“Next 40 years of Abstract Interpretation”

Abstract Interpretation – 40 years back + some years ahead

N40AI 2017
January 21st, 2017
Paris, France

Patrick Cousot

pcousot@cs.nyu.edu  cs.nyu.edu/~pcousot
Abstract interpretation: origin (abridged)
Before starting (1972-73): formal syntax

• Radhia Rezig: works on precedence parsing (R.W. Floyd, N. Wirth and H. Weber, etc.) for Algol 68

  ➡ Pre-processing (by static analysis and transformation) of the grammar before building the bottom-up parser

• Patrick Cousot: works on context-free grammar parsing (J. Earley and F. De Remer)

  ➡ Pre-processing (by static analysis and transformation) of the grammar before building the top-down parser

---


• Patrick Cousot. *Un analyseur syntaxique pour grammaires hors contexte ascendant sélectif et général*. In Congrès AFCET 72, Brochure 1, pages 106-130, Grenoble, France, 6-9 November 1972.
Before starting (1972-73): formal semantics

- **Patrick Cousot**: works on the operational semantics of programming languages and the derivation of implementations from the formal definition

  ➡ Static analysis of the formal definition and transformation to get the implementation by “pre-evaluation” (similar to the more recent “partial evaluation”)

---

Vision (1973)

Pas le niveau de "compréhension" des programmes. Les langages actuels ne sont pas faits pour l'optimisation. Entre autres, il y a certains faits sur un programme qui sont connus du programmeur et qui ne sont pas explicites dans le programme. On pourrait y remédier en incluant des assertions, tout comme on insère des déclarations de type pour les variables.

Exemple :

1. `pour i de 0 à 10 faire a[i] := i ; fin ;`
2. `pour i de 11 à 10000 faire a[i] := 0 ; fin ;`
3. `a[(a[j] + 1) * a[j + 1]] := j ;`
4. `si a[j * j + 2 * j + 1] ≠ a[j] aller à étiquette ;`

Pour un programme, il est important de savoir que `1 ≤ j < 99` (à charge éventuellement au système de le déduire à partir d'autres assertions), parce qu'on peut alors remplacer (4) par `(4') :`

`(4') si j < 10 aller à étiquette ;`

Cette insertion d'assertions peut donc servir de guide à une analyse automatique des programmes essentielle pour l'optimisation (mais également pour la mise au point, la documentation automatique, la décompilation, l'adaptation à un changement d'environnement d'exécution...).
Dans tous les exemples que nous avons pris, (équivalence de définitions de données, équivalence de définition d'opérateurs) nous avons conduit cette analyse sémantique à la main.

La possibilité de son automation, nous semble conditionner les progrès dans le domaine de l'optimisation de l'implantation automatisée d'un langage étant donnée sa définition, aussi bien que dans celui de l'optimisation des programmes [41].
An important encounter

- I do my military service as a scientist with Jean Ichbiah
- Work on the revision of LIS (ancestor of Green → ADA)
- Will always be a very strong support on our work
1973: Dijkstra’s handmade proofs

Radhia Rezig: attends Marktoberdorf summer school, July 25–Aug. 4, 1973

- Dijkstra shows program proofs (inventing elegant backward invariants)

- Radhia has the idea of automatically inferring the invariants by a backward calculus to determine intervals
Radhia Rezig shows her interval analysis ideas to Patrick Cousot

Patrick very critical on going backwards from $[-\infty, +\infty]$ and claims that going forward would be much better

Patrick also very skeptical on forward termination for loops

Radhia comes back with the idea of extrapolating bounds to $\pm\infty$ for the forward analysis

We discover widening $=\text{induction in the abstract}$ and that the idea is very general
Notes of Radhia Rezig
on forward iteration
from \( \square = \bot^{(1)} \) versus
backward iteration
from \([-\infty, +\infty]^{(2)}\)

________________________

(1) i.e. forward least fixed point
(2) i.e. backward greatest fixed point
First seminar in Grenoble: a warm welcome

- “Not all functions are increasing, for example, \( \sin \)”
- “This is woolly” (fumeux)
- “This will have applications in hundred years”
The IRIA-SESORI contract (1975–76)

