(\_)),f(g(\_,\_))),g(f(g(\_,\_)),f(f(\_)))), \\ q(g(a,a),g(a,a)),\ q(g(f(g(\_,\_)),f(g(\_,\_))),g(f(g(\_,\_)),f(g(\_,\_)))), \\ q(g(a,f(f(\_))),g(f(f(\_)),a)),\ q(g(f(g(\_,\_)),f(a)),g(f(a),f(g(\_,\_)))), \\ q(g(a,f(g(\_,\_))),g(f(g(\_,\_)),a)),\ q(g(f(f(\_)),f(a)),g(f(a),f(f(\_)))), \\ q(g(f(a),a),g(a,f(a))),\ q(g(g(\_,\_),f(g(\_,\_))),g(f(g(\_,\_)),g(\_,\_))), \\ q(g(f(a),f(a)),g(f(a),f(a))),\ q(g(g(\_,\_),g(\_,\_)),g(g(\_,\_),g(\_,\_))), \\ q(g(f(a),f(f(\_))),g(f(f(\_)),f(a))),\ q(g(f(a),g(\_,\_)),g(g(\_,\_),f(a))), \\ q(g(f(f(\_)),a),g(a,f(f(\_)))),\ q(g(f(a),f(g(\_,\_))),g(f(g(\_,\_)),f(a))), \\ q(g(a,g(\_,\_)),g(g(\_,\_),a)),\ q(g(f(f(\_)),f(f(\_))),g(f(f(\_)),f(f(\_)))), \\ q(g(f(g(\_,\_)),a),g(a,f(g(\_,\_)))),\ q(g(f(f(\_)),g(\_,\_)),g(g(\_,\_),f(f(\_)))), \\ q(g(f(g(\_,\_)),f(f(\_))),g(f(f(\_)),f(g(\_,\_)))), \\ q(g(f(g(\_,\_)),g(\_,\_)),g(g(\_,\_),f(g(\_,\_)))),\ q(g(g(\_,\_),a),g(a,g(\_,\_))), \\ q(g(g(\_,\_),f(a)),g(f(a),g(\_,\_))),\ q(g(g(\_,\_),f(f(\_))),g(f(f(\_)),g(\_,\_))), \end{array}\right\}$$

The heuristics used in algorithm 3.4 avoid some of the non-productive calls to *psi*. The number of such calls avoided depends on the program analysed and and the abstraction function. The time difference between the heuristic algorithm and the non-heuristic algorithm will be considerable for a larger program or a bigger $\alpha(B_P)$ because *psi* searches through the whole partial result m_i times whenever it is applied with clause C_i.

There exist some non-productive calls to *psi* in the heuristic algorithm because the heuristics used in the algorithm are conservative. More sophisticated heuristics could be used to further reduce non-productive calls to *psi*. For example, for the example program, the applications of ψ_{C_5} could be delayed until no new elements can be generated by $\psi_{C_1}, \ldots, \psi_{C_4}$. However, the task of gathering the information needed to use such heuristics would be much more complicated. So, it is uncertain if the use of such heuristics results in better performance. Furthermore, not all non-productive calls to *psi* can be avoided. There is always at least one non-productive call to *psi* before the least fixpoint is reached.

6 Conclusions

We have presented an abstract interpretation framework for a class of finite approximations of definite programs. Each finite approximation is characterised by an abstraction function and the finite approximation is given as the least fixpoint of a function over the abstract domain induced by the abstraction function. The development of a heuristic algorithm for the computation of

	heuristic		*non-heuristic*
clause	*elements*	*clause*	*elements*
1	q(g(X,Y),g(Y,X))	1	q(g(X,Y),g(Y,X))
2	r(f(g(_,_)))	2	r(f(g(_,_)))
3	r(f(f(_)))	3	r(f(f(_)))
4	r(g(f(_),f(_)))	4	r(g(f(_),f(_)))
5	p(g(f(_),f(_)))	5	p(g(f(_),f(_)))
3		1	
4	r(g(g(_,_),g(_,_)))	2	
3		3	
4		4	r(g(g(_,_),g(_,_)))
5	p(g(g(_,_),g(_,_)))	5	p(g(g(_,_),g(_,_)))
		1	
		2	
		3	
		4	
		5	

Figure 1: *The comparison of the performance of the heuristic algorithm 3.4 with that of the non-heuristic algorithm 3.2 for* $lfp(\Phi_{P,d_3})$

such finite approximations has been given in detail and the implementation of the algorithm in Prolog has been outlined.

