Formal verification of a static analyzer: abstract interpretation in type theory

Xavier Leroy

Collège de France and Inria, Paris

Séminaire Chocola, 2019-04-04

COLLÈGE DE FRANCE 1530

With thanks to…

Jacques-Henri Jourdan, Vincent Laporte,
David Pichardie, Sandrine Blazy

and all the participants in the ANR Verasco project.

Plan

1. An overview of static analysis
2. Naive abstract interpretation
3. Scaling up: the Verasco project
4. Technical zoom: the abstract interpreter and its proof
5. Conclusions and perspectives

Static analysis in a nutshell

Statically infer properties of a program that hold for all its executions.

At this program point, $0 < x \leq y$ *and pointer p is not* `NULL`.

Emphasis on infer: no help from the programmer.
(E.g. loop invariants are not written in the source.)

Emphasis on statically:

- The inputs to the program are not known.
- The analysis must terminate.
- The analysis must run in reasonable time and space.

Example of properties that can be inferred

Properties of the value of one variable: (value analysis)

$x = a$	constant propagation
$x > 0$ ou $x = 0$ ou $x < 0$	signs
$x \in [a, b]$	intervalles
$x = a \pmod{b}$	congruences
$\texttt{valid}(p[a \ldots b])$	memory validity
p $\texttt{pointsTo}$ x or $p \neq q$	(non-) aliasing between pointers

(a, b, c are constants inferred by the analyzer.)

Example of properties that can be inferred

Properties of several variables: (relational analysis)

$\sum a_i x_i \leq c$	convex polyhedra
$\pm x_1 \pm x_2 \leq c$	octogons
$expr_1 = expr_2$	Herbrand equivalences
$doubly\text{-}linked\text{-}list(p)$	shape analysis

Non-functional properties:

- Memory consumption.
- Worst-case execution time (WCET).

Using static analysis for code optimization

Apply algebraic identities when their conditions are met:

$$\texttt{x / 4} \quad \rightarrow \quad \texttt{x >> 2} \quad \text{if analysis says } \texttt{x} \geq 0$$

$$\texttt{x + 1} \quad \rightarrow \quad \texttt{1} \quad \text{if analysis says } \texttt{x} = 0$$

Optimize array accesses and pointer dereferences:

```
a[i]=1; a[j]=2; x=a[i];   →   a[i]=1; a[j]=2; x=1;
```

if analysis says $\texttt{i} \neq \texttt{j}$

```
*p = a; x = *q;   →   x = *q; *p = a;
```

if analysis says $\texttt{p} \neq \texttt{q}$

Automatic parallelization:

$$loop_1; loop_2 \rightarrow loop_1 \parallel loop_2 \quad \text{if } polyh(loop_1) \cap polyh(loop_2) = \emptyset$$

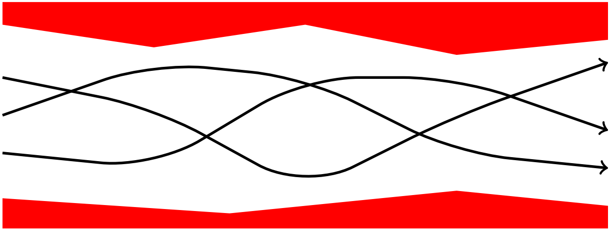
Using static analysis for verification

Use the results of static analysis to prove the absence of certain run-time errors:

$$y \in [a, b] \wedge 0 \notin [a, b] \implies x/y \text{ cannot fail}$$

$$\texttt{valid}(p[a \ldots b]) \wedge i \in [a, b] \implies \texttt{p[i]} \text{ cannot fail}$$

Report an alarm otherwise.



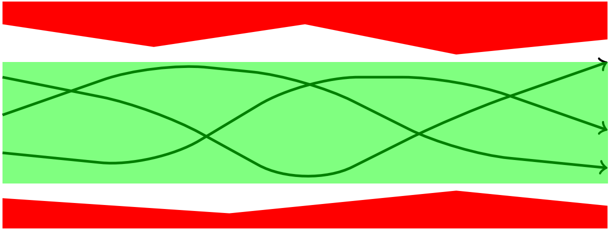
Using static analysis for verification

Use the results of static analysis to prove the absence of certain run-time errors:

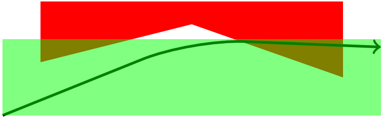
$$y \in [a, b] \wedge 0 \notin [a, b] \implies x/y \text{ cannot fail}$$

$$\texttt{valid}(p[a \ldots b]) \wedge i \in [a, b] \implies \texttt{p[i]} \text{ cannot fail}$$

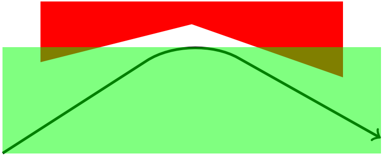
Report an alarm otherwise.



