Verasco: Formal verification of a C static analyzer based on abstract interpretation

Jacques-Henri Jourdan, Vincent Laporte
Sandrine Blazy, *Xavier Leroy*, David Pichardie

Inria / U. Rennes 1 / ENS Rennes

Workshop on Realistic Program Verification, 2015-12-02

Plan

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)

$\sum a_i x_i \leq c$	polyhedra
$\pm x_1 \pm \cdots \pm x_n \leq c$	octagons
$expr_1 = expr_2$	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:

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

Optimize array accesses and pointer dereferences:

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

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

Automatic parallelization:

$$\mathit{loop}_1; \mathit{loop}_2 \rightarrow \mathit{loop}_1 \parallel \mathit{loop}_2 \quad \text{if } \mathit{polyh}(\mathit{loop}_1) \cap \mathit{polyh}(\mathit{loop}_2) = \emptyset$$

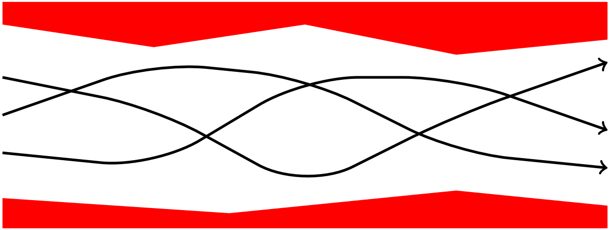
Using static analysis for verification

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.



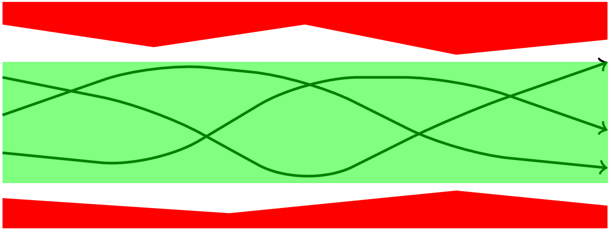
Using static analysis for verification

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

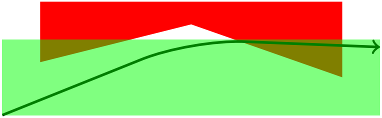
$$y \in [a, b] \wedge 0 \notin [a, b] \implies x/y \text{ cannot fail}$$

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

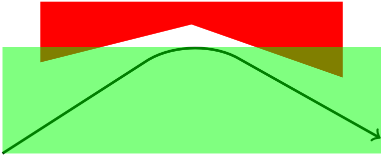
Report an alarm otherwise.



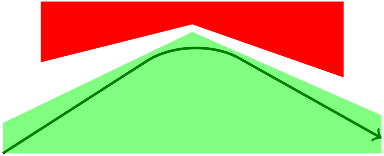
True alarms, false alarms



True alarm
(wrong behavior)



False alarm
(analysis too imprecise)



More precise analysis (octagons 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

Abstract interpretation in a nutshell

Execute ("interpret") the program with a nonstandard 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).
- Handles 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.

Example of abstract interpretation with intervals

$x \in [-\infty, \infty]$

```
IF x < 0 THEN
    x := 0;
ELSE IF x > 1000 THEN
    x := 1000;
ELSE
    SKIP;
ENDIF
```

Example of abstract interpretation with intervals

```
IF x < 0 THEN
    x := 0;
ELSE IF x > 1000 THEN
    x := 1000;
ELSE
    SKIP;
ENDIF
```

$\mathrm{x} \in [-\infty, \infty]$

$\mathrm{x} \in [0, 0]$

Example of abstract interpretation with intervals

$\mathrm{x} \in [-\infty, \infty]$

```
IF x < 0 THEN
    x := 0;                 x ∈ [0, 0]
ELSE IF x > 1000 THEN
    x := 1000;              x ∈ [1000, 1000]
ELSE
    SKIP;
ENDIF
```

Example of abstract interpretation with intervals

$x \in [-\infty, \infty]$

```
IF x < 0 THEN
    x := 0;
ELSE IF x > 1000 THEN
    x := 1000;
ELSE
    SKIP;
ENDIF
```

`x := 0;`	$x \in [0, 0]$
`x := 1000;`	$x \in [1000, 1000]$
`SKIP;`	$x \in [0, \infty] \cap [-\infty, 1000] = [0, 1000]$

Example of abstract interpretation with intervals

$\mathrm{x} \in [-\infty, \infty]$

```
IF x < 0 THEN
    x := 0;
ELSE IF x > 1000 THEN
    x := 1000;
ELSE
    SKIP;
ENDIF
```

