Certified Algorithms for Program Slicing

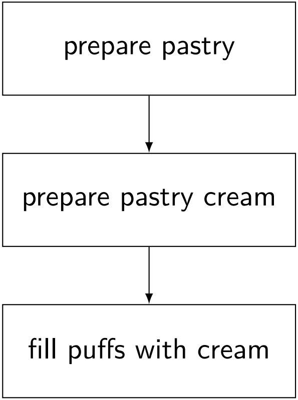
PhD thesis defense

Jean-Christophe Léchenet

July 19th, 2018
Palaiseau

Let's cook pastry cream

The recipe from my cookbook



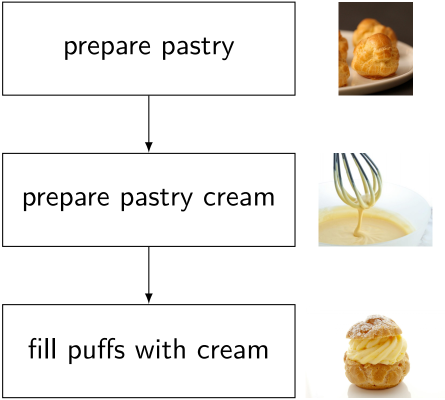
prepare pastry

prepare pastry cream

fill puffs with cream

Cream Puff Recipe

The recipe from my cookbook



prepare pastry

prepare pastry cream

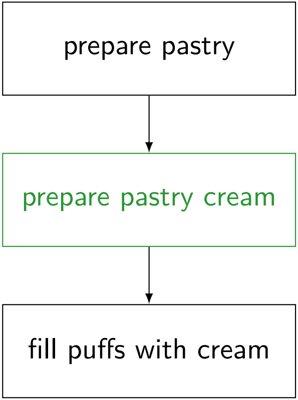
fill puffs with cream

Cream Puff Recipe

?

Pastry Cream Recipe

The recipe from my cookbook



prepare pastry

prepare pastry cream

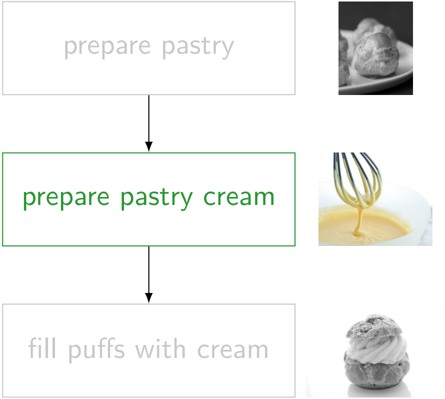
fill puffs with cream

?

Cream Puff Recipe

Pastry Cream Recipe

The recipe from my cookbook



prepare pastry

prepare pastry cream

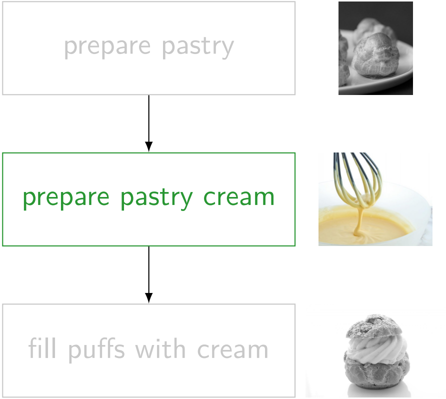
fill puffs with cream

?

Cream Puff Recipe

Pastry Cream Recipe

The recipe from my cookbook



prepare pastry

prepare pastry cream

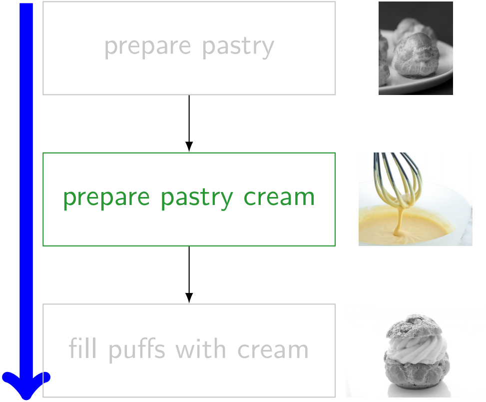
fill puffs with cream

Cream Puff Recipe

prepare pastry cream

Pastry Cream Recipe

The recipe from my cookbook

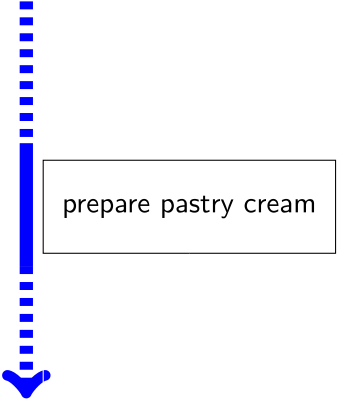


prepare pastry

prepare pastry cream

fill puffs with cream

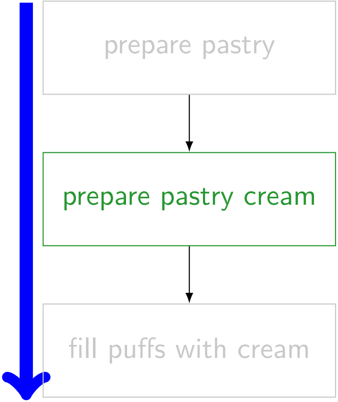
Cream Puff Recipe



prepare pastry cream

Pastry Cream Recipe

The recipe from my cookbook

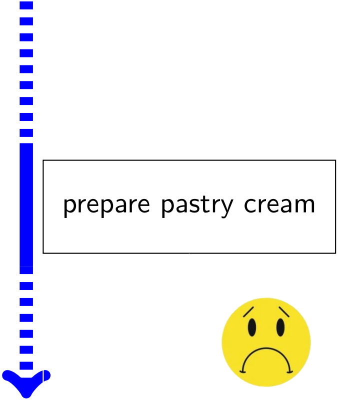


prepare pastry

prepare pastry cream

fill puffs with cream

Cream Puff Recipe



prepare pastry cream

Pastry Cream Recipe

My grandmother's recipe

mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

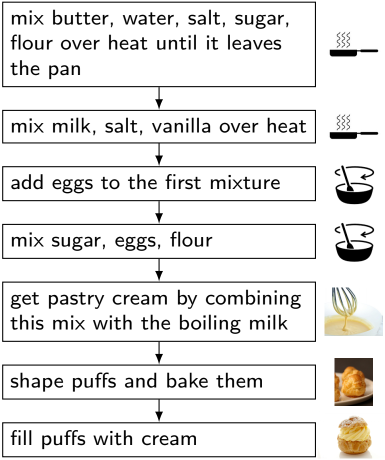
get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

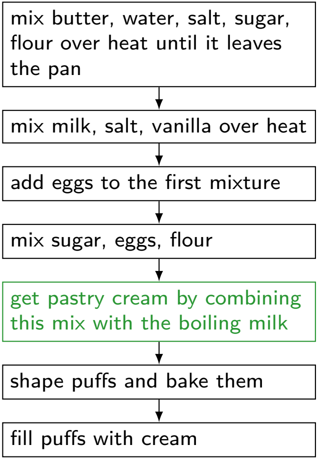
fill puffs with cream

Cream Puff Recipe

?

Pastry Cream Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

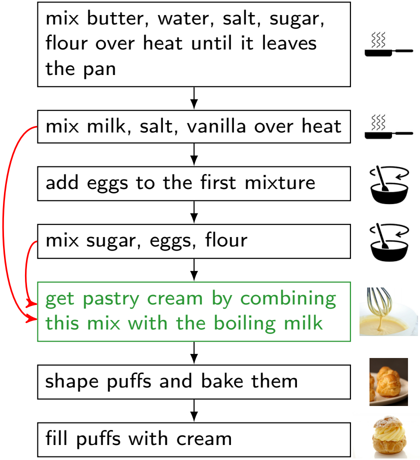
fill puffs with cream

?

