"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

---

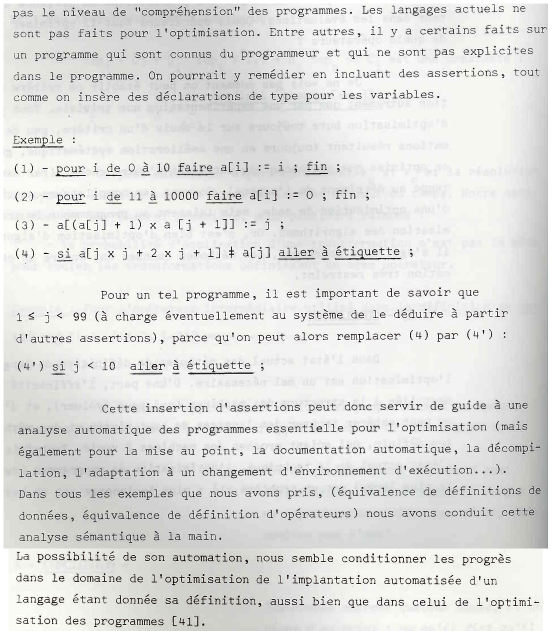
- Radhia Rezig. *Application de la méthode de précédence totale à l'analyse d'Algol 68*, Master thesis, Université Joseph Fourier, Grenoble, France, September 1972.
- Patrick Cousot. *Un analyseur syntaxique pour grammaires hors contexte ascendant sélectif et général.* In Congrès AFCET 72, Brochure 1, 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")

- Patrick Cousot. *Définition interprétative et implantation de languages de programmation.* Thèse de Docteur Ingénieur en Informatique, Université Joseph Fourier, Grenoble, France, 14 Décembre 1974 (submitted in 1973 but defended after finishing military service with J.D. Ichbiah at CII).

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) x a [j + 1]] := j ;

(4) - si a[j x j + 2 x j + 1] ≠ a[j] aller à étiquette ;

Pour un tel 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].

Intervals →

Assertions →

Static analysis →

An important encounter

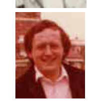
- I do my military service as a scientist with Jean Ichbiah
- Work on the revision of LIS (ancestor of Green → ADA)
- Will always be a very strong support on our work

1973: Dijkstra's handmade proofs

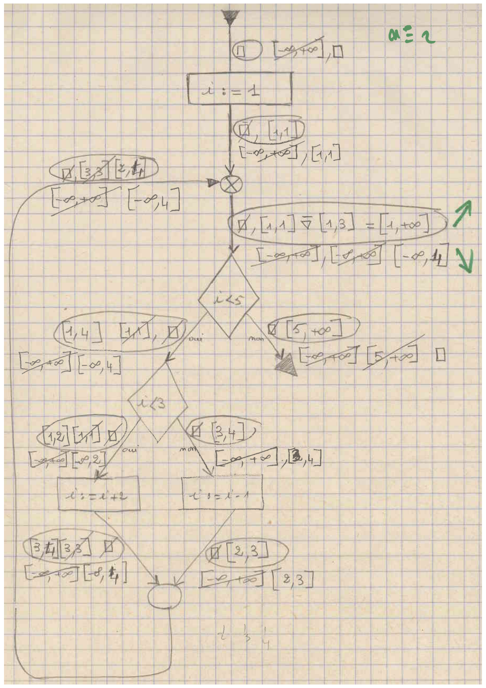
- Radhia Rezig: attends Marktoberdorf summer school, July 25–Aug. 4, 1973
  - ➡ Dijkstra shows program proofs (*inventing* elegant backward invariants)

  - ➡ Radhia has the idea of automatically *inferring* the invariants by a backward calculus to determine intervals

1974: origin

- 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 = 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)