- The project evaluator (Bernard Lohro) points us to the literature on constant propagation in data flow analysis (Kildall thesis).
- It appears that it is completely related to some of ours ideas, but *a.o.*
  - We are **not syntactic** (as in boolean DFA)
  - We have **no need for some hypotheses** (e.g. distributivity not even satisfied by constant propagation!)
  - We have **no restriction to finite lattices** (or ACC)
  - We have **no need of an a-posteriori proof of correctness** (e.g. with respect to the MOP as in DFA)
  - ...
The IRIA-SESORI contract (1975-76)

- **New general ideas**
  - The formal notions of *abstraction/approximation*
  - The formal notion of *abstract induction* (widening) to handle *infiniteness* and/or *complexity*
  - The *systematic correct design* with respect to a formal semantics
  - ...
The IRIA-SESORI contract (1975-76)

- The first contract report:

VERIFICATION STATIQUE DE LA COHERENCE
DYNAMIQUE DES PROGRAMMES

1) - PRESENTATION DU PROBLEME -

La notion de type ou de mode dans les langages de programmation, permet générale-
ment une vérification statique (à la compilation), de la cohérence dynamique
(à l'exécution) des programmes. Toute valeur a un mode unique, qui caractérise
les actions qui peuvent être exécutées sur ou avec elle. En ALGOL 68 par
exemple, les déclarations :

```
struct (int jour, string mois, int an) t = (11, "juillet", 1971);
struct (long liste, int u);
```

permettent au compilateur de détecter que les écritures

```
mois of u
```

ou

```
c of t
```

sont erronées.

Pour certains types toutefois, certaines opérations sur des objets de ce type
ne sont pas définies pour certaines valeurs ayant ce type. Par exemple, en
PASCAL [1] on peut déclarer :

```
type personne = record
    nom : alfa;
    père, mère : & personne;
end;

var x, y, z : & personne;
```

mais l'opération `+` n'est définie que si x `#` nil. Les techniques de
compilation actuelles ne permettent pas de déterminer que x n'est pas nil
quand on accède à un champ de l'enregistrement repéré par x. De ce fait,
certains compilateurs insèrent des tests dans le code généré pour détecter
ce genre d'erreur à l'exécution du programme.

- La plupart des compilateurs utilisent le mécanisme de protection mémoire
  (à un coût pratiquement nul), mais cette technique n'est pas utilisable pour
des langages d'implémentation de système, comme LISP.
The first reports (1975)

The first abstract interpreter with widening (as of 23 Sep. 1975)

The first research report (Nov. 1975)
The first publication (1976)

The first publication (ISOP II, Apr. 76)

In high level languages, compile time verifications are usually incomplete, and dynamic coherence checks must be inserted in object code. For example, in PASCAL one must dynamically verify that the values assigned to subranges type variables, or index expressions lies between two bounds, or that pointers are not nil. We present here a general algorithm allowing most of these certifications to be done at compile time. The static analysis of programs we do consists of an abstract evaluation of these programs, similar to those used by NAIR for verifying the type of expressions in ALGOL 60 [6], by SINTZOFF for verifying that a module corresponds to its logical specification [9], by KILDAL for global program optimization [5], by WEINREIT for extracting properties of programs, [9], by KARR for finding affine relationships among variables of a program [4], by SCHWARTZ for automatic data structure choice in SETL [8].

The essential idea is that, when doing abstract evaluation of a program, abstract values are associated with variables instead of the "concrete" values used while actually executing. The basic operations of the language are interpreted accordingly and the abstract interpretation then consists in a transitive closure mechanism. One may consider abstract values belonging to no finite sets, but the properties of the transitive closure algorithm are chosen such that the abstract interpretation stabilizes after finitely many steps.