Cream Puff Recipe

Pastry Cream Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

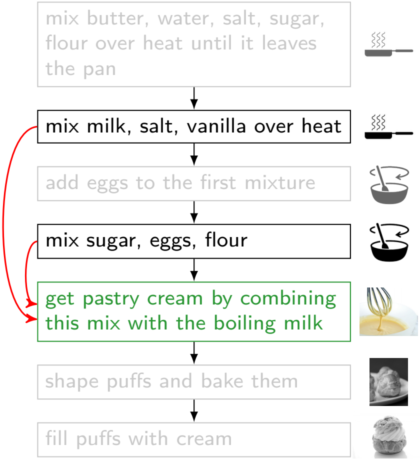
fill puffs with cream

Cream Puff Recipe

?

Pastry Cream Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

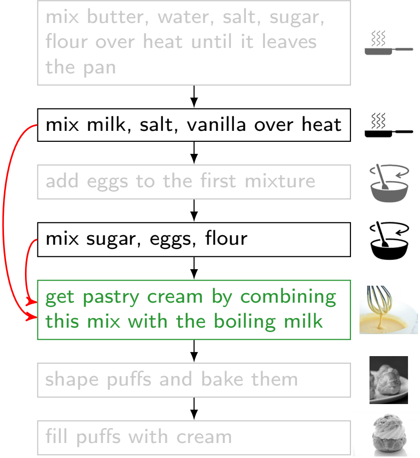
fill puffs with cream

Cream Puff Recipe

?

Pastry Cream Recipe

My grandmother’s recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe

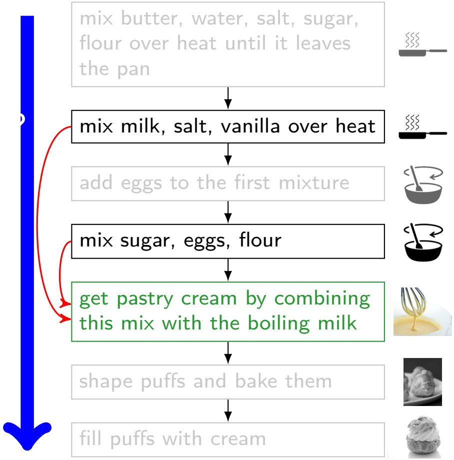
mix milk, salt, vanilla over heat

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

Pastry Cream Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

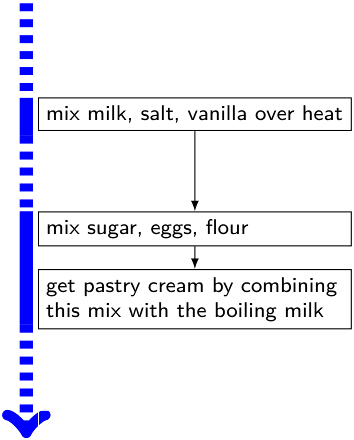
mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe



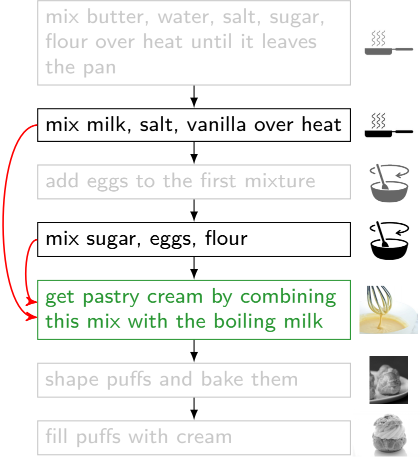
mix milk, salt, vanilla over heat

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

Pastry Cream Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe

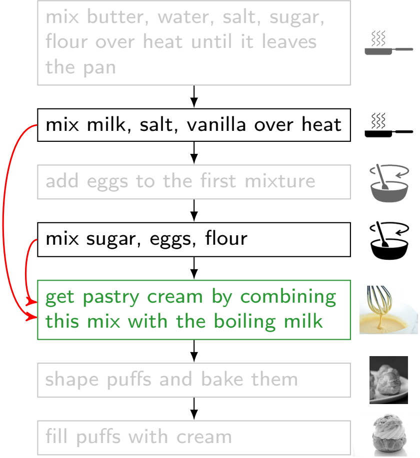
mix milk, salt, vanilla over heat

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

Pastry Cream Recipe

My grandmother's recipe



mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

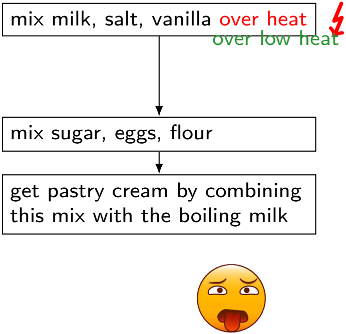
mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe



mix milk, salt, vanilla over heat

over low heat

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

Pastry Cream Recipe

My grandmother's recipe

?

mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

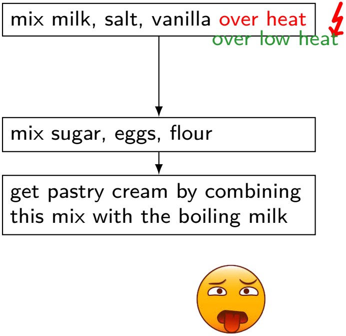
mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe



mix milk, salt, vanilla ~~over heat~~ ↯ over low heat

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

Pastry Cream Recipe

My grandmother's recipe

?

mix butter, water, salt, sugar, flour over heat until it leaves the pan

mix milk, salt, vanilla over heat

add eggs to the first mixture

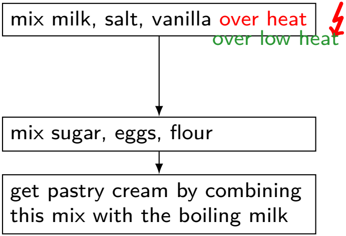
mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

shape puffs and bake them

fill puffs with cream

Cream Puff Recipe



mix milk, salt, vanilla over heat

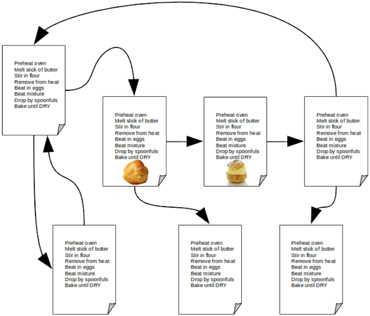
over low heat

mix sugar, eggs, flour

get pastry cream by combining this mix with the boiling milk

Pastry Cream Recipe

A recipe from the Internet



?

(a) Cream Puff Recipe

(b) Pastry Cream Recipe

Program Slicing vs. Recipe Extraction

- Program slicing $\simeq$ extraction of a sub-recipe
    - Extraction of the important steps w.r.t. a given goal
- Properties considered in this work:
    - Interpretation of errors
    - Handling of complex structures
    - Efficient
    - Provably correct

Outline

Context: Static Backward Slicing

Slicing in the Presence of Errors

An Algorithm for Arbitrary Control Dependence

An Optimized Algorithm

Conclusion

Outline

Context: Static Backward Slicing

Slicing in the Presence of Errors

An Algorithm for Arbitrary Control Dependence

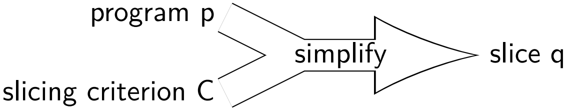
An Optimized Algorithm

Conclusion

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: test if b divides a $(a, b > 0)$

euclidean division of a by b

```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 1;
5 :     r = r - b;
    }
```

is the remainder equal to 0 ?

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

Original program p

?

Slice q w.r.t. line 8

Example: test if b divides a $(a, b > 0)$

euclidean division of a by b

is the remainder equal to 0 ?

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

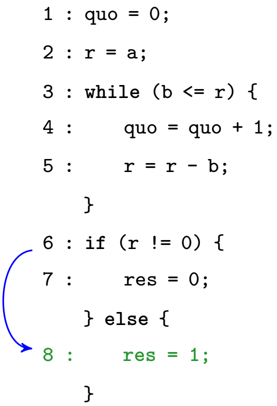
?

Original program p

Slice q w.r.t. line 8

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

Example: test if b divides a $(a, b > 0)$



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

?

