Construction of invariance proof methods for parallel programs with sequential consistency

Patrick Cousot

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012, USA
`pcousot@cims.nyu.edu`

1. History

Program proofs can be done informally [47] as most mathematical reasonings or as an application of a formal method. The formal method is a "recipe" to make the prove. In general, it requires the discovery of an inductive property implying the program property to be proved. The inductive property is in general stronger/more precise than the one to be proved. The program property to be proved is given but the inductive property has to be discovered (preferably automatically, but this is an undecidable problem). It must be shown to satisfy verification conditions that imply that it is inductive (meaning that it can be proved to hold by recurrence on a well-founded relation based on the program computation steps [47, 41, 27, 15], the program structure [30], or data manipulated by the program [2, 17]).

Although initially applied to sequential imperative programs, such program proof methods were rapidly extended to parallel programs [44, 43, 45, 35] with sequential consistency hypothesis *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* [36].

2. Empiricism

The verification conditions can be postulated out of thin air (*e.g.* [27, 25]) and claimed to define the semantics of the programming language (*i.e.* to "assign meanings to programs" [27]). The problem is that the design of the verification conditions is by trial and errors and they can be unsound and/or incomplete, without any way to prove these facts.

For example the conjunction and disjunction rules of Hoare logic are not sound for all assertion languages [19, Sect. 5]. Predicate transformers [25] were incomplete for unbounded nondeterminism and had to be later generalized [26]. For similar reasons of lack of expressiveness of the assertion language, the Owicki and Gries proof method for parallel programs with shared variables [44] and without auxiliary variables is incomplete. The Owicki and Gries proof method for parallel programs with resources [45] as well [37]. The separation logic [42] as well, *etc.* Moreover, this empirical approach is completely language-dependent and basic principles have to be rediscovered whenever the language or its semantics is changed [34].

3. Constructionism

An alternative to empiricism is to derive the program proof method from an operational semantics by calculational design [6, Ch. 3], [20], [22]. The general idea is that a program property is an abstraction of the most general program property induced by the operational semantics of the programming language.

For example the operational semantics of a program can be defined by a transition system $\langle S,\, t\rangle$ where S is a set of states and $t \in \wp(S \times S)$ a transition relation [6, Ch. 3]. The set of reachable states from a set of initial states $S_0 \in \wp(S)$ is $r = \{s' \in S \mid \exists s \in S_0 : \langle s,\, s'\rangle \in t^*\}$ where t^* is the reflexive transitive closure of t. An invariant $P \in \wp(S)$ is an over-approximation of the reachable states $r \subseteq P$. It is invariant in that during any execution, a reachable state will always satisfy P (*i.e.* belong to P). This can be formulated as a fixpoint problem in that $r = \mathsf{lfp}\ F$ where $F(X) = S_0 \cup \{s' \mid \exists s \in X : \langle s,\, s'\rangle \in t\}$ [6, 7, Ch. 3] so that we have to prove that $\mathsf{lfp}\ F \subseteq P$.

4. Fixpoint Induction

Since this strongest/most general property (reachable states r in the above example) can be expressed as a fixpoint $\mathsf{lfp}\ F$, and its abstractions as well, the verification conditions directly derive from fixpoint induction, a direct immediate consequence of Tarski's theorem $\mathsf{lfp}\ F = \sqcap\{X \in L \mid F(X) \sqsubseteq X\}$ (where $F \in L \mapsto L$ on the complete lattice $\langle L,\, \sqsubseteq\rangle$) [46]. We have $\mathsf{lfp}\ F \sqsubseteq P$ if and only if $\exists I : F(I) \sqsubseteq I \wedge I \sqsubseteq P$ (where $\mathsf{lfp}\ F$ is the strongest program property, $\sqsubseteq$ is logical implication, P is the property to be proved, I is a stronger inductive property, and F (or $F(I) \sqsubseteq I$) is the verification condition.

For the above example, this is $\exists I : S_0 \subseteq I \wedge \forall s \in I : \forall s' : \langle s,\, s'\rangle \in t \Rightarrow s' \in I$ *i.e.* the invariant must be true for all initial states and if it is true for one state s, it must be true for all its possible successors s' by one more transition, if any.

Fixpoint induction is a universal induction principle for proving invariance (with numerous variants [13], for example backward methods consist in inverting the transition relation [21]).

5. Soundness and Completeness

Soundness consists in proving that the proof method is correct. Soundness follows from $I \in \{X \in L \mid F(X) \sqsubseteq X\}$ so $\mathsf{lfp}\ F = \sqcap\{X \in L \mid F(X) \sqsubseteq X\} \sqsubseteq I$ and $I \sqsubseteq P$ proving $\mathsf{lfp}\ F \sqsubseteq P$ by transitivity. Completeness consists in proving that if the property to be proved does hold (*i.e.* $\mathsf{lfp}\ F \sqsubseteq P$) then the proof method is always applicable.

