Program and Model Analysis
(Graduiertenkolleg Programm- Und Modell-Analyse)

An Informal Introduction to Abstract Interpretation

Patrick Cousot

pcousot@cs.nyu.edu
http://cs.nyu.edu/~pcousot

cousot@ens.fr
http://www.di.ens.fr/~cousot

Technische Universität München
Fakultät für Informatik

May 22, 2009

Content

Mathematical Semantics

A sample program

Let us start with the following example program.

$$P \triangleq {}^{1}\texttt{x := 1 ; while } {}^{2}\mathfrak{true}\texttt{ do } {}^{3}\texttt{x := (x + 1); od}^{4}.$$

The mathematical semantics of this program can be informally described at follows.

- Execution start at program point ^{1} by assigning 1 to program variable x and goes on at program point ^{2}.
- When at program point ^{2} the evaluation of the loop test yields the value $\mathfrak{true}$ so execution continues at program ^{3} where the value of variable x is incremented by 1 before coming back to ^{2}.
- Since the loop condition is never $\mathfrak{false}$, program point ^{4} is unreachable so program execution never ends.

States

$P \triangleq {}^1\texttt{x := 1 ; while } {}^2\mathtt{true}\texttt{ do } {}^3\texttt{x := (x + 1); od}^4.$

we write $\langle {}^\ell, x\rangle$ for the state of program execution where execution is at program point ℓ, $\ell = {}^1, {}^2, {}^3, {}^4$, and variable x has integer value $x \in \mathbb{Z}$ (where $\mathbb{Z}$ is the set of all mathematical integers).

Execution trace

$P \triangleq {}^1\texttt{x := 1 ; while } {}^2\mathtt{true}\texttt{ do } {}^3\texttt{x := (x + 1); od}^4.$

A complete program execution can be described by the following execution trace which is an infinite sequence of states

$$\langle {}^1, z\rangle\langle {}^2, 1\rangle\langle {}^3, 1\rangle\langle {}^2, 2\rangle\langle {}^3, 2\rangle\ldots\langle {}^2, i\rangle\langle {}^3, i\rangle\langle {}^2, i+1\rangle\ldots$$

where $z \in \mathbb{Z}$ can be any initial integer value of x.

Trace semantics

$P \triangleq {}^1\texttt{x := 1 ; while } {}^2\mathtt{true}\texttt{ do } {}^3\texttt{x := (x + 1); od}^4.$

So the set of all such execution traces is

$$\{\langle {}^1, z\rangle\langle {}^2, 1\rangle\langle {}^3, 1\rangle\langle {}^2, 2\rangle\langle {}^3, 2\rangle\ldots\langle {}^2, i\rangle\langle {}^3, i\rangle\langle {}^2, i+1\rangle\ldots \mid z \in \mathbb{Z}\}$$

Mathematical Invariants

Invariance abstraction

$P \triangleq {}^1\texttt{x := 1 ; while}\ {}^2\mathfrak{true}\ \texttt{do}\ {}^3\texttt{x := (x + 1); od}^4.$

Let us now consider an abstraction of the set of all possible execution traces, which consists in remembering for each program point ℓ, $\ell = {}^1, {}^2, {}^3, {}^4$ the set I_ℓ of possible values that can be taken by variable x when execution reaches program point ℓ along any of these traces.

Invariance semantics

$P \triangleq {}^1\texttt{x := 1 ; while}\ {}^2\mathfrak{true}\ \texttt{do}\ {}^3\texttt{x := (x + 1); od}^4.$

This set I_ℓ is called a program local invariant at program point ℓ. We have

$$\begin{array}{lcl} I_1 & = & \mathbb{Z} \\ I_2 & = & \{z \in \mathbb{Z} \mid z > 0\} \\ I_3 & = & \{z \in \mathbb{Z} \mid z > 0\} \\ I_4 & = & \emptyset \end{array}$$

Mathematical Invariant Equations

Invariance Equations

$P \triangleq {}^1\texttt{x := 1 ; while}\ {}^2\mathfrak{true}\ \texttt{do}\ {}^3\texttt{x := (x + 1); od}^4.$

Observe that the set I_ℓ of possible values of variable x at program point $\ell = {}^1, {}^2, {}^3, {}^4$ satisfies the following conditions.

$$\left\{\begin{array}{lcl} X_1 & = & \mathbb{Z} \\ X_2 & = & \{1\} \cup \{\, x + 1 \mid x \in X_3\} \\ X_3 & = & X_2 \cap \{x \in \mathbb{Z} \mid \mathfrak{true}\} \\ X_4 & = & X_2 \cap \{x \in \mathbb{Z} \mid \mathfrak{false}\} \end{array}\right. \tag{3.1}$$

Invariance Equations

- At program point ^{1} the variable `x` can be initialized by any integer value $z \in \mathbb{Z}$ and so $X_1 = \mathbb{Z}$
- At program point ^{2}, either execution comes from program point ^{1} and so the value of variable `x` is 1 or execution comes from program point ^{2} and so the value of variable `x` is the value x that `x` had at this point ^{3} incremented by 1. So $X_2 = \{1\} \cup (\{x + 1 \mid x \in X_3\}$.
- At program point ^{3}, the possible values of `x` are those at point ^{2} for which the loop condition is true so $X_3 = X_2 \cap \{x \in \mathbb{Z} \mid \mathfrak{true}\} = X_2$.
- At program point ^{4}, the possible values of `x` are those at point ^{2} for which the loop condition is false so $X_4 = X_2 \cap \{x \in \mathbb{Z} \mid \mathfrak{false}\} = \emptyset$.

Fixpoint Equations

These conditions can be understood as a system of fixpoint equations $X = f(X)$ of the form

$$\begin{cases} X_i = f_i(X_1, \ldots, X_4) \\ i = 1, \ldots, 4 \end{cases}$$

with unknowns $X = \langle X_1, \ldots, X_4 \rangle$.

Fixpoint Solutions

$$\begin{cases} X_1 = \mathbb{Z} \\ X_2 = \{1\} \cup \{\, x + 1 \mid x \in X_3\} \\ X_3 = X_2 \cap \{x \in \mathbb{Z} \mid \mathfrak{true}\} \\ X_4 = X_2 \cap \{x \in \mathbb{Z} \mid \mathfrak{false}\} \end{cases} \qquad (3.1)$$

- So solving this system of equations might lead to the desired invariant I.
- However these equations do not have a unique solution. For example $X_1 = X_2 = X_3 = \mathbb{Z}$ and $X_4 = \emptyset$ is another solution which is larger for componentwise set inclusion $\subseteq$.
- So we will prefer the smallest solution (called the *least fixpoint* **lfp** f), which is included in all other solutions[1] and turns out to be I.

[1] by Tarski fixpoint theorem

Solving the Equations by Exhaustive Enumeration

Solving the equations iteratively ...

