Büchi, Lindenbaum, Tarski: A Program Analysis Appetizer

Vijay D'Silva
Google Inc., San Francisco

Caterina Urban
ETH Zürich, Zurich

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 $Exec(P)$ and assertion violations are encoded as a formula Err. The theorem below states that no bounded execution of P violates the assertion.

$$\vdash Exec(P) \implies \neg Err$$

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

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

$$P \models \neg Err$$

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

$$\mathcal{L}(\mathcal{A}_P \times \mathcal{A}_{Err}) = \emptyset$$

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 $[\![P]\!]$, 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 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.

$$[\![P]\!]_{\mathsf{A}} \sqcap [\![Err]\!]_{\mathsf{A}} \sqsubseteq \bot$$

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

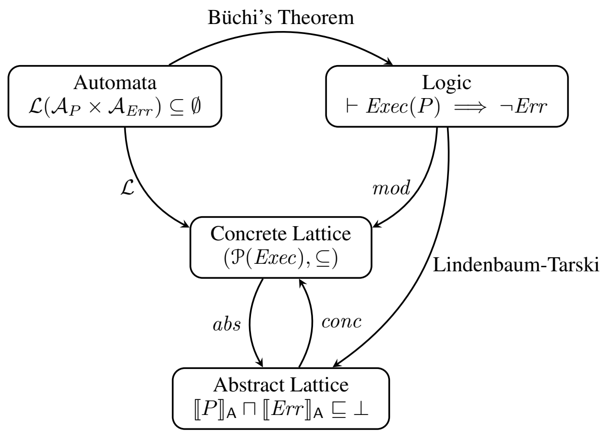

Figure 1: One may use deductive, automata theoretic, or lattice-theoretic techniques to prove a property of a program. Büchi's theorem allows us to view automata as formulae, and by extending it, we view control-flow graphs as formulae. The Lindenbaum-Tarski construction allows for generating a lattice from a logic and by inverting it, we identify logics and proof systems corresponding to the lattice-theoretic approach.

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 $\mathcal{L}(\mathcal{A})$ of an automaton or the models $mod(\varphi)$ 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 $\mathcal{L}$ characterizes the lattice A used in an abstract interpreter if the Lindenbaum-Tarski algebra of $\mathcal{L}$ 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 (`in`), the loop head (`hd`), the loop body (`bd`), the loop exit (`ex`), an error (`er`), reached if the assertion is violated, and the safe location (`sf`), reached if the assertion holds.

A CFG can be viewed as an automaton in which the states

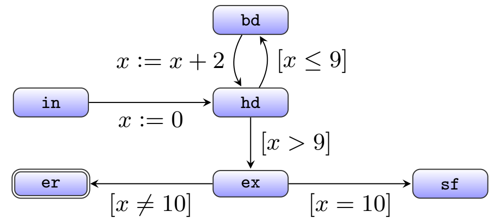

Figure 2: A CFG for a program with an assertion.

$$
\begin{aligned}
&\forall i.First(i) \Rightarrow X_{\texttt{in}}(i) \wedge \forall i.\forall j.X_{\texttt{hd}}(j) \wedge Suc(i,j) \Rightarrow \\
&((suc(x) = 0)(i) \wedge X_{\texttt{in}}(i)) \vee ((suc(x) = x + 2)(i) \wedge X_{\texttt{bd}}(i)) \\
&\wedge \forall i.\forall j.X_{\texttt{bd}}(j) \wedge Suc(i,j) \Rightarrow (x \leq 9 \Rightarrow suc(x) = x)(i) \wedge X_{\texttt{hd}}(i)) \\
&\wedge \forall i.\forall j.X_{\texttt{ex}}(j) \wedge Suc(i,j) \Rightarrow (x > 9 \Rightarrow suc(x) = x)(i) \wedge X_{\texttt{hd}}(i) \\
&\wedge \forall i.\forall j.X_{\texttt{er}}(j) \wedge Suc(i,j) \Rightarrow (x \neq 10 \Rightarrow suc(x) = x)(i) \wedge X_{\texttt{ex}}(i) \\
&\wedge \forall i.\forall j.X_{\texttt{sf}}(j) \wedge Suc(i,j) \Rightarrow (x = 10 \Rightarrow suc(x) = x)(i) \wedge X_{\texttt{ex}}(i) \\
&\wedge \forall i.Last(i) \Rightarrow X_{\texttt{ex}}(i)
\end{aligned}
$$

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.

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

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

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 executions.

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 structures over which such formulae are interpreted. Intuitively, a structure consists of a trace, which encodes the data flow of a single execution, and a position assignment, which encodes control flow. Formally, a *state* $s : Var \to Val$ associates program variables with values and a *trace* $\tau = s_0, s_1, \ldots, s_{n-1}$ is a sequence of states. A position assignment σ maps second-order variables to finite subsets of the natural numbers. A structure (τ, σ) consists of a trace of length n and a position assignment that only uses finite subsets of $[0, n-1]$.

