Logic in program analysis and verification[*]

Patrick Cousot

For program specification and verification, logic is a natural choice. However, for static analysis by abstract interpretation [10, 11], logic is rarely used, even as a user interface. We discuss the weaknesses of logic from this perspective.

1 Which logic for specification?

Choosing a decidable logic, such as Presburger arithmetic [20], to express program properties has the great advantage that user specified invariants can be mechanically checked. However, one can write a program that computes the multiplication `*` using iteration and addition `+` but the invariant is not expressible in Presburger arithmetic. Of course one can consider a richer specification logic by adding $*$ to the logic, but then iteration of `*` yields exponentiation `**` which is not in the logic, *etc.* The common choice of first-order logic is limited by the lack of a recursion mechanism necessary to reason on programs [23]. Transitive closure is not expressible [19]. The next generalization is to use Datalog [22]. But by choosing more and more expressive logics for specification language, one looses the decidability and completeness of provers, has to resort to user-interaction, which is generally excluded in automatic static program analysis tools.

Finally, to discuss the soundness of static analyzers, second-order logic is required. For example [1], a Hoare triple, $\{P\}C\{Q\}$ must satisfy $\forall P, Q \in \wp(S) \,.\, \forall x \in P \,.\, \forall y \in S \,.\, x \xrightarrow{C} y \Rightarrow y \in Q$ where S is a set of states and $\xrightarrow{C}$ the natural big-step semantics of command C. So called hyperproperties [3] would require $P \in \wp(\wp(S))$.

2 Which logic for property representation in a static analyzer?

Once a logic has been chosen for specification it, or a more expressive one, can be used as an internal representation of abstract properties.

An anonymous referee mentioned, with good reason, that "it is often convenient to use logic for describing, for instance how the domain of positive boolean functions can be used for analysis of the variable dependencies that arise within logic programs. By using global analysis via abstract interpretation a Prolog system like Ciao can reason with much richer information than, for example, traditional types as well as procedure-level properties such as determinacy, termination and non-failure. It is beneficial in this case that abstract interpretation algorithms are handled within the underlying logic programming preprocessor or and compiler" [18].

[*]Position paper for LPOP 2020.

The great advantage of logical representations of program properties is obviously uniformity. Any program property can be represented by (the abstract syntax) of a formula in the logic. For example, the simple connectives ($\vee$, $\wedge$, $\neg$, *etc.*) are very simple tree manipulations.

However, uniformity has the disadvantage that many algorithms manipulating these properties are based on specific representations. For example, linear equalities or inequalities will use matrices to represent a set of constraints and a frame of higher-dimensional polyhedra [16]. The efficiency of the algorithms is highly dependant on an adequate representation of the properties. For example the disjunction $\vee$ becomes an elaborated convex-hull algorithm. So, algorithmically, syntax-based representation uniformity is not tenable.

The concept of reduced product in abstract interpretation [11] was developed for expressing a conjunction of properties with non-uniform representations.

This problem was solved in the same way by SMT solvers, initially with Nelson-Oppen reduction [25, 13] and later with more general reductions when incorporating incomplete theories. Because the community has developed common interfaces for SMT solver competitions, such as SMTLIB2 [2], why not adopt them for specification? The obvious reason is that these formats were designed for machine interfaces, not memory representation, and are unreadable by the casual programmer.

3 Abstract domains

The algebraic concept of abstract domain in abstract interpretation and their combinations [11] were designed to handle non-uniform representations of program properties. Translating the concept in logic is difficult because the abstract properties have to be expressed in a concrete logic. If it is easy to express that x is in a given interval by $a \leqslant x \wedge x \leqslant b$, it is harder to express the concept of "to be a number between a and b". Abstract interpretation uses sets to express properties so this is $x \in P$ or $x \notin P$ with $P \triangleq \{x \mid a \leqslant x \wedge x \leqslant b\}$ easily encoded as a pair of numbers $[a, b]$. Of course set theory is expressed in logic so the task is not impossible but the encoding would introduce a useless layer of difficulty. Operations in abstract domains are predictable algorithms. This is hardly the case for theorem proving in logic. Explaining why an SMT solver failed may be a titanic task [17].

