Mechanizing abstract interpretation

Xavier Leroy (Collège de France)
Workshop on the Next 40 years of Abstract Interpretation, 2024-01-20
Why mechanize abstract interpretation?

Because static analyzers deserve formal verification!

- They participate in the production of critical software.
- Abstract interpretation as the framework that guides the construction of static analyzers and their verification.
Why mechanize abstract interpretation?

Because static analyzers deserve formal verification!

- They participate in the production of critical software.
- Abstract interpretation as the framework that guides the construction of static analyzers and their verification.

Because abstract interpretation is a beautiful theory!

- A general trend towards mechanizing nice mathematics.
- Might learn something new about A.I.
- A stress test for many proof assistants.
## Some previous attempts (using Coq or Agda)

<table>
<thead>
<tr>
<th>Name</th>
<th>Theoretical ambition</th>
<th>Practical ambition</th>
<th>Work force</th>
</tr>
</thead>
<tbody>
<tr>
<td>David Monniaux</td>
<td>Abstract domains, Galois connections</td>
<td>None</td>
<td>Master’s internship</td>
</tr>
<tr>
<td>(1998)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>David Pichardie</td>
<td>$\gamma$-only soundness + termination</td>
<td>Interval analysis for JavaCard</td>
<td>PhD thesis</td>
</tr>
<tr>
<td>Verasco project</td>
<td>$\gamma$-only soundness</td>
<td>Mini-Astrée</td>
<td>ANR project (3 PhDs, 5 labs)</td>
</tr>
<tr>
<td>(2012–2015)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>David Darais</td>
<td>Constructive Galois connections</td>
<td>Gradual typing</td>
<td>PhD thesis</td>
</tr>
</tbody>
</table>
Project Verasco:
Verification of a static analyzer for C based on abstract interpretation
The Verasco project


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.
• Coq proofs of soundness against the CompCert semantics.

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

CompCert compiler

source $\rightarrow$ C $\rightarrow$ Clight $\rightarrow$ C#minor $\rightarrow$ Cminor $\rightarrow$ $\cdots$

Control

Abstract interpreter $\rightarrow$ OK / Alarms

State

Memory & pointers abstraction

Numbers

$\mathbb{Z} \rightarrow$ int

Channel-based combination of domains

Convex polyhedra
Symbolic equalities
Integer & F.P. intervals
Integer congruences

NR $\rightarrow$ R
NR $\rightarrow$ R
What is a generic interface for a numerical domain?

For a non-relational domain: the “gamma-only” presentation used by Pichardie, Bertot, Nipkow, …

• A semilattice \((A, \sqsubseteq)\) of abstract values.

• A concretization relation \(\gamma : A \rightarrow \wp(\mathbb{Z})\)

• “Forward” abstract operators such as \(+\#,\) satisfying

\[
v_1 \in \gamma(a_1) \land v_2 \in \gamma(a_2) \Rightarrow v_1 + v_2 \in \gamma(a_1 + \# a_2)
\]

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

If \((a_1', a_2') = (a_1 <_{\#}^{-1} a_2),\)

\[
v_1 \in \gamma(a_1) \land v_2 \in \gamma(a_2) \land v_1 < v_2 \Rightarrow v_1 \in \gamma(a_1') \land v_2 \in \gamma(a_2')
\]
What is a generic interface for a numerical domain?

For a relational domain, the main abstract operations are:

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

\( var \) are program variables or abstract memory locations.

\( expr \) are simple expressions (\(+\) − × div mod ... ) over variables and constants.

