Abstract Interpretation

Patrick Cousot

LIENS, École Normale Supérieure
45 rue d'Ulm, 75230 Paris Cedex 05, France
cousot@dmi.ens.fr

January 16, 1996

Abstract interpretation is a theory for approximating the semantics of discrete dynamic systems, e.g. computations of programming languages.

1 An Historical Perspective

Abstract interpretation [9] can be traced back to data-flow analysis (see a survey in [35]) and the need for a posteriori verification of such ad-hoc program analysis algorithms. Abstract interpretation is often understood in the limited sense of pseudo-evaluation [40], that is of program analysis by running an abstract interpreter using abstract instead of concrete values. A classical example of such a pseudo-evaluation is the rule of signs [47]. By considering convergence acceleration methods such as widening/narrowing [8], one can extend the idea to infinite sets of abstract values. Program inversion can be used to explain backward program analysis methods of the same vein. However more elaborate program analysis methods such as the static determination of invariants in the form of a conjunction of linear inequalities holding among values of the variables at each program point [20] is closer to program proof methods than program evaluation by assignment of abstract values to variables. This was understood in [12] which proposes a framework for constructing semantics abstract of transition systems (i.e. small-step operational semantics) and designing abstract semantic domains compositionnally (using e.g. reduced products, disjunctive completions, etc.).

An early and conceptually important application of abstract interpretation was strictness analysis introduced by [38]. In the context of a lazy first-order functional language, it consists in studying the definedness of a function call $f(x_1, \ldots, x_n)$ in terms of the definedness (nontermination or run-time error) of the parameters x_1, ..., x_n. To handle nontermination, denotational semantics was considered more adequate than operational ones [41, 42, 43].

Using the idea of lifting an approximation of data in a poset domain $\mathcal{D}$ to monotonic functions $\mathcal{D} \mapsto \mathcal{D}$, as introduced in [11] for first-order procedures with value-result parameters, [4] lifted strictness analysis of functional programs at higher-order. This led to a stream of research on abstract interpretation, essentially based on denotational semantics which is (see [1] and [29] for surveys where this point of view is prominent) which is mainly adequate for functional languages whereas operational-based abstract interpretation (see [6] for an early survey where this point of view is prominent) turned out to be more adequate for imperative, logic (see [22] for a survey) and more recently concurrent, distributed and object-oriented languages.

Both frameworks can be unified [17] on the basis of [12], by lifting operational semantics to handle infinite behaviors [16], considering the equivalence between rule-based and fixpoint presentations of semantics specifications [18] and viewing denotational semantics as part of a hierarchy of abstractions of operational semantics [16].

2 Principles of Abstract Interpretation

A semantics S of a programming language L associates a semantic value $S[\![p]\!] \in \mathcal{D}$ in the semantic domain $\mathcal{D}$ to each program p of L. The semantic domain $\mathcal{D}$ can be transition systems (for small-step operational semantics), pomsets, traces, relations (for big-step operational semantics), higher-order functions (for denotational semantics), etc. $\mathcal{D}$ is usually defined compositionnally by induction on the structure of run-time objects (computations, data, etc.). S is defined compositionnally by induction on the syntactical structure of programs, using e.g. fixpoint definitions to handle iteration, recursion, etc.

An empirical approach to abstract interpretation consists in a priori choosing a problem-specific abstract semantics domain $\mathcal{D}^\sharp$ and an abstract semantics $S^\sharp \in \mathsf{L} \mapsto \mathcal{D}^\sharp$ which is designed intuitively for a specific language L. Then safety, correctness or soundness is established by proving that a

soundness relation σ [39] satisfies:

$$\forall p \in \mathsf{L} : \sigma(S[\![p]\!], S^\sharp[\![p]\!])$$

If the abstract semantics is computable ($\mathcal{D}^\sharp$ is usually assumed to be finite), we can infer that the abstract interpretation is sound in the sense that:

$$S[\![p]\!] \quad \in \quad \{S \mid \sigma(S, S^\sharp[\![p]\!])\}$$

which may be sufficient to prove program properties e.g. that some "programs cannot go wrong". This approach has the advantage of theoretical simplicity. By multiplication of superficial differences in the presentation of the abstract semantics $S^\sharp[\![p]\!]$ (e.g. rule-based versus fixpoint) and more fundamental variations on the choice of the abstract domain (corresponding to different and sometimes non-comparable degrees of approximation), one rapidly obtain an abundant literature which is difficult to gulp down and understand synthetically. Moreover, the choice of $\mathcal{D}^\sharp$ and σ offer no guideline for the design of the abstract semantics $S^\sharp$ with respect to the concrete semantics S.