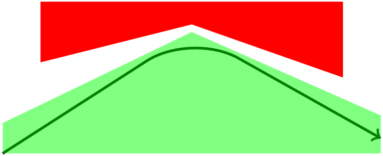
True alarms, false alarms, unsoundness



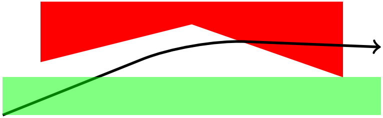
True alarm
(wrong behavior)



False alarm
(analysis too imprecise)



More precise analysis:
the false alarm goes away.



Unsound analyzer:
fails to account for all behaviors

Some properties verifiable by static analysis

Absence of run-time errors:

- Arrays and pointers:
  - No out-of-bound accesses.
  - No dereferencing the null pointer.
  - No access after a `free`.
  - Alignment constraints are respected.
- Integer arithmetic:
  - No division by zero.
  - No (signed) arithmetic overflows.
- Floating-point arithmetic:
  - No arithmetic overflows (result is $\pm\infty$)
  - No undefined operations (result *Not a Number*)
  - No catastrophic cancellation.

Information flow: e.g. "tainting".

Simple programmer-inserted assertions:

e.g. `assert (0 <= x && x < sizeof(tbl))`.

Plan

Basic idea:
analyzing a program is
executing it with a nonstandard semantics

Abstract interpretation in a nutshell

Execute ("interpret") the program with a semantics that:

- Computes over an abstract domain of the desired properties (e.g. "$x \in [a, b]$" for interval analysis) instead of computing with concrete values and states (e.g. numbers).
- Handle Boolean conditions even if they cannot be resolved statically:
  - The `then` and `else` branches of an `if` are both taken $\rightarrow$ joins.
  - Loops and recursions execute arbitrarily many times $\rightarrow$ fixpoints.
- Always terminates.

Examples of abstract interpretation

In the concrete	In the abstract
$\{ x = 3, y = 1 \}$	$\{ x^\# = [0,9], y^\# = [-1,1] \}$

```
z = x + 2 * y;
```

$\{ z = 3 + 2 \times 1 = 5 \}$	$\{ z^\# = [0,9] +^\# 2 \times^\# [-1,1] = [-2,11] \}$
$\{ b = \texttt{true}, x = 3, y = 1 \}$	$\{ b^\# = \top, x^\# = [0,9], y^\# = [-1,1] \}$

```
z = (if b then x else y);
```

$\{ z = 3 \}$	$\{ z^\# = [0,9] \sqcup [-1,1] = [-1,9] \}$

Examples of abstract interpretation

In the concrete	In the abstract
$\{ x = 3, y = 1 \}$	$\{ x^{\#} = [0,9], y^{\#} = [-1,1] \}$

```
z = x + 2 * y;
```

$\{ z = 3 + 2 \times 1 = 5 \}$	$\{ z^{\#} = [0,9] +^{\#} 2 \times^{\#} [-1,1] = [-2,11] \}$
$\{ b = \mathtt{true}, x = 3, y = 1 \}$	$\{ b^{\#} = \top, x^{\#} = [0,9], y^{\#} = [-1,1] \}$

```
z = (if b then x else y);
```

$\{ z = 3 \}$	$\{ z^{\#} = [0,9] \sqcup [-1,1] = [-1,9] \}$

Idea #2:
a variable can have different abstractions
at different program points

Sensitivity to control flow

Imperative variable assignment:

```
x = x + 1;
```

$\{\ x^\# = [0,9]\ \}$

$\{\ x^\# = [1,10]\ \}$

Refining the abstraction at conditionals:

```
if (x == 0) {
    ...
} else {
    ...
}
```

$\{\ x^\# = [0,9]\ \}$

$\{\ x^\# = [0,0]\ \}$

$\{\ x^\# = [1,9]\ \}$

Idea #3:
we can also infer relations
between the values of several variables

Non-relational / relational analysis

Non-relational analysis:

$$\textit{abstract environment} = \textit{variable} \mapsto \textit{abstract value}$$

(Like simple typing environments.)

Relational analysis:
abstract environments are a domain of their own, featuring:

- a semi-lattice structure: $\bot$, $\top$, $\sqsubset$, $\sqcup$
- an abstract operation for assignment / binding.

Example: convex polyhedra, i.e. conjunctions of linear inequalities $\sum a_i x_i \leq c$.

Idea # 4: widening
fixpoints can be computed
even in non-well-founded domains

Fixpoints – the recurring problem

Static analysis of a loop:

```
while (...) {
    ...
}
```

$\{\ e^{\#} = X_0\ \}$

$\{\ e^{\#} = X\ \}$

$\{\ e^{\#} = \Phi(X)\ \}$

Given X_0 (the abstract state before the loop)
and Φ (the transfer function for the loop body),
find X (the loop invariant).

$$X \sqsupseteq X_0 \text{ (first iteration)} \qquad X \sqsupseteq \Phi(X) \text{ (next iterations)}$$

X is, ideally, the smallest fixpoint of $F = X \mapsto X_0 \sqcup \Phi(X)$
or at least any post-fixpoint of F $\quad (X \sqsupseteq F(X))$.

