Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

APA
Abstract Interpretation

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

May 3, 2012

1. Abstract interpretation

Abstract Interpretation

Abstract Interpretation

=

analysis as a simplification of running a computer program.

Examples

- During program execution we compute the values of variables.
  - And our location in the program.
- During abstract interpretation we might
  - compute only the signs of integer variables,
  - compute where closures are created, but not the closures themselves,
  - compute only the lengths of lists,
  - compute only the types of variables.
- Typically, but not necessarily, we compute this for any given location.
- The right simplification depends on the analysis we are attempting.

The benefits of good abstractions

- For certain "good" abstract interpretations, soundness of the analysis follows "immediately" from the soundness of the semantics of the language.
- The latter needs to be proved only once, but many analyses may benefit.
- Semantics must be formally defined.
  - E.g., operational semantics, i.e., specification of interpreter
- Since static analyses must be sound for all executions, we need a collecting semantics for the language.
- Abstracting to a complete lattice with ACC gives guarantee of termination.

The State is everything

- An interpreter keeps track of the state of the program.
- Usually it contains:
  - What program point are we at?
  - For every variable, what value does it currently have?
  - What does the stack look like?
  - What is allocated on the heap?

Examples

- For `While` without procedures we track only the program point and the variables to value mapping.
- To deal with procedures, also track the stack.
- The state is determined by the language constructs we support.
  - Adding **new** implies the need to keep track of the heap.
- For the moment, we assume

$$\textbf{State} = \textbf{Lab} \times (\textbf{Var} \to \textbf{Data})$$

where **Data** typically contains integers, reals and booleans.

State too static

- In abstract interpretation we simplify the state.
- And operations on the state should behave consistently with the abstraction.
- What if the state is already so information poor that the information we want is not in the state to begin with?
- Our state

  $$\textbf{State} = \textbf{Lab} \times (\textbf{Var} \rightarrow \textbf{Data})$$

  has only momentaneous information:
- It does not record dynamic information for the program, e.g., executions.

The need for dynamic information

- Many program analyses concern dynamic properties.
- Examples:
  - Record the minimum and maximum value an integer identifier may take.
  - In a dynamically typed language: compute all types a variable may have.
  - Record all the function abstractions an identifier might evaluate to.
  - Record the set of pairs (x, ℓ) in case x may have gotten its last value at program point ℓ.
- We must first enrich the state to hold this information.

Single execution versus all executions

- Static analysis results should hold for *all* runs.
- Code is only dead if all executions avoid it.
- An interpreter considers only a single execution at the time.
- Redefine semantics to specify all executions "in parallel".
- This is called a collecting semantics.
- Static analysis is on a simplified version (abstraction) of the collecting semantics.
  - Because, usually, the collecting semantics is very infinite.

Collecting semantics

- A collecting semantics for `While` might record sets of execution histories:

$$\textbf{State} = \mathcal{P}([(\textbf{Lab}, \text{Maybe}(\textbf{Var}, \textbf{Data}))])$$

- Example: `if` $[$`x > 0`$]^1$ `then` $[$`y := -3`$]^2$ `else` $[$`skip`$]^3$
- $\{[(?, \text{Just }(x, 0)), (?, \text{Just }(y, 0)), (1, \text{Nothing}), (3, \text{Nothing})],$
  $[(?, \text{Just }(x, 2)), (?, \text{Just }(y, 0)), (1, \text{Nothing}), (2, \text{Just }(y, -3))]$

A different collecting semantics

- Consider **State** $=$ **Lab** $\to \mathcal{P}($**Var** $\to$ **Data**$)$.
  - Sets of functions telling us what values variables can have right before a given program point.
- We repeat: `if` [`x > 0`]^1 `then` [`y := -3`]^2 `else` [`skip`]^3
- For the above program we have (given the initial values): $[1 \mapsto \{[x \mapsto 0, y \mapsto 0], [x \mapsto 2, y \mapsto 0]\},$ $2 \mapsto \{[x \mapsto 2, y \mapsto 0]\}, 3 \mapsto \{[x \mapsto 0, y \mapsto 0]\}]$
- At the end of the program, we have $\{[x \mapsto 2, y \mapsto -3], [x \mapsto 0, y \mapsto 0]\}$
- The semantics does not record that $[x \mapsto 2, y \mapsto 0]$ leads to $[x \mapsto 2, y \mapsto -3]$.

