Automated Resource Analysis with paicc*
(Extended Abstract)

Michael Schaper[1]

1 **Department of Computer Science**
University of Innsbruck, Austria
michael.schaper@uibk.ac.at

Abstract

In this extended abstract we present paicc, a transformation from guarded control-flow programs to unstructured loop programs. We make use of recent results on the decidability of polynomial growth-rate of unstructured loop programs to develop a new method for assessing polynomial runtime of guarded control-flow programs.

1998 ACM Subject Classification F.3.2, F.4.1

Keywords and phrases program analysis, resource analysis, implicit computational complexity

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Automated resource analysis of imperative programs is an active area of research. In recent years several approaches have been investigated and implemented, to mention a few: KoAT, Rank, CoFloCo, Loopus and C4B [8, 1, 10, 12, 9]. In this work-in-progress we present paicc which aims to combine the state-of-the-art in program analysis and implicit computational complexity.

Program analysis tools often target low-level but expressive programming languages with unstructured control-flow, such as LLVM or Java bytecode. For these languages interesting properties like termination and resource consumption are undecidable. Hence, program abstractions, heuristics, and incomplete decision procedures are researched and incorporated in modern tools.

On the other hand, decidable properties of abstract but still expressive programming fragments have been investigated in implicit computational complexity [3, 7]. Most relevant, *polynomial growth-rate* of variables, ie. whether the value of a variable after executing a program is bounded by a polynomial in the input, is decidable for LOOP programs [3], a variant of Meyer and Ritchie's loop programs [11] with weak semantics. In a recent work by Ben-Amram and Pineles this result has been extended to a variant of LOOP programs with unstructured control-flow [6, 5]. Notable, this language is very close to, but not quite the same as, target abstractions used in state-of-the-art resource analysis tools.

In this work we are interested in closing this gap. We are going to present an abstraction from guarded control-flow programs to unstructured LOOP programs. This abstraction over-approximates the growth-rate of variables of the original program. In combination with the analysis of [6, 5] we obtain an automated method for assessing polynomial runtime of the original program.

* This work was partially supported by DARPA/AFRL contract number FA8750-17-C-088.

2 Preliminaries

We first introduce CFG-GOTO programs. We fix a finite set of program Variables $\mathcal{V}$ and represent programs as labelled *control flow graphs* (CFGs for short). Nodes are labelled with locations and edges are labelled with constraints. A *constraint* C is a conjunction of (in)equalities of polynomial expressions over $\mathcal{V} \cup \mathcal{V}' \cup \mathbb{Z}$. Here $\mathcal{V}'$ denotes primed versions of variables in $\mathcal{V}$ and indicates the valuation of all variables after traversing an edge in the CFG. So, constraints express guarded non-deterministic parallel assignments. A *configuration* with location l and state $\vec{x} \in \mathbb{Z}^n$ is a tuple $(l, \vec{x})$. A *trace* is a sequence of configurations $(l_i, \vec{x}_i) \to^{C_i} (l_{i+1}, \vec{x}_{i+1}) \to^{C_{i+1}} \dots$ such that for all $j \geqslant i$ there exists an edge $l_j \to^{C_j} l_{j+1}$ in the CFG and constraint C_j holds with pre-state $\vec{x}_j$ and post-state $\vec{x}'_{j+1}$. We usually denote a trace with ρ and the evaluation of a trace as $\vec{x} \rightsquigarrow_\rho \vec{x}'$. The *size* of a value $i \in \mathbb{Z}$ is its absolute value, denoted $\|i\|$. We abuse notation and extend $\|\cdot\|$ and $\leqslant$ componentwise, that is $\|\vec{x}\| = (\|x_1\|, \dots, \|x_n\|)$ and $\vec{x} \leqslant \vec{y}$ iff $(x_1 \leqslant y_1, \dots, x_n \leqslant y_n)$. The *growth-rate* of a variable x_i wrt. to a set of traces $\mathcal{T}$ and input size vector $\vec{n} \in \mathbb{N}^n$ is

$$gr_{x_i}(\mathcal{T}, \vec{n}) = \sup\{\|x'_i\| \mid \|\vec{x}\| \leqslant \vec{n} \wedge \exists \rho \in \mathcal{T}.\ \vec{x} \rightsquigarrow_\rho \vec{x}'\} \ .$$