The formula corresponding to our running example is shown in Figure 3. The formula contains second-order variables, such as $X_{\texttt{in}}$ or $X_{\texttt{ex}}$, for CFG locations. The variables are *monadic* meaning they are interpreted as one-place predicates. The predicate $X_{\texttt{ex}}(i)$ is interpreted as being true at position i of an execution if the program is in location `ex` after i steps. The predicate $Suc(i, j)$ is true if there is a transition from position i to position j in the trace. The condition

$$X_{\texttt{hd}}(j) \wedge Suc(i, j) \Rightarrow X_{\texttt{in}}(i)$$

expresses that the location `hd` is reached from the location `in`. The condition

$$X_{\texttt{hd}}(j) \wedge Suc(i, j) \Rightarrow (suc(x) = 0) \wedge X_{\texttt{in}}(i)$$

additionally states that the program variable x is 0 after this transition. The conjuncts in Figure 3 logically encode the structure of the CFG and the constraints imposed by its labels.

The logic in which this formula is expressed is called the *weak, monadic, second-order theory of one successor, modulo a theory*, abbreviated to WS1S(T), which extends Büchi's logic WS1S. The logic is weak because second-order variables are interpreted over finite (rather than infinite) sets of positions. The theory T enables reasoning about program data.

Theorem 1 *A location* `loc` *is reachable in a* CFG G *exactly if the formula* $Reach_{G,\texttt{loc}}$ *is satisfiable.*

The formula $Reach_{G,\texttt{loc}}$, defined by D'Silva and Urban [2015a], has similar structure to the one in Figure 3. Theorem 1 reduces the problem of finding an execution that violates an assertion to a satisfiability problem. It is important to recognize that Büchi's original theorem established a deeper connection between automata and logic than what our theorem does for CFGs. Büchi gave an algorithm for constructing a finite automaton from a WS1S formula. It followed, from the decidability of language inclusion for finite automata, that the satisfiability problem for WS1S was decidable. Our translation only goes from CFGs to formulae and not in the other direction. In fact, like reachability in programs, the satisfiability problem for WS1S(T) is not decidable.

3 Abstract Interpretation and the Lindenbaum-Tarski Construction

Loop invariants are fundamental to reasoning about programs. The loop invariant below enables proving that the assertion is not violated in Figure 2.

$$\varphi \mathrel{\hat{=}} x \geq 0 \wedge x \leq 10 \wedge x \equiv 0 \ (\mathrm{mod} 2)$$

A central tenet of lattice-theoretic program analysis is that loop invariants are fixed points of functions on lattices [Cousot and Cousot, 1977; Clarke, 1977]. We now recall elements of abstract interpretation and our initial results in identifying logics corresponding to lattices.

If the variable x in Figure 2 is an integer, the set of possible values of x is an element of the lattice $(\mathcal{P}(\mathbb{Z}), \subseteq)$ of subsets of integers. The function below models the loop.

$$F(X) = \{0\} \cup \{x + 2 \mid x \in X, x \leq 9\}$$

A fixed point of this function is a set X satisfying that $F(X) = X$. The set of values of x satisfying the formula φ 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

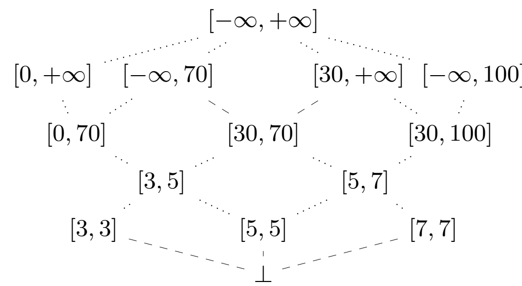

Figure 4: The lattice of intervals.

	1st iteration	2nd	$\cdots$	6th/fixed point
`in`	$x{:}\top$	$x{:}\top$	$\cdots$	$x{:}\top$
`hd`	$x{:}[0, 0]$	$x{:}[0, 2]$	$\cdots$	$x{:}[0, 10]$
`bd`	$x{:}[2, 2]$	$x{:}[2, 4]$	$\cdots$	$x{:}[2, 10]$
`ex`	$x{:}\perp$	$x{:}\perp$	$\cdots$	$x{:}[10, 10]$
`er`	$x{:}\perp$	$x{:}\perp$	$\cdots$	$x{:}\perp$
`sf`	$x{:}\perp$	$x{:}\perp$	$\cdots$	$x{:}[10, 10]$

Figure 5: A fixed point computation in the lattice of intervals.

depicts the lattice $(\mathsf{Itv}, \sqsubseteq)$ 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.

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

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 `in`, while `ex`, `er` and `sf` are considered unreachable. The final column contains bounds on x computed by the analysis. The interval at `hd` is $[0, 10]$, which is a loop invariant but not the strongest one. The naïve iteration shown here is inefficient and may not terminate, and numerous improvements are used in practice.

