

# Static Program Analysis


## Part 6 – path sensitivity

<http://cs.au.dk/~amoeller/spa/>

Anders Møller & Michael I. Schwartzbach  
Computer Science, Aarhus University

# Information in conditions

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



The interval analysis (with widening) concludes:

$x = [-\infty, \infty]$ ,  $y = [0, \infty]$ ,  $z = [-\infty, \infty]$

# Modeling conditions

Add artificial “assert” statements:

The statement `assert(E)` models that *E* is *true* in the current program state

- it causes a runtime error otherwise
- but we only insert it where the condition will always be true

# Encoding conditions

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

preserves semantics since `assert`s are guarded by conditions

(alternatively, we could add dataflow constraints on the CFG *edges*)

# Constraints for assert

A trivial but sound constraint:

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

A non-trivial constraint for `assert (x > E)`:

$$\llbracket v \rrbracket = JOIN(v)[x \rightarrow gt(JOIN(v)(x), eval(JOIN(v), E))]$$

where

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \cap [l_2 + 1, \infty]$$

this computes the interval for `x` that is consistent with the `assert (x > E)`

# Constraints for assert

A trivial but sound constraint:

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

A non-trivial constraint for `assert(x>E)`:

$$\llbracket v \rrbracket = JOIN(v)[x \rightarrow gt(JOIN(v)(x), eval(JOIN(v), E))]$$

where

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \sqcap [l_2+1, \infty]$$

For example:

$$gt([2, 4], [0, 6]) = [2, 4] \sqcap [1, \infty] = [2, 4]$$

$$gt([0, 6], [2, 4]) = [0, 6] \sqcap [3, \infty] = [3, 6]$$

$$gt([2, 2], [3, 4]) = [2, 2] \sqcap [4, \infty] = \perp$$

# Constraints for assert

A trivial but sound constraint:

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

A non-trivial constraint for `assert(x>E)`:

$$\llbracket v \rrbracket = JOIN(v)[x \rightarrow gt(JOIN(v)(x), eval(JOIN(v), E))]$$

where

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \cap [l_2+1, \infty]$$

Similar constraints can be defined for variants

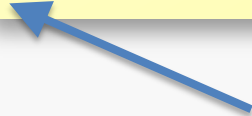
$$gte([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \cap [l_2, \infty]$$

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

$$lt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \cap [-\infty, h_2-1]$$

# Exploiting conditions

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



The interval analysis now concludes:

$x = [-\infty, 0]$ ,  $y = [0, 17]$ ,  $z = [0, \infty]$



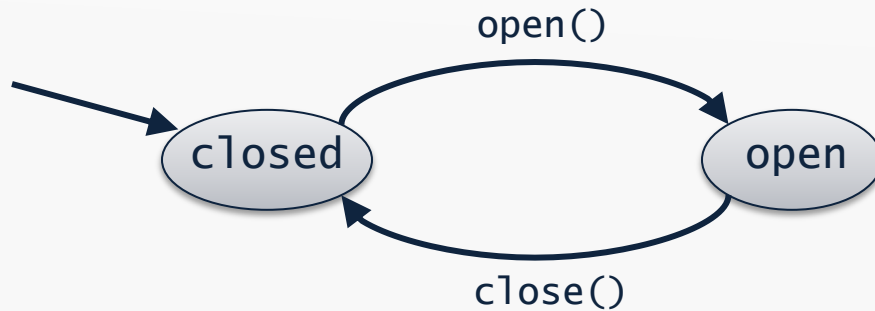
# Branch correlations

- With assert we have a simple form of *path sensitivity* (sometimes called *control sensitivity*)
- But it is insufficient to handle *correlation* of branches:

```
if (17 > x) { ... }  
... // statements that do not change x  
if (17 > x) { ... }  
...
```

# Open and closed files

- Built-in functions `open()` and `close()` on a file
- Requirements:
  - never `close` a closed file
  - never `open` an open file



- We want a static analysis to check this...  
(for simplicity, let us assume there is only one file)

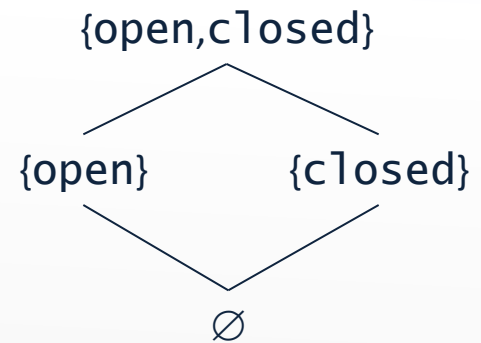
# A tricky example

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

# The naive analysis (1/2)

- The lattice models the status of the file:

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



- For every CFG node,  $v$ , we have a constraint variable  $\llbracket v \rrbracket$  denoting the status *after*  $v$
- $JOIN(v) = \bigcup_{w \in pred(v)} \llbracket w \rrbracket$

