

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

Xavier Leroy

Inria Paris-Rocquencourt

TYPES meeting, 2014-05-14



In memoriam Radhia Cousot, † 2014



With thanks to...

David Pichardie

and the Verasco project team:

Sandrine Blazy, Vincent Laporte, André Maronèze (Rennes)
Jacques-Henri Jourdan, Jérôme Feret, Xavier Rival, Arnaud Spiwack
(Paris-Rocquencourt)
Alexis Fouilhé, David Monniaux, Michael Périn (Grenoble)
Jean Souyris (Airbus)

Plan

- 1 An overview of static analysis
- 2 Abstract interpretation, in set theory and in type theory
- 3 Scaling up: the Verasco project
- 4 Conclusions and future work

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
$\text{valid}(p[a \dots b])$	memory validity
$p \text{ pointsTo } x$ 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)

$\sum a_i x_i \leq c$ polyhedra

$\pm x_1 \pm \dots \pm x_n \leq c$ octogons

$expr_1 = expr_2$ Herbrand equivalences

doubly-linked-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:

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

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

Optimize array accesses and pointer dereferences:

$$a[i]=1; a[j]=2; x=a[i]; \rightarrow a[i]=1; a[j]=2; x=1; \\ \text{if analysis says } i \neq j$$

$$*p = a; x = *q; \rightarrow x = *q; *p = a; \\ \text{if analysis says } p \neq q$$

Automatic parallelization:

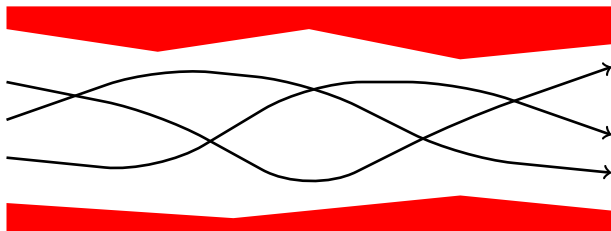
$$loop_1; loop_2 \rightarrow loop_1 \parallel loop_2 \quad \text{if } polyh(loop_1) \cap polyh(loop_2) = \emptyset$$

Using static analysis for verification

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

$$x \in [a, b] \wedge 0 \notin [a, b] \implies x/y \text{ cannot fail}$$
$$\text{valid}(p[a \dots b]) \wedge i \in [a, b] \implies p[i] \text{ cannot fail}$$

Report an **alarm** otherwise.

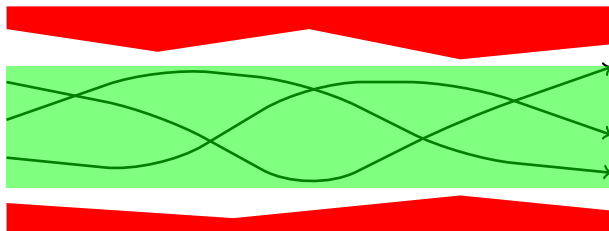


Using static analysis for verification

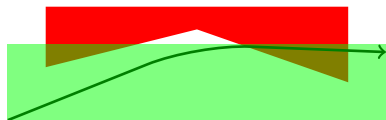
Use the results of static analysis to prove the absence of certain run-time errors:

$$x \in [a, b] \wedge 0 \notin [a, b] \implies x/y \text{ cannot fail}$$
$$\text{valid}(p[a \dots b]) \wedge i \in [a, b] \implies p[i] \text{ cannot fail}$$

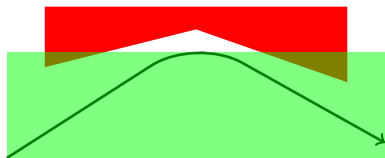
Report an **alarm** otherwise.



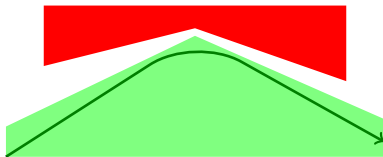
True alarms, false alarms



True alarm
(wrong behavior)



False alarm
(analysis too imprecise)



More precise analysis (polyhedron instead of intervals):
the false alarm goes away.

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.

