

8 Path Sensitivity

Until now, we have ignored the values of conditions by simply treating `if`- and `while`-statements as a nondeterministic choice between the two branches. This is called a *path insensitive* analysis as it does not distinguish different paths that lead to a given program point. This technique fails to include some information that could potentially be used in a static analysis. Consider for example the following program:

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  z = z+x;
  if (17 > y) { y = y+1; }
  x = x-1;
}
```

The previous interval analysis (with widening) will conclude that after the `while`-loop the variable `x` is in the interval $[-\infty, \infty]$, `y` is in the interval $[0, \infty]$, and `z` is in the interval $[-\infty, \infty]$. However, in view of the conditionals being used, this result seems too pessimistic.

To exploit the available information, we shall extend the language with two artificial statements: `assert(E)` and `refute(E)`, where *E* is a condition from our base language. In the interval analysis, the constraints for these new statement will narrow the intervals for the various variables by exploiting the information that *E* must be *true* respectively *false*.

The meanings of the conditionals are then encoded by the following program transformation:

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  assert(x > 0);
  z = z+x;
  if (17 > y) { assert(17 > y); y = y+1; }
  x = x-1;
}
refute(x > 0);
```

Constraints for a node *v* with an `assert` or `refute` statement may trivially be given as:

$$\llbracket v \rrbracket = JOIN(v)$$

in which case no extra precision is gained. In fact, it requires insight into the specific static analysis to define non-trivial and sound constraints for these constructs.

For the interval analysis, extracting the information carried by general conditions, or *predicates*, such as $E_1 > E_2$ or $E_1 == E_2$ relative to the lattice elements is complicated and in itself an area of considerable study. For our purposes, we need only consider conditions of the two kinds $id > E$ or $E > id$, the first of which for the case of **assert** can be handled by:

$$\llbracket v \rrbracket = JOIN(v)[id \mapsto gt(JOIN(v)(id), eval(JOIN(v), E))]$$

where:

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \cap [l_2, \infty]$$

Exercise 8.1: Argue that this constraint for **assert** is sound and monotone.

The cases of **refute** and the dual condition are handled in similar fashions, and all other conditions are given the trivial, but sound identity constraint.

With this refinement, the interval analysis of the above example will conclude that after the **while**-loop the variable **x** is in the interval $[-\infty, 0]$, **y** is in the interval $[0, 17]$, and **z** is in the interval $[0, \infty]$.

Exercise 8.2: Discuss how more conditions may be given non-trivial constraints for **assert** and **refute** to improve analysis precision further.

8.1 Branch Correlations

The use of **assert** or **refute** statements provides a simple kind of path sensitivity, however it is insufficient for reasoning about correlations of branches in programs. Here is a typical example:

```

    if (condition) {
        open();
        flag = 1;
    } else {
        flag = 0;
    }
    ...
    if (flag) {
        close();
    }

```

We here assume that **open** and **close** are built-in functions for opening and closing a single file. The file is initially closed, and the “...” are statements that do not call **open** or **close** or modify **flag**. We wish to design an analysis that can check that **close** is only called if the file is currently open.

As a starting point, we use this lattice for modeling the open/closed status of the file:

$$L = (2^{\{\text{open}, \text{closed}\}}, \subseteq)$$

For every CFG node v the variable $\llbracket v \rrbracket$ denotes the possible status of the file at the program point after the node. For `open` and `close` statements the constraints are:

$$\begin{aligned}\llbracket \text{open}() \rrbracket &= \{\text{open}\} \\ \llbracket \text{close}() \rrbracket &= \{\text{closed}\}\end{aligned}$$

For the entry node, we define:

$$\llbracket \text{entry} \rrbracket = \{\text{closed}\}$$

and for every other node, which does not modify the file status, the constraint is simply:

$$\llbracket v \rrbracket = \text{JOIN}(v)$$

where *JOIN* is defined as usual for a forward, may analysis:

$$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket$$

In the example program, the `close` function is clearly called if and only if `open` is called, but the current analysis fails to discover this.

Exercise 8.3: Write the constraints being produced for the example program and show that the solution for $\llbracket \text{flag} \rrbracket$ (the node for the last `if` condition) is $\{\text{open}, \text{closed}\}$.

