Pentagons

Eleonora Garbin 869831
Mattia Zonelli 870038

30/06/2021

Contents

1 Premises

After merging our implementation of Pentagons with last version of LiSA on 27th June, we noticed that the execution of program doesn't end but we need to stop it manually. In order to get the cfg we need to wait that the IDE give us the message "test passed" before stopping the execution.
We also noticed that sometimes if in the test function in file .imp we have some `if` statement without corresponding `else` branch, the program calls a lot of time the `assume` function on the false branch of each `if` and when it goes to intersect all the branches and performs the function `lubAux` of class `InverseSetLattice`, something goes wrong. Everything seems to run correctly if the for each `if` statement we write also the `else` branch.

2 Pentagons

We had to implement the abstract domain, Pentagons (`Pntg`), in LiSA. Pentagons are based on Intervals (`Intv`) and Strict Upper Bounds (`Sub`).
The elements of Pntg are of the form $x \in [a, b] \wedge x < y$, where x and y are program variables and $[a, b]$ is an Interval from a to b, $x < y$ is an Strict Upper Bound, more specifically y is a Strict Upper Bound of x.
Interval is a range of the possible values of the variable x, the inequalities are relations between x and other variables, these inequalities allow us to refine the Interval of the according variable.
For the implementation of `Pntg` we first implemented the Strict Upper Bound domain and we used the provided class for Intervals.

2.1 Intervals

`Intv` $[a, b]$, is a numeric abstract domain where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$ and $a < b$.
Lattice operations of Intervals:

- Order: $[a_1, b_1] \sqsubseteq_i \iff a_1 \geq a_2 \wedge b_1 \leq b_2$
- Bottom: $[a, b] = \bot_i \iff a \geq b$
- Top: $[a, b] = \top_i \iff a = -\infty \wedge b = +\infty$
- Join: $[a_1, b_1] \sqcup_i [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)]$
- Meet: $[a_1, b_1] \sqcap_i [a_2, b_2] = [max(a_1, a_2), min(b_1, b_2)]$
- Widening:$[a_1, b_1] \nabla_i [a_2, b_2] = [a_1 \leq a_2 ? a_2 : -\infty, b_1 \geq b_2 ? b_2 : +\infty]$

2.2 Strict Upper Bounds

`Sub` $x < y$ is an abstract domain, where the elements are represented with maps like $x \mapsto \{y_1, ..y_n\}$ which means that x is strictly smaller than each of the left values y.
Lattice operations of Strict Upper Bounds:

- Order: $s_1 \sqsubseteq_s s_2 \iff \forall x \in s_2.s_1(x) \supseteq s_2(x)$
- Bottom: $s = \bot_s \iff \exists x, y \in s.y \in s(x) \wedge x \in s(y)$
- Top: $s = \top_s \iff \forall x \in s.s(x) = \emptyset$
- Join: $s_1 \sqcup_s s_2 = \lambda x.s_1(x) \cap s_2(x)$
- Meet: $s_1 \sqcap_s s_2 = \lambda x.s_1(x) \cup s_2(x)$
- Widening: $s_1 \nabla_s s_2 = \lambda x.s_1(x) \subseteq s_2(x) ? s_2(x) : \emptyset$

Now finally, follow the lattice operations over `Pentagons`.

- Order: $\langle b_1, s_1\rangle \sqsubseteq_p \langle b_2, s_2\rangle \iff b_1 \sqsubseteq b_2 \wedge (\forall x \in s_2, \forall y \in s_2(x).y \in s_1(x) \vee sup(b_1(x)) < inf(b_1(y))$
- Bottom: $\langle b, s\rangle = \bot_p \Rightarrow b = \bot_b \vee s = \bot_s$
- Top: $\langle b, s\rangle = \top_p \Rightarrow b = \top_b \wedge s = \top_s$
- Join: $\langle b_1, s_1\rangle \sqcup_p^* \langle b_2, s_2\rangle = \langle b_1^* \sqcup_b b_2^*, s_1^* \sqcup_s s_2^*\rangle$
- Meet: $\langle b_1, s_1\rangle \sqcap_p \langle b_2, s_2\rangle = \langle b_1 \sqcap_b b_2, s_1 \sqcap_s s_2\rangle$
- Widening: $\langle b_1, s_1\rangle \nabla_p \langle b_2, s_2\rangle = \langle b_1 \nabla_b b_2, s_1 \nabla_s s_2\rangle$

3 Implementation

3.1 Intv

For the implementation of the Intervals we decided to use the already provided class, which extend the base non relation value domain.

