Abstract

One can prove that a program satisfies a correctness property in different ways. The deductive approach uses logic and is automated using decision procedures and proof assistants. The automata-theoretic approach reduces questions about programs to algorithmic questions about automata. In the abstract interpretation approach, programs and their properties are expressed in terms of fixed points in lattices and reasoning uses fixed point approximation techniques. We describe a research programme to establish precise, mathematical correspondences between these approaches and to develop new analyzers using these results. The theoretical tools we use are the theorems of Büchi that relate automata and logic and a construction of Lindenbaum and Tarski for generating lattices from logics. This research has lead to improvements in existing tools and we anticipate further theoretical and practical consequences.

1 Introduction

The problem of determining if a program does what it is supposed to do dates back to the origins of computer science. Goldstine and von Neumann [1947] included assertion boxes in their language for the IAS machine and stated that a programmer should guarantee that assertions were not violated. At a meeting in Cambridge, Turing [1949] presented techniques for reasoning about program correctness and termination. Their work has been rediscovered and significantly extended by program verification research [Knuth, 2003]. We briefly recall three approaches for reasoning about programs and describe our efforts to relate them mathematically and combine them algorithmically.

In satisfiability-based approaches, bounded executions of a program $P$ are encoded as a formula $\text{Exec}(P)$ and assertion violations are encoded as a formula $\text{Err}$. The theorem below states that no bounded execution of $P$ violates the assertion.

$$\vdash \text{Exec}(P) \implies \neg \text{Err}$$

Solvers for satisfiability of formulae in a theory prove such theorem by showing that $\text{Exec}(P) \land \text{Err}$ is unsatisfiable [Björner and de Moura, 2014].

Rather than viewing correctness as a theorem, in model checking, one checks if $P$ is a model of the formula $\neg \text{Err}$ [Clarke et al., 1999; Baier and Katoen, 2008].

$$P \models \neg \text{Err}$$

In the automata-theoretic approach to model checking [Vardi and Wolper, 1994], the executions of $P$ are viewed as words accepted by an automaton $A_P$, and erroneous executions are viewed as words accepted by an automaton $A_{\text{Err}}$. The program $P$ contains no assertion violation exactly if the language of the product automaton is empty.

$$\mathcal{L}(A_P \times A_{\text{Err}}) = \emptyset$$

One appeal of this approach is that it reduces questions about complex structures such as temporal properties and programs to language inclusion. Moreover, automata are labelled, directed graphs, so the model checking problem becomes one amenable to graph algorithms.

The lattice-theoretic approach to reasoning about programs has its origins in programming language semantics and compiler construction. Scott [1971] defined the meaning of a program, denoted $\llbracket P \rrbracket$, as a fixed point of a function on a lattice. The abstract interpretation framework of Cousot and Cousot [1977], extended early work in compiler construction, by showing how to interpret $P$ and $\text{Err}$ in a lattice $A$ of approximations. The program is error-free if the lattice element is separate from the lattice element denoted by the error.

$$\llbracket P \rrbracket_A \cap \llbracket \text{Err} \rrbracket_A \subseteq \bot$$

Algorithms for approximation of fixed points are used to determine if the order above holds.

There are currently both academic and commercial tools based on these techniques. These tools differ in their degree of automation, the programs they can reason about, and their performance. These differences have lead to research to combine these techniques. We describe here the initial steps of a programme that seeks to facilitate exchange of techniques between these approaches by establishing mathematical translations between them.

Our main observation, illustrated in Figure 1, is that by applying classic theorems in logic, automata theory and lattice theory, one can translate between the mathematical structures used in these three approaches. A theorem of Büchi [1960] shows that a word is accepted by a finite automaton exactly
if that word is a model of a formula in the weak, monadic, second-order theory of the successor function (WS1S). That is, a regular language, which is an element of the lattice of languages, can be defined as the language \( L(A) \) of an automaton or the models \( \text{mod}(\varphi) \) of a formula. We show that by extending Büchi’s construction, we can encode the executions of a CFG as the models of formulae in WS1S(T), an extension of WS1S that can represent program variables.

To relate logic and lattices, we use the Lindenbaum-Tarski construction [Rasiowa and Sikorski, 1963], which was originally used to relate the propositional calculus and Boolean algebras. Our observation is that a logic \( L \) characterizes the lattice \( A \) used in an abstract interpreter if the Lindenbaum-Tarski algebra of \( L \) is isomorphic to \( A \). The problem of giving a logical characterization of the lattice in an abstract interpreter amounts to inverting the Lindenbaum-Tarski construction.

## 2 Programs to Second-Order Logics

Program verification algorithms, like compiler optimizations, are often not formulated in terms of programs but rather in terms of control flow graphs (CFGs). We now describe a connection between CFGs and second-order logic, which follows from a simple extension of Büchi’s theorem. We use a condensed, non-standard, representation of CFGs, with labels on edges, to emphasise the similarity to automata.