Arguing that the program has the desired property obviously involves the `flag` variable, which the lattice above ignores. So, we can try with a slightly more sophisticated lattice:

$$L' = (2^{\{\text{open}, \text{closed}\}}, \subseteq) \times (2^{\{\text{flag}=0, \text{flag} \neq 0\}}, \subseteq)$$

Additionally, we insert `assert` and `refute` to make sure that conditionals are not ignored:

```

    if (condition) {
        assert(condition);
        open();
        flag = 1;
    } else {
        refute(condition);
        flag = 0;
    }
    ...
    if (flag) {
        assert(flag);
        close();
    } else {
        refute(flag);
    }

```

This is still insufficient, though. At the program point after the first `if-else` statement, the analysis only knows that `open` *may* have been called and `flag` *may* be 0.

Exercise 8.4: Specify the constraints that fit with the L' lattice. Then show that the analysis produces the lattice element $(2^{\{\text{open}, \text{closed}\}}, 2^{\{\text{flag}=0, \text{flag}\neq 0\}})$ for the first node after the the first `if-else` statement.

The present analysis is also called an *independent attribute analysis* as the abstract value of the file is independent of the abstract value of the boolean flag. What we need is a *relational* analysis that can keep track of relations between variables. This can be achieved by generalizing the analysis to maintain *multiple* abstract states per program point. If L is the existing lattice, we replace it by

$$C \mapsto L$$

where C is a finite set of *contexts*. A context is a predicate over the program state. (For instance, a condition expression in TIP defines such a predicate.) In general, each statement is then analyzed in $|C|$ different contexts, each describing a set of paths that lead to the statement. For the example above, we can use $C = \{\text{flag} = 0, \text{flag} \neq 0\}$.

The constraints for `open`, `close`, and `entry` are:

$$\llbracket \text{open}() \rrbracket = \lambda c. \{\text{open}\}$$

$$\llbracket \text{close}() \rrbracket = \lambda c. \{\text{closed}\}$$

$$\llbracket \text{entry} \rrbracket = \lambda c. \{\text{closed}\}$$

The constraints for assignments make sure that `flag` gets special treatment:

$$\llbracket \text{flag} = 0 \rrbracket = [\text{flag} = 0 \mapsto \bigcup_{c \in C} \text{JOIN}(v)(c), \text{flag} \neq 0 \mapsto \emptyset]$$

$$\llbracket \text{flag} = n \rrbracket = [\text{flag} \neq 0 \mapsto \bigcup_{c \in C} \text{JOIN}(v)(c), \text{flag} = 0 \mapsto \emptyset]$$

$$\llbracket \text{flag} = E \rrbracket = \lambda c. \bigcup_{c \in C} \text{JOIN}(v)(c)$$

Here, n is an *intconst* other than 0 and E is a non-*intconst* expression, and JOIN is defined pointwise:

$$\text{JOIN}(v)(c) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket(c)$$

The situation $\llbracket v \rrbracket(c) = \emptyset$ corresponds to c being an infeasible context at v . For `assert`, we also give special treatment to `flag`:

$$\llbracket \text{assert}(\text{flag}) \rrbracket = [\text{flag} \neq 0 \mapsto \text{JOIN}(v)(\text{flag} \neq 0), \text{flag} = 0 \mapsto \emptyset]$$

Notice the small but important difference from the constraint for the `flag = 1` case. As before, the case for `refute` is similar.

Exercise 8.5: Give an appropriate constraint for `refute(flag)`.

Finally, for any other node v , including other `assert/refute` nodes, the constraint simply propagates and joins the dataflow, which is always sound but maybe not very precise:

$$\llbracket v \rrbracket = \lambda c. \text{JOIN}(v)(c)$$

For our specific program, the following constraints are generated:

```

[[entry]] = λc.{closed}
[[condition]] = [[entry]]
[[assert(condition)]] = [[condition]]
[[open()]] = λc.{open}
[[flag = 1]] = [flag ≠ 0 ↦ ⋃c∈C [[open()]](c), flag = 0 ↦ ∅]
[[refute(condition)]] = [[condition]]
[[flag = 0]] = [flag = 0 ↦ ⋃c∈C [[refute(condition)]](c), flag ≠ 0 ↦ ∅]
[[...]] = λc.([[flag = 1]](c) ∪ [[flag = 0]](c))
[[flag]] = [[...]]
[[assert(flag)]] = [flag ≠ 0 ↦ [[flag]](flag ≠ 0), flag = 0 ↦ ∅]
[[close()]] = λc.{closed}
[[refute(flag)]] = [flag = 0 ↦ [[flag]](flag = 0), flag ≠ 0 ↦ ∅]
[[exit]] = λc.([[close()]](c) ∪ [[...]](c))