To report alarms, we also need to query the domain, e.g. “is \( x < y \)” or “is \( x \mod 4 = 0 \)”. The basic queries are

- get_itv \( expr \) what is a variation interval for \( expr \)?
- nonblock \( expr \) is \( expr \) always well-defined?
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

; $\gamma : t \rightarrow \wp (\text{var->num_val})$
; gamma_monotone: forall x y,
    leb x y = true -> $\gamma x \subseteq \gamma y$;
; gamma_top: forall x, x $\in \gamma$ top;
; join_sound: forall x y,
    $\gamma x \cup \gamma y \subseteq \gamma$ (join x y)
; forget_correct: forall x $\rho$ n ab,
    $\rho \in \gamma$ ab -> (upd $\rho$ x n) $\in \gamma$ (forget x ab)
; assign_correct: forall x e $\rho$ n ab,
    $\rho \in \gamma$ ab -> n $\in$ eval_nexpr $\rho$ e ->
    (upd $\rho$ x n) $\in \gamma$ (assign x e ab)
; assume_correct: forall e $\rho$ ab b,
    $\rho \in \gamma$ ab -> of_bool b $\in$ eval_nexpr $\rho$ e ->
    $\rho \in \gamma$ (assume e b ab)
; nonblock_correct: forall e $\rho$ ab,
    $\rho \in \gamma$ ab -> nonblock e ab = true -> block_nexpr $\rho$ e -> False
; get_itv_correct: forall e $\rho$ ab,
    $\rho \in \gamma$ ab -> (eval_nexpr $\rho$ e) $\subseteq \gamma$ (get_itv e ab)
An abstract interpreter for the C#minor language

\[ \text{Interp}_\#(\text{stmt}) (A_i, A_l) = (A_o, A_r, A_e, A_g) + \text{alarm} \]

Structural abstract interpretation, reflecting the different ways a C#minor statement can be entered and can terminate:

- \( A_i \) normally searching for label \( \ell \)
- \( A_l \) normally searching for label \( \ell \)
- \( A_o \) normally
- \( A_e \) early by \( \text{exit}(n) \)
- \( A_r \) early by \( \text{return}(v) \)
- \( A_g \) early by \( \text{goto}(\ell) \)

Local fixed-point iteration for loops + global iteration for \( \text{goto} \).

Functions are expanded at point of call.
Proving the soundness of the abstract interpreter

A two-step approach inspired by Y. Bertot (2005):

1. Soundness of the abstract interpreter w.r.t. a simple Hoare logic for C#minor:

   if $\text{Interp}_\#(\text{stmt}, A_i, A_l) = (A_o, A_r, A_e, A_g) \neq \text{alarm}$,

   then the weak Hoare 7-tuple

   $$\{ \gamma(A_i), \gamma(A_l) \} \; \text{stmt} \; \{ \gamma(A_o), \gamma(A_r), \gamma(A_e), \gamma(A_g) \}$$

   is derivable.

2. Soundness of the Hoare logic w.r.t. the small-step operational semantics of C#minor.
Verasco in practice

It works!

- Soundness 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).
- The memory abstraction is inefficient.
- Analysis is slow (up to one minute for 100 LOC).
Soundness first! …but many shortcuts

Compared with *The Calculational Design of a Generic Abstract Interpreter* and other holy texts:

- No Galois connections; gamma-only domains.
- No optimality guarantees: lubs are just upper bounds; abstract operators can over-approximate.
- No guarantees on the termination of fixed-point iteration (jump to $\top$ after a huge number of iterations).
- No proper collecting semantics.
- No calculations; implement then verify.

A good match for Coq’s constructive logic…

but too many $\top$ and some disappointment!
Fixed-point iterations in constructive logic
Let \((A, \sqsubseteq)\) be a partially-ordered set, with a smallest element \(\bot\), satisfying the ascending chain condition:

Every increasing sequence \(x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_i \sqsubseteq x_{i+1} \sqsubseteq \cdots\)
reaches a limit: there exists \(k\) s.t. \(x_k = x_{k+1} = \cdots = x_{k+i}\).

**Theorem (Knaster-Tarski)**

*Every increasing function \(F : A \to A\) has a smallest fixed point, which is the limit of the sequence \(\bot, F(\bot), \ldots, F^n(\bot), \ldots\).*
Accessibility and the chain condition

An alternate statement of the chain condition:

There are no infinite strictly increasing sequences
\[ x_0 \sqsubset x_1 \sqsubset \cdots \sqsubset x_i \sqsubset x_{i+1} \sqsubset \cdots \]

A constructive reformulation in terms of well-founded relations:

Every \( x : \text{A} \) is accessible, written \( \text{Acc}(x) \), meaning that all strictly ascending sequences starting in \( x \) are finite.

with an inductive characterization of accessibility:

\[
\forall y, \ x \sqsubset y \Rightarrow \text{Acc}(y) \\
\text{Acc}(x)
\]
Inductive predicate = no infinite branch in the derivation tree

Example: \( \mathbb{N} \cup \{\infty\} \) with the usual \(<\) ordering.
A computable function for Tarski iteration

Well-founded orders support defining recursive functions that always terminate because they are structurally recursive over a proof of accessibility $\text{Acc}(x)$ of their recursive argument $x$.