# The naive analysis (2/2)

- Constraints for interesting statements:

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

$\llbracket \text{open}() \rrbracket = \{\text{open}\}$

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

- For all other CFG nodes:

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

- Before the `close()` statement the analysis concludes that the file is `{open, closed}` 😞

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

...
if (flag) {
    close();
}
```

# The slightly less naive analysis

- We obviously need to keep track of the `flag` variable
- Our second attempt is the lattice:

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

- Additionally, we add `assert(...)` to model conditionals
- Even so, we still only know that the file is `{open, closed}` and that `flag` is `{flag=0, flag≠0}` 😞

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

...
if (flag) {
    close();
}
```

# Enhanced program

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

# Relational analysis

- We need an analysis that keeps track of *relations* between variables
- One approach is to maintain *multiple* abstract states per program point, one for each *path context*
- For the file example we need the lattice:

$$L = \text{Paths} \rightarrow 2^{\{\text{open}, \text{closed}\}} \quad (\text{note: isomorphic to } 2^{\text{Paths} \times \{\text{open}, \text{closed}\}})$$

where  $\text{Paths} = \{f \mid ag=0, f \mid ag \neq 0\}$  is the set of path contexts



# Relational constraints (1/2)

- For the file statements:

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

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

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

- For flag assignments:

$$\llbracket \text{flag} = 0 \rrbracket = [ \text{flag} = 0 \rightarrow \bigcup_{p \in P} \text{JOIN}(v)(p), \text{flag} \neq 0 \rightarrow \emptyset ]$$

$$\llbracket \text{flag} = n \rrbracket = [ \text{flag} \neq 0 \rightarrow \bigcup_{p \in P} \text{JOIN}(v)(p), \text{flag} = 0 \rightarrow \emptyset ]$$

$$\llbracket \text{flag} = E \rrbracket = \lambda q. \bigcup_{p \in P} \text{JOIN}(v)(p) \quad \text{for any other } E$$

"infeasible"



where  $n$  is a non-0 constant number

# Relational constraints (2/2)

- For `assert` statements:

$$\llbracket \text{assert}(f \wedge ag) \rrbracket =$$

$$[f \wedge ag \neq 0 \rightarrow \text{JOIN}(v)(f \wedge ag \neq 0), f \wedge ag = 0 \rightarrow \emptyset]$$

$$\llbracket \text{assert}(!f \wedge ag) \rrbracket =$$

$$[f \wedge ag = 0 \rightarrow \text{JOIN}(v)(f \wedge ag = 0), f \wedge ag \neq 0 \rightarrow \emptyset]$$

- For all other CFG nodes:

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

# Generated constraints

```
[[entry]] = λp.{c1osed}
[[condition]] = [[entry]]
[[assert(condition)]] = [[condition]]
[[open()]] = λp.{open}
[[flag = 1]] = [flag≠0→U [[open()]](p), flag=0→∅]
[[assert(!condition)]] = [[condition]]
[[flag = 0]] = [flag=0→U [[assert(!condition)]](p), flag≠0→∅]
[[...]] = λp.([[flag = 1]](p) U [[flag = 0]](p))
[[flag]] = [[...]]
[[assert(flag)]] = [[flag≠0→[[flag]](flag≠0), flag=0→∅]
[[close()]] = λp.{c1osed}
[[assert(!flag)]] = [flag=0→[[flag]](flag=0), flag≠0→∅]
[[exit]] = λp.([[close()]](p) U [[assert(!flag)]](p))
```

# Minimal solution

	flag = 0	flag ≠ 0
[[entry]]	{closed}	{closed}
[[condition]]	{closed}	{closed}
[[assert(condition)]]	{closed}	{closed}
[[open()]]	{open}	{open}
[[flag = 1]]	∅	{open}
[[assert(!condition)]]	{closed}	{closed}
[[flag = 0]]	{closed}	∅
[[...]]	{closed}	{open}
[[flag]]	{closed}	{open}
[[assert(flag)]]	∅	{open}
[[close()]]	{closed}	{closed}
[[assert(!flag)]]	{closed}	∅
[[exit]]	{closed}	{closed}

We now know the file is open before `close()` 😊

# Challenges

- The static analysis designer must choose Paths
  - often as boolean combinations of predicates from conditionals
  - iterative refinement (e.g. *counter-example guided abstraction refinement*) can be used for gradually finding relevant predicates
- Exponential blow-up:
  - for  $k$  predicates, we have  $2^k$  different contexts
  - redundancy often cuts this down
- Reasoning about `assert`:
  - how to update the lattice elements with sufficient precision?
  - possibly involves heavy-weight theorem proving

# Improvements

- Run auxiliary analyses first, for example:
  - constant propagation
  - sign analysis

will help in handling flag assignments

- Dead code propagation, change

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

into the still sound but more precise

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