Fast Computation of Arbitrary Control Dependencies

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall

list ceatech

CentraleSupélec

April 19th, 2018
ETAPS/FASE 2018 - Thessaloniki

Motivation

Create a certified and efficient generic slicer

Definition of static backward slicing

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;
    }
```

?

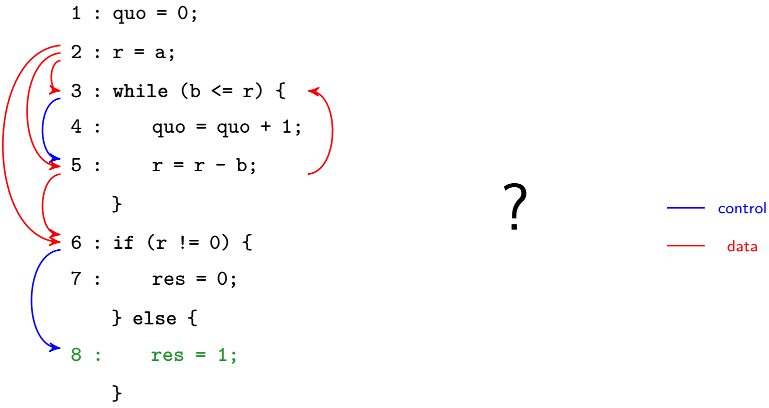
— control

— data

Original program p

Slice q w.r.t. line 8

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



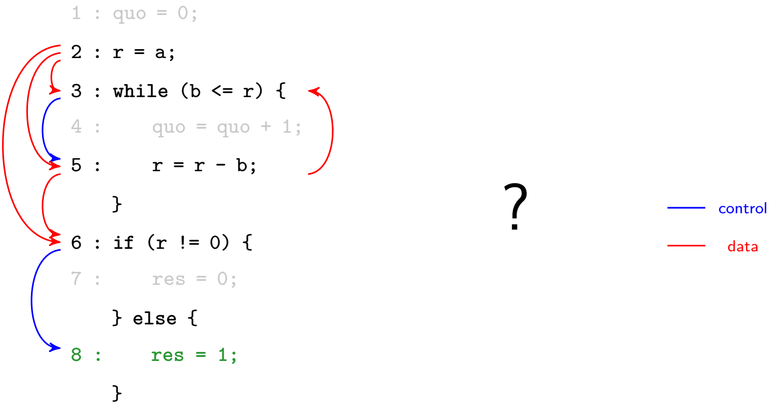
```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

?

Original program p

Slice q w.r.t. line 8

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

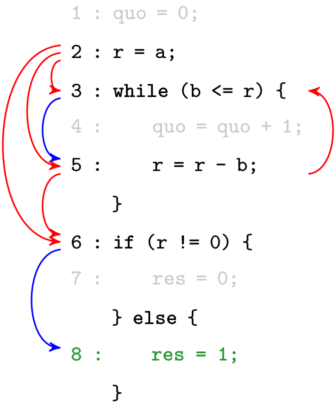


```
1 : quo = 0;
2 : r = a;
3 : while (b <= r) {
4 :     quo = quo + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

Slice q w.r.t. line 8

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

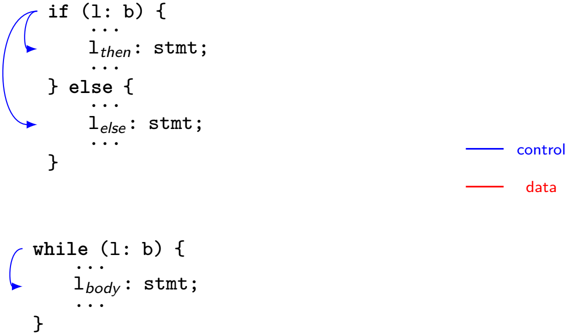
```
2 : r = a;
3 : while (b <= r) {