Figure 2 contains the CFG for a program that initializes a variable \( x \) to 0 and increments \( x \) by 2 as long as \( x \) is at most 9. The property we are interested in is encoded as the assertion that \( x \) is 10 after the program executes. The CFG has locations for the entry of the program (\( \text{in} \)), the loop head (\( \text{hd} \)), the loop body (\( \text{bd} \)), the loop exit (\( \text{ex} \)), an error (\( \text{er} \)), reached if the assertion is violated, and the safe location (\( \text{sf} \)), reached if the assertion holds.

A CFG can be viewed as an automaton in which the states correspond to locations in code and transitions are labelled with statements that are executed when moving from one location to the other. The initial location is an initial state. There are several choices for the final location. For this example we consider the error location as the final state, so that we reason about the assertion violation in terms of reachability of the final state.

An execution of a program is a path through a CFG and every execution corresponds to a word accepted by the automaton. The converse is not true: not every path from the initial to the final state corresponds to an execution. The sequence of locations and labels

\[
\text{in}, x := 0, \text{hd}, [x > 9], \text{ex}, [x \neq 10], \text{er}
\]

is a path in the automaton but it does not define an execution because the condition \([x > 9]\) is not satisfied after the assignment \( x := 0 \). A path only defines an execution if it is possible to execute all statements on that path in sequence. The program respects the assertion if the CFG, viewed as an automaton, accepts a word that defines an execution.

We now describe how, by extending Büchi’s theorem, the question of existence of a feasible path can be viewed as that of satisfiability of a formula. First, we describe the structure of satisfiability of a formula. We show that

\[
\forall i. \text{First}(i) \Rightarrow X_{\text{in}}(i) \land \forall i. \forall j. X_{\text{bd}}(j) \land \text{Suc}(i, j) \Rightarrow ((\text{Suc}(x) = 0)(i) \land X_{\text{in}}(i)) \lor ((\text{Suc}(x) = x + 2)(i) \land X_{\text{bd}}(i))
\]

\[
\land \forall i. \forall j. X_{\text{ex}}(j) \land \text{Suc}(i, j) \Rightarrow (x \leq 9 \Rightarrow \text{Suc}(x) = x)(i) \land X_{\text{bd}}(i)
\]

\[
\land \forall i. \forall j. X_{\text{err}}(j) \land \text{Suc}(i, j) \Rightarrow (x = 10 \Rightarrow \text{Suc}(x) = x)(i) \land X_{\text{ex}}(i)
\]

\[
\land \forall i. \text{Last}(i) \Rightarrow X_{\text{ex}}(i)
\]

Model of this formula correspond to executions of the CFG.

Figure 3: A formula in the monadic, second order theory of one successor extended with a first-order theory of arithmetic. Model of this formula correspond to executions of the CFG. 

\[
\text{exec} (i) = \left\{ \begin{array}{ll}
[x > 9] & 
\text{if } x \geq 10, \\
[x = 10] & 
\text{if } x = 10, \\
[x < 10] & 
\text{if } x < 10.
\end{array} \right.
\]
A fixed point of this function is a set $X$ satisfying that $F(X) = X$. The set of values of $x$ satisfying the formula $\varphi$ above is a fixed point and in fact, the least one. In general, the strongest invariant of a loop is not computable. In abstract interpretation, one reasons about fixed points of a different function over a different lattice. Figure 4 depicts the lattice $(\text{Itv}, \sqcup)$ of integer intervals. An interval is a pair $[a, b]$, where $a \leq b$ and $a$ and $b$ are in $\mathbb{Z} \cup \{-\infty, +\infty\}$. Every set of integers is abstracted by the unique, smallest interval that contains it: for instance $[1, 3]$ is the smallest interval containing $\{1, 3\}$. The interval abstracts $\{1, 3\}$ because one loses the information that 2 is not in the set.

The function $G$, below, lifts the loop to intervals.

$$G([a, b]) = [0, 0] \cup ([a, b] \cap [-\infty, 9]) \oplus [2, 2]$$

This function states that the values of $x$ are initially in the interval $[0, 0]$ and in each iteration, the sub-interval of $[a, b]$ below 9 is incremented by 2. By associating such a function with each edge in the CFG, one obtains a system of equations that can be solved to obtain a fixed point.

Figure 5 demonstrates an interval analysis of the loop. Each column contains the interval associated with each program location at each iteration. Initially, the value of $x$ is arbitrary at $\text{in}$, while $\text{ex}$, $\text{er}$, and $\text{sf}$ are considered unreachable. The final column contains bounds on $x$ computed by the analysis. The interval at $\text{hd}$ is $[0, 10]$, which is a loop invariant but not the strongest one. The naive iteration shown here is inefficient and may not terminate, and numerous improvements are used in practice.
Figure 6: A subset of the proof rules of the interval logic.

**Interval logic.** The sets of values definable by the intervals correspond to models of formulae in the logic below.

\[ \varphi := x \geq m \mid x \leq n \mid \varphi \land \varphi \]

For example the interval \( x: [-\infty, 9] \) corresponds to \( x \leq 9 \), \( x: [3, 9] \) to \( x \geq 3 \land x \leq 9 \), while the logical constants \( \top \) and \( \perp \) correspond to the maximal interval \( [-\infty, \infty] \) and the empty interval \( \perp \), respectively. Some proof rules for reasoning about interval formulae are shown in Figure 6.