— control

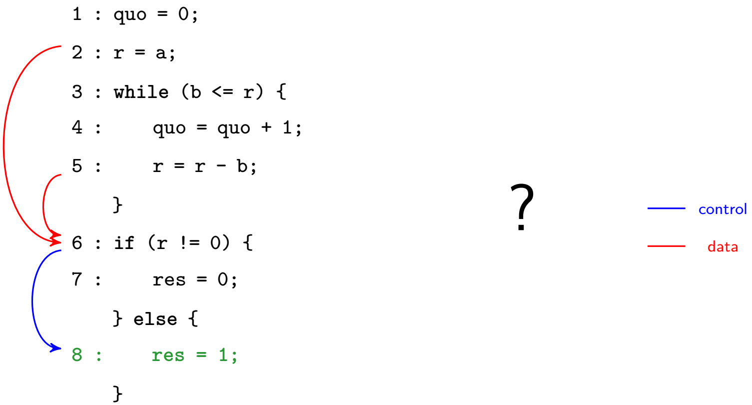
— data

Original program p

Slice q w.r.t. line 8

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

Example: test if b divides a $(a, b > 0)$



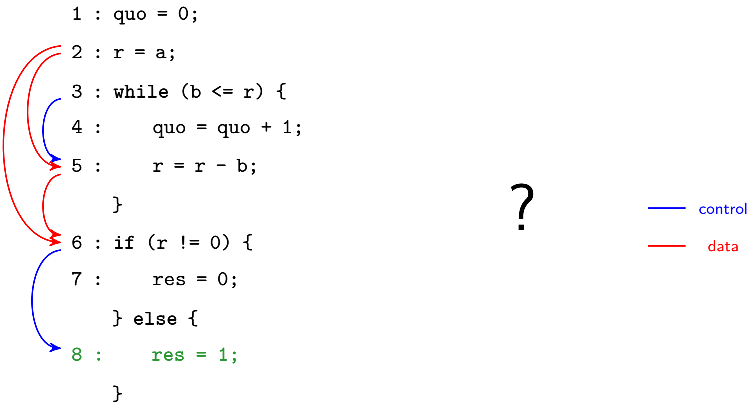
```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

Slice q w.r.t. line 8

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

Example: test if b divides a $(a, b > 0)$



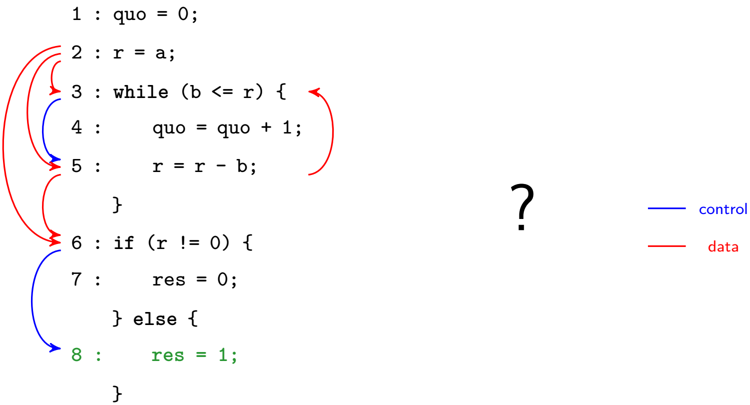
```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

Slice q w.r.t. line 8

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

Example: test if b divides a $(a, b > 0)$



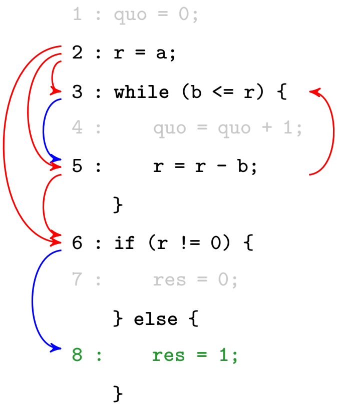
```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

Slice q w.r.t. line 8

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

Example: test if b divides a $(a, b > 0)$



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

?

control

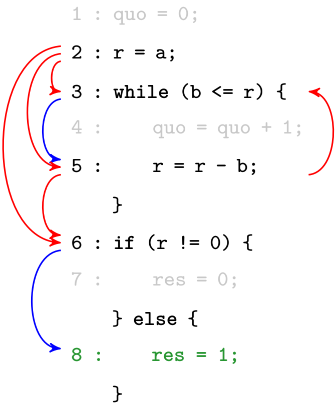
data

Original program p

Slice q w.r.t. line 8

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

Example: test if b divides a $(a, b > 0)$



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

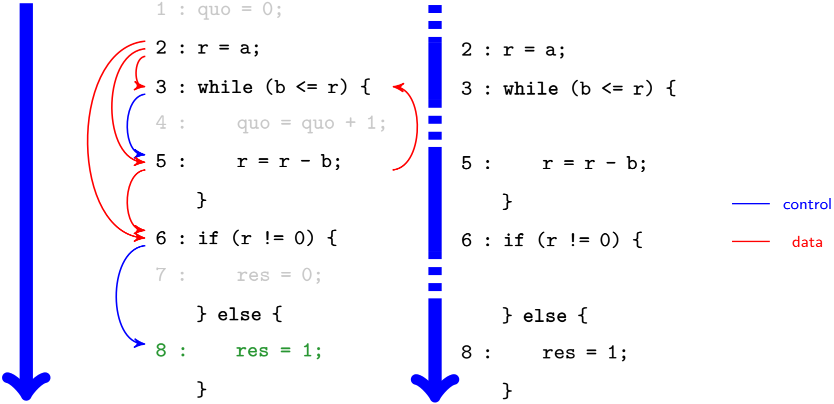
Original program p

```
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

Example: test if b divides a $(a, b > 0)$



```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 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

Outline

Context: Static Backward Slicing

Slicing in the Presence of Errors

An Algorithm for Arbitrary Control Dependence

An Optimized Algorithm

Conclusion

Motivation

Let q be a slice of p.

- 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 ?

Classic soundness property

Let p a program without failing instructions and q a slice of p.

Theorem (Classic soundness property, [Weiser, 1981] [Reps et al, 1989])

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.

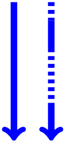
Formalized with a trajectory-based semantics as an equality of projections

Classic soundness property

Let p a program **without failing instructions** and q a slice of p.

Theorem (Classic soundness property, [Weiser, 1981] [Reps et al, 1989])

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 verification

Does this result also hold in the presence of errors and non-termination ?

Modeling

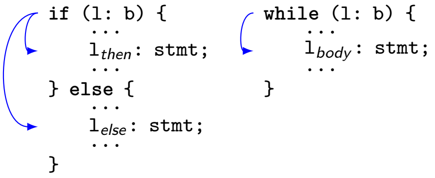
- WHILE language: `skip`, `x:=e`, `if`, `while`, `assert`
- Assertions make runtime errors explicit
- Assertions protect all statements that may cause a runtime error

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

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

Dependence-based slicing

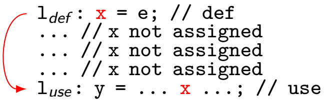
Control dependence



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

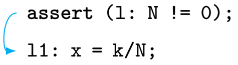
```
while (l: b) {
    ...
    l_body: stmt;
    ...
}
```

Data dependence

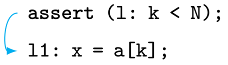


```
l_def: x = e; // def
... // x not assigned
... // x not assigned
... // x not assigned
l_use: y = ... x ...; // use
```

Assertion dependence



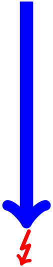
```
assert (l: N != 0);

l1: x = k/N;
```



```
assert (l: k < N);

l1: x = a[k];
```

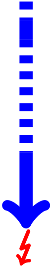
- Dependence-based slice q of p w.r.t. C: all statements on which one of the statements of C is (directly or indirectly) dependent
- Formally: $q = \{l \in p \mid l \to^* l', l' \in C\}$, where $\to = \xrightarrow{ctrl} \cup \xrightarrow{data} \cup \xrightarrow{assert}$

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");
```

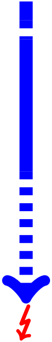
Slice *q* w.r.t. line 20

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");
```

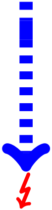
Slice q w.r.t. line 18

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] [Allen et al., 2003] [Rival, 2005]
- 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
    - ⚠ A weaker soundness property required

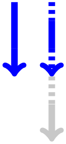
- Equality of projections does not hold in general, due to:
  - Non-termination [Ball et al.,1993] [Ranganath et al., 2007] [Amtoft, 2008]
  - Errors [Harman et al., 1995] [Allen et al., 2003] [Rival, 2005]
- Three possible directions:
  - change the semantics [Cartwright et al., 1989] [Giacobazzi et 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
    - ⚠ A weaker soundness property required: relaxed slicing

Soundness property of relaxed slicing

Let q be a slice of p.

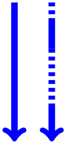
Theorem

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.

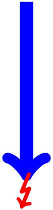


Corollary

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.

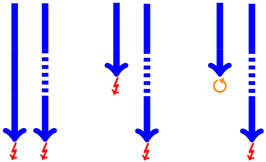
Theorem (No errors in the slice)

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

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.

