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)

	Theoretical ambition	Practical ambition	Work force
David Monniaux (1998)	Abstract domains, Galois connections	None	Master's internship
David Pichardie (2002–2005, adv. Cachera & Jensen)	γ-only soundness + termination	Interval analysis for JavaCard	PhD thesis
Verasco project (2012–2015)	γ-only soundness	Mini-Astrée	ANR project (3 PhDs, 5 labs)
David Darais (2013–2017, adv. David Van Horn)	Constructive Galois connections	Gradual typing	PhD thesis

Project Verasco:
Verification of a static analyzer for C based on abstract interpretation

The Verasco project

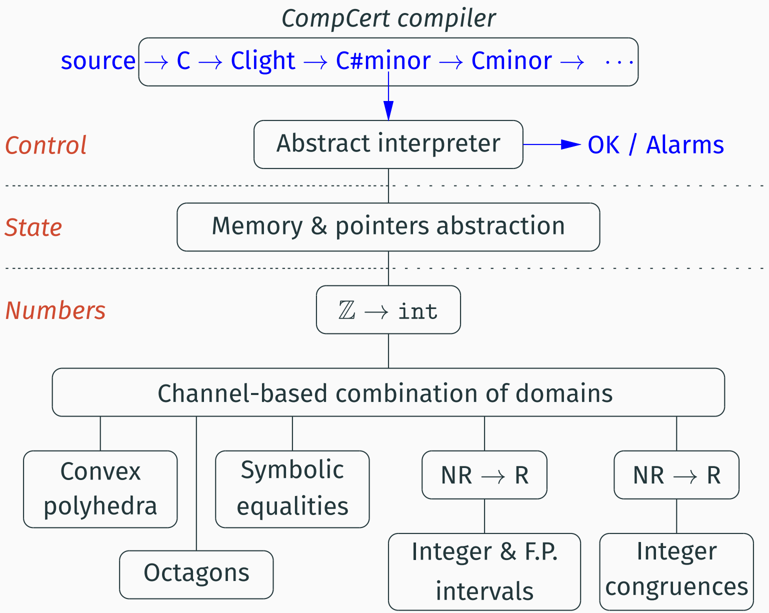
(PhDs: J.-H. Jourdan, V. Laporte, A. Fouilhé; PIs: D. Pichardie & X. Leroy; participants: S. Blazy, J. Feret, A. Miné, X. Rival, D. Monniaux, M. Périn.)

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



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 $\gamma : A \to \wp(\mathbf{Z})$
- “Forward” abstract operators such as $+_\#$, satisfying

$$v_1 \in \gamma(a_1) \wedge v_2 \in \gamma(a_2) \Rightarrow v_1 + v_2 \in \gamma(a_1 +_\# a_2)$$

- “Backward” abstract operators (to refine abstractions based on the results of conditionals) such as $<_\#^{-1}$
  If $(a_1', a_2') = (a_1 <_\#^{-1} a_2)$,

$$v_1 \in \gamma(a_1) \wedge v_2 \in \gamma(a_2) \wedge v_1 < v_2 \Rightarrow v_1 \in \gamma(a_1') \wedge v_2 \in \gamma(a_2')$$

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` $\ldots$) over variables and constants.

To report alarms, we also need to query the domain, e.g. “is $\mathtt{x} < \mathtt{y}$?” or “is $\mathtt{x}\ \mathtt{mod}\ 4 = 0$?”. 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+⊤+⊥
```

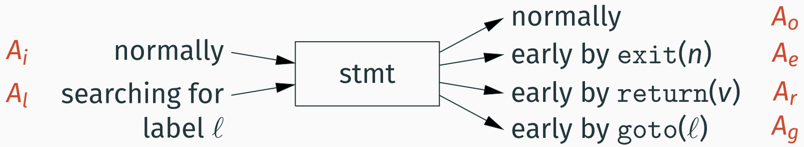
… and their specifications

