Coq a dit : fromage tranché ne peut cacher ses trous

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall

29 janvier 2016
JFLA - Saint-Malo

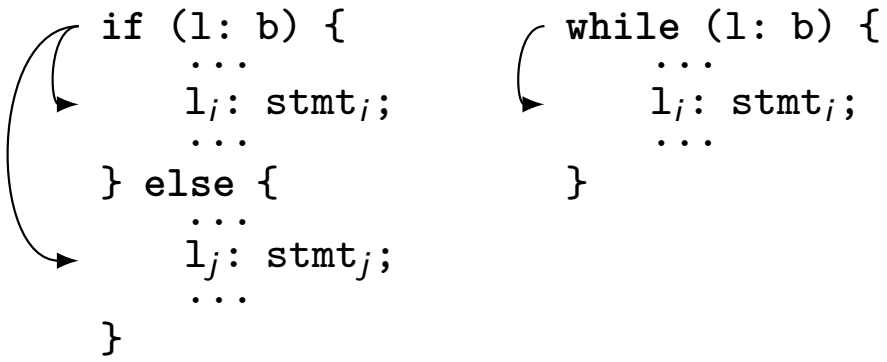
Static backward slicing (introduced by Weiser in 1981)

- simplifies a given program p w.r.t. a point of interest C (slicing criterion, typically a statement)
- removes irrelevant statements that do not impact C
- is based on control and data dependencies
- produces a simplified program q (slice)

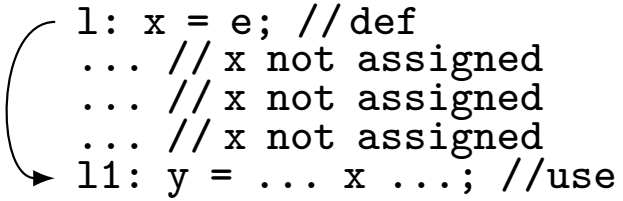
Definitions

- WHILE language: `skip`, `x:=e`, `if`, `while`, with usual expressions and fixed-size arrays, error-free

Control dependence



Data dependence



- Dependence-based slice q of p w.r.t. C: all statements on which one of the statements of C is (directly or indirectly) dependent
- Formally: $q = \{l \in p \mid l \rightarrow^* l', l' \in C\}$, where $\rightarrow = \xrightarrow{ctrl} \cup \xrightarrow{data}$

Classic soundness property

Theorem (Classic soundness property)

Let σ be an input state of p. Suppose that p halts on σ. Then q halts on σ and the executions of p and q on σ agree after each statement preserved in the slice on the variables that appear in this statement.

- Formalized with a trajectory-based semantics
- Equality of projection for the trajectories of p and q
- Does this result hold in the general case, i.e. in presence of errors and non-termination ? And more precisely:
  - If an error is found in q, is it also present in p ?
  - If there are no errors in q, what can be said about p ?

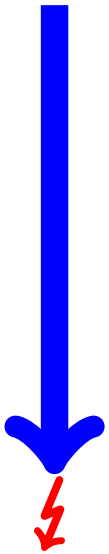
Assertions

- WHILE language: `skip`, `x:=e`, `if`, `while`, `assert`
- Assertions make runtime errors explicit
- Assertions protect all statements that may cause a runtime error

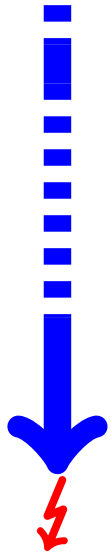
```
assert (l: N != 0);
l1: x = k/N;
```

```
assert (l: k < N);
l1: x = a[k];
```

Case 1: same error



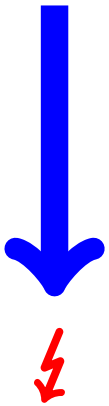
```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```



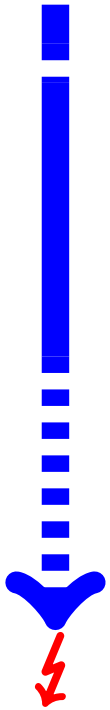
```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Example execution for test input: $N = 2$, $k = 4$

Case 2: error hidden by another error (not preserved)



```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```



```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Example execution for test input: $N = 0$, $k = 0$

Case 3: error hidden by a loop (not preserved)

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Example execution for test input: $N = 4$, $k = 0$

- Classic property does not hold in the general case
- Two solutions:
  - adding more dependencies
    -  Keep classic soundness property
    -  Bigger slices
  - keep same dependencies
    -  Keep slices small
    -  Another soundness property required

- Classic property does not hold in the general case
- Two solutions:
  - adding more dependencies
    -  Keep classic soundness property
    -  Bigger slices
  - **keep same dependencies**
    -  Keep slices small
    -  Another soundness property required

- In addition to (unmodified) control and data dependencies, assert dependence between an assertion and its protected statement

Assertion dependence

- Relaxed slice q of p w.r.t. C: all statements on which one of the statements of C is (directly or indirectly) dependent
- Formally: $\mathrm{q} = \{l \in p \mid l \rightarrow^* l', l' \in C\}$, where $\rightarrow = \xrightarrow{ctrl} \cup \xrightarrow{data} \cup \xrightarrow{assert}$

Soundness property of relaxed slicing

Theorem

The projection of the trajectory of p is a prefix of the projection of the trajectory of q. If the execution of p terminates normally, the projections are equal.