```
✓ s1 = 0;
? s2 = 0;
✓ i = 0;
✓ while (i < N){
✓   assert (i < N);
✓   s1 = s1 + a[i];
✓   i = i + k;
✓ }
? j = 0;
? assert (k != 0);
? last = N/k;
? while (j <= last){
?   assert (k*j < N);
?   s2 = s2 + a[k*j];
?   j = j + 1;
? }
        ...
```

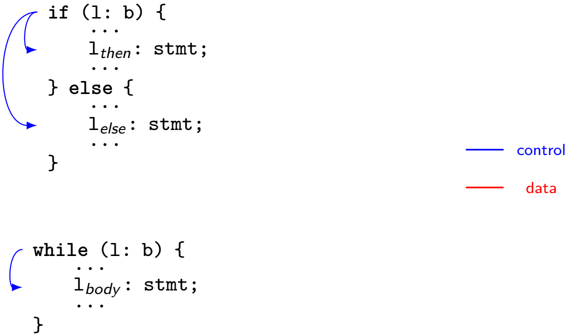


A few words about the formalization in Coq

- Results proved in Coq
- Size: 3,200 loc of spec, 6,500 loc of proof
- Certified slicer in OCaml extracted from Coq

Outline

Control dependence on a structured language

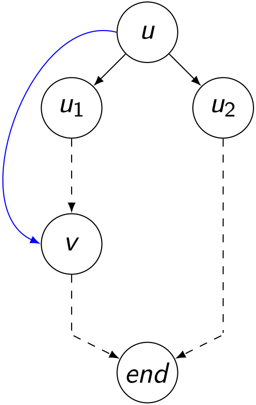


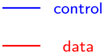
```
if (l: b) {
    ...
    l_then: stmt;
    ...
} else {
    ...
    l_else: stmt;
    ...
}
```

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

Control dependence on a control flow graph [Ferrante et al., 1987]

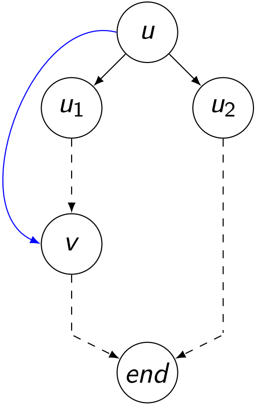
v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is always followed (*post-dominated*) by v, but not u_2





Control dependence on a control flow graph [Ferrante et al., 1987]

v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is always followed (*post-dominated*) by v, but not u_2



v is present
on all paths $u_1 \rightsquigarrow$ *end*,
but not on $u_2 \rightsquigarrow$ end

Control dependence on a finite directed graph

- Remove the unique end node requirement [Amtoft, 2008]

- Unifying theory [Danicic et al., 2011]
  - Generalizes previous formalizations [Ferrante et al., 1987] [Amtoft, 2008]

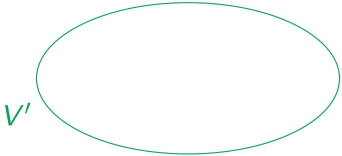
Control dependence on a finite directed graph

- Remove the unique end node requirement [Amtoft, 2008]

- **Unifying theory** [Danicic et al., 2011]
  - Generalizes previous formalizations [Ferrante et al., 1987] [Amtoft, 2008]

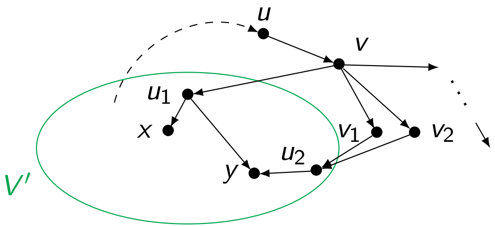
Definitions

- Defined for a subset of vertices V'
- Non-restrictive assumption for the talk:
  - all nodes are reachable from V'



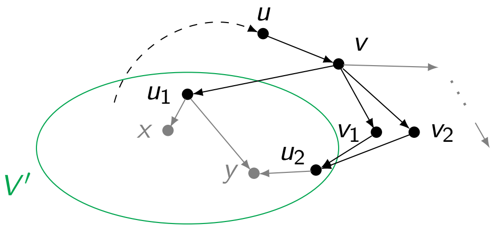
Definitions

- Defined for a subset of vertices V'
- Non-restrictive assumption for the talk:
  - all nodes are reachable from V'



Definitions

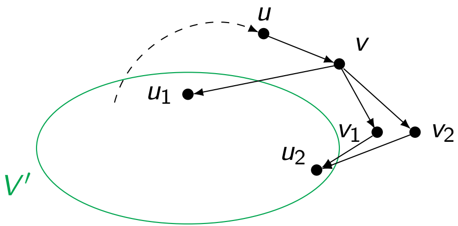
- *obs*(u): set of first-reachable nodes (observables) from u in V'



$\text{obs}(u) = \{u_1, u_2\}$
$\text{obs}(v) = \{u_1, u_2\}$
$\text{obs}(v_1) = \{u_2\}$
$\text{obs}(v_2) = \{u_2\}$
$\text{obs}(u_1) = \{u_1\}$
$\text{obs}(u_2) = \{u_2\}$

Definitions

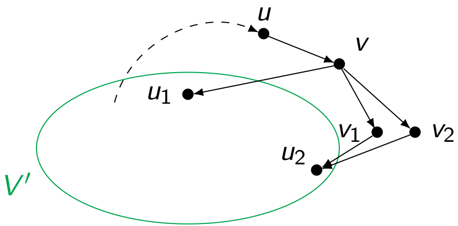
- *obs*(u): set of first-reachable nodes (observables) from u in V'
- V' is closed under control dependence (or control-closed) iff every node has at most one observable in V'



obs(u) $= \{u_1, u_2\}$ X
obs(v) $= \{u_1, u_2\}$ X
obs(v_1) $= \{u_2\}$ ✓
obs(v_2) $= \{u_2\}$ ✓
obs(u_1) $= \{u_1\}$ ✓
obs(u_2) $= \{u_2\}$ ✓

Definitions

- *obs*(u): set of first-reachable nodes (observables) from u in V'
- V' is closed under control dependence (or control-closed) iff every node has at most one observable in V'
- Control-closure of V': smallest control-closed superset

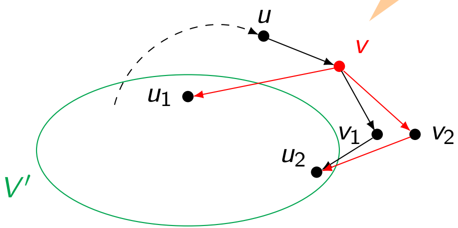


obs(u) = $\{u_1, u_2\}$X
obs(v) = $\{u_1, u_2\}$X
obs(v_1) = $\{u_2\}$✓
obs(v_2) = $\{u_2\}$✓
obs(u_1) = $\{u_1\}$✓
obs(u_2) = $\{u_2\}$✓

Definitions

- $obs(u)$: set of first-reachable nodes (observables) from u in V'
- V' is closed under control dependence (or control-closed) iff every node has at most one observable in V'
- Control-closure of V': smallest control-closed superset
- A node is V'-deciding if it gives rise to two non-trivial paths reaching V' that share no vertex except their origin

point of divergence closest to V'



$\text{obs}(u) = \{u_1, u_2\}$ X

$\text{obs}(v) = \{u_1, u_2\}$ X

$\text{obs}(v_1) = \{u_2\}$ ✓

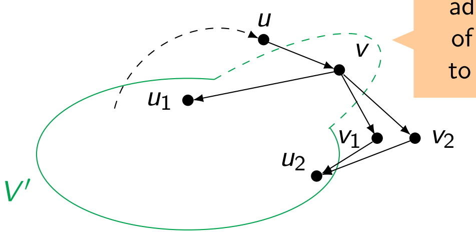
$\text{obs}(v_2) = \{u_2\}$ ✓

$\text{obs}(u_1) = \{u_1\}$ ✓

$\text{obs}(u_2) = \{u_2\}$ ✓

Definitions

- *obs*(u): set of first-reachable nodes (observables) from u in V'
- V' is closed under control dependence (or control-closed) iff every node has at most one observable in V'
- Control-closure of V': smallest control-closed superset
- A node is V'-deciding if it gives rise to two non-trivial paths reaching V' that share no vertex except their origin
- Theorem. control-closure(V') = V' ∪ { V'-deciding vertices }

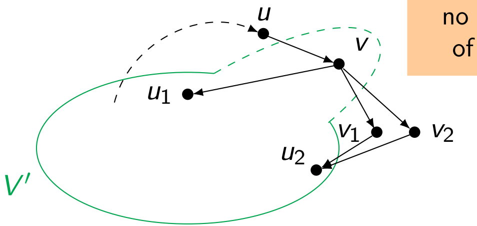


add all points of divergence to the closure

$\text{obs}(u) = \{u_1, u_2\}$
$\text{obs}(v) = \{u_1, u_2\}$
$\text{obs}(v_1) = \{u_2\}$
$\text{obs}(v_2) = \{u_2\}$
$\text{obs}(u_1) = \{u_1\}$
$\text{obs}(u_2) = \{u_2\}$

Definitions

- *obs*(u): set of first-reachable nodes (observables) from u in V'
- V' is closed under control dependence (or control-closed) iff every node has at most one observable in V'
- Control-closure of V': smallest control-closed superset
- A node is V'-deciding if it gives rise to two non-trivial paths reaching V' that share no vertex except their origin
- Theorem. control-closure(V') = V' ∪ { V'-deciding vertices }



no more points of divergence

$\text{obs}(u) = \{v\}$ ✓
$\text{obs}(v) = \{v\}$ ✓
$\text{obs}(v_1) = \{u_2\}$ ✓
$\text{obs}(v_2) = \{u_2\}$ ✓
$\text{obs}(u_1) = \{u_1\}$ ✓
$\text{obs}(u_2) = \{u_2\}$ ✓

Danicic's method to compute control-closure