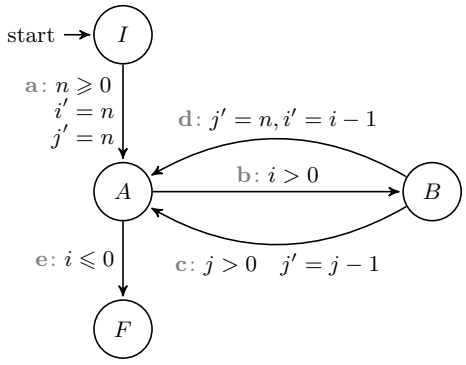
We say that the growth-rate of a variable x_i is *polynomially bounded* in $\mathcal{T}$ if there exists a polynomial p on $\vec{x}$, denoted $p(\vec{x})$, st. $gr_{x_i}(\mathcal{T}, \vec{n}) \leqslant p(\vec{x})$.

To analyse the *runtime complexity* of a program we make use of the well-known fact that we can instrument the program with a dedicated counter variable together with tick instructions and estimate the growth-rate of the counter variable.

▶ **Example 1.** Figure 1 depicts the motivating example of [6] together with its CFG-GOTO representation. When depicting CFG programs we omit the identity constraints, eg. $n' = n$.

```
assume n>=0
i = n; j = n
while i>0:
  if j > 0
    j = j-1
  else:
    j = n; i = i-1
```

(a)



(b)

Figure 1 Illustrating example and its CFG-GOTO representation.

We introduce CFG-LOOP programs as variants of CFG-GOTO programs. For details we refer to [6, 5]. States are n-tuples over natural numbers, ie. $\vec{x} \in \mathbb{N}^n$. Constraints are conjunctions of inequalities of the form $x'_i \leqslant c_1 x_1 + \dots + c_n x_n + c_k k$, that is, x'_i is not greater than a linear combination of $\vec{x}$. Here $c_1, \dots, c_n, c_k \in \mathbb{N}$ and k is a dedicated variable representing an arbitrary but constant value. Constraints in CFG-LOOP programs do not determine control-flow and without further restrictions such programs are non-terminating. To remedy this fact we introduce *loop structures*. Loop structures are akin to call-graphs and represent a nesting hierarchy of bounded subprograms, so called *loops*. Alternatively we can construe loop structures as an explicit representation of program decomposition.

More formally. A loop structure of a CFG is a set of subsets of edges, called loops, which form a rooted tree. The root consists of all edges of the CFG. The children of a loop L are disjoint proper subsets of L. The *cut set* of a loop L consists of all edges in L that are not edges of its descendants. Furthermore, associated to each loop is a bound l that is an expression $c_1x_1 + \cdots + c_nx_n + c_kk$. The bound associated to the root is constant 1.

The semantics of a loop L can be interpreted as follows: Assume loop L is entered with state $\vec{x}$. Then the bound expression of L instantiated with state $\vec{x}$ provides a concrete bound on the number of traversals of an edge of the cut set of L before exiting L.

CFG-LOOP programs have similar properties as LOOP programs, as presented in [3]. A CFG-LOOP program is always terminating and the growth-rates of variables are bounded by the class of primitive recursive functions.

▶ **Example 2** (Cont'd). Figure 2 illustrates a CFG-LOOP program. The loop structure is indicated by dotted boxes. The inner loop consists of edges $\{\mathbf{b}, \mathbf{c}\}$, the edges of its cut set $\{\mathbf{b}, \mathbf{c}\}$ can be traversed at most $\|j\|$ times, before exiting. The outer loop consists of edges $\{\mathbf{b}, \mathbf{c}, \mathbf{d}\}$, the edges of its cut set $\{\mathbf{d}\}$ can be traversed at most $\|i\|$ times. The root consists of all edges, edges $\mathbf{a}$ and $\mathbf{e}$ can be traversed only once.

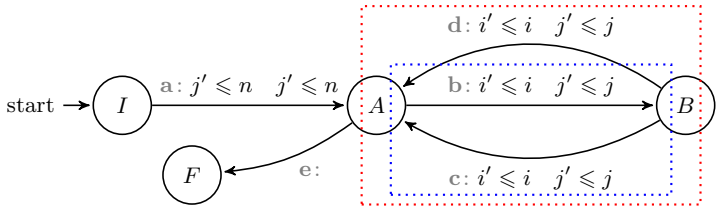


Figure 2 CFG-LOOP abstraction of our running example.