Simple programmer-inserted assertions:

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

Plan

- 1 An overview of static analysis
- 2 Abstract interpretation, in set theory and in type theory
- 3 Scaling up: the Verasco project
- 4 Conclusions and future work

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

$$\{ x = 3, y = 1 \}$$

$$\{ x^\# = [0, 9], y^\# = [-1, 1] \}$$

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

$$\{ z = 3 + 2 \times 1 = 5 \}$$

$$\{ z^\# = [0, 9] +^\# 2 \times^\# [-1, 1] = [-2, 11] \}$$

Examples of abstract interpretation

In the concrete

In the abstract

$\{ x = 3, y = 1 \}$

$\{ x^\# = [0, 9], y^\# = [-1, 1] \}$

$z = x + 2 * y;$

$\{ z = 3 + 2 \times 1 = 5 \}$

$\{ z^\# = [0, 9] +^\# 2 \times^\# [-1, 1] = [-2, 11] \}$

$\{ b = \text{true}, x = 3, y = 1 \}$

$\{ b^\# = \top, x^\# = [0, 9], y^\# = [-1, 1] \}$

$z = (\text{if } b \text{ then } x \text{ else } y);$

$\{ z = 3 \}$

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

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

Sensitivity to control flow

Imperative variable assignment:

<code>x = x + 1;</code>	$\{ x^\# = [0, 9] \}$
	$\{ x^\# = [1, 10] \}$

Refining the abstraction at conditionals:

<code>if (x == 0) {</code>	$\{ x^\# = [0, 9] \}$
<code>...</code>	
<code>} else {</code>	$\{ x^\# = [0, 0] \}$
<code>...</code>	
<code>}</code>	$\{ x^\# = [1, 9] \}$

Sensitivity to control flow

Contrast with dependent pattern-matching, where the type of the scrutinee is unchanged, but additional facts are added to the environment.

```
match eq_dec x 0 with
| left  (EQ: x = 0)    => ...
| right (NEQ: x <> 0) => ...
end.
```

```
match x as z return x = z -> T with
| None    => fun (P: x = None)    => ...
| Some y => fun (P: x = Some y) => ...
end (refl_equal x).
```

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

Non-relational / relational analysis

Non-relational analysis:

abstract environment = *variable* \mapsto *abstract value*

(Like simple typing environments.)

Relational analysis:

abstract environments are a domain of their own, featuring:

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

Example: 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).

$X \sqsupseteq X_0$ (first iteration) $X \sqsupseteq \Phi(X)$ (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 ($X \sqsupseteq F(X)$).

Theorem (Tarski)

Let (A, \sqsubseteq, \perp) a partially ordered set such that \sqsubseteq is well founded (no infinite increasing sequences).

Let $F : A \rightarrow A$ an increasing function.

Then F has a smallest fixpoint, obtained by finite iteration from \perp :

$$\exists n, \perp \sqsubseteq F(\perp) \sqsubseteq \dots \sqsubseteq F^n(\perp) = F^{n+1}(\perp)$$

Paradise lost

Most abstract domains are not well founded. Examples:

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

Moreover, even when Tarski 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 $\nabla : A \rightarrow A \rightarrow A$ computes a majorant of its second argument in such a way that the following iteration converges always and quickly:

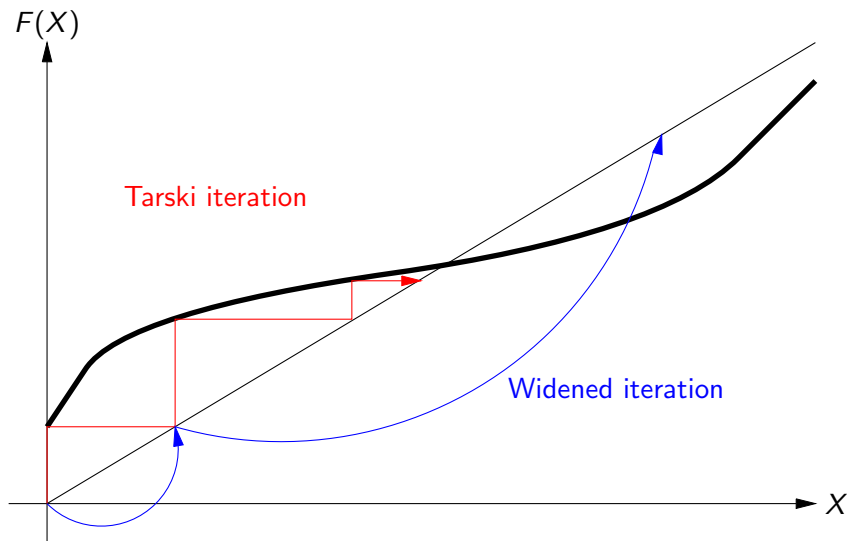
$$X_0 = \perp \quad X_{i+1} = \begin{cases} X_i & \text{if } F(X_i) \sqsubseteq X_i \\ X_i \nabla F(X_i) & \text{otherwise} \end{cases}$$

