Underapproximation of Procedure Summaries for Integer Programs*

Pierre Ganty[1], Radu Iosif[2], and Filip Konečný[2,3]

[1] IMDEA Software Institute, Madrid, Spain
[2] VERIMAG/CNRS, Grenoble, France
[3] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. We show how to underapproximate the procedure summaries of recursive programs over the integers using off-the-shelf analyzers for non-recursive programs. The novelty of our approach is that the non-recursive program we compute may capture unboundedly many behaviors of the original recursive program for which stack usage cannot be bounded. Moreover, we identify a class of recursive programs on which our method terminates and returns the precise summary relations without underapproximation. Doing so, we generalize a similar result for non-recursive programs to the recursive case. Finally, we present experimental results of an implementation of our method applied on a number of examples.

1 Introduction

Procedure summaries are relations between the input and return values of a procedure, resulting from its terminating executions. Computing summaries is important, as they are a key enabler for the development of modular verification techniques for interprocedural programs, such as checking safety, termination or equivalence properties. Summary computation is, however, challenging in the presence of *recursive procedures* with integer parameters, return values, and local variables. While many analysis tools exist for non-recursive programs, only a few ones address the problem of recursion.

In this paper, we propose a novel technique to generate arbitrarily precise *underapproximations* of summary relations. Our technique is based on the following idea. The control flow of procedural programs is captured precisely by the language of a context-free grammar. A k-index underapproximation of this language (where $k \geqslant 1$) is obtained by filtering out those derivations of the grammar that exceed a budget, called *index*, on the number (at most k) of occurrences of non-terminals occurring at each derivation step. As expected, the higher the index, the more complete the coverage of the underapproximation. From there we define the k-index summary relations of a program by considering the k-index underapproximation of its control flow.

Our method then reduces the computation of k-index summary relations for a recursive program to the computation of summary relations for a non-recursive program, which is, in general, easier to compute because of the absence of recursion. The reduction was inspired by a decidability proof [4] in the context of Petri nets.

* Supported by the French National Project ANR-09-SEGI-016 VERIDYC, the Spanish Ministry of Economy and Finance grant (TIN2010-20639), and the Rich Model Toolkit initiative.

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 245–259, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The contributions of this paper are threefold. First, we show that, for a given index, recursive programs can be analyzed using off-the-shelf analyzers designed for non-recursive programs. Second, we identify a class of recursive programs, with possibly unbounded stack usage, on which our technique is complete i.e., it terminates and returns the precise result. Third, we present experimental results of an implementation of our method applied on a number of examples.

Related Work. programs handling integers (in general, unbounded data domains) has gained significant interest with the seminal work of Sharir and Pnueli [21]. They proposed two approaches for interprocedural dataflow analysis. The first one keeps precise values (*call strings*) up to a limited depth of the recursion stack, which bounds the number of executions. In contrast to the methods based on the call strings approach, our method can also analyse precisely certain programs for which the stack is unbounded, allowing for unbounded number of executions to be represented at once.

The second approach of Sharir and Pnueli is based on computing the least fixed point of a system of recursive dataflow equations (the *functional approach*). This approach to interprocedural analysis is based on computing an increasing *Kleene sequence* of abstract summaries. It is to be noticed that abstraction is key to ensuring termination of the Kleene sequence, the result being an over-approximation of the precise summary. Recently [10], the *Newton sequence* was shown to converge at least as fast as the Kleene sequence. The intuition behind the Newton sequence is to consider control paths in the program of increasing *index*. Our contribution can be seen as a technique to compute the iterates of the Newton sequence for programs with integer parameters, return values, and local variables, the result being, at each step, an under-approximation of the precise summary.

The complexity of the functional approach was shown to be polynomial in the size of the (finite) abstract domain, in the work of Reps, Horwitz and Sagiv [20]. This result is achieved by computing summary information, in order to reuse previously computed information during the analysis. Following up on this line of work, most existing abstract analyzers, such as INTERPROC [17], also use relational domains to compute *overapproximations* of function summaries – typically widening operators are used to ensure termination of fixed point computations. The main difference of our method with respect to static analyses is the use of underapproximation instead of overapproximation. If the final purpose of the analysis is program verification, our method will not return false positives. Moreover, the coverage can be increased by increasing the bound on the derivation index.

Previous works have applied model checking based on abstraction refinement to recursive programs. One such method, known as *nested interpolants* represents programs as nested word automata [3], which have the same expressive power as the visibly pushdown grammars used in our paper. Also based on interpolation is the WHALE algorithm [2], which combines partial exploration of the execution paths (underapproximation) with the overapproximation provided by a predicate-based abstract post operator, in order to compute summaries that are sufficient to prove a given safety property. Another technique, similar to WHALE, although not handling recursion, is the SMASH algorithm [13] which combines may- and must-summaries for compositional verification of safety properties. These approaches are, however, different in spirit from ours, as their goal is proving given safety properties of programs, as opposed to computing the summaries

of procedures independently of their calling context, which is our case. We argue that summary computation can be applied beyond safety checking, e.g., to prove termination [5], or program equivalence.

2 Preliminaries

Grammars. A *context-free grammar* (or simply grammar) is a tuple $G = (\mathcal{X}, \Sigma, \delta)$ where $\mathcal{X}$ is a finite nonempty set of *nonterminals*, Σ is a finite nonempty *alphabet* and $\delta \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ is a finite set of *productions*. The production (X, w) may also be noted $X \to w$. Also define $head(X \to w) = X$ and $tail(X \to w) = w$. Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we define a *step* $u \Longrightarrow v$ if there exists a production $(X, w) \in \delta$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\Longrightarrow^*$ to denote the reflexive transitive closure of $\Longrightarrow$. The *language* of G produced by a nonterminal $X \in \mathcal{X}$ is the set $L_X(G) = \{w \in \Sigma^* \mid X \Longrightarrow^* w\}$ and we call any sequence of steps from a nonterminal X to $w \in \Sigma^*$ a *derivation* from X. Given $X \Longrightarrow^* w$, we call the sequence $\gamma \in \delta^*$ of productions used in the derivation a *control word* and write $X \overset{\gamma}{\Longrightarrow} w$ to denote that the derivation conforms to γ.

Visibly Pushdown Grammars. To model the control flow of procedural programs we use languages generated by visibly pushdown grammars, a subset of context-free grammars. In this setting, words are defined over a *tagged alphabet* $\widehat{\Sigma} = \Sigma \cup \langle\Sigma \cup \Sigma\rangle$, where $\langle\Sigma = \{\langle a \mid a \in \Sigma\}$ represents procedure *call* sites and $\Sigma\rangle = \{a\rangle \mid a \in \Sigma\}$ represents procedure *return* sites. Formally, a *visibly pushdown grammar* $G = (\mathcal{X}, \widehat{\Sigma}, \delta)$ is a grammar that has only productions of the following forms, for some $a, b \in \Sigma$:

$$X \to a \qquad\qquad X \to a\,Y \qquad\qquad X \to \langle a\,Y\,b\rangle\,Z$$

It is worth pointing that, for our purposes, we do not need a visibly pushdown grammar to generate the empty string ε. Each tagged word generated by visibly pushdown grammars is associated a *nested word* [3] the definition of which we briefly recall. Given a finite alphabet Σ, a *nested word* over Σ is a pair $(w, \rightsquigarrow)$, where $w = a_1 a_2 \ldots a_n \in \Sigma^*$, and $\rightsquigarrow \subseteq \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ is a set of *nesting edges* (or simply edges) where:

1. $i \rightsquigarrow j$ only if $i < j$, i.e. edges only go forward;
2. $|\{j \mid i \rightsquigarrow j\}| \leqslant 1$ and $|\{i \mid i \rightsquigarrow j\}| \leqslant 1$, i.e. no two edges share a call/return position;
3. if $i \rightsquigarrow j$ and $k \rightsquigarrow \ell$ then it is not the case that $i < k \leqslant j < \ell$, i.e. edges do not cross.

Intuitively, we associate a nested word to a tagged word as follows: there is an edge between tagged symbols $\langle a$ and $b\rangle$ iff both are generated at the same derivation step. For instance looking forward at Ex. 2 consider the tagged word $w = \tau_1\tau_2\langle\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\rangle\tau_4$ resulting from a derivation $Q_1^{init} \Rightarrow^* w$. The nested word associated to w is $(\tau_1\tau_2\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\tau_4, \{3 \rightsquigarrow 8\})$. Finally, let w_nw denote the mapping which given a tagged word in the language of a visibly pushdown grammar returns the nested word thereof.