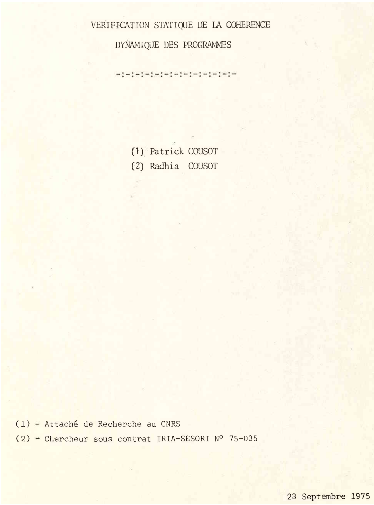
- The project evaluator (Bernard Lohro) points us to the literature on constant propagation in data flow analysis (Kildall thesis).
- It appears that it is completely related to some of ours ideas, but *a.o.*
  - We are not syntactic (as in boolean DFA)
  - We have no need for some hypotheses (e.g. distributivity 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

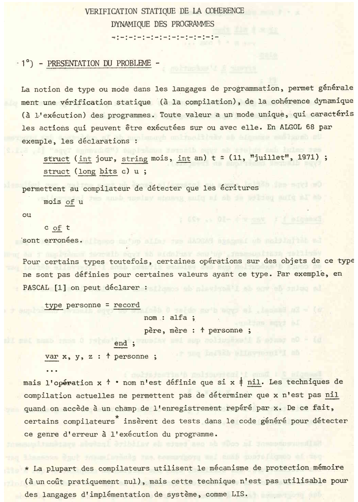
DYNAMIQUE DES PROGRAMMES

-:-:-:-:-:-:-:-:-:-:-:-:-

(1) Patrick COUSOT
(2) Radhia COUSOT

(1) - Attaché de Recherche au CNRS
(2) - Chercheur sous contrat IRIA-SESORI N° 75-035

23 Septembre 1975



VERIFICATION STATIQUE DE LA COHERENCE

DYNAMIQUE DES PROGRAMMES

-:-:-:-:-:-:-:-:-:-:-:-:-

1°) - PRESENTATION DU PROBLEME -

La notion de type ou 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 caractéris 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 bits c) u ;
```

permettent au compilateur de détecter que les écritures

```
mois of u
```

ou

```
c of t
```

sont erronées.

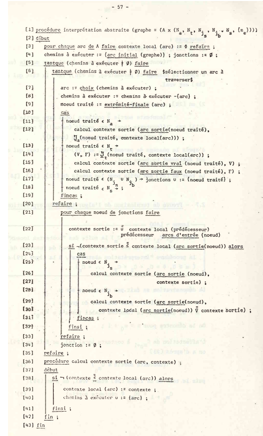
Pour certains types toutefois, certaines opérations sur des objets de ce type ne sont pas définies pour certaines valeurs ayant ce type. Par exemple, en PASCAL [1] on peut déclarer :