```
begin
    $W \leftarrow V'$;
    while there exists a node $u$ that is $V'$-deciding do
        add that node to $W$
    end
    return $W$;                // the control-closure of $V'$
end
```

Danicic's method to compute control-closure

```
begin
    $W \leftarrow V'$;
    while there exists a node $u$ that is $V'$-deciding do
        add that node to $W$
    end
    return $W$
end
```

u is the last point of divergence before V'

Danicic's method to compute control-closure

u is a rich parent
v is a poor child

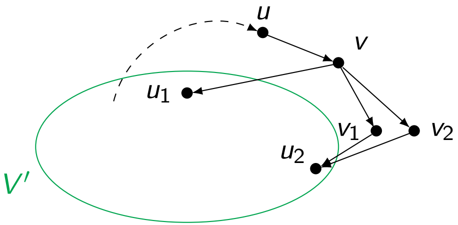
with strictly more observables in **W**
than one of its children *v*

```
begin
    $W \leftarrow V'$;
    while there exists a node $u$ ~~that is $V'$-deciding~~ do
        add that node to $W$
    end
    return $W$
end
```

Formally:

$1 \leq |obs(v)| < |obs(u)|$

Rich parent/poor child illustrated



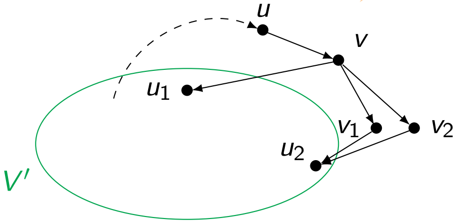
$\mathrm{obs}(u) = \{u_1, u_2\}$ ✗

$\mathrm{obs}(v) = \{u_1, u_2\}$ ✗

$\mathrm{obs}(v_1) = \{u_2\}$ ✓

$\mathrm{obs}(v_2) = \{u_2\}$ ✓

Rich parent/poor child illustrated



rich parent
(observes u_1 and u_2)

rich child
(observes u_1 and u_2)

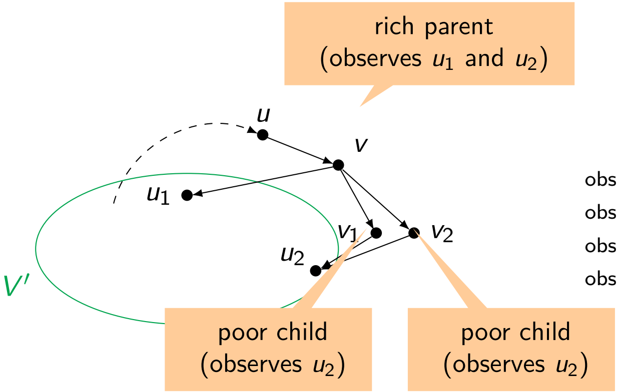
$\text{obs}(u) = \{u_1, u_2\}$ ✗

$\text{obs}(v) = \{u_1, u_2\}$ ✗

$\text{obs}(v_1) = \{u_2\}$ ✓

$\text{obs}(v_2) = \{u_2\}$ ✓

Rich parent/poor child illustrated



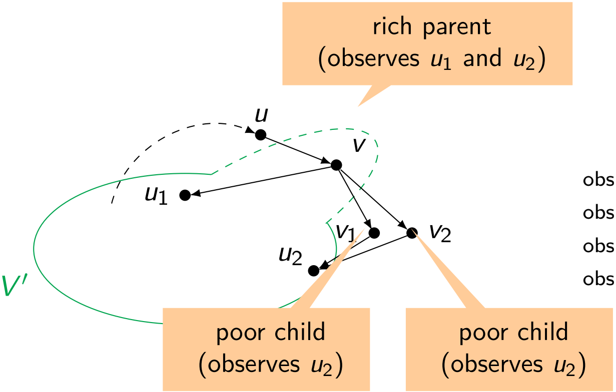
$\text{obs}(u) = \{u_1, u_2\}$ ✗

$\text{obs}(v) = \{u_1, u_2\}$ ✗

$\text{obs}(v_1) = \{u_2\}$ ✓

$\text{obs}(v_2) = \{u_2\}$ ✓

Rich parent/poor child illustrated



rich parent
(observes u_1 and u_2)

poor child
(observes u_2)

poor child
(observes u_2)

$\mathrm{obs}(u) = \{u_1, u_2\}$✗

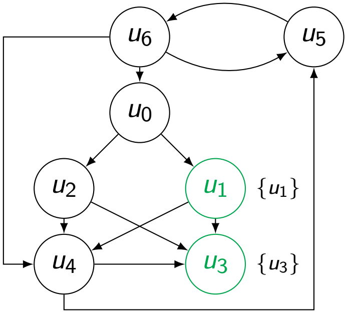
$\mathrm{obs}(v) = \{u_1, u_2\}$✗

$\mathrm{obs}(v_1) = \{u_2\}$✓

$\mathrm{obs}(v_2) = \{u_2\}$✓

Danicic's algorithm on an example

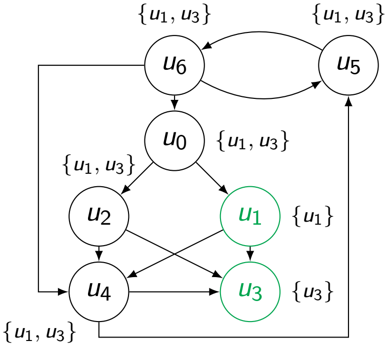
$V' = \{u_1, u_3\}$



$W = V' = \{u_1, u_3\}$

Danicic's algorithm on an example

Iteration 1a: compute the set of observables of every node

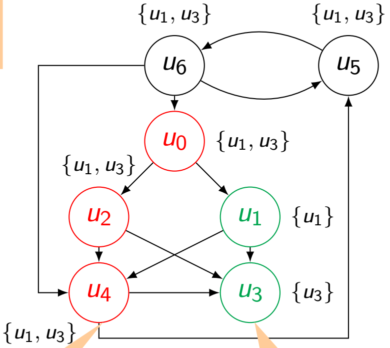


$W = \{u_1, u_3\}$

Danicic's algorithm on an example

Iteration 1b: identify edges (u, v) such that $1 \leq |obs(v)| < |obs(u)|$

identify rich parents with a poor child



rich parent (observes u_1 and u_3)

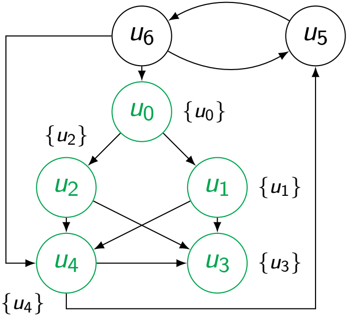
poor child (observes u_3)

$W = \{u_1, u_3\}$

Danicic's algorithm on an example

Iteration 1c: update W and throw away annotations

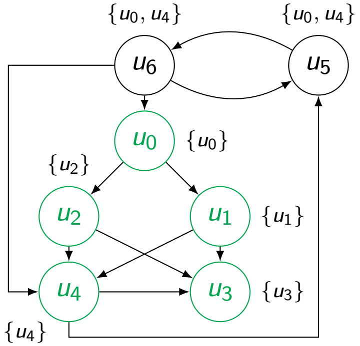
add rich parents
having a poor child



$$W = \{u_0, u_1, u_2, u_3, u_4\}$$

Danicic's algorithm on an example

Iteration 2a: compute the observables of every node

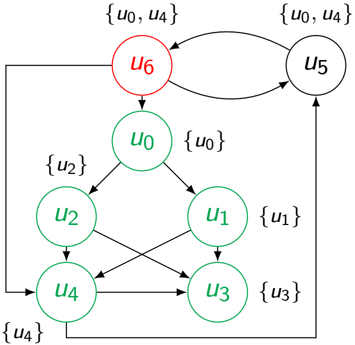


$W = \{u_0, u_1, u_2, u_3, u_4\}$

Danicic's algorithm on an example