3.2 Sub

3.2.1 UpperBounds

We create this class in order to save the strict upper bound of a variable, to do this the class extends `InverseSetLattice`.

Extending this abstract class, `UpperBounds` inherits functions like: `lubAux, glb, wideningAux` and the variable to store the set of elements contained in the lattice.

```
public class UpperBounds extends InverseSetLattice<UpperBounds, Identifier>
```

3.2.2 StrictUpperBounds

To implement the map $x \mapsto \{y_1, ..y_n\}$, where `x` is an identifier of variable in the program and $y_1, ..y_n$ are its strict upper bounds, we decided to extend `FunctionalLattice`.
In fact, `FunctionalLattice` provides us an implementation of a map between `Identifier` and a class that extend `BaseLattice` which in our case is `UpperBounds`.

```
public class StrictUpperBound
        extends FunctionalLattice<StrictUpperBound, Identifier, UpperBounds>
        implements ValueDomain<StrictUpperBound>
```

The main functions that we have implemented are:

```
public StrictUpperBound assign(Identifier id, ValueExpression expression, ProgramPoint pp)
throws SemanticException {
        if (pp.toString().contains("=")) {
            if (!function.containsKey(id)){

                Map<Identifier, UpperBounds> result_map  = mkNewFunction(function);
                // caso: y = x-1;
                if (expression instanceof BinaryExpression &&
                ((BinaryExpression)expression).getOperator() == BinaryOperator.NUMERIC_SUB){
                    BinaryExpression binaryExpression = (BinaryExpression) expression;
                    if (binaryExpression.getLeft() instanceof Identifier
                        && binaryExpression.getRight() instanceof Constant){
                        HashSet<Identifier> hs = new HashSet<>();
                        if(!result_map.get(binaryExpression.getLeft()).
                        toString().contains("#TOP#")) {
                            for (Identifier ub : result_map.get(binaryExpression.getLeft())
                            .elements()) {
                                hs.add(ub);
                            }
```

```
                    }
                    hs.add((Identifier) binaryExpression.getLeft());
                    result_map.put(id, new UpperBounds(hs));

                }
            }else {
                result_map.put(id, lattice.top());
            }
            return new StrictUpperBound(lattice, result_map);
        }
    }
    return new StrictUpperBound(lattice, function);
}
```

The function `assign` is called when in the test program, LiSA finds an assignment statement. We handle the first assignment of x where:

- `x = y - 1`, adding to the map the entry: $x \mapsto \{y \cup y.sub\}$
- in all the other cases, we add to the map $x \mapsto \{\top\}$.

```
public StrictUpperBound assume(ValueExpression expression, ProgramPoint pp)
throws SemanticException {
        if (expression instanceof UnaryExpression &&
        ((UnaryExpression) expression).getExpression() instanceof BinaryExpression) {
            expression = expression.removeNegations();
        }

        if (expression instanceof BinaryExpression) {

            BinaryExpression binaryExpression = (BinaryExpression) expression;
            Map<Identifier, UpperBounds>  result_map = mkNewFunction(function);
            if (binaryExpression.getLeft() instanceof Identifier &&
            binaryExpression.getRight() instanceof Identifier) {

                // case: == then sub of both identifier has to become the union of both.
                if (binaryExpression.getOperator() == BinaryOperator.COMPARISON_EQ) {
                    Identifier sx = (Identifier) binaryExpression.getLeft();
                    Identifier dx = (Identifier) binaryExpression.getRight();
```

```
            result_map.replace(sx, result_map.get(sx).glb(result_map.get(dx)));
            result_map.replace(dx, result_map.get(sx).glb(result_map.get(dx)));


            return new StrictUpperBound(lattice, result_map);
        }
        // case:  !=, nothing change
        if (binaryExpression.getOperator() == BinaryOperator.COMPARISON_NE){

            return new StrictUpperBound(lattice, result_map);
        }
        // cases < or > or <= or >=
        if (binaryExpression.getOperator() == BinaryOperator.COMPARISON_GT ||
                binaryExpression.getOperator() == BinaryOperator.COMPARISON_LT ||
                binaryExpression.getOperator() == BinaryOperator.COMPARISON_GE ||
                binaryExpression.getOperator() == BinaryOperator.COMPARISON_LE) {

            Identifier sx, dx;
            if (binaryExpression.getOperator() == BinaryOperator.COMPARISON_LT ||
                    binaryExpression.getOperator() == BinaryOperator.COMPARISON_LE){
                // "x < y" or "x <= y"
                sx = (Identifier) binaryExpression.getLeft();
                dx = (Identifier) binaryExpression.getRight();
            } else {
                // "x > y" or "x >= y"
                sx = (Identifier) binaryExpression.getRight();
                dx = (Identifier) binaryExpression.getLeft();
            }

            // "x < x" same identifier in both left and right hand-side of statement.
            if (sx.equals(dx) &&
            (binaryExpression.getOperator() == BinaryOperator.COMPARISON_GT ||
            binaryExpression.getOperator() == BinaryOperator.COMPARISON_LT)) {
                result_map.replace(sx, lattice.bottom());
            }
```

```
                HashSet<Identifier> hs = new HashSet<>();

                if (!result_map.get(sx).toString().contains("#TOP#")) {
                    // se il Sub di x aveva già delle varibili
                    for (Identifier ub : result_map.get(sx).elements()) {
                        hs.add(ub);
                    }
                }

                // se ho x < y devo aggiungere anche y all sub di x
                if (binaryExpression.getOperator() == BinaryOperator.COMPARISON_GT ||
                        binaryExpression.getOperator() == BinaryOperator.COMPARISON_LT){
                    hs.add(dx);
                }

                // x = s(x) U s(y)
                if (!result_map.get(dx).toString().contains("#TOP#")) {
                    // se il Sub di y aveva già delle varibili
                    for (Identifier ub : result_map.get(dx).elements()) {
                        hs.add(ub);
                    }
                }

                if (!hs.isEmpty()) result_map.replace(sx, new UpperBounds(hs));

                // to bottom
                if (result_map.containsKey(dx)) {
                    if (result_map.get(sx).elements().contains(dx)
                    && result_map.get(dx).elements().contains(sx)){
                        result_map.replace(sx, lattice.bottom());
                        result_map.replace(dx, lattice.bottom());

                    }
                }
                return new StrictUpperBound(lattice, result_map);

            }
        }
```

```
        }
        return new StrictUpperBound(lattice, function);
    }
```

The function `Assume` is called when in the code there is a comparison.
We handled only the following cases in these ways:
(with `x.sub` we refer to the set of the strict upper bounds of x)

- $x == y$
  $x \mapsto \{x.sub \cup y.sub\}$
  $y \mapsto \{x.sub \cup y.sub\}$
- $x != y$
  The corresponding strict upper bounds don't change;
- $x < y$
  $x \mapsto \{x.sub \cup y.sub \cup y\}$
- $x \leq y$
  $x \mapsto \{x.sub \cup y.sub\}$
- $x \geq y$
  $y \mapsto \{x.sub \cup y.sub\}$
- $x > y$
  $y \mapsto \{x.sub \cup y.sub \cup x\}$
- $x < x$
  $x \mapsto \{\bot\}$

If after the modification of the strict upper bounds of some variables we have in example that $x \in y.sub \wedge y \in x.sub$, we have to set both y.sub and x.sub to $\{\bot\}$ lattice.