The least solution $I = \mathbf{lfp}\, f$ of $X = f(X)$ for $\subseteq$ can be calculated iteratively, essentially by enumeration of all possible states reachable from the initial states.

Solving the equations iteratively ... (Cont'd)

- $X^0 = \langle X_1^0,\ X_2^0,\ X_3^0,\ X_4^0\rangle = \langle\emptyset,\ \emptyset,\ \emptyset,\ \emptyset\rangle$ ≀starting with the smallest possible approximationʃ
- $X^1 = \langle X_1^1,\ X_2^1,\ X_3^1,\ X_4^1\rangle = f(X^0) = \langle\mathbb{Z},\ \{1\}\cup\{x+1 \mid x\in X_3^0\},\ X_2^0\cap\{x\in\mathbb{Z}\mid \mathfrak{true}\},\ X_2^0\cap\{x\in\mathbb{Z}\mid\mathfrak{false}\}\rangle = \langle\mathbb{Z},\ \{1\},\ \emptyset,\ \emptyset\rangle$
- $X^2 = \langle X_1^2,\ X_2^2,\ X_3^2,\ X_4^2\rangle = f(X^1) = \langle\mathbb{Z},\ \{1\}\cup\{x+1 \mid x\in X_3^1\},\ X_2^1\cap\{x\in\mathbb{Z}\mid \mathfrak{true}\},\ X_2^1\cap\{x\in\mathbb{Z}\mid\mathfrak{false}\}\rangle = \langle\mathbb{Z},\ \{1\},\ \{1\},\ \emptyset\rangle$
- $X^3 = \langle X_1^3,\ X_2^3,\ X_3^3,\ X_4^3\rangle = f(X^2) = \langle\mathbb{Z},\ \{1\}\cup\{x+1 \mid x\in X_3^2\},\ X_2^2\cap\{x\in\mathbb{Z}\mid \mathfrak{true}\},\ X_2^2\cap\{x\in\mathbb{Z}\mid\mathfrak{false}\}\rangle = \langle\mathbb{Z},\ \{1,2\},\ \{1\},\ \emptyset\rangle$

  This calculation can go on like this ad infinitum since each iteration $X^{i+1} = f(X^i)$ of the equations corresponds to an iteration in the program loop and so adds one more possible value of variable `x` at program point ^{2}. The solution is to use mathematical induction which requires to invent the following inductive hypothesis

Solving the equations iteratively ... (Cont'd)

- $X^{2n} = \langle X_1^{2n},\ X_2^{2n},\ X_3^{2n},\ X_4^{2n}\rangle = \langle\mathbb{Z},\ \{1,\ldots,n\},\ \{1,\ldots,n\},\ \emptyset\rangle$ ≀induction hypothesis which holds for the basis $n = 1$ʃ
- $X^{2n+1} = \langle X_1^{2n+1}, X_2^{2n+1}, X_3^{2n+1}, X_4^{2n+1}\rangle = f(X^{2n}) = \langle\mathbb{Z},\ \{1\}\cup\{x+1 \mid x\in X_3^{2n}\},\ X_2^{2n}\cap\{x\in\mathbb{Z}\mid\mathfrak{true}\},\ X_2^{2n}\cap\{x\in\mathbb{Z}\mid\mathfrak{false}\}\rangle = \langle\mathbb{Z},\ \{1,\ldots,n+1\},\ \{1,\ldots,n\},\ \emptyset\rangle$
- $X^{2n+2} = \langle X_1^{2n+2},\ X_2^{2n+2},\ X_3^{2n+2},\ X_4^{2n+2}\rangle = f(X^{2n+1}) = \langle\mathbb{Z},\ \{1\}\cup\{x+1 \mid x\in X_3^{2n+1}\},\ X_2^{2n+1}\cap\{x\in\mathbb{Z}\mid\mathfrak{true}\},\ X_2^{2n+1}\cap\{x\in\mathbb{Z}\mid\mathfrak{false}\}\rangle = \langle\mathbb{Z},\ \{1,\ldots,n+1\},\ \{1,\ldots,n+1\},\ \emptyset\rangle$
- By recurrence on n, we have proved that

  $$\forall n : X^{2n} = \langle X_1^{2n},\ X_2^{2n},\ X_3^{2n},\ X_4^{2n}\rangle = \langle\mathbb{Z},\ \{1,\ldots,n\},\ \{1,\ldots,n\},\ \emptyset\rangle$$

Solving the equations iteratively ... (Cont'd)

- Passing to the limit, we get the desired strongest invariant

$$\begin{aligned} I &= \langle I_1,\ I_2,\ I_3,\ I_4\rangle && \text{≀invariantʃ} \\ &= \lim_{n\to+\infty} X^{2n} \\ &= \langle\mathbb{Z},\ \{n\in\mathbb{Z}\mid n>0\},\ \{n\in\mathbb{Z}\mid n>0\},\ \emptyset\rangle \end{aligned}$$

f is increasing

- A fundamental property of the invariants equations $X = f(X)$ is that f is *increasing*.
- This means that if $X \dot{\subseteq} Y$ then $f(X) \dot{\subseteq} f(Y)$ where $\langle X_1, \ldots, X_n\rangle \dot{\subseteq} \langle Y_1, \ldots, Y_n\rangle$ if and only if $\forall i \in [1, n] : X_i \subseteq Y_i$.
- The intuition is that if more states can be reached at some program point then more states will be reachable at next program point.
- It follows that the iterates form an ascending chain meaning $X^0 \dot{\subseteq} X^1 \dot{\subseteq} \ldots \dot{\subseteq} X^n \dot{\subseteq} X^{n+1} \dot{\subseteq} \ldots \dot{\subseteq} \lim_{n\to+\infty} X^n = \mathbf{lfp}\, f$.

Machine Invariants

Machine Integers

- No computer can represent any, arbitrary large, integer. In practice integer variables like `x` take their values in an interval [`min_int`, `max_int`] where `min_int` $< 0 <$ `max_int` are machine dependant[2].
- It follows that we have to decide what happens in case of overflow when evaluating expression (`x` + 1).
- We will assume that execution immediately stops in case of integer overflow[3].

[2] e.g. in two's complement representation on 64 bits, we have generally have `min_int` $= -2147483648$ and `max_int` $= 2147483647$.

[3] Which is a rather simplifying hypothesis since most computers will go on providing a result modulo `max_int` so that e.g. `max_int` $+ 1 =$ `min_int` in two's complement representation.

