Spring 2017
DD2457 Program Semantics and Analysis
Lab Assignment 2:
Abstract Interpretation

D. Gurov A. Lundblad

KTH Royal Institute of Technology

1 Introduction

In this lab assignment the abstract machine developed before is adapted to run with abstract values instead of with concrete ones, thus implementing the abstract interpretation technique for program analysis developed in Chapter 7 of the textbook, but in an operational semantics style.

The purpose of the lab assignment is, on one hand, to deepen the understanding of abstract interpretation by presenting its fundamental idea in an alternative setting, and on the other hand, to illustrate program analysis in a practical setting, showing that with simple means one can obtain an analysis tool that allows various program analyses and optimisations to be supported. In particular, the tool can discover points of potential division-by-zero, find redundant catch-blocks when no exception can be raised in the corresponding try-block, as well as provide support for optimisations such as the ones discussed in class.

The focus of this assignment should lie on analyses rather than on tool design, that is, on *using* the analysis tool rather than on developing it.

The assignment is carried out in teams of (at most) two.

2 Abstract Interpretation in Operational Style

Abstract interpretation is about executing programs with abstract values (corresponding to sets of concrete values, therefore also called *properties*) rather than with concrete ones, and using the result of the execution for various program analyses and program transformations. If the domain of abstract values is finite we obtain decidable analyses at the expense of precision. The usual goal is then to set up the domain so that the analysis is *safe*, in the sense that the loss of precision is always on the safe side of the analysis; what is considered to be the safe side depends on the application.

Abstract interpretation is presented in the course book in a denotational semantics style. However, the approach can equally well be applied in a operational semantics style. In this assignment, we adapt the already developed abstract machine for **While** with exceptions for the purpose of abstract interpretation in abstract machine operational semantics style. Note that due to the loss of precision in this case, we have to deal with *non-deterministic* execution.

We redefine the abstract machine for computing in the abstract domain of the *detection–of–signs* analysis considered in class, enriched with additional values to capture possible errors due to division by zero (see Section 4 below). Notice that we do not have to change the compilation (translation of **While** to **Code**), but only the rules of the operational semantics of the abstract machine so that the new abstract machine operates on abstract values in the evaluation stack and the store. We then use the new abstract machine to compute an abstract machine *configuration graph* that serves as the basis for all our analyses. The start property state is typically the one that maps every variable to Z, the most abstract non-error value.

Next, in the configuration graph generated by an abstract machine execution of the code for a given **While** statement, we group the normal configurations (i.e. configurations with normal, non-exceptional stores) "belonging" to the same control point. A *control point* can be thought of as a stepping point in an ordinary debugger; loosely speaking, there is a control point at every statement (an illustrative example is provided in Section 5.1). For each such group of configurations, we compute a property state *ps* by producing the least upper bound (*lub*) of the stores in the group.

Further, for transitions from a normal to an exceptional state, we flag the corresponding control point as a “possible exception raiser”. Then, consider the function $\mathcal{DA}$ defined in Table 7.1 in the book; to support certain useful analyses for control points at the entry of assignment statements $x := a$, we also compute $\mathcal{DA}[\![a]\!]\,(ps)$, and similarly, for control points at the entry of conditional statements **if** b **then** S_1 **else** S_2 and while loops **while** b **do** S we compute $\mathcal{DB}[\![b]\!]\,(ps)$, where ps is the property state computed for the given control point. Finally, we flag control points that cannot be reached in any execution as “unreachable”.

We shall refer to the information extracted in this way as the *base analysis*, consisting, for every control point, of a property state, the values of the “possible exception raiser” and “unreachable” flags, plus the value of the (possible) associated arithmetic or Boolean expression. We shall use pretty-printing to display the result of the analysis in the form of an *annotated program.*

3 Program Analysis and Transformation

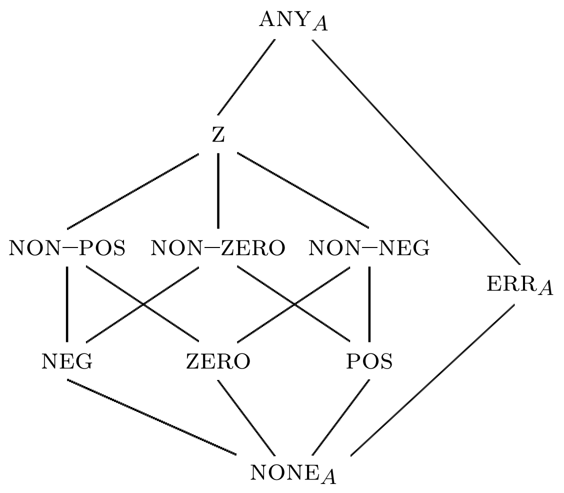
One interesting analysis in the presence of possible exceptions and a try-catch mechanism is a *safety analysis* for no-uncaught-exceptions: if all final configurations are normal (i.e. non-exceptional) we can deduce that in all terminating executions, all exceptions raised during program execution are eventually caught. If the result of the analysis suggests possible uncaught exceptions, we can inspect the abstract values of arithmetic expressions at the assignment statements to localise the possible source.