Paradise

Theorem (Kleene)

Let $(A, \sqsubseteq, \bot)$ a partially ordered set such that $\sqsupset$ is well founded (no infinite increasing sequences).
Let $F : A \to A$ an increasing, continuous function.
Then F has a smallest fixpoint, obtained by finite iteration from $\bot$:

$$\exists n,\ \ \bot \sqsubset F(\bot) \sqsubset \ldots \sqsubset F^n(\bot) = F^{n+1}(\bot)$$

Paradise lost

Most abstract domains are not well founded. Examples:

- Integer intervals: $[0, 0] \sqsubset [0, 1] \sqsubset [0, 2] \sqsubset \cdots \sqsubset [0, n] \sqsubset \cdots$
- Environments: *variable* $\mapsto$ *abstract values*.

Moreover, even when Kleene iteration converges, it converges too slowly:

```
x = 0;  while (x <= 10000) { x = x + 1; }
```

(Starting with $x^\# = [0, 0]$, it takes 10000 iterations to reach the fixpoint $x^\# = [0, 10000]$.)

Paradise regained: widening

A widening operator $\nabla : A \to A \to A$ computes a majorant of its second argument in such a way that the following iteration converges always and quickly:

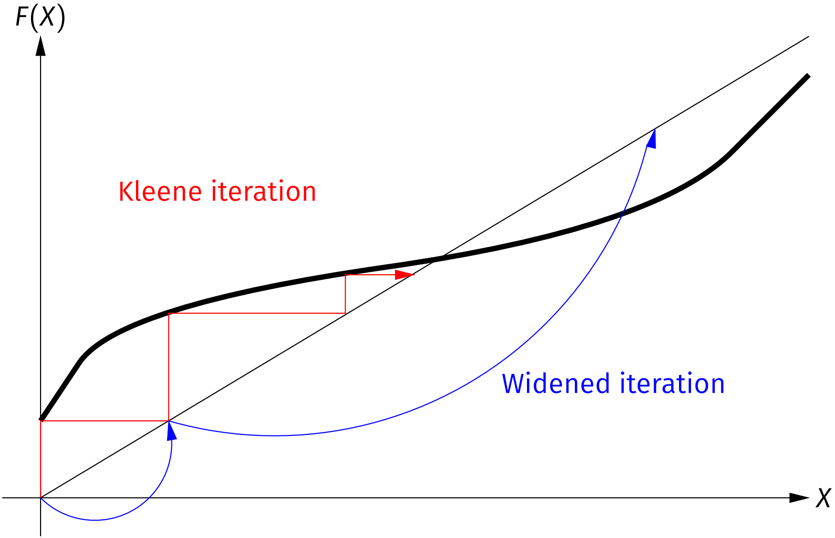
$$X_0 = \bot \qquad X_{i+1} = \begin{cases} X_i & \text{if } F(X_i) \sqsubseteq X_i \\ X_i \mathbin{\nabla} F(X_i) & \text{otherwise} \end{cases}$$

The limit X of this sequence is a post-fixpoint: $F(X) \sqsubseteq X$.

Example: widening for intervals:

$$\begin{aligned}[l_1, u_1] \mathbin{\nabla} [l_2, u_2] = \ & [\texttt{if } l_2 < l_1 \texttt{ then } -\infty \texttt{ else } l_1, \\ & \ \texttt{if } u_2 > u_1 \texttt{ then } \infty \texttt{ else } u_1]\end{aligned}$$

Widening in action



Narrowing the post-fixpoint

The quality of the post-fixpoint can be improved by iterating F some more:

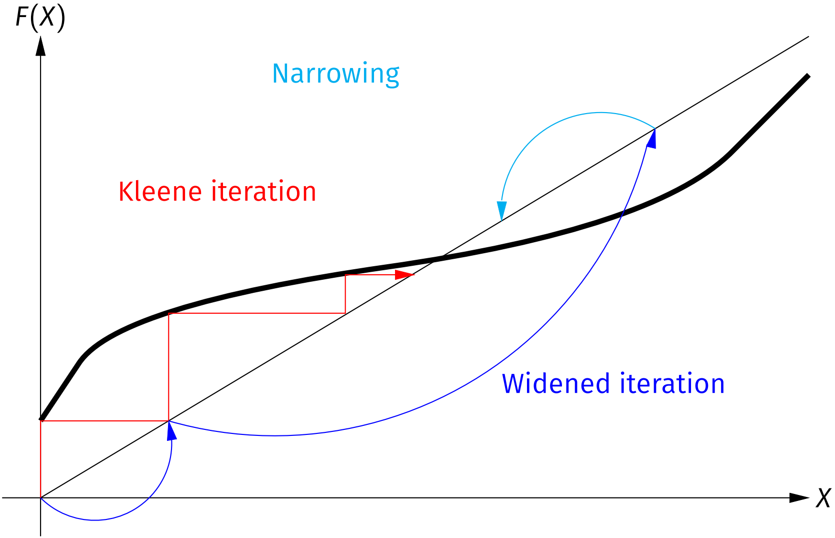
$$Y_0 = \text{a post-fixpoint} \qquad Y_{i+1} = F(Y_i)$$

If F is increasing, each Y_i is a post-fixpoint: $F(Y_i) \sqsubseteq Y_i$.

