Abstract Interpretation: Theory and Practice

Patrick COUSOT

École Normale Supérieure
45 rue d'Ulm, 75230 Paris cedex 05, France
`Patrick.Cousot@ens.fr`
`www.di.ens.fr/~cousot`

ETAPS 2002 — CC 2002 — SPIN 2002
Grenoble, France April 6–14, 2002

Abstract

Our objective in this talk is to give an intuitive account of abstract interpretation theory and to present and discuss its main applications.

Abstract interpretation theory formalizes the conservative approximation of the semantics of hardware or software computer systems. The *semantics* provides a formal model describing all possible behaviors of a computer system in interaction with any possible environment. By *approximation* we mean the observation of the semantics at some level of abstraction, ignoring irrelevant details. *Conservative* means that the approximation can never lead to an erroneous conclusion.

Abstract interpretation theory provides *thinking tools* since the idea of abstraction by conservative approximation is central to reasoning (in particular on computer systems) and *mechanical tools* since the idea of an effectively computable approximation leads to a systematic and constructive formal design methodology of automatic semantics-based program manipulation algorithms and tools.

We will present various applications of abstract interpretation theory to the design of hierarchies of semantics, program transformations, typing, model-checking and in more details static program analysis. Abstract interpretation provides a general theory behind all program analyzers, which only differ in their choice of considered programming languages, program properties and their abstractions. Finally, we will discuss the various possible designs of program analyzers, from general-purpose to application-specific ones.

Content

1. Motivations 4
2. Informal introduction to abstract interpretation 8
3. Elements of abstract interpretation 16
4. A potpourri of applications of abstract interpretation 24
   (a) Syntax 24
   (b) Semantics 28
   (c) Typing 40
   (d) Model Checking 56
   (e) Program Transformations 68
   (f) Static Program Analysis 72
5. On the design of static program analyzers 88
6. Conclusion 92

Motivations

This work was supported in part by the RTD project IST-1999-20527 Daedalus of the european IST FP5 programme.

Abstract Interpretation

- **Thinking tool**: the idea of abstraction is central to reasoning (in particular on computer systems);
- A framework for designing **mechanical tools**: the idea of effective approximation leads to automatic semantics-based program manipulation tools.

Reasonings about computer systems and their verification should ideally rely on a few principles rather than on a myriad of techniques and (semi-)algorithms.

Coping With Undecidability When Computing on the Program Semantics

- Ask the programmer to help (e.g. proof assistants);
- Consider decidable questions only or semi-algorithms (e.g. model-checking/model-debugging);
- Consider effective approximations to handle practical complexity limitations;

The above approaches can all be formalized within the abstract interpretation framework.

The Theory of Abstract Interpretation

- **Abstract interpretation**[1] is a theory of conservative approximation of the semantics/models of computer systems.

  Approximation: observation of the behavior of a computer system at some level of abstraction, ignoring irrelevant details;

  Conservative: the approximation cannot lead to any erroneous conclusion.

Informal Introduction to Abstract Interpretation

[1] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État ès sciences mathématiques. Grenoble, 21 Mar. 1978.

1 – Abstract Domains

- Program concrete properties are specified by the semantics of programming languages;
- Program abstract properties are elements of abstract domains (posets/lattices/...);
- Program property abstraction is performed by (effective) conservative approximation of concrete properties;
- The abstract properties (hence abstract semantics) are sound but may be incomplete with respect to the concrete properties (semantics);

2 – Correspondence between Concrete and Abstract Properties

- If any concrete property has a best approximation, approximation is formalized by Galois connections (or equivalently closure operators, Moore families, etc. [2]);
- Otherwise, weaker abstraction/ concretization correspondences are available [3];

[2] P. Cousot & R. Cousot. *Systematic design of program analysis frameworks.* ACM POPL'79, pp. 269–282, 1979.
[3] P. Cousot & R. Cousot. *Abstract interpretation frameworks.* JLC 2(4):511–547, 1992.

3 – Semantics Abstraction

- Program concrete semantics and specifications are defined by syntactic induction and composition of fixpoints (or using equivalent presentations [4]);
- The property abstraction is extended compositionally to all constructions of the concrete/abstract semantics, including fixpoints;
- This leads to a constructive design of the abstract semantics by approximation of the concrete semantics [5];

4 — Effective Analysis/Checking/ Verification Algorithms

- Computable abstract semantics lead to effective program analysis/checking/verification algorithms;
- Furthermore fixpoints can be approximated iteratively by convergence acceleration through widening/narrowing that is non-standard induction [6].

[4] P. Cousot & R. Cousot. *Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game theoretic form.* CAV '95, LNCS 939, pp. 293–308, 1995.
[5] P. Cousot & R. Cousot. *Inductive definitions, semantics and abstract interpretation.* POPL, 83–94, 1992.
[6] P. Cousot & R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* ACM POPL, pp. 238–252, 1977.

Elements of Abstract Interpretation

- P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État ès sciences mathématiques. Grenoble, 21 Mar. 1978.

Galois Connections [7]

$$\langle P, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$$

$\overset{\text{def}}{=}$

- $\langle P, \leq\rangle$ is a poset
- $\langle Q, \sqsubseteq\rangle$ is a poset
- $\forall x \in P : \forall y \in Q : \alpha(x) \sqsubseteq y \iff x \leq \gamma(y)$

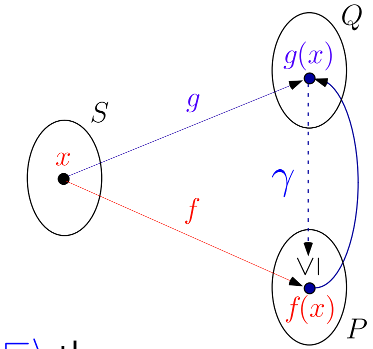
[7] The original Galois correspondence is semi-dual ($\sqsupseteq$ instead of $\sqsubseteq$).

Composing Galois Connections

- If $\langle P, \leq\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$ and $\langle Q, \sqsubseteq\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle R, \preceq\rangle$ then

$$\langle P, \leq\rangle \underset{\alpha_2\circ\alpha_1}{\overset{\gamma_1\circ\gamma_2}{\leftrightarrows}} \langle R, \preceq\rangle$$ [8]

Function Abstraction (1)

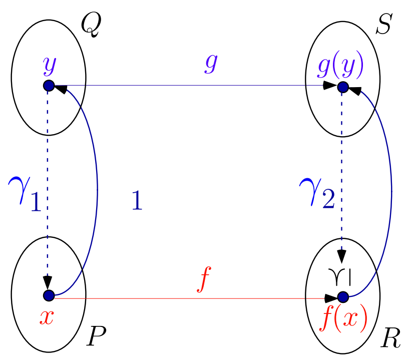


- If $\langle P, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$ then

$$\langle S \mapsto P, \dot{\leq}\rangle \underset{\boldsymbol{\lambda} f \cdot \boldsymbol{\lambda} x \cdot \alpha(f(x))}{\overset{\boldsymbol{\lambda} g \cdot \boldsymbol{\lambda} x \cdot \gamma(g(x))}{\leftrightarrows}} \langle S \mapsto Q, \dot{\sqsubseteq}\rangle$$

[8] This would not be true with the original definition of Galois correspondences.

Function Abstraction (2)



- If $\langle P, \leq\rangle \xleftrightarrow[\alpha_1]{\gamma_1} \langle Q, \subseteq\rangle$ and $\langle R, \preceq\rangle \xleftrightarrow[\alpha_2]{\gamma_2} \langle S, \sqsubseteq\rangle$ then

$$\langle P \xrightarrow{m} R, \dot{\subseteq}\rangle \xleftrightarrow[\boldsymbol{\lambda} f \bullet \alpha_2 \circ f \circ \gamma_1]{\boldsymbol{\lambda} g \bullet \gamma_2 \circ g \circ \alpha_1} \langle Q \xrightarrow{m} S, \dot{\sqsubseteq}\rangle$$

Fixpoint Approximation

Let $F \in L \xrightarrow{m} L$ and $\bar{F} \in \bar{L} \xrightarrow{m} \bar{L}$ be respective monotone maps on the cpos $\langle L, \bot, \sqsubseteq\rangle$ and $\langle \bar{L}, \bar{\bot}, \bar{\sqsubseteq}\rangle$ and $\langle L, \sqsubseteq\rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{L}, \bar{\sqsubseteq}\rangle$ such that $\alpha \circ F \circ \gamma \,\dot{\bar{\sqsubseteq}}\, \bar{F}$. Then [9]:

- $\forall \delta \in \mathbb{O}$: $\alpha(F^{\delta}) \mathrel{\bar{\sqsubseteq}} \bar{F}^{\delta}$ (iterates from the infimum);
- The iteration order of $\bar{F}$ is $\leq$ to that of F;
- $\alpha(\operatorname{lfp}^{\sqsubseteq} F) \mathrel{\bar{\sqsubseteq}} \operatorname{lfp}^{\bar{\sqsubseteq}} \bar{F}$;

Soundness: $\operatorname{lfp}^{\bar{\sqsubseteq}} \bar{F} \mathrel{\bar{\sqsubseteq}} \bar{P} \Rightarrow \operatorname{lfp}^{\sqsubseteq} F \sqsubseteq \gamma(\bar{P})$.