* Attaché de Recherche au CNRS, Laboratoire Associé N°7.
** This work was supported by IRIA-SEGRI under grant 75-O35.
Maturation (1976 – 77): from an algorithmic to an algebraic point of view

- **Narrowing**, duality
- **Transition systems**, traces
- **Fixpoints**, chaotic/asynchronous iterations, approximation
- **Abstraction**, formalized by Galois connections, closure operators, Moore families, ...;
- Numeric and symbolic **abstract domains**, combinations of abstract domains
- Recursive **procedures**, relational analyses, heap analysis
- etc.
A Visitor

- Hi, I am Steve Warshall
- The theorem?
- Yes
- Steve Schuman told me you are doing interesting work
- ...
- You should publish in Principles of Programming Languages.

---

Topology, higher-order fixpoints, operational/summary/... analysis

Galois connections, closure operators, Moore families, ideals,...
CONSTRUCTIVE VERSIONS OF TARSKI’S FIXED POINT THEOREMS

PATRICK COUSOT and RADHIA COUSOT

Let \( F \) be a monotone operator on the complete lattice \( L \) into itself. Tarski’s lattice theoretical fixed point theorem states that the set of fixed points of \( F \) is a nonempty complete lattice for the ordering of \( L \). We give a constructive proof of this theorem showing that the set of fixed points of \( F \) is the image of \( L \) by a lower and an upper preclosure operator. These preclosure operators are the composition of lower and upper closure operators which are defined by means of limits of stationary transfinite iteration sequences for \( F \). In the same way we give a constructive characterization of the set of common fixed points of a family of commuting operators. Finally we examine some consequences of additional semi-continuity hypotheses.

1. Introduction

Let \( L(\subseteq, \top, \bot, \cap, \cup) \) be a nonempty complete lattice with partial ordering \( \subseteq \), least upper bound \( \lor \), greatest lower bound \( \land \). The supremum \( \top \) of \( L \) is \( \bot \lor L \). (Birkhoff’s standard reference book [3] provides the necessary background material.) Set inclusion, union and intersection are respectively denoted by \( \subseteq, \lor \) and \( \land \).

Let \( F \) be a monotone operator on \( L(\subseteq, \top, \bot, \cap, \cup) \) into itself (i.e., \( FX \subseteq FY \Rightarrow F(X) \subseteq F(Y) \)).

The fundamental theorem of Tarski [19] states that the set \( \text{fp}(F) \) of fixed points of \( F \) (i.e., \( \text{fp}(F) = \{ X \in L; F(X) = X \} \)) is a nonempty complete lattice with ordering \( \subseteq \). The proof of this theorem is based on the definition of the least fixed point \( (\text{up}(F)) \) of \( F \) by \( \text{up}(F) = \cap \{ X \in L; F(X) \subseteq X \} \). The least upper bound of \( S \subseteq \text{fp}(F) \) is \( \text{up}(F) \) is the least fixed point of the restriction of \( F \) to the complete lattice \( (X \in L; (\subseteq S) \subseteq X) \). An application of the duality principle completes the proof.

This definition is not constructive and many applications of Tarski’s theorem (specially in computer science (Cousot [5]) and numerical analysis (Aumann [2]) use the alternative characterization of \( \text{fp}(F) \) as \( \{ F^n(X) ; X \in L \} \). This iteration scheme which originates from Kleene [10]’s first recursion theorem and which was used by Tarski [19] for complete morphisms, has the drawback to require the additional assumption that \( F \) is semi-continuous \( (F(S) \subseteq \cup F(S)) \) for every increasing nonempty chain \( S \), see e.g., Kolodner [11].

And a bit of mathematics...
On submitting to POPL

- For POPL’77, we submit (on Aug. 12, 1976) copies of a two-hands written manuscript of 100 pages. The paper is accepted!
On abstracting: transition system

Reachability semantics is an abstraction of the relational semantics (in PC’s thesis, 21 march 1978 also § 3 of POPL’79)

3.1.3 L’approche du point fixe à l’étude du comportement d’un système dynamique discret

DEFINITION 3.1.3.0.1

\[ wp \in (((S\times S) \rightarrow B) \rightarrow ((S \rightarrow B) \rightarrow (S \rightarrow B))) \]
\[ = \lambda \theta. \lambda \alpha. \{ e_1, e_2 \in S : \theta(e_1, e_2) \land \alpha(e_2) \} \]

Partant du fait que \( \tau^* = eq \cup \tau \circ \tau = eq \cup \tau \circ \tau^* \), nous obtiendrons \( wp(\tau^*) \) et \( sp(\tau^*) \) comme points fixes d’une équation.

