

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

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall



AFADL 2016, Besançon, 7 June 2016

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 : if (r != 0) {  
7 :     res = 0;  
} else {  
8 :     res = 1;  
}
```

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 : }  
7 : if (r != 0) {  
8 :     res = 0;  
9 : } else {  
10 :    res = 1;
```

Original program p

```
2 : r = a;  
3 : while (b <= r) {  
5 :     r = r - b;  
6 : }  
6 : if (r != 0) {  
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
 - program understanding
 - debugging : understand an already found error
[Weiser, 1982] [Agrawal et al., 1993] [Hierons et al., 1999]
- Frama-C: an extensible platform for analysis of C code
 - ACSL annotation language
 - Plugins for value analysis, proof, testing, slicing...
 - <http://frama-c.com/>

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

Data dependence

```
ldef: x = e; // def
... // x not assigned
... // x not assigned
... // x not assigned
luse: 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: $q = \{l \in p \mid l \xrightarrow{*} l', l' \in C\}$,
where $\xrightarrow{*} = \xrightarrow{ctrl} \cup \xrightarrow{data}$

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;  
8 : } else {  
9 :     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;  
}
```



?

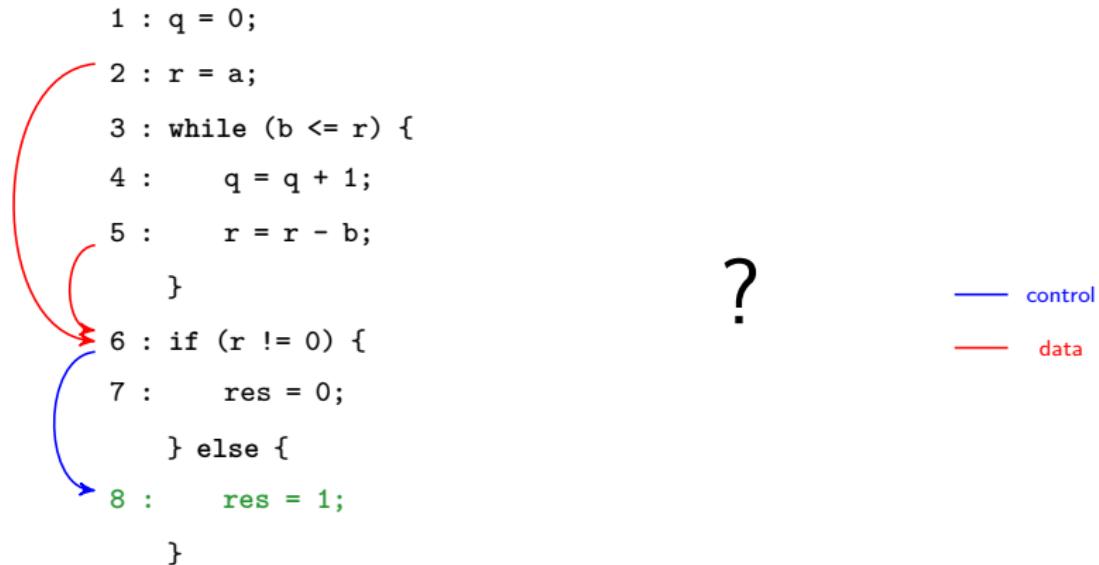
— control
— data

Original program p

Slice q w.r.t. line 8

Example: computing a slice

Check if a is divisible by b .



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 : }
```

?