The approach propounded in [9, 12] is constructive in the sense that once the standard semantics S and an approximation α are chosen, one can derive a best choice for the corresponding abstract semantics $S^\sharp$. More precisely let us call elements of the powerset:

$$\mathcal{D}_{\mathrm{Coll}} \quad \stackrel{\text{def}}{=} \quad \wp(\mathcal{D})$$

program (concrete) properties. Define $S_{\mathrm{Coll}} \in \mathsf{L} \mapsto \mathcal{D}_{\mathrm{Coll}}$ to be the collecting semantics:

$$S_{\mathrm{Coll}}[\![p]\!] \quad \stackrel{\text{def}}{=} \quad \{S[\![p]\!]\}$$

(This is a conceptual step, since no other detailed specification of S_{Coll} is needed (but for the design of formal proof methods).) $S_{\mathrm{Coll}}[\![p]\!]$ is the strongest program property. We have seen that an abstract property P, such as $\{S \mid \sigma(S, S^\sharp[\![p]\!])\}$ above, is weaker in that $S_{\mathrm{Coll}}[\![p]\!] \subseteq P$. We call $\subseteq$ the approximation ordering. Now the abstraction function is a map $\alpha \in \mathcal{D}_{\mathrm{Coll}} \mapsto \mathcal{D}_{\mathrm{Coll}}$. We call $\alpha[\mathcal{D}] \stackrel{\text{def}}{=} \{\alpha(S) \mid S \in \mathcal{D}\}$ the abstract domain. We often use an isomorphic representation $\mathcal{D}^\sharp$ for $\alpha[\mathcal{D}]$ and directly define $\alpha \in \mathcal{D}_{\mathrm{Coll}} \mapsto \mathcal{D}^\sharp$. An example would be the approximation of a trace-based semantics by a relational/denotational semantics:

$$\begin{array}{rcl} \alpha(P) & \stackrel{\text{def}}{=} & \{\langle s_0, s_n\rangle \mid s_0 s_1 \ldots s_n \in P\} \\ & & \{\langle s_0, \bot\rangle \mid s_0 s_1 \ldots \in P \text{ is an infinite trace}\} \end{array}$$

Another approximation, ignoring distinction between finite and infinite computations, would be the approximation of traces by sets of states:

$$\alpha(P) \quad \stackrel{\text{def}}{=} \quad \{s_i \mid \exists s_0 s_1 \ldots s_i \ldots \in P\}$$

which is adequate for safety/invariance properties. If states s_i map identifiers $\mathtt{X} \in \mathsf{I}$ to values, then a further attribute-independent approximation consists in ignoring intervariable dependencies:

$$\alpha(P) \stackrel{\text{def}}{=} \prod_{\mathtt{X} \in \mathsf{I}} \{s(\mathtt{X}) \in P\}$$

Once such abstract domain $\mathcal{D}^\sharp$ and abstraction function α have been chosen, it remains to derive the abstract semantics $S^\sharp[\![p]\!]$, $p \in \mathsf{L}$. We proceed compositionnally, by induction on the syntactic structure of programs p in order to follow that of $S[\![p]\!]$. The price to pay is that in general the ideal $\alpha(S_{\text{Coll}}[\![p]\!])$ has to be approximated by $S^\sharp[\![p]\!] \supseteq \alpha(S_{\text{Coll}}[\![p]\!])$. For example, consider a fixpoint definition:

$$S[\![p(p_1, \ldots, p_n)]\!] \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq} F_p[S[\![p_1]\!], \ldots, S[\![p_n]\!]]$$