THEOREME 3.1.3.0.3

(a) - \((S\times S) \rightarrow B\) est un treillis booléen complet,
(b) - Soient \( a, b \in (S\times S) \rightarrow B \) alors \( \lambda \alpha. [a \text{ ou } b \circ \alpha] \) et \( \lambda \alpha. [a \text{ ou } \alpha \circ b] \) sont des morphismes complets pour le disjonction,
(c) - Soient \( \tau \in (S\times S) \rightarrow B \) et \( eq \) la relation d’égalité alors \( \tau^* = lfp(\lambda \alpha. [eq \text{ ou } \alpha \circ \alpha]) = lfp(\lambda \alpha. [eq \text{ ou } \tau \circ \alpha]). \)

THEOREME 3.1.3.0.6.

Quels que soient \( a, b \in (S\times S) \rightarrow B \) et \( \beta \in (S \rightarrow B) \) nous avons:
\[ wp(lfp(\lambda \alpha. [a \text{ ou } b \circ \alpha]))(\beta) \]
\[ = lfp(\lambda \beta. [wp(\alpha)(\beta) \text{ ou } wp(\beta)(\alpha)]) \]
\[ = wp(b)(wp(\alpha)(\beta)) \text{ nouvelles} \]
\[ sp(lfp(\lambda \alpha. [a \text{ ou } \alpha \circ b]))(\beta) \]
\[ = lfp(\lambda \beta. [sp(\alpha)(\beta) \text{ ou } sp(\beta)(\alpha)]) \]
\[ = wp(b)(sp(\alpha)(\beta)) \text{ nouvelles} \]

\textbf{Fixeux:} Posons \( h = \lambda \theta. [wp(\theta)(\beta)] \), \( f = \lambda \alpha. [a \text{ ou } b \circ \alpha] \) et \( g = \lambda \alpha. [wp(\alpha)(\beta) \text{ ou } wp(\beta)(\alpha)] \) et montrons que \( ho = g \circ h. \)

\textbf{Fixpoint abstraction under commutativity with abstraction \( h \)}

\textbf{Iterative fixpoint computation}

\textbf{backward reachability}

\textbf{forward reachability}

\textbf{abstract transformer}

\textbf{concrete transformer}

\textbf{Fixpoint reflexive transitive closure}

\textbf{i.e. pre}

\textbf{i.e. post transformer}
On convincing ...

• During PC’s thesis defense, it was suggested that abstraction/approximation is useless since computers are finite and executions are timed-out (so, the second part of the thesis on fixpoint approximation/widening/narrowing/... is superfluous!)

• Fortunately we do not listen (otherwise we would have invented enumeration methods that fail to scale)

• On the contrary, in 1978, during a seminar at Harvard (1), G. Birkhoff appears interested, according to his questions & feedback, in the effective computational aspects of lattice fixpoint theory

(1) invited by Ed. Clarke.
The principles (1977–79) are lasting

- Define the semantics (operational, denotational, axiomatic, ...) of the programming language (as a ... / trace semantics / transition system / transformers / ...)
- Define the strongest property of interest (also called the collecting semantics)
- Express this collecting semantics in fixpoint (constraint, rule-based,...) form
- Define the abstraction/concretization compositionally (by composition of elementary abstractions and abstraction constructors/functors)
- Design the abstract proof / analysis semantics by calculus using [structural] abstraction i.e. abstract domain + abstract fixpoint
- Combine abstractions (e.g. reduced product)
Abstract interpretation: Research takes time
• Type checking and inference is an abstract interpretation:

3. Type checking: a compile-time example of abstract interpretation.

Summary: It is shown that our model defines the "types of expressions" in terms of the language defined "types of values". It is shown that
- "Type checking" simply consists in verifying that the program declarations lead to a valuable solution of the system of equations associated with a program (Naur [1965], Sintzoff [1972]).
- whereas, "Type discovery" consists in finding a solution to the system (Tennenbaum [1974]).
Types as Abstract Interpretations