— 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 : }  
7 : if (r != 0) {  
8 :     res = 0;  
9 : } else {  
10 :    res = 1;  
11 : }
```

The diagram illustrates the control and data flow in the program. Blue arrows represent control flow, showing the sequence of statements and loops. Red arrows represent data flow, specifically the propagation of the variable r from its assignment in line 2 through the loop in lines 3-5. A large red arrow points from line 2 to line 5. Another red arrow points from line 5 back to line 2, indicating the loop's iteration. A question mark is placed at the end of line 8, indicating the result of the slicing process.

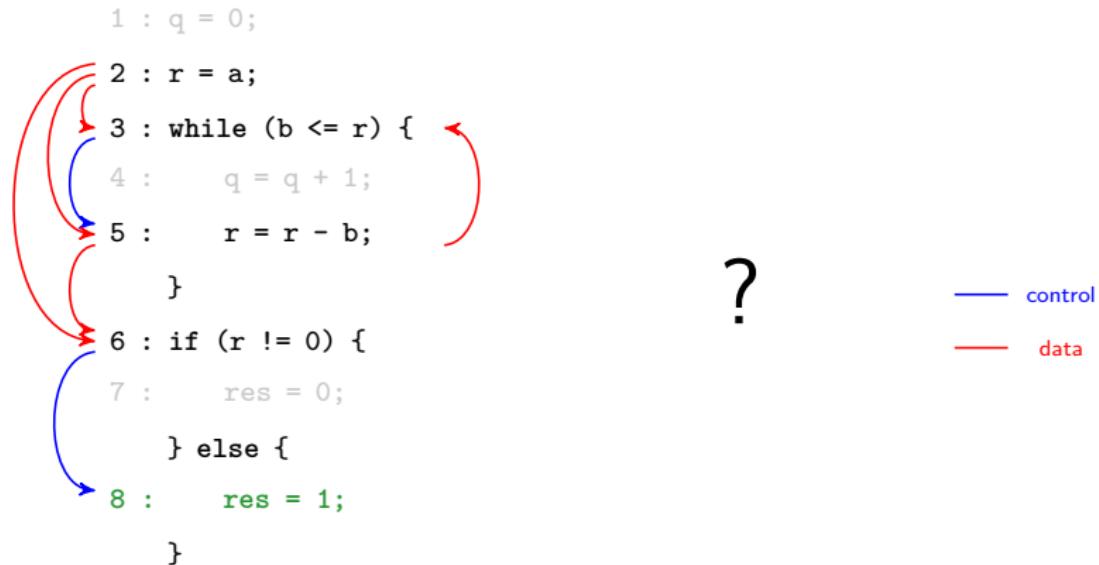
Original program p

Slice q w.r.t. line 8

control
data

Example: computing a slice

Check if a is divisible by b .

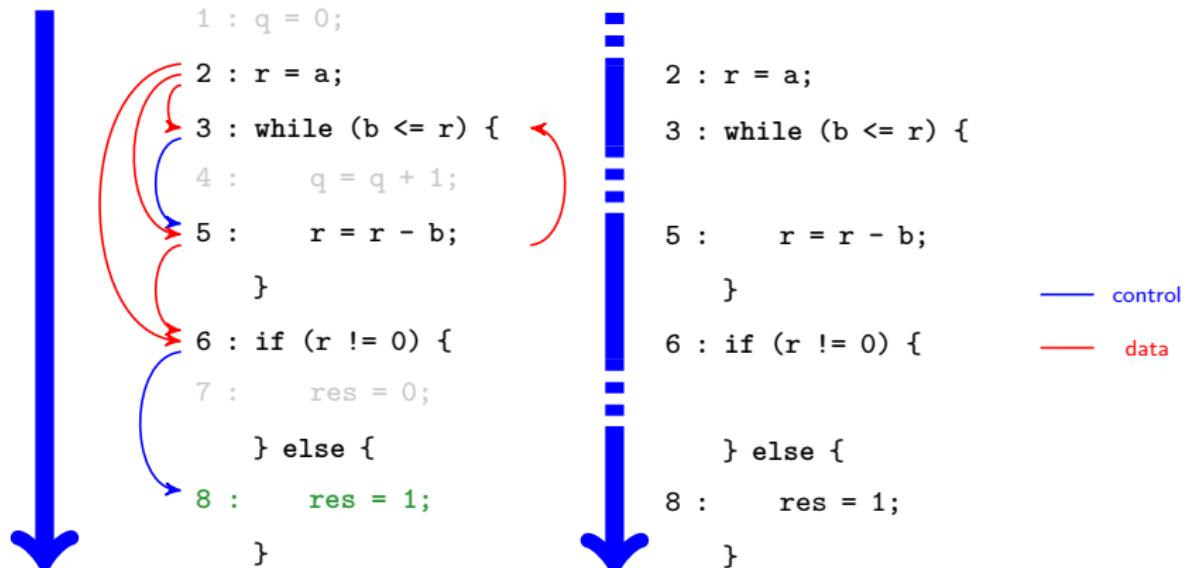


Original program p

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

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

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

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$(\text{label}, (q, r, res)) \\ \mathcal{T}[p]\sigma = \langle (1, (0, 0, 0))$$

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{aligned} (\text{label}, & (q, r, res)) \\ \mathcal{T}[p]\sigma = \langle (1, & (0, 0, 0)) \\ (2, & (0, 2, 0)) \end{aligned}$$

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{aligned} (\textit{label}, & (q, r, res)) \\ \mathcal{T}[p]\sigma = \langle (1, & (0, 0, 0)) \\ (2, & (0, 2, 0)) \\ (3, & (0, 2, 0)) \end{aligned}$$

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{aligned} (\textit{label}, & (q, r, res)) \\ \mathcal{T}[p]\sigma = \langle (1, & (0, 0, 0)) \\ (2, & (0, 2, 0)) \\ (3, & (0, 2, 0)) \\ (4, & (1, 2, 0)) \end{aligned}$$

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{aligned} & (\text{label}, (q, r, res)) \\ \mathcal{T}[p]\sigma = & \langle (1, (0, 0, 0)) \\ & (2, (0, 2, 0)) \\ & (3, (0, 2, 0)) \\ & (4, (1, 2, 0)) \\ & (5, (1, 0, 0)) \end{aligned}$$

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{aligned}\text{label, } & (q, r, res) \\ \mathcal{T}[p]\sigma = & \langle (1, (0, 0, 0)) \\ & (2, (0, 2, 0)) \\ & (3, (0, 2, 0)) \\ & (4, (1, 2, 0)) \\ & (5, (1, 0, 0)) \\ & \textcolor{red}{(3, (1, 0, 0))}\end{aligned}$$

}

Introduction to trajectories [Weiser, 1981]

