

Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices

Nikolai Kosmatov

Joint work with Jean-Christophe Léchenet, Pascale Le Gall
presented at ETAPS/FASE 2016



Journées DigiCosme, le 12 avril 2016.

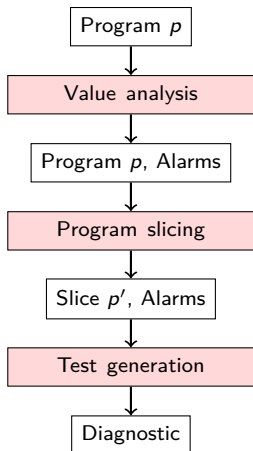
Context: Frama-C toolset



Frama-C: an extensible platform for analysis of C code

- Developed by CEA LIST and Inria
- Offers plugins for value analysis, proof, testing, slicing...
- Supports ACSL annotation language
- Facilitates combinations of individual analyzers
- <http://frama-c.com/>

SANTE: Combined analysis for runtime error detection



- Value analysis detects alarms
- Slicing reduces the program (w.r.t. one or several alarms)
- Testing (PathCrawler) is used to generate tests on a reduced program to diagnose alarms (after adding error branches to trigger errors)
- Diagnostic
 - bug if a counter-example found
 - if not, and all paths were explored, the alarm is safe
 - otherwise, unknown

Definition

Static backward slicing (introduced by Weiser in 1981)

- simplifies a given program p but preserves the behavior w.r.t. a point of interest C (**slicing criterion**, typically a statement)
- removes irrelevant statements that do not impact C
- produces a simplified program q (**slice**)

Example: a program and a slice

Check if a is divisible by b .

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
6 : }
7 : if (r != 0) {
8 :     res = 0;
9 : } else {
10 :    res = 1;
11 : }
```

Original program p

Example: a program and a slice

Check if a is divisible by b .

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
   }
6 : if (r != 0) {
7 :     res = 0;
   } else {
8 :     res = 1;
   }
```

euclidian division
of a by b

is the remainder
equal to 0 ?

Original program p

Example: a program and a slice

Check if a is divisible by b .

```
1 : q = 0;
```

```
2 : r = a;
```

```
3 : while (b <= r) {
```

```
4 :     q = q + 1;
```

```
5 :     r = r - b;
```

```
    }
```

```
6 : if (r != 0) {
```

```
7 :     res = 0;
```

```
    } else {
```

```
8 :     res = 1;
```

```
    }
```

Original program p

```
2 : r = a;
```

```
3 : while (b <= r) {
```

```
5 :     r = r - b;
```

```
    }
```

```
6 : if (r != 0) {
```

```
    } else {
```

```
8 :     res = 1;
```

```
    }
```

Slice q w.r.t. line 8

- Goal: preserve the behaviour of p w.r.t. line 8

Global motivation

- Traditional scope of slicing
 - software maintenance and re-engineering
 - compiler optimization
 - program differencing
 - debugging : understand an already found error
[Weiser, 1982] [Agrawal et al., 1993] [Hierons et al., 1999]

Our purpose

Conduct V&V on a slice instead of the initial program.

Dependence-based slicing

- **WHILE language**: skip, $x := e$, if, while

Control dependence

```

if (l: b) {
  ...
  l_then: stmt;
  ...
} else {
  ...
  l_else: stmt;
  ...
}

while (l: b) {
  ...
  l_body: stmt;
  ...
}

```

Data dependence

```

l_def: x = e; // def
... // x not assigned
... // x not assigned
... // x not assigned
l_use: y = ... x ...; // use

```

- **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 $\rightarrow = \xrightarrow{ctrl} \cup \xrightarrow{data}$

Example: computing a slice

Check if a is divisible by b .

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
   }
6 : if (r != 0) {
7 :     res = 0;
   } else {
8 :     res = 1;
   }
```

?

— control
— data

Original program p

Slice q w.r.t. line 8

Example: computing a slice

Check if a is divisible by b .

