Mechanized semantics, fifth lecture

Abstract art: static analysis by abstract interpretation

Xavier Leroy

2020-01-17

Collège de France, chair of software sciences

Static analysis (as seen in the third lecture)

Infer (without help from the programmeur) statically (without running the program) some properties that hold for all possible executions of the program.

Main uses:

- To improve performance by guiding optimizations.
- To improve safety by showing the absence / detecting the possible presence of programming errors.

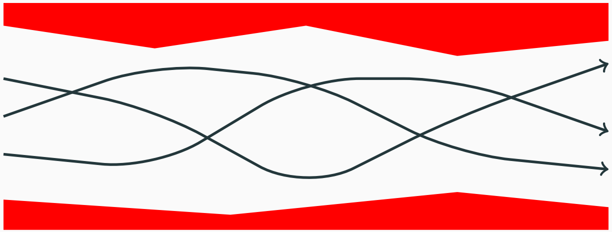
Using static analyses for verification purposes

Use properties inferred by static analysis to show the absence of run-time errors (such as division by zero, or out-of-bound array accesses). aux tableaux hors bornes).

$$\begin{aligned} b \in [n_1, n_2] \wedge 0 \notin [n_1, n_2] &\implies a/b \text{ causes no error} \\ \texttt{valid}(p[n_1 \ldots n_2]) \wedge i \in [n_1, n_2] &\implies \texttt{p[i]} \text{ causes no error} \end{aligned}$$

Raise an alarm where we cannot show the absence of errors.

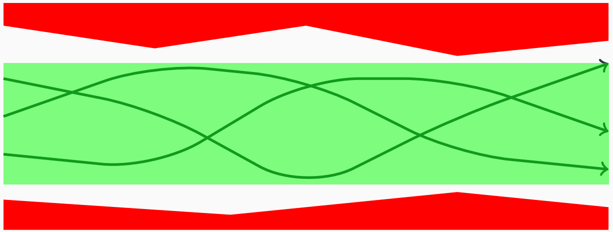
Static analysis and verification, graphically



Paths = possible executions of the program.
(For example, value of a variable *x* as a function of time *t*.)

Red zone = run-time errors, behaviors we want to exclude.

Static analysis and verification, graphically

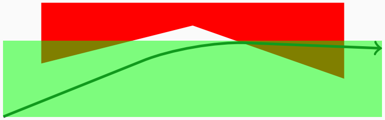


Paths = possible executions of the program.
(For example, value of a variable *x* as a function of time *t*.)

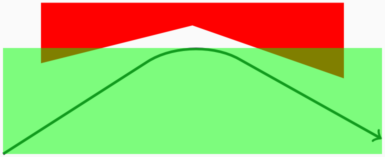
Red zone = run-time errors, behaviors we want to exclude.

Green zone = results of static analysis; over-approximation of the possible executions of the program.

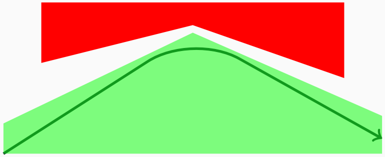
True alarms, false alarms



True alarm
(bad behavior)



False alarm
(imprecise analysis)



More precise analysis (e.g. polyhedra 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-bounds array accesses.
    - No dereferencing the null pointer.
    - No accesses after explicit deallocation (`free`).
    - Alignment constraints from the hardware.
- Integer arithmetic:
    - No division by zero.
    - No (signed) arithmetic overflows.
- Floating-point arithmetic:
    - No overflows (producing infinities)
    - No undefined operations (producing *Not-a-Number*)
    - No catastrophic cancellation (major loss of precision).

Some properties verifiable by static analysis

Validating assertions and invariants:

- Variation intervals for numerical outputs.
- Preconditions for library functions.
- Shape analysis for linked data structures.

Verifying security policies:

- Information flow analyses; “tainting”.

Establishing “non-functional” properties:

- Bounding memory usage.
- Bounding worst-case execution time (WCET).

Which static analyses for verification purposes?

Dataflow analyses (3rd lecture):

- Fully automatic, algorithmically efficient.
- Infer simple properties only, often insufficient to show the absence of run-time errors.