```
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;  
    }
```

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{aligned} & (\text{label}, (q, r, res)) \\ \mathcal{T}[p]\sigma = & \langle (1, (0, 0, 0)) \\ & (2, (0, 2, 0)) \\ & (3, (0, 2, 0)) \\ & (4, (1, 2, 0)) \\ & (5, (1, 0, 0)) \\ & (3, (1, 0, 0)) \\ & \textcolor{red}{(6, (1, 0, 0))} \end{aligned}$$

Original program p

}

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

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

Introduction to trajectories [Weiser, 1981]

```
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

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

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

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0 \} :$$

2 : `r = a;`
3 : `while (b <= r) {`

The trajectory of q on σ is:

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

5 : `r = r - b;`
6 : `if (r != 0) {`

} else {
8 : `res = 1;`
}

}

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\mathcal{T}[q]\sigma = \langle (2, (2, 0)) \rangle$$

```
5 :     r = r - b;  
6 : if (r != 0) {  
    }
```

```
} else {  
8 :     res = 1;  
}
```

}

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\mathcal{T}[q]\sigma = \langle (2, (r, res))
(3, (2, 0))$$

```
5 : r = r - b;  
} }  
6 : if (r != 0) {  
} else {  
8 : res = 1;  
}  
}
```

}

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\mathcal{T}[q]\sigma = \langle (2, (2, 0)), (3, (2, 0)), (5, (0, 0)) \rangle$$

```
5 : r = r - b;  
} }  
6 : if (r != 0) {  
} else {  
8 : res = 1;  
}
```

}

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\mathcal{T}[q]\sigma = \langle (2, (2, 0)), (3, (2, 0)), (5, (0, 0)), (3, (0, 0)) \rangle$$

```
5 : r = r - b;  
} }  
6 : if (r != 0) {  
} else {  
8 : res = 1;  
}
```