(Invited Paper)

Patrick Cousot
LIENS, Ecole Normale Superieure
45, rue d’Ulm
75230 Paris cedex 05 (France)
cousot@dmi.ens.fr, http://www.ens.fr/~cousot

Abstract
Starting from a denotational semantics of the eager untyped lambda-calculus with explicit runtime errors, the standard collecting semantics is derived as specifying the strongest program properties. By a first abstraction, a new sound type collecting semantics is derived in compositional fix-point form. Then by successive (semi-dual) Galois connections between abstractions, type systems and/or type inference algorithms are designed as abstract semantics or abstract interpreters approximating the type collecting semantics. This leads to a hierarchy of type systems, which is part of the lattice of abstract interpretations of the untyped lambda calculus. This hierarchy includes two new “la Church/Curry polytype systems. Abstractions of this polytype semantics lead to classical Milner/Mycroft and Damas/Milner polymorphic type schemes. Church/Curry monotypes and Hindley principal typing algorithm. This shows that types are abstract interpretations.

1 Introduction
The leading idea of abstract interpretation [6, 7, 9, 12] is that program semantics, proof and static analysis methods have common structures which can be exhibited by abstraction of the structure of run-time computations. This leads to an organisation of the more or less approximate or refined semantics into a lattice of abstract interpretations. This unifying point of view allows for a synthetic understanding of the relationship between these seemingly different approaches to program correctness and optimization.

2 Syntax
The syntax of the untyped eager lambda calculus is:

\[
x, f, \ldots \in \mathcal{X} : \text{program variables}
\]

\[
e \in \mathcal{E} : \text{program expressions}
\]

\[
e ::= x | \lambda x : e | e_1 (e_2) | \mu f. \lambda x : e | 1 | e_1 \cdot e_2 | (e_1 ? e_2 : e_3)
\]

\[
\lambda x : e \text{ is the lambda abstraction and } e_1 (e_2) \text{ the application.}
\]

\[
\mu f. \lambda x : e \text{ is the function } f \text{ with formal parameter } x \text{ is defined recursively. } (e_1 ? e_2 : e_3) \text{ is the test for zero.}
\]

3 Denotational Semantics
The semantic domain \( \mathcal{S} \) is defined by the following equations [26]:

\[
\mathcal{W} \mathrel{\triangleq} \{ \omega \} \quad \text{wrong}
\]

\[
x \in \mathbb{Z} \mathrel{\triangleq} \mathcal{S}, \quad \text{integers}
\]

\[
u, \varphi \in \mathbb{U} \mathrel{\triangleq} \mathcal{S} \cup [\mathbb{R} \rightarrow \mathbb{U}], \quad \text{values}
\]

\[
\mathbb{R} \in \mathcal{S} \mathrel{\triangleq} \mathcal{S} \rightarrow \mathcal{U} \quad \text{environments}
\]

\[
\phi \in \mathcal{S} \mathrel{\triangleq} \mathcal{R} \rightarrow \mathcal{U} \quad \text{semantic domain}
\]

where \( \omega \) is the wrong value, \( \bot \) denotes non-termination, \( D_+ \) is the lift of domain \( D \) (with up injection \( \bot \in D \rightarrow D_+ \) and partial down injection \( \bot (\bullet) \in D_+ \rightarrow D_+ \)), \( D_\bot \) is the coalesced sum of domains \( D_1 \) and \( D_2 \) (with left and right injections \( \bullet : D_1 \rightarrow D_\bot \) and \( \vdash : D_2 \rightarrow D_\bot \), \( \Omega = \{ \omega : \mathcal{W} \}, \) and \( [D_1 \rightarrow D_2] \) is the domain of continuous, \( \Delta \)-strict functions from \( D_1 \) into \( D_2, \) \( \sqcup \) is the computational ordering on \( \mathbb{U} \) and \( \sqcap \) is the least upper bound (lub) of increasing chains.

