Lab 6
Abstract Interpretation: Numerical Abstract Domains

Objective

- Write an abstract interpreter for the Mu language in Python.
- Implement classical finite abstract domains, and some infinite ones.

This lab is adapted from a lab by Pierre Roux, in the particular setting of Mu and PYTHON/ANTLR4.

No code is provided in the git. Pull for some examples.

Milestones for the 3 sessions

- Session 1: Operational code infrastructure, obtained by adapting code from previous lab, Makefile, and basic command line. Exercise 1 and 2 (augmenting the grammar and concrete evaluator with random expressions and assert statements). Adequating test files for the sign abstract domain. Starting the implementation of the sign abstract domain, in parallel with the abstract evaluator (Exercise 3).
- Session 2: Finishing the abstract evaluator and the sign abstract (Exercise 3), implementing the constant abstract domain (Exercise 4). Basic tests. Adapt the command line.
- Session 3: Interval lattice (Exercise 5 and 6), and automatic test infrastructure. Bonuses if everything else is working.

Your work is due on Tomuss before Dec. 16th at 23:59:59, STRICT. Instructions (infra, tests, documentation) are like all other labs.

6.1 Abstract Analyser for Mu programs

You are on your own. Based on what we did before, you have to implement an abstract interpreter for the Mu language. We give you some tests cases, but **they are not sufficient at all**.

EXERCISE #1 ▶ **Language extension**
Extend the grammar with random expressions and assert statements; extend your mu concrete evaluator from Lab 3 and test.

$$\begin{array}{lcll} expr & := & \ldots & | \quad \text{rand}(e1, e2) \\ stmt & := & \ldots & | \quad \text{assert}(b) \end{array}$$

The expression rand(n, m) should evaluate to a random (int) number in the interval $[n, m]$.

EXERCISE #2 ▶ **Generic Static Analyser**
Implement a static analyser, test infrastructure for the *sign* abstract domain (Section 6.2.1). The abstract interpretor should be a implemented as a **visitor** in the spirit of what you already did in Lab 3 (Mu evaluator). Try to be as generic as you can in your analyser since you will have to change abstract domains from the command line.

You will have to implement the following abstract domains: signs, constants, intervals (in bonus, polyhedra or the congruence abstract domain - see exercise sheet). For assert, the analyser should print `verified` or `unable to verify` if it is not capable of proving the assertion (see Section 6.4).

6.2 Finite height Abstract Domains

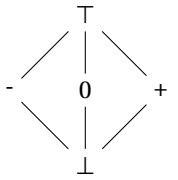
A cheat sheet about abstract domains can be found at the address:

`http://perso.ens-lyon.fr/pierre.roux/vas_2013_2014/rappels_domaines_abstraits.pdf`

(talk to your TA if you need help in reading the french there).

6.2.1 Signs

This domain makes it possible to find variables which are strictly positive or strictly negative, or zero, hence allowing to guarantee the correctness of more divisions.

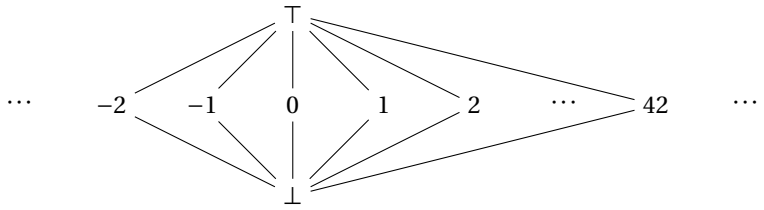


$$\begin{aligned}\gamma(\top) &= \mathbb{Z}\\ \gamma(+) &= \{n \in \mathbb{Z} \mid n > 0\}\\ \gamma(-) &= \{n \in \mathbb{Z} \mid n < 0\}\\ \gamma(0) &= \{0\}\\ \gamma(\bot) &= \emptyset\end{aligned}$$

EXERCISE #3 ▶ **Signs**
Implement this domain. Providing the option `--domain sign` should enable this domain.

6.2.2 Kildall (Constants)

This domain makes it possible to find variables which are constants at a certain point in the program. It can also be used to simplify programs in a compiler.

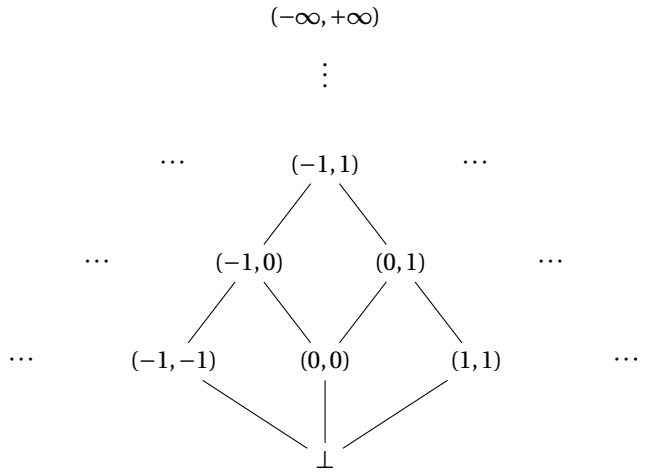


$$\begin{aligned}\gamma(\top) &= \mathbb{Z}\\ \gamma(n) &= \{n\}\\ \gamma(\bot) &= \emptyset\end{aligned}$$

EXERCISE #4 ▶ **Kildall**
Implement this domain. Providing the option `--domain constant` should enable this domain.