```
1 : q = 0;  
2 : r = a;  
3 : while (b <= r) {  
4 :     q = q + 1;  
5 :     r = r - b;  
    }  
6 : if (r != 0) {  
7 :     res = 0;  
    } else {  
8 :     res = 1;  
    }
```

?

— control
— data

Original program p

Slice q w.r.t. line 8

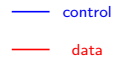
Example: computing a slice

Check if a is divisible by b .

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
   }
6 : if (r != 0) {
7 :     res = 0;
   } else {
8 :     res = 1;
   }
```



?



Original program p

Slice q w.r.t. line 8

Example: computing a slice

Check if a is divisible by b .

```
1 : q = 0;  
2 : r = a;  
3 : while (b <= r) {  
4 :     q = q + 1;  
5 :     r = r - b;  
6 : }  
7 : if (r != 0) {  
8 :     res = 0;  
9 : } else {  
10 :     res = 1;  
11 : }
```

Original program p

Slice q w.r.t. line 8

— control
— data

Example: computing a slice

Check if a is divisible by b .

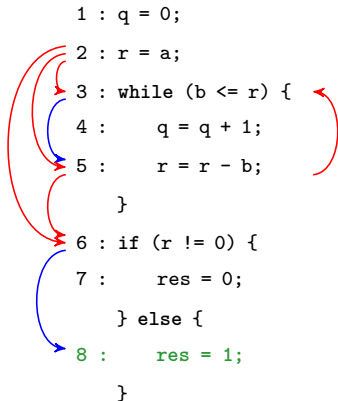
```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
6 : }
7 : if (r != 0) {
8 :     res = 0;
9 : } else {
10 :     res = 1;
11 : }
```

Original program p

Slice q w.r.t. line 8

Example: computing a slice

Check if a is divisible by b .



Original program p

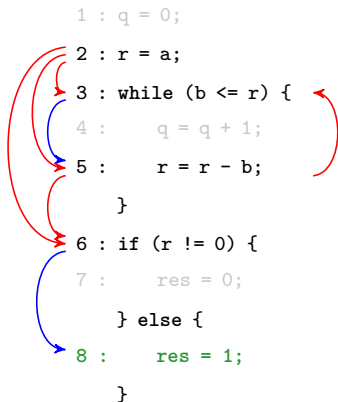
?

— control
— data

Slice q w.r.t. line 8

Example: computing a slice

Check if a is divisible by b .

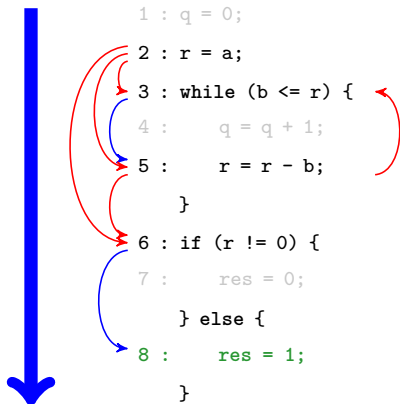


Original program p

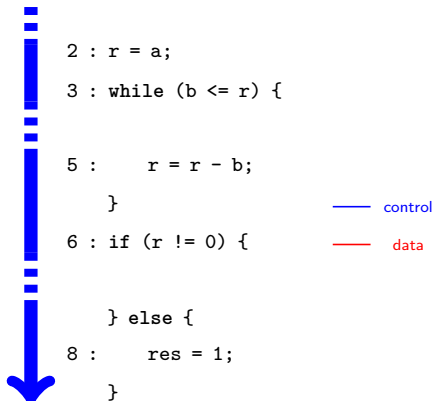
Slice q w.r.t. line 8

Example: computing a slice

Check if a is divisible by b .