```
type personne = record
                    nom : alfa ;
                    père, mère : ↑ personne ;
                end ;
var x, y, z : ↑ personne ;
...
```

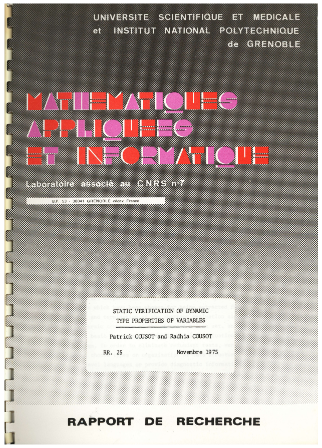
mais l'opération x ↑ • nom n'est définie que si x ≠ nil. Les techniques de compilation actuelles ne permettent pas de déterminer que x n'est pas nil quand on accède à un champ de l'enregistrement repéré par x. De ce fait, certains compilateurs* insèrent des tests dans le code généré pour détecter ce 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 LIS.

The first reports (1975)



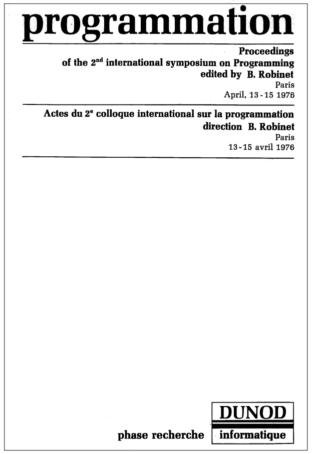
The first abstract interpreter with widening (as of 23 Sep. 1975)

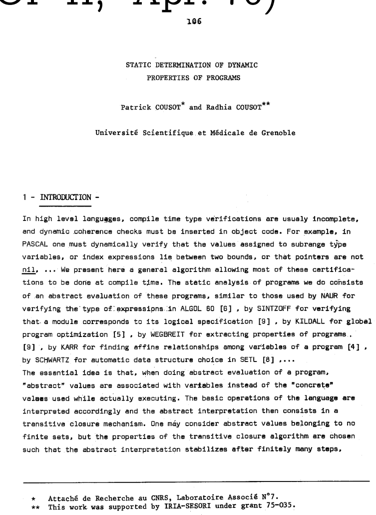


The first research report (Nov. 1975)

The first publication (1976)

- The first publication (ISOP II, Apr. 76)





106

STATIC DETERMINATION OF DYNAMIC
PROPERTIES OF PROGRAMS

Patrick COUSOT* and Radhia COUSOT**

Université Scientifique et Médicale de Grenoble

1 - INTRODUCTION -

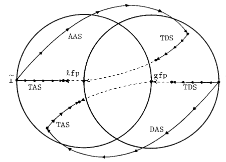
In high level languages, compile time type verifications are usualy incomplete, and dynamic coherence checks must be inserted in object code. For example, in PASCAL one must dynamically verify that the values assigned to subrange type variables, or index expressions lie between two bounds, or that pointers 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 NAUR for verifying the type of expressions in ALGOL 60 [6] , by SINTZOFF for verifying that a module corresponds to its logical specification [9] , by KILDALL for global program optimization [5] , by WEGBREIT 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-SESORI under grant 75-035.

cited by 551 Google scholar

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.

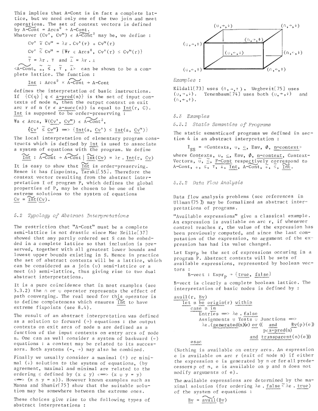


A Visitor

- 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.

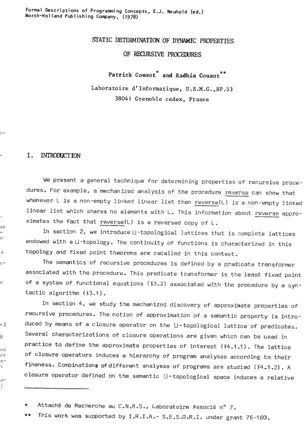
- Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM, 9(1):11–12* , *January 1962.*

POPL'77 FDPC'77 POPL'79



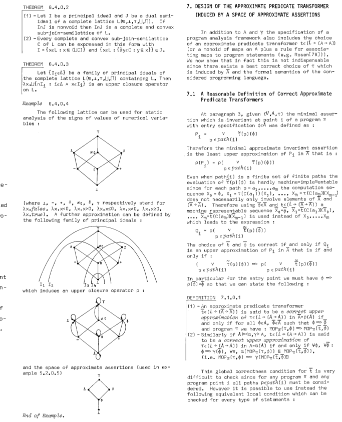
On this page: dual, conjugate and inversion:lfp/gfp wp/sp (i.e. pre/post) $\widetilde{\text{wp}}$/$\widetilde{\text{sp}}$)

Cited by 6381



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

Cited by 225

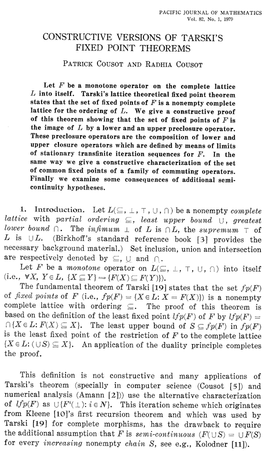


Galois connections, closure operators, Moore families, ideals,...

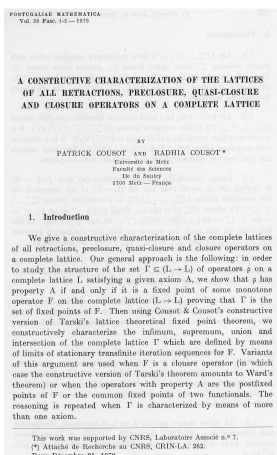
Cited by 1638

Google scholar

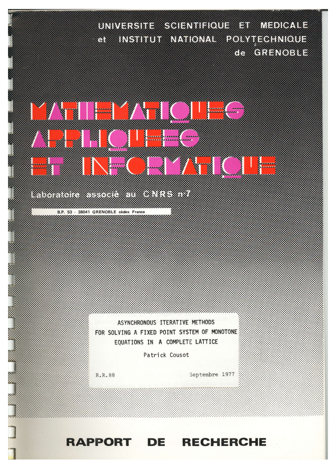
And a bit of mathematics...



cited by 208



cited by 31

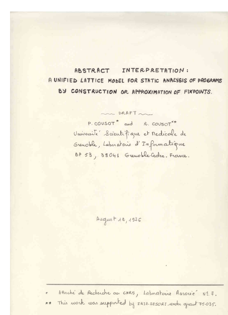


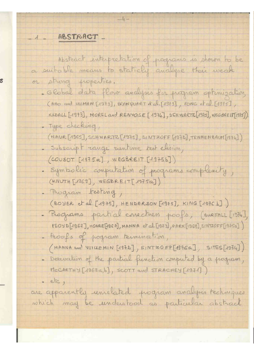
cited by 42

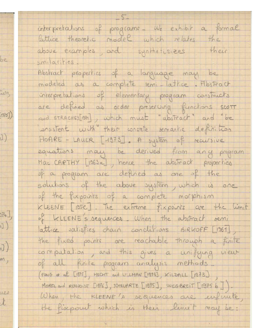
Google scholar

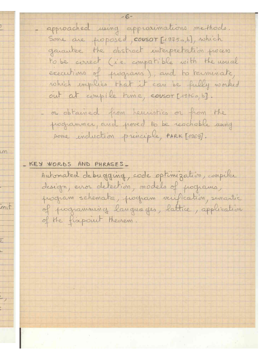
On submitting to POPL

- 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

i.e. pre

DEFINITION 3.1.3.0.1