Program Fixpoint iter $\left(x: A\right)$ (PRE: le $x$ (F $x$)) (ACC: Acc $x$) 
{struct ACC}: A := 
    let $x'$ := F $x$ in 
    if eq_dec $x$ $x'$ then $x$ else iter $x'$ _ _ .

Program Definition lfp := iter bot _ _ .

(The _ are proof terms that are found semi-automatically by Coq.)

Easy proof (by induction on Acc $x$) that $F \ lfp = \ lfp$. 
“Extraction” (automatic code generation) produces the following OCaml code:

```ocaml
let rec iter eq_dec f x =  
  let x' = f x in if eq_dec x x' then x else iter eq_dec f x'

let lfp bot eq_dec f =  
  iter eq_dec f bot
```

Proof terms have disappeared, since they don’t contribute to the computation.
Post fixed-point iteration with widening

A widening operator \( \nabla \), such that \( x \subseteq x \nabla y \) and \( y \subseteq x \nabla y \) for all \( x, y \), and moreover

for every increasing sequence \( x_0 \subseteq \cdots \subseteq x_i \subseteq x_{i+1} \subseteq \cdots \),

the sequence defined by \( y_0 = x_0 \) and \( y_{i+1} = y_i \nabla x_{i+1} \)

reaches a limit: there exists \( k \) s.t. \( x_k = x_{k+1} = \cdots \)

**Theorem (Cousot and Cousot, 1992)**

*If \( F : A \rightarrow A \) is increasing and \( a \subseteq F(a) \), the sequence*

\[
x_0 = a \quad x_{i+1} = x_i \nabla F(x_i)
\]

*reaches a limit \( b \), and \( b \subseteq F(b) \).*
Post fixed-point iteration with widening

Tarski iteration
\[ x_{n+1} = F(x_n) \]

Widened iteration
\[ x_{n+1} = x_n \nabla F(x_n) \]
Widened accessibility

A constructive reformulation of the widened ascending chains condition:

\[ \text{WAcc}(x, x) \text{ holds for any } x : A \]

where WAcc is the inductive predicate defined by

\[ \forall x', x \sqsubseteq x' \land y \triangledown x' \neq y \Rightarrow \text{WAcc}(x', y \triangledown x') \]

\[ \text{WAcc}(x, y) \]

(Read: for any ascending chain starting in \(x\), any widened chain starting in \(y\) reaches a limit after a finite number of steps.)
A computable function for post-fixed point computation

Program Fixpoint witer (x y: A) (LE: le x (F y)) (ACC: WAcc x y) {struct ACC}: A :=
    let y’ := F y in
    let y’’ := widen y y’ in
    if le_dec y’ y then y else witer y’ y’’ _ _ .

Program Definition pfp: A := witer bot bot _ _ .

Easy proof (by induction on WAcc x y) that F pfp ⊑ pfp.
The constructive reformulation of the ascending chain conditions, using inductive predicates, supports the direct definition of iteration functions by structural recursion, which terminates by construction.