The proof system is meant to capture the reasoning encoded in the lattice. For example, the meet operation of the lattice satisfies the identity \( [-\infty, 3] \cap [5, \infty] = \perp \), which we can derive, logically, by applying a proof rule.

\[ \frac{[3 < 5]}{x \leq 3 \land x \leq 5 \vdash \perp} \quad \text{frf} \]

Another example is \( [-\infty, 3] \cap [-\infty, 9] = [-\infty, 3] \), which we can view as two inequations.

\[ [-\infty, 3] \cap [-\infty, 9] \subseteq [-\infty, 3] \]
\[ [-\infty, 3] \subseteq [-\infty, 3] \cap [-\infty, 9] \]

These two inequations correspond to the two proofs shown below, which use standard sequent calculus rules in addition to the interval rules.

\[ \frac{x \leq 3 \vdash x \leq 3}{x \leq 3 \land x \leq 9 \vdash x \leq 3} \quad \text{L}_1 \]
\[ \frac{x \leq 3 \vdash x \leq 3}{x \leq 9 \vdash x \leq 9} \quad \text{UB-L} \]
\[ \frac{x \leq 3 \vdash x \leq 3}{x \leq 3 \land x \leq 9 \vdash x \leq 9} \quad \text{L}_R \]
\[ \frac{x \leq 3 \vdash x \leq 3}{x \leq 3 \land x \leq 9 \vdash x \leq 9} \quad \text{CL} \]

A major question that now remains is whether we can rigorously argue that the logic we have provided captures the interval lattice. A closely related question was addressed in the early days of logic, by Tarski, extending a construction of Lindenbaum. Two formulae in a logic \( \mathcal{L} \) are considered equivalent if they are interderivable in the proof system of \( \mathcal{L} \). The order relation below holds between two equivalence classes if some formula in the second is derivable from some formula in the first. A meet operation can be defined by lifting conjunction from formulae to equivalence classes.

\[ \varphi \equiv_{\mathcal{L}} \psi \text{ if } \varphi \vdash \psi \text{ and } \psi \vdash \varphi. \]

[\( [\varphi]_{\mathcal{L}} \leq [\psi]_{\mathcal{L}} \) if \( \theta_1 \vdash \theta_2 \) for some \( \theta_1 \in [\varphi]_{\mathcal{L}}, \theta_2 \in [\psi]_{\mathcal{L}} \).]

[\( [\varphi]_{\mathcal{L}} \land [\psi]_{\mathcal{L}} \equiv [\theta_1 \land \theta_2]_{\mathcal{L}} \) for \( \theta_1 \in [\varphi]_{\mathcal{L}}, \theta_2 \in [\psi]_{\mathcal{L}} \).]

The Lindenbaum-Tarski construction described above defines a lattice only for logics in which \( \equiv_{\mathcal{L}} \) is a congruence with respect to logical operations. Such logics are algebraizable [Rasiowa and Sikorski, 1963]. We have shown that applying this construction to the interval logic yields a lattice isomorphic to the intervals.

By providing an explicit logic that characterizes the interval lattice, we have made precise the intuition about the reasoning capabilities of that lattice. This characterization allows for calculations that are performed during interval analysis to be viewed as deductions in a proof system. Other features of the lattice highlighted by this logical treatment are that the atomic predicates are one-way infinite intervals, which correspond to meet-irreducibles: lattice elements that are not derivable from other, distinct lattice elements by meet operations. In [D’Silva and Urban, 2015a], we further show how the operation of an abstract interpreter, as illustrated in Figure 5, can be viewed as deduction in a satisfiability solver.

4 Discussion and Conclusion

There are currently multiple techniques for reasoning about programs. These techniques appear fundamentally different in the mathematical foundations they use. This difference impedes our ability to combine the strengths of the techniques, both in theory and practice.

We have shown that by using classic results in logic, lattice theory and automata theory, one can identify new relationships between the automata-theoretic, deductive and lattice-theoretic approaches to program verification. Though the work described here is a first step in a longer research effort, it has already led to improvements in a tool for termination analysis [D’Silva and Urban, 2015b].

There are several immediate extensions that will deepen our understanding of the relationships we have identified. We have focused on languages of finite words and the connection to reachability analysis. To model termination, procedure calls and concurrency, one has to consider the analogues of Büchi’s theorem for Büchi automata, nested word automata and asynchronous automata. We have restricted our study of abstract interpreters to lattices. We believe that functions in an abstract interpreter correspond to first-order modalities, and our proofs have to be extended using the Lindenbaum-Tarski construction for modal logics.

Our work has opened the door to a proof-theoretic interpretation and investigation of lattice-based analyzers. Questions concerning cut elimination, proof normalization, and lower bounds on proof size, which would not have made sense in the context before our work are now waiting to be answered. We believe that answering these and other questions will deepen our theoretical understanding of different approaches to reasoning about programs and also extend the boundary of what can be successfully automated in practice.

References