```
  ; 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)
}
```

An abstract interpreter for the C#minor language

$$Interp_{\#}(stmt)\ (A_i, A_l) = (A_o, A_r, A_e, A_g) + \texttt{alarm}$$

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



Local fixed-point iteration for loops + global iteration for `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 $\mathit{Interp}_{\#}(\mathit{stmt}, A_i, A_l) = (A_o, A_r, A_e, A_g) \neq \texttt{alarm}$,
   then the weak Hoare 7-tuple

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

   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

Tarski's fixed-point iteration

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:

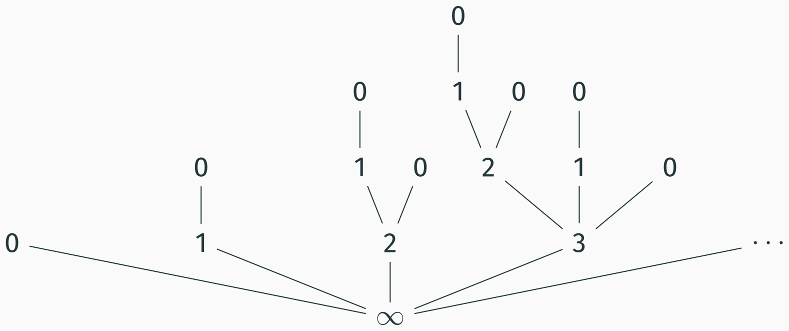
> Every $x : A$ is accessible, written $Acc(x)$, meaning that
> all strictly ascending sequences starting in x are finite.

with an inductive characterization of accessibility:

$$\frac{\forall y,\ x \sqsubset y \Rightarrow Acc(y)}{Acc(x)}$$

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 $Acc(x)$ of their recursive argument x.

```
Program Fixpoint iter (x: A) (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`.