Often, $Y_i \sqsubset Y_0$, improving the analysis quality.

Iteration can be stopped when Y_i is a fixpoint, or at any time.

Widening plus narrowing in action

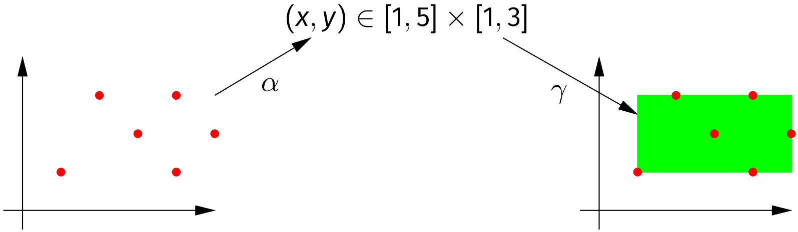


Idea #6: Galois connections:
abstract operators can be calculated
in a systematic, sound, and optimal manner

A Galois connection

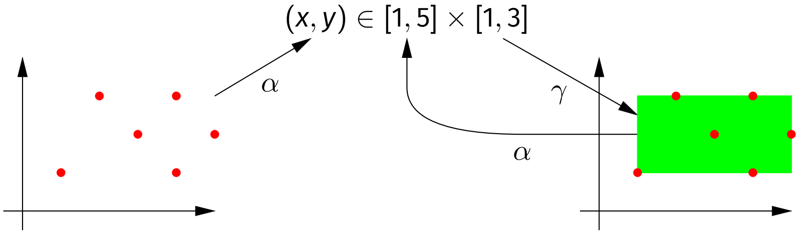
A semi-lattice $\mathcal{A}, \sqsubseteq$ of abstract states and two functions:

- Abstraction α : set of concrete states $\to$ abstract state
- Concretization γ : abstract state $\to$ set of concrete states



For intervals, $\alpha(S) = [\inf S, \sup S]$ and $\gamma([a, b]) = \{x \mid a \leq x \leq b\}$.

Axioms of Galois connections



The adjunction property:

$$\forall a, S, \quad \alpha(S) \sqsubseteq a \iff S \subseteq \gamma(a)$$

or, equivalently:

$$\begin{array}{ll} & \alpha, \gamma \text{ increasing} \\ \wedge & \forall S, \quad S \subseteq \gamma(\alpha(S)) \quad \text{(soundness)} \\ \wedge & \forall a, \quad \alpha(\gamma(a)) \sqsubseteq a \quad \text{(optimality)} \end{array}$$

Calculating abstract operators

For any concrete operator $F : C \to C$ we define its abstraction $F^\# : A \to A$ by

$$F^\#(a) = \alpha\{F(x) \mid x \in \gamma(a)\}$$

This abstract operator is:

- Sound: if $x \in \gamma(a)$ then $F(x) \in \gamma(F^\#(a))$.
- Optimally precise: every a' such that $x \in \gamma(a) \Rightarrow F(x) \in \gamma(a')$ is such that $F^\#(a) \sqsubseteq a'$.

Moreover, an algorithmic definition of $F^\#$ can be calculated from the definition above.

Calculating $+^{\#}$ for intervals

$$
\begin{aligned}
& [a_1, b_1] +^{\#} [a_2, b_2] \\
& \quad = \alpha\{x_1 + x_2 \mid x_1 \in \gamma[a_1, b_1], x_2 \in \gamma[a_2, b_2]\} \\
& \quad = [\ \inf\{x_1 + x_2 \mid a_1 \leq x_1 \leq b_1, a_2 \leq x_2 \leq b_2\}, \\
& \qquad\ \sup\{x_1 + x_2 \mid a_1 \leq x_1 \leq b_1, a_2 \leq x_2 \leq b_2\}\ ] \\
& \quad = [+\infty, -\infty] \text{ if } a_1 > b_1 \text{ or } a_2 > b_2 \\
& \quad = [a_1 + b_1, a_2 + b_2] \text{ otherwise}
\end{aligned}
$$

Note: the intuitive definition $[a_1, b_1] +^{\#} [a_2, b_2] = [a_1 + b_1, a_2 + b_2]$ is sound but not optimal.

Trouble ahead: Galois connections in type theory

Type-theoretic difficulties

Minor issue: the calculations of abstract operators are poorly supported by interactive theorem provers such as Coq:

$$F^{\#} a \;=\; \alpha(\lambda x.P) \;\; \underset{\substack{\uparrow \\ \text{because } \forall x, P \Leftrightarrow P'}}{=} \;\; \alpha(\lambda x.P') \;=\; \ldots$$

Either:

- use setoid equalities everywhere, or
- add extensionality axioms (functional, propositional).