Our base analysis also supports various *program optimisations* in the style of Section 7.4 in the book. One interesting optimisation is to eliminate unnecessary catch statements: replace **try** S_1 **catch** S_2 by S_1 if no control point within S_1 is flagged as a possible exception raiser.

Since the base analysis collects the information at all control points, it supports program transformation in *any* context, including the branches of conditionals and the bodies of loops (cf. Exercises 7.29 and 7.30). For example, we can replace the conditional statement **if** b **then** S_1 **else** S_2 by S_1 whenever $\mathcal{DB}[\![b]\!]\,(ps) = \text{TT}$, where ps is the property state computed for the entry of the statement.

4 Detection of Signs with Errors

In the presence of possible division-by-zero errors we have to enrich suitably the abstract domains of the values **Sign** and **TT** of the detection-of-sign analysis presented in the book.



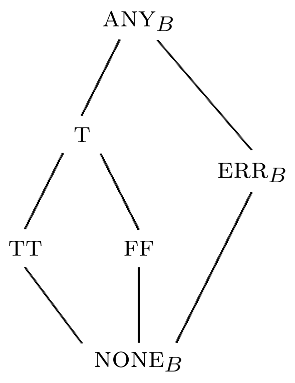


Figure 1: The lattices for **SignExc** and **TTExc**.

The abstract domain of integer values is modified to **SignExc** with ordering $\sqsubseteq_{SE}$, where ANY$_A$ is the most abstract value and Z is the most abstract non-error value (see Figure 1).

The abstraction function $\mathbf{abs}_{Z_\perp} : Z_\perp \to \mathbf{SignExc}$ is now defined as follows:

$$\mathbf{abs}_{Z_\perp}(z) \overset{\mathsf{def}}{=} \begin{cases} \text{NEG} & \text{if } z \in Z \text{ and } z < 0 \\ \text{ZERO} & \text{if } z \in Z \text{ and } z = 0 \\ \text{POS} & \text{if } z \in Z \text{ and } z > 0 \\ \text{ERR}_A & \text{if } z = \perp \end{cases}$$

To achieve maximal precision, the abstract version of the arithmetic operation $+_{SE}$ is defined so as to fulfill the condition:

$$v_1 +_{SE} v_2 = \bigsqcup_{SE} \{\mathbf{abs}_{Z_\perp}(z_1 + z_2) \mid \mathbf{abs}_{Z_\perp}(z_1) = v_1 \wedge \mathbf{abs}_{Z_\perp}(z_2) = v_2\}$$

and similarly for $-_{SE}$, $\star_{SE}$ and $/_{SE}$. For instance, POS $/_{SE}$ POS = NON−NEG (since we use integer division) and POS $/_{SE}$ ZERO = ERR$_A$.

With the same underlying idea we modify the abstract domain of Boolean values **TTExc** with ordering $\sqsubseteq_{TE}$ (see Figure 1), the abstract relations $=_{SE}$ and $\leq_{SE}$, and the abstract operations $\neg_{TE}$ and $\wedge_{TE}$.

Figures 2 and 3 below give an example of a base analysis.

```
1:   b := 3;
2:   p := 1;
3:   while ¬(b = 0) do
4:       p := p ⋆ a;
5:       b := b − 1
```

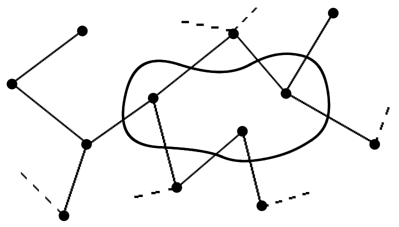


Figure 2: **While** program that computes a^b (left) and corresponding **AM** configuration graph with schematic group of configurations at control point 4 (right).

	$\{a\colon \mathrm{Z},\ b\colon \mathrm{POS},\ p\colon \mathrm{POS}\}$,	$p \star a\colon \mathrm{Z}$
	$\{a\colon \mathrm{Z},\ b\colon \mathrm{Z},\ \ p\colon \mathrm{Z}\ \}$,	$p \star a\colon \mathrm{Z}$
	...	...
lub:	$\{a\colon \mathrm{Z},\ b\colon \mathrm{Z},\ \ p\colon \mathrm{Z}\ \}$,	$p \star a\colon \mathrm{Z}$

```
...
while ¬(b = 0) do
    {a: Z, b: Z, p: Z},  p ⋆ a: Z
    p := p ⋆ a;
...
```

Figure 3: Stores and current value of right-hand-side of assignment for all configurations at control point 4 plus resulting property state (left), and analysis output (right).

