Lab 7
Abstract Interpretation: Numerical Non Relational Abstract Domains

Objective

- Play with an implementation of a fixpoint static analyser (In OCAML).
- Implement classical finite abstract domains, and the interval abstract domain.
- Understand how the fixpoint computation is made.

This lab is adapted from material kindly provided by Pierre Roux. Download the archive and untar it. **Binoms are authorized only for non ocaml-native speakers.**

7.1 Play with `tiny`

In the archive of today, you will find `tiny`, a static analyzer based on abstract interpretation, written in OCaml. To compile it, run `make` at the root of the (uncompressed) archive. This should produce a binary `src/tiny` that you can test as follows:

```
src/tiny examples/ex01.tiny
```

This outputs the input source code.

What `tiny` does `tiny` computes the abstract interpretation of a given program, which means at every stage, compute the abstract value for each variable, as well as loop invariants. This information is used to perform various static analysis such as: no division by zero occur.

How does `tiny` work `tiny` is parametrized over the domain to consider. In `src/domains`, you will find a few domains that you will have to implement throughout this lab session. To use a specific domain, use:

```
src/tiny --domain <name> file
```

where domain is the name of your domain (`dummy`, `kildall`, `sign`, and `intervals`).

EXERCISE #1 ► **Discovering the analysis**
In the `bin` directory of the archive, there is a compiled version of all the domains. Run it through the examples and become familiar with the output of `tiny` via:

```
bin/tiny --domain <domain> file
```

(You can make the output more verbose via the option `-v 4`.)

For the examples `ex05.tiny`, `ex06.tiny`, `ex07.tiny` which analysis can prove that the codes are correct? Why?

`tiny` source code architecture Here are the main files/functions of the implementation: (`src` directory)

- `lexer.mll`, `parser.mly`, `location.ml`, `ast.ml` are the front-end. From an input in the `tiny` language (that ressembles Mu), it computes an abstract syntax described in `ast.mli`, for instance a statement is one of the following types [1]:

[1] The location module is used to track back the corresponding lines of code in the source.

```
type stm =
  | Asn of Location.t * Name.t * expr  (** v = expr; *)
  | Seq of Location.t * stm * stm  (** stm stm *)
  | Ite of Location.t * expr * stm * stm  (** if (expr > 0) \{ stm \} else \{ stm \} *)
  | While of Location.t * expr * stm  (** while (expr > 0) \{ stm \} *)
```

- `main.ml` is responsible for the command line options, and calling either the compiler (useless in this lab) or the analyser.
- `analyser.ml` will perform the fixpoint analysis on *non-relational domains* only (which is the scope of this Lab). This analysis is parametrised by an abstract domain module called `Dom`, from which the computation is performed (see ex 2).
- The abstract domain module `Dom` is constructed by `nonrelation.ml`, which provides the generic following functions (see `nonrelation.mli` for their signatures): `order`, `top`, `bottom`, `join` (**union**), `meet` (**intersection**), as well as the forward abstract semantic function `assignment`.
- This module `Dom` is himself constructed from domains implemented in the `src/domains` directory. **All you have to do is to implement the abstract functions for each new domain by mimicking `dummy.ml`.**

EXERCISE #2 ► **Code review**
Quickly open the source code:
- Find the analysis function in `main.ml`.
- In `analyse.ml`, the analysis is performed by a call to `post_stm` on the program AST. Observe the code of this function. What is the name of the function that performs the fixpoint computation ?
- In this last function, find the code that permits to stop this fixpoint computation. The `order` function is specific to each abstract domain, you will have to implement it.

7.2 Implementing new domains in `tiny`

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`

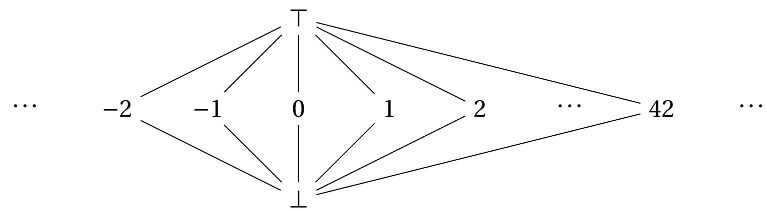
(talk to your TA if you need help in reading the french there).

For each domain, we provide a skeleton of source code (in `src/domain/`). You will have to define the type of abstract values, operation such as union and abstract transformers, as well as printing functions.

We recall that:
- Top is the biggest abstract value, Bottom the lowest.
- The order of two abstract values is given by their respective positions in the Hass diagram of the underlying issue.
- Join (union) computes an abstract value that gathers the information given by its two operands.
- Meet makes an intersection of the two operands.
- `sem_plus` is the abstract transfer function for the concrete '+', i.e. it gives the "effect of + in the abstract world".

Finite domains

EXERCISE #3 ► **Kildall**
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.