3.3 Pntg

We implemented `Pentagons` as a value domain, in which we have our value environment `StrictUpperBounds` and a value environment of `Interval`.

```
public class Pentagons implements ValueDomain<Pentagons> {

    private final ValueEnvironment<Interval> intv;
    private final StrictUpperBound sub;
```

All the operations are based on the combination of the functions of `Interval` and `StrictUpperBound`. To refine intervals according to the information provided by the sub, we implemented the following function:

```
private Pentagons refine(){
        Map<Identifier, Interval> result = new HashMap<>();
        for (Identifier sx : sub.getKeys()){
            if (sub.getState(sx).isBottom() || sub.getState(sx).isTop()){
                result.put(sx, intv.getState(sx));
            }else{
                for (Identifier dx : sub.getState(sx)){

                    if (!intv.getState(sx).isBottom() &&
                        !intv.getState(sx).isTop() &&
                        !intv.getState(dx).isBottom() &&
                        !intv.getState(dx).isTop()){
                        Integer s_low = intv.getState(sx).getLow();
                        Integer s_high = intv.getState(sx).getHigh();
                        Integer d_low = intv.getState(dx).getLow();
                        Integer d_high = intv.getState(dx).getHigh();

                        if (s_low >= d_low){
                            result.put(sx, new Interval());
                        }else if (s_high >= d_low){
                            result.put(sx, new Interval(s_low, d_low-1));
                        }else {
                            result.put(sx, intv.getState(sx));
```

```
                    }
                }else{
                    result.put(sx, intv.getState(sx));
                }
            }
        }
    }

    return new Pentagons(new ValueEnvironment<>(new Interval(), result), sub);
}
```

Function `refine` is called after functions Assign or Assume.
It checks for each identifier `x`, in the map of the StrictUpperBound, with Interval $[a_x, b_x]$ if there is any $y \in x.sub$ with Interval $[a_y, b_y]$ and $b_x \geq a_y$ or $a_x \geq a_y$.
In the first case we modify the Interval of `x` to $[a_x, a_y - 1]$; in the second case we change the Interval of `x` to $\top$.

4 Examples

4.1 Example 1

```
test1() {
    def x = 3;
    def y = 4;
    def p = 8;
    if (y < p){
        if (x <= y){
            y = y +1;
        }
    }else{
        y = 1;
    }
    p = 0;
}
```

As we can see from figure 1 in the true branch of statement $< (y, p)$ to the `y.sub` it adds the identifier `p` then in the true branch of statement $<= (x, y)$ the `x.sub` become the union of itself and `y.sub`. So the following rules have been applied:

- $x < y : x \mapsto \{x.sub \cup y.sub \cup y\}$
- $x \leq y : x \mapsto \{x.sub \cup y.sub\}$