Original program p



Slice q w.r.t. line 8

Projections of trajectories [Weiser, 1981]

How can we compare trajectories (i.e. executions) in p and in q ?
They should agree on preserved statements for preserved variables.

$$\mathcal{T}[[p]]\sigma = \langle \langle \begin{array}{l} (label, (q, r, res)) \\ (1, (0, 0, 0)) \\ (2, (0, 2, 0)) \\ (3, (0, 2, 0)) \\ (4, (1, 2, 0)) \\ (5, (1, 0, 0)) \\ (3, (1, 0, 0)) \\ (6, (1, 0, 0)) \\ (8, (1, 0, 1)) \end{array} \rangle \rangle$$

Trajectory in original program p

$$\mathcal{T}[[q]]\sigma = \langle \langle \begin{array}{l} (label, (r, res)) \\ (2, (2, 0)) \\ (3, (2, 0)) \\ (5, (0, 0)) \\ (3, (0, 0)) \\ (6, (0, 0)) \\ (8, (0, 1)) \end{array} \rangle \rangle$$

Trajectory in slice q

Projections of trajectories [Weiser, 1981]

How can we compare trajectories (i.e. executions) in p and in q ?
They should agree on preserved statements for preserved variables.

$$\mathcal{T}[[p]]\sigma = \langle (1, (0, 0, 0)) \rangle$$

$$\langle (2, (0, 2, 0)) \rangle$$

$$\langle (3, (0, 2, 0)) \rangle$$

$$\langle (4, (1, 2, 0)) \rangle$$

$$\langle (5, (1, 0, 0)) \rangle$$

$$\langle (3, (1, 0, 0)) \rangle$$

$$\langle (6, (1, 0, 0)) \rangle$$

$$\langle (8, (1, 0, 1)) \rangle$$

$$\rangle$$

Trajectory in original program p

$$\mathcal{T}[[q]]\sigma = \langle (label, (r, res)) \rangle$$

$$\langle (2, (2, 0)) \rangle$$

$$\langle (3, (2, 0)) \rangle$$

$$\langle (5, (0, 0)) \rangle$$

$$\langle (3, (0, 0)) \rangle$$

$$\langle (6, (0, 0)) \rangle$$

$$\langle (8, (0, 1)) \rangle$$

$$\rangle$$

Trajectory in slice q

Projections of trajectories [Weiser, 1981]

How can we compare trajectories (i.e. executions) in p and in q ?
They should agree on preserved statements for preserved variables.

$$\mathcal{T}[[p]]\sigma = \langle (1, (0, 0, 0)) \rangle$$

$$\langle (2, (0, 2, 0)) \rangle$$

$$\langle (3, (0, 2, 0)) \rangle$$

$$\langle (4, (1, 2, 0)) \rangle$$

$$\langle (5, (1, 0, 0)) \rangle$$

$$\langle (3, (1, 0, 0)) \rangle$$

$$\langle (6, (1, 0, 0)) \rangle$$

$$\langle (8, (1, 0, 1)) \rangle$$

$$\rangle$$

Trajectory in original program p

$$\mathcal{T}[[q]]\sigma = \langle (label, (r, res)) \rangle$$

$$\langle (2, (2, 0)) \rangle$$

$$\langle (3, (2, 0)) \rangle$$

$$\langle (5, (0, 0)) \rangle$$

$$\langle (3, (0, 0)) \rangle$$

$$\langle (6, (0, 0)) \rangle$$

$$\langle (8, (0, 1)) \rangle$$

$$\rangle$$

Trajectory in slice q

Classic soundness property

Let q be a slice of p .

Theorem (Classic soundness property, [Weiser, 1981])

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 (as an equality of projections)

Application to V&V

Does this result hold in the general case, i.e. in presence of errors and non-termination ?