Integer Relations. We denote by $\mathbb{Z}$ the set of integers. Let $\mathbf{x} = \{x_1, x_2, \ldots, x_d\}$ be a set of variables for some $d > 0$. Define $\mathbf{x}'$ the *primed* variables of $\mathbf{x}$ to be $\{x'_1, x'_2, \ldots, x'_d\}$. All variables range over $\mathbb{Z}$. We denote by $\overrightarrow{\mathbf{y}}$ an ordered sequence $\langle y_1, \ldots, y_k\rangle$ of variables, and by $|\overrightarrow{\mathbf{y}}|$ its length k. By writing $\overrightarrow{\mathbf{y}} \subseteq \mathbf{x}$ we mean that each variable in $\overrightarrow{\mathbf{y}}$ belongs to $\mathbf{x}$. For sequences $\overrightarrow{\mathbf{y}}$ and $\overrightarrow{\mathbf{z}}$ of length k, $\overrightarrow{\mathbf{y}} = \overrightarrow{\mathbf{z}}$ stands for the equality $\bigwedge_{i=1}^{k} y_i = z_i$.

A *linear term* t is a linear combination of the form $a_0 + \sum_{i=1}^{d} a_i x_i$, where $a_0, \ldots, a_d \in \mathbb{Z}$. An *atomic proposition* is a predicate of the form $t \leqslant 0$, where t is a linear term. We consider formulae in the first-order logic over atomic propositions $t \leqslant 0$, also known as *Presburger arithmetic*. A *valuation* of $\mathbf{x}$ is a function $\nu : \mathbf{x} \to \mathbb{Z}$. The set of all valuations of $\mathbf{x}$ is denoted by $\mathbb{Z}^{\mathbf{x}}$. If $\overrightarrow{\mathbf{y}} = \langle y_1, \ldots, y_k \rangle$ is an ordered sequence of variables, we denote by $\nu(\overrightarrow{\mathbf{y}})$ the sequence of integers $\langle \nu(y_1), \ldots, \nu(y_k) \rangle$. An arithmetic formula $\mathcal{R}(\mathbf{x}, \mathbf{y}')$ defining a relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{y}}$ is evaluated with respect to two valuations $\nu_1 \in \mathbb{Z}^{\mathbf{x}}$ and $\nu_2 \in \mathbb{Z}^{\mathbf{y}}$, by replacing each $x \in \mathbf{x}$ by $\nu_1(x)$ and each $y' \in \mathbf{y}'$ by $\nu_2(y)$ in $\mathcal{R}$. The composition of two relations $R_1 \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{y}}$ and $R_2 \subseteq \mathbb{Z}^{\mathbf{y}} \times \mathbb{Z}^{\mathbf{z}}$ is denoted by $R_1 \circ R_2 = \{(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{z}} \mid \exists \mathbf{t} \in \mathbb{Z}^{\mathbf{y}} . (\mathbf{u}, \mathbf{t}) \in R_1 \text{ and } (\mathbf{t}, \mathbf{v}) \in R_2\}$. For a subset $\mathbf{y} \subseteq \mathbf{x}$, we denote $\nu\downarrow_{\mathbf{y}} \in \mathbb{Z}^{\mathbf{y}}$ the projection of ν onto variables $\mathbf{y} \subseteq \mathbf{x}$. Finally, given two valuations $I, O \in \mathbb{Z}^{\mathbf{x}}$, we denote by $I \cdot O$ their concatenation and we define $\mathbb{Z}^{\mathbf{x} \times \mathbf{x}} = \{I \cdot O \mid I, O \in \mathbb{Z}^{\mathbf{x}}\}$.

3 Integer Recursive Programs

We consider in the following that programs are collections of procedures calling each other, possibly according to recursive schemes. Formally, an *integer program* is an indexed tuple $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$, where $P_1, \ldots, P_n$ are *procedures*. Each procedure is a tuple $P_i = \langle \mathbf{x}_i, \overrightarrow{\mathbf{x}}_i^{in}, \overrightarrow{\mathbf{x}}_i^{out}, S_i, q_i^{init}, F_i, \Delta_i \rangle$, where $\mathbf{x}_i$ are the *local* variables[1] of P_i ($\mathbf{x}_i \cap \mathbf{x}_j = \emptyset$ for all $i \neq j$), $\overrightarrow{\mathbf{x}}_i^{in}, \overrightarrow{\mathbf{x}}_i^{out} \subseteq \mathbf{x}_i$ are the ordered tuples of input and output variables, S_i are the *control states* of P_i ($S_i \cap S_j = \emptyset$, for all $i \neq j$), $q_i^{init} \in S_i$ is the *initial*, and $F_i \subseteq S_i$ are the *final* states of P_i, and Δ_i is a set of *transitions* of one of the following forms:

- $q \xrightarrow{\mathcal{R}(\mathbf{x}_i, \mathbf{x}_i')} q'$ is an *internal transition*, where $q, q' \in S_i$, and $\mathcal{R}(\mathbf{x}_i, \mathbf{x}_i')$ is a Presburger arithmetic relation involving only the local variables of P_i;
- $q \xrightarrow{\overrightarrow{\mathbf{z}}' = P_j(\overrightarrow{\mathbf{u}})} q'$ is a *call*, where $q, q' \in S_i$, P_j is the callee, $\overrightarrow{\mathbf{u}}$ are linear terms over $\mathbf{x}_i$, $\overrightarrow{\mathbf{z}} \subseteq \mathbf{x}_i$ are variables, such that $|\overrightarrow{\mathbf{u}}| = |\overrightarrow{\mathbf{x}}_j^{in}|$ and $|\overrightarrow{\mathbf{z}}| = |\overrightarrow{\mathbf{x}}_j^{out}|$.

We define the *size of the program* $\sharp\mathcal{P} = \sum_{i=1}^{n} |\Delta_i|$ to be the total number of transition rules, and $loc(\mathcal{P}) = \sum_{i=1}^{n} |S_i|$ be the number of control locations in $\mathcal{P}$. The *call graph* of a program $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ is a directed graph with vertices $P_1, \ldots, P_n$ and an edge (P_i, P_j), for each P_i and P_j, such that P_i has a call to P_j. A program is said to be *recursive* if its call graph has at least one cycle, and *non-recursive* if its call graph is a dag. Finally, let $n\mathcal{F}(P_i)$ denotes the set $S_i \setminus F_i$ of non-final states of P_i, and $n\mathcal{F}(\mathcal{P}) = \bigcup_{i=1}^{n} n\mathcal{F}(\mathcal{P})$ be the set of non-final states of $\mathcal{P}$.

Simplified Syntax. To ease the description of programs defined in this paper, we use a simplified, human readable, imperative language such that each procedure of the program conforms to the following grammar:[2]

$$P ::= \textbf{proc } P_i(id^*) \textbf{begin var } id^* \ S \textbf{ end} \qquad S ::= S; S \mid \textbf{assume } f$$
$$S ::= id^n \leftarrow t^n \mid id \leftarrow P_i(t^*) \mid P_i(t^*) \mid \textbf{return } (id + \varepsilon) \mid \textbf{goto } \ell^+ \mid \textbf{havoc } id^+$$

[1] Observe that there are no global variables in the definition of integer program. Those can be encoded as input and output variables to each procedure.

[2] Our simplified syntax does not seek to capture the generality of integer programs. Instead, our goal is to give a convenient notation for the programs given in this paper and only those.

