Information in conditions

\[
x = \text{input}; \\
y = 0; \\
z = 0; \\
\text{while (}x>0) \{ \\
    z = z+x; \\
    \text{if (17}>y) \{ y = y+1; \} \\
    x = x-1; \\
\}\n\]

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

```plaintext
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 asserts are guarded by conditions)

(alternatively, we could add dataflow constraints on the CFG edges)
Constraints for assert

• A trivial but sound constraint:
  \[ [v] = \text{JOIN}(v) \]

• A non-trivial constraint for \text{assert}(x > E):
  \[ [v] = \text{JOIN}(v)[x \mapsto \text{gt}(\text{JOIN}(v)(x), \text{eval}(\text{JOIN}(v), E))] \]
  where
  \[ \text{gt}([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \cap [l_2, \infty] \]

• Similar constraints are defined for the dual cases
• More tricky to define for other conditions...
Exploiting conditions

The interval analysis now concludes:

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

```plaintext
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)
if (condition) {
    open();
    flag = 1;
} else {
    flag = 0;
}
...
if (flag) {
    close();
}
The naive analysis (1/2)

- The lattice models the status of the file:
  \[ L = (2^{\{\text{open, closed}\}}, \subseteq) \]

- For every CFG node, \( v \), we have a constraint variable \([v]\) denoting the status after \( v \)

- \( \text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} [w] \)
The naive analysis (2/2)

• Constraints for interesting statements:
  \[ \text{entry} = \{ \text{closed} \} \]
  \[ \text{open()} = \{ \text{open} \} \]
  \[ \text{close()} = \{ \text{closed} \} \]

• For all other CFG nodes:
  \[ \[ v \] = \text{JOIN}(v) \]

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

```java
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 \neq 0\} 😞

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

```java
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:

\[
\begin{align*}
L &= \text{Paths} \rightarrow 2^{\{\text{open}, \text{closed}\}} \\
\text{where Paths} &= \{\text{flag}=0, \text{flag} \neq 0\} \text{ is the set of path contexts}
\end{align*}
\]

(note: isomorphic to \(2^{\text{Paths} \times \{\text{open}, \text{closed}\}}\))
Relational constraints (1/2)

• For the file statements:

\[
\begin{align*}
\llbracket \text{entry} \rrbracket &= \lambda p.\{\text{closed}\} \\
\llbracket \text{open()\} &= \lambda p.\{\text{open}\} \\
\llbracket \text{closed()\} &= \lambda p.\{\text{closed}\}
\end{align*}
\]

• For flag assignments:

\[
\begin{align*}
\llbracket \text{flag} = 0 \rrbracket &= [\text{flag}=0 \rightarrow \bigcup_{p \in P} JOIN(v)(p), \text{flag}\neq 0 \rightarrow \emptyset] \\
\llbracket \text{flag} = n \rrbracket &= [\text{flag}\neq 0 \rightarrow \bigcup_{p \in P} JOIN(v)(p), \text{flag}=0 \rightarrow \emptyset] \\
\llbracket \text{flag} = E \rrbracket &= \lambda q. \bigcup_{p \in P} JOIN(v)(p) \quad \text{for any other } E
\end{align*}
\]

\(\text{"infeasible"}\)
Relational constraints (2/2)

• For `assert` statements:

\[ [\text{assert}(\text{flag})] = \]
\[ [\text{flag} \neq 0 \rightarrow JOIN(v)(\text{flag} \neq 0), \text{flag} = 0 \rightarrow \emptyset] \]
\[ [\text{assert}(\neg \text{flag})] = \]
\[ [\text{flag} = 0 \rightarrow JOIN(v)(\text{flag} = 0), \text{flag} \neq 0 \rightarrow \emptyset] \]

• For all other CFG nodes:

\[ [v] = JOIN(v) = \lambda p. \bigcup \[w]\](p) \]
\[ w \in \text{pred}(v) \]
Generated constraints

\[
\begin{align*}
[\text{entry}] &= \lambda p.\{\text{closed}\} \\
[\text{condition}] &= [\text{entry}] \\
[\text{assert(\text{condition})}] &= [\text{condition}] \\
[\text{open()}] &= \lambda p.\{\text{open}\} \\
[\text{flag} = 1] &= [\text{flag} \neq 0 \rightarrow U [\text{open()}](p), \text{flag}=0 \rightarrow \emptyset] \\
[\text{assert(!\text{condition})}] &= [\text{condition}] \\
[\text{flag} = 0] &= [\text{flag} = 0 \rightarrow U [\text{assert(!\text{condition})}](p), \text{flag} \neq 0 \rightarrow \emptyset] \\
[\ldots] &= \lambda p.([\text{flag} = 1](p) \cup [\text{flag} = 0](p)) \\
[\text{flag}] &= [\ldots] \\
[\text{assert(\text{flag})}] &= [\text{flag} \neq 0 \rightarrow [\text{flag}](\text{flag} \neq 0), \text{flag}=0 \rightarrow \emptyset] \\
[\text{close()}] &= \lambda p.\{\text{closed}\} \\
[\text{assert(!\text{flag})}] &= [\text{flag} = 0 \rightarrow [\text{flag}](\text{flag} = 0), \text{flag} \neq 0 \rightarrow \emptyset] \\
[\text{exit}] &= \lambda p.([\text{close()}](p) \cup [\text{assert(!\text{flag})}](p))
\end{align*}
\]
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
  \[
  \llbracket \text{open()} \rrbracket = \lambda p.\text{if } JOIN(v)(p)=\emptyset \text{ then } \emptyset \text{ else } \{\text{open}\}
  \]