In the metalanguage for defining the denotational semantics below, \( \lambda x : \ldots \) or \( \lambda x \in \mathcal{S} \ldots \) is the lambda abstraction. \( (\ldots ? \ldots) \) is the conditional expression.

\[
\text{POPL 97, Paris, France}
\]

© 1997 ACM 0-89791-853-3/96/01 ...$3.50
Probabilistic static analysis

Applying Kirchhoff laws, we get the system of equations:

\[ C_0 = 1 \]
\[ C_1 = C_0 \]
\[ C_2 = C_1 + C_3 \]
\[ C_3 = C_2 - p \]
\[ C_4 = C_2 * (1-p) \]
\[ C_5 = C_4 * q \]
\[ C_6 = C_5 - (1-q) \]
\[ C_7 = C_6 * (1-q) \]

Thus the limit of the sequence leads for \( C_e \) to an infinite series, which limit is \( 1/p \):

\[ \frac{1}{p} = \frac{1}{1 - (1-p)} = 1 + (1-p) + \ldots + (1-p)^n + \ldots \]
Abstract. Abstract interpretation has been widely used for verifying properties of computer systems. Here, we present a way to extend this framework to the case of probabilistic systems.

The probabilistic abstraction framework that we propose allows us to systematically lift any classical analysis or verification method to the probabilistic setting by separating in the program semantics the probabilistic behavior from the (non-)deterministic behavior. This separation provides new insights for designing novel probabilistic static analyses and verification methods.

We define the concrete probabilistic semantics and propose different ways to abstract them. We provide examples illustrating the expressiveness and effectiveness of our approach.
Termination

Abstract interpretation of programs is shown to be a suitable means to statically analyse their weak or strong properties.

Proofs of program termination,
(Manna and Vuillemin [1972], Sintzoff [1976a], Sites [1974])
Termination

POPL 2012:

An Abstract Interpretation Framework for Termination

Patrick Cousot
CNRS, École Normale Supérieure, and INRIA, France
Courant Institute *, NYU, USA
cousot@ens.f, pcousot@cims.nyu.edu

Radhia Cousot
CNRS, École Normale Supérieure, and INRIA, France
rcousot@ens.fr

Abstract Proof, verification and analysis methods for termination all rely on two induction principles: (1) a variant function or induction on data ensuring progress towards the end and (2) some form of induction on the program structure.

The abstract interpretation design principle is first illustrated for the design of new forward and backward proof, verification and analysis methods for safety. The safety collecting semantics defining the strongest safety property of programs is first expressed in a constructive fixpoint form. Safety proof and checking/verification methods then immediately follow by fixpoint induction. Static analysis of abstract safety properties such as invariance are constructively designed by fixpoint abstraction (or approximation) to (automatically) infer safety properties. So far, no such clear design principle did exist for termination so that the existing approaches are scattered and largely not comparable with each other.

For (1), we show that this design principle applies equally well to potential and definite termination. The trace-based termination collecting semantics is given a fixpoint definition. Its abstraction yields a fixpoint definition of the best variant function. By further abstraction of this best variant function, we derive the Floyd/Turing termination proof method as well as new static analysis methods to effectively compute approximations of this best variant function.

For (2), we introduce a generalization of the syntactic notion of structural induction (as found in Hoare logic) into a semantic structural induction based on the new semantic concept of inductive trace cover covering execution traces by segments, a new basis for formulating program properties. Its abstractions allow for generalized recursive proof, verification and static analysis methods by induction on both program structure, control, and data. Examples of particular instances include Floyd’s handling of loop cut-points as well as nested loops, Burstall’s intermittent assertion total correctness proof method, and Podelski-Rybalchenko transition invariants.
Denotational Semantics

---

**ABSTRACT**

Abstract interpretation of programs is shown to be a suitable means to statically analyse their weak or strong properties.

---

Derivation of the partial function computed by a program; McCARTHY [1963a,b], SCOTT and STRACHEY [1971]
Hierarchy of semantics

• POPL 1992:

Inductive Definitions, Semantics and Abstract Interpretation*

Patrick Cousot
LIENS, École Normale Supérieure
45, rue d’Ulm
75230 Paris cedex 05 (France)
cousot@ensi.ens.fr

Radhia Cousot
LIX, École Polytechnique
91128 Palaiseau cedex (France)
cousot@polytechnique.fr

Abstract

We introduce and illustrate a specification method combining rule-based inductive definitions, well-founded induction principles, fixed-point theory and abstract interpretation for general use in computer science. Finite as well as infinite objects can be specified, at various levels of details related by abstraction. General proof principles are applicable to prove properties of the specified objects.