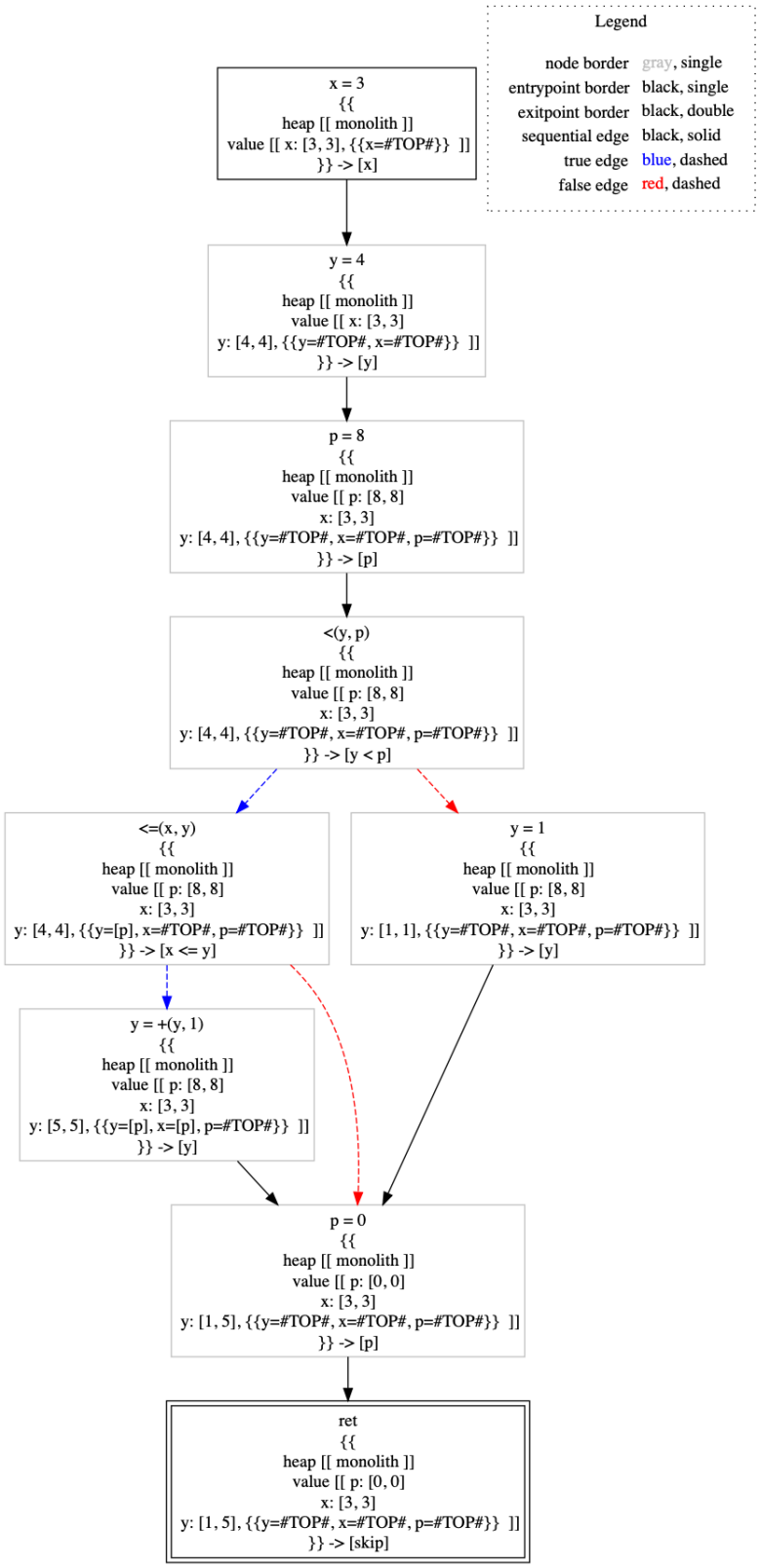

Figure 1: Example 1

4.2 Example 2

```
test2() {
    def y = 4;
    def w = 8;
    def p = w - 1;
    if (y < p){
        y = y+1;
    }else{
        y = 1;
    }
    return p;
}
```

As we can see from figure 2 the assignment $p = -(w, 1)$, it adds the identifier w to the $p.sub$ and $p.intv$ is refined as $[a_w - 1, b_w - 1]$. Then in the true branch of statement $< (y, p)$, the rule $y < p : x \mapsto \{y.sub \cup p.sub \cup p\}$ has been applied in fact the $y.sub$ became $\{w, p\}$