Iteration 2b: identify edges (u, v) such that $1 \leq |obs(v)| < |obs(u)|$

identify rich parents with a poor child

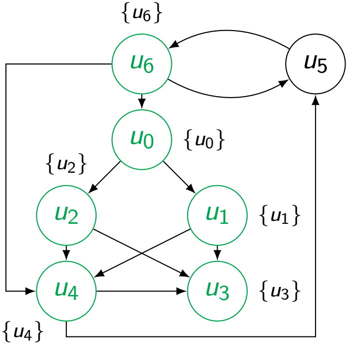


$W = \{u_0, u_1, u_2, u_3, u_4\}$

Danicic's algorithm on an example

Iteration 2c: update W and throw away annotations

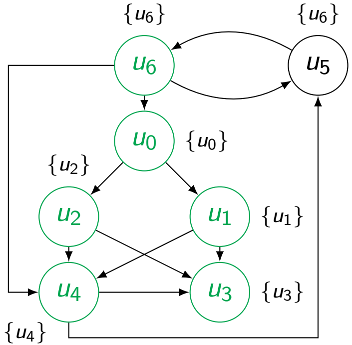
add rich parents
having a poor child



$W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$

Danicic's algorithm on an example

Iteration 3a: compute the observables of every node

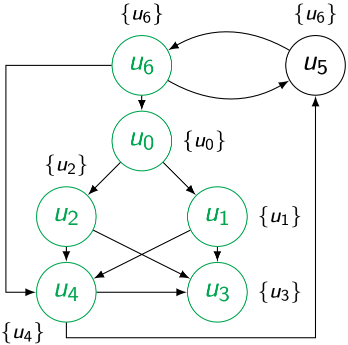


$W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$

Danicic's algorithm on an example

Iteration 3b: identify edges (u, v) such that $1 \leq |obs(v)| < |obs(u)|$

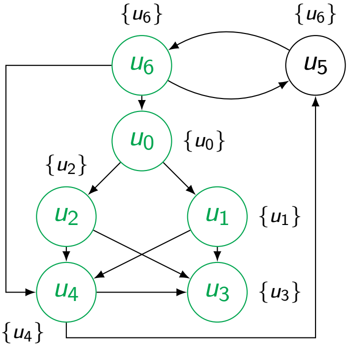
identify rich parents with a poor child



$W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$

Danicic's algorithm on an example

Iteration 3c: no new node, return W



Closure: $\{u_0, u_1, u_2, u_3, u_4, u_6\}$

A few words about the formalization in Coq

- We formalized control-closure in Coq
  - We found and fixed a minor inconsistency in Danicic's paper proof
- We implemented (a slightly optimized version of) Danicic's algorithm and proved it correct
- Size: 2,000 loc of spec, 4,600 loc of proof

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

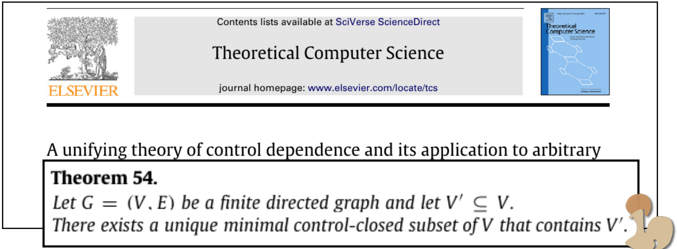
ELSEVIER

A unifying theory of control dependence and its application to arbitrary program structures

Sebastian Danicic[a,*], Richard W. Barraclough[b], Mark Harman[c], John D. Howroyd[a], Ákos Kiss[d], Michael R. Laurence[e]

A few words about the formalization in Coq

- We formalized control-closure in Coq
  - We found and fixed a minor inconsistency in Danicic's paper proof
- We implemented (a slightly optimized version of) Danicic's algorithm and proved it correct
- Size: 2,000 loc of spec, 4,600 loc of proof



ELSEVIER

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Theoretical Computer Science

A unifying theory of control dependence and its application to arbitrary

Theorem 54.
Let $G = (V, E)$ be a finite directed graph and let $V' \subseteq V$.
There exists a unique minimal control-closed subset of V that contains V'.

Outline

Context: Static Backward Slicing

Slicing in the Presence of Errors

An Algorithm for Arbitrary Control Dependence

An Optimized Algorithm

Conclusion

Motivation for a more efficient algorithm

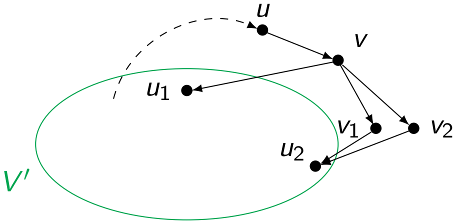
- Danicic et al. believe that “better than $O(|V|^3)$ worst-case time complexity algorithms may exist”

- Fundamental limitation of Danicic’s algorithm:
    - It does not take advantage of previous iterations to speed up the following ones

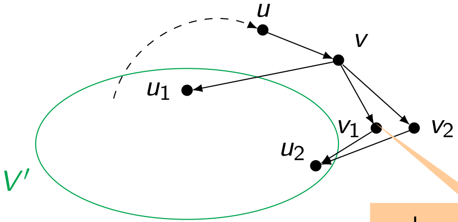
New iterative algorithm: key ideas

- Rich parent/poor child detection