Machine states and execution traces

Hence the set of program states $\mathcal{S} \triangleq \{1, 2, 3, 4\} \times [\texttt{min\_int}, \texttt{max\_int}]$ is now finite and the execution traces are now finite of the form

$$\{\langle ^1, z\rangle\langle ^2, 1\rangle \ldots \langle ^2, i\rangle\langle ^3, i\rangle\langle ^2, i+1\rangle \ldots \langle ^3, \texttt{max\_int}\rangle \mid z \in [\texttt{min\_int}, \texttt{max\_int}]\} \ .$$

Machine Invariant Equations

$P \triangleq {}^1\texttt{x := 1 ; while } {}^2\mathfrak{true}\texttt{ do } {}^3\texttt{x := (x + 1); od}^4.$

It follows that the machine invariant satisfies the following equations

$$\left\{\begin{array}{lcl} X_1 & = & [\texttt{min\_int}, \texttt{max\_int}] \\ X_2 & = & \{1\} \cup \{x+1 \in [\texttt{min\_int}, \texttt{max\_int}] \mid x \in X_3\} \\ X_3 & = & X_2 \cap \{x \in [\texttt{min\_int}, \texttt{max\_int}] \mid \mathfrak{true}\} \\ X_4 & = & X_2 \cap \{x \in [\texttt{min\_int}, \texttt{max\_int}] \mid \mathfrak{false}\} \end{array}\right. \tag{3.2}$$

Convergence

- Now the convergence of the iterations is guaranteed but is so slow that it cannot be of any practical use, but for programs with very few program variables.
- Moreover, mathematical sets of integers can be arbitrarily complex hence very expensive to represent in computer memory which is likely to produce memory overflows after lengthy computations, a flaw of all program verification methods based upon the exhaustive enumeration of all possible cases.

Interval Abstraction

Interval Abstraction

- A further abstraction must be used to solve the machine invariant computer representation problem.
- We will use intervals $[l, h] \triangleq \{x \in \mathbb{Z} \mid l \leqslant x \leqslant h\}$ with the convention that $[l, h] = \emptyset$ whenever $h < l$.
- In doing so we perform an approximation of a non-empty set $X \subseteq [\texttt{min\_int}, \texttt{max\_int}]$ by the interval $[\min X, \max X]$.
- This approximation is sound in that whenever the value of variable `x` belongs to a set X_i whenever execution reaches program point ^{i}, it definitely also belongs to the set $[\min X_i, \max X_i]$.
- This information is certainly correct but just less precise.

Interval Invariance Equations

$$P \triangleq {}^{1}\mathtt{x := 1\ ;\ while}\ {}^{2}\mathfrak{true}\ \mathtt{do}\ {}^{3}\mathtt{x := (x + 1);\ od}^{4}.$$

The interval invariance equations are now

$$\begin{cases} X_1 = [\mathtt{min\_int},\ \mathtt{max\_int}]\} \\ X_2 = [1,\ 1] \sqcup (\!|\ X_3 = \emptyset \ ?\ \emptyset \ ¿\ \mathtt{let}\ [a,\ b] = X_3\ \mathtt{in} \\ \qquad\qquad [\min(a+1,\ \mathtt{max\_int}), \min(b+1, \mathtt{max\_int})]\ |\!)\} \\ X_3 = X_2 \sqcap [\mathtt{min\_int},\ \mathtt{max\_int}] \\ X_4 = X_2 \sqcap \emptyset \end{cases}$$

Interval Operations

- the interval join is $\emptyset \sqcup \emptyset \triangleq \emptyset$, $\emptyset \sqcup [l,\ h] \triangleq [l,\ h] \sqcup \emptyset \triangleq [l,\ h]$, and

$$[a,\ b] \sqcup [c,\ d] \ \triangleq\ [\min(a, c),\ \max(b, d)]$$

- and the interval meet is $\emptyset \sqcap \emptyset \triangleq \emptyset$, $\emptyset \sqcap [l,\ h] \triangleq [l,\ h] \sqcap \emptyset \triangleq \emptyset$, and

$$\begin{array}{lcll} [a,\ b] \sqcap [c,\ d] & \triangleq & [\max(a, c),\ \min(b, d)] & \text{when } b \geqslant c \wedge d \geqslant a \\ [a,\ b] \sqcap [c,\ d] & \triangleq & \emptyset & \text{when } b < c \ \vee d < a \end{array}$$

Over-approximation

- The interval equations over-estimate the machine invariant in than they will provide in general more states that possible in actual program executions.
- For example the set $\{1, 2, 5\}$ will be overapproximated by $[1,\ 5]$ which introduces the spurious values 3 and 4.
- Notice that overapproximation preserve invariance. For example if the values of variable x are always greater than one at some program point then they are certainly positive (although the value 0 is spurious).

Example of incorrect approximations

For x $\in \{1, 2, 5\}$

- Underapproximations (such as x are always greater than 10) would be incorrect.
- Similarly, incomparable approximations (such as x is negative) are also unsound.

In particular the interval join $\sqcup$ overapproximates the interval union $\cup$ and the interval meet $\sqcap$ overapproximates the interval intersection $\cap$.

An Interval Abstract Interpreter

Objective

- We now briefly sketch the design and functional encoding in OCaml of the interval abstract interpreter.
- Such an interval abstract interpreter reads any program, builds the interval invariance equations, and then solve them.
- For simplicity, we concentrate on the second part and will provide encodings of the interval invariance equations manually.

The Interval Abstract Domain

- We first encode the interval abstract domain, implementing a computer representation of abstract interval propeties with a type `interval` (where `EMPTY` encodes the empty set $\emptyset$). In OCaml, we have `max_int` $= 1073741823$ and `min_int` $= -1073741824$[4].
- We also encode the basic interval operations $\sqsubseteq$ (`less`, interval inclusion), $\sqcup$ (interval `join`), $\sqcap$ (interval `meet`), interval printing (`print`) and interval incrementation (`add1`).
- Of course many more interval operations are needed to handle a full language, but we aim at extreme simplicity.

[4]One of the 64 bits is used for garbage collection.

```
(* interval.ml, interval abstract domain *)
type interval = EMPTY | INT of (int * int);;
let less x y = match x,y with
| EMPTY, _ -> true
| _, EMPTY -> false
| INT (a,b), INT (c,d) -> (c<=a)&&(b<=d);;
let join x y = match x,y with
| EMPTY, _ -> y
| _, EMPTY -> x
| INT (a,b), INT (c,d) -> INT (min a c,max b d);;
let meet x y = match x,y with
| EMPTY, _ -> EMPTY
| _, EMPTY -> EMPTY
| INT (a,b), INT (c,d) ->
    if (b<c) or (d<a) then EMPTY
    else INT (max a c,min b d);;
let add1 x = match x with
| EMPTY -> EMPTY
| INT (a,b) ->
    (INT ((if a<max_int then a+1 else max_int),
          (if b<max_int then b+1 else max_int)));;
let print x = match x  with
| EMPTY -> print_string "_|_"
| INT (a,b) -> print_string "(";  print_int a;
     print_string ","; print_int b;  print_string ")";;
```