We recall the main result of [6]. Sometimes we refer to the growth-rate analysis as bp-analysis. The bounds that are obtained via the bp-analysis are not precise as it only proofs the existence of a polynomial that bounds the growth-rate. It is an open problem how to obtain more precise bounds, like the degree of the polynomial, via the bp-analysis.

▶ Proposition 1 (Polynomial Growth-Rate). Polynomial growth-rate is decidable for CFG-LOOP programs.

3 Transformation

In this section we present a (lossy) transformation from CFG-GOTO programs to CFG-LOOP programs that safely approximates the growth-rate of variables.

3.1 Synthesis of Loop Structures

The loop structure of a CFG is not unique and a semantic rather than syntactic property. The example program of Figure 1b is used in [6] as motivating example, though no algorithm for the automated construction of its loop structure is given.

For construction, we synthesize a *lexicographic combination of linear ranking functions*, that is a k-dimensional ranking function with co-domain $(\mathbb{N}^k, \preceq_k)$, consisting of k linear functions. Here $\preceq_k$ is the standard lexicographic order on integer vectors. Multidimensional ranking functions can be used to bound the runtime of subprograms. This variant of ranking functions have been proposed for runtime analysis in [1] and are used in combination

with global size invariants on variables to estimate the cardinality of the state space. In contrast, here, the obtained bounds are used in combination with the bp-analysis to infer the growth-rate of variables.

Synthesis of linear ranking functions is a powerful tool in program analysis. Necessary constraints can be encoded via LP or SMT [2]. In our approach we synthesise ranking functions with strict and weak oriented components, in notation $(\succ, \succsim)$. That is, given a set of edges we encode following properties: (i) all edges are *non-increasing*, (ii) some edges are *decreasing* and *bounded*. This provides a bound on decreasing edges and conforms to the bound of traversing edges of the cut set of a loop.

Following algorithm synthesises a lexicographic combination of linear ranking functions and constructs a loop structure for the given CFG. The algorithm fails if no ranking function for the considered (sub)program can be established.

(i) The root node L of the loop structure consists of all edges of the CFG.
(ii) For the considered (sub)program add the edges of each SCC_i in the CFG as child node L_i to the current node.
(iii) For each leaf L_i synthesize a linear ranking function $(\succ_i, \succsim_i)$ of L_i such that $\succ_i$ is non-empty. The ranking function is a linear polynomial, $c_1x_1 + \cdots + c_nx_n + c_k$. The cut set of L_i is $\succ_i$ and the bound l_i of L_i is $\|c_1\|x_1 + \cdots + \|c_n\|x_n + \|c_k\|k$.
(iv) For $L_i \setminus \succ_i$ apply recursively (ii).

▶ Proposition 2 (Loop Structure). Given a CFG the proposed algorithm provides a valid loop structure. Furthermore, let $\rho = (l_0, \vec{x}_0) \to^{C_0} \cdots \to^{C_{i-1}} \rho' \cdots$ be a program trace of the CFG with sub-trace $\rho' = (l_i, \vec{x}_i) \to^{C_i} \cdots$. Assume that all edges in ρ' are contained in some loop L' but are not contained in any of its descendants. Let l' be the bound associated to L'. Then the number of occurrences of any edge in the cut set of L' is bounded in ρ' by l' instantiated with the start state of the sub-trace $\vec{x}_i$.

Conceptually similar approaches for decomposition of programs is also present in other methods albeit often more implicitly. Though the growth-rate or size-analysis either depends on globally inferred invariants or incomplete procedures [12, 8].

3.2 Local Growth-Rate Abstraction

Inferring the *local growth-rate*, ie. the growth-rate of a variable along an edge of the CFG, is a key component in the KoAT tool [8]. In combination with runtime bounds on subprograms global invariants on variables are inferred, which in turn are used to infer runtime bounds on subsequent subprograms. Conceptually the bp-analysis and thus paicc work in a similar way. The key difference is, that KoAT uses more refined bound expressions but at the same time the proposed growth-rate analysis is known to be incomplete. This trade-off between expressiveness of the abstraction and the strength of the analysis is what we want to investigate.