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

Example: widening for intervals:

$$[l_1, u_1] \nabla [l_2, u_2] = \begin{cases} \text{if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \\ \text{if } u_2 > u_1 \text{ then } \infty \text{ else } u_1 \end{cases}$$

Widening in action



Narrowing the post-fixpoint

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

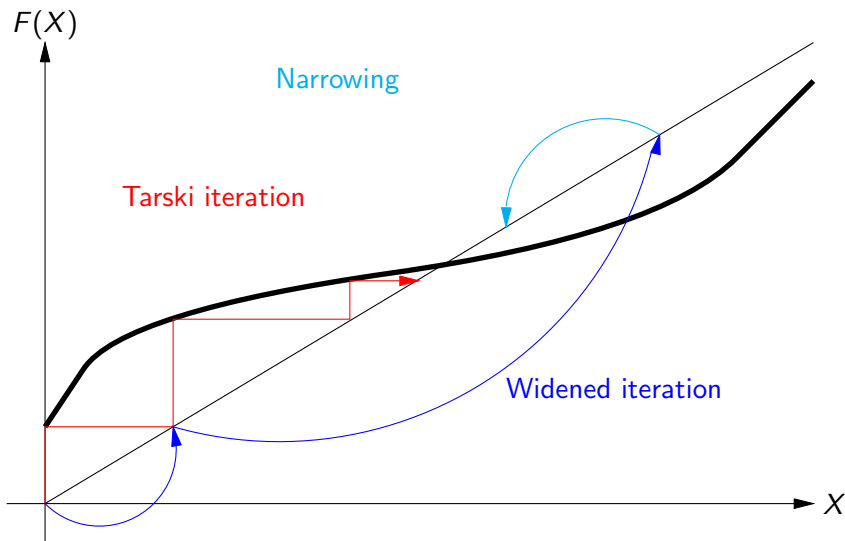
$$Y_0 = \text{a post-fixpoint} \quad Y_{i+1} = F(Y_i)$$

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



Specification of widening

A simple variation on the constructive definition of well foundedness:

```
Inductive Acc: A -> Prop :=  
| Acc_intro:  $\forall x,$   
  ( $\forall y, y \sqsubset x \rightarrow \text{Acc } y$ )  $\rightarrow$   
  Acc x.
```

```
Definition well_founded :=  
 $\forall x, \text{Acc } x.$ 
```

```
Inductive AccW: A -> Prop :=  
| AccW_intro:  $\forall x,$   
  ( $\forall y, y \sqsubset x \rightarrow \text{AccW } (x \nabla y)$ )  $\rightarrow$   
  AccW x.
```

```
Definition widening_correct :=  
 $\forall x, \text{AccW } x.$ 
```

Specification of widening

A simple variation on the constructive definition of well foundedness:

Inductive Acc: A -> Prop := Acc_intro: $\forall x,$ $(\forall y, y \sqsubseteq x \rightarrow \text{Acc } y) \rightarrow$ Acc x.	Inductive AccW: A -> Prop := AccW_intro: $\forall x,$ $(\forall y, y \sqsubseteq x \rightarrow \text{AccW } (x \nabla y)) \rightarrow$ AccW x.
--	---

Definition well_founded := $\forall x, \text{Acc } x.$	Definition widening_correct := $\forall x, \text{AccW } x.$
---	--