An important idea about abstract domains is that of a hierarchy of abstractions, using Galois connections or concretization functions *e.g.* [5]. If a concretization can easily be used to relate logics at different levels of abstraction, this is much harder for Galois connections since in general logic formulas have no normal form ($a \leqslant x \wedge x \leqslant b$ is also $x \leqslant b \wedge a \leqslant x$ or $a < x + 1 \wedge x - 1 < b$, *etc.*). Reasoning on equivalence classes of formulæ with same interpretation to get a best abstraction is not impossible in logic but not entirely natural and algorithmically costly. Without Galois connections, calculational design [7] is impossible, completeness is much harder to prove, *etc.*

4 Induction

The central problem in program analysis is to infer inductive arguments in proofs. Of course one can ask the user to provide inductive arguments. This makes verification simpler than program analysis [14]. This is a common approach in the small but is infeasible in the large for program of several million lines [12].

An approximate induction or co-induction is formalized in abstract interpretation by widening, narrowing and their duals [6]. For example widening has a simple geometric interpretation:

extrapolated in the direction of growth.

This is very hard to translate in terms of logic since the changes in the syntax of the formula are not necessarily related to the changes in the semantics of the formula. The complexity of an object and its logical denotation may be completely unrelated. Short formulæ may be very difficult to prove because they encode very complex objects while huge formulæ may be very easy to prove. An example is Craig interpolation [21], which is a dual narrowing [6], but one for which the result is not unique, without best choice, which is problematic and multiplies the necessary attempts for finding the right induction. So for induction, even simple ones such as widening, narrowing, and their duals, the evolution of the formula during fixpoint iteration provides no clue for induction. Moreover, this evolution of the iterates must be monitored for induction [15] and it is hard to use a logic to reason on itself. The same problem arises in the even simpler context of typing where types are represented syntactically by terms. For example the hidden widening in Milner's type inference system is syntactic identity of monotypes [24, 4] whereas liquid types [26] can handle only finitary abstract domains to avoid widening, which is known to be a severe expressivity limitation [9].

5 Conclusion

Logic reduces the representations of properties and formal reasonings to purely syntactic manipulations. Although logic is widely accepted for program verification and analysis, this is often limitative in static analysis by abstract interpretation. we prefer set theoretic and algebraic approaches to abstract interpretation for expressivity, efficiency, and scalability reasons. Of course this does not prevent us to widely use logic as a metalanguage, to study and prove the soundness or completeness of verification and static analysis methods [8].

References

[1] Ralph-Johan Back. Refinement calculus, lattices and higher order logic. In *NATO ASI PDC*, volume 118 of *NATO ASI Series*, pages 53–71. Springer, 1992.

[2] Clark W. Barrett, Leonardo Mendonça de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In *Haifa Verification Conference*, volume 6504 of *Lecture Notes in Computer Science*, page 3. Springer, 2010.

[3] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.

[4] Patrick Cousot. Types as abstract interpretations. In *POPL*, pages 316–331. ACM Press.

[5] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.

[6] Patrick Cousot. Abstracting induction by extrapolation and interpolation. In *VMCAI*, volume 8931 of *Lecture Notes in Computer Science*, pages 19–42. Springer, 2015.

[7] Patrick Cousot. Calculational design of a regular model checker by abstract interpretation. In *ICTAC*, volume 11884 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2019.

[8] Patrick Cousot. *Principles of Abstract Interpretation.* MIT Press, 2021 (to appear).

[9] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[11] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM Press, 1979.

[12] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[13] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Theories, solvers and static analysis by abstract interpretation. *J. ACM*, 59(6):31:1–31:56, 2012.

[14] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. Program analysis is harder than verification: A computability perspective. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 2018.

[15] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. A^2i: abstract^2 interpretation. *Proc. ACM Program. Lang.*, 3(POPL):42:1–42:31, 2019.

[16] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.

[17] Leonardo Mendonça de Moura and Grant Olney Passmore. The strategy challenge in SMT solving. In *Automated Reasoning and Mathematics*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2013.

[18] Isabel Garcia-Contreras, José F. Morales, and Manuel V. Hermenegildo. Incremental analysis of logic programs with assertions and open predicates. In *LOPSTR*, volume 12042 of *Lecture Notes in Computer Science*, pages 36–56. Springer, 2019.

[19] Erich Grädel. On transitive closure logic. In *CSL*, volume 626 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 1991.

[20] Christoph Haase. A survival guide to Presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018.

[21] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2005.

[22] Uwe Keller. Some remarks on the definability of transitive closure in first-order logic and datalog. June 2004.

[23] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. *Log. Methods Comput. Sci.*, 5(2), 2009.

[24] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375.

[25] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[26] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169. ACM, 2008.