6.3 Infinite Height: intervals

In this section, we wish to implement a domain of intervals, where variables are interpreted by the range of values they can take.
The lattice is $(\mathscr{D}^\sharp, \sqsubseteq^\sharp)$ with $\mathscr{D}^\sharp = \bot \cup \{(n_1, n_2) \in (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\}) \mid n_1 \leq n_2\}$.



$$\begin{aligned}\gamma(-\infty, +\infty) &=]\!]-\infty, +\infty[\![\\ \gamma(-\infty, n) &=]\!]-\infty, n]\!]\\ \gamma(n, +\infty) &= [\![n, +\infty[\![\\ \gamma(n_1, n_2) &= [\![n_1, n_2]\!]\\ \gamma(\bot) &= \emptyset\end{aligned}$$

Reminder: a widening operator can be used to accelerate the convergence of the fixpoint calculation. The idea is to extrapolate in the computation, so that we reach a result without going upwards ad infinitum in a lattice

of unbounded height:

$$x^\sharp \triangledown y^\sharp = \begin{cases} [\![a,b]\!] & \text{if } x^\sharp = [\![a,b]\!], y^\sharp = [\![c,d]\!], c \geq a, d \leq b \\ [\![a,+\infty[\![& \text{if } x^\sharp = [\![a,b]\!], y^\sharp = [\![c,d]\!], c \geq a, d > b \\ ]\!-\infty,b]\!] & \text{if } x^\sharp = [\![a,b]\!], y^\sharp = [\![c,d]\!], c < a, d \leq b \\ ]\!-\infty,+\infty[\![& \text{if } x^\sharp = [\![a,b]\!], y^\sharp = [\![c,d]\!], c < a, d > b \\ y^\sharp & \text{if } x^\sharp = \bot \\ x^\sharp & \text{if } y^\sharp = \bot \end{cases}$$

EXERCISE #5 ▶ **Intervals**
Implement this domain. Providing the option `--domain interval` should enable this domain. Firstly, implement without widening, then test, then implement widening. You will have to change your generic analyser to apply widening at loop heads. Test on well-chosen examples.

EXERCISE #6 ▶ **Descending sequence, widening delay**
On the following program:

```
i = 0; j = 0;
while (i < 10) {
  if (i <= 0) {
    j = 1;
    ++i;
  } else {
    ++i; } }
```

What interval does one get for variable `j`? First try to improve by using a descending sequence, then by a widening delay. Augment the command line of your tool:`--descend n`, `--delay m`.

6.4 Diagnostics

EXERCISE #7 ▶ **Printing logs**
For every print statement, make the analyzer print the abstract value of the expression given to the print statement if the `--verbose` option is given on the command line.

EXERCISE #8 ▶ **Assertions**
Augment the language with a construction `assert` that takes a boolean expression and prints **once, at the end of the analysis, in ascending line order**:

- "assert on line n: verified" if the abstract value is sufficient to prove the property
- "assert on line n: failed to verify" otherwise. If no `--verbose` or `--debug` (optional) options are given, no other output should be generated.

EXERCISE #9 ▶ **Division by zero**
Make your analyzer find potential divisions by zero, ie detect all cases when there is a risk of a division by zero.

EXERCISE #10 ▶ **Assume**
Add an `assume` statement that takes a boolean expression, so that your analyzer assumes the assertion is true after it in the flow.

6.5 Optional extensions

Only do this section if you have time. Exercises below are in ascending order of difficulty. Documentation and appropriate test cases are always required.

EXERCISE #11 ▶ **Arrays, 1**
Add arrays to the Mu language. Describe their semantics and operations on the arrays on paper. Add support for checking out of bound access to arrays (negative and illegal accesses).

EXERCISE #12 ▶ **Arrays, 2**
Add basic support for abstract values in the arrays. The easiest way to do it is to join (ie. make the union) abstract values of all variables in the array to a single value ("array smashing" in the literature). Find appropriate tests cases.

EXERCISE #13 ▶ **Overflows**
Find a way to safely deal with overflows.

EXERCISE #14 ▶ **Polyhedra**
Add a new abstract domain, that uses a convex polyhedron as the type of an abstract value. You may use a third-party library if its license allows it.