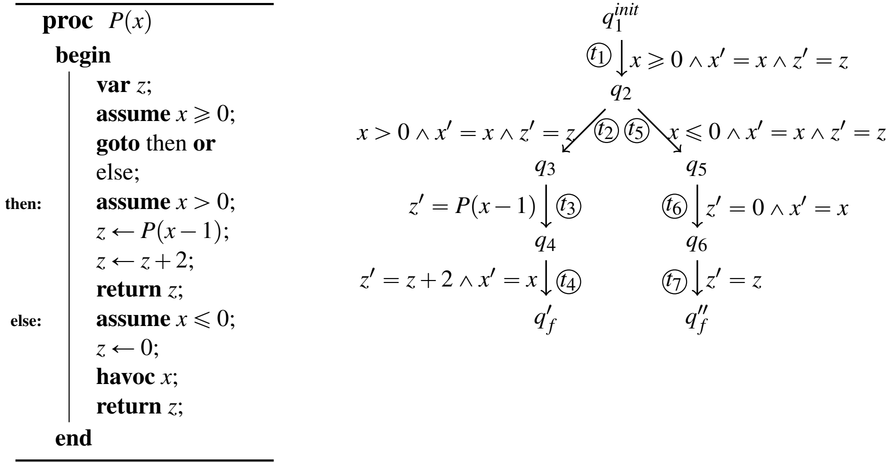

Fig. 1. Example of a simplified imperative program and its integer program thereof

The local variables occurring in P are denoted by id, linear terms by t, Presburger formulae by f, and control labels by ℓ. Each procedure consists in local declarations followed by a sequence of statements. Statements may carry a label. Program statements can be either assume statements[3], (parallel) assignments, procedure calls (possibly with a return value), return to the caller (possibly with a value), non-deterministic jumps **goto** ℓ_1 **or** $\ldots$ **or** ℓ_n, and **havoc** $x_1, x_2, \ldots, x_n$ statements[4]. We consider the usual syntactic requirements (used variables must be declared, jumps are well defined, no jumps outside procedures, etc.). We do not define them, it suffices to know that all simplified programs in this paper comply with the requirements. A program using the simplified syntax can be easily translated into the formal syntax, as shown at Fig. 1.

Example 1. Figure 1 shows a program in our simplified imperative language and its corresponding integer program $\mathcal{P}$. Formally, $\mathcal{P} = \langle P \rangle$ where P is defined as: $\langle \{x,z\}, \langle x \rangle, \langle z \rangle, \{q_1^{init}, q_2, q_3, q_4, q_5, q_6, q'_f, q''_f\}, q_1^{init}, \{q'_f, q''_f\}, \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\} \rangle$. Since P calls itself (t_3), this program is recursive. ■

Semantics. We are interested in computing the *summary relation* between the values of the input and output variables of a procedure. To this end, we give the semantics of a program $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ as a tuple of relations R_q describing, for each non-final control state $q \in n\mathcal{F}(P_i)$, the effect of the program when started in q upon reaching a state in F_i. An *interprocedurally valid path* is represented by a tagged word over an alphabet $\widehat{\Theta}$, which maps each internal transition t to a symbol τ, and each call transition t to a pair of symbols $\langle \tau, \tau \rangle \in \widehat{\Theta}$. In the sequel, we denote by Q the nonterminal corresponding to the control state q, and by $\tau \in \Theta$ the alphabet symbol corresponding to the transition t of $\mathcal{P}$. Formally, we associate $\mathcal{P}$ a visibly pushdown grammar, denoted in the rest of the paper by $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$, such that $Q \in \mathcal{X}$ if and only if $q \in n\mathcal{F}(\mathcal{P})$ and:

[3] **assume** f is executable if and only if the current values of the variables satisfy f.

[4] **havoc** assigns non deterministically chosen integers to $x_1, x_2, \ldots, x_n$.

(a) $Q \to \tau \in \delta$ if and only if $t\colon q \xrightarrow{\mathcal{R}} q'$ and $q' \notin n\mathcal{F}(\mathcal{P})$

(b) $Q \to \tau\ Q' \in \delta$ if and only if $t\colon q \xrightarrow{\mathcal{R}} q'$ and $q' \in n\mathcal{F}(\mathcal{P})$

(c) $Q \to \langle \tau\ Q_j^{init}\ \tau \rangle\ Q' \in \delta$ if and only if $t\colon q \xrightarrow{\overrightarrow{\mathbf{z}}' = P_j(\overrightarrow{\mathbf{u}})} q'$

It is easily seen that interprocedurally valid paths in $\mathcal{P}$ and tagged words in $G_{\mathcal{P}}$ are in one-to-one correspondence. In fact, each interprocedurally valid path of $\mathcal{P}$ between state $q \in n\mathcal{F}(P_i)$ and a state of F_i, where $1 \leqslant i \leqslant n$, corresponds exactly to one tagged word of $L_Q(G_{\mathcal{P}})$.

Example 2. (continued from Ex. 1) The visibly pushdown grammar $G_{\mathcal{P}}$ corresponding to $\mathcal{P}$ consists of the following variables and labelled productions:

$$
\begin{array}{ll}
p_1^b \stackrel{def}{=} Q_1^{init} \to \tau_1\ Q_2 & p_3^c \stackrel{def}{=} Q_3 \to \langle \tau_3\ Q_1^{init}\ \tau_3 \rangle\ Q_4 \\
p_2^b \stackrel{def}{=} Q_2 \quad \to \tau_2\ Q_3 & p_4^a \stackrel{def}{=} Q_4 \to \tau_4 \\
p_5^b \stackrel{def}{=} Q_2 \quad \to \tau_5\ Q_5 & p_6^b \stackrel{def}{=} Q_5 \to \tau_6\ Q_6 \\
 & p_7^a \stackrel{def}{=} Q_6 \to \tau_7
\end{array}
$$

In the following, we use superscripts a,b,c to distinguish productions of the form $Q \to \tau$, $Q \to \tau Q'$ or $Q \to \langle \tau Q_j^{init} \tau \rangle Q'$, respectively. $L_{Q_1^{init}}(G_{\mathcal{P}})$ includes the word $w = \tau_1\tau_2\langle\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\rangle\tau_4$, of which $w_nw(w) = (\tau_1\tau_2\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\tau_4, \{3 \rightsquigarrow 8\})$ is the corresponding nested word. The word w corresponds to an interprocedurally valid path where P calls itself once. Let $\gamma_1 = p_1^b p_2^b p_3^c p_4^a p_1^b p_5^b p_6^b p_7^a$ and $\gamma_2 = p_1^b p_2^b p_3^c p_1^b p_5^b p_6^b p_7^a p_4^a$ be two control words such that $Q_1^{init} \stackrel{\gamma_1}{\Longrightarrow} w$ and $Q_1^{init} \stackrel{\gamma_2}{\Longrightarrow} w$. ■

The semantics of a program is the union of the semantics of the nested words corresponding to its executions, each of which being a relation over input and output variables. To define the semantics of a nested word, we first associate to each $\tau \in \widehat{\Theta}$ an integer relation ρ_τ, defined as follows:

- for an internal transition $t\colon q \xrightarrow{\mathcal{R}} q' \in \Delta_i$, let $\rho_\tau \equiv \mathcal{R}(\mathbf{x}_i, \mathbf{x}_i') \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$;
- for a call transition $t : q \xrightarrow{\overrightarrow{\mathbf{z}}' = P_j(\overrightarrow{\mathbf{u}})} q' \in \Delta_i$, we define a *call relation* $\rho_{\langle\tau} \equiv (\overrightarrow{\mathbf{x}}_j^{in\prime} = \overrightarrow{\mathbf{u}}) \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_j}$, a *return relation* $\rho_{\tau\rangle} \equiv (\overrightarrow{\mathbf{z}}' = \overrightarrow{\mathbf{x}}_j^{out}) \subseteq \mathbb{Z}^{\mathbf{x}_j} \times \mathbb{Z}^{\mathbf{x}_i}$ and a *frame relation* $\phi_\tau \equiv \bigwedge_{x \in \mathbf{x}_i \setminus \overrightarrow{\mathbf{z}}} x' = x \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$.

We define the semantics of the program $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ in a top-down manner. Assuming a fixed ordering of the non-final states in the program, i.e. $n\mathcal{F}(\mathcal{P}) = \langle q_1, \ldots, q_m \rangle$, the semantics of the program $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]$, is the tuple of relations $\langle [\![q_1]\!], \ldots, [\![q_m]\!] \rangle$. For each non-final control state $q \in n\mathcal{F}(P_i)$ where $1 \leqslant i \leqslant n$, we denote by $[\![q]\!] \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$ the relation (over the local variables of procedure P_i) defined as $[\![q]\!] = \bigcup_{\alpha \in L_Q(G_{\mathcal{P}})} [\![\alpha]\!]$.