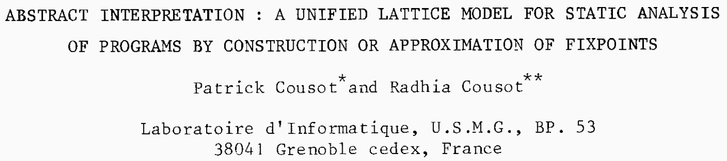
Deductive verification in Hoare logic / separation logic (4th lecture):

- Can verify arbitrarily-complex properties.
- Not automatic (loop invariants).

Is there anything in between?

Classic abstract interpretation

Abstract interpretation


ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

(POPL, 1977)

A very general formalism to describe and implement precise static analyses.

Intuitions for abstract interpretation

Analyzing a program = executing it with a non-standard semantics:

- Compute with the abstract domains of the desired properties (e.g. “$x \in [n_1, n_2]$” for interval analysis) instead of with concrete objects (numbers, values, stores).
- Guarantee that the abstract computation is always an over-approximation of the concrete computation.

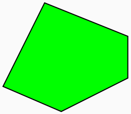
Examples of abstract domains

Properties of a single numerical variable:

$x \in [a, b]$ (intervals) $\qquad$ $x \bmod a = b$ (congruences)

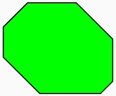
Relations between several numerical variables:

polyhedra



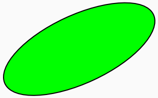
$\sum a_i x_i \leq c$

octagons



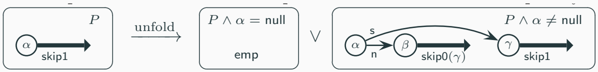
$\pm x_1 \pm x_2 \leq c$

ellipses



$$ax^2 + by^2 + cxy + dx + ey \leq f$$

Shape analysis for linked data structures:



Abstract interpretation using intervals

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

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

$\{\, z = 3 + 2 \times 1 = 5 \,\}$	$\{\, z^{\#} = [0, 9] +^{\#} 2 \times^{\#} [-1, 1] = [-2, 11] \,\}$

(We write “$\#$” for the abstractions of variables and operators.)

Abstract interpretation using intervals

$x^{\#} = [0, 10]$

```
if x < 0 then
    s := -1
else if x > 0 then
    s := 1
else
    s := 0
```

$s^{\#} = \emptyset$

$s^{\#} = [1, 1]$

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

$$s^{\#} = \emptyset \cup [1, 1] \cup [0, 0] = [0, 1]$$

In general, Boolean conditions cannot be resolved statically
$\rightarrow$ execution of all branches + union of the abstractions.

Some conditions are statically resolved
$\rightarrow$ we mark $\emptyset$ or $\perp$ the branch not taken.

Abstract interpretation with intervals

In general, loops are assumed to run an arbitrary number of times.

```
x := 0;
while condition do
    x := x + 1
done
```

$$x^\# = [0, +\infty]$$

Counted loops can often be analyzed more precisely.

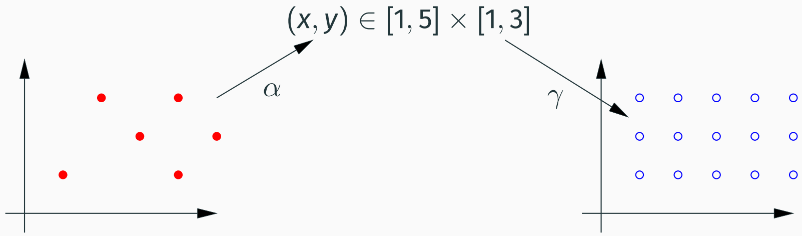
```
x := 0;
for i := 1 to 10 do
  if condition then x := x + 1
done
```

$$x^\# = [0, 10]$$

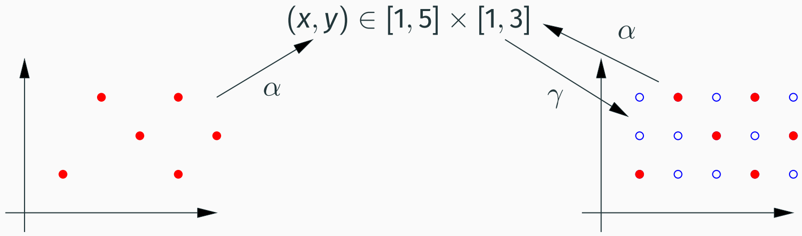
Formalizing abstract domains

A lattice $(A, \leq)$ of abstract values and two functions:

- α, the abstraction function:
  sets of concrete values $\rightarrow$ abstract value
- γ, the concretization function:
  abstract value $\rightarrow$ set of concrete values.