The specification method is illustrated by introducing $G^\infty$SOS, a structured operational semantics generalizing Plotkin’s [28] structured operational semantics (SOS) so as to describe the finite, as well as the infinite behaviors of programs in a uniform way and by constructively deriving inductive presentations of the other (relational, denotational, predicate transformers, …) semantics from $G^\infty$SOS by abstract interpretation.
Hierarchy of semantics

TCS 2002:

Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation

Patrick Cousot

Département d’Informatique, École Normale Supérieure, 45 rue d’Ulm, 75230 Paris cedex 05, France, Patrick.Cousot@ens.fr, http://www.di.ens.fr/~cousot

We construct a hierarchy of semantics by successive abstract interpretations. Starting from the maximal trace semantics of a transition system, we derive the big-step semantics, termination and nontermination semantics, Plotkin’s natural, Smyth’s demoniac and Hoare’s angelic relational semantics and equivalent nondeterministic denotational semantics (with alternative powerdomains to the Egli-Milner and Smyth constructions), D. Scott’s deterministic denotational semantics, the generalized and Dijkstra’s conservative/liberal predicate transformer semantics, the generalized/total and Hoare’s partial correctness axiomatic semantics and the corresponding proof methods. All the semantics are presented in a uniform fixpoint form and the correspondences between these semantics are established through composable Galois connections, each semantics being formally calculated by abstract interpretation of a more concrete one using Kleene and/or Tarski fixpoint approximation transfer theorems.
Bi-inductive Structural Semantics

Patrick Cousot

Département d’informatique, École normale supérieure, 45 rue d’Ulm,
75230 Paris cedex 05, France

Radhia Cousot

CNRS & École polytechnique, 91128 Palaiseau cedex, France

Abstract

We propose a simple order-theoretic generalization, possibly non monotone, of set-theoretic inductive definitions. This generalization covers inductive, co-inductive and bi-inductive definitions and is preserved by abstraction. This allows structural operational semantics to describe simultaneously the finite/terminating and infinite/diverging behaviors of programs. This is illustrated on grammars and the structural bifinitary small/big-step trace/relational/operational semantics of the call-by-value λ-calculus (for which co-induction is shown to be inadequate).

Key words: fixpoint definition, inductive definition, co-inductive definition, bi-inductive definition, non-monotone definition, grammar, structural operational semantics, SOS, trace semantics, relational semantics, small-step semantics, big-step semantics, divergence semantics.
CHAPTER 12

Invariance Proof Methods And Analysis Techniques For
Parallel Programms

Patrick Cousot
Universite de Nantes
Faculte des Sciences
Ecole Normale Supérieure

Radhia Cousot
Centre de Recherche en Informatique de Nancy

A. Introduction

We propose a unified approach for the study, comparison and systematic construction of program proof and analysis methods. Our presentation will be mostly informal but the underlying formal theory can be found in Cousot and Cousot [1980, 1984], and Cousot, P. [1984].

4.3.1.7 Dérivation de conditions de vérification correctes


t et une correspondance \( e(r, s) \) entre \( r \) et \( s \), toute preuve.

\[ f \in C(f) \]

telle que

\[ C \rightarrow C' \]
Parallelism

- POPL 2017:

Ogre and Pythia: An Invariance Proof Method for Weak Consistency Models

Jade Alglave
University College London
Microsoft Research Cambridge, UK
jaalglav@microsoft.com, j.alglave@ucl.ac.uk

Patrick Cousot
New York University, USA
emer. École Normale Supérieure, PSL, France
pcousot@cims.nyu.edu, cousot@ens.fr
Abstract interpretation: Industrialization
Industrialization

• Very first industrial implementation:

  The **interval analysis** was implemented in the **AdaWorld compiler** for IBM PC 80286 by **J.D. Ichbiah** and his **Alsys SA corporation** team in 1980–87.
Warm welcome