    - no need to compute the set of observables exactly
    - just exhibit a witness: a node observable from the parent, but not from the child
- Label each vertex with a candidate observable (if any)
    - each vertex is labeled with at most one vertex
    - can be temporarily outdated
    - the labeling survives the iterations and can be reused
- One additional output
    - at the end, labels are the true observables

Rich parent/poor child illustrated again



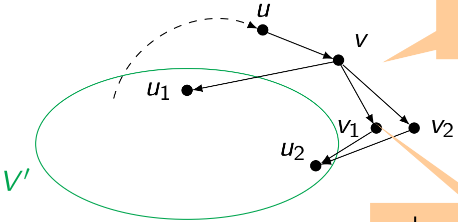
Rich parent/poor child illustrated again



observes u_1

does not observe u_1

Rich parent/poor child illustrated again

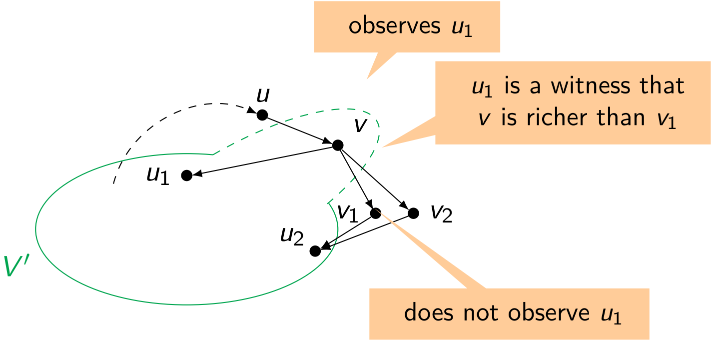


observes u_1

u_1 is a witness that v is richer than v_1

does not observe u_1

Rich parent/poor child illustrated again



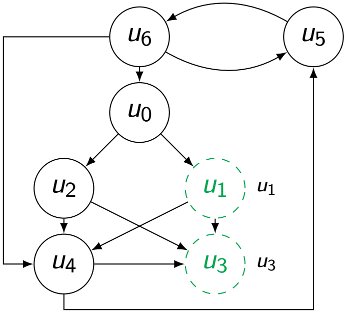
observes u_1

u_1 is a witness that v is richer than v_1

does not observe u_1

The optimized algorithm on an example

$V' = \{u_1, u_3\}$

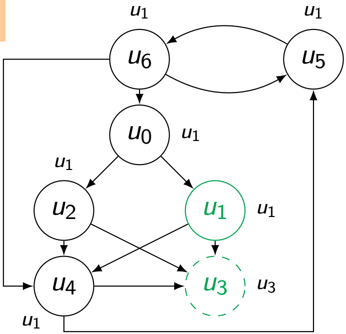


$W = V' = \{u_1, u_3\}$

The optimized algorithm on an example

Iteration 1a: propagate u_1 backwards

which vertex has u_1 as observable?



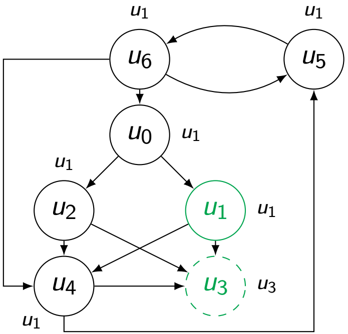
$W = \{u_1, u_3\}$

The optimized algorithm on an example

Iteration 1b: identify edges (u, v) such that $u_1 \in obs(u)$, $u_1 \notin obs(v)$

identify
rich parent/poor child
with witness u_1

none found

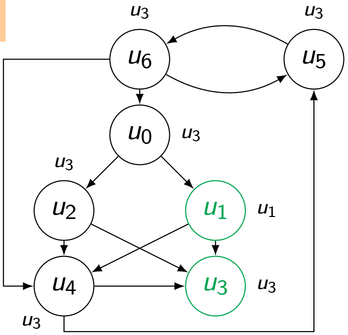


$W = \{u_1, u_3\}$

The optimized algorithm on an example

Iteration 2a: propagate u_3 backwards

which vertex has u_3 as observable?

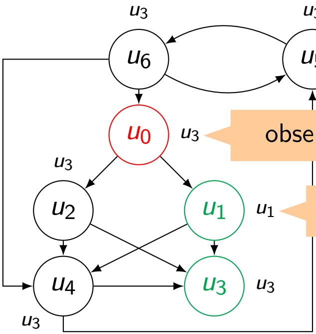


$W = \{u_1, u_3\}$

The optimized algorithm on an example

Iteration 2b: identify edges (u, v) such that $u_3 \in obs(u)$, $u_3 \notin obs(v)$

identify
rich parent/poor child
with witness u_3



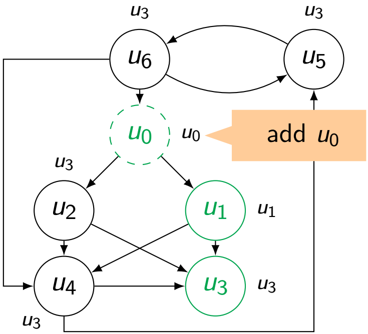
observes u_3

does not observe u_3

$$W = \{u_1, u_3\}$$

The optimized algorithm on an example

Iteration 2c: update W

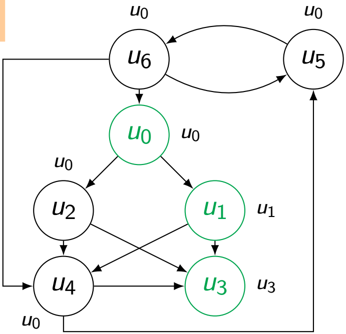


$W = \{u_0, u_1, u_3\}$

The optimized algorithm on an example

Iteration 3a: propagate u_0 backwards

which vertex has u_0 as observable?

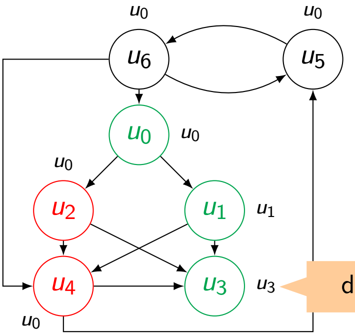


$$W = \{u_0, u_1, u_3\}$$

The optimized algorithm on an example

Iteration 3b: identify edges (u, v) such that $u_0 \in obs(u)$, $u_0 \notin obs(v)$

identify
rich parent/poor child
with witness u_0



observes u_0

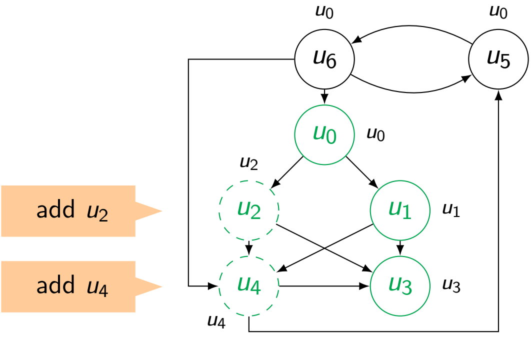
observes u_0

does not observe u_0

$$W = \{u_0, u_1, u_3\}$$

The optimized algorithm on an example

Iteration 3c: update W

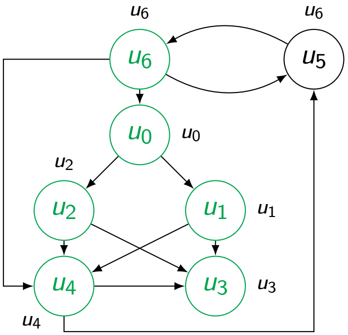

add u_2

add u_4

$$W = \{u_0, u_1, u_2, u_3, u_4\}$$

The optimized algorithm on an example

Iteration 7: no more unprocessed vertex, return W