Properties of concretization and abstraction



$$\begin{array}{lll} & \alpha \text{ et } \gamma \text{ are increasing} & \\ \wedge & \forall X,\ \ X \subseteq \gamma(\alpha(X)) & \text{(soundness)} \\ \wedge & \forall a,\ \ \alpha(\gamma(a)) \leq a & \text{(optimality)} \end{array}$$

$(\mathcal{P}(C), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \leq)$ is an isotone Galois connection.

Calculating abstract operators

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

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

This abstract operator is:

- Sound: if $x \in \gamma(a)$, then $F(x) \in \gamma(F^{\#}(a))$.
- Optimally precise: $F(a)$ is the smallest abstraction a' such that $x \in \gamma(a) \Rightarrow F(x) \in \gamma(a')$.

Furthermore, an effective implementation of $F^{\#}$ can be calculated (derived by symbolic computation) from the definition.

Calculating $+^\#$ for intervals

We have $\alpha(X) = [\inf X, \sup X]$ and $\gamma([a, b]) = \{n \mid a \leq x \leq b\}$.
Therefore:

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

The “obvious” definition $[a_1, b_1] +^\# [a_2, b_2] = [a_1 + b_1, a_2 + b_2]$
is sound but not precise.

Abstract interpretation of commands

Consider an operational semantics $c/s \Downarrow s'$
and an abstraction for stores $(\mathcal{P}(store), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \leq)$.

We define the abstract interpretation $\mathtt{exec}^{\#}(c) : A \to A$
of a command c as a function from the abstract store "before" to the abstract store "after":

$$\mathtt{exec}^{\#}(c) \overset{def}{=} \lambda a.\, \alpha\{s' \mid s \in \gamma(a),\ c/s \Downarrow s'\}$$

(Alternate, orthodox approach: use a collecting semantics program point $\to$ set of concrete stores.)

Abstract interpretation of commands

$$\texttt{exec}^{\#}(c) \overset{\mathit{def}}{=} \lambda a.\, \alpha\{s' \mid s \in \gamma(a),\ c/s \Downarrow s'\}$$

We can, then, calculate equations for $\texttt{exec}^{\#}$, such as

$$\begin{aligned}
\texttt{exec}^{\#}(\texttt{skip}) &= \lambda a.\, a \\
\texttt{exec}^{\#}(c_1; c_2) &= \texttt{exec}^{\#}(c_2) \circ \texttt{exec}^{\#}(c_1) \\
\texttt{exec}^{\#}(\texttt{while } b \texttt{ do } c) &= \texttt{exec}^{\#}(\texttt{if } b \\
&\qquad\qquad\qquad \texttt{then c}; \texttt{while } b \texttt{ do } c \\
&\qquad\qquad\qquad \texttt{else skip})
\end{aligned}$$

Abstract interpretation and Hoare logic

$$\mathtt{exec}^{\#}(c) \overset{def}{=} \lambda a.\, \alpha\{s' \mid s \in \gamma(a),\ c/s \Downarrow s'\}$$

We can read an abstract store a as an assertion in Hoare logic: the assertion "the current store belongs to $\gamma(a)$".

With this interpretation, $\mathtt{exec}^{\#}$ satisfies the Hoare triple

$$\{\, a \,\}\; c \;\{\, \mathtt{exec}^{\#}(a) \,\}$$

Moreover, $\mathtt{exec}^{\#}(a)$ is the strongest post-condition of c and a that can be expressed in the abstract domain of stores.