Executable OCaml code for Tarski iteration

“Extraction” (automatic code generation) produces the following OCaml code:

```
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 ∇, such that $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$ for all x, y, and moreover

> for every increasing sequence $x_0 \sqsubseteq \cdots \sqsubseteq x_i \sqsubseteq x_{i+1} \sqsubseteq \cdots$,
> 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. $x_k = x_{k+1} = \cdots$

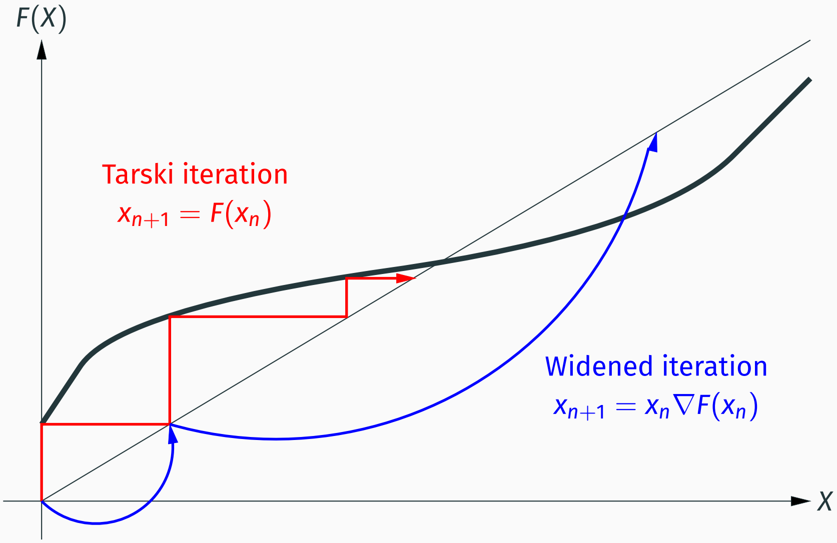
Theorem (Cousot and Cousot, 1992)

If $F : A \to A$ is increasing and $a \sqsubseteq F(a)$, the sequence

$$x_0 = a \qquad x_{i+1} = x_i \nabla F(x_i)$$

reaches a limit b, and $b \sqsubseteq F(b)$.

Post fixed-point iteration with widening



Widened accessibility

A constructive reformulation of the widened ascending chains condition:

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

where *WAcc* is the inductive predicate defined by

$$\frac{\forall x', x \sqsubseteq x' \wedge y \nabla x' \neq y \Rightarrow WAcc(x', y \nabla x')}{WAcc(x, y)}$$

(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` $\sqsubseteq$ `pfp`.

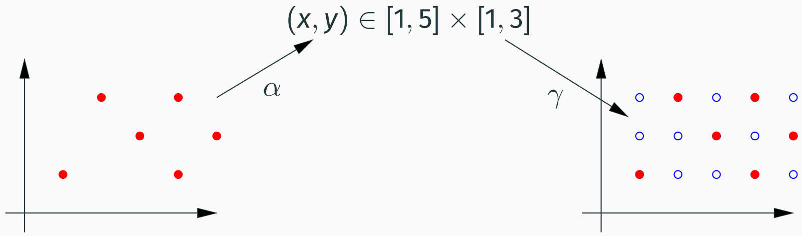
A success story for constructive mechanization

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

Classic Galois connection $(\wp(C), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$



The adjunction property:

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

Calculating sound and optimal abstract operations

$$F_{\#} \overset{def}{=} \alpha \circ \overline{F} \circ \gamma$$

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

$$\begin{aligned}
\mathtt{succ}_{\#}(a) &= \alpha\,\{x+1 \mid x \in \gamma(a)\} && \wr\ \text{by definition of } \mathtt{succ}_{\#}\ \int \\
&= \alpha\,\{\cdots \mid \cdots\} && \wr\ \text{for good reasons}\ \int \\
&= \cdots && \wr\ \text{case distinction}\ \int \\
&= \cdots && \wr\ \text{more good reasons}\ \int \\
&= \begin{cases} \bot & \text{if } a = \bot \\ \top & \text{if } a = \top \\ \mathtt{ODD} & \text{if } a = \mathtt{EVEN} \\ \mathtt{EVEN} & \text{if } a = \mathtt{ODD} \end{cases}
\end{aligned}$$

Problems with α

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

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

A computable α 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 α; use only $\gamma : A \to \wp(C)$ 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 $F_\#(a) \sqsubseteq a'$.

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

(Plus: elementary proofs of optimality are a pain.)

Plan M like Monads (Darais and Van Horn)

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.
- `ret` : $A \to M(A)$ to inject values as trivial computations.
- `bind` : $M(A) \to (A \to M(B)) \to M(B)$ for sequencing.
- but no projection $M(A) \to A$.

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 α (w/ classical logic).

Calculations in a monadic style

$$\begin{aligned}
\texttt{ret}(\texttt{succ}_{\#}(a)) &= \alpha\ \{x+1 \mid x \in \gamma(a)\} && \wr\ \text{by definition of } \texttt{succ}_{\#}\ \int \\
&= \ldots && \wr\ \text{using monad laws}\ \int \\
&= \cdots && \wr\ \text{for good reasons}\ \int \\
&= \texttt{ret(match } a \texttt{ with}
\end{aligned}$$

```
                | Bot => Bot
                | Top => Top
                | Even => Odd
                | Odd => Even)
```

It follows that $\texttt{succ}_{\#}(a) = \texttt{match}\ a\ \texttt{with}\ \ldots$.

Constructive Galois connections (Darais and Van Horn, ICFP 2016, JFP 2019)

Start with two constructive functions:

$$\text{extraction } \nu : C \to A \qquad \text{interpretation } \mu : A \to \wp(C)$$

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

Derive a Galois connection $(\wp(C), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\wp(A), \subseteq)$ by taking

$$\alpha(c^*) = \{\nu(c) \mid c \in c^*\} \qquad \gamma(a^*) = \bigcup\{\mu(a) \mid a \in a^*\}$$

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

$$\mathtt{mon}(X) \overset{def}{=} X \to \mathtt{Prop}$$

as opposed to computable finite sets of A, which are the arguments and results of abstract operators.

An example of calculation mechanized in Agda (Darais and Van Horn)

On paper

Case $ae = x$:

$$
\begin{array}{ll}
\{\eta^{z}(i) \mid \rho \in \mu^{r}(\rho^{\sharp}) \wedge \rho \vdash x \Downarrow^{a} i\} & \\
= \{\eta^{z}(\rho(x)) \mid \rho \in \mu^{r}(\rho^{\sharp})\} & \wr \text{ defn. of } \rho \vdash x \Downarrow^{a} i \int \\
= \{\eta^{z}(i) \mid i \in \mu^{z}(\rho^{\sharp}(x))\} & \wr \text{ defn. of } \mu^{r}(\rho^{\sharp}) \int \\
\subseteq \{\rho^{\sharp}(x)\} & \wr \text{ Eq. CGC-Red} \int \\
\triangleq \{\mathcal{A}^{\sharp}[x](\rho^{\sharp})\} & \wr \text{ defining } \mathcal{A}^{\sharp}[x] \int
\end{array}
$$

Mechanized