Following algorithm is used in paicc to infer local growth-rates: Let $l \to^C l'$ be an edge in the CFG of a `CFG-GOTO` program. We synthesize for each variable $x_i \in \mathcal{V}$ two linear ranking functions from C satisfying $x_i' \geqslant c_1x_1 + \cdots + c_nx_n + c_kk$ and $x_i' \leqslant d_1x_1 + \cdots + d_nx_n + d_kk$, representing the lower and upper bound respectively. We define the local growth-rate of x_i as $x_i' \leqslant \max(\|c_1\|, \|d_1\|)x_1 + \cdots + \max(\|c_n\|, \|d_n\|)x_n + \max(\|c_k\|, \|d_k\|)k$. The algorithm may fail to provide a bound. Following [5] we introduce for this case a special variable *unknown* and a special expression $x_i' \leqslant \mathit{unknown}$. In practice we use the bp-analysis to infer whether the counter variable depends on *unknown*.

▶ Proposition 3 (Local Growth-Rate). Let $l \to^C l'$ be an edge in the CFG and $(l, \vec{x}) \to^C (l', \vec{x}')$ be a sub-trace of a `CFG-GOTO` program. Furthermore, let D be the local growth-rate obtained from C. Then $(l, \|\vec{x}\|) \to^D (l', \|\vec{x}'\|)$ is a trace in the `CFG-LOOP` program.

▶ **Example 3** (Cont'd). The example program in Figure 2 has all its edges replaced with the local growth-rate. During execution the variables i and j never increase after assigning n at first. The bp-analysis infers that i, j and n have an *identity* dependency on n, that is they are bounded by the initial value of n. If we instrument the code with a counter variable together with tick instructions, we further obtain that the counter has a *multiplicative* dependency on n. It follows that the example program runs in polynomial time.

4 Conclusion and Future Work

In this extended abstract we investigated abstractions from guarded control-flow programs to unstructured `LOOP` programs incorporating ideas from Rank and KoAT. Using the bp-analysis we can verify polynomial growth-rate of variables and polynomial runtime of the program.

In the near future we are going to investigate the trade-off between the expressiveness of the abstractions and the strength of the analysis in practice. A prototype of paicc and initial experiments are available at `http://cbr.uibk.ac.at/tools/paicc/`. It is an open problem whether the bp-analysis can be extended to support expressions with concrete constants [4], which would provide a natural and more expressive abstraction domain.

References

1 C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Proc. 17th SAS*, volume 6337 of *LNCS*, pages 117–133, 2010.

2 R. Bagnara and F. Mesnard. Eventual Linear Ranking Functions. In *Proc. of 15th PPDP*, pages 229–238, 2013.

3 A. M. Ben-Amram, N. D. Jones, and L. Kristiansen. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Proc. of 4th CiE, 2008*, pages 67–76, 2008.

4 A. M. Ben-Amram and L. Kristiansen. On the Edge of Decidability in Complexity Analysis of Loop Programs. 23(7):1451–1464, 2012.

5 A. M. Ben-Amram and A. Pineles. Growth-Rate Analysis of Flowchart Programs, 2014. Masterthesis.

6 A. M. Ben-Amram and A. Pineles. Flowchart Programs, Regular Expressions, and Decidability of Polynomial Growth-Rate. In *Proc. of 4th VPT@ETAPS*, pages 24–49, 2016.

7 A. M. Ben-Amram and M. Vainer. Bounded Termination of Monotonicity-Constraint Transition Systems. *CoRR*, abs/1202.4281, 2012. URL: `http://arxiv.org/abs/1202.4281`.

8 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Proc. 20th TACAS*, pages 140–155, 2014.

9 Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *Proc. of 36th PLDI*, pages 467–478, 2015.

10 A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Proc. 12th APLAS*, pages 275–295, 2014.

11 A. R. Meyer and D. M. Ritchie. The Complexity of Loop Programs. In *Proceedings of the 1967 22Nd National Conference*, Proc . ACM '67, pages 465–469. ACM, 1967.

12 M. Sinn, F. Zuleger, and H. Veith. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *JAR*, 59(1):3–45, 2017.