Constructive abstract interpretation

Galois connections are problematic in type theory

Galois connections $(\mathcal{P}(C), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \leq)$ cannot, in general, be defined in type theory.

Concretization γ is easily modeled as

$$\gamma : A \to (C \to \texttt{Prop}) \qquad\qquad \text{(a relation between } A \text{ and } C\text{)}$$

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

$$\begin{array}{ll} \alpha : (C \to \texttt{Prop}) \to A & \text{constant functions only?} \\ \alpha : (C \to \texttt{bool}) \to A & \text{can only depend on finitely many } C\text{'s} \end{array}$$

Example: $\alpha(S) = [\inf S, \sup S]$ is not computable, because inf and sup are not computable for infinite sets of integers.

Galois connections are problematic in general

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

Example 1: rational intervals.

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

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

Galois connections are problematic in general

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

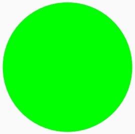
Example 1: rational intervals.

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

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

Example 2: polyhedra.

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



Galois connections are problematic in general

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

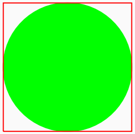
Example 1: rational intervals.

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

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

Example 2: polyhedra.

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



Galois connections are problematic in general

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

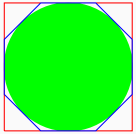
Example 1: rational intervals.

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

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

Example 2: polyhedra.

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



Abstract interpretation without abstraction functions

(P. Cousot et R. Cousot, *Abstract interpretation frameworks*, JLC, 1992.)
(D. Pichardie, *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*, thèse, U. Rennes 1, 2005.)

The abstraction function α is required only to calculate abstract operators. If we are gven an abstract operator $F^\#$ pour l'opérateur F, the concretization γ suffices to prove

- the semantic soundness of $F^\#$:

$$x \in \gamma(a) \Rightarrow F(x) \in \gamma(F^\#(a))$$

- optionally, the relative optimality of $F^\#$:

$$(\forall x \in \gamma(a),\ F(x) \in \gamma(a')) \Rightarrow F^\#(a) \leq a'$$

This approach works perfectly in type theory.

Soundness is essential, optimality is optional

Optimality is not required to obtain a sound analyzer (no alarms $\Rightarrow$ no run-time errors).

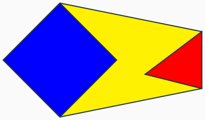
Making optimality optional enables us to simplify the abstract interpreter and its soundness proof:

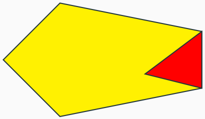
- Abstract operators that return over-approximations (or just $\top$) in uncommon or expensive cases.
- Join operators $\sqcup$ that return an upper bound of their arguments, but not always the least upper bound.
- Fixed-point iterations that produce a post-fixed point but not necessarily the least fixed point.
- Approaches by validation a posteriori.

Validation a posteriori

Abstract operations can be implemented by an unverified external “oracle”, provided we can easily validate the result a posteriori. Only the validator algorithm needs to be proved sound.

Example: the join $\sqcup$ of two polyhedra.





Computing the join vs. Inclusion test

(convex hull) (Presburger formula)

Mechanizing an abstract interpreter for IMP

Modeling abstract domains

As module interfaces:

- `VALUE_ABSTRACTION`: abstraction of integer values
- `STORE_ABSTRACTION`: abstraction of stores

Each interface declares:

- A type `t` of abstract “things”.
- A predicate `In` connecting concrete things and abstract things.
  (`In` c a can be read as $c \in \gamma(a)$)
- Abstract operations over type `t`
  (arithmetic operations; `get` and `set` for stores).
- The statements of semantic soundness for these operations.

(See Coq file `AbstrInterp`.)

Abstract interpretation of arithmetic expressions

Let `ST` be a store abstraction and `V` the corresponding value abstraction.

```
Fixpoint Aeval (a: aexp) (S: ST.t) : V.t :=
  match a with
  | CONST n => V.const n
  | VAR x => ST.get x S
  | PLUS a1 a2 => V.add (Aeval a1 S) (Aeval a2 S)
  | MINUS a1 a2 => V.sub (Aeval a1 S) (Aeval a2 S)
  end.
```

(What else could we possibly write?)