Abstract Environments

- For programs with more than one variable, we would have to encode an abstract environment assigning intervals to program variables.
- Writing $X = \{\mathtt{x}_1 \leftarrow v_1, \ldots, \mathtt{x}_n \leftarrow v_n\}$ for the function X mapping $\mathtt{x}_i$ to v_i such that $X(\mathtt{x}_i) = v_i$, $i = 1, \ldots, n$, the interval invariance equations would be

$$\left\{\begin{array}{lcl} X_1 & = & \{\mathtt{x} \leftarrow [\mathtt{min\_int},\ \mathtt{max\_int}]\} \\ X_2 & = & \{\mathtt{x} \leftarrow [1,\ 1] \sqcup (\!|\ X_3(\mathtt{x}) = \emptyset \ ?\ \emptyset \ ⸽\ \mathtt{let}\ [a,\ b] = X_3(\mathtt{x})\ \mathtt{in} \\ & & \qquad [\min(a+1,\ \mathtt{max\_int}), \min(b+1, \mathtt{max\_int})]\ |\!)\} \\ X_3 & = & X_2 \mathbin{\dot\sqcap} \{\mathtt{x} \leftarrow [\mathtt{min\_int},\ \mathtt{max\_int}]\} \\ X_4 & = & X_2 \mathbin{\dot\sqcap} \{\mathtt{x} \leftarrow \emptyset\} \end{array}\right.$$

where the abstract operations are extended pointwise such as $\{\mathtt{x}_1 \leftarrow v_1, \ldots, \mathtt{x}_n \leftarrow v_n\} \mathbin{\dot\sqcap} \{\mathtt{x}_1 \leftarrow v'_1, \ldots, \mathtt{x}_n \leftarrow v'_n\} \triangleq \{\mathtt{x}_1 \leftarrow v_1 \sqcap v'_1, \ldots, \mathtt{x}_n \leftarrow v_n \sqcap v'_n\}$.

- Since our example has only one variable, this boils down to using the interval abstract domain (and leaving implicit the variable name x).

Abstract Invariants

- Then we have to encode an abstract domain for representing abstract invariants$\langle X^1, X^2, X^3, X^4\rangle$ which attach to each program point ^{i} an abstract local invariant X^i which holds whenever controls reaches program point ^{i}.
- Each abstract local invariant X^i is represented by an abstract environment (abstract intervals in our simplified case).
- The encoding is very simple as a 4-tuple specifying the value of program variable x at each program point (^{1}, ^{2}, ^{3}, ^{4}).

Abstract Invariants (Cont'd)

We essentially have to represent the logical structure, which boils down to

- the partial order $\dot\sqsubseteq$ (pless), encoding abstract implication ($\subseteq$ in set theory and $\Rightarrow$ in logic);
- $\dot\sqsupseteq$ (pgreater), the abstract inverse implication ($\supseteq$ in set theory and $\Leftarrow$ in logic);
- the pointwise infimum $(\emptyset)^4$ (pbot), the abstract encoding of $\mathfrak{false}$,
- the pointwise meet (for later use), and
- the printing of local abstract invariants attached to program points (pprint).

```
(* invariant.ml, interval invariant abstract domain *)
open Interval
type invariant = interval*interval*interval*interval;;
let cless (x1,x2,x3,x4) (x'1,x'2,x'3,x'4) =
   (less x1 x'1, less x2 x'2, less x3 x'3, less x4 x'4);;
let pless x x' =
   let (b1, b2, b3, b4) = cless x x' in
      b1 && b2 && b3 && b4;;
let pgreater x x' = pless x' x;;
let pbot = (EMPTY, EMPTY, EMPTY, EMPTY);;
let pmeet (x1,x2,x3,x4) (x'1,x'2,x'3,x'4) =
   (meet x1 x'1, meet x2 x'2, meet x3 x'3, meet x4 x'4);;
let pprint (x1,x2,x3,x4) =
   print_string " 1:";print x1; print_string " 2:";
   print x2; print_string " 3:";print x3;
   print_string " 4:";print x4; print_newline ();;
```

The Iterator

- Next the iterator module implements the iterative computation of the least solution of the invariance equations (lfp[5]).
- It is parameterized by the order (leq), the starting point (a) and the abstract transformer (f) so as to compute a, f(a), $\texttt{f}^2$(a), …, $\texttt{f}^n$(a), …, until reaching the limit $\texttt{f}^\ell$(a) such that $\texttt{f}(\texttt{f}^\ell(\texttt{a})) \sqsubseteq \texttt{f}^\ell(\texttt{a})$.
- Of course, convergence may not be guaranteed in which case lfp does not terminate (or terminates with a runtime error, e.g. out of memory).

[5] least fixpoint.

```
(* iterator.ml, iteration of f from a to x >= f(x) *)
let lfp leq a f =
  let rec iterate x =
    let y = f x in
      if leq y x then x
      else iterate y
  in iterate a;;
```

Jacobi versus chaotic iteration strategies

Of course the Jacobi iteration strategy

$$\left\{\begin{array}{lcl} X_i^{k+1} & = & f_i(X_1^k, \ldots, X_4^k) \qquad k = 1, 2, 3, \ldots \\ i = 1, \ldots, 4 & & \end{array}\right.$$

is simplistic, more elaborate ones would use e.g. a working list

Abstract Invariant Equations X=f(X)

Then we encode the abstract reachable state transformer $f(X) = f(\langle X_1, \ldots, X_4\rangle)$ using the environment abstract domain (the intervals in our simplified case).

```
(* transformerUnbounded.ml, abstract transformer *)
open Interval
open Invariant
let f (x1,x2,x3,x4) =
  (INT (min_int,max_int),
   join (INT (1,1)) (add1 x3),
   meet x2 (INT (min_int,max_int)),
   meet x2 EMPTY);;
```

encoding:

$$\left\{\begin{array}{lcl} X_1 & = & [\texttt{min\_int},\ \texttt{max\_int}]\} \\ X_2 & = & [1,\ 1] \sqcup (\!|\ X_3 = \emptyset\ ?\ \emptyset\ ⦂\ \texttt{let}\,[a,\ b] = X_3\ \texttt{in} \\ & & \quad [\min(a+1,\ \texttt{max\_int}), \min(b+1, \texttt{max\_int})]\ |\!)\} \\ X_3 & = & X_2 \sqcap [\texttt{min\_int},\ \texttt{max\_int}] \\ X_4 & = & X_2 \sqcap \emptyset \end{array}\right.$$

The Abstract Interpreter

The abstract interpreter performs the iterative abstract reachability fixpoint computation and prints the least fixpoint result.

```
(* reachability interval analysis *)
open Invariant
open TransformerUnbounded
open Iterator
let analyzer () = pprint (lfp pless pbot f);;
analyzer ();;
```

Infinitary Iteration

Iterative Resolution of the Interval Equations

Because the abstract domains are finite, the static analysis will always terminate. In our case, after more that 40mn of computation[6], we get

```
% ocamlc interval.ml invariant.ml transformeUnbounded.ml iterator.ml \
? reachability_unbounded.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:_|_
2977.460u 9.632s 50:43.46 98.1% 0+0k 0+0io 0pf+0w
%
```

[6]On a MacBook Pro with Intel Core 2 Duo at 2.6 GHz.