[9] P. Cousot & R. Cousot. *Systematic design of program analysis frameworks.* ACM POPL'79, pp. 269–282, 1979. Numerous variants!

Fixpoint Abstraction

Moreover, the *commutation condition* $\bar{F} \circ \alpha = \alpha \circ F$ implies [10]:

- $\bar{F} = \alpha \circ F \circ \gamma$, and
- $\alpha(\operatorname{lfp}^{\sqsubseteq} F) = \operatorname{lfp}^{\bar{\sqsubseteq}} \bar{F}$;

Completeness: $\operatorname{lfp}^{\sqsubseteq} F \sqsubseteq \gamma(\bar{P}) \Rightarrow \operatorname{lfp}^{\bar{\sqsubseteq}} \bar{F} \mathrel{\bar{\sqsubseteq}} \bar{P}$.

Systematic Design of an Abstract Semantics

By structural induction on the language syntax, for each language construct:

- Define the concrete semantics $\operatorname{lfp}^{\sqsubseteq} F$;
- Choose the abstraction $\alpha = \kappa(\alpha_1, \ldots, \alpha_n)$ and check $\langle L, \sqsubseteq\rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{L}, \bar{\sqsubseteq}\rangle$;
- Calculate $\bar{F} \overset{\text{def}}{=} \alpha \circ F \circ \gamma$ and check that $\bar{F} \circ \alpha = \alpha \circ F$;
- It follows, by construction, that $\alpha(\operatorname{lfp}^{\sqsubseteq} F) = \operatorname{lfp}^{\bar{\sqsubseteq}} \bar{F}$.

(and similarly in case of approximation).

[10] P. Cousot & R. Cousot. *Systematic design of program analysis frameworks.* ACM POPL'79, pp. 269–282, 1979. Numerous variants!

Abstract Domains

An abstraction α is a specification of an abstract domain, including:

- the representation of the abstract properties;
- the approximation ordering lattice structure ($\leq$, 0, 1, $\vee$, $\wedge$, ...);
- the computational ordering cpo structure ($\sqsubseteq$, $\bot$, $\sqcup$, ...);
- the abstract operators, e.g. *non-relational abstract multiplication*:
  - $P \otimes Q \stackrel{\text{def}}{=} \alpha(\{x \times y \mid x \in \gamma(P) \wedge y \in \gamma(Q)\})$ *postcondition*
  - $\otimes^{-1}(R) \stackrel{\text{def}}{=} \alpha(\{\langle x, y\rangle \mid x \times y \in \gamma(R)\})$ *precondition*

A Potpourri of Applications of Abstract Interpretation

Combinations of Abstract Domains [11]

Operation	$\kappa(\alpha_1, \ldots, \alpha_n)$	*Intuition*
Composition	$\alpha_n \circ \ldots \circ \alpha_1$	*Successive abstractions*
Duality	$\neg\kappa(\neg\alpha_1, \ldots, \neg\alpha_n)$	*Contraposition* [12]
Reduced product	$\alpha_1 \sqcap \ldots \sqcap \alpha_n$	*Conjunction*
Reduced power	$\alpha_1 \mapsto \ldots \mapsto \alpha_n$	*Case analysis*

[11] P. Cousot & R. Cousot. *Systematic design of program analysis frameworks.* ACM POPL'79, pp. 269–282, 1979.

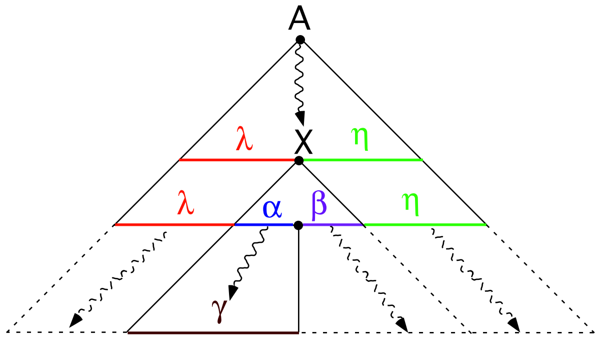
[12] P. Cousot. *Semantic Foundations of Program Analysis.* In *Program Flow Analysis: Theory and Applications*, Prentice-Hall, pp. 303–342, 1981.

Application to Syntax

- P. Cousot & R. Cousot. *Parsing as Abstract Interpretation of Grammar Semantics*, TCS, 2002, to appear.

The Semantics of Syntax

- The semantics of a grammar $G = \langle N, T, P, A\rangle$ is the set of items $[\lambda, X := \alpha/\gamma \bullet \beta]$ such that $\exists \eta : \exists X := \alpha\beta \in P$:



The Fixpoint Semantics of Syntax

$$S = \operatorname{lfp}^{\subseteq} F$$

$$\begin{aligned} F(I) \stackrel{\text{def}}{=} & \{[\epsilon, A := \epsilon/\epsilon \bullet \beta] \mid A := \beta \in P\} \\ & \cup \{[\lambda, X := \alpha Y/\gamma\delta \bullet \beta] \mid [\lambda, X := \alpha/\gamma \bullet Y\beta] \in I \wedge \\ & \qquad Y := \delta \in P\} \\ & \cup \{[\lambda, X := \alpha Y/\gamma\xi \bullet \beta] \mid [\lambda, X := \alpha/\gamma \bullet Y\beta] \in I \wedge \\ & \qquad [\lambda\gamma, Y := \delta/\xi \bullet \epsilon] \in I\} \\ & \cup \{[\lambda, X := \alpha a/\gamma a \bullet \beta] \mid [\lambda, X := \alpha/\gamma \bullet a\beta] \in I\} . \end{aligned}$$

Syntactic Abstractions

- $\alpha_\ell(I) \stackrel{\text{def}}{=} \{\gamma \in T^\star \mid [\epsilon, A := \alpha/\gamma \bullet \epsilon] \in I\}$

  Language of the grammar $G = \langle N, T, P, A\rangle$

- $\omega = \omega_1 \ldots \omega_i\omega_{i+1} \ldots \omega_j \ldots \omega_n$ input string

  $\alpha_\omega(I) \stackrel{\text{def}}{=} \{\langle X := \alpha \bullet \beta, i, j\rangle \mid 0 \leq i \leq j \leq n \wedge$
  $[\omega_1 \ldots \omega_i, X := \alpha/\omega_{i+1} \ldots \omega_j \bullet \beta] \in I\}$

  Earley's algorithm

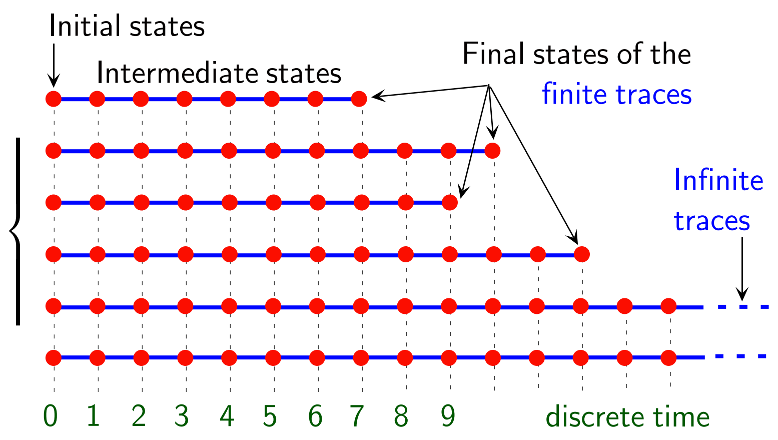
- $\alpha_f(I) \stackrel{\text{def}}{=} \{a \in T \mid [\lambda, X := \alpha/a\gamma \bullet \beta] \in I\}$
  $\cup \{\epsilon \mid [\lambda, X := \alpha\beta/\epsilon \bullet \epsilon] \in I\}$

  FIRST algorithm

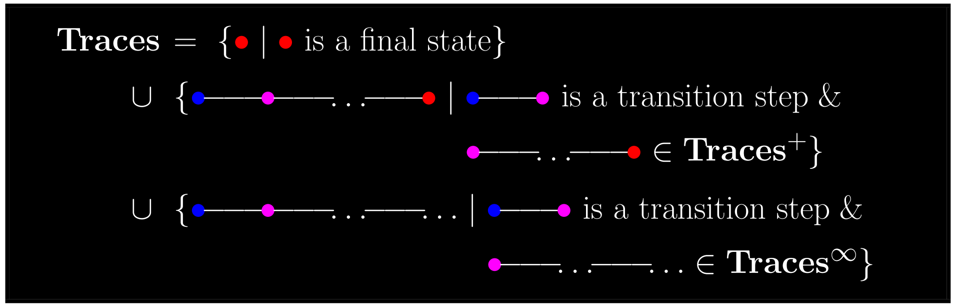
Application to Semantics

- P. Cousot, *Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation*. MFPS XIII, ENTCS 6, 1997. http://www.elsevier.nl/locate/entcs/volume6.html, 25 p.
- P. Cousot, *Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation*, TCS, 2002, to appear.