Yet, these constructive formulations are just as easy to use as the usual formulations!
Towards a mechanization of Galois connections and the calculational style
The adjunction property:

$$\forall a, s, \quad \alpha(s) \sqsubseteq a \iff s \subseteq \gamma(a)$$
Calculating sound and optimal abstract operations

\[ F_\# \overset{\text{def}}{=} \alpha \circ \overline{F} \circ \gamma \]

This equation defines an abstraction \( F_\# : A \to A \) of \( F : C \to C \) that

- is sound:
  \[ c \in \gamma(a) \Rightarrow F(c) \in \gamma(F_\#(a)) \]

- is optimal:
  \[ \text{if } \forall c \in \gamma(a), F(c) \in \gamma(a') \text{ then } F_\#(a) \sqsubseteq a' \]

- can be calculated systematically.
Calculating sound and optimal abstract operations

\[
\text{succ}\#(a) = \alpha \{ x + 1 \mid x \in \gamma(a) \}
\]

\[
= \alpha \{ \cdots \mid \cdots \}
\]

\[
= \cdots
\]

\[
= \cdots
\]

\[
= \begin{cases} 
\bot & \text{if } a = \bot \\
\top & \text{if } a = \top \\
\text{ODD} & \text{if } a = \text{EVEN} \\
\text{EVEN} & \text{if } a = \text{ODD}
\end{cases}
\]

\[\iff by \text{ definition of } \text{succ}\# \]

\[\iff \text{for good reasons} \]

\[\iff \text{case distinction} \]

\[\iff \text{more good reasons} \]
Problems with $\alpha$

**Sometimes not well defined**

Ex: $\{x \mid x^2 \leq 2\}$ has no best abstraction as a rational interval.

**Generally not computable** (as soon as $C$ is infinite)

Ex with intervals: $\alpha(s) \overset{\text{def}}{=} [\inf(s), \sup(s)]$.

For infinite sets $s$ of integers, $\inf(s)$ and $\sup(s)$ are not computable.

A computable $\alpha$ can only look at a finite number of elements of $s$. 
Proof assistants and non-computable functions

Proof assistants vary in their support for classical logic:

- Agda: purely constructive
- Coq, Lean: constructive + classical axioms (excluded middle)
- HOL4, Isabelle/HOL: fully classical.

But they are all logics of computable functions (Milner’s LCF).

This supports computation within the prover and extraction to efficient functional code.

Axioms can be added to define non-computable functions (similar to the axiom of choice, $\forall x, \exists y, P(x, y) \Rightarrow \exists f, \forall x, P(x, f(x))$) but this can break executability and extraction.
Plan B: no calculations; gamma-only optimality

Pichardie’s approach:

• No $\alpha$; use only $\gamma : A \rightarrow \wp(C)$ concretization predicates.
• No calculations; define abstract operators $F\#$ a priori (or: calculate them on paper).
• Prove soundness: $c \in \gamma(a) \Rightarrow F(c) \in F\#(a)$
• Optionally, prove optimality:
  if $\forall c \in \gamma(a), F(c) \in a'$ then $F\#(a) \subseteq a'$.

It works! But some elegance is lost compared with the calculational approach.

(Plus: elementary proofs of optimality are a pain.)
Handling I/O and other effectful computations in pure functional languages via monads:

- a type $M(A)$ of effectful computations of a value of type $A$.
- $\text{ret} : A \rightarrow M(A)$ to inject values as trivial computations.
- $\text{bind} : M(A) \rightarrow (A \rightarrow M(B)) \rightarrow M(B)$ for sequencing.
- but no projection $M(A) \rightarrow A$.

Darais and Van Horn (2016), working in Agda, propose to use a monad to separate

- computable functions such as $F_\# \ (w/ \ constructive \ logic)$
- non-computable functions such as $\alpha \ (w/ \ classical \ logic)$. 
Calculations in a monadic style

\[
\text{ret}(\text{succ\#}(a)) = \alpha \{ x + 1 \mid x \in \gamma(a) \} \\
\text{\hspace{1cm}} = \ldots \\
\text{\hspace{1cm}} = \ldots \\
\text{\hspace{1cm}} = \text{ret}(\text{match } a \text{ with} \\
\text{\hspace{2cm}} | \ \text{Bot} \Rightarrow \text{Bot} \\
\text{\hspace{2cm}} | \ \text{Top} \Rightarrow \text{Top} \\
\text{\hspace{2cm}} | \ \text{Even} \Rightarrow \text{Odd} \\
\text{\hspace{2cm}} | \ \text{Odd} \Rightarrow \text{Even})
\]

\[
\text{by definition of succ\#} \\
\text{using monad laws} \\
\text{for good reasons}
\]

It follows that \(\text{succ\#}(a) = \text{match } a \text{ with } \ldots.\)
Start with two constructive functions:

\[
\text{extraction } \nu : C \to A \quad \text{interpretation } \mu : A \to \wp(C)
\]

(Extraction must be computable! E.g. to abstract constants...)