```
-- Agda Calculation of Case $ae = x$:
$\alpha[\mathcal{A}]$ (Var $x$) $\rho\sharp$  =  [proof-mode]
  do [[ (pure · $\eta^{z}$) ∗ · ($\mathcal{A}$[ Var $x$ ] ∗ · ($\mu^{r}$ · $\rho\sharp$)) ]]
   ▪ [focus-right [·] of (pure · $\eta^{z}$) ∗ ] begin
     do [[ $\mathcal{A}$[ Var $x$ ] ∗ · ($\mu^{r}$ · $\rho\sharp$) ]]
      ▪ $\wr$ $\mathcal{A}$[Var]/≡ $\int$
      ▪ [[ (pure · lookup[ $x$ ]) ∗ · ($\mu^{r}$ · $\rho\sharp$) ]]
      ▪ $\wr$ lookup/$\mu^{r}$/≡ $\int$
      ▪ [[ $\mu^{z}$ ∗ · (pure · lookup♯[ $x$ ] · $\rho\sharp$) ]]
     end
   ▪ [[ (pure · $\eta^{z}$) ∗ · ($\mu^{z}$ ∗ · (pure · lookup♯[ $x$ ] · $\rho\sharp$)) ]]
   ▪ $\wr$ reductive[$\eta\mu$] $\int$
   ▪ [[ ret · (lookup♯[ $x$ ] · $\rho\sharp$) ]]
   ▪ [[ pure · $\mathcal{A}\sharp$[ Num $n$ ] · $\rho\sharp$ ]]  □
```

Plan U like Universes: exploiting the Prop/Type separation

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`.

Prop vs Type

From a proof of $\mathit{terminates}(\mathit{tm}) \vee \neg\mathit{terminates}(\mathit{tm})$,
we can reason by case whether *tm* terminates or not:

$$\mathit{terminates}(\mathit{tm}) \vee \neg\mathit{terminates}(\mathit{tm}) \to P : \texttt{Prop}$$

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

$$\mathit{terminates}(\mathit{tm}) \vee \neg\mathit{terminates}(\mathit{tm}) \to \texttt{bool} : \texttt{Type}$$

(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 `Prop`).

Plan U in Coq

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

```
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.
```

Plan U in Coq

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

2) Show that it satisfies the adjunction property:

```
Lemma adjunction: ∀ cc a, is_alpha cc a →
  ∀ a', le a a' ↔ (∀  c, cc c → gamma c a').
```

Plan U in Coq

1) Define α 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.

Plan U in Coq

1) Define α 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`.

```
Lemma succsharp_exist:
  ∀ a, {a' | is_alpha (fun n ⇒ ∃ m, gamma a m ∧ n = succ m) a'}
```

Plan U in Coq

1) Define α 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.

```
Definition succsharp (a: A) := proj1_sig (succsharp_exist a).
Lemma succsharp_sound_complete: ∀ a,
  is_alpha (fun x ⇒ ∃ y, gamma y a ∧ x = y + 1) (succsharp a).
```

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.
```

A Pyrrhic victory for mechanization

In the end, it is possible to mechanize some forms of Galois connections (Darais & Van Horn's η/μ or my $\texttt{is}{-}\alpha/\gamma$) 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, γ-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.