- **Real-time software development companies:** we have to pay for this new option of the ADA compiler, but:
  - The machine code size is significantly reduced → we cannot sell as much memory as we did before;
  - Many bugs are found at compile time → we make less money with our debugging services.
• Astrée sold by AbsInt:
Abstract interpretation based static analyzers

- **Ait** [www.absint.com/ait/](http://www.absint.com/ait/), **StackAnalyzer** [www.absint.com/stackanalyzer](http://www.absint.com/stackanalyzer) from AbSint
- **Polyspace static analysis** [www.mathworks.com/products/polyspace.html](http://www.mathworks.com/products/polyspace.html)
- **Julia** (Java) [www.juliasoft.com](http://www.juliasoft.com)
- **Ikos**, NASA [ti.arc.nasa.gov/opensource/ikos/](http://ti.arc.nasa.gov/opensource/ikos/)
- **Clousot** for code contract, Microsoft, [github.com/Microsoft/CodeContracts](http://github.com/Microsoft/CodeContracts)
- **Infer** (Facebook) [http://fbinfer.com](http://fbinfer.com)
- **Zoncolan** (Facebook)
- Google
- ...
Abstract interpretation: Prospective
The future is hard to predict

• From my thesis in 1978:

Le concept de système dynamique discret est évidemment très général. Il s'applique aussi bien aux systèmes informatiques qu'économiques ou biologiques, à condition que le modèle du système étudié soit à évolution discrète dans le temps. En particulier, les systèmes dynamiques discrets sont des modèles des programmes aussi bien séquentiels que parallèles.

computer, economical and biological systems
sequential and parallel programs
The future is hard to predict

- From “30 years of Abstract Interpretation”:

  Programming
  - The evolution of programming languages and programming assistance systems has greatly helped to considerably **speed up the development** and **scale up the size** of conceivable programs
  - Software **quality** remains much far beyond, essentially because it is anti-economical
  - ... until the next catastrophe, evolution of the standards, revolution of the customers, or new laws holding computer scientists accountable for bugs

  Abstract interpretation
  - Beyond programming, abstraction is the only way to apprehend **complex systems**
  - Therefore, the **scope of application** of abstract interpretation ideas is large
  - Over 30 years, abstract interpretation theory, practice and achievements have grown despite trends and evanescent applications
  - Hopefully, abstract interpretation will continue to be useful in the future

  Formal methods
  - Formal methods might then become **profitable** at every stage of program design
  - The winners, if any, will definitely have to **scale up**, at a reasonable cost
  - Up to now, research has mainly concentrated on easy avenues with short-term rewards
  - Small groups cannot make it, large groups fail to share common interests
  - There is still a long long way to go

THE END

Many thanks to all of you who contributed to abstract interpretation!
The future is hard to predict

- From the Dagstuhl Seminar “Formal Methods — Just a Euro-Science?” in December 2010:

  - More **properties**:
    - Security (not dynamically checkable)
    - ...
  
  - More **systems** and **tools**:
    - Parallel and distributed systems,
    - Cyber-physical (continuous+discrete)
    - Biological, financial, ...
  
  - Better **practices**:
    - Verification from design to implementation
Hopes (10 years)

- Complex data structures (libraries like for numerical domains)
- Program security
- Parallel & distributed systems, weak consistency models
1. The semantics is specified structurally and compositionally

2. The abstraction is specified by composition of Galois connections

POPL 2014:

A Galois Connection Calculus for Abstract Interpretation*

Patrick Cousot
CIMS**, NYU, USA pcousot@cims.nyu.edu

Radhia Cousot
CNRS Emeritus, ENS, France rcousot@ens.fr

3. The calculational design of the abstract interpreter is supported by libraries and tools

4. All modular and compositional
4. The design of static analyzers is computer-assisted by automatic composition of certified public-domain modules for:

- Abstract domains
- Syntax and semantics to fixpoint equations
- Parallel/distributed fixpoint solvers (direct or with convergence acceleration)
- User-interface automatic design
- Automatic fixing of errors
The End