5 Implementing the Analyser

Given a **While** program as input, the analyser should carry out the following steps:

Step 1 Compile the **While** program into **AM** code.

Step 2 In the abstract domains, compute all configurations reachable from the initial one.

Step 3 For each control point, perform the following steps:

(a) Compute the *lub* of all states of all non-exceptional configurations corresponding to the control point.

Class	Purpose
`SignExc`, `TTExc`	Enumerations for **SignExc** and **TTExc** respectively.
`SignExcLattice`, `TTExcLattice`	Implementation of lattice operations (lub and glb) for `SignExc` and `TTExc` respectively. (Implements the `Lattice` interface.)
`SignExcOps`	Implementation of arithmetic and boolean operations for `SignExc` and `TTExc` required for execution. (Implements the `Operations` interface.)

Table 1: Brief description of the available classes.

(b) If the control point refers to an assignment, compute the *lub* of the possible values of the arithmetic expression.

(c) If the control point refers to an **if**- or **while**-statement, compute the *lub* of the possible values of the Boolean expression.

Step 4 Pretty-print the **While** program, along with the above results and the values of the "Possible exception raiser!" and "Unreachable code!" flags, in the form of control-point annotations.

Step 5 Give some indication of whether the program can terminate normally and/or exceptionally, in the form of an annotation associated with the control point at the end of the program.

The source code of the classes available for this lab can be found in the folder `/info/DD2457/semant15/lab2`. Table 1 summarizes the purpose of each class. Here comes a detailed description of how to proceed at each step.

5.1 Step 1

To be able to translate the analysis of the **AM** configurations back to the original **While** program, you need to keep track of which statement each instruction corresponds to. One way of accomplishing this is to introduce a `controlPoint` field in the `Stm` class (containing a unique value for each statement), and a `stmControlPoint` field in the `Inst` class, containing the value of the control point of the originating statement. The `controlPoint` and `stmControlPoint` fields can be initialized during step 1. Figure 4 shows an example.

```
❶ a := 21;
❷ b := 70;
❸ while !(a = b) do
    ❹ if a <= b then
        ❺ b := b - a
      else
        ❻ a := a - b
❼
```

```
❶ PUSH-21          | ❹ LE
❶ STORE-a          | ❹ BRANCH
❷ PUSH-70          |
❷ STORE-b          |   ❺ FETCH-a
❸ LOOP             |   ❺ FETCH-b
                   |   ❺ SUB
  ❸ FETCH-b        |   ❺ STORE-b,
  ❸ FETCH-a        |
  ❸ EQ             |   ❻ FETCH-b
  ❸ NEG,           |   ❻ FETCH-a
                   |   ❻ SUB
  ❹ FETCH-b        |   ❻ STORE-a
  ❹ FETCH-a        | ❼
```

Figure 4: Control points for the **While** and **AM** version of GCD.

5.2 Step 2

To implement step 2, you need to change your abstract machine, which originally worked with concrete `Integer`s and `Boolean`s, to work with the abstract domains representing **SignExc** and **TTExc**. These domains have been prepared for you in the enumeration types `SignExc` and `TTExc`.

You will find implementations of the related operations $+_{SE}$, $-_{SE}$, $\star_{SE}$, $/_{SE}$, $=_{SE}$, $\leq_{SE}$, $\wedge_{TE}$, and $\neg_{TE}$ in the `SignExcOps` class. This class also contains implementations for $\mathbf{abs}_{Z_\perp}$ and $\mathbf{abs}_{T_\perp}$ (used when translating constants to abstract values) and methods to, for a given v, decide $\text{Z} \sqsubseteq_{SE} v$, $\text{TT} \sqsubseteq_{TE} v$, $\text{FF} \sqsubseteq_{TE} v$ and $\text{ERR}_B \sqsubseteq_{TE} v$ (called `possiblyInt`, `possiblyTrue`, `possiblyFalse` etc., used when implementing the step method for BRANCH and STORE).

As previously mentioned, the execution under abstract interpretations is *non–deterministic*; $\langle \text{BRANCH}(c_1, c_2) : c, \text{T} : e, ps \rangle$ for instance, has two possible successor configurations, so if your step method looks something like this:

```
public Configuration step(Configuration conf)
```

it needs to be rewritten into something like

```
public Set<Configuration> step(Configuration conf).
```

This new step method now implicitly defines the configuration graph (`step(c)` returns all neighbours of configuration `c`). The step method can thus be used

```
{x=Z} rhs:  POS
x := 7;
{x=POS}
try
    {x=POS} rhs:  Z
    x := (x - 7);
    {x=Z} rhs:  ANY_A (Possible exception raiser!)
    x := (7 / x);
    {x=Z} rhs:  Z
    x := (x + 7)
catch
    {x=Z} rhs:  Z
    x := (x - 7)
{x=Z} (normal termination)
```

Figure 5: Example pretty-printed output of the tool.

in a of the configuration graph, to collect all reachable configurations. Note that there is no need to store the edges (transitions) of the configuration graph, only the nodes (configurations).