In the absence of errors and non-termination, the classic property holds.

Case 2: error hidden by another error (not preserved)

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Example execution for test input: $N = 0$, $k = 0$

Verification on relaxed slices

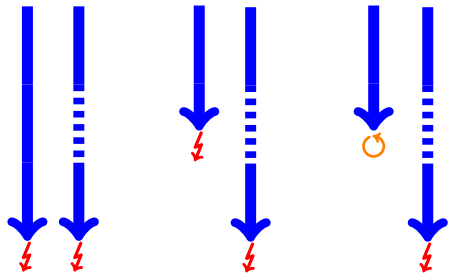
Theorem (No errors in the slice)

If there are no runtime errors in q, then there are none in p, in the statements preserved in q.

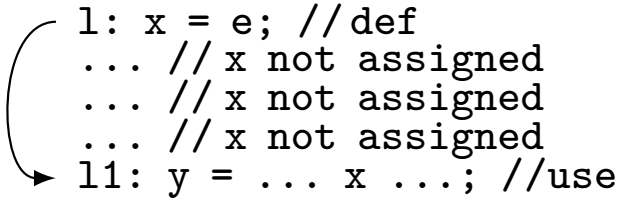
```
✓1  s1 = 0;
?2  s2 = 0;
✓3  i = 0;
✓4  while (i < N){
✓5    assert (i < N);
✓6    s1 = s1 + a[i];
✓7    i = i + k;
✓8  }
?9  j = 0;
?10 assert (k != 0);
?11 last = N/k;
?12 while (j <= last){
?13   assert (k*j < N);
?14   s2 = s2 + a[k*j];
?15   j = j + 1;
?16 }
✓17 assert (N != 0);
✓18 avg1 = s1 / N;
?19 assert (N != 0);
?20 avg2 = s2 / N;
?21 if(avg1 == avg2)
?22   print("equal");
```

Theorem (An error in the slice)

If there is a runtime error in q, then either the same error occurs in p, or another error or an infinite loop caused by a statement not preserved in q masks it.



Earlier definition of data dependence



Not easy to use in induction proofs

$data(p1; p2) = ?$

We introduce:

- $maydef(p) = \{(l, x) |\ l$ can be the last definition of x in $p\ \}$
- $mayuse(p) = \{(l, x) |\ x$ can be read at l before being defined $\}$
- $mustdef(p) = \{x |\ x$ is always defined by $p\ \}$
- $mayuse(p_1; p_2) = mayuse(p1) \cup (mayuse(p2) \setminus mustdef(p1))$
- $data(p1; p2) = data(p1) \cup data(p2) \cup \{(l, l') \in (p1, p2) | \exists x : (l, x) \in maydef(p1) \wedge (l', x) \in mayuse(p2)\}$

Conclusion:

- Relaxed slicing: soundness, yet slices of reasonable size
- A formal link about the presence or the absence of errors in the program and its slices
- Formalization in Coq (10,000 LOC)
- Certified slicer in Ocaml extracted from Coq

Future work:

- Consider a wider class of errors
- Extend the langage
- Measure the benefits of relaxed slicing for verification

References:
Léchenet, J-C., Kosmatov, N., Le Gall, P. : Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices. In FASE (2016). To appear.

Demo: extracted slicer on an example

Slice of the program in test_prog w.r.t. line 9.

$ ./test_slice.byte test_prog \[9\]

```
Original program:
0: ASSERT (not ((x1) <= (0))) && (not ((x1) == (0))) =>> 0;
1: ASSERT (not ((x5) <= (0))) && (not ((x5) == (0))) =>> 1;
WHILE 2: not ((x5) == (0)) DO
    3: x10 := 0;
    4: x11 := x1;
    WHILE 5: (x11) <= (x5) DO
        6: x10 := (x10) + (1);
        7: x11 := (x11) + (x1)
    END;
    8: x1 := x5;
    9: x5 := x11
END
```

```
Slice:
WHILE 2: not ((x5) == (0)) DO
    4: x11 := x1;
    WHILE 5: (x11) <= (x5) DO
        7: x11 := (x11) + (x1)
    END;
    8: x1 := x5;
    9: x5 := x11
END
```