Completeness follows from the fact that it is always possible to choose $I = \mathsf{lfp}\ F$ so $F(I) = I \sqsubseteq I \wedge I \sqsubseteq P$. Although $I = \mathsf{lfp}\ F$ can always be calculated iteratively, the iterates might have to be transfinite [10], and so this does not provide an effective algorithm to compute the inductive property. This is why in deductive methods the inductive property must be provided by the end-user, the proof system just generating the verification conditions $F(I) \sqsubseteq I$ and $I \sqsubseteq P$, and a theorem prover or SMT-solver is used to check the implication $\sqsubseteq$.

To continue our example, the operational semantics of a parallel program $[\![\mathtt{P}_1 \| \ldots \| \mathtt{P}_n]\!]$ with shared variables $\mathtt{x}_1, \ldots, \mathtt{x}_m$ and sequential consistency can be formalized by the transition relation $t = \bigcup_{i=1}^{n} t_i$ where t_i is the transition relation for process $\mathtt{P}_i$, $i = 1, \ldots, n$. The states are of the form $\langle c_1, \ldots, c_n, x_1, \ldots, x_m\rangle \in S$ where $c_i \in C_i$ is the value of the program counter of process $\mathtt{P}_i$, $i = 1, \ldots, n$ and $x_1, \ldots, x_m$ are the values of the shared variables. So a transition t_i by process $\mathtt{P}_i$ changes the program counter c_i of that process and potentially the values $x_1, \ldots, x_m$ of the shared variables $\mathtt{x}_1, \ldots, \mathtt{x}_m$. The program execution is the interleaving of the process executions. In the Lamport proof method [35] for parallel programs $[\![\mathtt{P}_1 \| \ldots \| \mathtt{P}_n]\!]$ with shared variables $\mathtt{x}_1, \ldots, \mathtt{x}_m$, an assertion on $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n, x_1, \ldots, x_m$ is attached to each program point c_i of each process $\mathtt{P}_i$, $i = 1, \ldots, n$. The method is sound and complete. The verification condition states that for all processes i, $i = 1, \ldots, n$, the local assertions attached to program points $c_i \in C_i$ should be invariant for transitions t_i (so-called sequential proof) as well as transitions t_j, $j \in [1, n] \setminus \{i\}$ (so-called absence of interference proof). In the Owicki and Gries proof method [44] for parallel programs $[\![\mathtt{P}_1 \| \ldots \| \mathtt{P}_n]\!]$, the assertion attached to each program point c_i of each process $\mathtt{P}_i$, $i = 1, \ldots, n$ is on the values $x_1, \ldots, x_m$ of the shared variables $\mathtt{x}_1, \ldots, \mathtt{x}_m$ only. This is sound but incomplete (since it is not possible to specify when process $\mathtt{P}_i$ is at some point c_i where the other processes should be, *e.g.* out of a critical section). Adding auxiliary variables as proposed by [44] makes the method complete (since, as shown in [20, Th. 10.0.4 applied to Ex. 10.0.5 and Ex. 10.0.6], it is always possible to extend to transition system with auxiliary variables simulating the program counters of the processes). Rely-guarantee methods [32, 3] are a different way of expressing the strongest invariant as a fixpoint [38] based on asynchronous iterations with memory [5].

6. Galois connections

More generally, proof methods do not directly refer to the semantics but to an abstraction of the semantics [9, 11]. Most often the abstraction can be be formalized by a Galois connection $\langle \wp(S), \subseteq \rangle \xleftarrow[\alpha]{\gamma} \langle A, \sqsubseteq \rangle$ such that $\alpha(P) \sqsubseteq Q \iff P \subseteq \gamma(Q)$ where $P \in \wp(S)$ is a concrete property, and $Q \in A$ is an abstract property, $\alpha(P)$ is the best abstraction of the concrete property P, and $\gamma(Q)$ is the concretization of the abstract property Q [9, 11].