$\mathrm{x} \in [0, 0]$

$\mathrm{x} \in [1000, 1000]$

$\mathrm{x} \in [0, \infty] \cap [-\infty, 1000] = [0, 1000]$

$\mathrm{x} \in [0, 0] \cup [1000, 1000] \cup [0, 1000] = [0, 1000]$

Example of abstract interpretation with intervals

```
x := 0;
WHILE x <= 1000 DO
    x := x + 1;
DONE
```

$x \in [0, 0]$

Example of abstract interpretation with intervals

```
x := 0;
```
$x \in [0, 0]$

```
WHILE x <= 1000 DO
```
$x \in [0, 0] \cap [-\infty, 1000] = [0, 0]$

```
    x := x + 1;
```
$x \in [1, 1]$

```
DONE
```

Example of abstract interpretation with intervals

```
x := 0;
```
$x \in [0, 0]$

```
WHILE x <= 1000 DO
```
$x \in ([0, 0] \cup [1, 1]) \cap [-\infty, 1000] = [0, 1]$

```
    x := x + 1;
```
$x \in [1, 2]$

```
DONE
```

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    x := x + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, 2]) \cap [-\infty, 1000] = [0, 2]$

$x \in [1, 3]$

Example of abstract interpretation with intervals

```
x := 0;                    
WHILE x <= 1000 DO
    x := x + 1;
DONE
```

$x \in [0, 0]$

$x \in [0, \infty]$

$x \in [1, \infty]$

Widening heuristic to accelerate convergence

Example of abstract interpretation with intervals

```
x := 0;
WHILE x <= 1000 DO
    x := x + 1;
DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, \infty]) \cap [-\infty, 1000] = [0, 1000]$

$x \in [1, 1001]$

Narrowing iteration to improve the result

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    x := x + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, 1001]) \cap [-\infty, 1000] = [0, 1000]$

$x \in [1, 1001]$

Fixpoint reached!

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    x := x + 1;

DONE
```

$\mathrm{x} \in [0, 0]$

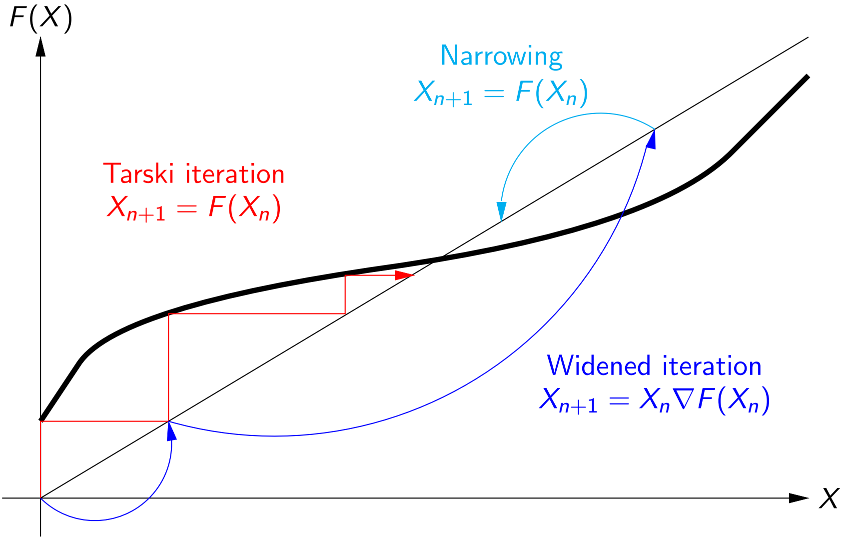
$\mathrm{x} \in ([0, 0] \cup [1, 1001]) \cap [-\infty, 1000] = [0, 1000]$

$\mathrm{x} \in [1, 1001]$

$\mathrm{x} \in [1001, \infty] \cap [1, 1001] = [1001, 1001]$

Fixpoint reached!

Fixpoint computations with widening and narrowing



Non-relational vs. 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:

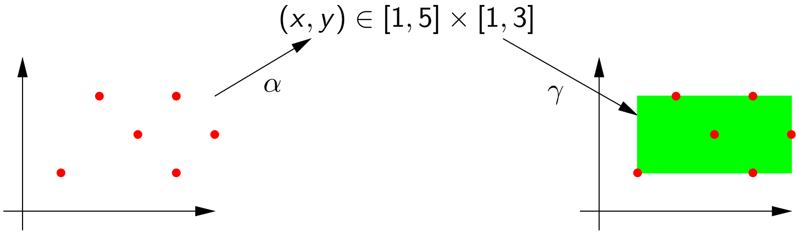
- a semi-lattice structure: $\bot$, $\top$, $\sqsubseteq$, $\sqcup$
- an abstract operation for assignment / binding.