Abstract interpretation of commands

Compute the abstract store “after” executing command *c* as a function of the abstract store “before” *S*.

```
Fixpoint Cexec (c: com) (S: ST.t) : ST.t :=
  match c with
  | SKIP => S
  | ASSIGN x a => ST.set x (Aeval a S) S
  | SEQ c1 c2 => Cexec c2 (Cexec c1 S)
  | IFTHENELSE b c1 c2 => ST.join (Cexec c1 S) (Cexec c2 S)
  | WHILE b c => postfixpoint (fun X => ST.join S (Cexec c X))
  end.
```

Abstract interpretation of commands

```
Fixpoint Cexec (c: com) (S: ST.t) : ST.t :=
  match c with
  | SKIP => S
  | ASSIGN x a => ST.set x (Aeval a S) S
  | SEQ c1 c2 => Cexec c2 (Cexec c1 S)
  | IFTHENELSE b c1 c2 => ST.join (Cexec c1 S) (Cexec c2 S)
  | WHILE b c => postfixpoint (fun X => ST.join S (Cexec c X))
  end.
```

For the time being, the analysis makes no attempt to determine the value of a Boolean expression

$\rightarrow$ we execute abstractly both branches `then` and `else`

$\rightarrow$ we take the join of their final stores.

Abstract interpretation of commands

```
Fixpoint Cexec (c: com) (S: ST.t) : ST.t :=
  match c with
  | SKIP => S
  | ASSIGN x a => ST.set x (Aeval a S) S
  | SEQ c1 c2 => Cexec c2 (Cexec c1 S)
  | IFTHENELSE b c1 c2 => ST.join (Cexec c1 S) (Cexec c2 S)
  | WHILE b c => postfixpoint (fun X => ST.join S (Cexec c X))
  end.
```

Let X be the abstract store “before” the loop body c.

- First iteration: the store is S, therefore $S \leq X$.
- Next iterations: the store is `Cexec` $c\ X$, therefore `Cexec` $c\ X \leq X$.

We solve these two constraints by computing a post-fixed point.

Semantic soundness proof

We show easily that the result of a concrete execution belongs (in the sense of the “In” predicates) to the result of the corresponding abstract execution.

```
Lemma Aeval_sound:
  forall s S a,
  ST.In s S -> V.In (aeval a s) (Aeval a S).

Theorem Cexec_sound:
  forall c s s' S,
  ST.In s S -> cexec s c s' -> ST.In s' (Cexec c S).
```

An example of a store abstraction

Parameterized by a value abstraction `V`.

Abstract stores = functions `ident` $\rightarrow$ `V.t`
such that inclusion $\forall x,$ `V.le` $(f\ x)\ (g\ x)$ is decidable.

Representation: $\perp$ or a finite function `ident` $\rightarrow$ `V.t`
(default value: `V.top`)

Appropriate for all non-relational analyses.

An example of a value abstraction

Abstract domain = the flat lattice of integers:

$$\begin{tikzcd}
 & & \top = \mathbb{Z} \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] \arrow[drrr, dash] & & & \\
\{-1\} \arrow[drr, dash] & \{0\} \arrow[dr, dash] & \{1\} \arrow[d, dash] & \{2\} \arrow[dl, dash] & \{3\} \arrow[dll, dash] & \cdots \arrow[dlll, dash] \\
 & & \bot = \emptyset & & &
\end{tikzcd}$$

Abstract operations are trivial:

$$\bot +^{\#} x = x +^{\#} \bot = \bot \quad \{n_1\} +^{\#} \{n_2\} = \{n_1 + n_2\} \quad \top +^{\#} x = x +^{\#} \top = \top$$

Improving the analysis of conditionals

So far we do not analyze Boolean expressions
$\rightarrow$ both `then` and `else` branches of an `if` are assumed taken.

We can do better if abstract information suffices to statically resolve the condition of an `if`. Example:

```
x := 0;
if x = 0 then y := 1 else y := 2
```

Our constant analysis infers $y^{\#} = \{1\} \sqcup \{2\} = \top$.