It remains to define $[\![\alpha]\!]$, the semantics of the tagged word α. Out of convenience, we define the semantics of its corresponding nested word $w_nw(\alpha) = (\tau_1 \ldots \tau_\ell, \rightsquigarrow)$ over alphabet Θ, and define $[\![\alpha]\!] = [\![w_nw(\alpha)]\!]$. For a nesting relation $\rightsquigarrow \subseteq \{1, \ldots, \ell\} \times \{1, \ldots, \ell\}$, we define $\rightsquigarrow_{i,j} = \{(s-(i-1), t-(i-1)) \mid (s,t) \in \rightsquigarrow \cap \{i, \ldots, j\} \times \{i, \ldots, j\}\}$, for some $i, j \in \{1, \ldots, \ell\}$, $i < j$. Finally, we define $[\![(\tau_1 \ldots \tau_\ell, \rightsquigarrow)]\!] \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$ (recall that $\alpha \in L_Q(G_{\mathcal{P}})$ and q is a state of P_i) as follows:

$$[\![(\tau_1 \dots \tau_\ell, \rightsquigarrow)]\!] = \begin{cases} \rho_{\tau_1} & \text{if } \ell = 1 \\ \rho_{\tau_1} \circ [\![(\tau_2 \dots \tau_\ell, \rightsquigarrow_{2,\ell})]\!] & \text{if } \ell > 1 \text{ and } \forall 1 \leqslant j \leqslant \ell \colon 1 \not\rightsquigarrow j \\ CaRet_\tau \circ [\![(\tau_{j+1} \dots \tau_\ell, \rightsquigarrow_{j+1,\ell})]\!] & \text{if } \ell > 1 \text{ and } \exists 1 \leqslant j \leqslant \ell \colon 1 \rightsquigarrow j \end{cases}$$

where, in the last case, which corresponds to call transition $t : q \xrightarrow{\overrightarrow{\mathbf{z}}' = P_d(\overrightarrow{\mathbf{u}})} q' \in \Delta_i$, we have $\tau_1 = \tau_j = \tau$ and define $CaRet_\tau = \left(\rho_{\langle\tau} \circ [\![\tau_2 \dots \tau_{j-1}, \rightsquigarrow_{2,j-1})]\!] \circ \rho_{\tau\rangle}\right) \cap \phi_\tau$.

Example 3. (continued from Ex. 2) The semantics of a given the nested word $\theta = (\tau_1\tau_2\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\tau_4, \{3 \rightsquigarrow 8\})$ is a relation between valuations of $\{x, z\}$, given by:

$$[\![\theta]\!] = \rho_{\tau_1} \circ \rho_{\tau_2} \circ \left((\rho_{\langle\tau_3} \circ \rho_{\tau_1} \circ \rho_{\tau_5} \circ \rho_{\tau_6} \circ \rho_{\tau_7} \circ \rho_{\tau_3\rangle}) \cap \phi_{t_3}\right) \circ \rho_{\tau_4}$$

One can verify that $[\![\theta]\!] \equiv x = 1 \wedge z' = 2$, i.e. the result of calling P with an input valuation $x = 1$ is the output valuation $z = 2$. ■

Finally, we introduce a few useful notations. By $[\![\mathcal{P}]\!]_q$ we denote the component of $[\![\mathcal{P}]\!]$ corresponding to $q \in n\mathcal{F}(\mathcal{P})$. Slightly abusing notations, we define $L_{P_i}(G_\mathcal{P})$ as $L_{Q_i^{init}}(G_\mathcal{P})$ and $[\![\mathcal{P}]\!]_{P_i}$ as $[\![\mathcal{P}]\!]_{q_i^{init}}$. Finally, define $[\![\mathcal{P}]\!]_{P_i}^{i/o} = \{\langle I\downarrow_{x_i^{in}}, O\downarrow_{x_i^{out}}\rangle \mid \langle I, O\rangle \in [\![\mathcal{P}]\!]_{P_i}\}$.

4 Underapproximating the Program Semantics

In this section we define a family of underapproximations of $[\![\mathcal{P}]\!]$, called *bounded-index underapproximations*. Then we show that each k-index underapproximation of the semantics of a (possibly recursive) program $\mathcal{P}$ coincides with the semantics of a non-recursive program computable from $\mathcal{P}$ and k. The central notion of bounded-index derivation is introduced in the following followed by basic properties about them.

Definition 1. *Given a grammar $G = (\mathcal{X}, \Sigma, \delta)$ with relation $\Longrightarrow$ between strings, for every $k \geqslant 1$ we define the relation $\overset{(k)}{\Longrightarrow} \subseteq \Longrightarrow$ as follows: $u \overset{(k)}{\Longrightarrow} v$ iff $u \Longrightarrow v$ and both u and v contain at most k occurrences of variables from $\mathcal{X}$. We denote by $\overset{(k)}{\Longrightarrow}^\star$ the reflexive transitive closure of $\overset{(k)}{\Longrightarrow}$. Hence given X and k define $L_X^{(k)}(G) = \{w \in \Sigma^* \mid X \overset{(k)}{\Longrightarrow}^\star w\}$, and we call the $\overset{(k)}{\Longrightarrow}$-derivation of $w \in \Sigma^*$ from X a k-*index derivation. *A grammar G is said to have* index k *whenever $L_X(G) = L_X^{(k)}(G)$ for each $X \in \mathcal{X}$.*[5]

Lemma 1. *For every grammar the following properties hold: (1) $\overset{(k)}{\Longrightarrow} \subseteq \overset{(k+1)}{\Longrightarrow}$ for all $k \geqslant 1$; (2) $\Longrightarrow = \bigcup_{k=1}^{\infty} \overset{(k)}{\Longrightarrow}$; (3) $BC \overset{(k)}{\Longrightarrow}^\star w \in \Sigma^*$ iff there exist w_1, w_2 such that $w = w_1 w_2$ and either (i) $B \overset{(k-1)}{\Longrightarrow}^\star w_1$, $C \overset{(k)}{\Longrightarrow}^\star w_2$, or (ii) $C \overset{(k-1)}{\Longrightarrow}^\star w_2$ and $B \overset{(k)}{\Longrightarrow}^\star w_1$.*

The main intuition behind our method is to filter out interprocedurally valid paths which can not be produced by k-index derivations. Our analysis is then carried out on the remaining paths produced by k-index derivations only.

[5] Gruska [15] proved that deciding whether $L_X(G) = L_X^{(k)}(G)$ for some $k \geqslant 1$ is undecidable.

Example 4. (continued form Ex. 2) P is a (non-tail) recursive procedure and $G_{\mathcal{P}}$ models its control flow. Inspecting $G_{\mathcal{P}}$ reveals that $L_{Q_1^{init}}(G_{\mathcal{P}}) = \{(\tau_1\tau_2\langle\tau_3)^n\tau_1\tau_5\tau_6\tau_7(\tau_3\rangle\tau_4)^n \mid n \geqslant 0\}$. For each value of n we give a 2-index derivation capturing the word: repeat n times the steps $Q_1^{init} \overset{p_1^b p_2^b p_3^c}{\Longrightarrow} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle Q_4 \overset{p_4^a}{\Longrightarrow} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle\tau_4$ followed by the steps $Q_1^{init} \overset{p_1^b p_5^b p_6^b p_7^a}{\Longrightarrow} \tau_1\,\tau_5\,\tau_6\,\tau_7$. Therefore the 2-index approximation of $G_{\mathcal{P}}$ shows that $L_{Q_1^{init}}(G_{\mathcal{P}}) = L^{(2)}_{Q_1^{init}}(G_{\mathcal{P}})$. However bounding the number of times P calls itself up to 2 results in 3 interprocedurally valid paths (for $n = 0,1,2$). $\square$

Given $k \geqslant 1$, we define the *k-index semantics* of $\mathcal{P}$ as $[\![\mathcal{P}]\!]^{(k)} = \langle [\![q_1]\!]^{(k)}, \ldots, [\![q_m]\!]^{(k)} \rangle$, where the k-index semantics of a non-final control state q of a procedure P_i is the relation $[\![q]\!]^{(k)} \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$, defined as $[\![q]\!] = \bigcup_{\alpha \in L_Q^{(k)}(G_{\mathcal{P}})} [\![\alpha]\!]$.

4.1 Computing Bounded-Index Underapproximations

In what follows, we define a source-to-source transformation that takes in input a recursive program $\mathcal{P}$, an integer $k \geqslant 1$ and returns a *non-recursive* program $\mathcal{H}^k$ which has the same semantics as $[\![\mathcal{P}]\!]^{(k)}$ (modulo projection on some variables). Therefore every off-the-shelf tool, that computes the summary semantics for a non-recursive program, can be used to compute the k-index semantics of $\mathcal{P}$, for any given $k \geqslant 1$.

Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a program, and $\mathbf{x} = \bigcup_{i=1}^n \mathbf{x}_i$ be the set of all variables in $\mathcal{P}$. As we did previously, we assume a fixed ordering $\langle q_1, \ldots, q_m \rangle$ on the set $n\mathcal{F}(\mathcal{P})$. Let $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$ be the visibly pushdown grammar associated with $\mathcal{P}$, such that each non-final state q of $\mathcal{P}$ is associated a nonterminal $Q \in \mathcal{X}$. Then we define a *non-recursive* program $\mathcal{H}^K$ that captures the K-index semantics of $\mathcal{P}$ (Algorithm 1), for $K \geqslant 1$. Formally, we define $\mathcal{H}^K = \times_{k=0}^{K} \langle query^k_{Q_1}, \ldots, query^k_{Q_m} \rangle$, where:

- for each $k = 0, \ldots, K$ and each control state $q \in n\mathcal{F}(\mathcal{P})$, we have a procedure $query^k_Q$;
- in particular, $query^0_{Q_1}, \ldots, query^0_{Q_m}$ consists of one **assume** *false* statement;
- each procedure $query^k_Q$ has five sets of local variables, all of the same cardinality as $\mathbf{x}$: two sets, named $\mathbf{x}_I$ and $\mathbf{x}_O$, are used as input variables, whereas the other three sets, named $\mathbf{x}_J, \mathbf{x}_K$ and $\mathbf{x}_L$ are used locally by $query^k_Q$. Besides, $query^k_Q$ has a local variable called PC. There are no output variables.

Observe that each procedure $query^k_Q$ calls only procedures $query^{k-1}_{Q'}$ for some Q', hence the program $\mathcal{H}^K$ is non-recursive, and therefore amenable to summarization techniques that cannot handle recursion. Also the hierarchical structure of $\mathcal{H}^K$ enables modular summarization by computing the summaries ordered by increasing values of $k = 0, 1, \ldots, K$. The summaries of $\mathcal{H}^{K-1}$ are reused to compute $\mathcal{H}^K$. Finally, it is routine to check that the size of $\mathcal{H}^K$ is in $O(K \cdot \sharp\mathcal{P})$. Furthermore, the time needed to generate $\mathcal{H}^K$ is linear in the product $K \cdot \sharp\mathcal{P}$.

Given that $query^k_Q$ has two copies of $\mathbf{x}$ as input variables, and no output variables, the input output semantics $[\![\mathcal{H}]\!]^{i/o}_{query^k_Q} \subseteq \mathbb{Z}^{\mathbf{x} \times \mathbf{x}}$ is a set of tuples, rather than a (binary) relation. We denote $pre(query^k_Q) = \{I \cdot O \in \mathbb{Z}^{\mathbf{x} \times \mathbf{x}} \mid query^k_Q(I, O) \text{ returns with empty stack}\}$. Clearly $pre(query^k_Q) = [\![\mathcal{H}]\!]^{i/o}_{query^k_Q}$.

Algorithm 1. proc $query^k_Q(\mathbf{x}_I, \mathbf{x}_O)$ for $k \geqslant 1$

```
begin
    var PC, x_J, x_K, x_L;
    PC ← Q;
start:  goto p^a_1 or ··· or p^a_{n_a} or p^b_1 or ··· or p^b_{n_b} or p^c_1 or ··· or p^c_{n_c} ;
p^a_1:  assume (PC = head(p^a_1)); assume ρ_{tail(p^a_1)}(x_I, x_O); return;
        ⋮
p^a_{n_a}: assume (PC = head(p^a_{n_a})); assume ρ_{tail(p^a_{n_a})}(x_I, x_O); return;
p^b_1:  assume (PC = head(p^b_1)); [ paste code for 2nd case: tail(p^b_1) ∈ Θ × X ];
        ⋮
p^b_{n_b}: assume (PC = head(p^b_{n_b})); [ paste code for 2nd case: tail(p^b_{n_b}) ∈ Θ × X ];
p^c_1:  assume (PC = head(p^c_1)); [ paste code for 3rd case: tail(p^c_1) ∈ ⟨Θ × X × Θ⟩ × X ];
        ⋮
p^c_{n_c}: assume (PC = head(p^c_{n_c})); [ paste code for 3rd case: tail(p^c_{n_c}) ∈ ⟨Θ × X × Θ⟩ × X ];
end
```

2nd case. $tail(p^b_i) = \tau Q' \in \Theta \times \mathcal{X}$

```
havoc (x_J);
assume ρ_τ(x_I, x_J);
x_I ← x_J;
PC ← Q' ;     // query^k_{Q'}(x_I, x_O)
goto start;   // return
```

3rd case. $tail(p^c_i) = \langle \tau\, Q^{init}_j\, \tau \rangle\, Q' \in \langle \Theta \times \mathcal{X} \times \Theta \rangle \times \mathcal{X}$

```
     havoc (x_J, x_K, x_L);
     assume ρ_{⟨τ}(x_I, x_J) ;        /* call relation */
     assume ρ_{τ⟩}(x_K, x_L) ;        /* return relation */
     assume φ_τ(x_I, x_L) ;           /* frame relation */
     goto ord or rod;
ord: query^{k-1}_{Q^init_j}(x_J, x_K);   /* in order exec. */
     x_I ← x_L ;
     PC ← Q' ;                           // query^k_{Q'}(x_I, x_O)
     goto start;                         // return
rod: query^{k-1}_{Q'}(x_L, x_O);       /* out of order exec. */
     x_I ← x_J ;
     x_O ← x_K ;
     PC ← Q^init_j ;                     // query^k_{Q^init_j}(x_I, x_O)
     goto start;                         // return
```

In Alg. 1, p^α_i where $\alpha \in \{a, b, c\}$ refers to a production of the visibly pushdown grammar $G_{\mathcal{P}}$. The same symbol in boldface refers to the labelled statements in Alg. 1. The superscript $\alpha \in \{a, b, c\}$ differentiate the productions whether they are of the form $Q \to \tau$, $Q \to \tau Q'$ or $Q \to \langle \tau Q^{init}_j \tau \rangle Q'$, respectively.

Theorem 1 relates the semantics of $\mathcal{H}^K$ and the K-index semantics of $\mathcal{P}$. Given k, $1 \leqslant k \leqslant K$ and a control state q of $\mathcal{P}$, we show equality between $[\![\mathcal{H}^K]\!]^{i/o}_{query^k_Q}$ and $[\![\mathcal{P}]\!]^{(k)}_q$ over common variables. Before starting, we fix an arbitrary value for K and require that each k is such that $1 \leqslant k \leqslant K$. Hence, we drop K in $\mathcal{H}^K$ and write $\mathcal{H}$.

Inspection of the code of $\mathcal{H}$ reveals that $\mathcal{H}$ simulates k-index depth first derivations of $G_{\mathcal{P}}$ and interprets the statements of $\mathcal{P}$ on its local variables while applying derivation steps. The main difference with the normal execution of $\mathcal{P}$ is that $\mathcal{H}$ may interpret a procedure call and its continuation in an order which differs from the expected one.

Example 5. Let us consider an execution of *query* for the call $query^2_{Q_1^{init}}((1\ 0),(1\ 2))$ following $Q_1^{init} \stackrel{p_1^b p_2^b p_3^c}{\Longrightarrow} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle Q_4 \stackrel{p_4^a}{\Longrightarrow} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle\tau_4 \stackrel{p_1^b p_5^b p_6^b p_7^a}{\Longrightarrow} \tau_1\tau_2\langle\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\rangle\tau_4$. In the table below, the first row (labelled k/PC) gives the caller ($1 = query^1_{Q_4}$, $2 = query^2_{Q_1^{init}}$) and the value of PC when control hits the labelled statement given at the second row (labelled ip). The third row (labelled $\mathbf{x}_I/\mathbf{x}_O$) represents the content of the two arrays. $\mathbf{x}_I/\mathbf{x}_O = (a\ b)(c\ d)$ says that, in $\mathbf{x}_I$, x has value a and z has value b; in $\mathbf{x}_O$, x has value c and z has value d.

k/PC	$2/Q_1^{init}$	$2/Q_1^{init}$	$2/Q_2$	$2/Q_2$	$2/Q_3$	$2/Q_3$	$2/Q_3$
ip	**start**	$\mathbf{p_1^b}$	**start**	$\mathbf{p_2^b}$	**start**	$\mathbf{p_3^c}$	**rod**
$\mathbf{x}_I/\mathbf{x}_O$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$
k/PC	$1/Q_4$	$1/Q_4$	$2/Q_1^{init}$	$2/Q_1^{init}$	$2/Q_2$	$2/Q_2$	$2/Q_5$
ip	**start**	$\mathbf{p_4^a}$	**start**	$\mathbf{p_1^b}$	**start**	$\mathbf{p_5^b}$	**start**
$\mathbf{x}_I/\mathbf{x}_O$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$
k/PC	$2/Q_5$	$2/Q_6$	$2/Q_6$				
ip	$\mathbf{p_6^b}$	**start**	$\mathbf{p_7^a}$				
$\mathbf{x}_I/\mathbf{x}_O$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$				