$wp \in (((S \times S) \to B) \to ((S \to B) \to (S \to B)))$
$= \lambda\theta.\{\lambda\beta.[\lambda e_1.(\exists e_2 \in S : \theta(e_1,e_2) \text{ \underline{et} } \beta(e_2))]\}$

i.e. post transformer

DEFINITION 3.1.3.0.2

$sp \in (((S \times S) \to B) \to ((S \to B) \to (S \to B)))$
$= \lambda\theta.\{\lambda\beta.[\lambda e_2.(\exists e_1 \in S : \beta(e_1) \text{ \underline{et} } \theta(e_1,e_2))]\}$

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

THEOREME 3.1.3.0.3

(a) - $((S \times S) \to B)(\Rightarrow, \lambda(e_1,e_2).\underline{\text{faux}}, \lambda(e_1,e_2).\underline{\text{vrai}}, \underline{\text{OU}}, \underline{\text{ET}}, \underline{\text{non}})$ est un treillis booléen complet,

(b) - Soient $a, b \in ((S \times S) \to B)$ alors $\lambda\alpha.[a \text{ \underline{ou} } b \circ \alpha]$ et $\lambda\alpha.[a \text{ \underline{ou} } \alpha \circ b]$ sont des morphismes complets pour la disjonction,

(c) - Soient $\tau \in ((S \times S) \to B)$ et eq la relation d'égalité alors
$\tau^* = lfp(\lambda\alpha.[eq \text{ \underline{ou} } \alpha \circ \tau]) = lfp(\lambda\alpha.[eq \text{ \underline{ou} } \tau \circ \alpha])$.

fixpoint reflexive transitive closure

abstract transformer   concrete transformer

THEOREME 3.1.3.0.6.

Quels que soient $a, b \in ((S \times S) \to B)$ et $\beta \in (S \to B)$ nous avons:

- $wp(lfp(\lambda\alpha.[a \text{ \underline{ou} } b \circ \alpha]))(\beta)$
  $= lfp(\lambda\alpha.[wp(a)(\beta) \text{ \underline{ou} } wp(b)(\alpha)])$
  $= \underset{n \in \omega}{\underline{\text{OU}}}\ wp(b^n)(wp(a)(\beta))$
- $sp(lfp(\lambda\alpha.[a \text{ \underline{ou} } \alpha \circ b]))(\beta)$
  $= lfp(\lambda\alpha.[sp(a)(\beta) \text{ \underline{ou} } sp(b)(\alpha)])$
  $= \underset{n \in \omega}{\underline{\text{OU}}}\ sp(b^n)(sp(a)(\beta))$

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

fixpoint •••

backward reachability

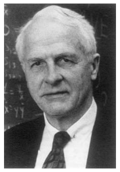
forward reachability

Fixpoint abstraction under commutativity with abstraction h

iterative fixpoint computation

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.

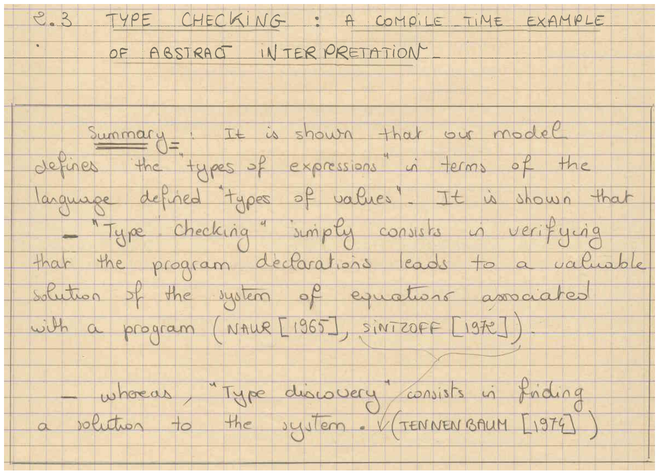
The principles (1977–79) are lasting

- 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

Typing

- Type checking and inference is an abstract interpretation:



2.3 TYPE CHECKING : A COMPILE TIME EXAMPLE OF ABSTRACT INTERPRETATION.

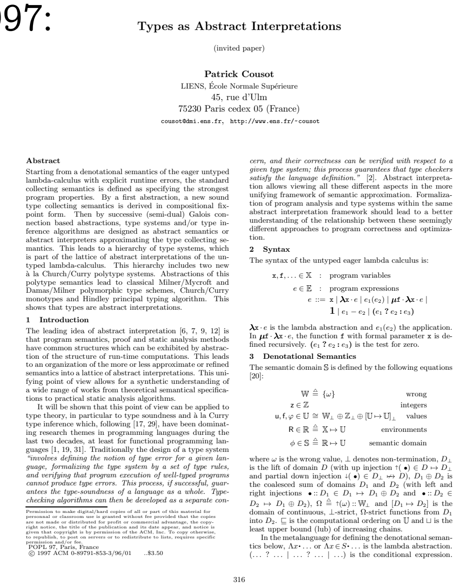
Summary: It is shown that our model defines the "types of expressions" in terms of the language defined "types of values". It is shown that

— "Type checking" simply consists in verifying that the program declarations leads 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])

Typing

- POPL 1997:



Types as Abstract Interpretations

(invited paper)

Patrick Cousot
LIENS, École Normale Supérieure