A look at the iterates...

The Jacobi iterates are as follows

```
% ocamlc interval.ml invariant.ml transformerUnbounded.ml \
? iteratorPartialUnboundedTrace.ml reachability_unbounded_trace.ml
% time ./a.out
 1:_|_ 2:_|_ 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,2) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,2) 3:(1,2) 4:_|_
 1:(-1073741824,1073741823) 2:(1,3) 3:(1,2) 4:_|_
 1:(-1073741824,1073741823) 2:(1,3) 3:(1,3) 4:_|_
 ...
 ...
 1:(-1073741824,1073741823) 2:(1,1073741821) 3:(1,1073741820) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741821) 3:(1,1073741821) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741822) 3:(1,1073741821) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741822) 3:(1,1073741822) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741822) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:_|_
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:_|_
3115.012u 7.706s 52:49.34 98.5% 0+0k 0+0io 0pf+0w
%
```

On the Convergence Criterion

- Notice that the abstract invariance equations $X = f(X)$ are increasing, if $X \mathrel{\dot{\sqsubseteq}} Y$ then $f(X) \mathrel{\dot{\sqsubseteq}} f(Y)$.
- The intuition is that the interval of possible value of a variable is larger at a program point, it should be also larger at the next program point.
- It follows that the iterates $X^0 \mathrel{\dot{\sqsubseteq}} \ldots \mathrel{\dot{\sqsubseteq}} X^n \mathrel{\dot{\sqsubseteq}} \ldots \mathrel{\dot{\sqsubseteq}} \lim_{n\to+\infty} X^n$ are increasing.
- Since the abstract interpreter stops iterating when reaching of postfixpoint $f(\lim_{n\to+\infty} X^n) \mathrel{\dot{\sqsubseteq}} \lim_{n\to+\infty} X^n$, the limit satisfies $f(\lim_{n\to+\infty} X^n) = \lim_{n\to+\infty} X^n$ by antisymmetry.

On Slow Convergence !

Of course the convergence is extremely slow and in practice must be accelerated.

Convergence Acceleration

Objective

- When convergence requires infinitely many steps or is very slow, it may not be possible, due to undecidability or high complexity, to exactly calculate the least solution to the abstract system of equations.(*)
- The only sound solution is then to have overapproximations of the desired result.
- We have already exploited the overapproximation idea when replacing sets of integer values in the invariant equations by interval of values.
- We now exploit the approximation idea a second time now while computing the solution of the invariance equations.
- The possibility of computing sound but approximate solutions to the invariant equations leads to powerful sound and fast static program analysis methods.

(*) Of course direct solutions do sometimes exist e.g. linear equations on regular languages

Convergence Acceleration by Widening

Convergence Acceleration by Widening

- The intuition for convergence acceleration is to speed up the increasing iteration $X^0 = \bot, \ldots, X^{n+1} = f(X^n), \ldots, \lim_{n\to+\infty} X^n$ so as to reach an overapproximation $\hat{A}$ of the least solution $\lim_{n\to+\infty} X^n$ of the fixpoint equation $X = f(X)$[7].
- Convergence acceleration means that X^{n+1} will be a function of X^n and $f(X^n)$[8] and so $X^{n+1} = X^n \nabla f(X^n)$ where ∇ is called a *widening*[9].

[7]The justification is again by Tarski theorem since $f(\hat{A}) \sqsubseteq \hat{A}$ implies $\mathbf{lfp}\, f \sqsubseteq \hat{A}$.

[8]and more generally X^{n+1} could depend on the sequence of previous iterates $X^0, f(X^0), \ldots, X^n, f(X^n)$

[9]We use a binary operator notation rather than a functional notation because of the analogy between widenings ∇ and joins $\vee$, $\sqcup$, etc

Soundness

- For soundness, the widening must perform over-approximations, that is $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$.

Convergence enforcement

- For convergence, the widening must ensure termination with an overapproximation of the desired solution.

Example: Interval Widening

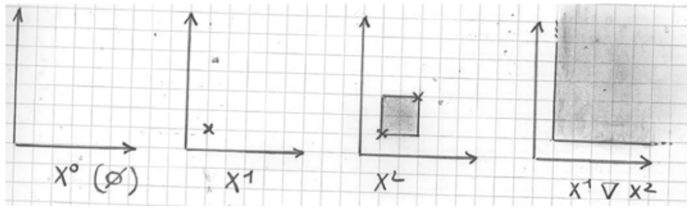
For example, a widening for intervals could be

$$\begin{aligned} \emptyset \mathbin{\nabla} y &\triangleq y \\ x \mathbin{\nabla} \emptyset &\triangleq x \\ [a,\ b] \mathbin{\nabla} [c,\ d] &\triangleq [\llparenthesis\, c < a\ ?\ -\infty : a \,\rrparenthesis,\ \llparenthesis\, d > b\ ?\ +\infty : b \,\rrparenthesis] \end{aligned}$$

- Recall than in $x \mathbin{\nabla} y$ the x is an iterate and y is the next iterate $f(x)$. So in $[a,\ b] \mathbin{\nabla} [c,\ d]$ if $c < a$ the next iterate decreases the lower limit of the interval so widening to $-\infty$ ensures this cannot happen infinitely often.
- Similarly, if $d > b$ then the next iterate increases the upper limit of the interval so widening to $+\infty$ ensures this cannot happen infinitely often. Moreover the widened interval is larger which ensures that we perform an overapproximation.

Example: Interval Widening (Cont'd)

The extrapolation of bounds to infinity is illustrated on the following iteration (for two variables).



Widenings are not increasing!

- Observe than the interval widening is not increasing. For example $[0,\ 1] \sqsubseteq [0,\ 2]$ but $[0,\ 1] \mathbin{\nabla} [0,\ 2] = [0,\ +\infty] \not\sqsubseteq [0,\ 2] = [0,\ 2] \mathbin{\nabla} [0,\ 2]$

- It can be shown that if the widening stops loosing information when a solution is found and is monotone then it cannot enforce termination

Encoding the interval widening

A functional encoding in of the widening in OCaml could be

```
(* intervalWidening.ml, interval widening *)
open Interval
let widen x y = match x,y with
| EMPTY, _ -> y
| _, EMPTY -> x
| INT (a,b), INT (c,d) ->
    let a' = if c<a then min_int else a in
      let b' = if d>b then max_int else b in
        INT (a',b');;
```

Environment Widening

If we had abstract environments to handle several variables, the widening would have to be applied individually for each of these variables.