For simplicity, we consider the simple case when the computational ordering $\sqsubseteq$ coincide with the approximation ordering $\subseteq$. Assuming, by induction hypothesis, that we know a sound abstract semantics for the program components $p_1, \ldots, p_n$:

$$\alpha(S_{\text{Coll}}[\![p_i]\!]) \subseteq S^\sharp[\![p_i]\!], \ i = 1, \ldots, n$$

then we look, by algebraic formula manipulation, for $F_p^\sharp$ satisfying for all $C_1, \ldots, C_n \in \mathcal{D}_{\text{Coll}}$:

$$\alpha(\{F_p[S_1, \ldots, S_n] \mid \forall i = 1, \ldots, n : S_i \in C_i\}) \subseteq F_p^\sharp[\alpha(C_1), \ldots, \alpha(C_n)]$$

in order to conclude (under suitable hypotheses, see [12]) that:

$$\alpha(\text{lfp}^{\sqsubseteq} F_p[C_1, \ldots, C_n]) \subseteq \text{lfp}^{\sqsubseteq^\sharp} F_p^\sharp[\alpha(C_1), \ldots, \alpha(C_n)]$$

This leads to the definition of the abstract semantics:

$$S^\sharp[\![p(p_1, \ldots, p_n)]\!] \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq^\sharp} F_p^\sharp[S^\sharp[\![p_1]\!], \ldots, S^\sharp[\![p_n]\!]]$$

which is sound, by construction, so that no a posteriori verification is necessary. When equality holds, we have a completeness property which is useful to design hierarchies of semantics [15]. By identifying useful abstract algebras consisting of an abstract domain $\mathcal{D}^\sharp$ and abstract operations $F^\sharp$ corresponding to common primitive operations F used in the semantic definition of programming languages, on can design abstraction libraries of general scope. Finally, the abstraction can be parameterized with other abstractions in order to obtain generic semantic definitions and abstract interpreters.

We have sketched an abstract interpretation framework based on abstraction function α. Other alternatives (using a soundness relation σ, a concretization γ, an abstraction/concretization pair $\langle \alpha, \gamma \rangle$, ...) are studied

in [13]. The abstraction function α is often chosen as the upper-adjoint of a Galois connection in order to ensure the existence of a best (most precise) approximation among all the possible sound ones. Other equivalent formulations using Moore families, closure operators, ideals, ... where introduced in [12]. The case when the computational ordering $\sqsubseteq$ does not coincide with the approximation ordering $\subseteq$ in their concrete or abstract versions (as is the case in comportment analysis) is considered in [17] in a language and semantic-independent way.

3 Application of Abstract Interpretation to Program Verification and Analysis

Abstract semantics can be understood as specifications of proof methods. For example the Floyd-Naur/Hoare invariance proof method directly derives from the observation that the existence of invariants $I_1, \ldots, I_n, I$ satisfying the verification condition:

$$\bigwedge_{i=1}^{n} S^\sharp[\![p_i]\!] \sqsubseteq^\sharp I_i \quad \wedge \quad F_p^\sharp[I_1, \ldots, I_n](I) \sqsubseteq^\sharp I$$

implies, as a direct consequence of Tarski's fixpoint theorem, that:

$$\left(\mathrm{lfp}^{\sqsubseteq^\sharp} F_p^\sharp[S^\sharp[\![p_1]\!], \ldots, S^\sharp[\![p_n]\!]]\right) \quad \sqsubseteq^\sharp \quad I$$

(see [7] for more details).

The most popular use of abstract semantics $S^\sharp$ is for the specification of program analyzers also called abstract interpreters. The design is often generic that is parameterized by abstractions for basic data or control structures. For program analysis, the abstract domain $\mathcal{D}^\sharp$ must be computer-representable. An important aspect of research of abstract interpretation is concerned by the composable design of abstractions α by induction on the mathematical structure of the semantic domain $\mathcal{D}$ (which, for typed languages, often coincide with the type structure of the language) for all possible data and control structures encountered in programming languages. A basic idea is that abstracting an abstraction yields another abstraction. For example, sets of pairs (e.g. records with two fields or program states consisting of a control and a data component) can be approximated in a relational way [30] as a set of pairs of abstract properties:

$$\alpha(A) \quad \stackrel{\text{def}}{=} \quad \{\langle \alpha(\{x\}), \alpha(\{y\})\rangle \mid \langle x, y\rangle \in A\}$$

A further attribute-independent abstraction leads to a less expensive but

also less precise approximation of a set of pairs by a pair of abstract properties:

$$\alpha(A) \stackrel{\text{def}}{=} \langle \alpha(\{x \mid \langle x, y\rangle \in A\}), \alpha(\{y \mid \langle x, y\rangle \in A\})\rangle$$

For basic control and data structures one is interested in general purpose abstractions. Let us consider a small range of samples of data abstractions:

- For attribute-independent abstractions of sets of vectors of numbers, one can consider signs [12], intervals [8, 9], simple congruences [26] generalizing parity [12]. For relational abstractions, one can consider linear equalities [33], inequalities [20], congruences [27], congruencial trapezoids [36], etc.