It is folk wisdom in the abstract interpretation community that lattices such as the intervals can be viewed as logics that are closed under conjunction but not disjunction or negation. The intuition behind this wisdom is that the meet operation on intervals $[0, 5] \sqcap [3, 7] = [3, 5]$ is the intersection of the values in these intervals. However, the join, such as $[0, 3] \sqcup [5, 7] = [0, 7]$, may contain more values than the union. We now describe how this intuition can be made precise.

$$\frac{\Gamma, x \leq n \vdash \varphi}{\Gamma, x \leq m \vdash \varphi}\ \text{UB-L}\quad [m \leq n]$$

$$\frac{\Gamma \vdash x \leq m}{\Gamma \vdash x \leq n}\ \text{UB-R}\quad [m \leq n]$$

$$\frac{\Gamma, x \geq m \vdash \varphi}{\Gamma, x \geq n \vdash \varphi}\ \text{LB-L}\quad [m \leq n]$$

$$\frac{\Gamma \vdash x \geq n}{\Gamma \vdash x \geq m}\ \text{LB-R}\quad [m \leq n]$$

$$\frac{}{\Gamma, x \leq m \wedge x \geq n \vdash \mathsf{ff}}\ \mathsf{ff}\text{R}\quad [m < n]$$

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.

$$\varphi ::= x \geq m \mid x \leq n \mid \varphi \wedge \varphi$$

For example the interval $x{:}[-\infty, 9]$ corresponds to $x \leq 9$, $x{:}[3, 9]$ to $x \geq 3 \wedge x \leq 9$, while the logical constants tt and ff 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 $[-\infty, 3] \sqcap [5, \infty] = \perp$, which we can derive, logically, by applying a proof rule.

$$\frac{}{x \leq 3 \wedge x \geq 5 \vdash \mathsf{ff}}\ \mathsf{ff}\text{R}\quad [3 < 5]$$

Another example is $[-\infty, 3] \sqcap [-\infty, 9] = [-\infty, 3]$, which we can view as two inequalities.

$$[-\infty, 3] \sqcap [-\infty, 9] \sqsubseteq [-\infty, 3]$$
$$[-\infty, 3] \sqsubseteq [-\infty, 3] \sqcap [-\infty, 9]$$

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

$$\frac{\dfrac{}{x \leq 3 \vdash x \leq 3}\ \text{I}}{x \leq 3 \wedge x \leq 9 \vdash x \leq 3}\ \wedge\text{L}_1$$

$$\frac{\dfrac{\dfrac{}{x \leq 3 \vdash x \leq 3}\ \text{I} \qquad \dfrac{\dfrac{}{x \leq 9 \vdash x \leq 9}\ \text{I}}{x \leq 3 \vdash x \leq 9}\ \text{UB-L}}{x \leq 3, x \leq 3 \vdash x \leq 3 \wedge x \leq 9}\ \wedge\text{R}}{x \leq 3 \vdash x \leq 3 \wedge x \leq 9}\ \text{CL}$$

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.

$\varphi \equiv_{\mathcal{L}} \psi$ if $\varphi \vdash \psi$ and $\psi \vdash \varphi$.

$[\varphi]_{\mathcal{L}} \preccurlyeq [\psi]_{\mathcal{L}}$ if $\theta_1 \vdash \theta_2$ for some $\theta_1 \in [\varphi]_{\mathcal{L}}, \theta_2 \in [\psi]_{\mathcal{L}}$.

$[\varphi]_{\mathcal{L}} \curlywedge [\psi]_{\mathcal{L}} \mathrel{\hat{=}} [\theta_1 \wedge \theta_2]_{\mathcal{L}}$ 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 lead 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

[Baier and Katoen, 2008] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[Bjørner and de Moura, 2014] N. Bjørner and L. de Moura. Applications of SMT solvers to program verification. In *Notes for the Summer School on Formal Techniques*, 2014.

[Büchi, 1960] J. Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford Univ. Press, 1960.

[Clarke *et al.*, 1999] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[Clarke, 1977] Edmund M. Clarke. Program Invariants as Fixed Points. *Foundations of Computer Science*, pages 18–29, 1977.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[D'Silva and Urban, 2015a] Vijay D'Silva and Caterina Urban. Abstract Interpretation as Automated Deduction. In *Proc. of Automated Deduction*, pages 450–464. Springer, 2015.

[D'Silva and Urban, 2015b] Vijay D'Silva and Caterina Urban. Conflict-Driven Conditional Termination. In *Proc. of Computer Aided Verification*, pages 271–286. Springer, 2015.

[Goldstine and von Neumann, 1947] Herman H. Goldstine and John von Neumann. Planning and Coding of Problems for an Electronic Computing Instrument. Technical report, Institute for Advanced Study, 1947.

[Knuth, 2003] Donald E. Knuth. Robert w floyd, in memoriam. *SIGACT News*, 34(4):3–13, December 2003.

[Rasiowa and Sikorski, 1963] H. Rasiowa and R. Sikorski. *The Mathematics of Metamathematics*. Polish Academy of Science, Warsaw, 1963.

[Scott, 1971] Dana S. Scott. The Lattice of Flow Diagrams. *Semantics of Algorithmic Languages*, 188:311–366, 1971.

[Turing, 1949] Alan Turing. Checking a Large Routine. *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[Vardi and Wolper, 1994] Moshe Y. Vardi and Pierre Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.