Science of Computer Programming 102 (2015) 142–157

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Symbolic execution as a basis for termination analysis ☆

Germán Vidal

MiST, DSIC, Universitat Politècnica de València, E-46022, Valencia, Spain

ARTICLE INFO

Article history:
Received 24 April 2014
Received in revised form 1 November 2014
Accepted 29 January 2015
Available online 4 February 2015

Keywords:
Program termination
Symbolic execution
Program analysis

ABSTRACT

Program termination is a relevant property that has been extensively studied in the context of many different formalisms and programming languages. Traditional approaches to proving termination are usually based on inspecting the source code. Recently, a new semantics-based approach has emerged, which typically follows a two-stage scheme: first, a finite data structure representing the computation space of the program is built; then, termination is analyzed by inspecting the transitions in this data structure using traditional, syntax-based techniques.

Unfortunately, this approach is still specific to a programming language and semantics. In this work, we present instead a general, high-level framework that follows the semantics-based approach to proving termination. In particular, we focus on the first stage and advocate the use of symbolic execution, together with appropriate subsumption and abstraction operators, for producing a finite representation of the computations of a program. Hopefully, this higher level approach will provide useful insights for designing new semantics-based termination tools for particular programming languages.

1. Introduction

As witnessed by the extensive literature on the subject, determining whether a program terminates for all input data is a fundamental problem in computer science (see, e.g., [14,17,18,40], and references therein). In general, the techniques for proving program termination are specific to a programming language. Traditional approaches often rely on inspecting the *shape* of the program. For instance, one of the most popular approaches to analyzing termination of rewrite systems, the *dependency pairs* approach [7], is based on finding appropriate orderings that relate the left-hand side of the rules with some subterms of the right-hand side. *Size-change termination* analysis [28] for functional programs is also based on inspecting the source code, in this case finding an ordering that relates the size of the arguments of consecutive function calls. In the context of imperative programming, one can also find a flurry of works that are aimed at finding appropriate invariants to prove that all program loops are terminating by inspecting the source code (see e.g., [12,13,38,24] and references therein).

Despite the fact that some of these approaches are quite powerful, some drawbacks still exist. On the one hand, their syntax-driven nature makes them specific to a particular programming language and semantics. For example, the techniques developed for proving termination of *eager* functional programs and those for proving termination of *lazy* functional programs are different. Recently, a new semantics-based approach has emerged. Intuitively speaking, rather than inspecting the source code, this approach is based on constructing a finite representation of all the computations of a program—usually

☆ This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

E-mail address: gvidal@dsic.upv.es.

http://dx.doi.org/10.1016/j.scico.2015.01.007
0167-6423/

a graph—and, then, inspecting the transitions in this graph in order to analyze the termination of the program. Actually, it suffices to restrict the analysis to those transitions that belong to a loop of the graph, i.e., a (potential) loop of the program. The construction of the graph can be seen as a front-end that depends on the considered programming language and semantics. Once the graph is built, one can express its transitions in a common language (e.g., a rule-based formalism such as term rewriting or logic programming) and, then, apply the existing syntax-directed approaches. In this way, the back-end could be shared by the different termination provers. We call this approach *semantics-based* since the front-end (the construction of the graph) is driven by the *semantics* of the program rather than by its syntax.

In the literature, we can already find a number of approaches that mostly follow this semantics-based scheme, e.g., to prove the termination of Haskell [21], Prolog with impure features [23,42], narrowing [35,46], or Java bytecode [2,3,9,10,36,43]. While all these approaches have proven useful in practice, they are still tailored to the specific features of the considered programming language and semantics. Unfortunately, this makes it rather difficult to grasp the key ingredients of the approach and, thus, it is not easy to design a termination tool for a different programming language by following the same pattern.

