Through the Lens of Abstraction

Thomas Reps[†,‡] and Aditya Thakur[†]
[†]Computer Sciences Department, University of Wisconsin and [‡]GrammaTech, Inc.

This talk deals with the use of *abstraction* in two areas of automated reasoning: verification of programs, and decision procedures for logics.

Establishing that a program is correct is undecidable in general. Program-analysis and verification tools sidestep this tar-pit of undecidability by working on an abstraction of a program, which over-approximates the original program's behavior. The theory underlying this approach is called *abstract interpretation* [1]. Abstract interpretation provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Instead, it explores the program's behavior for *all* possible inputs, thereby accounting for all possible states that the program can reach.

Operationally, one can think of abstract interpretation as running the program "in the aggregate". That is, rather than executing the program on ordinary states, the program is executed on *abstract states*, which are finite-sized descriptors that represent collections of states. For example, one can use abstract states that represent only the *sign* of a variable's value: `neg`, `zero`, `pos`, or `unknown`. If the abstract state is $[a \mapsto \texttt{neg}, b \mapsto \texttt{neg}]$), the product "$a{*}b$" would be performed as "`neg*neg`", yielding `pos`. This approximation discards information about the specific *values* of a and b: $[a \mapsto \texttt{neg}, b \mapsto \texttt{neg}]$ represents all states in which a and b hold negative integers.

However, there is a glitch: abstract interpretation has a well-deserved reputation of being a kind of "black art", and consequently difficult to work with.

The first part of this talk will describe a fifteen-year quest to raise the level of automation in abstract interpretation, by presenting three different approaches to creating correct-by-construction analyzers:

1. The TVLA system [4] introduced a way to create abstractions of systems specified in first-order logic. Different analyses are defined using TVLA by varying the relation symbols of the logic, and, in particular, by varying which of the unary relations control how nodes are folded together. The specified set of relations determines the set of properties that will be tracked by the analyzer.
2. The TSL system [2] provides a framework for creating correct-by-construction implementations of the state-transformation functions needed in tools that analyze machine code. From a single specification of the concrete semantics of a machine-code instruction set, TSL automatically generates state-transformation functions needed for static analysis, dynamic analysis, symbolic analysis, or any combination of the three.
3. Our recent work on symbolic methods for abstract interpretation [6] aims to bridge the gap between (i) the use of logic for specifying program semantics and program correctness, and (ii) abstract interpretation. Many of the issues can be reduced to the problem of *symbolic abstraction*:

   Given a formula φ in some logic $\mathcal{L}$, and an abstract domain $\mathcal{A}$, find the most-precise descriptor a in $\mathcal{A}$ that over-approximates the meaning of φ.

The second part of the talk describes the use of abstraction in the design of decision procedures for logics. We start by explaining Stålmarck's method, a decision procedure for propositional logic, using abstract-interpretation terminology [7]. In particular, we show how Stålmarck's method is an instantiation of a generic framework parameterized by an abstract domain over Booleans. Furthermore, different instantiations of the framework lead to new decision procedures for propositional logic. Furthermore, this abstraction-based view allowed us to lift Stålmarck's method from propositional logic to richer logics: to obtain a method for richer logics, instantiate the parameterized version of Stålmarck's method with richer abstract domains [8]. We call such a decision-procedure design, which is parameterized by an abstract domain, a *Satisfiability Modulo Abstraction (SMA)* solver.

The talk will conclude by describing an SMA solver for separation logic [5]. Separation logic (SL) [3] is an expressive logic for reasoning about heap structures in programs, and provides a mechanism for concisely describing program states by explicitly localizing facts that hold in separate regions of the heap. SL is undecidable in general, but by using an abstract domain of shapes [4] we were able to design a semi-decision procedure for SL.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[2] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.

[3] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Symp. on Logic in Comp. Sci.*, 2002.

[4] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[5] A. Thakur, J. Breck, and T. Reps. Satisfiability modulo abstraction for separation logic with linked lists. TR 1800, CS Dept., Univ. of Wisconsin, Madison, WI, 2014.

[6] A. Thakur, M. Elder, and T. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, 2012.

[7] A. Thakur and T. Reps. A Generalization of Stålmarck's Method. In *SAS*, 2012.

[8] A. Thakur and T. Reps. A method for symbolic computation of abstract operations. In *CAV*, 2012.