Variations

- Also track the heap and/or stack (if the language needs it).
- In an instrumented semantics information is stored that does not influence the outcome of the execution.
  - For example, timing information.
- Choose one which is general enough to accommodate all your analyses.
  - You cannot analyze computation times if there is no information about it in your collecting semantics

The need to abstract

- We cannot compute all the states for an arbitrary program: it might take an infinite amount of time and space.
- We now must simplify the semantics.
- How far?
  - Trade-off between resources and amount of detail.
- The least one can demand is that the amount of time is finite.
- In some cases, we have to give up more detail than we can allow.
  - Therefore: widening

Example abstractions

- We take $\mathcal{P}(\textbf{Var} \to \textbf{Data})$ as a starting point.
- Example: $S = \{[x \mapsto 2, y \mapsto 0], [x \mapsto -2, y \mapsto 0]\}$
- Abstract to $\textbf{Var} \to \mathcal{P}(\textbf{Data})$ (relational to independent):
  - S now becomes $[x \mapsto \{-2, 2\}, y \mapsto \{0\}]$.
- Abstract further to intervals $[x, y]$ for $x \leq y$:
  - S now becomes represented by $[x \mapsto [-2, 2], y \mapsto [0, 0]]$
- Abstract further to $\textbf{Var} \to \mathcal{P}(\{0, -, +\})$:
  - S now becomes $[x \mapsto \{-, 0, +\}, y \mapsto \{0\}]$.
- Mappings are generally not injective:
  $\{[x \mapsto 2, y \mapsto 0], [x \mapsto -2, y \mapsto 0], [x \mapsto 0, y \mapsto 0]\}$ also maps to $[x \mapsto \{-, 0, +\}, y \mapsto \{0\}]$.

Computing with abstract values

- Consider: you have an interpreter for your language.
- It knows how to add integers, but not how to add signs.
- Would be great if the operators followed immediately from the abstraction.
- This is the case, but the method is not constructive:
  - How to (effectively) compute $\{-\} +_S \{-\}$ in terms of $+$ for integers?
- It does give some correctness criteria for the abstracted operators: the result of $\{-\} +_S \{-\}$ must include $-$.

Non-determinism

- Consider abstraction from
$$\mathbf{Lab} \to \mathcal{P}(\mathbf{Var} \to \mathbf{Z})$$
to
$$\mathbf{Lab} \to \mathbf{Var} \to \mathcal{P}(\{0, -, +\}) \ .$$
- When we add integers, the result is deterministic: two integers go in, one comes out.
- If we add signs $+$ and $-$, then we must get $\{+, 0, -\}$.
- The abstract add is non-deterministic.
- Another reason for working with sets of abstraction of integers.
  - We already needed those to deal with sets of executions.

Connecting back to dataflow analysis

- Practically, Abstract Interpretation concerns itself with the "right" choice of lattice, and how to compute safely with its elements.
- Assume semantics is $L = \textbf{Lab}_* \to \mathcal{P}(\textbf{Var}_* \to \textbf{Z})$ where $\sqsubseteq$ is elementwise $\subseteq$.
  - Forms a complete lattice, but does not satisfy ACC!
- For Constant Propagation, abstract L to

$$M = \textbf{Lab}_* \to (\textbf{Var}_* \to \textbf{Z}^\top)_\perp \text{ with } \textbf{Z}^\top = \textbf{Z} \cup \{\top\} \ .$$

- M does have ACC.

The abstraction function

- Recall:

$$L = \textbf{Lab}_* \to \mathcal{P}(\textbf{Var}_* \to \textbf{Z})$$
$$M = \textbf{Lab}_* \to (\textbf{Var}_* \to \textbf{Z}^\top)_\perp \text{ with } \textbf{Z}^\top = \textbf{Z} \cup \{\top\}$$

- For each label, $\alpha : L \to M$ maps $\emptyset$ to $\perp$, collects all values for a given variable together in a set and then maps $\{i\}$ to i and others to $\top$.
- Example:

$$\alpha(f) = [1 \mapsto [x \mapsto \top, y \mapsto 0], 2 \mapsto [x \mapsto 8, y \mapsto 1]]$$

where $f = [1 \mapsto \{[x \mapsto -8, y \mapsto 0], [x \mapsto 8, y \mapsto 0]\},$
$\qquad 2 \mapsto \{[x \mapsto 8, y \mapsto 1]\}]$

Analyze ... analyze

- Solve equations on the complete lattice M (MFP).
- Initial value $\iota = \alpha(x)$, where x represents what values the program may legally start with.
- Variables are initialized to zero: choose $\iota = \lambda v.\{0\}$.
- Variables are not initialized: take $\iota = \lambda v.\top$.