}

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\begin{aligned}\mathcal{T}[q]\sigma = & \langle (2, (2, 0)) \\ & (3, (2, 0)) \\ & (5, (0, 0)) \\ & (3, (0, 0)) \\ & \textcolor{red}{(6, (0, 0))} \\ & \rangle\end{aligned}$$

```
5 : r = r - b;  
}   
6 : if (r != 0) {  
} else {  
8 : res = 1;  
}
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\mathcal{T}[q]\sigma = \langle (2, (2, 0)) \\ (3, (2, 0)) \\ (5, (0, 0)) \\ (3, (0, 0)) \\ (6, (0, 0)) \\ (8, (0, 1)) \\ \rangle$$

```
5 : r = r - b;  
} }  
6 : if (r != 0) {  
} else {  
8 : res = 1;  
}
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{ a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0 \} :$$

```
2 : r = a;  
3 : while (b <= r) {
```

The trajectory of q on σ is:

$$\begin{aligned}\mathcal{T}[q]\sigma = \langle & (2, (2, 0)) \\ & (3, (2, 0)) \\ & (5, (0, 0)) \\ & (3, (0, 0)) \\ & (6, (0, 0)) \\ & (8, (0, 1)) \\ & \rangle\end{aligned}$$

```
5 : r = r - b;  
} }  
6 : if (r != 0) {  
} else {  
8 : res = 1;  
}
```

Slice q w.r.t. line 8

Introduction to projections [Weiser, 1981]

How to compare both trajectories ? They should agree on preserved statements for preserved variables.

$$\mathcal{T}[\![p]\!]\sigma = \langle (1, (0, 0, 0)) | (2, (0, 2, 0)), (3, (0, 2, 0)), (4, (1, 2, 0)), (5, (1, 0, 0)), (3, (1, 0, 0)), (6, (1, 0, 0)), (8, (1, 0, 1)) \rangle$$
$$\mathcal{T}[\![q]\!]\sigma = \langle (2, (2, 0)), (3, (2, 0)), (5, (0, 0)), (3, (0, 0)), (6, (0, 0)), (8, (0, 1)) \rangle$$

Introduction to projections [Weiser, 1981]

How to compare both trajectories ? They should agree on preserved statements for preserved variables.

$$\mathcal{T}[\![p]\!]\sigma = \langle (1, (0, 0, 0)) | (2, (0, 2, 0)), (3, (0, 2, 0)), (4, (1, 2, 0)), (5, (1, 0, 0)), (3, (1, 0, 0)), (6, (1, 0, 0)), (8, (1, 0, 1)) \rangle$$
$$\mathcal{T}[\![q]\!]\sigma = \langle (2, (2, 0)) | (3, (2, 0)), (5, (0, 0)), (3, (0, 0)), (6, (0, 0)), (8, (0, 1)) \rangle$$

Introduction to projections [Weiser, 1981]

How to compare both trajectories ? They should agree on preserved statements for preserved variables.

$$\mathcal{T}[\![p]\!]\sigma = \langle (1, (0, 0, 0)), (2, (0, 2, 0)), (3, (0, 2, 0)), (4, (1, 2, 0)), (5, (1, 0, 0)), (3, (1, 0, 0)), (6, (1, 0, 0)), (8, (1, 0, 1)) \rangle$$
$$\mathcal{T}[\![q]\!]\sigma = \langle (2, (2, 0)), (3, (2, 0)), (5, (0, 0)), (3, (0, 0)), (6, (0, 0)), (8, (0, 1)) \rangle$$

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: $Proj_{L(q)}(\mathcal{T}[\![p]\!]\sigma) = Proj_{L(q)}(\mathcal{T}[\![q]\!]\sigma)$

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 (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");

```

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 20

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");

```

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 18

Execution for test input: $N = 0, k = 0$

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

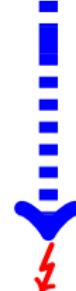


```

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");

```

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 20

Execution 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] [Barracough 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] [Barracough 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 (l: N != 0);
    ↳ l1: x = k/N;                                assert (l: 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: $q = \{I \in p \mid I \xrightarrow{*} I', I' \in C\}$,
where $\xrightarrow{*} = \xrightarrow{ctrl} \cup \xrightarrow{data} \cup \xrightarrow{assert}$

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...



