Trace-Based Abstract Interpretation of Operational Semantics

DAVID A. SCHMIDT
Kansas State University, Computing and Information Sciences Department, 234 Nichols Hall, Manhattan, KS 66506 USA

Abstract. We present trace-based abstract interpretation, a unification of several lines of research on applying Cousot-Cousot-style abstract interpretation (a.i.) to operational semantics definitions (such as flowchart, big-step, and small-step semantics) that express a program’s semantics as a concrete computation tree of trace paths. A program’s trace-based a.i. is also a computation tree whose nodes contain abstractions of state and whose paths simulate the paths in the program’s concrete computation tree. Using such computation trees, we provide a simple explanation of the central concept of collecting semantics, and we distinguish concrete from abstract collecting semantics and state-based from path-based collecting semantics. We also expose the relationship between collecting semantics extraction and results garnered from flow-analytic and model-checking-based analysis techniques. We adapt concepts from concurrency theory to formalize “safe” and “live” a.i.’s for computation trees; in particular, coinduction techniques help extend fundamental results to infinite computation trees.

Problems specific to the various operational semantics methodologies are discussed: Big-step semantics cannot express divergence, so we employ a mixture of induction and coinduction in response; small-step semantics generate sequences of program configurations unbounded in size, so we abstractly interpret source language syntax. Applications of trace-based a.i. to data-flow analysis, model checking, closure analysis, and concurrency theory are demonstrated.

Keywords: abstract interpretation, operational semantics, collecting semantics, simulation

1. Introduction

Abstract interpretation (a.i.) is accepted as the correctness foundation for data-flow analysis of flowchart programs [17, 18, 45], and related research has demonstrated that a.i. can be applied to nonflowchart programs defined by denotational semantics [2, 8, 21, 25, 44, 45, 55, 57–60] and structural operational semantics [19, 30, 68–71, 81]. Model checking is another important applications area [6, 10, 22, 23, 76, 77].

The theory of a.i. of denotational semantics definitions is mature, and we present an initial unification of several lines of research on the a.i. of operational semantics definitions. Common to all of flowchart (state-transition), big-step (natural), and small-step (SOS) operational semantics is the notion of computation tree: A program’s concrete semantics is a tree whose paths represent execution traces and whose nodes display the program’s changing states. Therefore, the program’s a.i. must also be a computation tree whose nodes contain abstractions of state and whose paths simulate the paths in the corresponding concrete computation tree. For this reason, we coin the term trace-based abstract interpretation.
to denote techniques for a.i. of operational semantics that build computation trees of traces.

Perhaps the primary objective of an a.i. is to calculate a program’s collecting semantics, and a major contribution of this paper is a simple definition of collecting semantics extraction from a computation tree. Also, we clarify the distinctions between concrete and abstract collecting semantics, between state-based and path-based collecting semantics, and we expose the relationship between collecting semantics extraction and flow-analytic-based and model-checking-based static analysis techniques.

We adapt concepts from concurrency theory to formalize “safe” and “live” a.i.’s for computation trees; in particular, coinduction techniques help extend fundamental results to infinite computation trees.

In addition, we address challenges that arise within the specific semantics forms: Big-step semantics cannot express divergence, so we employ a mixture of induction and coinduction; small-step semantics generate sequences of program configurations unbounded in size, so we abstractly interpret source language syntax in response. Applications of trace-based a.i. to data-flow analysis, model checking, closure analysis, and concurrency theory are demonstrated.

Many of the paper’s technical concepts are taken from the trailblazing research of Cousot and Cousot [16–21]—indeed, the first mention of a.i. of execution traces appears in P. Cousot’s doctoral thesis [16]. Our primary contribution is an expository one: We unify several lines of research in abstract interpretation and concurrency theory to achieve a simple methodology for a.i. upon operational semantics.

The structure of the paper goes as follows: Basic concepts appear in Section 1.1; Section 2 applies the concepts to a thorough redevelopment of abstract interpretation of flowchart semantics. Section 3 presents key aspects of definition and proof by coinduction. Section 4 surveys briefly liveness abstract interpretations, and Sections 5 and 6 present traced-based a.i. for big-step semantics and small-step semantics, respectively, reaffirming the methodology’s utility and addressing problems specific to these formats. Applications are intertwined with the semantic forms upon which they are based. Section 7 concludes.

1.1. What is trace-based abstract interpretation?

An operational semantics that defines run-time executions is called a concrete semantics or concrete interpretation (c.i.). Using the concrete semantics, a program plus its run-time input data can be executed; the result is a concrete computation tree, or concrete tree, for short. The tree’s paths form an “execution trace,” which displays at the tree’s nodes the changing state of the computation; if the computation is divergent, an infinite tree results.

If the run-time data is abstracted to a set of “tokens” that represent properties of the data and if the semantics is revised to compute upon such tokens, one obtains an abstract semantics or abstract interpretation (a.i.). A program plus an input data token can be interpreted by the abstract semantics; the result is again a tree, called the abstract computation tree, or abstract tree, for short.

It is convenient to think of an a.i. as a “symbolic execution” where the symbols have semantic content. One example is the implementation of type inference by an a.i. where