The concretization function

- Afterwards, if necessary, transform the solution back to one for L.
- Transformation by concretization function γ from M to L.
- Let $m = [1 \mapsto [x \mapsto \top, y \mapsto 0], 2 \mapsto [x \mapsto 8, y \mapsto 1]]$.
- Then $\gamma(m) = [1 \mapsto \{[x \mapsto a, y \mapsto 0] \mid a \in \mathbf{Z}\},$
  $2 \mapsto \{[x \mapsto 8, y \mapsto 1]\}]$
- Note: $\gamma(m)$ is infinite!
  - But the original concrete value was not.
- If α and γ have certain properties then abstraction may lose precision, but not correctness.

2. Galois Connections and Galois Insertions

"Good" abstractions

- Not every combination of abstraction and concretization function is "good".
- When we abstract, we prefer the soundness of the concrete lattice to be inherited by the abstract one.
  - In particular, the soundness of an analysis derives from the soundness of the collecting operational semantics.
  - NB: executing the collecting operational semantics is also a sort of analysis.
- The Cousots defined when this is the case.
- These abstractions are termed Galois Insertions
  - Slightly more general, Galois Connections aka adjoints.
- Abstraction can be a stepwise process.
- In the end everything relates back to the soundness of the collecting semantics.

Abstraction and concretization

- Let $L = (\mathcal{P}(\mathbf{Z}), \subseteq)$ and $M = (\mathcal{P}(\{0, +, -\}), \subseteq)$.
- Let $\alpha : L \to M$ be the abstraction function defined as

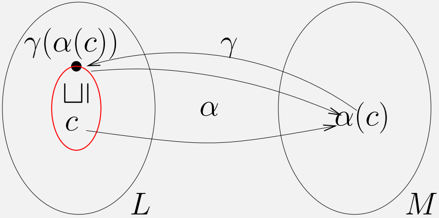
  $$\alpha(S) = \{\text{sign}(z) \mid z \in S\} \text{ where}$$

  $\text{sign}(x) = 0$ if $x = 0$, $+$ if $x > 0$ and $-$ if $x < 0$.
- For example: $\alpha(\{0, 2, 20, 204\}) = \{0, +\}$ and $\alpha(O) = \{-, +\}$ where O is the set of odd numbers.
- Obviously, α is monotone: if $x \subseteq y$ then $\alpha(x) \subseteq \alpha(y)$.

Abstract and concretization

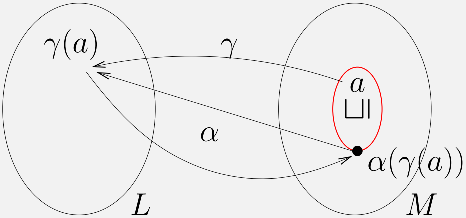
- Let $L = (\mathcal{P}(\mathbf{Z}), \subseteq)$ and $M = (\mathcal{P}(\{0, +, -\}), \subseteq)$.
- The concretization function γ is defined by:
  $$\begin{aligned}\gamma(T) = &\ \{1, 2, \ldots \mid + \in T\} \\ &\cup \{\ldots, -2, -1 \mid - \in T\} \\ &\cup \{0 \mid 0 \in T\}\end{aligned}$$
- Again, obviously, γ monotone.
- Monotonicity of α and γ and two extra demands make (L, α, γ, M) into a Galois Connection.

Demand number 1



- α removes detail, so when going back to L we expect to lose information.
  - Gaining information would be non-monotone.
- Demand 1: for all $c \in L$, $c \sqsubseteq_L \gamma(\alpha(c))$
- For the set O of odd numbers, $O \subseteq \gamma(\alpha(O)) = \gamma(\{+, -\}) = \{\ldots, -2, -1, 1, 2, \ldots\}$
- What about $\alpha(\gamma(\alpha(c)))$? It equals $\alpha(c)$.

Demand number 2



- Demand 2: for all $a \in M$, $\alpha(\gamma(a)) \sqsubseteq_M a$
- Dual version of demand 1.
- Abstracting the concrete value of an abstract values gives a lower bound of the abstract value.
- For $a = \{+, 0\} \in M$, $\alpha(\gamma(a)) = \alpha(\{0, 1, 2, \ldots\}) = \{0, +\}$
- What about $\gamma(\alpha(\gamma(a)))$? It equals $\gamma(a)$.