- For the context-free abstraction of formal languages, that is sets of strings on a finite alphabet, one can consider regular expressions, grammars (thus making the link with set-based analysis [19]). Unitary-prefix monomial decompositions of rational subsets of a free monoid (such as $\{\texttt{X.tl}^m\texttt{.hd.tl}^n\texttt{.hd.tl}^p \mid m = n = p\}$) [24] provide an example of context-sensitive abstraction. These abstractions can be used for pointer analysis based on location-free/storeless models of computation [23].

- For the abstraction of sets of graphs, as used e.g. in store-based pointer analysis pioneered by [31], one can refer to the survey [25].

For control structures, even in the limited case of interprocedural analysis, a complete survey would be necessary. For example and to remain very succinct, program loops involves solving an equation $X = F_\ell^\sharp(X)$ originating from the fixpoint definition $\text{lfp}^{\sqsubseteq^\sharp} F_\ell^\sharp$ of the abstract semantics $S^\sharp\ell$ of a program ℓ. This is one of the typical algorithmic problems involved in abstract interpretation. One classical solution, when $F_\ell^\sharp$ is a monotonic operator on a poset $\langle \mathcal{D}^\sharp, \sqsubseteq^\sharp\rangle$ with infimum $\bot$, is iteration: $X^0 \stackrel{\text{def}}{=} \bot$, $X^{n+1} \stackrel{\text{def}}{=} F_\ell^\sharp(X^n)$ until convergence. Widenings [14], which can be understood as local dynamic change of abstract semantic domain during fixpoint iteration, can be used on one hand when least upper bounds are missing and on the other hand to accelerate convergence (in large or infinite domains). When the abstract domain $\mathcal{D}^\sharp$ is a cartesian product, chaotic iteration methods are available [10]. Such demand-driven chaotic iterations can be generalized to first-order equations $f(X) = F_p^\sharp[f](X)$ involved e.g. in interprocedural analysis using the idea that the function f needs to be known for some but in general not all its parameters, see [11, 3] and the reformulation as minimal function

graphs in [32]. This generalizes at higher-order [45] but efficient algorithms for solving such higher-order equations, including elimination methods, remain to be studied. Observe that, contrary to e.g. type inference, soundness (and relative completeness) can be established once for all (may be parameterized by basic abstractions for generic implementations) and not for each particular instance of the program analysis problem.

The complexity issues in abstract interpretation have only been touched upon, see e.g. [21, 30, 44, 46]. A common error is that abstract interpretation is though to be inherently exponential (as opposed to polynomial dataflow analysis). This can be exactly the contrary! It can be polynomial or polynomial on the average but exponential in pathological cases which are rare enough to be cut off by widening (see e.g. [34] for polymorphic type inference à la Hindley-Milner). In general, exponential costs can be avoided by using widenings introducing further approximation as analysis time elapses. The common idea that program analyzers should be as fast as compilers does not necessarily take the cost/benefit trade-off into account. Is it better to spend 8 (night) hours of CPU time or 8 (day) hours of man-power for finding a crucial programming error?

4 Applications of Abstract Interpretation

Abstract interpretation has been used for highly-performant compilers (e.g. strictness analysis), program transformation (e.g. determination of invariants for program vectorization and parallelization, binding-time analysis for partial evaluation, etc.), test generation for program debugging, abstract debugging (involving abstract values/properties instead of concrete ones [2]), polymorphic type inference [37], effect systems, model checking [5], verification of hybrid systems [28], etc.

5 Conclusion and Research Perspectives

Although the designer of program analyzers may prefer empirical approaches, abstract interpretation is often an indispensable guideline to avoid conceptual errors since it provides a methodology to design a formal specification.