Type-theoretic difficulties

Major issue: γ is easily modeled as

$$\gamma : A \to (C \to \texttt{Prop}) \quad \text{(two-place predicate)}$$

but α is generally not computable as soon as C is infinite:

$$\alpha : (C \to \texttt{Prop}) \to A \quad \text{morally constant functions only?}$$
$$\alpha : (C \to \texttt{bool}) \to A \quad \text{can only query a finite number of } C\text{'s}$$

(E.g. $\alpha(S) = [\inf S, \sup S]$, no more computable than inf and sup.)

$\to$ Need more axioms (description, Hilbert's epsilon).

Fundamental difficulty

For some domains, the abstraction function α does not exist!
(The optimality condition $a \sqsubseteq \alpha(\gamma(a))$ cannot be satisfied.)

Example 1: rational intervals.

$$\alpha\{x \mid x^2 \leq 2\} = ???$$

There is no best rational approximation of $[-\sqrt{2}, \sqrt{2}]$.

Example 2: polyhedra

$$\alpha\{(x, y) \mid x^2 + y^2 \leq 1\} = ???$$

Fundamental difficulty

For some domains, the abstraction function α does not exist!
(The optimality condition $a \sqsubseteq \alpha(\gamma(a))$ cannot be satisfied.)

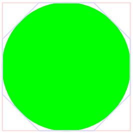
Example 1: rational intervals.

$$\alpha\{x \mid x^2 \leq 2\} = ???$$

There is no best rational approximation of $[-\sqrt{2}, \sqrt{2}]$.

Example 2: polyhedra

$$\alpha\{(x, y) \mid x^2 + y^2 \leq 1\} = ???$$



Fundamental difficulty

For some domains, the abstraction function α does not exist!
(The optimality condition $a \sqsubseteq \alpha(\gamma(a))$ cannot be satisfied.)

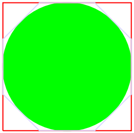
Example 1: rational intervals.

$$\alpha\{x \mid x^2 \leq 2\} = ???$$

There is no best rational approximation of $[-\sqrt{2}, \sqrt{2}]$.

Example 2: polyhedra

$$\alpha\{(x, y) \mid x^2 + y^2 \leq 1\} = ???$$



Fundamental difficulty

For some domains, the abstraction function α does not exist!
(The optimality condition $a \sqsubseteq \alpha(\gamma(a))$ cannot be satisfied.)

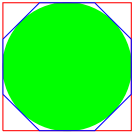
Example 1: rational intervals.

$$\alpha\{x \mid x^2 \leq 2\} = ???$$

There is no best rational approximation of $[-\sqrt{2}, \sqrt{2}]$.

Example 2: polyhedra

$$\alpha\{(x, y) \mid x^2 + y^2 \leq 1\} = ???$$



Plan B:
soundness (γ) is essential,
optimality (α) is optional

Getting rid of α

Remember the two properties of abstract operators $F^{\#}$ calculated from $F^{\#}(a) = \alpha\{F(x) \mid x \in \gamma(a)\}$:

1. Soundness: if $x \in \gamma(a)$ then $F(x) \in \gamma(F^{\#}(a))$.
2. Optimality: every a' such that $x \in \gamma(a) \Rightarrow F(x) \in \gamma(a')$ is such that $F^{\#}(a) \sqsubseteq a'$.

Instead of calculating $F^{\#}$, we can guess a definition for $F^{\#}$, then verify

- property 1: soundness (mandatory!)
- possibly property 2: optimality (optional sanity check).

These proofs only need the concretization relation γ, which is unproblematic.

Soundness first!

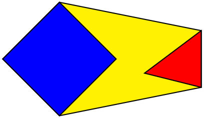
Having made optimality entirely optional, we can further simplify the analyzer and its soundness proof, while increasing its algorithmic efficiency:

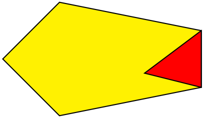
- Abstract operators that return over-approximations (or just $\top$) in difficult / costly cases.
- Join operators $\sqcup$ that return an upper bound for their arguments but not necessarily the least upper bound.
- “Fixpoint” iterations that return a post-fixpoint but not necessarily the smallest (widening + return $\top$ when running out of fuel).
- Validation a posteriori of algorithmically-complex operations, performed by an untrusted external oracle. (Next slide.)

Validation a posteriori

Some abstract operations can be implemented by unverified code if it is easy to validate the results a posteriori by a validator. Only the validator needs to be proved correct.

Example: the join operator $\sqcup$ over polyhedra.





Computing the join (convex hull) vs. Inclusion test (Presburger formula)

The inclusion test can itself use validation a posteriori via Farkas certificates.