Galois Insertions

- Sometimes Demand 2 becomes
  Demand 2': for all $a \in M$, $\alpha(\gamma(a)) = a$.
- It is then called a Galois Insertion.
- Often an Insertion is a Connection, but not always.
- A Connection can always be made into an Insertion
  - Remove values from abstract domain that cannot be reached.

A Connection that is not an Insertion

- Consider the complete lattices $L = (\mathcal{P}(\mathbf{Z}), \subseteq)$ and $M = \mathcal{P}(\{0, +, -\} \times \{\text{odd}, \text{even}\}, \ldots)$ and the obvious abstraction $\alpha : L \to M$.
- Concretization: what is $\gamma(\{(0, \text{odd}), (-, \text{even})\})$?

A Connection that is not an Insertion

- Consider the complete lattices $L = (\mathcal{P}(\mathbf{Z}), \subseteq)$ and $M = \mathcal{P}(\{0, +, -\} \times \{\text{odd}, \text{even}\}, \ldots)$ and the obvious abstraction $\alpha : L \to M$.
- Concretization: what is $\gamma(\{(0, \text{odd}), (-, \text{even})\})$?
- What happens to $(0, \text{odd})$? We ignore it!
- Abstracting back:

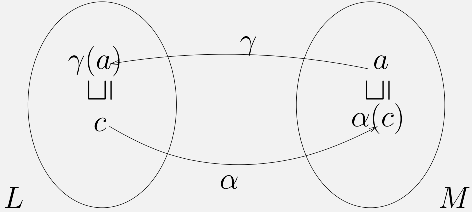
  $$\alpha(\gamma(\{(0, \text{odd}), (-, \text{even})\})) \text{ gives } \{(-, \text{even})\}$$

  and note that

  $$\{(-, \text{even})\} \subset \{(0, \text{odd}), (-, \text{even})\}$$

- Why be satisfied before you have na Insertion?
  - The Connection may be much easier to specify.

Adjoints



- Now α and γ are total functions between L and M.
- Abstraction of less gives less: $c \sqsubseteq \gamma(a)$ implies $\alpha(c) \sqsubseteq a$.
- Concretization of more gives more: $\alpha(c) \sqsubseteq a$ implies $c \sqsubseteq \gamma(a)$.
- Together: (L, α, γ, M) is an adjoint.
- Thm: adjoints are equivalent to Galois Connections.

Some (related) example abstractions

- Reachability:
  $M = \mathbf{Lab}_* \to \{\bot, \top\}$ where
  $\bot$ describes "not reachable",
  $\top$ describes "might be reachable".
- Undefined variable analysis:
  $M = \mathbf{Var}_* \to \{\bot, \top\}$ where
  $\top$ describes "might get a value",
  $\bot$ describes "never gets a value".
- Undefined before use analysis:
  $M = \mathbf{Lab}_* \to \mathbf{Var}_* \to \{\bot, \top\}$

Combinators for Galois Connections

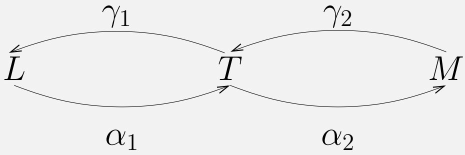
- Building Galois Connections from smaller ones.
- Reuse to save on proofs and implementations.
- Quick look at:
  - composition of Galois Connections,
  - total function space,
  - independent attribute combination,
  - direct product.

The running example

- Construct a Galois Connection from the collecting semantics
$$L = \textbf{Lab}_* \to \mathcal{P}(\textbf{Var}_* \to \textbf{Z})$$
to
$$M = \textbf{Lab}_* \to \textbf{Var}_* \to \textbf{Interval}$$
- M can be used for Array Bound Analysis:
  - Of interest are only the minimal and maximal values.
- First we abstract L to $T = \textbf{Lab}_* \to \textbf{Var}_* \to \mathcal{P}(\textbf{Z})$, and then T to M.
- The abstraction α from L to M is the composition of these two.
- The intermediate Galois Connections are built using the total function space combinator.