The work presented is concerned with the computation of finite approximations by means of a bottom-up abstract interpretation framework. The problem of finite approximations has also been studied in other ways. For example, the problem of type inference is in essence a problem of finite approximations. There have been many type inference methods proposed for logic programs [7, 12, 1]. However, there has been no general framework in which to put them. Heintze and Jaffar [3] put some type inference methods in their Cartesian closure framework, whilst others [6, 2] put some type inference methods in various abstract interpretation frameworks. Our abstract interpretation framework has been generalised from depth abstractions and stump abstractions and both abstractions have been used in type inference systems [10, 12]. It is our speculation that the framework can also be used to explain some other type inference methods.

Peyton-Jones and Clack's [9] approach to the efficient calculation of fixpoints comprises an efficient representation and detection of fixpoints. Our approach to the problem is to use heuristics derived from the program to avoid non-productive computations. The two approaches can be combined to result in a more efficient solution.

O'Keefe [8] presents an algorithm for a more general class of fixpoint problems. When applied to find the least fixpoint of $\Phi_{P,\alpha}$, O'Keefe's algorithm corresponds to a variation of our algorithm 3.3 with a selection function that always selects the most recently activated clause from the agenda. This is accomplished by maintaining the agenda as a stack. The clause is selected by

heuristic		*non-heuristic*	
clause	*elements*	*clause*	*elements*
1	q(g(X,Y),g(Y,X))	1	q(g(X,Y),g(Y,X))
2	r(f(g(a,a)))	2	r(f(g(a,a)))
3	r(f(f(_)))	3	r(f(f(_)))
4	r(g(f(g(_,_)),f(g(_,_))) r(g(f(f(_)),f(f(_))))	4	r(g(f(g(_,_)),f(g(_,_)))) r(g(f(f(_)),f(f(_))))
5	p(g(f(g(_,_)),f(g(_,_)))) p(g(f(f(_)),f(f(_))))	5	p(g(f(g(_,_)),f(g(_,_)))) p(g(f(f(_)),f(f(_))))
3	r(f(g(f(_),f(_))))	1	
4	r(g(g(_,_),g(_,_)))	2	
3	r(f(g(g(_,_),g(_,_))))	3	r(f(g(f(_),f(_))))
4		4	r(g(g(_,_),g(_,_)))
5	p(g(g(_,_),g(_,_)))	5	p(g(g(_,_),g(_,_)))
3		1	
		2	
		3	r(f(g(g(_,_),g(_,_))))
		4	
		5	
		1	
		2	
		3	

Figure 2: *The comparison of the performance of the heuristic algorithm 3.4 with that of the non-heuristic algorithm 3.2 for* $lfp(\Phi_{P,s_1})$

popping it off the top of the stack. The newly activated clauses are pushed onto the stack. For each clause C in the agenda, our algorithm 3.4 evaluates the probability that the clause may produce new elements and selects that clause with the largest probability.

Acknowledgements

The authors are grateful to the anonymous referees for their many constructive criticisms and suggestions.

References

[1] H. Azzoune. Type inference in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the ninth International Conference on Automated Deduction*, pages 258–277, Argonne, Illinois, USA, May 23-26 1988. Springer-Verlag. Springer-Verlag Lecture Notes in Computer Science 310.

[2] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: towards the global optimisation of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204. The IEEE Society Press, 1987.

[3] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *The seventh Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 17-19 1990. The ACM Press.

[4] A.M.J. Hutching, D.L. Bowen, L. Byrd, P.W.H. Chung, F.C.N. Pereira, L.M. Pereira, R.Rae, and D.H.D. Warren. Edinburgh Prolog (the new implementation) user's manual. AI Applications Institute, University of Edinburgh, 8 October 1986.

[5] L. Lu and P. Greenfield. Abstract fixpoint semantics and abstract procedural semantics of definite logic programs. In *Proceedings of IEEE Computer Society 1992 International Conference on Computer Languages*, pages 147–154, Oakland, California, USA, April 20-23 1992. IEEE Computer Society Press.

[6] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 733–748. The MIT Press, 1988.

[7] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE international Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[8] R. A. O'Keefe. Finite fixed-point problems. In J.-L. Lassez, editor, *Proceedings of the fourth International Conference on Logic programming*, volume 2, pages 729–743. The MIT Press, 1987.

[9] S. Peyton-Jones and C. Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 246–265. Ellis Horwood Limited, 1987.

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

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

[12] J. Xu and D.S. Warren. A type inference system for Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 604–619. The MIT Press, 1988.