Assertions to model errors

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


```
assert (1: N != 0);  
l1: x = k/N;
```

```
assert (1: 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");
```

Original program p 


```
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");
```

Slice q w.r.t. line 20Execution 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");
```

Original program p 

```
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");
```

Slice q w.r.t. line 18Execution 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");

```

Original program p 

```

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");

```

Slice q w.r.t. line 20Execution for test input: $N = 4, k = 0$

- Equality of projections does not hold in general, due to:
 - Non-termination [Ball et al.,1993] [Ranganath et al., 2007] [Amtoft, 2008]
 - Errors [Harman et al., 1995]
- Three possible directions:
 - change the semantics [Cartwright et al., 1989] [Giacobazzi et al., 2003] [Nestra, 2009] [Barraclough et al., 2010]
 - 😊 Extend the classic soundness property
 - 😞 Consider non-existing trajectories
 - add more dependencies [Ranganath et al., 2007]
 - 😊 Extend the classic soundness property
 - 😞 Bigger slices
All loops and assertions preceding the criterion will be systematically preserved
 - keep same kind of dependencies [Amtoft, 2008]
 - 😊 Keep slices small
 - ⚠️ Another soundness property required

- Equality of projections does not hold in general, due to:
 - Non-termination [Ball et al.,1993] [Ranganath et al., 2007] [Amtoft, 2008]
 - Errors [Harman et al., 1995]
- Three possible directions:
 - change the semantics [Cartwright et al., 1989] [Giacobazzi et al., 2003] [Nestra, 2009] [Barraclough et al., 2010]
 - 😊 Extend the classic soundness property
 - 😞 Consider non-existing trajectories
 - add more dependencies [Ranganath et al., 2007]
 - 😊 Extend the classic soundness property
 - 😞 Bigger slices
All loops and assertions preceding the criterion will be systematically preserved
 - **keep same kind of dependencies** [Amtoft, 2008]
 - 😊 Keep slices small
 - ⚠️ Another soundness property required

Relaxed dependence-based slicing

- In addition to (unmodified) control and data dependencies, we introduce assertion dependencies between assertions and their protected statements

Assertion dependence

`assert (1: N != 0);`
└─ `l1: x = k/N;`

`assert (1: k < N);`
└─ `l1: x = a[k];`

- 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: $q = \{l \in p \mid l \rightarrow^* l', l' \in C\}$,
where $\rightarrow = \xrightarrow{ctrl} \cup \xrightarrow{data} \cup \xrightarrow{assert}$

Coq proof assistant

- An interactive theorem prover
- Developed since 1984 by Inria
- Extraction of certified Coq (OCaml, Haskell)
- Used for: the four color theorem, CompCert...



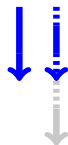
<https://coq.inria.fr/>

Soundness property of relaxed slicing

Let q be a slice of p .

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.




Corollary


The classic soundness property.



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");
```

Original program p 

```
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");
```

Slice q w.r.t. line 18Execution for test input: $N = 0, k = 0$

Verification on relaxed slices

Let q be a slice of p .

Theorem (No errors in the slice)

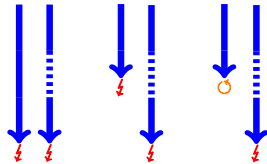
If there are no runtime errors in q , then there are none in p , in the statements preserved in q .

```
✓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 }
```

...

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.



Technical point

Control dependence

```
if (l: b) {  
  ...  
  lthen: stmt;  
  ...  
} else {  
  ...  
  lelse: stmt;  
  ...  
}
```

(Note: Blue arrows in the original image indicate control flow from the condition to the then/else branches.)

```
while (l: b) {  
  ...  
  lbody: stmt;  
  ...  
}
```

(Note: Blue arrows in the original image indicate the loop body feeding back into the condition.)

Data dependence

```
ldef: x = e; // def  
... // x not assigned  
... // x not assigned  
... // x not assigned  
luse: y = ... x ...; // use
```

(Note: Red arrows in the original image indicate data flow from the definition to the use.)

Problems with data dependence:

- Loops give an infinite number of unbounded executions
- Not induction-friendly
 - How to compute the data dependencies of the sequence of two programs, given the data dependencies of each one ?

Our solution:

- Reformulating data dependence in an induction-friendly way

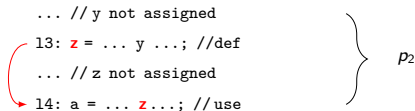
Reformulation of data dependence for $(p_1; p_2)$