Galois Connection/Insertion composition

- The general picture:



- The composition of the two can be taken directly from the picture:

$$(L, \alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2, M)\ .$$

- Thm: always a Connection (Insertion) if the two ingredients are Connections (Insertions)

To get from L to T

- $L = \mathbf{Lab}_* \to \mathcal{P}(\mathbf{Var}_* \to \mathbf{Z})$ is a relational lattice, $T = \mathbf{Lab}_* \to \mathbf{Var}_* \to \mathcal{P}(\mathbf{Z})$ is only suited for independent attribute analysis.
- This kind of step occurs quite often: define separately for reuse.
- Example:

$$[1 \mapsto \{[x \mapsto 2, y \mapsto -3], [x \mapsto 0, y \mapsto 0]\}]$$

should abstract to

$$[1 \mapsto [x \mapsto \{0, 2\}, y \mapsto \{-3, 0\}]]\ .$$

Abstraction

- We first try to get from
  $L' = \mathcal{P}(\mathbf{Var}_* \to \mathbf{Z})$ to
  $T' = \mathbf{Var}_* \to \mathcal{P}(\mathbf{Z})$.
  - "Add" back the $\mathbf{Lab}_*$ by invoking the total function space combinator.
- Start by finding a Galois Connection (α'_1, γ'_1) from $L' = \mathcal{P}(\mathbf{Var}_* \to \mathbf{Z})$ to $T' = \mathbf{Var}_* \to \mathcal{P}(\mathbf{Z})$.
- $\{[x \mapsto 2, y \mapsto -3], [x \mapsto 0, y \mapsto 0]\}$ should abstract to $[x \mapsto \{0, 2\}, y \mapsto \{-3, 0\}]$.
- $\alpha'_1(S) = \lambda v \;.\; \{z \mid \exists f \in S \;.\; z = f(v)\}$
  - Collect for each variable v all the values it maps to.

Concretization

- $L' = \mathcal{P}(\textbf{Var}_* \to \textbf{Z})$
  $T' = \textbf{Var}_* \to \mathcal{P}(\textbf{Z})$.
- γ_1' unfolds sets of values to sets of functions,
  - simply by taking all combinations.
- From $[x \mapsto \{0,2\}, y \mapsto \{-3,0\}]$ we obtain
  $\{[x \mapsto 2, y \mapsto -3], [x \mapsto 0, y \mapsto 0],$
  $[x \mapsto 2, y \mapsto 0], [x \mapsto 0, y \mapsto -3]\}$

The total function space combinator

- Let $(L', \alpha'_1, \gamma'_1, T')$ be the Galois Connection just constructed.
- How can we obtain a Galois Connection $(L, \alpha_1, \gamma_1, T)$?
  - Use the total function space combinator.
- For a fixed set, say $S = \textbf{Lab}_*$, $(L', \alpha'_1, \gamma'_1, T')$ is transformed into a Galois Connection between $L = S \to L'$ and $T = S \to T'$.
- L and T are complete lattices if L' and T' are (App. A).
- The construction tells us how to build α_1 and γ_1 out of α'_1 and γ_1.
- Apply primed versions pointwise:
  - For each $\phi \in L$: $\alpha_1(\phi) = \alpha'_1 \circ \phi$ (see also p. 96)
  - Similarly, for each $\psi \in T$: $\gamma_1(\psi) = \gamma'_1 \circ \psi$.

From T to M (quickly)

- What remains is getting from $T = \textbf{Lab}_* \to \textbf{Var}_* \to \mathcal{P}(\mathbf{Z})$ to $M = \textbf{Lab}_* \to \textbf{Var}_* \to \textbf{Interval}$.
- Intervals: $\bot = [\infty, -\infty]$, $[0, 0]$, $[-\infty, 2]$, $\top = [-\infty, \infty]$.
- Abstraction from $\mathcal{P}(\mathbf{Z})$ to **Interval**:
  - if set empty take $\bot$,
  - find minimum and maximum,
  - if minimum undefined: take $-\infty$,
  - if maximum undefined: take ∞.
- Invoke total function space combinator twice to "add" $\textbf{Lab}_*$ and $\textbf{Var}_*$ on both sides.

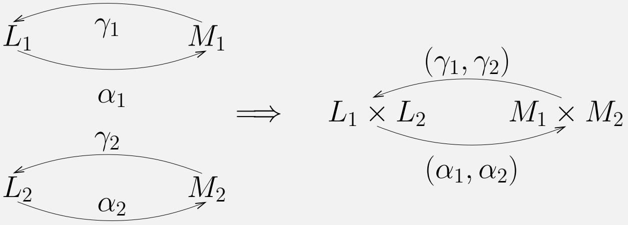
Direct product

- Starting from the lattice $\mathcal{P}(\mathbf{Z})$ we can abstract to $M_1 = \mathcal{P}(\{\text{odd}, \text{even}\})$ and $M_2 = \mathcal{P}(\{-, 0, +\})$.
- Combine the two into one Galois Connection between $L = \mathcal{P}(\mathbf{Z})$ and $M = \mathcal{P}(\{\text{odd}, \text{even}\}) \times \mathcal{P}(\{-, 0, +\})$.
- Given that we have $(L, \alpha_1, \gamma_1, M_1)$ and $(L, \alpha_2, \gamma_2, M_2)$ we obtain $(L, \alpha, \gamma, M_1 \times M_2)$ where
  - $\alpha(c) = (\alpha_1(c), \alpha_2(c))$ and
  - $\gamma(a_1, a_2) = \gamma_1(a_1) \sqcap \gamma_2(a_2)$
- Why take the meet (greatest lower bound)?