Even Coq understands that widened iteration terminates:

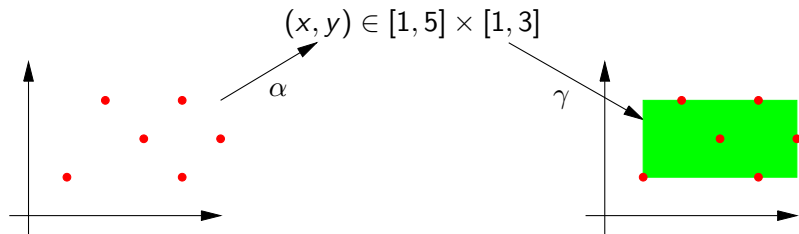
```
Fixpoint postfixpoint (F: A->A) (x: A) (acc: AccW x) {struct acc} :=  
  let y := F x in  
  match decide (x  $\sqsubseteq$  y) with  
  | left LE => x  
  | right GT => postfixpoint F (x  $\nabla$  y) (AccW_inv x acc y GT)  
end.
```


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

A Galois connection

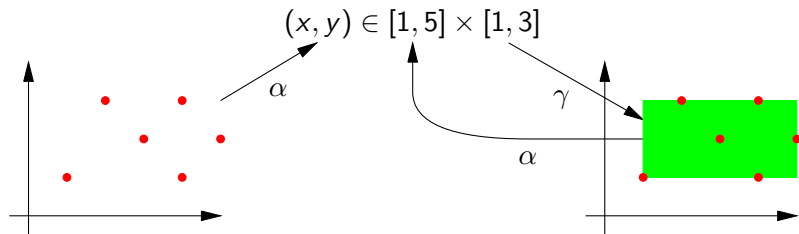
A semi-lattice \mathcal{A}, \sqsubseteq of abstract states and two functions:

- **Abstraction function** α : set of concrete states \rightarrow abstract state
- **Concretization function** γ : abstract state \rightarrow set of concrete states



E.g. 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:

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

or, equivalently:

α increasing

\wedge γ increasing

\wedge $\forall S, S \subseteq \gamma(\alpha(S))$ (soundness)

\wedge $\forall a, \alpha(\gamma(a)) \sqsubseteq a$ (optimality)

Calculating abstract operators

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

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

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:

$$F^{\#} a = \alpha(\lambda x. P) = \alpha(\lambda x. P') = \dots$$

↑
because $\forall x, P \Leftrightarrow P'$

Either:

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

Type-theoretic difficulties

Major issue: γ is easily modeled as

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

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

$\alpha : (C \rightarrow \text{Prop}) \rightarrow A$ morally constant functions only?

$\alpha : (C \rightarrow \text{bool}) \rightarrow A$ can only query a finite number of C 's

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

→ 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: intervals of rationals.

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

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

Fundamental difficulty

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

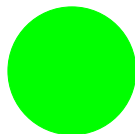
Example 1: intervals of rationals.

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

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

Example 2: polyhedra

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



(It works in practice nonetheless, because the abstract interpreter and abstract operators are set up in such a way that non-abstractible sets like the above never occur.)

Fundamental difficulty

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

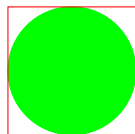
Example 1: intervals of rationals.

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

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

Example 2: polyhedra

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



(It works in practice nonetheless, because the abstract interpreter and abstract operators are set up in such a way that non-abstractible sets like the above never occur.)

Fundamental difficulty

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

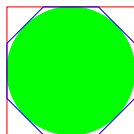
Example 1: intervals of rationals.

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

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

Example 2: polyhedra

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



(It works in practice nonetheless, because the abstract interpreter and abstract operators are set up in such a way that non-abstractible sets like the above never occur.)

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!

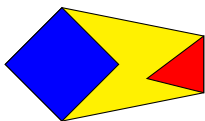
Having made optimality entirely optional, we can further simplify the analyzer and its soundness proof, while increasing its algorithmic efficiency:

- 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

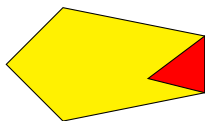
Some abstract operations can be implemented by unverified code if it is easy to validate the results a posteriori by a validator. Only the validator needs to be proved correct.