In this work, we present instead a general, high-level framework that follows the semantics-based approach to proving termination. Our purpose in this paper is not to introduce a new, semantics-based termination prover but to present a language-independent formulation that uses well-known principles from symbolic execution and partial evaluation, so that the vast literature on these subjects can be reused. In particular, we focus on the first stage—constructing a finite representation of all program computations—and advocate the use of symbolic execution, together with appropriate subsumption and abstraction operators. Symbolic execution [26,11] is a well-known technique for program verification, testing, debugging, etc. In contrast to concrete execution, symbolic execution considers that the values of some input data are unknown, i.e., some input parameters $x, y, \ldots$ take *symbolic values* $X, Y, \ldots$. Because of this, symbolic execution is often non-deterministic: at some control statements, we need to follow more than one execution path because the available information does not suffice to determine the validity of a control expression, e.g., symbolic execution may follow both branches of the conditional "`if` $(x{>}0)$ `then` *exp*1 `else` *exp*2" when the symbolic value X of variable x is not constrained enough to imply neither $x{>}0$ nor $\neg(x{>}0)$. Symbolic states include a *path condition* that stores the current constraints on symbolic values, i.e., the conditions that must hold to reach a particular execution state. For instance, after symbolically executing the above conditional, the derived states for *exp*1 and *exp*2 would add the conditions $X{>}0$ and $X{\leq}0$, respectively, to their path conditions.

Traditionally, formal techniques based on symbolic execution have enforced soundness, i.e., the definition of *underapproximations*: if a symbolic state is reached and its path condition is satisfiable, there must be a concrete execution path that reaches the corresponding concrete state. These approaches, however, are often incomplete, i.e., there are some concrete computations that are not covered by the symbolic executions. In contrast, we consider a complete approach to symbolic execution, so that it *overapproximates* concrete execution. While underapproximations are useful for testing and debugging (since there are no false positives), overapproximations are important for verifying *liveness* properties such as program termination.

A preliminary version of some of the ideas in this paper appeared in [47]. Basically, [47] proposed a method for proving program termination by i) first producing a finite—but complete—symbolic execution of a program, ii) extracting a rewrite system that reproduces the transitions of symbolic execution and, finally, iii) using an off-the-shelf tool for proving the termination of the rewrite system (i.e., AProVE [22]). Here, in contrast to [47], we focus only on the front-end—constructing a symbolic execution graph—but introduce a more detailed approach. On the one hand, we consider a generalized notion of abstraction (w.r.t. [47]) which is much more useful in practice. Also, we introduce an algorithm for the construction of a finite representation of symbolic executions and define concrete subsumption and abstraction operators. Finally, we also prove the correctness of the resulting finite symbolic execution graphs (using the generic operations of subsumption and—generalized—abstraction), which guarantee their usefulness in the context of termination analysis.

The remainder of this paper is organized as follows. Section 2 introduces the notion of *complete* symbolic execution. We illustrate our developments using both a simple imperative language and a first-order eager functional language. Section 3 presents appropriate subsumption and abstraction operators so that a finite (but still complete) representation of the computation space can be obtained. Finally, Section 4 discusses some related work and Section 5 concludes.

2. Complete symbolic execution

In this section, we recall the notion of symbolic execution and introduce the conditions for completeness. We illustrate our developments with two simple programming languages.

Analogously to, e.g., [33], we abstract away from the syntax of a concrete programming language and represent a *program* P by a transition system denoted by a tuple $\langle \Sigma, \Theta, \mathcal{T}, \rho \rangle$ where Σ is a (possibly infinite) set of states, $\Theta \subseteq \Sigma$ are the initial states, $\mathcal{T}$ is a finite set of transitions (can be thought of as labels of program statements), and ρ is a mapping that assigns to each transition a binary relation over states: $\rho_\tau \subseteq \Sigma \times \Sigma$, for $\tau \in \mathcal{T}$. The *transition relation* of a program P is denoted by R_P and defined as follows:

$$R_P = \bigcup_{\tau \in \mathcal{T}} \rho_\tau$$