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 defined as specifying the strongest program properties. By a first abstraction, a new sound type collecting semantics is derived in compositional fixpoint form. Then by successive (semi-dual) Galois connection based 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 à 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 organization 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 a wide range of works from theoretical semantical specifications to practical static analysis algorithms.

It will be shown that this point of view can be applied to type theory, in particular to type soundness and à la Curry type inference which, following [17, 29], have been dominating research themes in programming languages during the last two decades, at least for functional programming languages [1, 19, 31]. Traditionally the design of a type system *"involves defining the notion of type error for a given language, formalizing the type system by a set of type rules, and verifying that program execution of well-typed programs cannot produce type errors. This process, if successful, guarantees the type-soundness of a language as a whole. Type-checking algorithms can then be developed as a separate concern, and their correctness can be verified with respect to a given type system; this process guarantees that type checkers satisfy the language definition."* [2]. Abstract interpretation allows viewing all these different aspects in the more unifying framework of semantic approximation. Formalization of program analysis and type systems within the same abstract interpretation framework should lead to a better 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:

$$\begin{aligned} \mathtt{x}, \mathtt{f}, \ldots \in \mathbb{X} &: \text{program variables} \\ e \in \mathbb{E} &: \text{program expressions} \\ e ::= {} & \mathtt{x} \mid \boldsymbol{\lambda}\mathtt{x} \cdot e \mid e_1(e_2) \mid \boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e \mid \\ & \mathbf{1} \mid e_1 - e_2 \mid (e_1 \mathbin{?} e_2 : e_3) \end{aligned}$$

$\boldsymbol{\lambda}\mathtt{x} \cdot e$ is the lambda abstraction and $e_1(e_2)$ the application. In $\boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e$, the function $\mathtt{f}$ with formal parameter $\mathtt{x}$ is defined recursively. $(e_1 \mathbin{?} e_2 : e_3)$ is the test for zero.

3 Denotational Semantics

The semantic domain $\mathbb{S}$ is defined by the following equations [20]:

$$\begin{aligned} \mathbb{W} &\triangleq \{\omega\} && \text{wrong} \\ \mathsf{z} &\in \mathbb{Z} && \text{integers} \\ \mathsf{u}, \mathsf{f}, \varphi \in \mathbb{U} &\cong \mathbb{W}_\perp \oplus \mathbb{Z}_\perp \oplus [\mathbb{U} \mapsto \mathbb{U}]_\perp && \text{values} \\ \mathsf{R} \in \mathbb{R} &\triangleq \mathbb{X} \mapsto \mathbb{U} && \text{environments} \\ \phi \in \mathbb{S} &\triangleq \mathbb{R} \mapsto \mathbb{U} && \text{semantic domain} \end{aligned}$$