As design choices affecting the precision versus the efficiency of the analysis, think of what traversal strategy to use (breadth-first or depth-first), and whether to split executions from $\textsc{any}_A$ into an execution from $\textsc{z}$ and another one with $\textsc{err}_A$ (and similarly for Boolean values).

5.3 Step 3

Step 3 requires computing *lub*'s. The *lub* of two abstract values can be computed with the `SignExcLattice.lub` method or the `TTExcLattice.lub` method.

5.4 Step 4

The implementation of step 4 should utilize the `PrettyPrinter` to pretty print the original **While** program, along with the results from the analysis as control-point annotations (i.e. above each statement). See Figure 5 for an example output.

5.5 Step 5

The final configurations (those with an empty instruction sequence) should provide you with enough information to carry out step 5.

6 Tasks

The present lab assignment consists of the following tasks:

1. Modify Table 4.1 in the book (Operational Semantics for **AM**) for execution in the abstract domain. Here are two examples describing how the modified table should look like:

$$\langle \textsc{add} : c, v_1 : v_2 : e, ps\rangle \quad \triangleright \quad \langle c, (v_1 +_{SE} v_2) : e, ps\rangle$$

$$\langle \textsc{branch}(c_1, c_2) : c, v : e, ps\rangle \;\triangleright\; \begin{cases} \langle c_1 : c, e, ps\rangle & \text{if } \textsc{tt} \sqsubseteq_{TE} v \\ \langle c_2 : c, e, ps\rangle & \text{if } \textsc{ff} \sqsubseteq_{TE} v \\ \langle c, e, \hat{ps}\rangle & \text{if } \textsc{err}_B \sqsubseteq_{TE} v \end{cases}$$

2. Create a copy of your source directory tree from lab 1 as a starting point for the implementation.

3. Implement steps 1 through 5 described in Section 5 above.

4. On the basis of suitably chosen example programs, prepare a demonstration of how the output produced by your tool supports the safety analysis and the optimisations suggested in Section 3 (as well as other meaningful analyses and optimisations you can think of). Discuss the *limitations* of the approach by presenting valid examples which the tool cannot handle.

5. Write a *report* containing all your results, both theoretical and practical. These should incude all example programs, pretty-printed with the results of the basic analysis as described above, plus the additional analyses and optimizations performed in the previous task and your conclusions.

7 Tips and Hints

- An elegant way of rewriting the virtual machine to work with abstract domains is to first parameterize the domains (using generic programming) and the operations (by providing an `Operations`-implementation in the constructor), and then simply replace

```
new YourExecuter<Integer, Boolean>(new IntBoolOps())
```

with

```
new YourExecuter<SignExc, TTExc>(new SignExcOps()).
```

- During the traversal of the configuration graph it is important that you do not return a mutated version of the configuration currently being processed! The visited configurations need to stay intact since 1) you want to keep track of which configurations you have already visited, and 2) the configurations are needed for the analysis after the traversal.

  The easiest way to make sure that the visited configurations stay intact is to let your configuration objects be immutable or at least override the `clone` method. A `clone` method should look something like this:

  ```
  public Configuration clone() {
      Configuration clone = new Configuration();
      clone.c = (Code) c.clone();
      clone.e = (Stack) e.clone();
      clone.s = (State) s.clone();
      return clone;
  }
  ```

- To decide if a configuration has already been visited or not, you may want to store the visited configurations in a `HashSet`. Keep in mind that this requires you to override the `hashCode` and `equals` methods in your configuration class. Here follows a sample implementation of these two methods:

  ```
  public int hashCode() {
      return c.hashCode() ^ e.hashCode() ^ s.hashCode();
  }

  public boolean equals(Object o) {
      if (!(o instanceof Configuration))
          return false;

      Configuration oc = (Configuration) o;
      return oc.c.equals(c) &&
  ```

```
                oc.e.equals(e) &&
                oc.s.equals(s);
    }
```

- Step 3 (b) can be simplified by noting that assignments are the only statements giving rise to STORE instructions. To compute the lub of the possible values of the arithmetic expression at an assignment, it suffices to compute the lub of the top-of-stack values for each related configuration that has STORE as its current instruction.

- Similarly, step 3 (c) can be simplified by noting that the **if**- and **while**-statements always rely on the BRANCH instruction. To compute the lub of branching and looping conditions, it suffices to compute the lub of the top-of-stack values for each related configuration that has BRANCH as its current instruction.