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

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

Slice q w.r.t. line 8

On a structured language

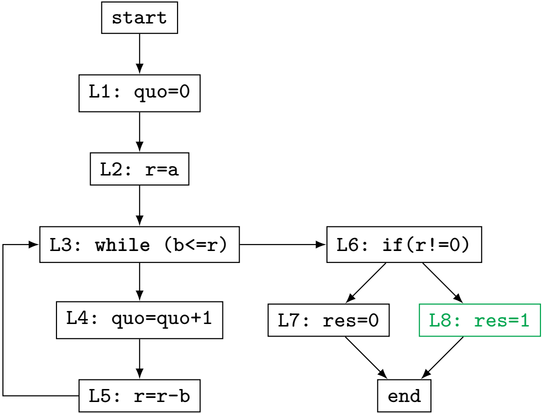


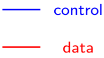
```
if (l: b) {
    ...
    l_then: stmt;
    ...
} else {
    ...
    l_else: stmt;
    ...
}
```

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

On a control flow graph [Ferrante et al., 1987]

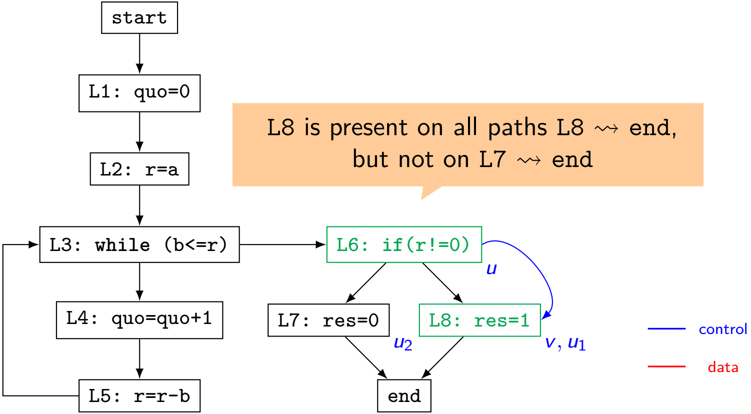
v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is post-dominated by v, but not u_2





On a control flow graph [Ferrante et al., 1987]

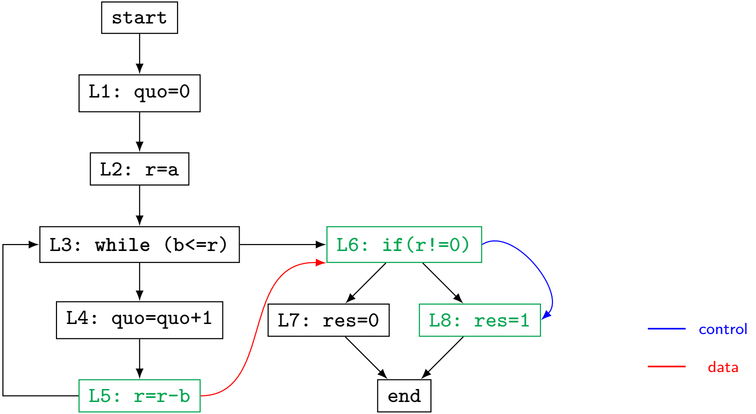
v is control-dependent on u iff u has two children u_1 and u_2 such that u_1 is post-dominated by v, but not u_2



L8 is present on all paths L8 $\rightsquigarrow$ end, but not on L7 $\rightsquigarrow$ end

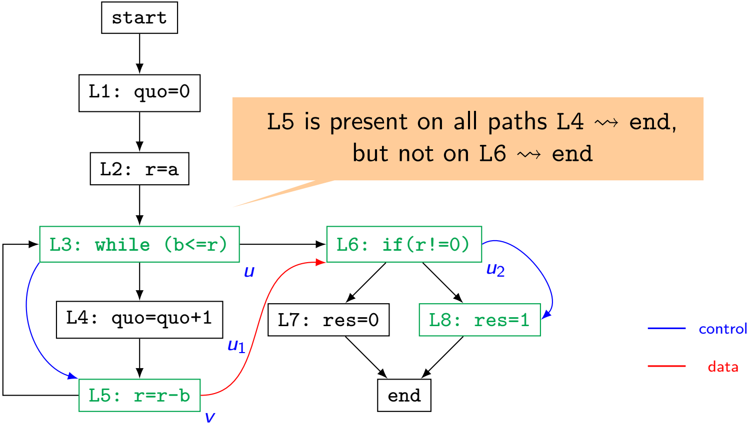
On a control flow graph [Ferrante et al., 1987]