Trace Semantics, intuition



Least Fixpoint Trace Semantics


Traces = {● | ● is a final state}
∪ {●——●——. . .——● | ●——● is a transition step &
●——. . .——● ∈ Traces⁺}
∪ {●——●——. . .——. . . | ●——● is a transition step &
●——. . .——. . . ∈ Traces^∞}

- In general, the equation has multiple solutions;
- Choose the least one for the computational ordering:

"more finite traces & less infinite traces".

Trace Semantics, Formally

Trace semantics of a transition system $\langle\Sigma, \tau\rangle$:

- $\Sigma^+ \overset{\text{def}}{=} \bigcup_{n>0} [0, n[\longmapsto \Sigma$ finite traces
- $\Sigma^w \overset{\text{def}}{=} [0, \omega[\longmapsto \Sigma$ infinite traces
- $S = \text{lfp}^{\sqsubseteq}\ F \in \Sigma^+ \cup \Sigma^\omega$ trace semantics
- $F(X) = \{s \in \Sigma^+ \mid s \in \Sigma \wedge \forall s' \in \Sigma : \langle s, s'\rangle \notin \tau\}$
  $\cup \{ss'\sigma \mid \langle s, s'\rangle \in \tau \wedge s'\sigma \in X\}$ trace transformer
- $X \sqsubseteq Y \overset{\text{def}}{=} (X \cap \Sigma^+) \subseteq (Y \cap \Sigma^+) \wedge (X \cap \Sigma^\omega) \supseteq (Y \cap \Sigma^\omega)$ computational ordering

Semantics Abstractions

1 — Relational Semantics Abstractions

$$\langle\wp(\Sigma^+ \cup \Sigma^\omega), \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle\wp(\Sigma \times (\Sigma \cup \{\bot\})), \subseteq\rangle$$

1 — Relational Semantics Abstractions (Cont'd)

- $\alpha^{\natural}(X) = \{\langle s, s'\rangle \mid s\sigma s' \in X \cap \Sigma^+\}$
  $\qquad \cup \{\langle s, \bot\rangle \mid s\sigma \in X \cap \Sigma^\omega\}$
  trace to natural relational semantics
- $\alpha^{\flat}(X) = \{\langle s, s'\rangle \mid s\sigma s' \in X \cap \Sigma^+\}$
  trace to angelic relational semantics
- $\alpha^{\sharp}(X) = \{\langle s, s'\rangle \mid s\sigma s' \in X \cap \Sigma^+\}$
  $\qquad \cup \{\langle s, s'\rangle \mid s\sigma \in X \cap \Sigma^\omega \wedge s' \in \Sigma \cup \{\bot\}\}$
  trace to demoniac relational semantics

2 — Functional/Denotational Semantics Abstractions

$$\langle \wp(\Sigma \times (\Sigma \cup \{\bot\})), \subseteq\rangle \underset{\alpha^{\varphi}}{\overset{\gamma^{\varphi}}{\leftrightarrows}} \langle \Sigma \longmapsto \wp(\Sigma \cup \{\bot\}), \dot{\subseteq}\rangle$$

- $\alpha^{\varphi}(X) = \lambda s.\{s' \in \Sigma \cup \{\bot\} \mid \langle s, s'\rangle \in X\}$
  relational to denotational semantics

3 — Predicate Transformer Semantics Abstractions

$$\langle \Sigma \longmapsto \wp(\Sigma \cup \{\bot\}), \dot{\subseteq}\rangle \underset{\alpha^{\pi}}{\overset{\gamma^{\pi}}{\leftrightarrows}} \langle \wp(\Sigma) \overset{\cup}{\longmapsto} \wp(\Sigma \cup \{\bot\}), \dot{\subseteq}\rangle$$

- $\alpha^{\pi}(\phi) = \lambda P.\{s' \in \Sigma \cup \{\bot\} \mid \exists s \in P : s' \in \phi(s)\}$
  denotational to predicate transformer semantics

4 — Predicate Transformer Semantics Abstractions (Cont'd)

$$\begin{tikzcd}
\langle \wp(\Sigma) \overset{\cup}{\mapsto} \wp(\Sigma \cup \{\bot\}), \dot{\subseteq}\rangle \arrow[r, shift right, "\alpha^{\sim}"'] \arrow[d, shift right, "\alpha^{\cup}"'] & \langle \wp(\Sigma) \overset{\cap}{\mapsto} \wp(\Sigma \cup \{\bot\}), \dot{\supseteq}\rangle \arrow[l, shift right, "\gamma^{\sim}"'] \arrow[d, shift right, "\alpha^{\cap}"'] \\
\langle \wp(\Sigma \cup \{\bot\}) \overset{\cup}{\mapsto} \wp(\Sigma), \dot{\subseteq}\rangle \arrow[r, shift right, "\alpha^{\sim}"'] \arrow[u, shift right, "\gamma^{\cup}"'] & \langle \wp(\Sigma \cup \{\bot\}) \overset{\cap}{\mapsto} \wp(\Sigma), \dot{\supseteq}\rangle \arrow[l, shift right, "\gamma^{\sim}"'] \arrow[u, shift right, "\gamma^{\cap}"']
\end{tikzcd}$$

- $\alpha^{\sim}(\Phi) = \lambda P.\neg(\Phi(\neg P))$ dual
- $\alpha^{\cup}(\Phi) = \lambda Q.\{s \in \Sigma \mid \Phi(\{s\}) \cap Q \neq \emptyset\}$ $\cup$-inversion
- $\alpha^{\cap}(\Phi) = \lambda Q.\{s \in \Sigma \mid \Phi(\neg\{s\}) \cup Q = \Sigma \cup \{\bot\}\}$ $\cap$-inversion

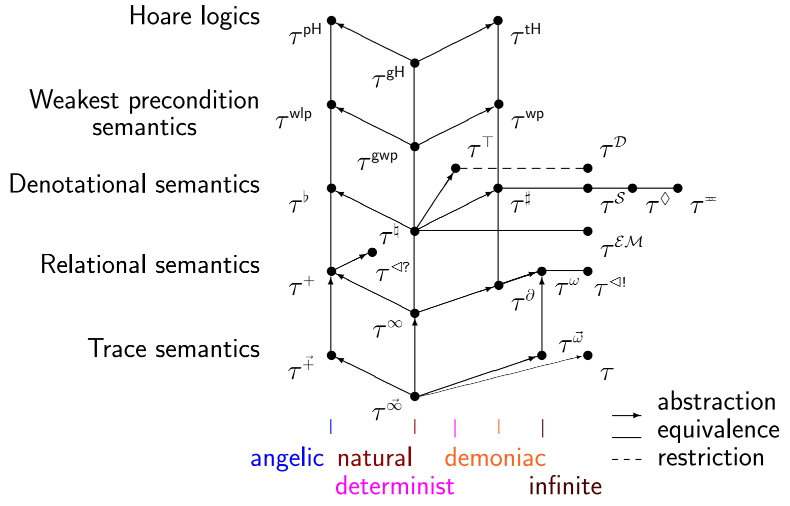
5 — Hoare Logic Semantics Abstractions

$$\langle \wp(\Sigma) \overset{\cap}{\longmapsto} \wp(\Sigma \cup \{\bot\}), \dot{\supseteq}\rangle \underset{\alpha^H}{\overset{\gamma^H}{\leftrightarrows}} \wp(\Sigma) \otimes \text{[13]}\ \wp(\Sigma \cup \{\bot\}), \dot{\supseteq}\rangle$$

- $\alpha^H(\Phi) = \{\langle P, Q\rangle \mid P \subseteq \Phi(Q)\}$

  predicate transformer to Hoare logic semantics

Lattice of Semantics



[13] Semi-dual Shmuely tensor product.

Application to Typing

- P. Cousot, *Types as Abstract Interpretations*, ACM 24th POPL, 1997, pp. 316-331.