Derive a Galois connection \((\wp(C), \subseteq) \leftrightarrow \left( \wp(A), \subseteq \right)\) by taking

\[
\begin{align*}
\alpha(c^*) &= \{\nu(c) \mid c \in c^*\} \\
\gamma(a^*) &= \bigcup\{\mu(a) \mid a \in a^*\}
\end{align*}
\]

As the monad, use classical, infinite, non-computable sets

\[
\text{mon}(X) \overset{\text{def}}{=} X \to \text{Prop}
\]

as opposed to computable finite sets of \(A\), which are the arguments and results of abstract operators.
On paper

Case \( ae = x \):
\[
\{ \eta^z(i) \mid \rho \in \mu^r(\rho^z) \land \rho \vdash x \downarrow^a i \} \\
= \{ \eta^z(\rho(x)) \mid \rho \in \mu^r(\rho^z) \} \\
\subseteq \{ \rho^z(x) \} \\
= \{ A^z[x](\rho^z) \}
\]

-- Agda Calculation of Case \( ae = x \):
\[
\alpha[A] (\text{Var } x) \rho^\# = \text{[proof-mode]}
\]
\[
do \text{[pure} \cdot \eta^z \text{]} \cdot (A[ \text{Var } x ] \cdot (\mu^r \cdot \rho^\#))
\]
\[
\text{[focus-right [\cdot] of (pure} \cdot \eta^z \text{]} \begin{align*}
do (A[ \text{Var } x ] \cdot (\mu^r \cdot \rho^\#)) \\
& \cdot (A[\text{Var}]/\equiv) \\
& \cdot (\text{[pure} \cdot \text{lookup}[x] \cdot (\mu^r \cdot \rho^\#)) \\
& \cdot (\text{[lookup}/\mu^r/\equiv) \\
& \cdot (\text{[\mu^z} \cdot (\text{pure} \cdot \text{lookup}[x] \cdot \rho^\#)) \\
& \text{end}
\end{align*}
\]
\[
\text{[pure} \cdot \eta^z \text{]} \cdot (\mu^z \cdot (\text{pure} \cdot \text{lookup}[x] \cdot \rho^\#)) \\
\text{[reductive} [\eta \mu] \\
[\text{ret} \cdot (\text{lookup}[x] \cdot \rho^\#)) \\
[\text{pure} \cdot A^\#[\text{Num } n] \cdot \rho^\#] \square
\]
Instead of monads, some languages use type and effect systems to control effects (e.g. `throws` clauses in Java).

Likewise(?) Coq and Lean distinguish between

- `Prop`, the universe of logical propositions and their proofs;
- `Type`, the universe of data types and their computations.

Various combinations are possible, but a computation in `Type` cannot depend on a proof in `Prop`. 
From a proof of $\text{terminates}(tm) \lor \neg \text{terminates}(tm)$, we can reason by case whether $tm$ terminates or not:

$$\text{terminates}(tm) \lor \neg \text{terminates}(tm) \rightarrow P : \text{Prop}$$

but we cannot define a Boolean-valued “it terminates” function:

$$\text{terminates}(tm) \lor \neg \text{terminates}(tm) \rightarrow \text{bool} : \text{Type}$$

(All functions with this type are constant.)