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



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 post-dominated by v, but not u_2



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

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]

Outline

A brief history of control dependence
- Context: static backward slicing
- Definitions of control dependence

Danicic's theory of control dependence
- Concepts
- Algorithm
- Contribution: formalization in Coq

A new optimized algorithm
- Presentation
- Formalization in Why3
- Experiments

Conclusion

Outline

A brief history of control dependence

- Context: static backward slicing
- Definitions of control dependence

Danicic's theory of control dependence

- Concepts
- Algorithm
- Contribution: formalization in Coq

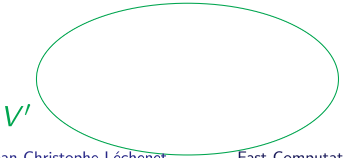
A new optimized algorithm

- Presentation
- Formalization in Why3
- Experiments

Conclusion

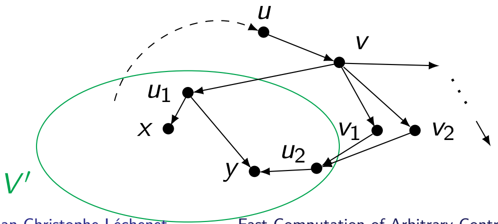
Definitions

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



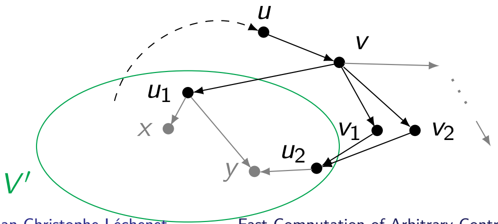
Definitions

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



Definitions

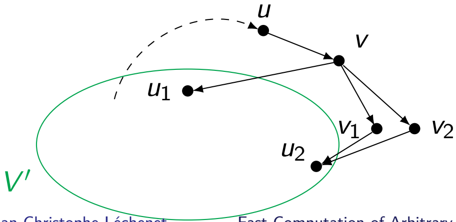
- *obs*(u): set of first-reachable nodes (observables) from u in V'



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

Definitions

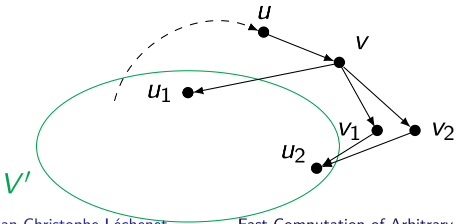
- *obs*(u): set of first-reachable nodes (observables) from u in V'
- V' is closed under control dependence (or control-closed) iff every node has at most one observable in V'



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

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

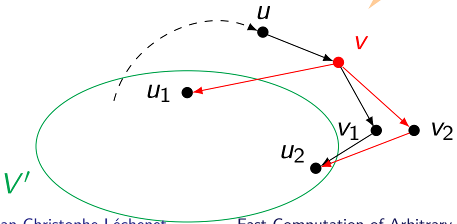


obs(u) = $\{u_1, u_2\}$✗
obs(v) = $\{u_1, u_2\}$✗
obs(v_1) = $\{u_2\}$✓
obs(v_2) = $\{u_2\}$✓
obs(u_1) = $\{u_1\}$✓
obs(u_2) = $\{u_2\}$✓

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 sha eir origin

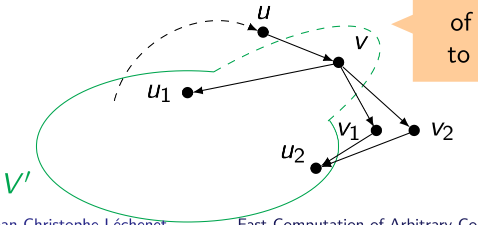
point of divergence closest to V'



obs(u) = $\{u_1, u_2\}$ ✗
obs(v) = $\{u_1, u_2\}$ ✗
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
- Theorem. control-closure(V') = V' ∪ { V'-deciding vertices }