Syntax of the Eager Lambda Calculus

$\mathtt{x}, \mathtt{f}, \ldots \in \mathbb{X}$	:	variables
$e \in \mathbb{E}$	:	expressions
$e ::=$	$\mathtt{x}$	variable
$\mid$	$\boldsymbol{\lambda}\mathtt{x} \cdot e$	abstraction
$\mid$	$e_1(e_2)$	application
$\mid$	$\boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e$	recursion
$\mid$	$\mathbf{1}$	one
$\mid$	$e_1 - e_2$	difference
$\mid$	$(e_1\ \mathbf{?}\ e_2 : e_3)$	conditional

Semantic Domains

Ω	wrong/runtime error value
$\bot$	non-termination
$\mathbb{W} \stackrel{\text{def}}{=} \{\Omega\}$	wrong
$z \in \mathbb{Z}$	integers
$u, f, \varphi \in \mathbb{U} \cong \mathbb{W}_\bot \oplus \mathbb{Z}_\bot \oplus [\mathbb{U} \mapsto \mathbb{U}]^{14}{}_\bot$	values
$R \in \mathbb{R} \stackrel{\text{def}}{=} \mathbb{X} \mapsto \mathbb{U}$	environments
$\phi \in \mathbb{S} \stackrel{\text{def}}{=} \mathbb{R} \mapsto \mathbb{U}$	semantic domain

Denotational Semantics with Run-Time Type Checking

$$\mathbf{S}[\![1]\!]R \stackrel{\text{def}}{=} 1$$

$$\begin{aligned}\mathbf{S}[\![e_1 - e_2]\!]R \stackrel{\text{def}}{=} (\ & \mathbf{S}[\![e_1]\!]R = \bot \vee \mathbf{S}[\![e_2]\!]R = \bot \ ?\ \bot \\ |\ & \mathbf{S}[\![e_1]\!]R = z_1 \wedge \mathbf{S}[\![e_2]\!]R = z_2 \ ?\ z_1 - z_2 \\ |\ & \Omega\ )\end{aligned}$$

$$\begin{aligned}\mathbf{S}[\![(e_1 \mathbin{\mathbf{?}} e_2 \mathbin{\mathbf{:}} e_3)]\!]R \stackrel{\text{def}}{=} (\ & \mathbf{S}[\![e_1]\!]R = \bot \ ?\ \bot \\ |\ & \mathbf{S}[\![e_1]\!]R = 0 \ ?\ \mathbf{S}[\![e_2]\!]R \\ |\ & \mathbf{S}[\![e_1]\!]R = z \neq 0 \ ?\ \mathbf{S}[\![e_3]\!]R \\ |\ & \Omega\ )\end{aligned}$$

[14] $[\mathbb{U} \mapsto \mathbb{U}]$: continuous, $\bot$-strict, Ω-strict functions from values $\mathbb{U}$ to values $\mathbb{U}$.

$$\mathbf{S}[\![x]\!]R \stackrel{\text{def}}{=} R(x)$$

$$\begin{aligned}\mathbf{S}[\![\boldsymbol{\lambda} x \cdot e]\!]R \stackrel{\text{def}}{=} \boldsymbol{\lambda} u \cdot (\ & u = \bot \ ?\ \bot \\ |\ & u = \Omega \ ?\ \Omega \\ |\ & \mathbf{S}[\![e]\!]R[x \leftarrow u]\ )\end{aligned}$$

$$\begin{aligned}\mathbf{S}[\![e_1(e_2)]\!]R \stackrel{\text{def}}{=} (\ & \mathbf{S}[\![e_1]\!]R = \bot \vee \mathbf{S}[\![e_2]\!]R = \bot \ ?\ \bot \\ |\ & \mathbf{S}[\![e_1]\!]R = f \in [\mathbb{U} \mapsto \mathbb{U}] \ ?\ f(\mathbf{S}[\![e_2]\!]R) \\ |\ & \Omega\ )\end{aligned}$$

$$\mathbf{S}[\![\boldsymbol{\mu} \mathtt{f} \cdot \boldsymbol{\lambda} x \cdot e]\!]R \stackrel{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq} \boldsymbol{\lambda} \varphi \cdot \mathbf{S}[\![\boldsymbol{\lambda} x \cdot e]\!]R[\mathtt{f} \leftarrow \varphi]$$

Standard Denotational & Collecting Semantics

- The denotational semantics is:
$$\mathbf{S}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{S}$$
- A concrete property P of a program is a set of possible program behaviors:
$$P \in \mathbb{P} \stackrel{\text{def}}{=} \wp(\mathbb{S})$$
- The standard collecting semantics is the strongest concrete property:
$$\mathbf{C}[\![\bullet]\!] \in \mathbb{E} \mapsto \mathbb{P} \qquad \mathbf{C}[\![e]\!] \stackrel{\text{def}}{=} \{\mathbf{S}[\![e]\!]\}$$

Church/Curry Monotypes

- Simple types are monomorphic:

$$m \in \mathbb{M}^{c}, \quad m ::= \texttt{int} \mid m_1 \rightarrow m_2 \qquad \text{monotype}$$

- A type environment associates a type to free program variables:

$$H \in \mathbb{H}^{c} \stackrel{\text{def}}{=} \mathbb{X} \mapsto \mathbb{M}^{c} \qquad \text{type environment}$$

Church/Curry Monotypes (continued)

- A typing $\langle H, m\rangle$ specifies a possible result type m in a given type environment H assigning types to free variables:

$$\theta \in \mathbb{I}^{c} \stackrel{\text{def}}{=} \mathbb{H}^{c} \times \mathbb{M}^{c} \qquad \text{typing}$$

- An abstract property or program type is a set of typings;

$$T \in \mathbb{T}^{c} \stackrel{\text{def}}{=} \wp(\mathbb{I}^{c}) \qquad \text{program type}$$

Concretization Function

The meaning of types is a program property, as defined by the concretization function γ^{c}:[15]

- Monotypes $\gamma_1^{c} \in \mathbb{M}^{c} \mapsto \wp(\mathbb{U})$:

$$\begin{aligned}
\gamma_1^{c}(\texttt{int}) &\stackrel{\text{def}}{=} \mathbb{Z} \cup \{\bot\}\\
\gamma_1^{c}(m_1 \rightarrow m_2) &\stackrel{\text{def}}{=} \{\varphi \in [\mathbb{U} \mapsto \mathbb{U}] \mid \\
&\qquad\qquad \forall \mathsf{u} \in \gamma_1^{c}(m_1) : \varphi(\mathsf{u}) \in \gamma_1^{c}(m_2)\}\\
&\qquad \cup \{\bot\}
\end{aligned}$$

- type environment $\gamma_2^{c} \in \mathbb{H}^{c} \mapsto \wp(\mathbb{R})$:

$$\gamma_2^{c}(H) \stackrel{\text{def}}{=} \{\mathsf{R} \in \mathbb{R} \mid \forall \mathtt{x} \in \mathbb{X} : \mathsf{R}(\mathtt{x}) \in \gamma_1^{c}(H(\mathtt{x}))\}$$

- typing $\gamma_3^{c} \in \mathbb{I}^{c} \mapsto \mathbb{P}$:

$$\gamma_3^{c}(\langle H, m\rangle) \stackrel{\text{def}}{=} \{\phi \in \mathbb{S} \mid \forall \mathsf{R} \in \gamma_2^{c}(H) : \phi(\mathsf{R}) \in \gamma_1^{c}(m)\}$$

- program type $\gamma^{c} \in \mathbb{T}^{c} \mapsto \mathbb{P}$:

$$\begin{aligned}
\gamma^{c}(T) &\stackrel{\text{def}}{=} \bigcap_{\theta \in T} \gamma_3^{c}(\theta)\\
\gamma^{c}(\emptyset) &\stackrel{\text{def}}{=} \mathbb{S}
\end{aligned}$$

[15] For short up/down lifting/injection are omitted.