Since $x^{\#} = \{0\}$ before the `if`, the `else` branch is never executed, and we should have $y^{\#} = \{1\}$ in the end.

Improving the analysis of conditionals

Even when the Boolean expression cannot be resolved statically, the analysis can learn interesting facts by considering which branch of an `if` is taken.

$x^\# = \top$ initially

```
if x = 0 then
```

we learn that $x^\# = \{0\}$

```
    y := x + 1
```

therefore $y^\# = \{1\}$

```
else
    y := 1
```

here we also have $y^\# = \{1\}$

therefore $y^\# = \{1\}$ and not $\top$

Improving the analysis of loops

We can also learn something from the fact that a `while` loop terminates:

$x^\# = \top$ initially

```
while not (x = 10) do
    x := x + 1
done
```

we learn that $x^\# = \{10\}$

A more realistic example with intervals instead of constants:

$x^\# = \top = [-\infty, +\infty]$ initially

```
while x <= 1000 do
    x := x + 1
done
```

we learn that $x^\# = [1001, +\infty]$

Backward analysis of expressions

`assume_test` *b res S*

returns an abstract store $S' \leq S$ reflecting the fact that b (a Boolean expression) evaluates to *res* (`true` or `false`).

`assume_eval` *a Res S*

returns an abstract store $S' \leq S$ reflecting the fact that a (an arithmetic expression) evaluates to an integer that belongs to *Res* (an abstract value).

Examples:

$\texttt{assume\_test}\ (x = 0)\ \texttt{true}\ (x \mapsto \top) = (x \mapsto \{0\})$

$\texttt{assume\_test}\ (x = 0)\ \texttt{true}\ (x \mapsto \{1\}) = \bot$

$\texttt{assume\_eval}\ (x + 1)\ \{10\}\ (x \mapsto \top) = (x \mapsto \{9\})$

Backward analysis of expressions

We enrich the abstract domain of values with inverse abstract tests `eq_inv`, `ne_inv`, `le_inv`, `gt_inv`.

Formally:

$$\begin{aligned}\texttt{le\_inv}\ N_1\ N_2 \quad = \quad & (\alpha\{x \mid x \in \gamma(N_1), y \in \gamma(N_2), x \le y\}, \\ & \ \alpha\{y \mid x \in \gamma(N_1), y \in \gamma(N_2), x \le y\})\end{aligned}$$

Examples using intervals:

```
eq_inv [0,5] [4,9] = ([4,5], [4,5])
le_inv [0,9] [2,5] = ([0,5], [2,5])
```

(See Coq file `AbstrInterp2`.)

Improved analysis of commands

```
Fixpoint Cexec (c: com) (S: ST.t) : ST.t :=
  match c with
  | SKIP => S
  | ASSIGN x a => ST.set x (Aeval a S) S
  | SEQ c1 c2 => Cexec c2 (Cexec c1 S)
  | IFTHENELSE b c1 c2 =>
      ST.join (Cexec c1 (assume_test b true S))
              (Cexec c2 (assume_test b false S))
  | WHILE b c =>
      assume_test b false
        (postfixpoint
          (fun X => ST.join S (Cexec c (assume_test b true X))))
  end.
```

Computing fixed points with widening

Improving the computation of post-fixed points

Analyzing loops require a post-fixed point to be computed.

In the third lecture we saw two approaches:

1. An approach based on the Knaster-Tarski theorem:
   an iteration that terminates provided there are no infinite, strictly-increasing sequences of abstract values
   $N_0 < N_1 < N_2 < \cdots$
2. An approach using "fuel":
   we return $\top$ after N unsuccessful iterations.

Approach 1 does not always apply, or is too slow.

Approach 2 is too imprecise.

Non-well-founded abstract domains

Many interesting abstract domains have infinite, strictly-increasing sequences. For example, intervals:

$$[0, 0] < [0, 1] < [0, 2] < \cdots < [0, n] < \cdots$$

This is an issue when analyzing non-counted loops:

```
x := 0;
while cond do x := x + 1
```

$x^\#$ is successively $[0, 0]$, then $[0, 1]$, then $[0, 2]$, then …

Excessively slow convergence

In other cases, Knaster-Tarski iteration terminates but takes too long.

```
x := 0;
while x <= 1000 do x := x + 1
```

Starting from $x^\# = [0, 0]$, it takes 1000 iterations to reach the fixed point $x^\# = [0, 1000]$.