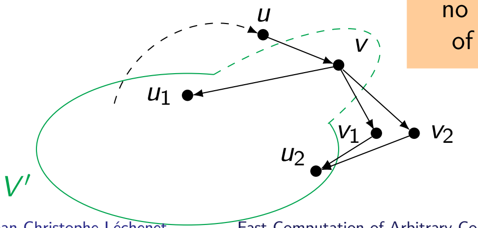


add all points of divergence to the closure

obs(u) = $\{u_1, u_2\}$
obs(v) = $\{u_1, u_2\}$
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
- Theorem. control-closure(V') = V' ∪ { V'-deciding vertices }

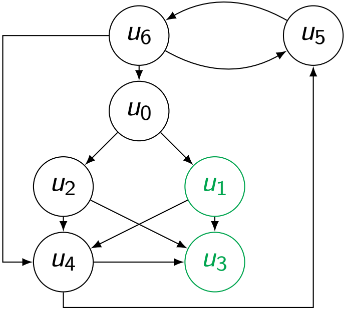


no more points of divergence

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

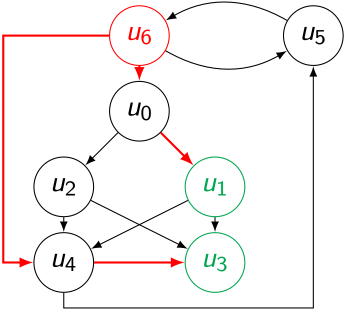
Running example

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



Running example

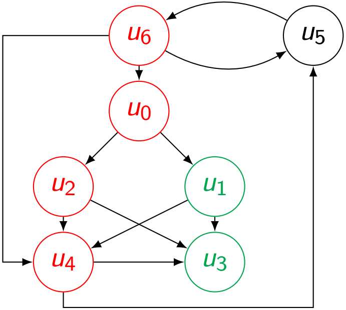
$V' = \{u_1, u_3\}$



u_6 is V'-deciding (point of divergence closest to V')

Running example

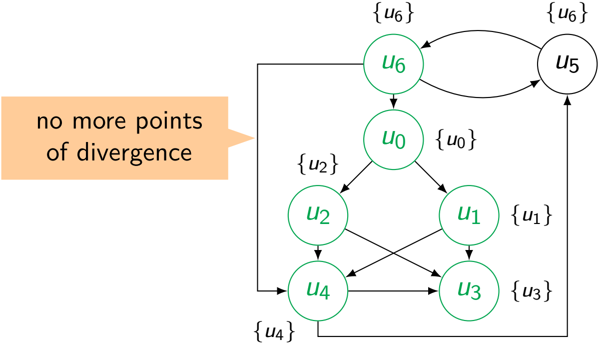
$V' = \{u_1, u_3\}$



u_0, u_2 and u_4 are V'-deciding too

Running example

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



no more points of divergence

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

Danicic's method to compute control closure

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

Danicic's method to compute control closure

```
begin
    $W \leftarrow V'$;
    while there exists a node $u$ $\overbrace{\text{~~that is } V'\text{-deciding~~}}^{\text{with strictly more observables in W than one of its children } v}$ do
        add all such nodes to $W$
    end
    return $W$
end
```

u is a rich parent
v is a poor child

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

Danicic's method to compute control closure

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

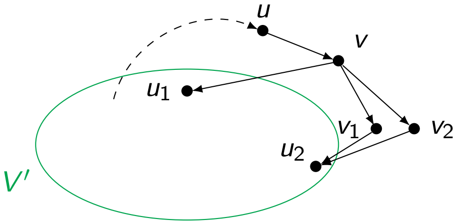
```
begin
    $W \leftarrow V'$;
    while there exists a node $u$ ~~that is $V'$-deciding~~ do
        add all such nodes to $W$
    end
    return $W$
end
```

u is a rich parent
v is a poor child

Formally:

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

Rich parent illustrated



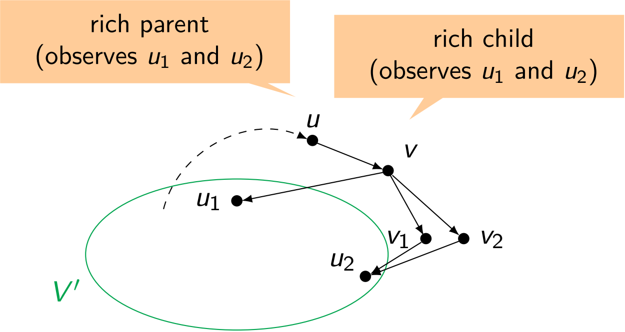
obs(u) = $\{u_1, u_2\}$✗

obs(v) = $\{u_1, u_2\}$✗

obs(v_1) = $\{u_2\}$✓

obs(v_2) = $\{u_2\}$✓

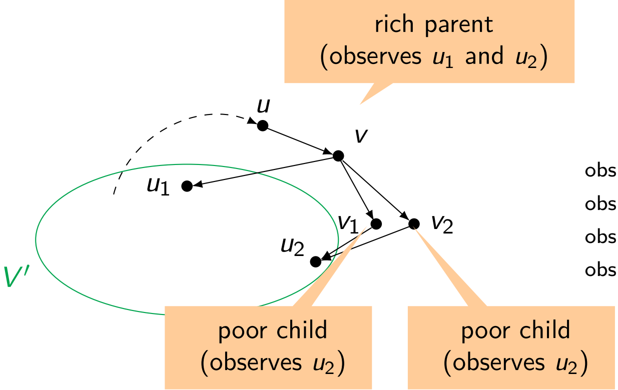
Rich parent illustrated


rich parent
(observes u_1 and u_2)

rich child
(observes u_1 and u_2)

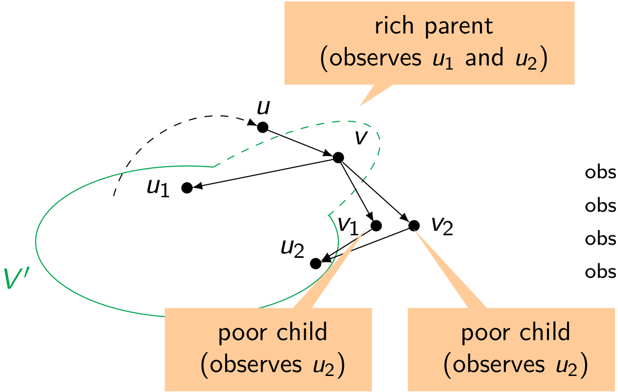
$\mathrm{obs}(u) = \{u_1, u_2\}$ X
$\mathrm{obs}(v) = \{u_1, u_2\}$ X
$\mathrm{obs}(v_1) = \{u_2\}$ ✓
$\mathrm{obs}(v_2) = \{u_2\}$ ✓

Rich parent 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 illustrated



$\mathrm{obs}(u) = \{u_1, u_2\}$ ✗

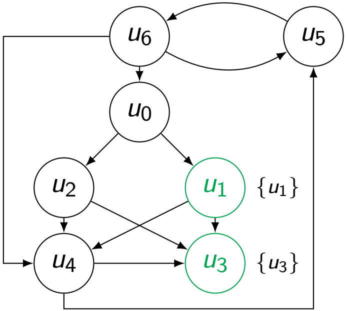
$\mathrm{obs}(v) = \{u_1, u_2\}$ ✗

$\mathrm{obs}(v_1) = \{u_2\}$ ✓

$\mathrm{obs}(v_2) = \{u_2\}$ ✓

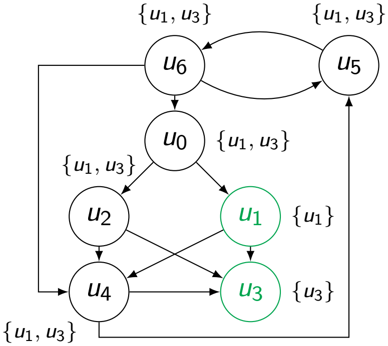
Danicic's algorithm on an example

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



Danicic's algorithm on an example

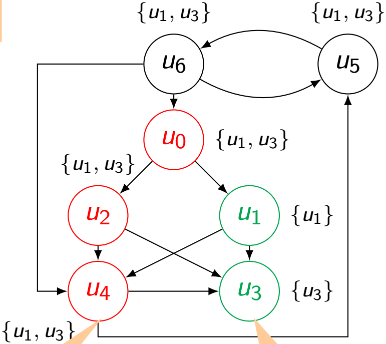
Iteration 1a: compute the set of observables of every node