Program Types

- Galois connection:
$$\langle \mathbb{P},\ \subseteq,\ \emptyset,\ \mathbb{S},\ \cup,\ \cap\rangle \underset{\alpha^{c}}{\overset{\gamma^{c}}{\leftrightarrows}} \langle \mathbb{T}^{c},\ \supseteq,\ \mathbb{I}^{c},\ \emptyset,\ \cap,\ \cup\rangle$$

- Types $\mathbf{T}[\![e]\!]$ of an expression e:
$$\mathbf{T}[\![e]\!] \subseteq \alpha^{c}(\mathbf{C}[\![e]\!]) = \alpha^{c}(\{\mathbf{S}[\![e]\!]\})$$

Typable Programs Cannot Go Wrong

$$\Omega \in \gamma^{c}(\mathbf{T}[\![e]\!]) \quad\Longleftrightarrow\quad \mathbf{T}[\![e]\!] = \emptyset$$

Church/Curry Monotype Abstract Semantics

$$\mathbf{T}[\![\mathtt{x}]\!] \stackrel{\text{def}}{=} \{\langle H, H(\mathtt{x})\rangle \mid H \in \mathbb{H}^{c}\} \qquad \text{(VAR)}$$

$$\mathbf{T}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \stackrel{\text{def}}{=} \{\langle H, m_1 \to m_2\rangle \mid \langle H[\mathtt{x}\leftarrow m_1], m_2\rangle \in \mathbf{T}[\![e]\!]\} \qquad \text{(ABS)}$$

$$\mathbf{T}[\![e_1(e_2)]\!] \stackrel{\text{def}}{=} \{\langle H, m_2\rangle \mid \langle H, m_1 \to m_2\rangle \in \mathbf{T}[\![e_1]\!] \wedge \langle H, m_1\rangle \in \mathbf{T}[\![e_2]\!]\} \qquad \text{(APP)}$$

$$\mathbf{T}[\![1]\!] \stackrel{\text{def}}{=} \{\langle H, \mathtt{int}\rangle \mid H \in \mathbb{H}^{c}\} \qquad \text{(CST)}$$

$$\mathbf{T}[\![e_1 - e_2]\!] \stackrel{\text{def}}{=} \{\langle H, \mathtt{int}\rangle \mid \langle H, \mathtt{int}\rangle \in \mathbf{T}[\![e_1]\!] \cap \mathbf{T}[\![e_2]\!]\} \qquad \text{(DIF)}$$

$$\mathbf{T}[\![(e_1\ \mathbf{?}\ e_2 : e_3)]\!] \stackrel{\text{def}}{=} \{\langle H, m\rangle \mid \langle H, \mathtt{int}\rangle \in \mathbf{T}[\![e_1]\!] \wedge \langle H, m\rangle \in \mathbf{T}[\![e_2]\!] \cap \mathbf{T}[\![e_3]\!]\} \qquad \text{(CND)}$$

$$\mathbf{T}[\![\boldsymbol{\mu}\mathtt{f}\cdot\boldsymbol{\lambda}\mathtt{x}\cdot e]\!] \stackrel{\text{def}}{=} \{\langle H, m\rangle \mid \langle H[\mathtt{f}\leftarrow m], m\rangle \in \mathbf{T}[\![\boldsymbol{\lambda}\mathtt{x}\cdot e]\!]\} \qquad \text{(REC)}[16]$$

The Herbrand Abstraction to Get Hindley's Unification-Based Type Inference Algorithm

$$\langle\wp(\mathbf{ground}(T)),\ \subseteq,\ \emptyset,\ \mathbf{ground}(T),\ \cup,\ \cap\rangle \underset{\mathbf{lcg}}{\overset{\mathbf{ground}}{\leftrightarrows}} \langle T^{\emptyset}/_{\equiv},\ \leq,\ \emptyset,\ [\text{'a}]_{\equiv},\ \mathbf{lcg},\ \mathbf{gci}\rangle$$

where:

- T: set of terms with variables 'a, ...,
- $\mathbf{lcg}$: least common generalization,
- $\mathbf{ground}$: set of ground instances,
- $\leq$: instance preordering,
- $\mathbf{gci}$: greatest common instance.

[16] The abstract fixpoint has been eliminated thanks to fixpoint induction: $\mathrm{lfp} F \sqsubseteq P \Leftrightarrow \exists I : F(I) \sqsubseteq I \wedge I \sqsubseteq P$.

Application to Model Checking

- P. Cousot & R. Cousot, *Temporal Abstract Interpretation*, ACM 27th POPL, 2000, pp. 12-25.

Objective of Model Checking

1) Built a model M of the computer system;
2) Check (i.e. prove enumeratively) or semi-check (with semi-algorithms) that the model satisfies a specification given (as a set of traces φ) by a (linear) temporal formula: $M \subseteq \varphi$ or $M \cap \varphi \neq \emptyset$.

- The model and specification should be proved to be correct abstractions of the computer system (often taken for granted, could be done by abstract interpretation);

Model-checking is an abstraction

- Universal abstraction:

$$\langle\wp(\Sigma^+ \cup \Sigma^\omega), \supseteq\rangle \underset{\alpha_M^\forall}{\overset{\gamma_M^\forall}{\longleftrightarrow}} \langle\wp(\Sigma), \supseteq\rangle$$

$$\alpha_M^\forall(\Phi) \stackrel{\text{def}}{=} \{s \mid \{\sigma \in M \mid \sigma_0 = s\} \subseteq \Phi\}$$

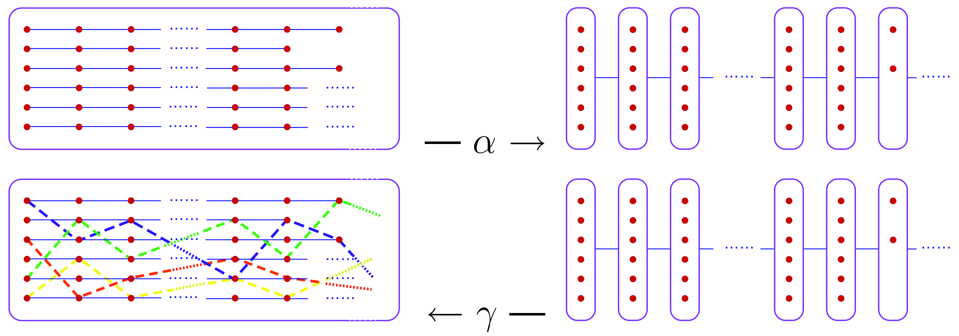
- Existential abstraction:

$$\langle\wp(\Sigma^+ \cup \Sigma^\omega), \subseteq\rangle \underset{\alpha_M^\exists}{\overset{\gamma_M^\exists}{\longleftrightarrow}} \langle\wp(\Sigma), \subseteq\rangle$$

$$\alpha_M^\exists(\Phi) \stackrel{\text{def}}{=} \{s \mid \{\sigma \in M \mid \sigma_0 = s\} \cap \Phi \neq \emptyset\}$$

These abstractions lead, by fixpoint approximation of the trace semantics, to the classical (finite-state or nonterminating) model-checking algorithms.

Implicit Abstraction in Model Checking



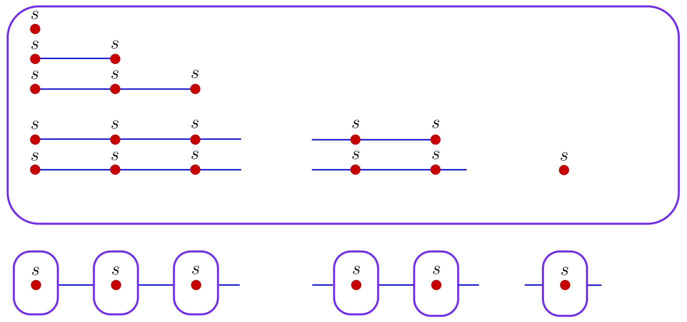
Spurious traces: - - - / - - - / - - - / - - - / ⋯ ;
The semantics of the μ-calculus is closed under this abstraction.

Soundness

For a *given class* of properties, soundness means that:

> Any property (in the *given class*) of the abstract world must hold in the concrete world;

Example for Unsoundness



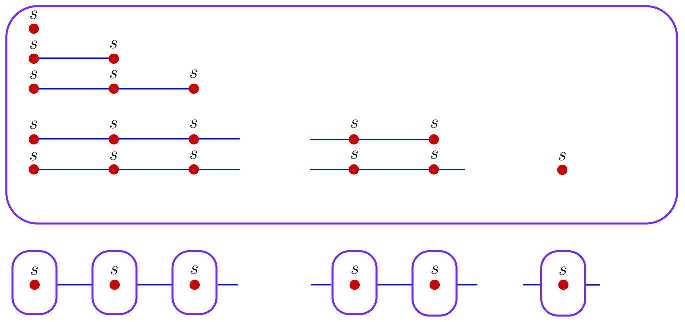
All abstract traces are infinite but not the concrete ones!

Completeness

For a *given class* of properties, completeness means that:

> Any property (in the *given class*) of the concrete world must hold in the abstract world;

Example for Incompleteness



All concrete traces are finite but not the abstract ones!

On the Completeness of Model-Checking

- Contrary to program analysis, model checking is complete;
- Completeness is relative to the model, not the program semantics;
- Completeness follows from restrictions on the models and specifications (e.g. closure under the implicit abstraction);
- There are models/specifications (such as the $\overset{\curvearrowleft}{\mu}$-calculus using bidirectional traces) for which:
  - The implicit abstraction is incomplete (POPL'00),
  - Any abstraction is incomplete (Ranzato, ESOP'01).

  in both cases, even for *finite* transition systems.