Closure: $\{u_0, u_1, u_2, u_3, u_4, u_6\}$

A few words about the formalization in Why3

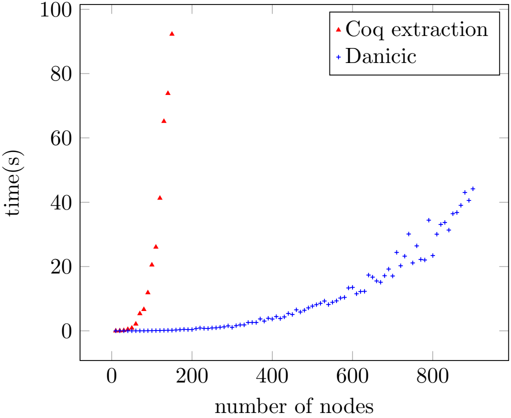
The Why3 development has two parts:

- the new algorithm (250 loc)
  - split into 3 functions
  - most proofs are discharged automatically
  - preservation of the main invariants proved manually in Coq (100 lines of Coq proof)
- a small fragment of control dependence theory (80 loc)
  - everything proved
  - one lemma admitted (but proved in the Coq formalization)

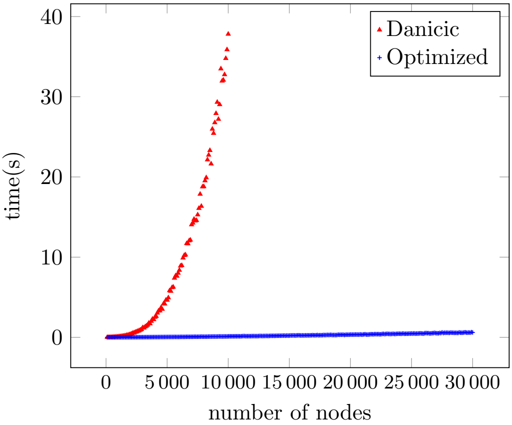
Experiments

- Both algorithms were implemented in OCaml using OCamlgraph
- They were run on randomly generated graphs
- Checked on small graphs against a certified version extracted from Coq

Coq extraction vs. Danicic



Danicic vs. the new optimized algorithm



Contributions

- A theoretical justification of slicing in the presence of errors and non-termination
- An algorithm computing efficiently control dependence on finite graphs

- All results are certified (in Coq or Why3)
  - including Danicic's algorithm

Perspectives

- Formalize strong control dependence from [Danicic et al., 2011]
- Perform experiments on realistic CFGs
- Investigate more optimizations

Graph theory

	Alt-Ergo (1.30)	CVC4 (1.5)	Coq (8.6.1)	Eprover (2.0)	Z3 (4.5.0)
Number	10	14	4	6	0
Min time (s)	0	0,02	0,27	0,01	0
Max time (s)	0,01	0,67	0,37	0,44	0
Avg time (s)	0,01	0,083	0,3	0,093	N/A

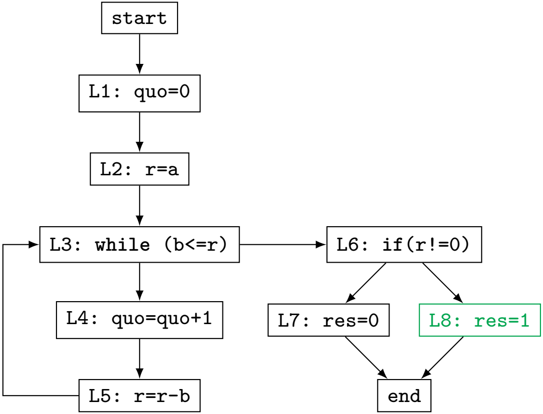
+ 1 axiom (but proved in the Coq formalization)

Algorithm

	Alt-Ergo (1.30)	CVC4 (1.5)	Coq (8.6.1)	Eprover (2.0)	Z3 (4.5.0)
Number	233	12	4	4	2
Min time (s)	0,01	0,08	0,32	0,08	0,34
Max time (s)	3,96	0,83	0,76	2,35	3,18
Avg time (s)	0,18	0,46	0,48	0,72	1,76

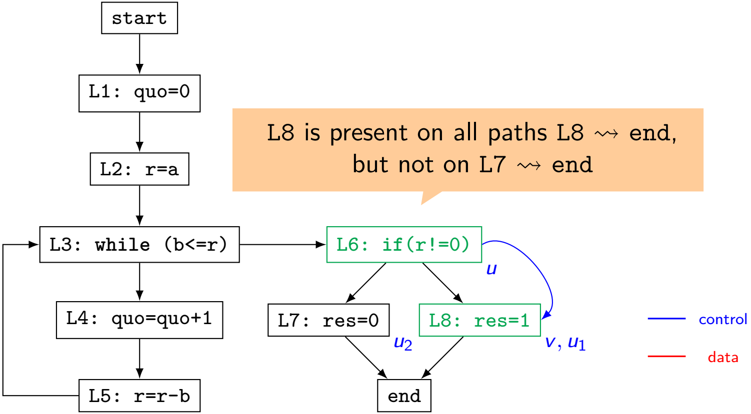
Control dependence on a control flow graph [Ferrante et al., 1987]

v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is always followed (*post-dominated*) by v, but not u_2



Control dependence on a control flow graph [Ferrante et al., 1987]

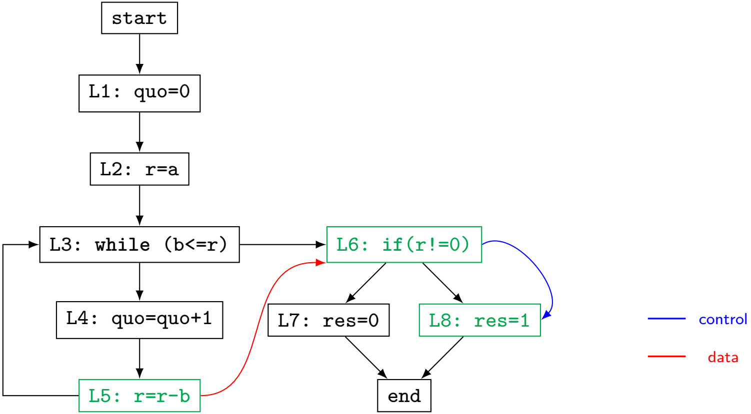
v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is always followed (*post-dominated*) by v, but not u_2



L8 is present on all paths L8 ⇝ end, but not on L7 ⇝ end

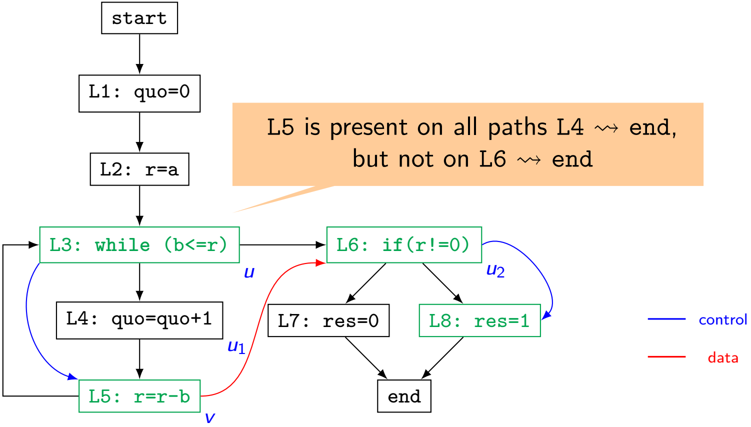
Control dependence on a control flow graph [Ferrante et al., 1987]

v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is always followed (*post-dominated*) by v, but not u_2



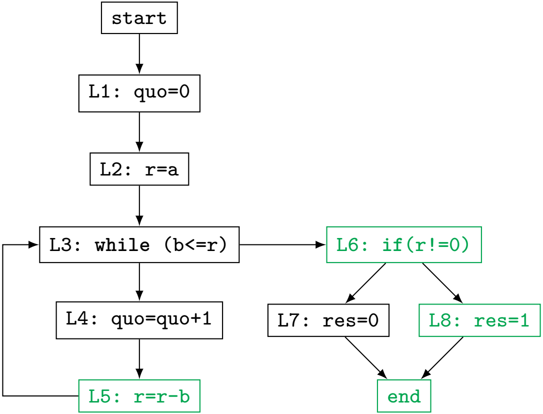
Control dependence on a control flow graph [Ferrante et al., 1987]

v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is always followed (*post-dominated*) by v, but not u_2



L5 is present on all paths L4 ⇝ end, but not on L6 ⇝ end

Control-closure for our running example



Control-closure for our running example