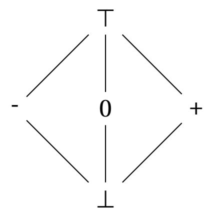
$$\gamma(\top) = \mathbb{Z}$$
$$\gamma(n) = \{n\}$$
$$\gamma(\bot) = \emptyset$$

Implement this domain in `src/domains/kildall.ml`. Check your implementation on examples. What happens using your domain on the example `examples/ex08.tiny`?

To solve this problem, if you have time, you can try to rely on the module `InfInt`, which is provided (see `src/domains/infInt.mli`), to handle this situation [2]

EXERCISE #4 ▸ **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.



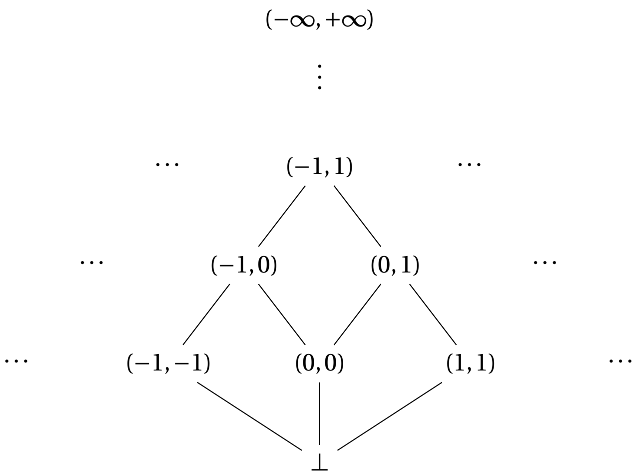
$$\begin{aligned}\gamma(\top) &= \mathbb{Z}\\ \gamma(+) &= \{n\in\mathbb{Z} \mid n>0\}\\ \gamma(-) &= \{n\in\mathbb{Z} \mid n<0\}\\ \gamma(0) &= \{0\}\\ \gamma(\bot) &= \emptyset\end{aligned}$$

Implement this domain in `src/domains/sign.ml`

Intervals and widening

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 $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ with $\mathcal{D}^\sharp = \bot \cup \{(n_1,n_2) \in (\mathbb{Z}\cup\{-\infty\}) \times (\mathbb{Z}\cup\{+\infty\}) \mid n_1 \le n_2\}$.



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

EXERCISE #5 ▸ **Intervals**

Implement this domain in `src/domains/intervals.ml`: for the moment, do not modify the definitions for `widening`, `sem_times`, `sem_div` (and forget about `backsem_times` and `backsem_div`).

Some hints :

- You can use the following type :

```
type t = Bot | Itv of int option * int option
```

  where `None` stands for $\pm\infty$ and `Some` n stands for the finite bound n[3].

- It will be useful to extend some functions acting on integers to $\mathbb{Z}\cup\{-\infty\}$ or $\mathbb{Z}\cup\{+\infty\}$. For instance, for "≤" :

```
(* Extending <= to Z U {-oo}. *)
let leq_minf x y = match x, y with
  | None, _ -> true  (* -oo <= y *)
```

[2] documentation : `src/doc/InfInt.html`).

[3] Reminder: the `option` type constructor, which is provided by OCaml, is defined as follows :
`type 'a option = None | Some of 'a.`

```
    | _, None -> false  (* x > -oo (x != -oo) *)
    | Some x, Some y -> x <= y

  (* Extending <= to Z U {+oo}. *)
  let leq_pinf x y = match x, y with
    | _, None -> true  (* x <= +oo *)
    | None, _ -> false  (* +oo > y (y != +oo) *)
    | Some x, Some y -> x <= y
```

- You can use the following function to enforce the invariant $n_1 \leq n_2$ when defining intervals :

```
let mk_itv o1 o2 = match o1, o2 with
  | None, _ | _, None -> Itv (o1, o2)
  | Some n1, Some n2 -> if n1 > n2 then Bot else Itv (o1, o2)
```

Test the domain on the following program (file `examples/ex09.tiny`) :

```
i=0;
while (i < 10) {
  ++i; }
```

then on the same program, after replacing 10 with 1 000 000. What can be observed? (use option `-v 2` if no difference shows up)

The problem is now that our domain is has infinite depth, so the fixpoint iteration to compute the interpretation of a while loop may take infinitely many steps: computing the exact interpretation becomes undecidable. In the next exercise, we will see a way to over-approximate the fixpoint through *widening*.

EXERCISE #6 ► **Widening in the domain of intervals**

- To address this problem, exploit *widening*, by implementing the following operator.

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

  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.
- Run the program using the new domain on the programs tested before, then on the following program (file `examples/ex10.tiny`) :

```
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 (`-d n` option). If it doesn't work, come up with a new widening operator that makes it possible to obtain the exact answer $[\![0,1]\!]$ (hint : this widening is called *delayed*).
- Additional question : what happens in the domain if the program contains expressions such as those that appear in `examples/ex08.tiny` ? This can in some cases be handled using the module `InfInt`, which is provided [4]

[4] (documentation : `src/doc/InfInt.html`).