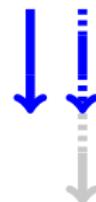
<https://coq.inria.fr/>

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 18

Execution 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 .



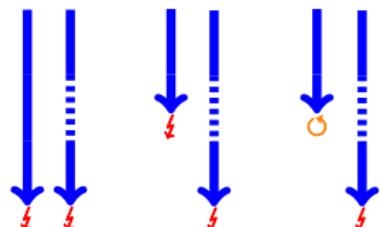
```

✓ 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;
    ...
}
  
```

Data dependence

```

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

Problems with data dependence:

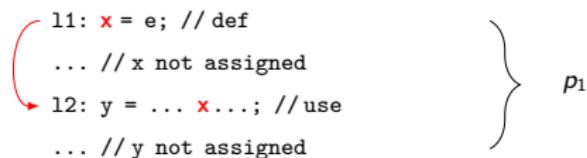
- Loops give an infinite number of unbounded executions
- Not induction-friendly
 - How to compute the data dependencies of the sequence of two programs, given the data dependencies of each one ?

Our solution:

- Reformulating data dependence in an induction-friendly way

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

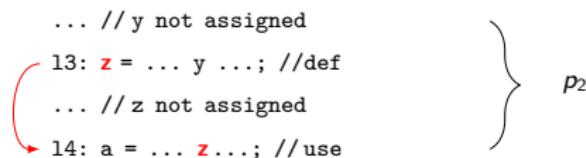
```
11: x = e; // def
... // x not assigned
12: y = ... x ...; // use
... // y not assigned
```



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
```



The diagram illustrates the reformulation of data dependence for parallel programs p_1 and p_2 . It shows two code snippets with annotations:

- Left side (p1):** Lines 13 and 14 are grouped by a red curly brace. Line 13 is labeled "13: z = ... y ...; //def" and line 14 is labeled "14: a = ... z ...; //use". Both lines have comments indicating they are not assigned to variables.
- Right side (p2):** Lines 13 and 14 are grouped by a black curly brace. Line 13 is labeled "13: z = ... y ...; //def" and line 14 is labeled "14: a = ... z ...; //use". Both lines have comments indicating they are not assigned to variables.

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

The diagram illustrates the reformulation of data dependence for two parallel programs, p_1 and p_2 . The code is presented in two columns, each enclosed in curly braces.

Program p_1 :

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

Program p_2 :

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

Red arrows point from the 'not assigned' annotations in p_1 to the corresponding lines in p_2 , indicating that the data dependencies between them have been resolved or redefined.

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
```

The diagram illustrates the reformulation of data dependence for the parallel composition $(p_1; p_2)$. It shows two sets of statements, p_1 and p_2 , with dependencies indicated by red arrows.

- Set p_1 :** Contains statements 11 and 12.
 - Statement 11: `x = e; //def`
 - Statement 12: `y = ... x ...; //use`
- Set p_2 :** Contains statements 13 and 14.
 - Statement 13: `z = ... y...; //def`
 - Statement 14: `a = ... z ...; //use`

Dependencies are shown as red arrows:

- An arrow points from statement 11 to statement 12, indicating that the value of `x` defined in 11 is used in 12.
- An arrow points from statement 13 to statement 14, indicating that the value of `y` defined in 13 is used in 14.

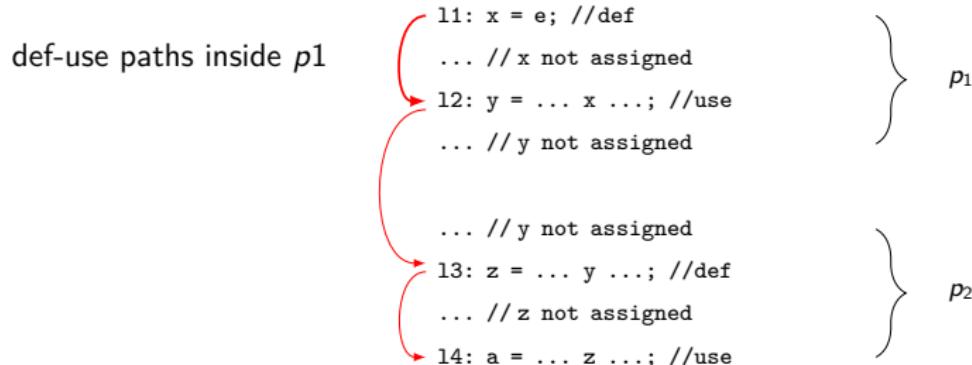
Reformulation of data dependence for $(p_1; p_2)$