Plan

The Verasco project

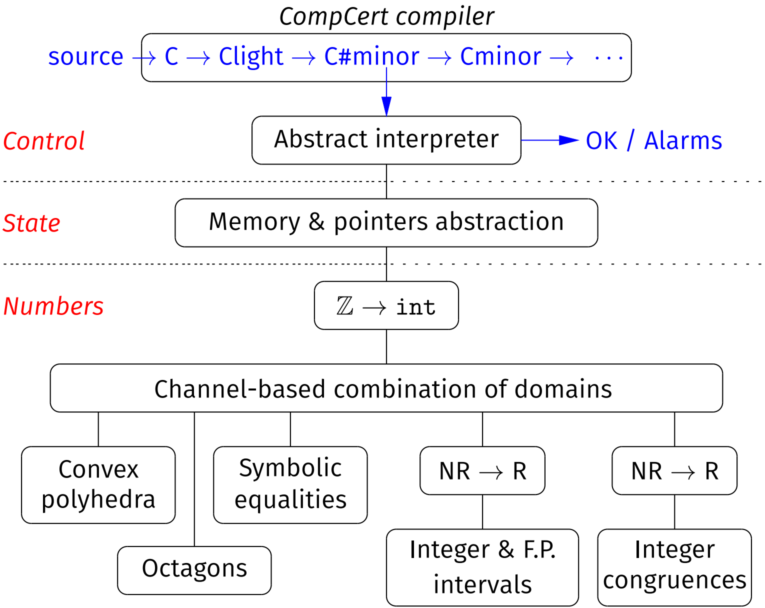
Inria Celtique, Gallium, Antique, Toccata + Verimag + Airbus

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation:

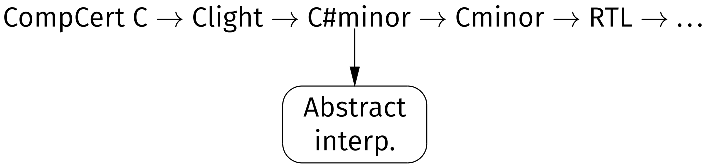
- Language analyzed: the CompCert subset of C.
- Goal: proving the absence of run-time errors.
- Nontrivial abstract domains, including relational domains.
- Modular architecture inspired from Astrée's.
- Decent alarm reporting.

Slogan: if "CompCert = 1/10th of GCC but formally verified", likewise "Verasco = 1/10th of Astrée but formally verified".

Architecture



Upper layer: the abstract interpreter



Connected to the C#minor intermediate language of the CompCert compiler (≈ C without types).

Parameterized by a relational abstract domain for execution states (environment + memory state + call stack).

Local fixpoints for each loop + per-function fixpoint for `goto` + unrolling of functions at call point.

Lower layer: numerical domains

Non-relational:

- Integer intervals (over $\mathbb{Z}$).
- Floating-point intervals (on top of the Flocq library).
- Integer congruences (over $\mathbb{Z}$).

Relational:

- Symbolic equalities *var* $=$ *expr* and facts *expr* $=$ `true` or `false`.
- The VPL library (Fouilhé, Monniaux, Périn, SAS 2013):
  polyhedra with rational coefficients, implemented in OCaml, producing certificates verifiable in Coq.
- Octagons (Jourdan, NSAD 2016):
  direct Coq implementation.

Side contribution: a clean, generic interface for relational domains.

What is a generic interface for a numerical domain?

For a non-relational domain:

- A semilattice $(A, \sqsubseteq)$ of abstract values.
- A concretization relation $\gamma : A \to \mathbb{Z} \to \texttt{Prop}$
- "Forward" abstract operators such as

```
forward_unop: unary_operation → A → A+⊥;
forward_unop_sound: ∀ op x a,
   x ∈ γ a -> eval_unop op x ⊆ γ (forward_unop op x);
```

- "Backward" abstract operators (to refine abstractions based on the results of conditionals) such as

```
backward_unop: unary_operation → A → A → A+⊥;
backward_unop_sound: ∀ op x a res b,
  x ∈ γ a -> res ∈ γ b -> res ∈ eval_unop op x ->
  x ∈ γ (backward_unop op a b);
```

What is a generic interface for a numerical domain?

For a relational domain, the main abstract operations are:

- `assign` *var* $=$ *expr*
- `forget` *var* $=$ *any-value*
- `assume` *expr* is true or *expr* is false

var are program variables or abstract memory locations.

expr are simple expressions ($+ \; - \; \times$ `div` `mod` $\ldots$) over variables and constants.

To report alarms, we also need to query the domain, e.g. "is $\mathtt{x} < \mathtt{y}$?" or "is $\mathtt{x} \bmod 4 = 0$?". The basic query is

- `get_itv` *expr* $\rightarrow$ *variation interval*

(Next slide: Coq interface.)