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

Classic presentation: Galois connections

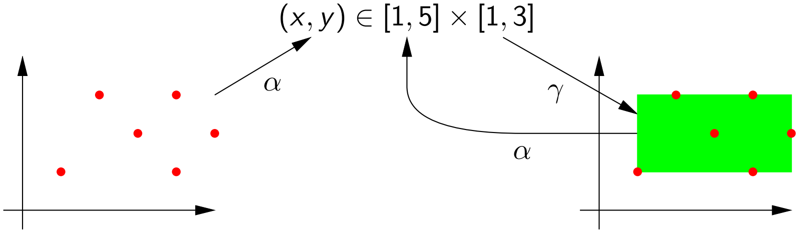
A semi-lattice $\mathcal{A}, \sqsubseteq$ of abstract states and two functions:

- 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:

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

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

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

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.

Problems with Galois connections

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 $[-\sqrt{2}, \sqrt{2}]$.

Problems with Galois connections

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

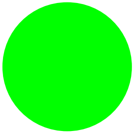
Example 1: intervals of rationals.

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

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

Example 2: polyhedra

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



Problems with Galois connections

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

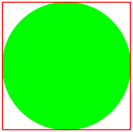
Example 1: intervals of rationals.

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

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

Example 2: polyhedra

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



Problems with Galois connections

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

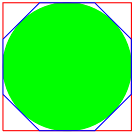
Example 1: intervals of rationals.

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

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

Example 2: polyhedra

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



Type-theoretic difficulties

In the context of a Coq/Agda verification:

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:

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

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

Plan B: γ-only presentation

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!

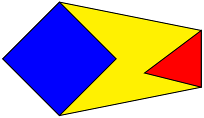
Having made optimality entirely optional, we can further simplify the analyzer and its soundness proof, while increasing its algorithmic efficiency:

- 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

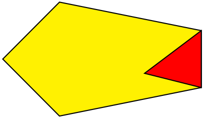
Some abstract operations can be implemented by unverified code if it is easy to validate the results a posteriori by a validator. Only the validator needs to be proved correct.

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 (Farkas certificate).

Plan

The Verasco project

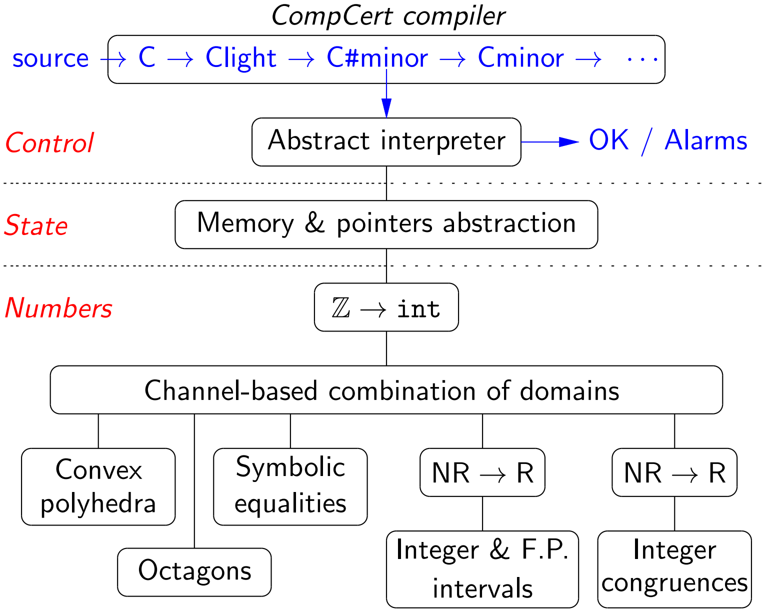
Inria Celtique, Gallium, Abstraction, Toccata + Verimag + Airbus

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

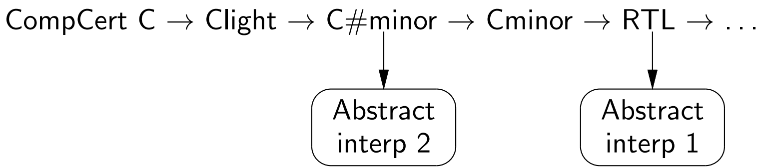
- Language analyzed: the CompCert subset of C.
- Nontrivial abstract domains, including relational domains.
- Modular architecture inspired from Astrée's.
- Decent alarm reporting.