Invariant widening

$P \triangleq {}^1\texttt{x := 1 ; while } {}^2\mathtt{true}\texttt{ do } {}^3\texttt{x := (x + 1); od}^4.$

We must also extend the widening to local invariants attached to program points. In our example, the widening is applied once around the loop at program point ^{2} as follows.

```
(* invariantWidening.ml, invariant widening *)
open IntervalWidening
let pwiden (x1,x2,x3,x4) (x'1,x'2,x'3,x'4) =
   (x'1,widen x2 x'2,x'3,x'4);;
```

Abstract Interpreter with Widening

The abstract interpreter now calls the iterator using the invariant widening.

```
(* reachability analysis with widening *)
open Invariant
open InvariantWidening
open TransformerUnbounded
open Iterator
let analyzer () =
  let fw x = pwiden x (f x) in
    pprint (lfp pless pbot fw);;
analyzer ();;
```

Static Analysis with Widening

The result is now almost instantaneous.

```
% ocamlc interval.ml intervalWidening.ml invariant.ml \
? invariantWidening.ml transformerUnbounded.ml iterator.ml \
? reachability_widening.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:_|_
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

Trace of the Iterations with Widening

The Jacobi iterates with widening are extremely fast as shown below.

```
% ocamlc interval.ml intervalWidening.ml invariant.ml \
? invariantWidening.ml transformerUnbounded.ml \
? iteratorTrace.ml reachability_widening_trace.ml
% time ./a.out
 1:_|_ 2:_|_ 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:_|_
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1073741823) 4:_|_
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

Imprecision of the Widening

Of course, the widening cannot, in general, provide the exact result! To see that, consider the bounded iteration

$$P \triangleq {}^{1}\texttt{x} := 1 \ ; \ \texttt{while} \ {}^{2}(\texttt{x} <= 100) \ \texttt{do} \ {}^{3}\texttt{x} := (\texttt{x} + 1); \ \texttt{od}^{4} \, .$$

so that the abstract interval equations become

$$\begin{cases} X_1 &= \{\texttt{x} \leftarrow [\texttt{min\_int}, \ \texttt{max\_int}]\} \\ X_2 &= \{\texttt{x} \leftarrow [1, \ 1] \sqcup (\!| \ X_3(\texttt{x}) = \emptyset \ \text{?} \ \emptyset \ \text{:} \ \texttt{let} \ [a, \ b] = X_3(\texttt{x}) \ \texttt{in} \\ & \qquad [\min(a+1, \ \texttt{max\_int}), \min(b+1, \texttt{max\_int})] \ |\!)\} \\ X_3 &= X_2 \ \dot{\sqcap} \ \{\texttt{x} \leftarrow [\texttt{min\_int}, \ 100]\} \\ X_4 &= X_2 \ \dot{\sqcap} \ \{\texttt{x} \leftarrow [101, \ \texttt{max\_int}]\} \end{cases}$$

1) Direct iteration (without widening)

A direct iteration

```
(* reachability interval analysis *)
open Invariant
open TransformerBounded
open Iterator
let analyzer () = pprint (lfp pless pbot f);;
analyzer ();;
```

yields

```
% ocamlc interval.ml invariant.ml transformerBounded.ml \
? iterator.ml reachability_bounded.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.001u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

in more details

```
% ocamlc interval.ml invariant.ml transformerBounded.ml \
? iteratorPartialBoundedTrace.ml reachability_bounded_trace.ml
% time ./a.out
 1:_|_ 2:_|_ 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,2) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,2) 3:(1,2) 4:_|_
 1:(-1073741824,1073741823) 2:(1,3) 3:(1,2) 4:_|_
 ...
 1:(-1073741824,1073741823) 2:(1,99) 3:(1,99) 4:_|_
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,99) 4:_|_
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:_|_
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:_|_
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.001u 0.001s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

Again convergence is guaranteed but slow.

II) Iteration with widening

```
(* reachability analysis with widening *)
open Invariant
open InvariantWidening
open TransformerBounded
open Iterator
let analyzer () =
  let fw x = pwiden x (f x) in
    pprint (lfp pless pbot fw);;
analyzer ();;
```

we rapidly get a strictly less precise result.

```
% ocamlc interval.ml intervalWidening.ml invariant.ml \
? invariantWidening.ml transformerBounded.ml iterator.ml \
? reachability_widening_bounded.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,100) 4:(101,1073741823)
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
% ocamlc interval.ml intervalWidening.ml invariant.ml \
? invariantWidening.ml transformerBounded.ml iteratorTrace.ml \
```