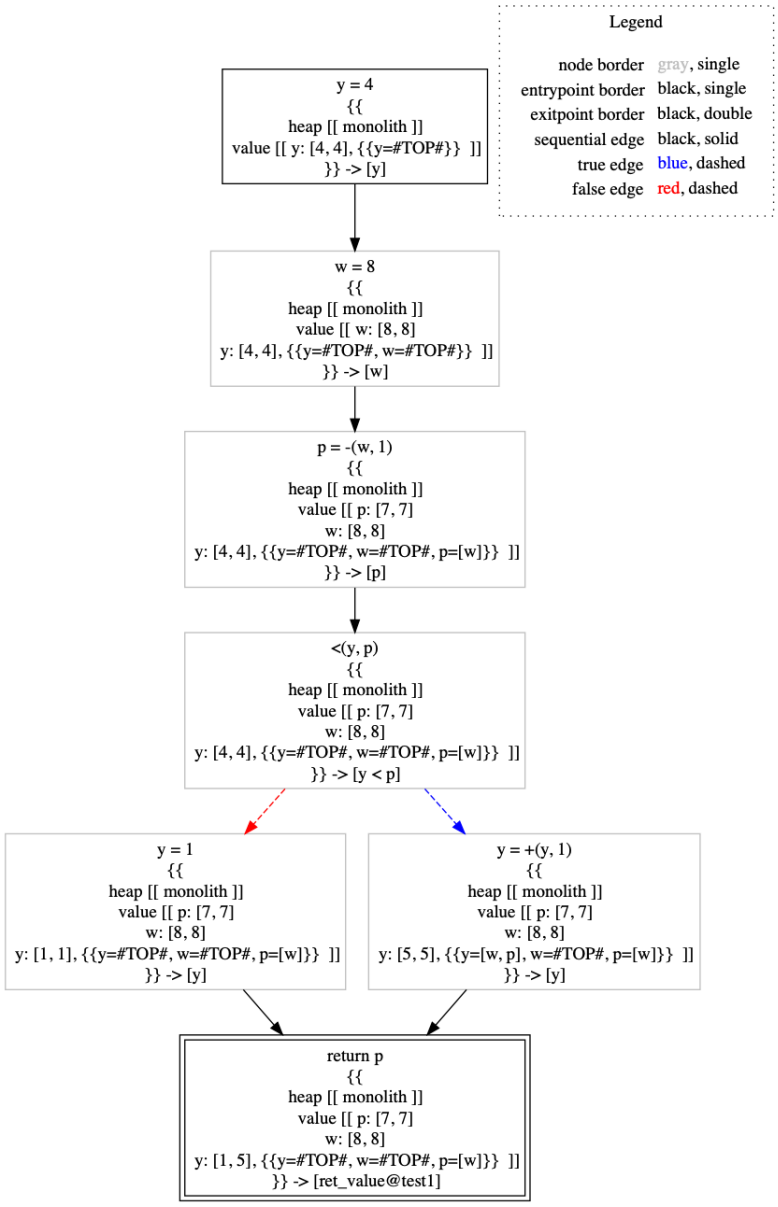

Figure 2: Example 2

4.3 Example 3

```
test3(z) {
    def y = 4;
    def x = 1;
    if (z < 10){
        x = 1;
    }else{
        x = 5;
    }
    if (x < y){
        z = z+1;
    }
    return z;
}
```

In the figure 3, after the statement $< (x, y)$, in the true branch the identifier y is added to $x.sub$ and as consequence $x.sub$ is used to refine $x.intv$ as $[a_x, b_y - 1]$ becoming $[1, 3]$.