Bidirectional Traces

• $\langle i, \sigma \rangle$	bidirectional trace
$\sigma \in \mathbb{Z} \longmapsto \Sigma$	trace
$i \in \mathbb{Z}$	present time

The reversible $\overset{\curvearrowleft}{\mu}$-calculus

$\varphi ::= \boldsymbol{\sigma}_S$ [17]	$[\![\boldsymbol{\sigma}_S]\!]\rho \overset{\text{def}}{=} \{\langle i, \sigma \rangle \mid \sigma_i \in S\}$
$\mid \boldsymbol{\pi}_t$ [18]	$[\![\boldsymbol{\pi}_t]\!]\rho \overset{\text{def}}{=} \{\langle i, \sigma \rangle \mid \langle \sigma_i, \sigma_{i+1} \rangle \in t\}$
$\mid \oplus\varphi_1$ [19]	$[\![\oplus\varphi_1]\!]\rho \overset{\text{def}}{=} \{\langle i, \sigma \rangle \mid \langle i+1, \sigma \rangle \in [\![\varphi_1]\!]\rho$
$\mid \varphi_1^{\curvearrowleft}$	$[\![\varphi_1^{\curvearrowleft}]\!]\rho \overset{\text{def}}{=} \{\langle i, \sigma \rangle \mid \langle -i, \lambda j.\sigma_{-j} \rangle \in [\![\varphi_1]\!]\rho\}$
$\mid \varphi_1 \vee \varphi_2$	$[\![\varphi_1 \vee \varphi_2]\!]\rho \overset{\text{def}}{=} [\![\varphi_1]\!]\rho \cup [\![\varphi_2]\!]\rho$
$\mid \neg\, \varphi_1$	$[\![\neg\, \varphi_1]\!]\rho \overset{\text{def}}{=} \neg[\![\varphi_1]\!]\rho$

The reversible $\overset{\curvearrowleft}{\mu}$-calculus (cont'd)

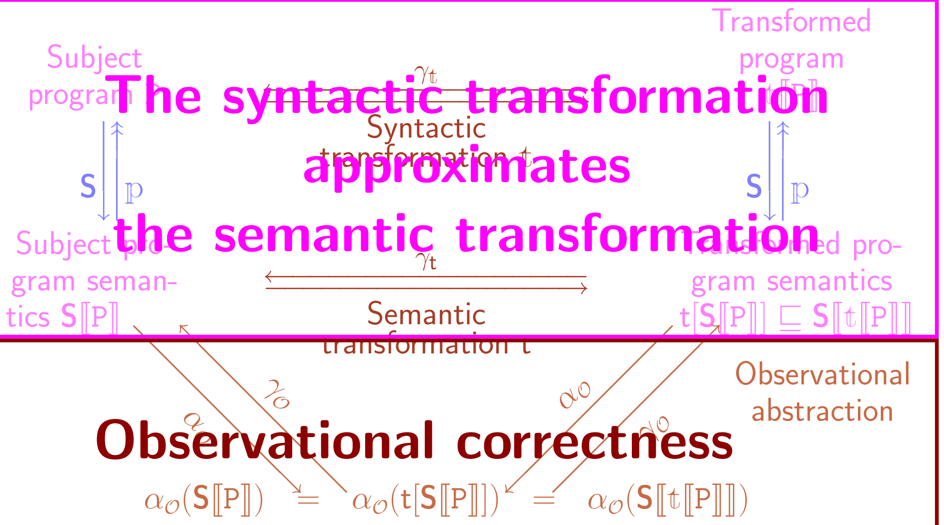
$\mid \ldots$	
$\mid X$ [20]	$[\![X]\!]\rho \overset{\text{def}}{=} \rho(X)$
$\mid \boldsymbol{\mu}\, X \cdot \varphi_1$	$[\![\boldsymbol{\mu}\, X \cdot \varphi_1]\!]\rho \overset{\text{def}}{=} \operatorname{lfp}^{\subseteq} \boldsymbol{\lambda}\, x \cdot [\![\varphi_1]\!]\rho X x$
$\mid \boldsymbol{\nu}\, X \cdot \varphi_1$	$[\![\boldsymbol{\nu}\, X \cdot \varphi_1]\!]\rho \overset{\text{def}}{=} \operatorname{gfp}^{\subseteq} \boldsymbol{\lambda}\, x \cdot [\![\varphi_1]\!]\rho X x$
$\mid \forall\, \varphi_1 : \varphi_2$ [21]	$[\![\forall\, \varphi_1 : \varphi_2]\!]\rho \overset{\text{def}}{=} \{\langle i, \sigma \rangle \in [\![\varphi_1]\!]\rho \mid \{\langle i, \sigma' \rangle \in [\![\varphi_1]\!]\rho \mid \sigma'_i = \sigma_i\} \subseteq [\![\varphi_2]\!]\rho\}$

17 $S \in \wp(\Sigma)$.
18 $t \in \wp(\Sigma \times \Sigma)$.
19 $\oplus$ is next time.
20 variable.
21 The traces of φ_1 such that all traces of φ_1 with same present state satisfy φ_2.

Application to Program Transformation

- P. Cousot & R. Cousot, *Systematic Design of Program Transformation Frameworks by Abstract Interpretation*, ACM 29th POPL, 2002, pp. 178—190.

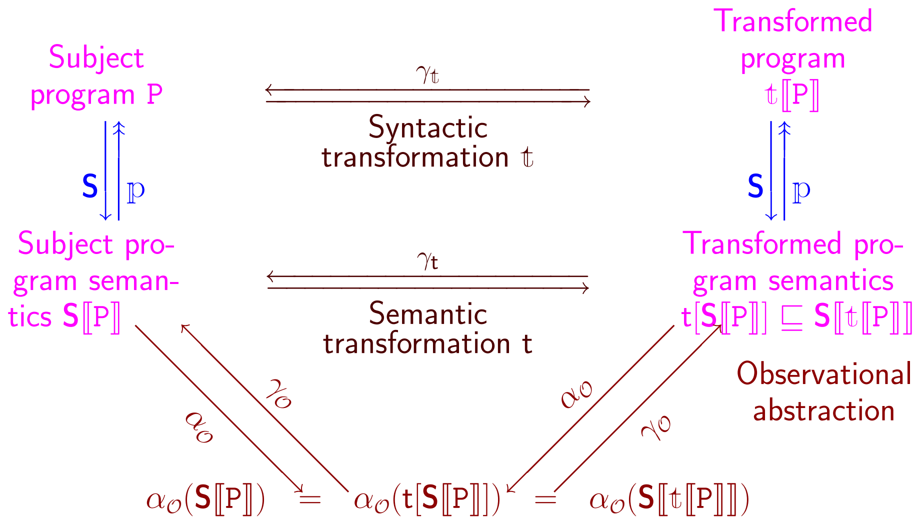
Principle of Online Program Transformation


Subject program P
Transformed program t[[P]]
γ_t
Syntactic transformation t
S
p
Subject program semantics S[[P]]
Transformed program semantics t[S[[P]]] ⊑ S[[t[[P]]]]
γ_t
Semantic transformation t
α_O
γ_O
Observational abstraction
$\alpha_O(S[[P]]) = \alpha_O(t[S[[P]]]) = \alpha_O(S[[t[[P]]]])$

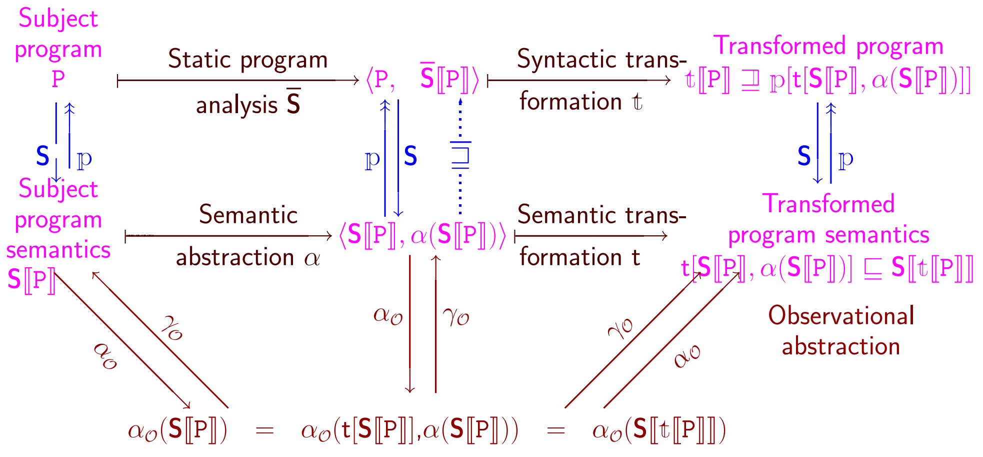
The syntactic transformation approximates the semantic transformation