where ω is the wrong value, $\perp$ denotes non-termination, $D_\perp$ is the lift of domain D (with up injection $\uparrow(\bullet) \in D \mapsto D_\perp$ and partial down injection $\downarrow(\bullet) \in D_\perp \not\mapsto D$), $D_1 \oplus D_2$ is the coalesced sum of domains D_1 and D_2 (with left and right injections $\bullet :: D_1 \in D_1 \mapsto D_1 \oplus D_2$ and $\bullet :: D_2 \in D_2 \mapsto D_1 \oplus D_2$), $\Omega \triangleq \uparrow(\omega) :: \mathbb{W}_\perp$ and $[D_1 \mapsto D_2]$ is the domain of continuous, $\perp$-strict, Ω-strict functions from D_1 into D_2. $\sqsubseteq$ is the computational ordering on $\mathbb{U}$ and $\sqcup$ is the least upper bound (lub) of increasing chains.

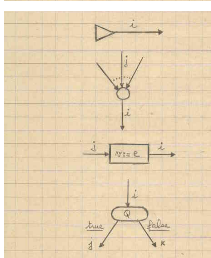
In the metalanguage for defining the denotational semantics below, $\Lambda x \bullet \ldots$ or $\Lambda x \in S \bullet \ldots$ is the lambda abstraction. $(\ldots \mathbin{?} \ldots \mid \ldots \mathbin{?} \ldots \mid \ldots)$ is the conditional expression.

POPL 97, Paris, France
© 1997 ACM 0-89791-853-3/96/01 ..$3.50

Probabilistic static analysis

-66-

4.2.5.2 _ Performance Analysis of Programs _



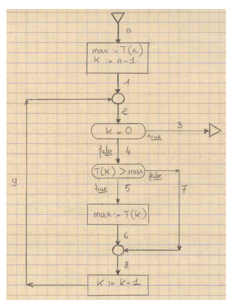
$C_i = 1$. (unique entry node).

$C_i = \sum_{j \in pred(i)} C_j$

$C_i = C_j$

$C_j = C_i * Prob(Q = \underline{true})$

$C_k = C_i * (1 - Prob(Q = \underline{true}))$



Applying KIRCHOFF laws, we get the system of equations:

$C_0 = 1$

$C_1 = C_0$

$C_2 = C_1 + C_9$

$C_3 = C_2 * p$ $\{p = Prob(k = 0) = 1/n\}$

$C_4 = C_2 * (1-p)$

$C_5 = C_4 * q$

$C_6 = C_5$ $\left\{ \begin{array}{l} q = Prob(T(k) > max) \\ \text{not simple, see KNUTH [1969, page 95]} \end{array} \right\}$

$C_7 = C_4 * (1-q)$

$C_8 = C_6 + C_7$

$C_9 = C_8$

iteration	C_2
0	1
1	$1 + (1-p)$
2	$1 + (1-p) + (1-p)^2$
...	...
n	$1 + (1-p) + (1-p)^2 + \ldots + (1-p)^n$
...	...

thus the limit of the sequence leads for C_2 to an infinite series, which limit is $1/p$:

$$\frac{1}{p} = \frac{1}{1-(1-p)} = 1 + (1-p) + \cdots + (1-p)^n + \cdots$$

Probabilistic static analysis

- ESOP 2012:

Probabilistic Abstract Interpretation

Patrick Cousot and Michael Monerau

Courant Institute, NYU and École Normale Supérieure, France

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

− 1 − ABSTRACT −

Abstract interpretation of programs is shown to be a suitable means to staticly 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

− 1 − ABSTRACT −

Abstract interpretation of programs is shown to be a suitable means to staticly 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@dmi.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^{∞}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^{∞}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[a]

[a]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.

Hierarchy of semantics

- Information and computation 2009:

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.

Parallelism

SEMANTIC ANALYSIS OF COMMUNICATING SEQUENTIAL PROCESSES

(Shortened Version)

Patrick Cousot* and Radhia Cousot**

1. INTRODUCTION

We present *semantic analysis techniques for concurrent programs* which are designed as networks of nondeterministic sequential processes, communicating with each other explicitly, by the sole means of synchronous, unbuffered message passing. The techniques are introduced using a version of Hoare[78]'s programming language CSP (*Communicating Sequential Processes*).

One goal is to propose an *invariance proof method* to be used in the development and verification of correct programs. The method is suitable to *partial correctness, absence of deadlock* and *non-termination proofs*. The design of this proof method is formalized so as to prepare the way to possible alternatives.

A complementary goal is to propose an *automatic technique for gathering information about CSP programs* that can be useful to both optimizing compilers and program partial verification systems.

2. SYNTAX AND OPERATIONAL SEMANTICS

2.1 Syntax

The set sCSP of syntactically valid programs is informally defined so as to capture the essential features of CSP.

- Programs Pr : [P(1) ‖ P(2) ‖ ... ‖ P(π)] where π≥2
  (A program consists of a single parallel command specifying concurrent execution of its constituent disjoint processes).
- Processes P(i), i∈[1,π] : Pℓ(i) :: D(i);λ(i,1):S(i)(1);...;λ(i,σ(i)):S(i)(σ(i)) where σ(i)≥1
  (Each process P(i) has a unique name Pℓ(i) and consists of a sequence of simple commands prefixed with declarations D(i) of local variables).
- Process labels Pℓ(i), i∈[1,π].
- Declarations D(i), i∈[1,π] : x(i)(1):t(i)(1);...;x(i)(δ(i)):t(i)(δ(i)) where δ(i)≥1.
- Variables x(i)(j), i∈[1,π], j∈[1,δ(i)].
- Types t(i)(j), i∈[1,π], j∈[1,δ(i)].
- Program locations λ(i,j), i∈[1,π], j∈[1,σ(i)].
  (Each command has been labeled to ease future references).
- Simple commands S(i)(j), i∈[1,π], j∈[1,σ(i)] :
  . Null commands S(i)(j), i∈[1,π], j∈N(i) : skip
  . Assignment commands S(i)(j), i∈[1,π], j∈A(i) : x(i)(α(i,j)):=e(i,j)(x(i)) where α(i,j)∈[1,δ(i)]

* Université de Metz, Faculté des Sciences, Ile du Saulcy, 57000 Metz, France.
** CRIN Nancy - Laboratoire Associé au CNRS n°262.
This work was supported by INRIA (SESORI-78208) and by CNRS (ATP Intelligence Artif.).

cited by 55

INVARIANCE PROOF METHODS 243

CHAPTER 12

Invariance Proof Methods And Analysis Techniques For Parallel Programs

Patrick Cousot
Université de Metz
Faculté des Sciences
Ile du Saulcy
57045 Metz cedex
France

Radhia Cousot
Centre de Recherche en Informatique de Nancy
France

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 [1980b, 1979], and Cousot, P. [1981].

cited by 36

THÈSE

présentée à l'

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir le grade de
DOCTEUR D'ÉTAT ÈS SCIENCES MATHÉMATIQUES

par

Radhia COUSOT