The execution of $query^2_{Q_1^{init}}$ starts on row 1, column 1 and proceeds until the call to $query^1_{Q_4}$ at row 2, column 1 (the out of order case). The latter ends at row 2, column 2, where the execution of $query^2_{Q_1^{init}}$ resumes. Since the execution is out of order, and the previous $\mathbf{havoc}(\mathbf{x}_J, \mathbf{x}_K, \mathbf{x}_L)$ results into $\mathbf{x}_J = (0\ 0)$, $\mathbf{x}_K = (42\ 0)$ and $\mathbf{x}_L = (1\ 0)$ (this choice complies with the call relation), the values of $\mathbf{x}_I/\mathbf{x}_O$ are updated to $(0\ 0)/(42\ 0)$. The choice for equal values (0) of z in both $\mathbf{x}_I$ and $\mathbf{x}_0$ is checked in row 3, column 3. ■

Theorem 1. *Let $\mathcal{P} = \langle P_1, \ldots, P_n\rangle$ be a program, $\mathbf{x} = \bigcup_{i=1}^n \mathbf{x}_i$ be the set of all variables in $\mathcal{P}$, and let $q \in n\mathcal{F}(P_i)$ be a non-final control state of some procedure $P_i = \langle \mathbf{x}_i, \overrightarrow{\mathbf{x}}_i^{in}, \overrightarrow{\mathbf{x}}_i^{out}, S_i, q_i^{init}, F_i, \Delta_i\rangle$. Then, for any $k \geqslant 1$, we have:*

$$[\![\mathcal{H}]\!]^{i/o}_{query^k_Q} = \{I \cdot O \in \mathbb{Z}^{\mathbf{x}\times\mathbf{x}} \mid \langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q\}$$

Consequently, we also have:

$$[\![\mathcal{P}]\!]^{(k)}_q = \{\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \mid I \cdot O \in [\![\mathcal{H}]\!]^{i/o}_{query^k_Q}\}$$

The proof of Thm. 1 is based on the following lemma.

Lemma 2. *Let $k \geqslant 1$, q be a non-final control state of P_i and $I, O \in \mathbb{Z}^{\mathbf{x}}$. If $I \cdot O \in pre(query^k_Q)$ then $\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q$. Conversely, if $\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q$ then there exists $I', O' \in \mathbb{Z}^{\mathbf{x}}$ such that $I'\downarrow_{\mathbf{x}_i} = I\downarrow_{\mathbf{x}_i}$, $O'\downarrow_{\mathbf{x}_i} = O\downarrow_{\mathbf{x}_i}$ and $I' \cdot O' \in pre(query^k_Q)$.*

Proof: First we consider a tail-recursive version of Algorithm 1 which is obtained by replacing every two statements of the form $PC \leftarrow X$; **goto start**; by statements $query^k_X(\mathbf{x}_I, \mathbf{x}_O)$; **return**; (as it appears in the comments of Alg. 1). The equivalence between Algorithm 1 and its tail-recursive variant is an easy exercise.

"$\Leftarrow$" Let $\langle I\downarrow_{\mathbf{x_i}}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q$. By definition of k-index semantics, there exists a tagged word $\alpha \in L^{(k)}_Q(G_{\mathcal{P}})$ such that $\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\alpha]\!]$. Let p_1 be the first production used in

the derivation of α and let $\ell \geqslant 1$ be the length (in number of productions used) of the derivation. Our proof proceeds by induction on ℓ. If $\ell = 1$ then we find that p_1 must be of the form $Q \to \tau$ and that $\alpha = \tau$. Therefore we have $[\![\alpha]\!] = [\![\tau]\!] = \rho_\tau$ and moreover $\langle I{\downarrow_{\mathbf{x}_i}}, O{\downarrow_{\mathbf{x}_i}} \rangle \in \rho_\tau$. Since $k \geqslant 1$, we let $I' = I$, $O' = O$ and we find that $query_Q^k(I', O')$ returns by choosing to jump to the label corresponding to p_1, then executing the **assume** statement and finally the **return** statement. Thus $I' \cdot O' \in pre(query_Q^k)$. For $\ell > 1$, the proof divides in two parts.

1. If p_1 is of the form $Q \to \tau Q'$ then we find that $\alpha = \tau\beta$, for some tagged word β. Moreover, $\langle I{\downarrow_{\mathbf{x}_i}}, O{\downarrow_{\mathbf{x}_i}} \rangle \in [\![\alpha]\!] = \rho_\tau \circ [\![\beta]\!]$ by definition of the semantics. This implies that there exists $J \in \mathbb{Z}^{\mathbf{x}}$ such that $\langle I{\downarrow_{\mathbf{x}_i}}, J{\downarrow_{\mathbf{x}_i}} \rangle \in \rho_\tau$ and $\langle J{\downarrow_{\mathbf{x}_i}}, O{\downarrow_{\mathbf{x}_i}} \rangle \in [\![\beta]\!]$. Hence, we conclude from $\beta \in L_{Q'}^{(k)}(G_{\mathcal{P}})$, and the fact that the derivation $Q' \overset{(k)}{\Longrightarrow}{}^\star \beta$ has less productions than $Q \overset{(k)}{\Longrightarrow}{}^\star \alpha$, that $\langle J{\downarrow_{\mathbf{x}_i}}, O{\downarrow_{\mathbf{x}_i}} \rangle \in [\![\mathcal{P}]\!]_{Q'}^{(k)}$. Applying the induction hypothesis on this last fact, we find that $J \cdot O \in pre(query_{Q'}^k)$. Finally consider the call $query_Q^k(I, O)$ where at label **start** the jump goes to label corresponding to p_1. At this point in the execution **havoc**$(\mathbf{x}_J)$ returns J. Next **assume** $\rho_\tau(I, J)$ succeeds. Finally we find that the call to $query_Q^k(I, O)$ returns because so does the call $query_{Q'}^k(J, O)$ which is followed by **return**. Hence $I \cdot O \in pre(query_Q^k)$.

2. If p_1 is of the form $Q \to \langle \tau Q_j^{init} \tau \rangle Q'$ then we find that $\alpha = \langle \tau \beta' \tau \rangle \beta$ for some β', β. Lemma 1 (prop. 3) shows that either $\beta' \in L_{Q_j^{init}}^{(k-1)}(G_{\mathcal{P}})$ and $\beta \in L_{Q'}^{(k)}(G_{\mathcal{P}})$ or $\beta' \in L_{Q_j^{init}}^{(k)}(G_{\mathcal{P}})$ and $\beta \in L_{Q'}^{(k-1)}(G_{\mathcal{P}})$, and both derivations have less productions than ℓ. We will assume the former case, the latter being treated similarly. Moreover, $\langle I{\downarrow_{\mathbf{x}_i}}, O{\downarrow_{\mathbf{x}_i}} \rangle \in [\![\alpha]\!] = CaRet_\tau \circ [\![\beta]\!] = \Big(\big(\rho_{\langle\tau} \circ [\![\beta']\!] \circ \rho_{\tau\rangle}\big) \cap \phi_\tau \Big) \circ [\![\beta]\!] \subseteq \Big(\big(\rho_{\langle\tau} \circ [\![\mathcal{P}]\!]_{q_j^{init}}^{(k-1)} \circ \rho_{\tau\rangle}\big) \cap \phi_\tau \Big) \circ [\![\mathcal{P}]\!]_{q'}^{(k)}$. Hence there exists $J, K, L \in \mathbb{Z}^{\mathbf{x}}$ such that $\langle I{\downarrow_{\mathbf{x}_i}}, J{\downarrow_{\mathbf{x}_i}} \rangle \in \rho_{\langle\tau}$, $\langle J{\downarrow_{\mathbf{x}_i}}, K{\downarrow_{\mathbf{x}_j}} \rangle \in [\![\mathcal{P}]\!]_{q_j^{init}}^{(k-1)}$, $\langle K{\downarrow_{\mathbf{x}_j}}, L{\downarrow_{\mathbf{x}_j}} \rangle \in \rho_{\tau\rangle}$, and $\langle L{\downarrow_{\mathbf{x}_i}}, O{\downarrow_{\mathbf{x}_i}} \rangle \in [\![\mathcal{P}]\!]_{q'}^{(k)}$. Applying the induction hypothesis on the derivations of β' and β, we obtain that $J \cdot K \in pre(query_{Q_j^{init}}^{k-1})$ and $L \cdot O \in pre(query_{Q'}^k)$. Given those facts, it is routine to check that $query_Q^k(I', O')$ returns by choosing to jump to the label corresponding to p_1, then having **havoc**$(\mathbf{x}_J, \mathbf{x}_K, \mathbf{x}_L)$ return (J, K, L), hence $I' \cdot O' \in pre(query_Q^k)$.