Observational correctness

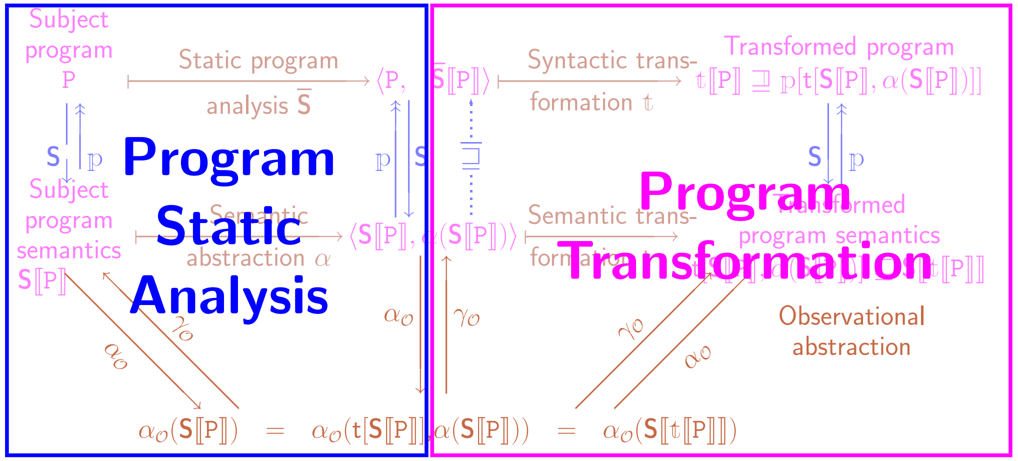
Principle of Online Program Transformation


Subject program P
Transformed program t[[P]]
γ_t
Syntactic transformation t
S
p
Subject program semantics S[[P]]
Transformed program semantics t[S[[P]]] ⊑ S[[t[[P]]]]
γ_t
Semantic transformation t
α_O
γ_O
Observational abstraction
$\alpha_O(S[[P]]) = \alpha_O(t[S[[P]]]) = \alpha_O(S[[t[[P]]]])$

Principle of Offline Program Transformation


Subject program P
Static program analysis $\overline{S}$
⟨P, $\overline{S}$[[P]]⟩
Syntactic transformation t
Transformed program t[[P]] ⊒ p[t[S[[P]], α(S[[P]])]]
S
p
⊒
Subject program semantics S[[P]]
Semantic abstraction α
⟨S[[P]], α(S[[P]])⟩
Semantic transformation t
Transformed program semantics t[S[[P]], α(S[[P]])] ⊑ S[[t[[P]]]]
α_O
γ_O
Observational abstraction
$\alpha_O(S[[P]]) = \alpha_O(t[S[[P]], \alpha(S[[P]])) = \alpha_O(S[[t[[P]]]])$

Principle of Offline Program Transformation


Subject program P
Static program analysis $\overline{S}$
$\langle P, \overline{S}[\![P]\!]\rangle$
Syntactic transformation t
Transformed program
$t[\![P]\!] \sqsupseteq p[t[S[\![P]\!], \alpha(S[\![P]\!])]]$
S
p
Program Static Analysis
Subject program semantics $S[\![P]\!]$
Semantic abstraction α
$\langle S[\![P]\!], \alpha(S[\![P]\!])\rangle$
Semantic transformation
Program Transformation
Transformed program semantics
Observational abstraction
$\alpha_{\mathcal{O}}$
$\gamma_{\mathcal{O}}$
$\alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(t[S[\![P]\!], \alpha(S[\![P]\!])]) = \alpha_{\mathcal{O}}(S[\![t[\![P]\!]]\!])$

Examples of Program Transformations

- Constant propagation;
- Online and offline partial evaluation;
- Slicing;
- Static program monitoring,

$$\alpha_{\mathcal{O}}(S[\![t[\![P, M]\!]]\!]) = \alpha_{\mathcal{O}}(S[\![P]\!]) \sqcap \alpha_{\mathcal{O}}(S[\![M]\!]):$$

  - run-time checks elimination,
  - security policy enforcement,
  - proof by transformation ($\alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(S[\![t[\![P, M]\!]]\!])$).
- Code and analysis translation.

Application to Static Program Analysis[22]

- P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État ès sciences mathématiques. Grenoble, 21 Mar. 1978.
- P. Cousot. *Semantic Foundations of Program Analysis.* Ch. 10 of *Program Flow Analysis: Theory and Applications*, S.S. Muchnick & N.D. Jones, pp. 303–342. Prentice-Hall, 1981.

What is static program analysis?

- Automatic static/compile time determination of dynamic/run-time properties of programs;
- **Basic idea:** use effective computable approximations of the program semantics;

  Advantage: fully automatic, no need for error-prone user designed model or costly user interaction;

  Drawback: can only handle properties captured by the approximation.

[22] Now called *software model checking*!

Collecting Semantics Abstractions

$$\langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \wp(\Sigma), \subseteq\rangle$$

Example 1: reachable states (forward analysis)

$$\alpha_I(X) \overset{\text{def}}{=} \{\sigma_i \mid \sigma \in X \wedge \sigma_0 \in I \wedge i \in \text{Dom}(\sigma)\}$$

Example 2: ancestor states (backward analysis)

$$\alpha_F(X) \overset{\text{def}}{=} \{\sigma_i \mid \sigma \in X \wedge \exists n \in \text{Dom}(\sigma) : 0 \leq i \leq n \wedge \sigma_n \in F\}$$

Partitioning

- If $\Sigma = C \times M$ (control and store state) and C is finite [23], we can partition:

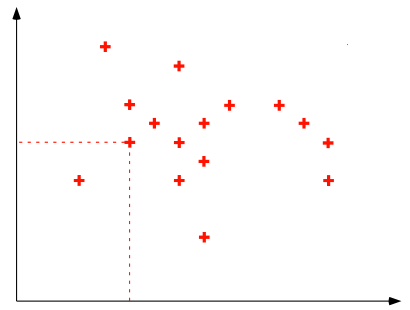
$$\langle \wp(C \times M), \subseteq\rangle \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} \langle C \mapsto \wp(M), \dot{\subseteq}\rangle$$

$$\alpha_c(S) \;=\; \boldsymbol{\lambda} c \in C \cdot \{m \mid \langle c, m\rangle \in S\}$$

- It remains to find abstractions of the store $M = V \mapsto D$ (variables to data) e.g. of [in]finite set of points of the euclidian space.

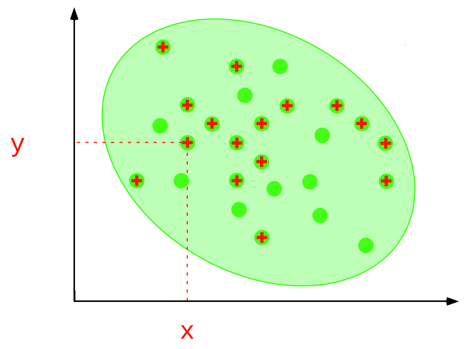
[23] use e.g. dynamic partitioning if C is infinite

Approximations of an [in]finite set of points;



$$\{\ldots, \langle 19,\ 77\rangle, \ldots,\ \langle 20,\ 02\rangle, \ldots\}$$

Approximations of an [in]finite set of points: From Above

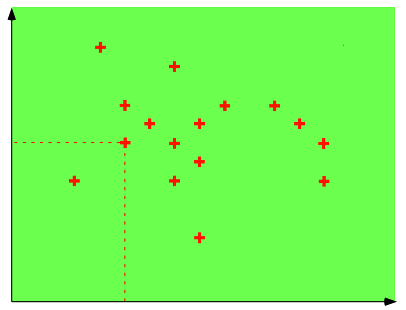


$$\{\ldots, \langle 19,\ 77\rangle, \ldots,\ \langle 20,\ 02\rangle, \langle ?,\ ?\rangle, \ldots\}$$

From Below: dual [24] + combinations.

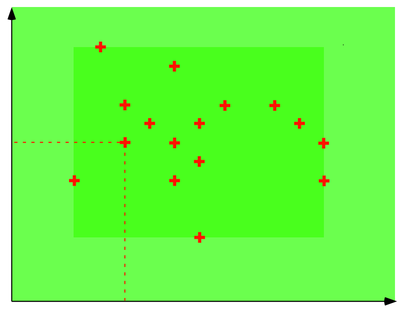
[24] Trivial for finite states (liveness model-checking), more difficult for infinite states (variant functions).