FONDEMENTS DES MÉTHODES
DE PREUVE D'INVARIANCE ET DE FATALITÉ
DE PROGRAMMES PARALLÈLES

Thèse soutenue le 15 novembre 1985 devant le jury :

Président	:	C. Pair	Rapporteur
Éxaminateurs	:	J.P. Jouannaud	
		G. Roucairol	Rapporteur extérieur
		M. Sintzoff	Rapporteur extérieur
		J.P. Verjus	

cited by 21

Google scholar

4.3.1.7 Dérivation de conditions de vérificationn correctes

Etant donnés un principe d'induction correct et sémantiquement complet

$[\exists I \in As.\ Cv(I)]$

et une correspondance $\gamma \in (\tilde{As} \to As)$ entre $\tilde{As}$ et As, toute preuve

$[\exists \tilde{I} \in \tilde{As}.\ \tilde{Cv}(\tilde{I})]$

telle que

$\tilde{Cv} \Rightarrow Cv \circ \gamma$

est correcte. On peut donc choisir $\tilde{Cv}[\![Pr, P]\!]$ comme étant $Cv[\![S[\![Pr]\!], A[\![Pr]\!], \Sigma[\![Pr]\!], t[\![Pr]\!], \epsilon[\![Pr]\!], \psi[\![Pr, P]\!]]\!] \circ \gamma[\![Pr, P]\!]$. Par diverses manipulations algébriques on cherchera à exprimer cette condition sous forme d'une conjonction de conditions plus simples correspondant chacune à une commande élémentaire du programme Pr. Des simplifications sont possibles puisque c'est une implication et non pas une égalité qui est réquise. La méthode de preuve obtenue de cette manière est correcte par construction. Pour que le résultat soit valable non pas pour un programme Pr particulier mais pour le langage Pr considéré il faut procéder par induction sur la syntaxe du langage.

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:

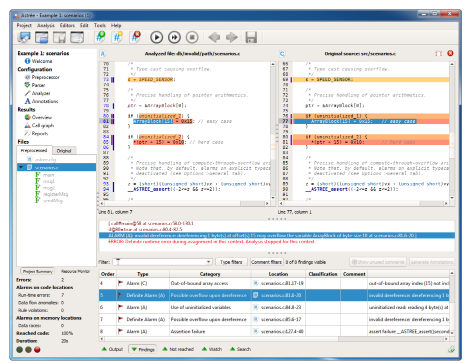
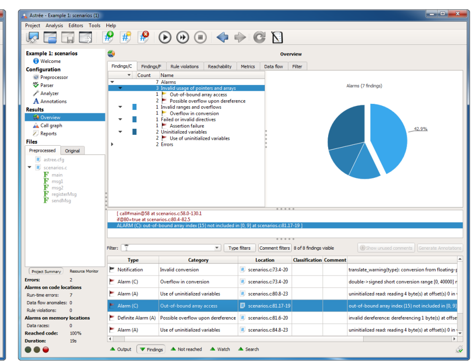
  The interval analysis was implemented in the AdaWorld compiler for IBM PC 80286 by J.D. Ichbiah and his Alsys SA corporation team in 1980–87.

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 $\rightarrow$ we cannot sell as much memory as we did before;
  - Many bugs are found at compile time $\rightarrow$ we make less money with our debugging services.

AbsInt Angewandte Informatik GmbH

- Astrée sold by AbsInt:

Abstract interpretation based static analyzers

- Ait www.absint.com/ait/, StackAnalyzer www.absint.com/stackanalyzer from AbSint
- Polyspace static analysis www.mathworks.com/products/polyspace.html
- Julia (Java) www.juliasoft.com
- Ikos, NASA ti.arc.nasa.gov/opensource/ikos/
- Clousot for code contract, Microsoft, github.com/Microsoft/CodeContracts
- Infer (Facebook) http://fbinfer.com
- Zoncolan (Facebook)
- Google
- ...

Abstract interpretation: Prospective

The future is hard to predict

- From my thesis in 1978:

computer, economical and biological systems

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.

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 catastrophy, 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

Dreams (40 years)

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

Dreams (40 years)

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