The only if direction is proven in the technical report [11]. □

As a last point, we prove that the bounded-index sequence $\{[\![\mathcal{P}]\!]^{(k)}\}_{k=1}^{\infty}$ satisfies several conditions that advocate its use in program analysis, as an underapproximation sequence. The subset order and set union is extended to tuples of relations, point-wise.

$$[\![\mathcal{P}]\!]^{(k)} \subseteq [\![\mathcal{P}]\!]^{(k+1)} \quad \text{for all } k \geqslant 1 \quad (A1)$$
$$[\![\mathcal{P}]\!] = \bigcup_{k=1}^{\infty} [\![\mathcal{P}]\!]^{(k)} \quad (A2)$$

Condition $(A1)$ requires that the sequence is monotonically increasing, the limit of this increasing sequence being the actual semantics of the program $(A2)$. These conditions follow however immediately from the two first points of Lemma 1. To decide whether the limit $[\![\mathcal{P}]\!]$ has been reached by some iterate $[\![\mathcal{P}]\!]^{(k)}$, it is enough to check that the

tuple of relations in $[\![\mathcal{P}]\!]^{(k)}$ is inductive with respect to the statements of $\mathcal{P}$. This can be implemented as an SMT query.

5 Completeness of Underapproximations for Bounded Programs

In this section we define a class of recursive programs for which the precise summary semantics of each program in that class is effectively computable. We show for each program $\mathcal{P}$ in the class that (*a*) $[\![\mathcal{P}]\!] = [\![\mathcal{P}]\!]^{(k)}$ for some value $k \geqslant 1$, bounded by a linear function in the total number $loc(\mathcal{P})$ of control states in $\mathcal{P}$, and moreover (*b*) the semantics of $\mathcal{H}^k$ is effectively computable (and so is that of $[\![\mathcal{P}]\!]^{(k)}$ by Thm. 1).

Given an integer relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$, its *transitive closure* $R^+ = \bigcup_{i=1}^{\infty} R^i$, where $R^1 = R$ and $R^{i+1} = R^i \circ R$, for all $i \geqslant 1$. In general, the transitive closure of a relation is not definable within decidable subsets of integer arithmetic, such as Presburger arithmetic. In this section we consider two classes of relations, called *periodic*, for which this is possible, namely octagonal relations, and finite monoid affine relations. The formal definitions are deferred to the technical report [11].

We define a *bounded-expression* $\mathbf{b}$ to be a regular expression of the form $\mathbf{b} = w_1^* \ldots w_k^*$, where $k \geqslant 1$ and each w_i is a non-empty word. A language (not necessarily context-free) L over alphabet Σ is said to be *bounded* if and only if L is included in (the language of) a bounded expression $\mathbf{b}$.

Theorem 2 ([18]). *Let $G = (\mathcal{X}, \Sigma, \delta)$ be a grammar, and $X \in \mathcal{X}$ be a nonterminal, such that $L_X(G)$ is bounded. Then there exists a linear function $\mathcal{B} : \mathbb{N} \to \mathbb{N}$ such that $L_X(G) = L_X^{(k)}(G)$ for some $1 \leqslant k \leqslant \mathcal{B}(\mathcal{X})$.*

If the grammar in question is $G_{\mathcal{P}}$, for a program $\mathcal{P}$, then clearly $\mathcal{X} = loc(\mathcal{P})$, by definition. The class of programs for which our method is complete is defined below:

Definition 2. *Let $\mathcal{P}$ be a program and $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$ be its corresponding visibly pushdown grammar. Then $\mathcal{P}$ is said to be* bounded periodic *if and only if:*

1. *$L_X(G_{\mathcal{P}})$ is bounded for each $X \in \mathcal{X}$;*
2. *each relation ρ_τ occurring in the program, for some $\tau \in \widehat{\Theta}$, is periodic.*

Example 6. (continued from Ex. 4) Recall that $L_{Q_1^{init}}(G_{\mathcal{P}}) = L^{(2)}_{Q_1^{init}}(G_{\mathcal{P}})$ which equals to the set $\{(\tau_1\tau_2\langle\tau_3)^n\tau_1\tau_5\tau_6\tau_7(\tau_3\rangle\tau_4)^n \mid n \geqslant 0\} \subseteq (\tau_1\tau_2\langle\tau_3)^*\tau_1^*\tau_5^*\tau_6^*\tau_7^*(\tau_3\rangle\tau_4)^*$. ■

Concerning condition 1, it is decidable [12] and previous work [14] defined a class of programs following a recursion scheme which ensures boundedness of the set of interprocedurally valid paths.

This section shows that the underapproximation sequence $\{[\![\mathcal{P}]\!]^{(k)}\}_{k=1}^{\infty}$, defined in Section 4, when applied to any bounded periodic programs $\mathcal{P}$, always yields $[\![\mathcal{P}]\!]$ in at most $\mathcal{B}(loc(\mathcal{P}))$ steps, and moreover each iterate $[\![\mathcal{P}]\!]^{(k)}$ is computable and Presburger definable. Furthermore the method can be applied *as it is* to bounded periodic programs, without prior knowledge of the bounded expression $\mathbf{b} \supseteq L_Q(G_{\mathcal{P}})$.

The proof goes as follows. Because $\mathcal{P}$ is bounded periodic, Thm. 2 shows that the semantics $[\![\mathcal{P}]\!]$ of $\mathcal{P}$ coincide with its k-index semantics $[\![\mathcal{P}]\!]^{(k)}$ for some $1 \leqslant k \leqslant$

$\mathcal{B}(loc(\mathcal{P}))$. Hence, the result of Thm. 1 shows that for each $q \in n\mathcal{F}(\mathcal{P})$, the k-index semantics $[\![\mathcal{P}]\!]_q^{(k)}$ is given by the semantics $[\![\mathcal{H}]\!]_{query_Q^k}$ of procedure $query_Q^k$ of the program $\mathcal{H}$. Then, because $\mathcal{P}$ is bounded, we show in Thm. 3 that every procedure $query_Q^k$ of program $\mathcal{H}$ is *flattable* (Def. 3). Moreover, since the only transitions of $\mathcal{H}$ which are not from $\mathcal{P}$ are equalities and **havoc**, all transitions of $\mathcal{H}$ are periodic. Since each procedure $query_Q^k$ is flattable then $[\![\mathcal{P}]\!]$ is computable in finite time by existing tools, such as FAST [6] or FLATA [8, 7]. In fact, these tools are guaranteed to terminate provided that (a) the input program is flattable; and (b) loops are labeled with periodic relations.

Definition 3. *Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a non-recursive program and $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$ be its corresponding visibly pushdown grammar. Procedure P_i is said to be* flattable *if and only if there exists a bounded and regular language R over $\widehat{\Theta}$, such that $[\![\mathcal{P}]\!]_{P_i} = \bigcup_{\alpha \in L_{P_i}(G_{\mathcal{P}}) \cap R} [\![\alpha]\!]$.*

Notice that a flattable program is not necessarily bounded (Def. 2), but its semantics can be computed by looking only at a bounded subset of interprocedurally valid sequence of statements.