Direct product

- Starting from the lattice $\mathcal{P}(\mathbf{Z})$ we can abstract to $M_1 = \mathcal{P}(\{\text{odd}, \text{even}\})$ and $M_2 = \mathcal{P}(\{-, 0, +\})$.
- Combine the two into one Galois Connection between $L = \mathcal{P}(\mathbf{Z})$ and $M = \mathcal{P}(\{\text{odd}, \text{even}\}) \times \mathcal{P}(\{-, 0, +\})$.
- Given that we have $(L, \alpha_1, \gamma_1, M_1)$ and $(L, \alpha_2, \gamma_2, M_2)$ we obtain $(L, \alpha, \gamma, M_1 \times M_2)$ where
  - $\alpha(c) = (\alpha_1(c), \alpha_2(c))$ and
  - $\gamma(a_1, a_2) = \gamma_1(a_1) \sqcap \gamma_2(a_2)$
- Why take the meet (greatest lower bound)?
  - It enables us to ignore combinations (a_1, a_2) that cannot occur.
- $$\begin{aligned}\gamma((\{\text{odd}\}, \{0\})) &= \gamma_1(\{\text{odd}\}) \cap \gamma_2(\{0\}) \\ &= \{\ldots, -1, 1, \ldots\} \cap \{0\} = \emptyset.\end{aligned}$$

The independent attribute method (tupling)



- Example: $L_1 = L$ and $M_1 = M$, and M_2 is some abstraction of L_2 which describes the state of the heap at different program points.
- Define α and γ between $L_1 \times L_2$ and $M_1 \times M_2$ as follows:
  - $\alpha(c_1, c_2) = (\alpha_1(c_1), \alpha_2(c_2))$
  - $\gamma(a_1, a_2) = (\gamma_1(a_1), \gamma_2(a_2))$.
- Abstractions are done independently.

3. Widening

Array Bound Analysis

- We abstracted from $L = \textbf{Lab}_* \to \mathcal{P}(\textbf{Var}_* \to \textbf{Z})$ to $M = \textbf{Lab}_* \to \textbf{Var}_* \to \textbf{Interval}$.
- M prime candidate for Array Bound Analysis:

  At every program point, determine the minimum and maximum value for every variable.

M has its problems

- Consider the program

```
[x := 0]^1
while [x >= 0]^2 do
    [x := x + 1]^3;
```

- The intervals for x in $\text{Analysis}_\circ(2)$ turn out to be

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq \ \ldots$$

- Not having ACC prevents termination.
- When the loop is bounded (e.g., $[\texttt{x < 10000}]^2$) convergence to $[0, 10001]$ takes a long time.

Consider the options

- Two ways out:
  - abstract M further to a lattice that does have ACC, or
  - ensure all infinite chains in M are traversed in finite time.
- In this case, there does not seem to be any further abstraction possible.
- So let's consider the second: widening.

Widening

- Widening $\approx$ a non-uniform coarsening of the lattice.
- We promise not to visit some parts of the lattice.
  - Which parts typically depends on the program.
- Essentially making larger skips along ascending chains than necessary.
- This buys us termination.
- But we pay a price: no guarantee of a least fixed point.
  - By choosing a clever widening we can hope it won't be too bad.

Array Bound Analysis

- Consider the following program:

```
int i, c, n,
int A[20], C[], B[];
C = new int[9];
input n; B = new int[n];
if (A[i] < B[i]) then
   C[i/2] = B[i];
```

- Which bound checks are certain to succeed?
  - Arrays A and C have static sizes, which can be determined 'easily' (resizing is prohibited).
  - Therefore: find the possible values of i.
  - If always $i \in [0, 17]$, then omit checks for A and C.
  - If always $i \in [0, 19]$, then omit checks for A.
  - Nothing to be gained for B: it is dynamic.

The key realization

- For the arrays `A` and `C`, the fact `i` $\in [-20, 300]$ is (almost) as bad as $[-\infty, \infty]$.
- Why then put such large intervals in the lattice?
- Widening allows us to tune (per program) what intervals are of interest.