```

The minimal solution is, for each $\llbracket v \rrbracket(c)$:

	flag = 0	flag ≠ 0
$\llbracket \text{entry} \rrbracket$	{closed}	{closed}
$\llbracket \text{condition} \rrbracket$	{closed}	{closed}
$\llbracket \text{assert}(\text{condition}) \rrbracket$	{closed}	{closed}
$\llbracket \text{open}() \rrbracket$	{open}	{open}
$\llbracket \text{flag} = 1 \rrbracket$	∅	{open}
$\llbracket \text{refute}(\text{condition}) \rrbracket$	{closed}	{closed}
$\llbracket \text{flag} = 0 \rrbracket$	{closed}	∅
$\llbracket \dots \rrbracket$	{closed}	{open}
$\llbracket \text{flag} \rrbracket$	{closed}	{open}
$\llbracket \text{assert}(\text{flag}) \rrbracket$	∅	{open}
$\llbracket \text{close}() \rrbracket$	{closed}	{closed}
$\llbracket \text{refute}(\text{flag}) \rrbracket$	{closed}	∅
$\llbracket \text{exit} \rrbracket$	{closed}	{open}

The analysis produces the lattice element $[\text{flag} = 0 \mapsto \{\text{closed}\}, \text{flag} \neq 0 \mapsto \{\text{open}\}]$ for the program point after the first `if-else` statement. The constraint for the `assert(flag)` statement will eliminate the possibility that the file is closed at that point. This ensures that `close` is only called if the file is open, as desired. In fact, the analysis is able to produce *precise* information for this

particular program, in the sense that the abstract value $\{\text{open}, \text{closed}\}$ does not occur in the solution.

Exercise 8.6: For the present example, the basic lattice L is defined as a powerset of a finite set A . Show that $C \mapsto 2^A$ is isomorphic to $2^{C \times A}$. (This explains why such analyses are called *relational*.)

Exercise 8.7: Describe a variant of the example program above where the present analysis would be improved if combining it with constant propagation.

In general, the program analysis designer is left with the choice of C . Often C consists of combinations of predicates that appear in conditionals in the program. This quickly results in an exponential blow-up: for k predicates, each statement may need to be analyzed in 2^k different contexts. In practice, however, there is usually much redundancy in these analysis steps. Thus, in addition to the challenge of reasoning about the **assert/refute** predicates relative to the lattice elements, it requires a considerable effort to avoid too many redundant computations in path sensitive analysis. One approach is *iterative refinement* where C is initially a single universal context, which is then iteratively refined by adding relevant predicates until either the desired properties can be established or disproved or the analysis is unable to select relevant predicates and hence gives up.

Exercise 8.8: Assume that we change the rule for **open** from

$$\llbracket \text{open}() \rrbracket = \lambda c. \{\text{open}\}$$

to

$$\llbracket \text{open}() \rrbracket = \lambda c. \text{if } JOIN(v)(c) = \emptyset \text{ then } \emptyset \text{ else } \{\text{open}\}$$

Argue that this is sound and for some programs more precise than the original rule.

Exercise 8.9: The following is a variant of the previous example program:

```
if (condition) {
    flag = 1;
} else {
    flag = 0;
}
...
if (condition) {
    open();
}
...
if (flag) {
    close();
}
```

Show how a path sensitive analysis can prove for this variant that `close` is called if and only if `open` has been called.

Exercise 8.10: Construct yet another variant of the `open/close` example program where the desired property can only be established with a choice of C that includes a predicate that does *not* occur as a conditional expression in the program source. (Such a program may be challenging to handle with the iterative refinement technique.)