Theorem 3. *Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a bounded program, and let $q \in n\mathcal{F}(\mathcal{P})$. Then, for any $k \geqslant 1$, procedure $query_Q^k$ of program $\mathcal{H}$ is and flattable.*

The proof of Thm. 3 roughly goes as follows: recall that we have $[\![\mathcal{P}]\!]_q = [\![\mathcal{P}]\!]_q^{(k)}$ for each $q \in n\mathcal{F}(\mathcal{P})$ and so it is sufficient to consider the set $L_Q^{(k)}(G_{\mathcal{P}})$ of interprocedurally valid paths. We further show (Thm. 4) that a strict subset of the k-index derivations of $G_{\mathcal{P}}$ is sufficient to capture $L_Q^{(k)}(G_{\mathcal{P}})$. Moreover this subset of derivations is characterizable by a regular bounded expression $\mathbf{b}_\Gamma$ over the productions of $G_{\mathcal{P}}$. Next, we map $\mathbf{b}_\Gamma$ into a set $f(\mathbf{b}_\Gamma)$ of interprocedurally valid paths of procedure $query_Q^k$ of $\mathcal{H}$, which is sufficient to capture $[\![\mathcal{H}]\!]_{query_Q^k}$. Finally, using existing results [12], we show in Thm. 5 that $f(\mathbf{b}_\Gamma)$ is a bounded and regular set. Hence, we conclude that each procedure $query_Q^k$ is flattable. A full proof of Thm. 3 is given in the technical report [11].

Given a grammar $G = (\mathcal{X}, \Sigma, \delta)$, we call any subset of δ^* a *control set*. Let Γ be a control set, we denote by $L_X(\Gamma, G) = \{w \in \Sigma^* \mid \exists \gamma \in \Gamma \colon X \stackrel{\gamma}{\Longrightarrow} w\}$, the set of words resulting from derivations with control word in Γ.

Definition 4 ([19]). *Let $D \equiv X = w_0 \Longrightarrow^* w_m = w$ be a derivation. Let $k > 0$, $x_i \in \Sigma^*$, $A_i \in \mathcal{X}$ such that $w_m = x_0 A_1 x_1 \cdots A_k x_k$; and for each m and i such that $0 \leqslant m \leqslant n$, $1 \leqslant i \leqslant k$, let $f_m(i)$ denote the index of the first word in D in which the particular occurrence of variable A_i appears. Let A_j be the nonterminal replaced in step $w_m \Longrightarrow w_{m+1}$ of D. Then D is said to be* depth-first *if and only if for all m, $0 \leqslant m < n$ we have $f_m(i) \leqslant f_m(j)$, for all $1 \leqslant i \leqslant k$.*

We define the set $DF_X(G)$ ($DF_X^{(k)}(G)$) of words produced using only depth-first derivations (of index at most k) in G starting from X. Clearly, $DF_X(G) \subseteq L_X(G)$ and similarly $DF_X^{(k)}(G) \subseteq L_X^{(k)}(G)$ for all $k \geqslant 1$. We further define the set $DF_X(\Gamma, G)$ ($DF_X^{(k)}(\Gamma, G)$) of words produced using depth-first derivations (of index at most k) with control words from Γ.

The following theorem shows that $L_Q^{(k)}(G_{\mathcal{P}})$ is captured by a subset of depth-first derivations whose control words belong to some bounded expression.

Theorem 4. *Let $G = (\mathcal{X}, \Theta, \delta)$ be a visibly pushdown grammar, $X_0 \in \mathcal{X}$ be a nonterminal such that $L_{X_0}(G)$ is bounded. Then for each $k \geqslant 1$ there exists a bounded expression $\mathbf{b}_\Gamma$ over δ such that $DF_{X_0}^{(k)}(\mathbf{b}_\Gamma, G) = L_{X_0}^{(k)}(G)$.*

Finally, to conclude that $query_Q^k$ is flattable, we map the k-index depth-first derivations of G into the interprocedurally valid paths of $query_Q^k$. Then, applying Thm. 5 on that mapping, we conclude the existence of a bounded and regular set of interprocedurally valid paths of $query_Q^k$ sufficient to capture its semantics.

Theorem 5. *Given two alphabets Σ and Δ, let f be a function from Σ^* into Δ^* such that (i) if u is a prefix of v then $f(u)$ is a prefix of $f(v)$; (ii) there exists an integer M such that $|f(wa)| - |f(w)| \leqslant M$ for all $w \in \Sigma^*$ and $a \in \Sigma$; (iii) $f(\varepsilon) = \varepsilon$; (iv) $f^{-1}(R)$ is regular for all regular languages R. Then f preserves regular sets. Furthermore, for each bounded expression $\mathbf{b}$ we have that $f(\mathbf{b})$ is bounded.*

6 Experiments

We have implemented the proposed method in the FLATA verifier [16] and experimented with several benchmarks. First, we have considered several programs, taken from [1], that perform arithmetic and logical operations in a recursive way such as `plus` (addition), `timesTwo` (multiplication by two), `leq` (comparison), and `parity` (parity checking). It is worth noting that these programs have finite index and stabilization of the underapproximation sequence is thus guaranteed. Our technique computes summaries by verifying that $[\![\mathcal{P}]\!]^{(2)} = [\![\mathcal{P}]\!]^{(3)}$ for all these benchmarks, see Table 1 (the platform used for experiments is Intel® Core™ i7-3770K CPU, 3.50GHz, with 16GB of RAM).

Table 1. Experiments

Program	Time [s]	k
`timesTwo`	0.7	2
`leq`	0.8	2
`parity`	0.8	2
`plus`	1.5	2
$F_{a=2}$	1.5	3
$F_{a=8}$	36.9	4
$G_{b=12}$	3.5	3
$G_{b=13}$	23.2	3
$G_{b=14}$	23.4	3

$$F_a(x) = \begin{cases} x - 10 & \text{if } x \geqslant 101 \\ (F_a)^a(x + 10 \cdot a - 9) & \text{if } x \leqslant 100 \end{cases} \qquad G_b(x) = \begin{cases} x - 10 & \text{if } x \geqslant 101 \\ G(G(x+b)) & \text{if } x \leqslant 100 \end{cases}$$

Next, we have considered the generalized McCarthy 91 function [9], a well-known verification benchmark that has long been a challenge. We have automatically computed precise summaries of its generalizations F_a and G_b above for $a = 2, \ldots, 8$ and $b = 12, \ldots, 14$. The indices of the recursive programs implementing the F_a, G_b functions are not bounded, however the sequence reached the fixpoint after at most 4 steps.

7 Conclusions

We have presented an underapproximation method for computing summaries of recursive programs operating on integers. The underapproximation is driven by bounding

the index of derivations that produce the execution traces of the program, and computing the summary, for each index, by analyzing a non-recursive program. We also present a class of programs on which our method is complete. Finally, we report on an implementation and experimental evaluation of our technique.

References

1. Termination Competition 2011, `http://termcomp.uibk.ac.at/termcomp/home.seam`
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: WHALE: An Interpolation-Based Algorithm for Inter-procedural Verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012)
3. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56(3), 16 (2009)
4. Atig, M.F., Ganty, P.: Approximating petri net reachability along context-free traces. In: FSTTCS 2011. LIPIcs, vol. 13, pp. 152–163. Schloss Dagstuhl (2011)
5. Cook, B., Podelski, A., Rybalchenko, A.: Summarization for termination: no return! Formal Methods in System Design 35, 369–387 (2009)
6. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Fast Acceleration of Symbolic Transition Systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 118–121. Springer, Heidelberg (2003)
7. Bozga, M., Iosif, R., Konečný, F.: Fast Acceleration of Ultimately Periodic Relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
8. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundamenta Informaticae 91(2), 275–303 (2009)
9. Cowles, J.: Knuth's generalization of McCarthy's 91 function. In: Computer-aided Reasoning, pp. 283–299 (2000)
10. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian program analysis. JACM 57(6), 33:1–33:47 (2010)
11. Ganty, P., Iosif, R., Konečný, F.: Underapproximation of procedure summaries for integer programs. CoRR abs/1210.4289 (2012)
12. Ginsburg, S.: The Mathematical Theory of Context-Free Languages. McGraw-Hill, Inc., New York (1966)
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL 2010, pp. 43–56. ACM (2010)
14. Godoy, G., Tiwari, A.: Invariant Checking for Programs with Procedure Calls. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 326–342. Springer, Heidelberg (2009)
15. Gruska, J.: A few remarks on the index of context-free grammars and languages. Information and Control 19(3), 216–223 (1971)
16. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A Verification Toolkit for Numerical Transition Systems - Tool Paper. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012)
17. Lalire, G., Argoud, M., Jeannet, B.: Interproc., `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
18. Luker, M.: A family of languages having only finite-index grammars. Information and Control 39(1), 14–18 (1978)
19. Luker, M.: Control sets on grammars using depth-first derivations. Mathematical Systems Theory 13, 349–359 (1980)
20. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61. ACM (1995)
21. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, ch. 7, pp. 189–233. Prentice-Hall, Inc. (1981)