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.
(Cf. talk by Foulhe, Boulmé and Périn.)

Plan

- 1 An overview of static analysis
- 2 Abstract interpretation, in set theory and in type theory
- 3 Scaling up: the Verasco project**
- 4 Conclusions and future work

The Verasco project

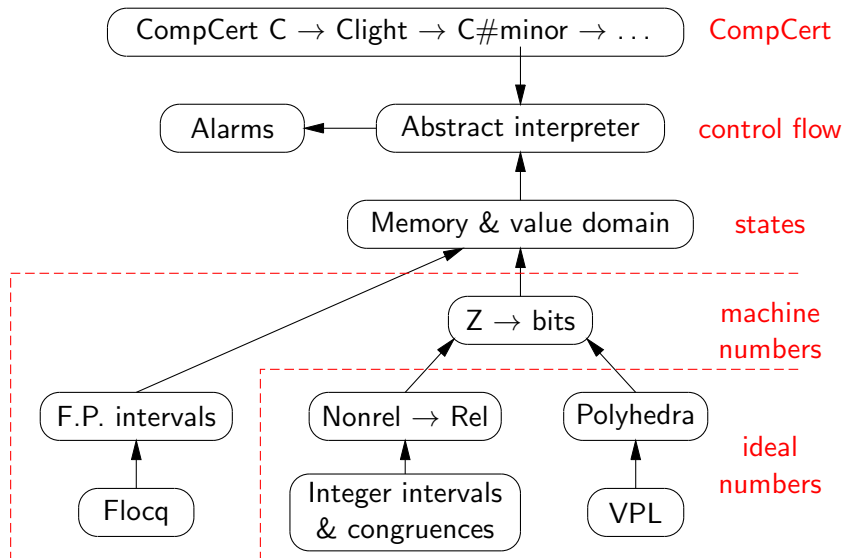
Inria Celtique, Gallium, Abstraction, Toccata + Verimag + Airbus

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

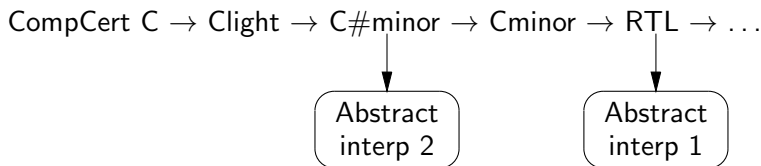
- Language analyzed: the CompCert subset of C.
- Nontrivial abstract domains, including relational domains.
- Modular architecture inspired from Astrée's.
- Decent alarm reporting.

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 intermediate languages of the CompCert compiler.

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

- 1 Abstract interpreter for RTL (Blazy, Maronèze, Pichardie, SAS 2013)
Unstructured control \rightarrow per-function fixpoints (Bourdoncle).
- 2 Abstract interpreter for C#minor (Jourdan, in progress)
Local fixpoints for each loop + per-function fixpoint for goto + per-program fixpoint for function calls.

Lower layer: numerical domains

Non-relational:

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

Relational:

- The VPL library (Fouilhé, Monniaux, Périn, SAS 2013):
polyhedra with rational coefficients, implemented in OCaml,
producing certificates verifiable in Coq.
- Integration in progress in Verasco.

What is a generic interface for a numerical domain?

For a non-relational domain:

- A semilattice (A, \sqsubseteq) of abstract values.
- A concretization relation $\gamma : A \rightarrow \mathbf{Z} \rightarrow \text{Prop}$
- Abstract operators such as

```
add: A -> A -> A;  
add_sound: forall a b x y,  
  x ∈ γ a -> y ∈ γ b -> (x + y) ∈ γ (add a b);
```

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

```
eq_inv: A -> A -> bool -> A * A;  
eq_inv_sound: forall a b c x y,  
  x ∈ γ a -> y ∈ γ b ->  
  (if c then x = y else x <> y) ->  
  x ∈ γ (fst (eq_inv a b c))  
  ∧ y ∈ γ (snd (eq_inv a b c));
```

What is a generic interface for a numerical domain?