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



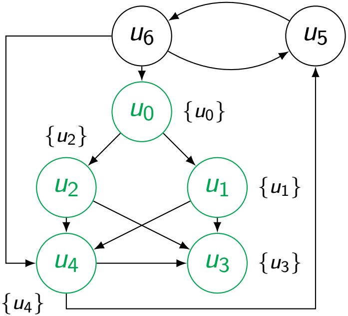
rich parent
(observes u_1 and u_3)

poor child
(observes u_3)

Danicic's algorithm on an example

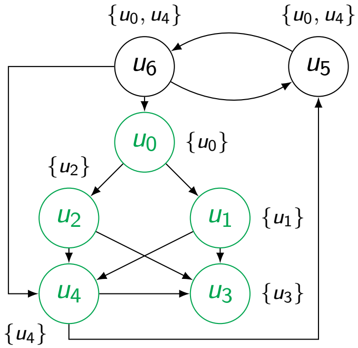
Iteration 1c: update W and throw away annotations

add rich parents
having a poor child



Danicic's algorithm on an example

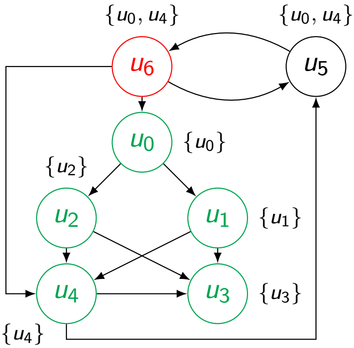
Iteration 2a: compute the observables of every node



Danicic's algorithm on an example

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

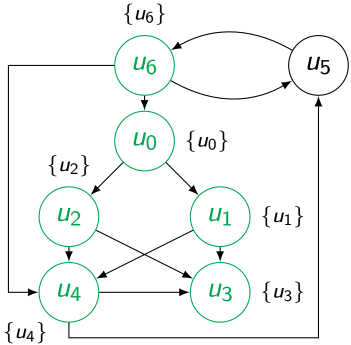
identify rich parents with a poor child



Danicic's algorithm on an example

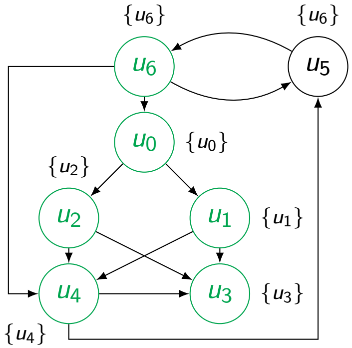
Iteration 2c: update W and throw away annotations

add rich parents having a poor child



Danicic's algorithm on an example

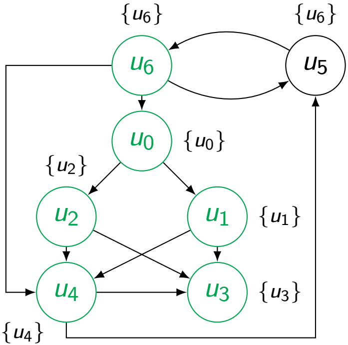
Iteration 3a: compute the observables of every node



Danicic's algorithm on an example

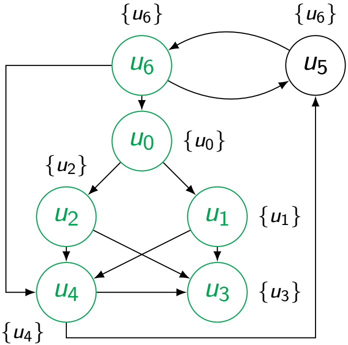
Iteration 3b: identify edges (u, v) such that $1 \leq |obs(v)| < |obs(u)|$

identify rich parents with a poor child



Danicic's algorithm on an example

Iteration 3c: no new node, return W



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

Danicic's initial algorithm

Less optimized:

- Restricted rich parent/poor child definition:
    - $|\,\text{obs}(v)| = 1$ instead of $1 \leq |\,\text{obs}(v)|$.
- At each iteration, adds only one rich parent having a poor child to W instead of all

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

Fundamental limitation of Danicic's algorithm

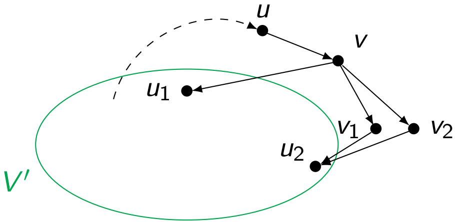
Danicic's algorithm does not take advantage of previous iterations to speed up the following ones.