In more details the widening effect is not compensated by the test on loop exit.

```
? reachability_widening_bounded_trace.ml
% time ./a.out
 1:_|_ 2:_|_ 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,100) 4:(101,1073741823)
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,100) 4:(101,1073741823)
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

Convergence Acceleration by Narrowing

Intuition for Convergence Acceleration with Narrowing

- Because the upward iteration sequence with widening concerges to a post-fixpoint $\hat{A}$ of f such that **lfp** $f \sqsubseteq \hat{A} \wedge f(\hat{A}) \sqsubseteq \hat{A}$, we have, by recurrence and since f is increasing, that **lfp** $f \sqsubseteq f^n(\hat{A}) \sqsubseteq \hat{A}$.
- When $\hat{A}$ is not a fixpoint of f, any iterate in the sequence $Y^0 = \hat{A}, \ldots, Y^{n+1} = f(Y^n) = f^n(\hat{A})$ is an overapproximation of the unknown **lfp** f more precise than $\hat{A}$.
- However, this downward iteration $\langle Y^n, n \in \mathbb{N}\rangle$ might be infinite or converging slowly.
- It is therefore necessary to ensure its fast convergence. Convergence acceleration means that Y^{n+1} will be a function of Y^n and $f(Y^n)$[10] and so $Y^{n+1} = Y^n \mathbin{\Delta} f(Y^n)$ where Δ is called a *narrowing*[11].

[10] and more generally Y^{n+1} could depend on the sequence of previous iterates Y^0, $f(Y^0)$, …, Y^n, $f(Y^n)$, as was also the case for widening.

[11] We use a binary operator notation rather than a functional notation because of the analogy between narrowing Δ and meets $\wedge$, $\sqcap$, etc

Soundness

- For soundness, the narrowing must perform over-approximations, that is $y \sqsubseteq x \,\Delta\, y$, so as to stay <u>above</u> the unknown least fixpoint, which requires remaining above any fixpoint (which we have no way to distinguish from the least one) [12].

[12] By recurrence, if $X = f(X)$ is any fixpoint of f such that $X \sqsubseteq Y^n$ then $X = f(X) \sqsubseteq f(Y^n)$ since f is increasing so $X \sqsubseteq Y^n \sqsubseteq Y^n \,\Delta\, f(Y^n) = Y^{n+1}$ by the overapproximation hypothesis.

Convergence

- For convergence, the narrowing must ensure termination with a fixpoint.

Example: Interval Narrowing

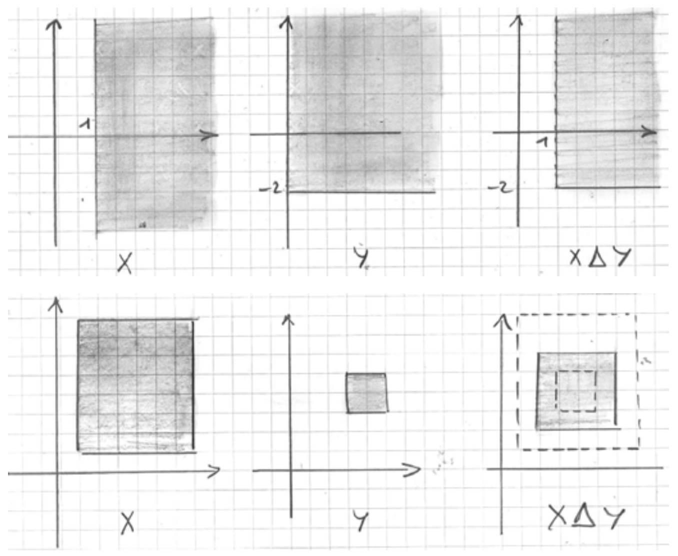
For example, a narrowing for intervals could be

$$\begin{array}{rcl}
\emptyset \,\Delta\, y & \triangleq & \emptyset \\
x \,\Delta\, \emptyset & \triangleq & \emptyset \\
[a,\ b] \,\Delta\, [c,\ d] & \triangleq & [(\!|\ a = -\infty \ ?\ c \ ⨾\ a\ |\!),\ (\!|\ b = +\infty \ ?\ d \ ⨾\ b\ |\!)]
\end{array}$$

Recall than in $x \,\Delta\, y$ the x is an iterate and y is the next iterate $f(x)$. So $[a,\ b] \,\Delta\, [c,\ d]$ will just eliminate the infinite bounds in $[a,\ b]$ and replace them by the bounds of the next iterate $[c,\ d]$.

So the narrowed interval is larger than $[c,\ d]$ that is $f(x)$ which ensures that we perform an overapproximation. Because only finitely many bounds can be infinite hence potentially removed, termination is guaranteed.

Example: Interval Narrowing (Cont'd)



Encoding the Interval Narrowing

A functional encoding in of the narrowing in OCaml could be

```
(* interval narrowing *)
open Interval
let narrow x y = match x,y with
| EMPTY, _ -> EMPTY
| _, EMPTY -> EMPTY
| INT (a,b), INT (c,d) ->
    let a' = if a=min_int then c else a in
      let b' = if b=max_int then d else b in
        INT (a',b');;
```

Invariant Narrowing

$$P \triangleq {}^{1}\mathtt{x} := 1 \;;\; \mathtt{while}\ {}^{2}\mathit{true}\ \mathtt{do}\ {}^{3}\mathtt{x} := (\mathtt{x} + 1);\ \mathtt{od}^{4}.$$

In our example, the narrowing is applied once around the loop at program point ^{2}, like the widening.

```
(* invariantNarrowing.ml, invariant narrowing *)
open IntervalNarrowing
let pnarrow (x1,x2,x3,x4) (x'1,x'2,x'3,x'4) =
   (x'1,narrow x2 x'2,x'3,x'4);;
```

Abstract Interpreter with Widening/Narrowing

The abstract interpreter now calls the iterator using the invariant widening until reaching a postfixpoint and then calls the iterator using the invariant narrowing until reaching a fixpoint.

```
(* reachability analysis with widening and narrowing *)
open Invariant
open InvariantWidening
open InvariantNarrowing
open TransformerBounded
open Iterator
let analyzer () =
  let fw x = pwiden x (f x) in
    let w = (lfp pless pbot fw) in
      let fn x = pnarrow x (f x) in
         pprint (lfp pgreater w fn);;
analyzer ();;
```

Example of convergence acceleration by widening/narrowing

The result is now almost instantaneous.

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iterator.ml \
? reachability_narrowing_bounded.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0%     0+0k 0+0io 0pf+0w
%
```

Details of the iteration with Narrowing/Widening

When compared to the Jacobi iterations, the chaotic iterates with widening and narrowing are extremely fast as shown below.

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iteratorTrace.ml \
? reachability_narrowing_bounded_trace.ml
% time ./a.out
 1:_|_ 2:_|_ 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,100) 4:(101,1073741823)
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,1073741823) 3:(1,100) 4:(101,1073741823)
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,1073741823)
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

On the (im)precision of the analysis...

Of course the narrowing cannot always recover all information lost by the widening, in particular because it is blocked by fixpoints jumped over by the widening.

Widening/Narrowing are not duals

	Iteration starts from	Iteration stabilizes
Widening ∇	below	above
Narrowing Δ	above	above
Dual widening $\widetilde{\nabla}$	above	below
Dual narrowing $\widetilde{\Delta}$	below	below

No dual widening $\widetilde{\nabla}$ has ever been found but trivial ones such as bounded execution (bounded model-checking), execution on a few cases (debugging), etc.

On actual abstract interpreters

- For simplicity, we have designed a specific abstract interpreter for a specific program.
- In practice, abstract interpreters are parameterized by the program they have to analyze, and by the abstraction which should be used for the analysis.
- Observe that the code defining the transformer could be directly generated from the program text and so we have a model of an abstract compiler.
- (An alternative would use a computer representation of the equations and an abstract interpreter would be used to evaluate the transformer by calls to the `interval` abstract domain). □

Verification

Verifier

- The abstract interpreter that we have designed is a sound *static analyzer*. Given a program it produces interval information always valid at runtime.
- We can turn it into a *verifier* checking an interval specification.
- The specification can be provided by the user or remain implicit (e.g. absence of runtime errors such as overflows).
- One kind of user specification is a type declaration, for example an interval declaration for integer variables like `var x` : 1..100;.
- Let us understand this declaration as: "only values between 1 and 100 can be assigned to `x`, otherwise execution stops" (with a runtime error).
- Observe that this does not mean that `x` always has a value betwwen 1 and 100 because it can be initialized with any integer value.[13].

[13]This interpretation of the interval declaration is that of the PASCAL programming language, see K. Jensen and N. Wirth: Pascal User Manual and Report, Second Edition, Springer, 1975.