Theoretical researchers working on semantics, proofs and program analysis may find the abstract interpretation approach too demanding and prefer presentation methods looking brand new, often superficially different from existing ones.

However there is a need for models and theories allowing for the synthetic understanding of the abundant literature on semantics, type systems, logics of computations, program verification, program analysis, partial evaluation, etc. A synthetic point of view consists in considering these topics as involving the specification of more or less refined or abstract semantics of computations and observing that abstract interpretation is a constructive theory for studying their relationship or better deriving them from one another. Although this methodology is more demanding than a specific study for each particular problem, it might be necessary to have these topics evolve from craft to science.

A number of problems remain to be considered or more thoroughly studied, e.g.:

- General frameworks formalizing the notion of semantic approximation;
- Models of computations (e.g. true concurrency) to be used in concrete semantics;
- Design of hierarchies of parameterized semantics for programming languages (such as concurrent and object-oriented languages);
- Abstract domains for non-numerical objects (formal languages, graphs, types, etc.);
- Design of abstraction functions specifying different program analysis methods in order to compare their relative power;
- Optimal combination of existing program analyses;
- Equation resolution and convergence acceleration methods;
- Formal or experimental study of the complexity/benefit tradeoff of program analyses;
- Design and implementation of general-purpose libraries of abstract domains and their associated operations,
- Design of language-specific generic abstract interpreters.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Computers and their Applications. Ellis Horwood, 1987.

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. PLDI*, pages 46–55. ACM Press, 1993.

[3] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjø rner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Academgorodok, Novosibirsk, RUS, LNCS 735, pages 128–141. Springer-Verlag, June 28–July 2, 1993.

[4] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis of higher-order functions. *Sci. Comput. Prog.*, 7:249–278, Nov. 1986.

[5] R. Cleaveland, P. Iyer, and D. Yankelevitch. Optimality in abstractions of model checking. In A. Mycroft, editor, *Proc. Static Analysis Symposium, SAS'95*, Glasgow UK, 25–27 Sept. 1995, LNCS 983, pages 51–63. Springer-Verlag, 1995.

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

[7] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier, 1990.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2^{nd} International Symposium on Programming*, pages 106–130. Dunod, 1976.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *POPL*, pages 238–252, Los Angeles, Calif., 1977. ACM Press.

[10] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, N.Y., SIGPLAN Notices 12(8):1–12, 1977.

[11] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conference on Formal Description of Programming Concepts*, St-Andrews, N.B., CAN, pages 237–277. North-Holland, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press.

[13] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, Aug. 1992.

[14] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings of the Fourth International Symposium, PLILP'92*, Leuven, BEL, 13–17 Aug. 1992, LNCS 631, pages 269–295. Springer-Verlag, 1992.

[15] P. Cousot and R. Cousot. Constructing a hierarchy of semantics of functional programs by abstract interpretation, invited lecture. In *Workshop on Static Analysis, WSA'92*, Bordeaux, FRA, 23–25 Sept. 1992.

[16] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *19th POPL*, pages 83–94, Albuquerque, N.M., 1992. ACM Press.

[17] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. 1994 ICCL*, Toulouse, FRA, pages 95–112. IEEE Comp. Soc. Press, 16–19 May 1994.

[18] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In P. Wolper, editor, *Computer Aided Verification, Proc. 7th International Conference, CAV'95*, Liège, BEL, LNCS 939, pages 293–308. Springer-Verlag, 3–5 July 1995.

[19] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pages 170–181, La Jolla, Calif., 25–28 June 1995. ACM Press.

[20] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, Ariz., 1978. ACM Press.

[21] S.K. Debray. The mythical free lunch: Notes on the complexity/precision tradeoff in dataflow analysis of logic programs. In R. Giacobazzi, editor, *Proc. ICLP'91 Pre-Conference Workshop on Semantics-Based Analysis of Logic Programs*, INRIA, Rocquencourt,

FRA. Dipartimento di Informatica, Università di Pisa, ITA, June 24, 1991.

[22] S.K.. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in Logic Programming Theory*, International Schools for Computer Scientists, section 3, pages 115–182. Clarendon Press, 1994.

[23] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proc. 1992 ICCL*, Oakland, Calif., pages 2–13. IEEE Comp. Soc. Press, 20–23 Apr. 1992.