Outline

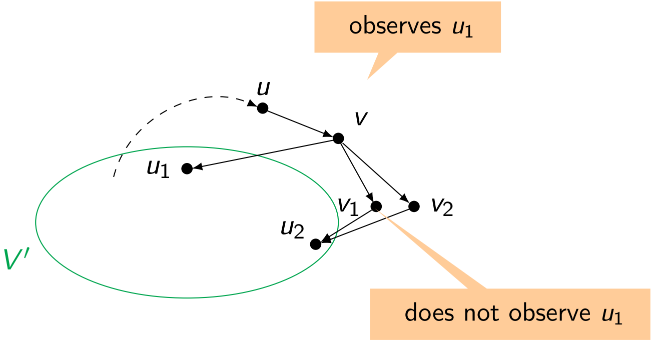
New iterative algorithm: key ideas

- Rich parent/poor child
  - 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)
  - the label is a single vertex
  - can be outdated
  - the labeling survives the iterations and can be reused
  - at the end, labels are the true observables

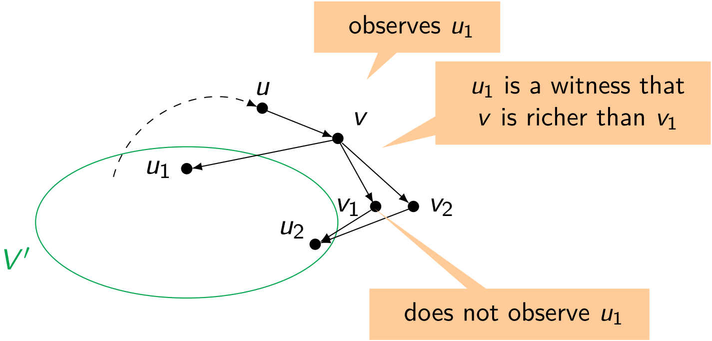
Rich parent illustrated again



Rich parent illustrated again



Rich parent illustrated again

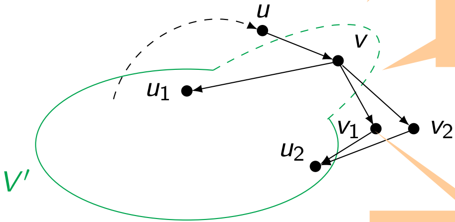


observes u_1

u_1 is a witness that v is richer than v_1

does not observe u_1

Rich parent illustrated again



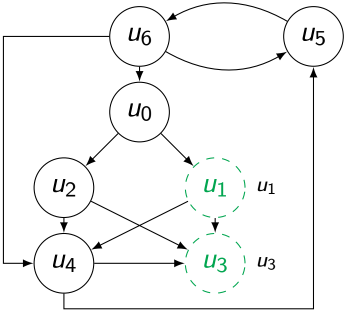
observes u_1

u_1 is a witness that v is richer than v_1

does not observe u_1

The optimized algorithm on an example

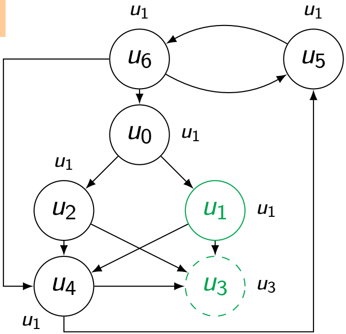
$V' = \{u_1, u_3\}$



The optimized algorithm on an example

Iteration 1a: propagate u_1 backwards

which vertex has u_1 as observable?

