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 artifical “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:

$$[\![v]\!] = JOIN(v)$$

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

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

where

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

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

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*)

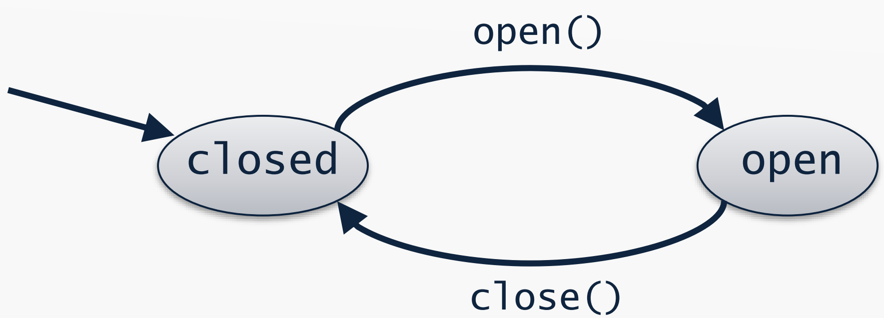
- But it is insufficient to handle *correlation* of branches:

```
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)

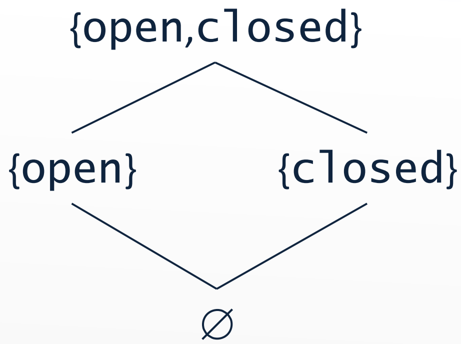
A tricky example

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

The naive analysis (1/2)

- The lattice models the status of the file:

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



- For every CFG node, v, we have a constraint variable ⟦v⟧ denoting the status *after* v

- $JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$

The naive analysis (2/2)

- Constraints for interesting statements:

  $[\![entry]\!] = \{\texttt{closed}\}$

  $[\![\texttt{open()}]\!] = \{\texttt{open}\}$

  $[\![\texttt{close()}]\!] = \{\texttt{closed}\}$

- For all other CFG nodes:

  $[\![v]\!] = JOIN(v)$

- 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:

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

- 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:

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

  where $\text{Paths} = \{\texttt{flag}=0, \texttt{flag}\neq 0\}$ is the set of path contexts

Relational constraints (1/2)

- For the file statements:

$$[\![\mathit{entry}]\!] = \lambda p.\{\texttt{closed}\}$$

$$[\![\texttt{open()}]\!] = \lambda p.\{\texttt{open}\}$$

$$[\![\texttt{closed()}]\!] = \lambda p.\{\texttt{closed}\}$$

"infeasible"

- For `flag` assignments:

$$[\![\texttt{flag = 0}]\!] = [\texttt{flag}{=}0 \rightarrow \bigcup_{p \in P} \mathit{JOIN}(v)(p),\ \texttt{flag}{\neq}0 \rightarrow \varnothing]$$

$$[\![\texttt{flag = }n]\!] = [\texttt{flag}{\neq}0 \rightarrow \bigcup_{p \in P} \mathit{JOIN}(v)(p),\ \texttt{flag}{=}0 \rightarrow \varnothing]$$

where n is a non-0 constant number

$$[\![\texttt{flag = }E]\!] = \lambda q. \bigcup_{p \in P} \mathit{JOIN}(v)(p)$$ for any other E

Relational constraints (2/2)

- For `assert` statements:

$$[\![\texttt{assert(flag)}]\!] = [\texttt{flag}\neq 0 \to \mathit{JOIN}(\mathrm{v})(\texttt{flag}\neq 0),\ \texttt{flag}=0 \to \emptyset]$$

$$[\![\texttt{assert(!flag)}]\!] = [\texttt{flag}=0 \to \mathit{JOIN}(\mathrm{v})(\texttt{flag}=0),\ \texttt{flag}\neq 0 \to \emptyset]$$

- For all other CFG nodes:

$$[\![\mathrm{v}]\!] = \mathrm{JOIN}(\mathrm{v}) = \lambda \mathrm{p}.\ \bigcup_{\mathrm{w}\in \mathit{pred}(\mathrm{v})} [\![\mathrm{w}]\!](\mathrm{p})$$

Generated constraints

$[\![\mathit{entry}]\!] = \lambda p.\{\texttt{closed}\}$
$[\![\texttt{condition}]\!] = [\![\mathit{entry}]\!]$
$[\![\texttt{assert(condition)}]\!] = [\![\texttt{condition}]\!]$
$[\![\texttt{open()}]\!] = \lambda p.\{\text{open}\}$
$[\![\texttt{flag = 1}]\!] = [\texttt{flag}\neq 0 \rightarrow \cup\, [\![\texttt{open()}]\!](p),\ \texttt{flag}=0 \rightarrow \emptyset]$
$[\![\texttt{assert(!condition)}]\!] = [\![\texttt{condition}]\!]$
$[\![\texttt{flag = 0}]\!] = [\texttt{flag}=0 \rightarrow \cup\, [\![\texttt{assert(!condition)}]\!](p),\ \texttt{flag}\neq 0 \rightarrow \emptyset]$
$[\![\texttt{...}]\!] = \lambda p.([\![\texttt{flag = 1}]\!](p) \cup [\![\texttt{flag = 0}]\!](p))$
$[\![\texttt{flag}]\!] = [\![\ldots]\!]$
$[\![\texttt{assert(flag)}]\!] = [\![\texttt{flag}\neq 0 \rightarrow [\![\texttt{flag}]\!](\texttt{flag}\neq 0),\ \texttt{flag}=0 \rightarrow \emptyset]$
$[\![\texttt{close()}]\!] = \lambda p.\{\texttt{closed}\}$
$[\![\texttt{assert(!flag)}]\!] = [\texttt{flag}=0 \rightarrow [\![\texttt{flag}]\!](\texttt{flag}=0),\ \texttt{flag}\neq 0 \rightarrow \emptyset]$
$[\![\mathit{exit}]\!] = \lambda p.([\![\texttt{close()}]\!](p) \cup [\![\texttt{assert(!flag)}]\!](p))$

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

  $[\![\texttt{open()}]\!] = \lambda p.\{\texttt{open}\}$

  into the still sound but more precise

  $[\![\texttt{open()}]\!] = \lambda p.\text{if } JOIN(v)(p)=\emptyset \text{ then } \emptyset \text{ else } \{\texttt{open}\}$