This makes it possible to add excluded middle as an axiom without endangering extraction (which erases all of $\text{Prop}$).
1) Define $\alpha$ as a relation between $\wp(C)$ and $A$:

```coq
Inductive is_alpha (cc: Z → Prop) : A → Prop :=
  | is_alpha_bot:
    (∀ c, (∼cc c)) →
    is_alpha cc Bot
  | is_alpha_even: ∀ c1,
    cc c1 →
    (∀ c, cc c → c mod 2 = 0) →
    is_alpha cc Even
  | is_alpha_odd: ∀ c1,
    cc c1 →
    (∀ c, cc c → c mod 2 = 1) →
    is_alpha cc Odd
  | is_alpha_top: ∀ c0 c1,
    cc c0 → cc c1 →
    c0 mod 2 = 0 → c1 mod 2 = 1 →
    is_alpha cc Top.
```
1) Define \( \alpha \) as a relation between \( \wp(C) \) and \( A \):

2) Show that it satisfies the adjunction property:

**Lemma adjunction:** \( \forall cc a, \text{is}_\alpha cc a \rightarrow \forall a', \text{le} a a' \leftrightarrow (\forall c, cc c \rightarrow \text{gamma} c a') \).
Plan U in Coq

1) Define $\alpha$ as a relation between $\wp(C)$ and $A$:

2) Show that it satisfies the adjunction property:

3) Do not use a choice / description axiom to get an $\alpha$ function from the $\text{is\_alpha}$ predicate. This leads nowhere and endangers extraction.
1) Define $\alpha$ as a relation between $\wp(C)$ and $A$:

2) Show that it satisfies the adjunction property:

3) Do not use a choice / description axiom to get an alpha function from the $\text{is\_alpha}$ predicate. This leads nowhere and endangers extraction.

4) Describe abstract operators with the help of $\text{is\_alpha}$.

Lemma `succsharp_exist`:

$$\forall a, \{a' | \text{is\_alpha} (\text{fun } n \Rightarrow \exists m, \text{gamma } a \ m \land n = \text{succ } m) \ a'\}$$
1) Define $\alpha$ as a relation between $\wp(C)$ and $A$:

2) Show that it satisfies the adjunction property:

3) Do not use a choice / description axiom to get an alpha function from the is_alpha predicate. This leads nowhere and endangers extraction.

4) Describe abstract operators with the help of is_alpha.

5) Use interactive proof mode to construct a’.

6) Project out the abstract operator and its soundness and optimality proof.

\begin{verbatim}
Definition succsharp (a: A) := proj1_sig (succsharp_exist a).
Lemma succsharp_sound_complete: \forall a,
  is_alpha (\fun x \Rightarrow \exists y, gamma y a \land x = y + 1) (succsharp a).
\end{verbatim}
A glimpse of the “calculation”

Proof.

intros. destruct a; unfold gamma.

- (* Bot *)
econstructor; eapply is_alpha_bot. firstorder.

- (* Top *)
econstructor; eapply is_alpha_top with (c0 := 0) (c1 := 1).
  ∃ (-1); auto. ∃ 0; auto. reflexivity. reflexivity.

- (* Even *)
  ∃ Odd. apply is_alpha_odd with (c1 := 1).
  ∃ 0; auto.
  intros c’ (y & P & Q). subst c’. unfold eta.
  rewrite ← Z.add_mod_idemp_l, P by lia. reflexivity.

- (* Odd *)
  ∃ Even. apply is_alpha_even with (c1 := 0).
  ∃ (-1); auto.
  intros c’ (y & P & Q). subst c’. unfold eta.
  rewrite ← Z.add_mod_idemp_l, P by lia. reflexivity.

Defined.
In the end, it is possible to mechanize some forms of Galois connections (Darais & Van Horn’s $\eta/\mu$ or my is $-\alpha/\gamma$) and use them to calculate abstract operations.

The mechanized approaches don’t have (yet?) the crispness nor the immediacy of the classic pencil-and-paper presentation of AI.
What’s next?
On the formal verification of static analyzers

Naive, $\gamma$-only abstract interpretation provides a robust approach.

Textbook material (e.g. Concrete Semantics, Nikpow & Klein).

Scales to nontrivial relational analysis (see the Verasco project).

Hard to scale much further: complex algorithms, difficult proofs.

Little interest for verified static analysis (outside of compilers).
On the mechanization of abstract interpretation

Still a challenge for contemporary proof assistants:

- Abstract interpretation is deeply grounded in set theory, yet produces algorithms in the end.
- Poor support for the calculational style, not just for A.I. but also for Bird-style program derivations, or even for undergraduate mathematics.