```
11: x = e; // def  
... // x not assigned  
12: y = ... x...; // use  
... // y not assigned
```

} p_1

Reformulation of data dependence for $(p_1; p_2)$

```
... // y not assigned  
13: z = ... y ...; //def  
... // z not assigned  
14: a = ... z ...; // use
```



p_2

Reformulation of data dependence for $(p_1; p_2)$

```
11: x = e; //def
... // x not assigned
12: y = ... x ...; //use
... // y not assigned
... // y not assigned
13: z = ... y ...; //def
... // z not assigned
14: a = ... z ...; //use
```

p_1

p_2

Reformulation of data dependence for $(p_1; p_2)$

```
11: x = e; //def
... // x not assigned
12: y = ... x ...; //use
... // y not assigned

... // y not assigned
13: z = ... y ...; //def
... // z not assigned
14: a = ... z ...; //use
```

p_1

p_2

Reformulation of data dependence for $(p_1; p_2)$

```
11: x = e; //def
... // x not assigned
12: y = ... x ...; //use
... // y not assigned

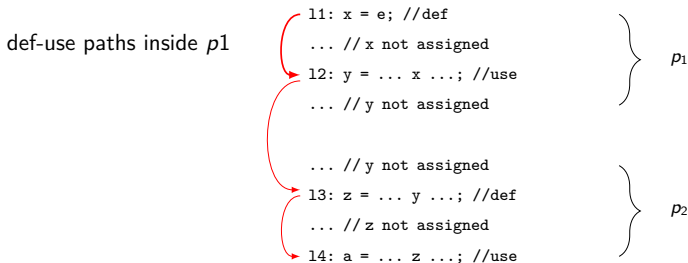
... // y not assigned
13: z = ... y ...; //def
... // z not assigned
14: a = ... z ...; //use
```

p_1

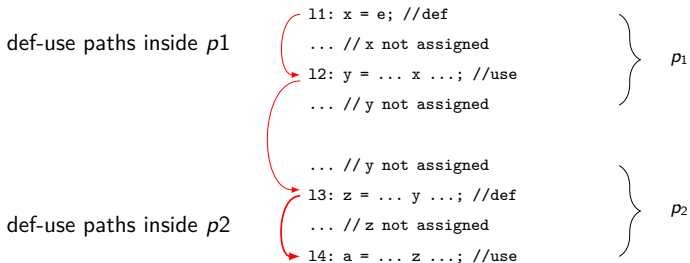
p_2

$data(p_1; p_2) = ?$

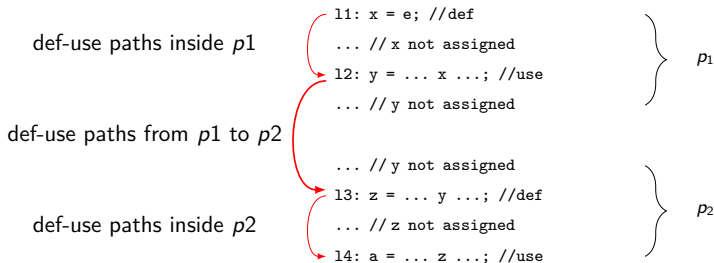
Reformulation of data dependence for $(p_1; p_2)$



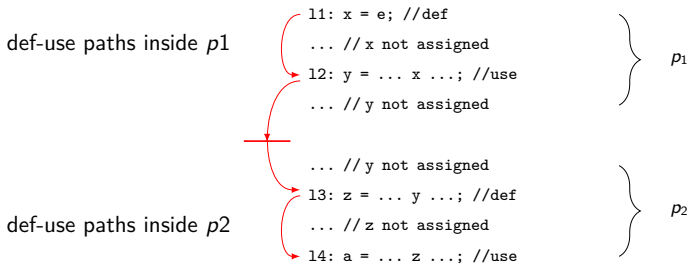
$$data(p_1; p_2) = \mathbf{data(p_1)} \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$ 

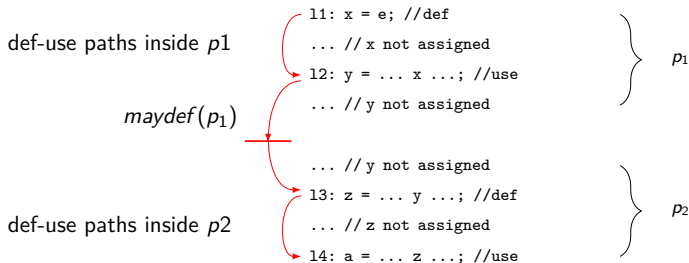
$$data(p_1; p_2) = data(p_1) \cup \mathbf{data(p_2)} \cup ?$$

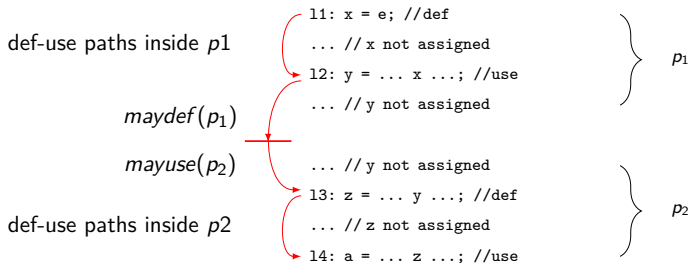
Reformulation of data dependence for $(p_1; p_2)$ 

$$data(p_1; p_2) = data(p_1) \cup data(p_2) \cup ?$$

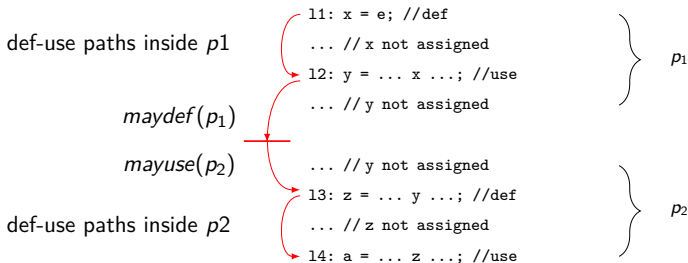
Reformulation of data dependence for $(p_1; p_2)$ 