Excessively imprecise convergence

The fuel-based approach converges in at most N iterations. But in the case where it returns $\top$ we lose too much information.

```
x := 0;
y := 0;
while x <= 1000 do x := x + 1
```

In the final abstract store, not only $x^{\#} = \top$ but also $y^{\#} = \top$.

Widening

A widening operator $\nabla : A \to A \to A$ computes an upper bound of its two arguments, chosen big enough to ensure that the following iteration converges always and quickly:

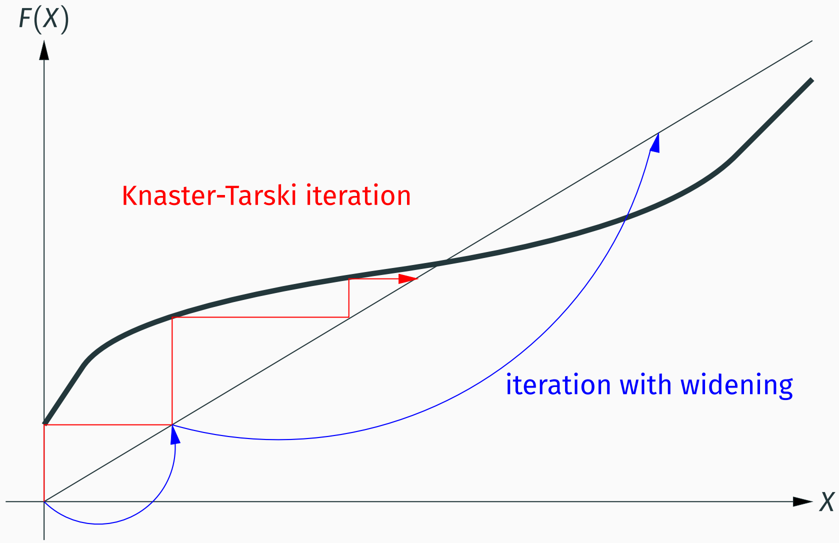
$$X_0 = \bot \qquad X_{i+1} = \begin{cases} X_i & \text{if } F(X_i) \leq X_i \\ X_i \nabla F(X_i) & \text{otherwise} \end{cases}$$

The limit of this sequence is a post-fixed point of F.

Example: the classic widening for intervals

$$\begin{aligned}[l_1, h_1] \nabla [l_2, h_2] = \ & [\, \texttt{if } l_2 < l_1 \texttt{ then } -\infty \texttt{ else } l_1, \\ & \quad \texttt{if } h_2 > h_1 \texttt{ then } +\infty \texttt{ else } h_1 \,]\end{aligned}$$

Widening in action



Static analysis with widening

```
x := 0;
while x <= 1000 do x := x + 1
```

The abstraction of `x` is a post fixed-point of the operator $F(X) = [0,0] \cup (X \cap [-\infty, 1000]) + 1$.

$X_0 = \bot$
$X_1 = X_0 \nabla F(X_0) = \bot \nabla [0,0] = [0,0]$
$X_2 = X_1 \nabla F(X_1) = [0,0] \nabla [0,1] = [0,+\infty]$
X_2 is a post fixed-point: $F(X_2) = [0,1001] \leq [0,+\infty]$.

Final abstraction: $x^\# = [0,+\infty] \cap [1001,+\infty] = [1001,+\infty]$.

Narrowing the post-fixed point

We can obtain a better post-fixed point by iterating a few more rounds.