For the Owicki-Gries example, $A = \prod_{i=1}^{n} \prod_{c_i \in C_i} V^m$ where V is the domain of values $x_1, \ldots, x_m$ of the shared variables. The abstraction is $\alpha(P) \triangleq \prod_{i=1}^{n} \prod_{c_i \in C_i} \{\langle x_1, \ldots, x_m \rangle \mid \exists c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n : \langle c_1, \ldots, c_i, \ldots, c_n, x_1, \ldots, x_m \rangle \in P\}$. The concretization is $\gamma(\prod_{i=1}^{n} \prod_{c_i \in C_i} Q_{i,c_i}) \triangleq \{\langle c_1, \ldots, c_{i-1}, c_i, c_{i+1}, \ldots, c_n, x_1, \ldots, x_m \rangle \mid \exists c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n : \langle x_1, \ldots, x_m \rangle \in Q_{i,c_i}\}$. The local assertions Q_{i,c_i} ignore the program counters of other processes (which can be any $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n$). This is the source of the incompleteness. The partial order $\sqsubseteq$ is pointwise set-theoretic implication (*i.e.* inclusion $\subseteq$ for sets) $\prod_{i=1}^{n} \prod_{c_i \in C_i} Q_{i,c_i} \sqsubseteq \prod_{i=1}^{n} \prod_{c_i \in C_i} Q'_{i,c_i}$ if and only if $\forall i \in [1, n] : \forall c_i \in C_i : Q_{i,c_i} \subseteq Q'_{i,c_i}$.

Then for an abstract property $Q \in A$, the program verification consists in proving is $\alpha(\mathsf{lfp}\ F) \sqsubseteq Q$.

If $\alpha \circ F = \overline{F} \circ \alpha$ then this is equivalent to $\mathsf{lfp}\ \overline{F} \sqsubseteq Q$ where $\overline{F} = \alpha \circ F \circ \gamma \in A \mapsto A$ defines the abstract verification conditions [11, theorem 7.1.0.4(3)], [23, lemma 4.3], [1, fact 2.3]. Then, as before but this time in the abstract, $\alpha(\mathsf{lfp}\ F) \sqsubseteq Q \iff \exists I \in A : \overline{F}(I) \sqsubseteq I \wedge I \sqsubseteq Q$.

If $\alpha \circ F \mathrel{\dot{\sqsubseteq}} \overline{F} \circ \alpha$ (where $\dot{\sqsubseteq}$ is $\sqsubseteq$ pointwise) then the method is sound (*i.e.* $\alpha(\mathsf{lfp}\ F) \sqsubseteq Q \Leftarrow \exists I : \overline{F}(I) \sqsubseteq I \wedge I \sqsubseteq Q$) but in general incomplete ($\not\Rightarrow$). This is almost always the case in static analysis [9, 11].

7. Abstraction

Not all abstractions can be formalized by Galois connections. This is the case when there is no best/most precise abstraction.

An example is the use of logics like first-order logic, which is an abstraction. In general the strongest/most general property is not expressible in the logic which makes it incomplete. In that case only relative completeness is provable (*e.g.* [4] for Hoare logic [30]) *i.e.* under the assumption that the strongest/most general set-theoretic property is expressible in the logic. This is why we represent properties by the set of all objects which have this property rather than by a logical formula. In that case only the concretization function γ is used [16]. This is the case for formal logics or types which may not way to have a best way to express a property (*e.g.* a first-order logic with addition only cannot express a multiplication while a program with addition only can expression multiplication with a loop. Adding multiplication to the logic, is not enough for exponentiation, *etc.*).

8. The Hierarchy of Proof Methods

Finally by choosing different abstractions, one obtained a hierarchy of famous (as well as completely forgotten) proof methods for parallel programs with sequential consistency [6, 20, 14, 37, 38].

9. Methodology

In summary, given a language, an operational semantics, a fixpoint definition of runtime properties, an abstraction into properties of interest, a further abstraction to express such properties of interest locally, there is a mathematical methodology, checkable by a proof checker [33], to construct a proof method which is guaranteed to be sound, and depending on the appropriate choice of the abstractions, complete. Beyond proof methods, this methodology extends to all program semantics [8, 28].

10. Applications

Using approximate abstractions, this contructive methodology leads to the design of sound program enumerative checking methods (aiming at checking a given property in the abstract *e.g.* [18]) and analysis methods (aiming at automatically infer the invariant property from the program text only *e.g.* [9, 11]). For calculability, these abstractions have to be incomplete (sometimes even for sets of states S which are finite [18, 29]). In particular for parallel programs, approximation is indispensable for scalability [12, 14], from prototype [31, 40] to production-quality implementations [38, 39].

References

[1] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, 1986.

[2] R. Burstall. Program proving as hand simulation with a little induction. In J. Rosenfeld, editor, *Information Processing 74, Proceedings of IFIP Congress 74, Stockholm, Sweden*, pages 308–312. North-Holland, August 5–10 1974.

[3] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

[4] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[5] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Sep. 1977. 15 p., `http://cs.nyu.edu/~pcousot/publications.www/Cousot-IMAG-RR88-Sep-1977.pdf`.

[6] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French).* Thèse d'État ès

sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978. `http://cs.nyu.edu/~pcousot/CousotTheseEsSciences1978.pdf`.

[7] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

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

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[10] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In A. V. Aho, S. N. Zilles, and B. K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979.