The abstract operations

```
Class ab_machine_env (t var: Type): Type :=
  { leb: t -> t -> bool
  ; top: t
  ; join: t -> t -> t
  ; widen: t -> t -> t
  ; forget: var -> t -> t+⊥
  ; assign: var -> nexpr var -> t -> t+⊥
  ; assume: nexpr var -> bool -> t -> t+⊥
  ; nonblock: nexpr var -> t -> bool
  ; get_itv: nexpr var -> t -> num_val_itv+⊤+⊥
```

… and their specifications

```
  ; γ : t -> ℘ (var->num_val)
  ; gamma_monotone: forall x y,
      leb x y = true -> γ x ⊆ γ y;
  ; gamma_top: forall x, x ∈ γ top;
  ; join_sound: forall x y,
      γ x ∪ γ y ⊆ γ (join x y)
  ; forget_correct: forall x ρ n ab,
      ρ ∈ γ ab -> (upd ρ x n) ∈ γ (forget x ab)
  ; assign_correct: forall x e ρ n ab,
      ρ ∈ γ ab -> n ∈ eval_nexpr ρ e ->
      (upd ρ x n) ∈ γ (assign x e ab)
  ; assume_correct: forall e ρ ab b,
      ρ ∈ γ ab -> of_bool b ∈ eval_nexpr ρ e ->
      ρ ∈ γ (assume e b ab)
  ; nonblock_correct: forall e ρ ab,
      ρ ∈ γ ab -> nonblock e ab = true -> block_nexpr ρ e -> False
  ; get_itv_correct: forall e ρ ab,
      ρ ∈ γ ab -> (eval_nexpr ρ e) ⊆ γ (get_itv e ab)
}.
```

The middle layer: domain transformers

Communications between numerical domains.

From mathematical integers to N-bit machine integers
(accounts for overflow and wrap-around).

Memory and pointer domain:
1 abstract memory cell = 1 variable of the numerical domains
Plus: points-to information and type information.

Plan

1. An overview of static analysis
2. Naive abstract interpretation
3. Scaling up: the Verasco project
4. Technical zoom: the abstract interpreter and its proof
5. Conclusions and perspectives

Abstract interpretation of structured control

For a simple imperative language like IMP:

$$F(s, \text{abstract state “before” } s) = \text{abstract state “after” } s + \texttt{alarm}$$

Follows the structure of statement s.

No need to talk about program points (unlike in dataflow analysis).

Some cases of the abstract interpreter F

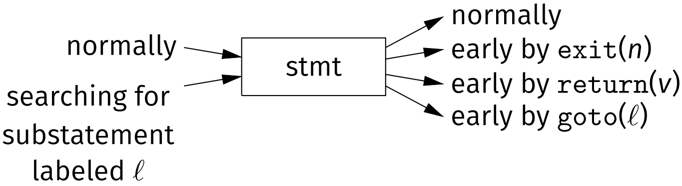
$$\begin{array}{rcl} F((s_1; s_2), A) & = & F(s_2, F(s_1, A)) \\ F((\texttt{IF}\ b\ \texttt{THEN}\ s_1\ \texttt{ELSE}\ s_2), A) & = & F(s_1, A \wedge b) \sqcup F(s_2, A \wedge \neg b) \\ F((\texttt{WHILE}\ b\ \texttt{DO}\ s\ \texttt{DONE}), A) & = & \texttt{pfp}\ (\lambda X.\ A \sqcup F(s, X \wedge b)) \wedge \neg b \end{array}$$

Note: taking a post-fixpoint `pfp` at every loop.

Notation: $A \wedge b$ is A where we assert that b is true.

Control flow in the C#minor language

Unlike in IMP, a C#minor statement can terminate in several different ways, and can also be entered in several ways:



The abstract interpreter becomes:

$$F(s, A_i, A_l) = (A_o, A_r, A_e, A_g) + \mathtt{alarm}$$

A_i : abstract state (normal entry)
A_l : label $\rightarrow$ abstract state (incoming `goto`)
A_o : abstract state (normal termination)
A_r : abstract value $\times$ abstract state (early return)
A_e : exit level $\rightarrow$ abstract state
A_g : label $\rightarrow$ abstract state (outgoing `goto`)

Proving the soundness of an abstract interpreter

For IMP, a simple soundness property:

> *If* $F(s, A) \neq \texttt{alarm}$ *and* $m \in \gamma(A)$*,*
> *statement s, started in memory m, does not go wrong;*
> *moreover, if it terminates with memory* m'*, then* $m' \in \gamma(F(s, A))$*.*

Can be stated formally and proved directly using big-step operational semantics with error rules:

$m \vdash s \Rightarrow m'$	safe termination on state m'
$m \vdash s \Rightarrow \texttt{err}$	termination by going wrong

> *If* $F(s, A) \neq \texttt{alarm}$ *and* $m \in \gamma(A)$*,*
> *then* $m \vdash s \not\Rightarrow \texttt{err}$*,*
> *and* $m \vdash s \Rightarrow m'$ *implies* $m' \in \gamma(F(s, A))$*.*