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

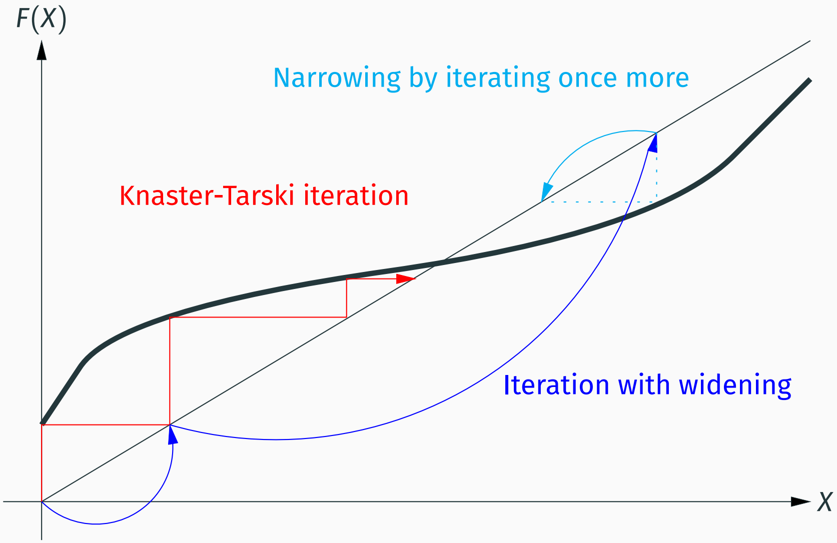
If F is increasing, every Y_i is a post-fixed point: $F(Y_i) \leq Y_i$.

Often, $Y_i < Y_0$, which gives a more precise post-fixed point.

We can stop iteration when Y_i is a fixed point, or after a fixed number of rounds. We can also use a narrowing operator to accelerate convergence.

(P. Cousot et R. Cousot, *Comparing the Galois connection and widening/narrowing approaches to abstract interpretation*, PLILP 1992)

Widening plus narrowing in action



Static analysis with widening and narrowing

```
x := 0;
while x <= 1000 do x := x + 1
```

The abstraction of `x` is a post-fixed point of the operator
$F(X) = [0, 0] \cup (X \cap [-\infty, 1000]) + 1$.

The post-fixed point found by iteration with widening is $[0, +\infty]$.

$Y_0 = [0, +\infty]$
$Y_1 = F(Y_0) = [0, 1001]$
$Y_2 = F(Y_1) = [0, 1001]$

The final post-fixed point is $Y_1 = [0, 1001]$ (it's a fixed point).

Final abstract result: $x^{\#} = [0, 1001] \cap [1001, +\infty] = [1001, 1001]$.

Implementation in the Coq development

We enrich the `VALUE_ABSTRACTION` and `STORE_ABSTRACTION` interfaces with a `widen` operator and the properties that guarantee that the widened iteration order

```
Definition widen_order (S S1: t) :=
  exists S2, S = widen S1 S2 /\ ble S2 S1 = false.
```

is well founded.

We implement the computation of post fixed-points by Noetherian recursion on this order.

(See the Coq file `AbstrInterp2`.)

Summary

Summary

The abstract interpretation approach results in very modular analyzers:

- One abstract interpreter for each programming language.
- Abstract domains that are independent of the language.
- Mechanisms to combine domains together.

Soundness w.r.t. the concrete semantics is obtained either by construction (in the “calculational” approach), or by manual proofs that are modular too.

Relational analyses are more difficult (but more precise!) than the non-relational analyses studied in this lecture.

To be continued in David Pichardie’s seminar of January 30th 2020.

References

References

Foundations of abstract interpretation:

- P. Cousot and R. Cousot, *Basic Concepts of Abstract Interpretation*, in *Building the Information Society*, Kluwer, 2004.
- P. Cousot, *Abstract Interpretation Based Formal Methods and Future Challenges*, LNCS(2000), 2001.

A few industrial applications:

- D. Kästner et al, *Astrée: Proving the Absence of Runtime Errors*, ERTS 2010.
- M. Fähndrich, F. Logozzo, *Static Contract Checking with Abstract Interpretation*, FoVeOOS 2010.
- C. Ferdinand, R. Heckmann, R. Wilhelm, *Analyzing the Worst-Case Execution Time by Abstract Interpretation of Executable Code*, ASWSD 2004.

Mechanizations of abstract interpreters:

- T. Nipkow, G. Klein, *Concrete Semantics*, chap. 13.
- J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie, *A Formally-Verified C Static Analyzer*, POPL 2015.