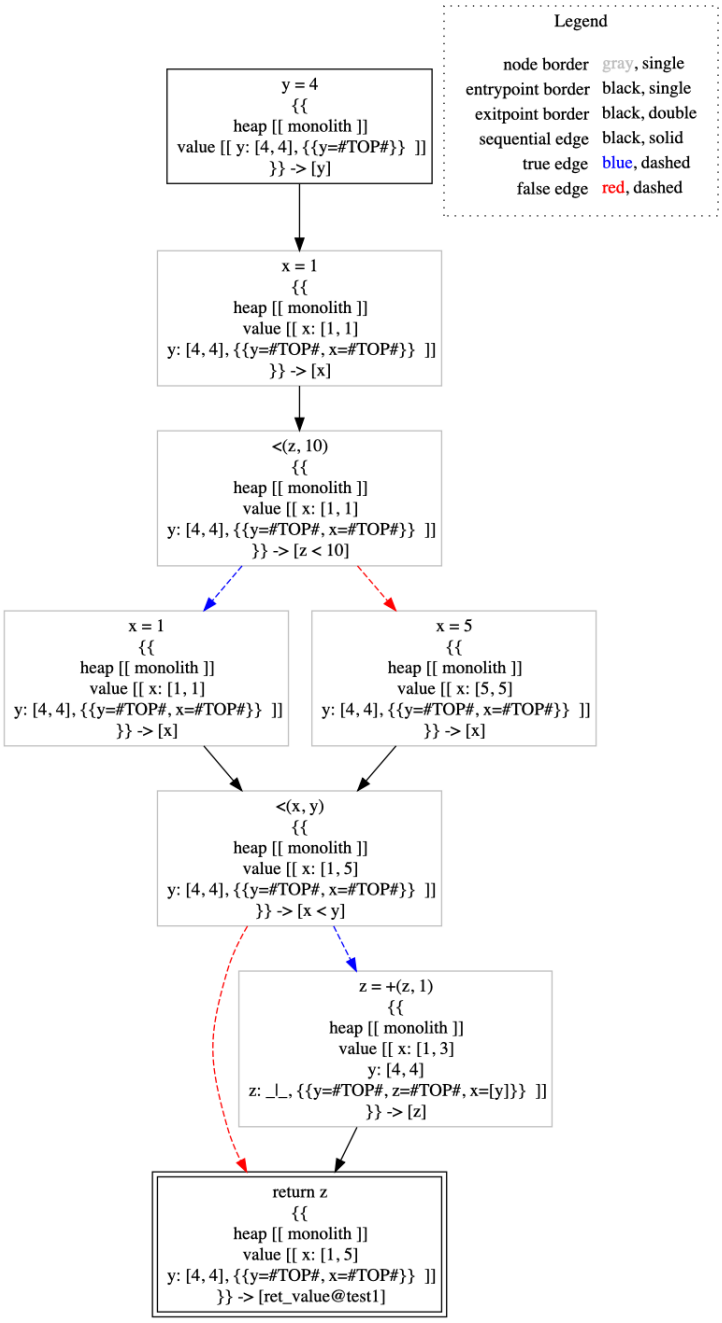

Figure 3: Example 3

4.4 Example 4

```
test4() {
    def y = 4;
    def x = 1;
    if (x < x){
        y = y+1;
    }
    return y;
}
```

In the following example using the rule $x < x$: $x \mapsto \{\bot\}$, and then in the true branch of statement $<(x, x)$ the whole lattice becomes a $\bot$ because $x.sub$ is $\bot$. In fact if at least one Sub is $\bot$ then the whole lattice becomes $\bot$. This happen according to the rule $\langle b, s\rangle = \bot_p \Rightarrow b = \bot_b \vee s = \bot_s$, where $\langle b, s\rangle$ is our Pentagons.

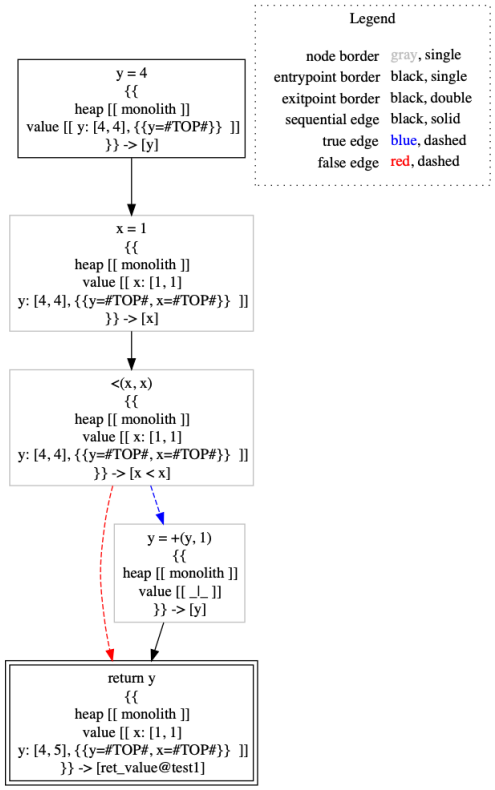

Figure 4: Example 4

References

[1] Manuel Fähndrich Francesco Logozzo. A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. 2010.