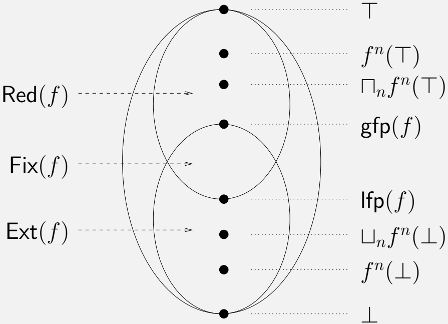
What intervals are interesting?

- Consider, for simplicity, the set of all constants C in a program P.
  - Includes those that are used to define the sizes of arrays.
- What if, when we join two intervals, we consider as result only intervals, the boundaries of which consist of values taken from $C \cup \{-\infty, \infty\}$?
- To keep it safe, every value over $\sup(C)$ must be mapped to ∞, and below $\inf(C)$ to $-\infty$.
- A program has only a finite number of constants: number of possible intervals for every program point is now finite.

Variations

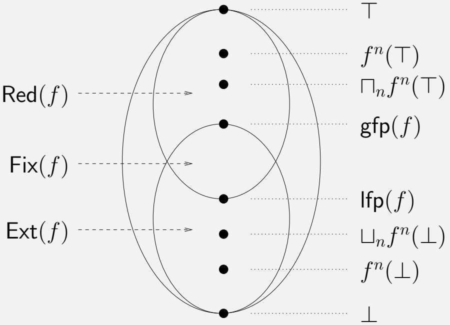
- Which constants work well depends on how the arrays are addressed: `A[2*i + j] = B[3*i] - C[i]`
- Variations can be made: take all constants plus or minus one, etc. etc.
- In a language like `Java` and `C` all arrays are zero-indexed
  - Consider only positive constants (`A[-i]`?).
- What works well can only be empirically established.

Back to the lattice



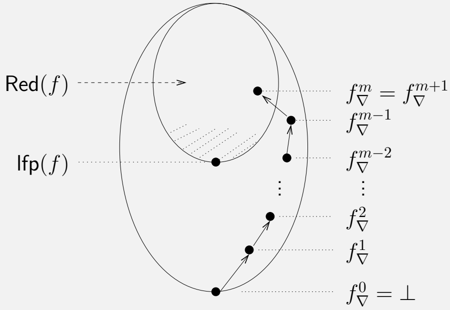
- $\mathsf{Red}(f) = \{x \mid f(x) \sqsubseteq x\}$
- $\mathsf{Ext}(f) = \{x \mid x \sqsubseteq f(x)\}$ and
- $\mathsf{Fix}(f) = \mathsf{Red}(f) \cap \mathsf{Ext}(f)$.

Back to the lattice



- Start from $\bot$ so that we obtain the least fixed point.
- Another possibility is to start in $\top$ and move down. Whenever we stop, we are safe.
  - But....no guarantee that we reach lfp

Pictorial view of widening



- Widening: replace $\sqcup$ with a widening operator ∇ (nabla).
- ∇ is an upper bound operator, but not least: for all $l_1, l_2 \in L : l_1 \sqcup l_2 \sqsubseteq l_1 \nabla l_2$.
- The point: take larger steps in the lattice than is necessary.
- Not precise, but definitely sound.

How widening affects sequences

- Consider a sequence

$$l_0, l_1, l_2, \ldots$$

- Note: any sequence will do.
- Under conditions, it becomes an ascending chain

$$l_0 \sqsubseteq l_0 \nabla l_1 \sqsubseteq (l_0 \nabla l_1) \nabla l_2 \sqsubseteq \ldots$$

- that is guaranteed to stabilize.
- Stabilization point is known to be a reductive point,
  - I.e. a sound solution to the constraints

How widening affects sequences

- Consider a sequence

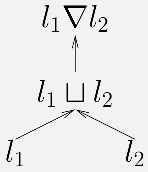
$$l_0, l_1, l_2, \ldots$$

- Note: any sequence will do.
- Under conditions, it becomes an ascending chain

$$l_0 \sqsubseteq l_0 \nabla l_1 \sqsubseteq (l_0 \nabla l_1) \nabla l_2 \sqsubseteq \ldots$$

- that is guaranteed to stabilize.
- Stabilization point is known to be a reductive point,
  - I.e. a sound solution to the constraints
- but is not always a fixed point. Bummer.