For a relational domain, the main abstract operations are:

- `assign` $var = expr$
- `forget` $var = any\text{-}value$
- `assume` $expr$ is true or $expr$ is false

var are program variables or abstract memory locations.

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

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

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

(Next slide: Coq interface.)

```

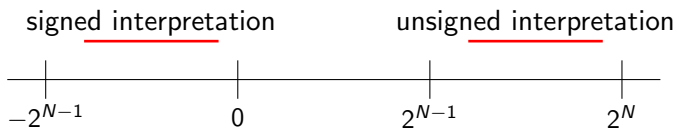
Class ab_ideal_env (var t:Type) '{EqDec var}: Type := {
  id_wl:> weak_lattice t;
  id_gamma:> gamma_op t (var->ideal_num);
  id_adom:> adom t (var->ideal_num) id_wl id_gamma;
  get_itv: iexpr var -> t -> IdealIntervals.abs+⊥;
  assign: var -> iexpr var -> t -> t+⊥;
  forget: var -> t -> t+⊥;
  assume: iexpr var -> bool -> t -> t+⊥;
  get_itv_sound: forall e ρ ab,
    ρ ∈ γ ab ->
    eval_iexpr ρ e ⊆ γ (get_itv e ab);
  assign_sound: forall x e ρ n ab,
    ρ ∈ γ ab ->
    n ∈ eval_iexpr ρ e ->
    (upd ρ x n) ∈ γ (assign x e ab);
  forget_sound: forall x ρ n ab,
    ρ ∈ γ ab ->
    (upd ρ x n) ∈ γ (forget x ab);
  assume_sound: forall c ρ ab b,
    ρ ∈ γ ab ->
    (INz (if b:bool then 1 else 0)) ∈ eval_iexpr ρ c ->
    ρ ∈ γ (assume c b ab)
}.

```

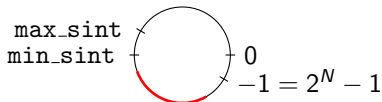

Machine integers vs. mathematical integers

Machine integers = N -bit vectors, with arithmetic modulo 2^N , and two possible interpretations (signed or unsigned).

For intervals, ad-hoc solutions based on pairs of **Z**-intervals:



or on cyclic intervals:



What about relational domains?

A domain transformer for machine integers

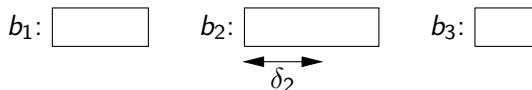
(J-H. Jourdan)

Given a relational domain (A, γ) over \mathbf{Z} , construct a relational domain over N -bit machine integers as follows:

- Same abstract domain A .
- New concretization:
$$\gamma'(a) = \{b : \text{bitvect}(N) \mid \exists n : \mathbf{Z}, n \in \gamma(a) \wedge n = b \pmod{2^N}\}$$
- Same abstract operators for addition, subtraction, multiplication.
- For other operators (comparisons, division, ...): try first to reduce the ideal integers modulo 2^N to the interval $[0, 2^N)$ or $[-2^{N-1}, 2^{N-1})$, depending on whether the operation is signed or unsigned.

Middle layer: abstracting memory and state

The CompCert memory model: memory location = block $b \times$ offset δ .



Abstraction of offsets \rightarrow integer domain.

Abstraction of blocks:

- First attempt (Pichardie): 1 concrete block = 1 abstract block “global variable x ” or “local variable y of function f ”.
- Recursion, dynamic allocation \rightarrow need for imprecise abstract blocks (standing for several concrete blocks).
- In progress (Laporte): abstract memory model with block fusion and weak updates.

Plan

- 1 An overview of static analysis
- 2 Abstract interpretation, in set theory and in type theory
- 3 Scaling up: the Verasco project
- 4 Conclusions and future work

Conclusions

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

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

Future work

Much remains to be done to reach a realistic static analyzer:

- “Good” abstractions for memory.
- More (combinations of) abstract domains:
symbolic equalities, reduced products, trace partitioning, ...
- Algorithmic efficiency needs more work, esp. on sharing between representations of abstract states.
- Good alarm reports.
- Debugging the precision of the analyses.

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.