```
11: x = e; //def
    ...
    12: y = ... x ...; //use
    ...
    ...
    13: z = ... y ...; //def
    ...
    14: a = ... z ...; //use
```

The diagram illustrates the data dependence between two parallel processes, p_1 and p_2 . Process p_1 contains statements 11 and 12. Statement 11 defines variable x and statement 12 uses it. Process p_2 contains statements 13 and 14. Statement 13 defines variable y and statement 14 uses it. Red arrows indicate the flow of data from the definition of x to its use in p_1 , and from the definition of y to its use in p_2 .

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

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



$$data(p_1; p_2) = \mathbf{data(p1)} \cup ?$$

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

def-use paths inside p_1

```

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

```

p_1

def-use paths inside p_2

```

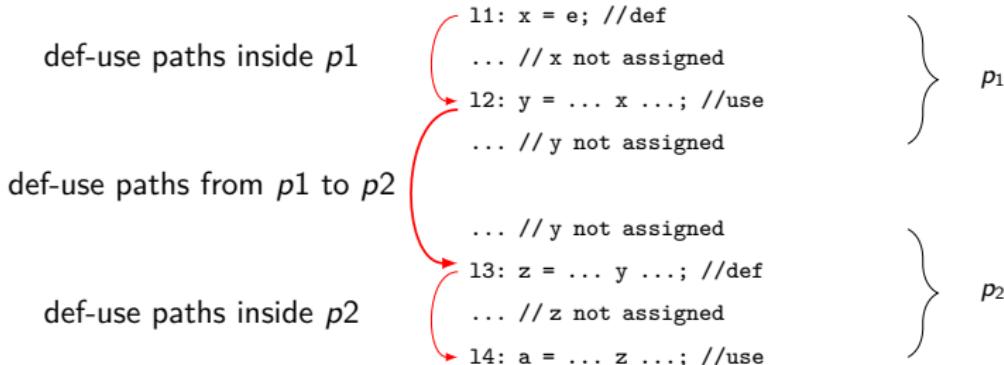
... //y not assigned
13: z = ... y ...; //def
... //z not assigned
14: a = ... z ...; //use

```

p_2

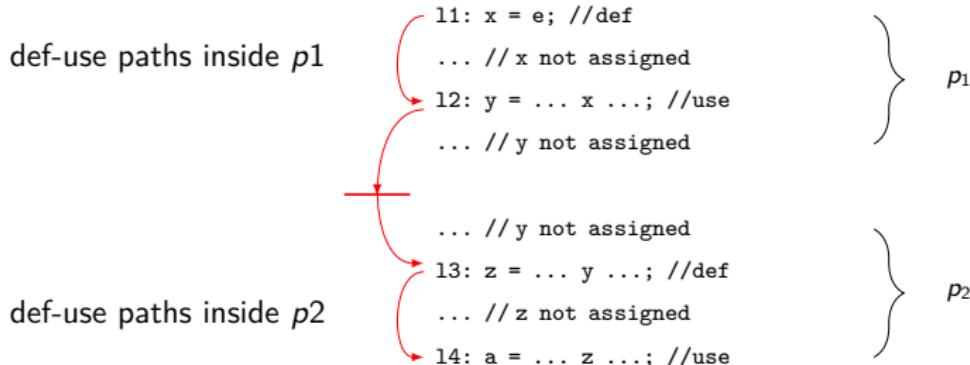