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)
   Unstructured control + all functions inlined
   → one global fixpoint (Bourdoncle).
2. Abstract interpreter for C#minor (Jourdan)
   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 $\mathbf{Z}$).
- Floating-point intervals (on top of the Flocq library).
- Integer congruences (over $\mathbf{Z}$).

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 (direct Coq implementation).

What is a generic interface for a numerical domain?

For a non-relational domain:

- A semilattice $(A, \sqsubseteq)$ of abstract values.
- A concretization relation $\gamma : A \to \wp(\mathbf{Z})$
- "Forward" abstract operators such as $+_\#$, satisfying

$$\frac{v_1 \in \gamma(a_1) \qquad v_2 \in \gamma(a_2)}{v_1 + v_2 \in \gamma(a_1 +_\# a_2)}$$

- "Backward" abstract operators (to refine abstractions based on the results of conditionals) such as $<_\#^{-1}$

$$\frac{v_1 \in \gamma(a_1) \qquad v_2 \in \gamma(a_2) \qquad v_1 < v_2 \qquad (a_1', a_2') = (a_1 <_\#^{-1} a_2)}{v_1 \in \gamma(a_1') \wedge v_2 \in \gamma(a_2')}$$

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` ...) over variables and constants.

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

- `get_itv` *expr* → *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+⊥
  ; 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)
  ; 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

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

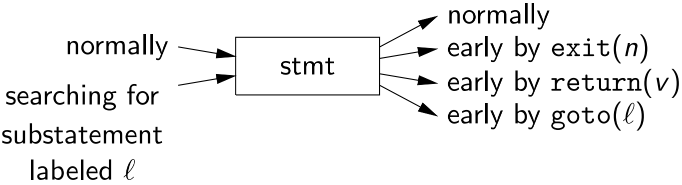
$$\begin{array}{rcl} F((s_1; s_2), A) & = & F(s_2, F(s_1, A)) \\ F((\texttt{IF}\ b\ \texttt{THEN}\ s_1\ \texttt{ELSE}\ s_2), A) & = & F(s_1, A \wedge b) \sqcup F(s_2, A \wedge \neg b) \\ F((\texttt{WHILE}\ b\ \texttt{DO}\ s\ \texttt{DONE}), A) & = & \texttt{pfp}\ (\lambda X.\ A \sqcup F(X \wedge b, s))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

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:

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

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

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:

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

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

Representative rules:

$$\begin{array}{rcl}
((s_1; s_2), k, m) & \to & (s_1, \mathtt{Kseq}\ s_2\ k, m) \\
(\mathtt{block}\ s, k, m) & \to & (s, \mathtt{Kblock}\ k, m) \\
(\mathtt{skip}, \mathtt{Kseq}\ s\ k, m) & \to & (s, k, m) \\
(\mathtt{exit}\ 0, \mathtt{Kblock}\ k, m) & \to & (\mathtt{skip}, k, m) \\
(\mathtt{exit}\ (n+1), \mathtt{Kblock}\ k, m) & \to & (\mathtt{exit}\ n, k, m)
\end{array}$$

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 $F(s, A_i, A_l) = (A_o, A_r, A_e, A_g)$ (and not `alarm`),
  then the weak Hoare 7-tuple

  $$\{\gamma(A_i), \gamma(A_l\}\ s\ \{\gamma(A_o), \gamma(A_r), \gamma(A_e), \gamma(A_g)\}$$

  is derivable.

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

Small-step soundness of a Hoare logic

(Andrew Appel and Sandrine Blazy, 2007)

Going back to IMP and standard Hoare triples $\{P\}\ s\ \{Q\}$ for simplicity:

Definition

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.

Definition

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

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:

$$\frac{\{P_1\}\ s\ \{Q\} \quad \{P_2\}\ s\ \{Q\}}{\{P_1 \vee P_2\}\ s\ \{Q\}} \qquad \frac{\{P\}\ s\ \{Q_1\} \quad \{P\}\ s\ \{Q_2\}}{\{P\}\ s\ \{Q_1 \wedge Q_2\}}$$

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

Plan

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.

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.