The C#minor operational semantics

A big-step semantics for C#minor is painful to define, owing to `goto` statements. Instead, we use CompCert's small-step semantics with continuations:

$$(s, k, m) \rightarrow (s', k', m') \rightarrow \cdots$$

where	s	statement under focus
	k	continuation term (what to do after s terminates)
	m	current memory state and environment

Representative rules for IMP:

$$\begin{array}{rcll}
((s_1; s_2), k, m) & \rightarrow & (s_1, \mathtt{Kseq}\ s_2\ k, m) & \\
((\mathtt{IF}\ b\ \mathtt{THEN}\ s_1\ \mathtt{ELSE}\ s_2), k, m) & \rightarrow & (s_1, k, m) & \text{if } b \Rightarrow \mathtt{true} \\
((\mathtt{IF}\ b\ \mathtt{THEN}\ s_1\ \mathtt{ELSE}\ s_2), k, m) & \rightarrow & (s_2, k, m) & \text{if } b \Rightarrow \mathtt{false} \\
(\mathtt{skip}, \mathtt{Kseq}\ s\ k, m) & \rightarrow & (s, k, m) &
\end{array}$$

Using a Hoare logic

(Yves Bertot, 2005)

Proving the abstract interpreter sound w.r.t. the small-step semantics is feasible but painful. Instead, we break the proof in two steps, using a weak Hoare logic:

- Step 1: "Hoare soundness" of the abstract interpreter:
  If $F(s, A) = A'$ (and not `alarm`),
  then the weak Hoare triple $\{\gamma(A)\}\ s\ \{\gamma(A')\}$ is derivable.

- Step 2: soundness of the Hoare logic w.r.t. the operational semantics.

NB: for C#, we need Hoare "7-tuples"
$\{\gamma(A_i), \gamma(A_l)\}\ s\ \{\gamma(A_o), \gamma(A_r), \gamma(A_e), \gamma(A_g)\}$.

Small-step soundness of a Hoare logic

(Andrew Appel and Sandrine Blazy, 2007)

Definitions:

- A configuration (s, k, m) is safe for n steps if no sequence of at most n transitions starting with (s, k, m) reaches a "going wrong" state.
- A continuation k is safe for n steps w.r.t. postcondition Q if, for all memory states m satisfying Q, the configuration $(\texttt{skip}, k, m)$ is safe for n steps.

Theorem (soundness of a weak Hoare logic)

If the Hoare triple $\{P\}\ s\ \{Q\}$ holds, then for all n, all continuations k safe for n steps w.r.t. Q, and all memory states m satisfying P, the configuration (s, k, m) is safe for n steps.

Two ways to define the Hoare logic

Shallow embedding: (Appel and Blazy)

- use the soundness theorem as the definition of $\{P\}\ s\ \{Q\}$;
- show the usual Hoare logic rules as lemmas.

Deep embedding: (what we use in CompCert)

- define $\{P\}\ s\ \{Q\}$ as a coinductive predicate, with each rule as a constructor;
- prove the soundness theorem by induction on the number n of steps.

(The coinductive definition helps to handle function calls just by unrolling of the function definition.)

Conjunction and disjunction rules

The Verasco abstract interpreter contains some heuristics (unrolling of the last N iterations of a loop) whose soundness proof makes use of unusual Hoare logic rules:

$$\frac{\{P_1\}\ s\ \{Q\} \quad \{P_2\}\ s\ \{Q\}}{\{P_1 \lor P_2\}\ s\ \{Q\}} \qquad \frac{\{P\}\ s\ \{Q_1\} \quad \{P\}\ s\ \{Q_2\}}{\{P\}\ s\ \{Q_1 \land Q_2\}}$$

These rules are admissible in the deep embedding approach (with the coinductive predicate), but we could not prove the rule on the right (conjunction) in the shallow embedding approach.

Plan

1. An overview of static analysis
2. Naive abstract interpretation
3. Scaling up: the Verasco project
4. Technical zoom: the abstract interpreter and its proof
5. Conclusions and perspectives

Status of Verasco

It works!

- Fully proved (30 000 lines of Coq)
- Executable analyzer obtained by extraction.
- Able to show absence of run-time errors in small but nontrivial C programs.

It needs improving!

- Some loops need manual unrolling
(to show that an array is fully initialized at the end of a loop).
- Analysis is slow (up to one minute for 100 LOC).

Future work

- Improve algorithmic efficiency, esp. sharing between representations of abstract states (hash-consing?).
- More precise and more efficient abstractions of memory states. (Cf. Antoine Miné's memory domain, LCTES 2006.)
- More (combinations of) abstract domains, e.g. trace partitioning, array-specific domains.
- Debugging the precision of the analyses.

Conclusions

Trying to bridge elegant foundations and nitty-gritty details (low-level language, algorithmic efficiency).

Abstract interpretation is an effective guideline once we forget about optimality of the analysis.

The modular architecture of the analyzer and its well-specified interfaces are essential.

One step at a time…

… we get closer to the formal verification of the tools that participate in the production and verification of critical embedded software.