[12] P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes (shortened version). In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1980.

[13] P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In D. Néel, editor, *Tools & Notions for Program Construction: an Advanced Course*, pages 75–119. Cambridge University Press, Cambridge, UK, Aug. 1982. `http://cs.nyu.edu/~pcousot/publications.www/CousotCousot82-TNPC.PDF`.

[14] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, New York, United States, 1984. `http://cs.nyu.edu/~pcousot/publications.www/CousotCousot-Macmillan-84-p243--271-1984.pdf`.

[15] P. Cousot and R. Cousot. "À la Floyd" induction principles for proving inevitability properties of programs. In M. Nivat and J. Reynolds, editors, *Algebraic methods in semantics*, pages 277–312. Cambridge University Press, Cambridge, UK, Dec. 1985.

[16] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[17] P. Cousot and R. Cousot. "À la Burstall" intermittent assertions induction principles for proving inevitable ability properties of programs. *Theoretical Computer Science*, 120(1):123–155, 1993.

[18] P. Cousot and R. Cousot. Temporal abstract interpretation. In M. N. Wegman and T. W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 12–25. ACM, 2000.

[19] P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 213–232. ACM, 2012.

[20] R. Cousot. Reasoning about program invariance proof methods. Res. rep. CRIN-80-P050, Centre de Recherche en Informatique de Nancy (CRIN), Institut National Polytechnique de Lorraine, Nancy, France, July 1980. `http://www.di.ens.fr/~cousot/publications.www/CRIN-80-P050-jul-1980.PDF`.

[21] R. Cousot. Proving invariance properties of parallel programs by backward induction. Res. rep. LRIM-82-02, University Paul Verlaine, Metz, France, Mar. 1981. `http://www.di.ens.fr/~rcousot/publications.www/RR-LRIM-82-02-mar-1981.PDF`.

[22] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles (in French).* Thèse d'État ès sciences mathématiques, Institut National Polytechnique de Lorraine, Nancy, France, 21 November 1985. `http://www.di.ens.fr/~rcousot/publications.www/RadhiaCousotTheseEsSciences.PDF`.

[23] J. W. de Bakker, J. C. Meyer, and J. I. Zucker. On infinite computations in denotational semantics. *Theor. Comput. Sci.*, 26:53–82, 1983. Corrigendum in [24].

[24] J. W. de Bakker, J. C. Meyer, and J. I. Zucker. On infinite computations in denotational semantics. *Theor. Comput. Sci.*, 29:229–230, 1984.

[25] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

[26] E. W. Dijkstra and W. H. J. Feijen. *A method of programming.* Addison-Wesley, 1988.

[27] R. W. Floyd. Assigning meaning to programs. In J. Schwartz, editor, *Proc. Symp. in Applied Math.*, volume 19, pages 19–32. Amer. Math. Soc., 1967.

[28] R. Giacobazzi and I. Mastroeni. Non-standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003.

[29] R. Giacobazzi and F. Ranzato. States vs. traces in model checking by abstract interpretation. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 461–476. Springer, 2002.

[30] C. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, October 1969.

[31] B. Jeannet. Relational interprocedural verification of concurrent programs. *Software and System Modeling*, 12(2):285–306, 2013.

[32] C. B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University Computing Laboratory, June 1981. `https://www.cs.ox.ac.uk/files/3291/PRG25.pdf`.

[33] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015.

[34] O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2015.

[35] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[36] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[37] A. Malkis and L. Mauborgne. On the strength of Owicki-Gries for resources. In H. Yang, editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2011.

[38] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014.

[39] A. Miné and D. Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *2015 International Conference on Embedded Software, EMSOFT 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 65–74. IEEE, 2015.

[40] R. Monat. Thread-modular analysis designing relational abstractions of interferences. Internship report, École normale supérieure de Lyon, France, July 2015. `http://rmonat.fr/batman/report.pdf`.

[41] P. Naur. Proofs of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[42] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009.

[43] S. S. Owicki. A consistent and complete deductive system for the verification of parallel programs. In A. K. Chandra, D. Wotschke, E. P. Friedman, and M. A. Harrison, editors, *Proceedings of the 8th Annual ACM Symposium on Theory of Computing, May 3-5, 1976, Hershey, Pennsylvania, USA*, pages 73–86. ACM, 1976.

[44] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[45] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[46] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

[47] A. M. Turing. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines, Mathematical Laboratory, Cambridge, UK*, pages 67–69, 24 June 1949. Reproduced as "An early program proof by Alan Turing", in F.L. Morris and C.B. Jones (Eds), *Annals of the History of Computing*, Vol. 6, Apr. 1984, `http://www.turingarchive.org/browse.php/B/8`.