Effective computable approximations of an [in]finite set of points; Signs [25]



$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

Effective computable approximations of an [in]finite set of points; Intervals [26]

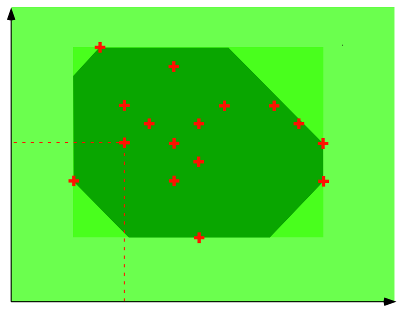


$$\begin{cases} x \in [19,\ 77] \\ y \in [20,\ 02] \end{cases}$$

[25] P. Cousot & R. Cousot. *Systematic design of program analysis frameworks.* ACM POPL'79, pp. 269–282, 1979.

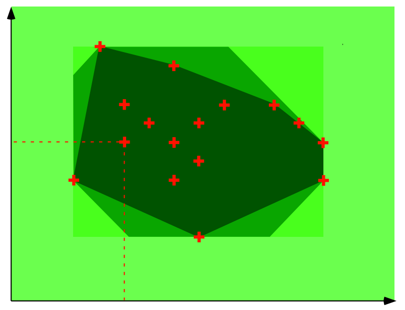
[26] P. Cousot & R. Cousot. *Static determination of dynamic properties of programs.* Proc. 2nd Int. Symp. on Programming, Dunod, 1976.

Effective computable approximations of an [in]finite set of points; Octagons [27]



$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 77 \\ 1 \leq y \leq 9 \\ x - y \leq 99 \end{cases}$$

Effective computable approximations of an [in]finite set of points; Polyhedra [28]

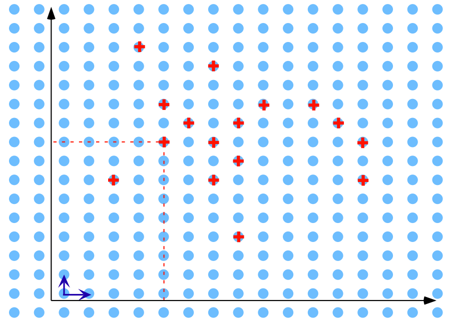


$$\begin{cases} 19x + 77y \leq 2002 \\ 20x + 02y \geq 0 \end{cases}$$

[27] A. Miné. *A New Numerical Abstract Domain Based on Difference-Bound Matrices.* PADO'2001. LNCS 2053, pp. 155–172. Springer 2001.

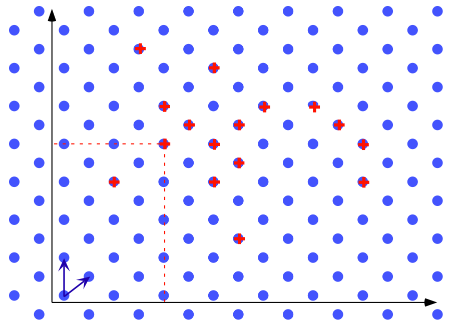
[28] P. Cousot & N. Halbwachs. *Automatic discovery of linear restraints among variables of a program.* ACM POPL, 1978, pp. 84–97.

Effective computable approximations of an [in]finite set of points; Simple congruences [29]



$$\begin{cases} x = 19 \bmod 77 \\ y = 20 \bmod 99 \end{cases}$$

Effective computable approximations of an [in]finite set of points; Linear congruences [30]

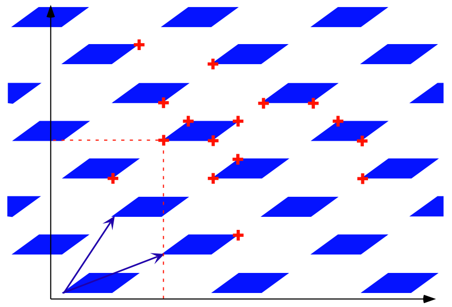


$$\begin{cases} 1x + 9y = 7 \bmod 8 \\ 2x - 1y = 9 \bmod 9 \end{cases}$$

[29] Ph. Granger. *Static Analysis of Arithmetical Congruences.* Int. J. Comput. Math. 30, 1989, pp. 165–190.

[30] Ph. Granger. *Static Analysis of Linear Congruence Equalities among Variables of a Program.* TAPSOFT '91, pp. 169–192. LNCS 493, Springer, 1991.

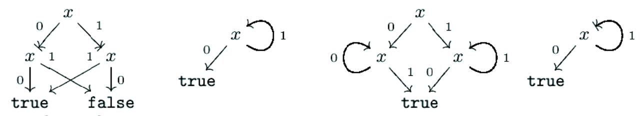
Effective computable approximations of an [in]finite set of points; Trapezoidal linear congruences [31]



$$\begin{cases} 1x + 9y \in [0, 77] \bmod 10 \\ 2x - 1y \in [0, 99] \bmod 11 \end{cases}$$

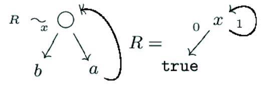
Example of Effective Abstractions of Infinite Sets of Infinite Trees [32]

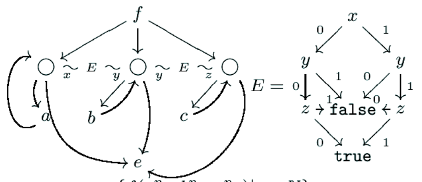
Binary Decision Graphs:



$\{0^\omega, 1^\omega\}$ infinite number of 0's fair vectors ends by 0^ω

Tree Schemata:





$\{f(a^n e, b^n e, c^n e) | n \in \mathbb{N}\}$

$\{a^n b | n \in \mathbb{N}\}$ Note that E is the equality relation.

[31] F. Masdupuy. *Array Operations Abstraction Using Semantic Analysis of Trapezoid Congruences.* ACM ICS '92.

[32] L. Mauborgne. *Improving the Representation of Infinite Trees to Deal with Sets of Trees.* ESOP'00. LNCS 1782, pp. 275–289, Springer, 2000.

On the Design of Program Static Analyzers

- P. Cousot. *The Calculational Design of a Generic Abstract Interpreter.* In *Calculational System Design*, M. Broy and R. Steinbrüggen (Eds). Vol. 173 of NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, pp. 421–505, 1999.
- The corresponding *generic abstract interpreter* (written in Ocaml) is available at URL www.di.ens.fr/~cousot

On the Design of Program Analyzers

- The abstract interpretation theory provides the design principles;
- In practice, one must find the appropriate tradeoff between generality, precision and efficiency;
- There is a full range of program analyzers from

  general purpose analyzers for programming languages

  to

  specific analyzers for a given program.

Specific Static Program Analyzers

- A complete specific analyzer [33] (for a given software or hardware program) can always use a finite abstract domain [34];
- The design of a complete specific analyzer is logically equivalent to a correctness proof of the program;
- Such analyzers are precise but not reusable hence very costly to develop.

General-Purpose Static Program Analyzers

- To handle infinitely many programs for non-trivial properties, a general-purpose analyser must use an infinite abstract domain [35];
- Such analyzers are huge for complex languages hence very costly to develop but reusable;
- There are always programs for which they lead to false alarms;
- Although incomplete, they are very useful for verifying/testing/debugging.

[33] Called a *software model checker?*
[34] P. Cousot. *Partial completeness of abstract fixpoint checking.* SARA'2000. LNAI 1864, pp. 1–25. Springer.
[35] P. Cousot & R. Cousot. *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation.* PLILP'92. LNCS 631, pp. 269–295. Springer.

Parametric Specializable Static Program Analyzers

- The abstraction can be tailored to significant classes of programs (e.g. critical synchronous real-time embedded systems);
- This leads to *very efficient analyzers* with *almost zero-false alarm* even for large programs.

Conclusion

Conclusion on Formal Methods

- Formal methods concentrate on the deductive/exhaustive verification of (abstract) models of the execution of programs;
- Most often this abstraction into a model is *manual* and left completely *informal*, if not tortured to meet the tool limitations;
- Semantics concentrates on the rigorous formalization of the execution of programs;
- So models should abstract the program semantics. This is the whole purpose of Abstract Interpretation!

Conclusion on Abstract Interpretation

- Abstract interpretation provides mathematical foundations of most semantics-based program verification and manipulation techniques;
- In abstract interpretation, the abstraction of the program semantics into an approximate semantics is automated so that one can go *much beyond* examples modelled by hand;
- The abstraction can be tailored to classes of programs so as to design *very efficient analyzers* with *almost zero-false alarm*.

THE END

More references at URL www.di.ens.fr/~cousot.