[24] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. PLDI*, pages 230–241. ACM Press, Orlando, Fla, June 1994.

[25] A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers, invited paper. In *Proc. PEPM'95*, La Jolla, Calif., 21–23 June 1995, pages 226–229. ACM Press, June 1995.

[26] P. Granger. Static analysis of arithmetical congruences. *Int. J. of Comp. Math.*, 30:165–190, 1989.

[27] P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, Proc. International Joint Conference on Theory and Practice of Software Development*, Brighton, U.K., Volume 1 (CAAP'91), LNCS 493, pages 169–192. Springer-Verlag, 1991.

[28] N. Halbwachs, J.-É. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, editor, *Proc. Static Analysis Symposium, SAS'94*, Namur BEL, 20–22 Sept. 1994, LNCS 864, pages 223–237. Springer-Verlag, 1994.

[29] N. Jones and F. Nielson. Abstract interpretation: a semantic-based tool for program analysis. In S. Abramsky, Gabbay D.M., and Maibaum T.S.E., editors, *Semantic modelling*, number 4 in Handbook of Logic in Computer Science. Clarendon Press, 1995.

[30] N.D. Jones and S.S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12, pages 380–393. Prentice-Hall, 1981.

[31] N.D. Jones and S.S. Muchnick. Flow-analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program*

Flow Analysis: Theory and Applications, chapter 4, pages 102–131. Prentice-Hall, 1981.

[32] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs: abridged version. In *13th POPL*, pages 296–306, St. Petersburg Beach, Fla, 1986. ACM Press.

[33] G. Kildall. A unified approach to global program optimization. In *1st POPL*, pages 194–206, Boston, Mass., Oct. 1973. ACM Press.

[34] H.G. Mairson. Deciding ml typability is complete for deterministic exponential time. In *17th POPL*, pages 382–401., San Francisco, Calif., Jan. 17-19 1990. ACM Press.

[35] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks: A unified model. *Acta Inf.*, 28:121–163, 1990.

[36] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proc. ACM International Conference on Supercomputing, ICS'92*, pages 226–235, Washington D.C., July 1992.

[37] B. Monsuez. System F and abstract interpretation. In A. Mycroft, editor, *Proc. Static Analysis Symposium, SAS'95*, Glasgow, UK, 25–27 Sept. 1995, LNCS 983, pages 279–295. Springer-Verlag, 1995.

[38] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Dissertation, CST-15-81, Department of Computer Science, University of Edinburgh, Edinburgh, SCO, Dec. 1981.

[39] A. Mycroft and N.D. Jones. A relational framework for abstract interpretation. In N.D. Jones and H. Ganzinger, editors, *Programs as Data Objects, Proceedings of a Workshop*, Copenhagen, DEN, 17-19 Oct. 1985, LNCS 215, pages 156–171. Springer-Verlag, 1986.

[40] P. Naur. Checking of operand types in ALGOL compilers. *BIT*, 5:151–163, 1965.

[41] F. Nielson. A denotational framework for data flow analysis. *Acta Inf.*, 18:265–287, 1982.

[42] F Nielson. Abstract interpretation of denotational definitions (a survey). In B. Monien and G. Vidal-Naquet, editors, *STACS 86*, LNCS 210, pages 1–20. Springer-Verlag, 1986.

[43] F. Nielson. Two-level semantics and abstract interpretation. *TCS — Fund. St.*, 69:117–242, 1989.

[44] H.R. Nielson and F. Nielson. Bounded fixed-point iteration. *J. Logic and Comp.*, 2(4):437–464, Aug. 1992.

[45] M. Rosendahl. Higher order chaotic iteration sequences. In *Programming Language Implementation and Logic Programming, Proc. 5th Int. Symp., PLILP'93*, Leuven, BEL, 13–17 Aug. 1993, LNCS 714, Springer-Verlag, 1993.

[46] R.C. Sekar, P. Mishra, and I.V. Ramakrishnan. On the power and limitation of strictness analysis based on abstract interpretation. In *18th POPL*, pages 37–48, Orlando, Fla, 1991. ACM Press.

[47] M. Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of an ACM Conference on Proving Assertions about Programs*, Las Cruses, N.M., SIGPLAN Notices 7(1):203–207, Jan. 6–7, 1972.