$$data(p_1; p_2) = data(p_1) \cup data(p_2) \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$ 
$$maydef(p) = \{ (l, v) \mid l \text{ can be the last definition of } v \text{ in } p \}$$
$$data(p_1; p_2) = data(p_1) \cup data(p_2) \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$ 
$$maydef(p) = \{ (l, v) \mid l \text{ can be the last definition of } v \text{ in } p \}$$
$$mayuse(p) = \{ (l, v) \mid v \text{ can be read at } l \text{ before being redefined} \}$$
$$data(p_1; p_2) = data(p_1) \cup data(p_2) \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$



$maydef(p) = \{ (l, v) \mid l \text{ can be the last definition of } v \text{ in } p \}$

$mayuse(p) = \{ (l, v) \mid v \text{ can be read at } l \text{ before being redefined} \}$

$data(p_1; p_2) = data(p_1) \cup data(p_2) \cup$

$$\{ (l, l') \in (p_1, p_2) \mid \exists v : \begin{array}{l} (l, v) \in maydef(p_1) \wedge \\ (l', v) \in mayuse(p_2) \end{array} \}$$

Certified

- Certified intra- and inter-procedural program slicing in Isabelle/HOL
[Wasserab, 2011]
- A posteriori validation of program slicing integrated in CompCert (in Coq)
[Blazy et al., 2015]

Unlike the previous works, our purpose was:

- a machine-checked proof of the soundness of slicing in presence of errors and non-termination
- a justification of V&V on slices

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
 - expressions \rightarrow uninitialized variables \rightarrow ... \rightarrow ACSL ?
- Extend the language
 - WHILE \rightarrow unstructured control flow \rightarrow ... \rightarrow C ?
- Measure the benefits of relaxed slicing for verification

Demo: extracted certified 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
```