Example of Interval Verification

For the following example

$$P' \triangleq \texttt{var x} : 1..100\ ;\ {}^{1}\texttt{x} := 1\ ;\ \texttt{while}\ {}^{2}(\texttt{x} <= 100)\ \texttt{do}\ {}^{3}\texttt{x} := (\texttt{x} + 1);\ \texttt{od}^{4}\,.$$

the abstract interval equations become

$$\begin{cases} X_1 &= \{\texttt{x} \leftarrow [\texttt{min\_int},\ \texttt{max\_int}]\} \\ X_2 &= \{\texttt{x} \leftarrow ([1,\ 1] \sqcup (\!| X_3(\texttt{x}) = \emptyset \ ? \ \emptyset \ ¿ \ \texttt{let}\,[a,\ b] = X_3(\texttt{x})\ \texttt{in} \\ & \qquad [\min(a+1,\ \texttt{max\_int}), \min(b+1, \texttt{max\_int})]\, |\!)) \sqcap [1,\ 100]\} \\ X_3 &= X_2 \,\dot\sqcap\, \{\texttt{x} \leftarrow [\texttt{min\_int},\ 100]\} \\ X_4 &= X_2 \,\dot\sqcap\, \{\texttt{x} \leftarrow [101,\ \texttt{max\_int}]\} \end{cases}$$

since execution stops if and when a value outside $[1,\ 100]$ is going to be assigned to `x`.

Encoding the Declaration

This declaration is encoded in OCAML as follows

```
(* declaration.ml *)
open Interval
open Invariant
let d =
  (INT (min_int,max_int),
   INT (1,100),
   INT (min_int,max_int),
   INT (min_int,max_int));;
```

Encoding the Verification Phase

The verification of absence of errors checks that at any point during an execution without error up to some point in the computation will not have an error at the next execution step.

```
(* verifier.ml, interval invariant abstract domain *)
let pwarning (b1, b2, b3, b4) =
  let m = "Potential error at line " in
  if not b1 then print_string (m^"1\n");
  if not b2 then print_string (m^"2\n");
  if not b3 then print_string (m^"3\n");
  if not b4 then print_string (m^"4\n");;
let pverify leq f a d =
   let b = leq (f a) d in
      pwarning b;
```

Encoding the Verifier

The abstract interpreter performs the iterative abstract reachability fixpoint overapproximation with widening/narrowing and intersection with the declaration, then prints the least fixpoint result, and finally checks for errors.

```
(* reachability verification with widening and narrowing *)
open Invariant
open InvariantWidening
open InvariantNarrowing
open TransformerBounded
open Iterator
open Declaration
open Verifier
let verifier () =
  let fw x = (pmeet (pwiden x (f x)) d) in
    let w = (lfp pless pbot fw) in
       let fn x = pnarrow x (f x) in
          let a = (lfp pgreater w fn) in
             pprint a; pverify cless f a d;;
verifier ();;
```

Result of the Analysis

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iterator.ml declaration.ml \
? verifier.ml reachability_narrowing_declaration.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:_|_
Potential error at line 2
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

- Observe that the program execution always stops at program point ^{3} with an overflow outside the range [1, 100] so program point ^{4} is now unreachable (with an overapproximation we can prove the presence of dead code but not its absence).
- Notice that the error is signaled as potential (with an overapproximation we can prove the values to definitely be within given bounds but not to prove that execution ever assigns a given value to a variable). Here is a trace of the analysis.

Details of the Analysis

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iteratorTrace.ml declaration.ml\
? verifier.ml reachability_narrowing_declaration_trace.ml
% time ./a.out
 1:_|_ 2:_|_ 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:_|_ 4:_|_
 1:(-1073741824,1073741823) 2:(1,1) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,1) 4:_|_
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:_|_
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:_|_
 converged to lfp.
 1:(-1073741824,1073741823) 2:(1,100) 3:(1,100) 4:_|_
Potential error at line 2
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

Correcting the Declaration

```
(* declarationCorrect.ml *)
open Interval
open Invariant
let d =
  (INT (min_int,max_int),
   INT (1,101),
   INT (min_int,max_int),
   INT (min_int,max_int));;
```

yields no error, the verification is completed.

```
% ocamlc interval.ml intervalWidening.ml intervalNarrowing.ml \
? invariant.ml invariantWidening.ml invariantNarrowing.ml \
? transformerBounded.ml iterator.ml declarationCorrect.ml \
? verifier.ml reachability_narrowing_declaration_correct.ml
% time ./a.out
 1:(-1073741824,1073741823) 2:(1,101) 3:(1,100) 4:(101,101)
0.000u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
%
```

When to do the verification?

Notice that in general the verification cannot be done during the analysis since a widening may cause an overapproximation potentially raising a potential error while the narrowing may refine the analysis well enough to that this potential error disappears.

Conclusion & references

Conclusion

- The presentation relied purely on intuition, can be made formal
- The abstraction ideas can scale up with enough precision, e.g.
  - ASTRÉE:
    - http://www.astree.ens.fr/
    - http://www.absint.de/astree/

Bibliography

The very first report on static analysis in infinite abstract domains not satisfying the ACC with widening/narrowing (Cousot and Cousot, 1975) was published in (Cousot and Cousot, 1976). The most cited reference is (Cousot and Cousot, 1977a). It is extended in (Cousot and Cousot, 1977b; Cousot, 1978) to handle procedures, see also (Bourdoncle, 1993). (Cousot, 1978) contains a presentation of reachability analysis using transition systems (i.e. language independent semantics and equational analyzers) later published in (Cousot, 1981).

An online course : http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/

References

Bourdoncle, F. (1993). Efficient chaotic iteration strategies with widenings, *in* D. Bjørner, M. Broy and I. Pottosin (eds), *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, Akademgorodok, Novosibirsk, Lecture Notes in Computer Science 735, Springer, Berlin, pp. 128–141.

Cousot, P. (1978). *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*, Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble.

Cousot, P. (1981). Semantic foundations of program analysis, invited chapter, *in* S. Muchnick and N. Jones (eds), *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, chapter 10, pp. 303–342.

Cousot, P. and Cousot, R. (1975). Static verification of dynamic type properties of variables, *Research report R.R. 25*, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble. 18 p.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs, *Proceedings of the Second International Symposium on Programming*, Dunod, Paris, Paris, pp. 106–130.

Cousot, P. and Cousot, R. (1977a). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, Los Angeles, pp. 238–252.

Cousot, P. and Cousot, R. (1977b). Static determination of dynamic properties of recursive procedures, *in* E. Neuhold (ed.), *IFIP Conference on Formal Description of Programming Concepts, St-Andrews, N.B.*, North-Holland Pub. Co., Amsterdam, pp. 237–277.

Cousot, P. and Cousot, R. (1992). Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper, *in* M. Bruynooghe and M. Wirsing (eds), *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming, PLILP '92*, Leuven, 26–28 August 1992, Lecture Notes in Computer Science 631, Springer, Berlin, pp. 269–295.

Next time ?

- More formal on foundations ?, or
- More on the notion of abstraction ?, or
- More practical on applications (like ASTRÉE)

Your choice ?

⇒ a little on the notion of abstraction

⇒ a little on applications (like ASTRÉE)

⇒ nothing formal on foundations!