$$data(p_1; p_2) = data(p_1) \cup \text{data}(p_2) \cup ?$$

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



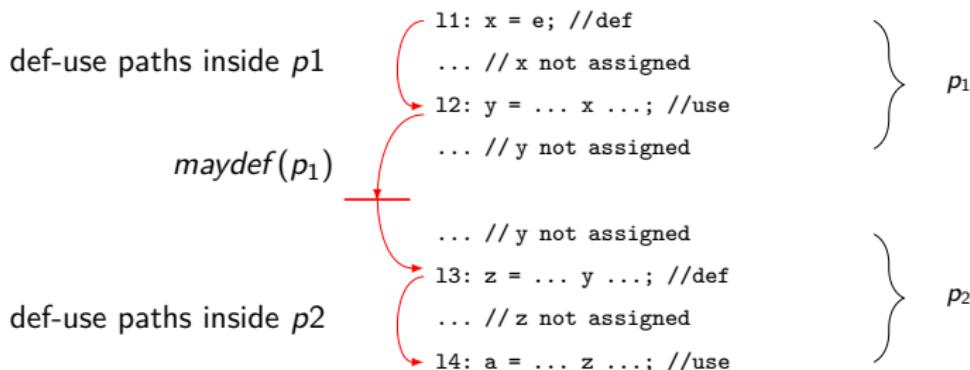
$$data(p_1; p_2) = data(p_1) \cup data(p_2) \cup ?$$

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



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

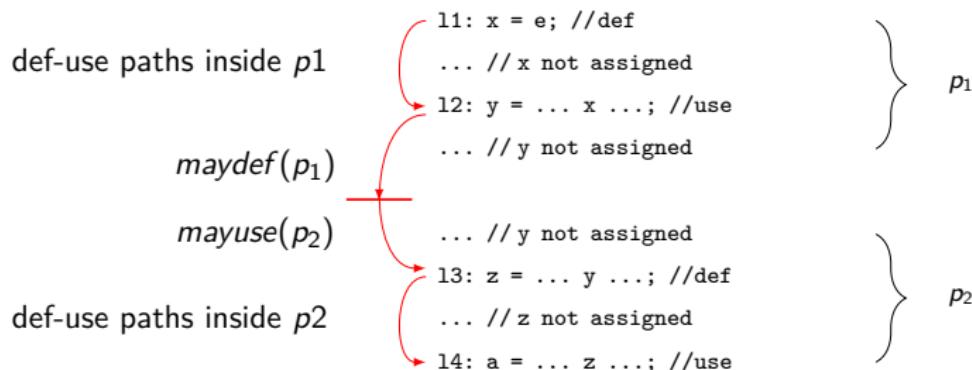
Reformulation of data dependence for $(p_1; p_2)$



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

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

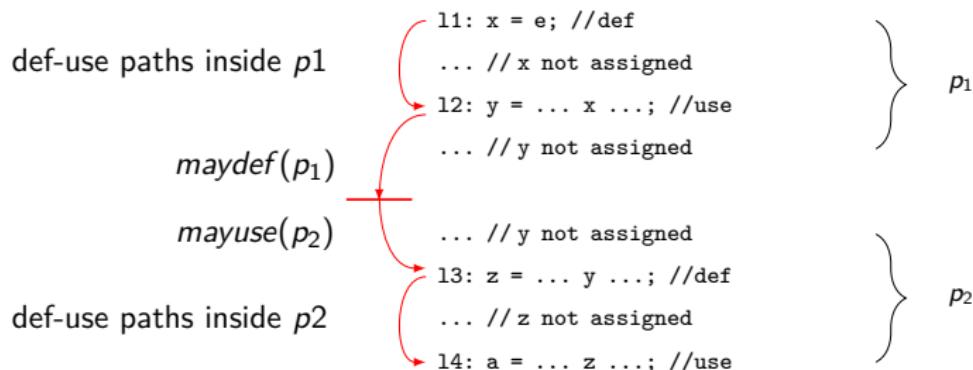
Reformulation of data dependence for $(p_1; p_2)$



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

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

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

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

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 → uninitialized variables → ... → ACSL ?
- Extend the language
 - WHILE → unstructured control flow → ... → 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
```