What it takes to be ∇



- Let a lattice L be given and ∇ a widening operator, i.e.,
  - for all $l_1, l_2 \in L$: $l_1 \sqsubseteq l_1 \nabla l_2 \sqsupseteq l_2$, and
  - for all ascending chains (l_i), the ascending chain $l_0, l_0 \nabla l_1, (l_0 \nabla l_1) \nabla l_2, \ldots$ eventually stabilizes.
- The latter seems a rather selffulfilling property.

Iterating with ∇

- How can we use ∇ to find a reductive point of a function?
- $f_\nabla^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\nabla^{n-1} & \text{if } n > 0 \land f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1} \\ f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) & \text{otherwise} \end{cases}$
- First argument represent all previous iterations, second represents result of new iteration.

An example

- Define ∇_C to be the following upper bound operator: $[i_1, j_1] \ \nabla_C \ [i_2, j_2] = [\mathsf{LB}_C(i_1, i_2), \mathsf{UB}_C(j_1, j_2)]$ where
  - $\mathsf{LB}_C(i_1, i_2) = i_1$ if $i_1 \leq i_2$, otherwise
  - $\mathsf{LB}_C(i_1, i_2) = k$ where $k = \max\{x \mid x \in C, x \leq i_2\}$ if $i_2 < i_1$

An example

- Define ∇_C to be the following upper bound operator: $[i_1, j_1] \nabla_C [i_2, j_2] = [\mathsf{LB}_C(i_1, i_2), \mathsf{UB}_C(j_1, j_2)]$ where
  - $\mathsf{LB}_C(i_1, i_2) = i_1$ if $i_1 \leq i_2$, otherwise
  - $\mathsf{LB}_C(i_1, i_2) = k$ where $k = \max\{x \mid x \in C, x \leq i_2\}$ if $i_2 < i_1$
  - And similar for UB_C.
  - Exception: $\bot \nabla_C I = I = I \nabla_C \bot$.
- Essentially, only the boundaries of the first argument interval, values from C, and $-\infty$ and ∞ are allowed as boundaries of the result.
- Let $C = \{3, 5, 100\}$. Then
  - $[0, 2] \nabla_C [0, 1] = [0, 2]$
  - $[0, 2] \nabla_C [-1, 2] = [-\infty, 2]$
  - $[0, 2] \nabla_C [1, 14] = [0, 100]$

Ascending chains will stabilize

- Intuition by example.
- Consider the chain $[0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq [0,4] \ldots$ and choose $C = \{3,5\}$.
- From it we obtain the stabilizing chain:

$$\begin{array}{rcl} & & [0,1], \\ [0,1] \nabla_C [0,2] & = & [0,3], \\ [0,3] \nabla_C [0,3] & = & [0,3], \\ [0,3] \nabla_C [0,4] & = & [0,5], \\ [0,5] \nabla_C [0,5] & = & [0,5], \\ [0,5] \nabla_C [0,6] & = & [0,\infty], \\ [0,\infty] \nabla_C [0,7] & = & [0,\infty], \ldots \end{array}$$

- Essentially, we fold ∇ over the sequence.

Analyzing the infinite loop

- Recall the program

```
[x := 0]1
while [x >= 0]2 do
    [x := x + 1]3;
```

- Iterating with ∇_C with $C = \{3, 5\}$ gives

$A_\circ(1)$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$A_\bullet(1)$	$\bot$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,($
$A_\circ(2)$	$\bot$	$[0,0]$	$[0,0]\nabla_C[1,1] = [0,3]$	$[0,5]$	$[0,\infty]$	$[0,\infty$
$A_\bullet(2)$	$\bot$	$[0,0]$	$[0,3]$	$[0,5]$	$[0,\infty]$	$[0,\infty$
$A_\circ(3)$	$\bot$	$[0,0]$	$[0,3]$	$[0,5]$	$[0,\infty]$	$[0,\infty$
$A_\bullet(3)$	$\bot$	$[1,1]$	$[1,4]$	$[1,6]$	$[1,\infty]$	$[1,\infty$

- Note: not all interval boundaries are values from C

Final remarks on widening

- Widening operator ∇ replaces join $\sqcup$:
  - Bigger leaps in lattice guarantee stabilisation.
  - guarantees reductive point, not necessarily a fixed point
- Widening operator: verify the two properties.
- Any complete lattice supports a range of widening operators. Balance cost and coarseness.
- Widening operator often a-symmetric: the first operand is treated more respectfully.
- Widening usually parameterized by information from the program:
  - C is the set of constants occuring in the program.
- We visit a finite, program dependent part of the lattice.