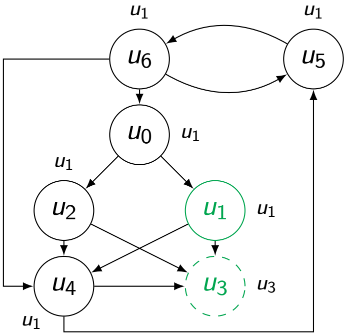


The optimized algorithm on an example

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

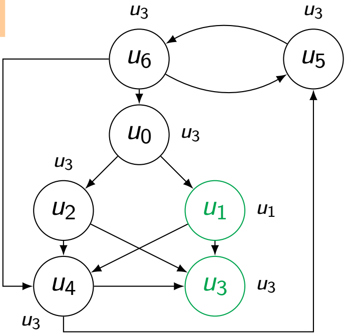
none found



The optimized algorithm on an example

Iteration 2a: propagate u_3 backwards

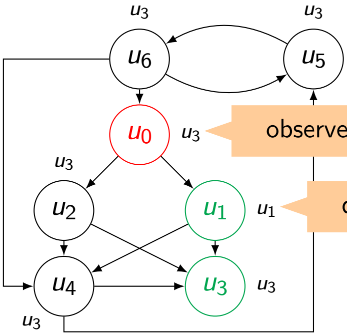
which vertex has u_3 as observable?



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

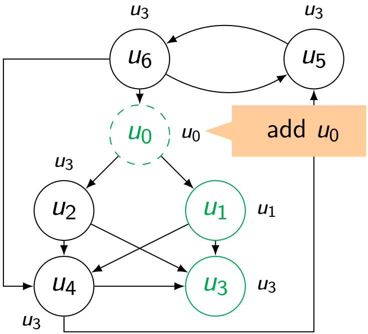


observes u_3

does not observe u_3

The optimized algorithm on an example

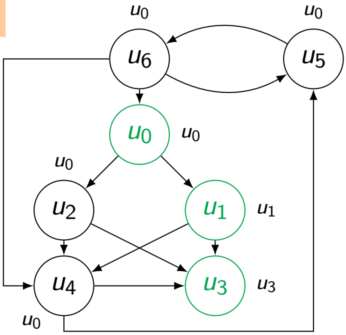
Iteration 2c: update W



The optimized algorithm on an example

Iteration 3a: propagate u_0 backwards

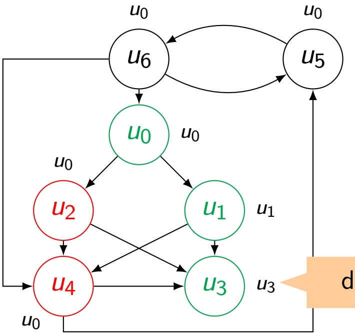
which vertex has u_0 as observable?



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



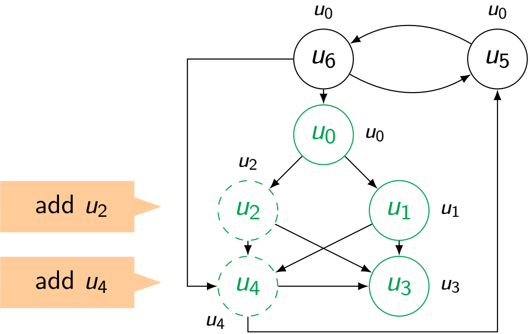
observes u_0

observes u_0

does not observe u_0

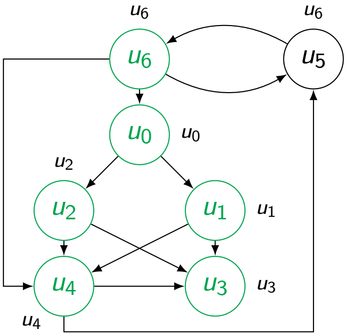
The optimized algorithm on an example

Iteration 3c: update W



The optimized algorithm on an example

Iteration 7: no more unprocessed vertex, return W



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

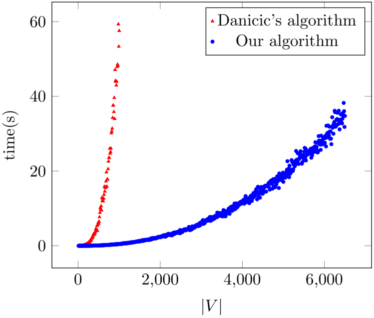
A few words about the formalization in Why3

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
- a small fragment of control dependence's 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



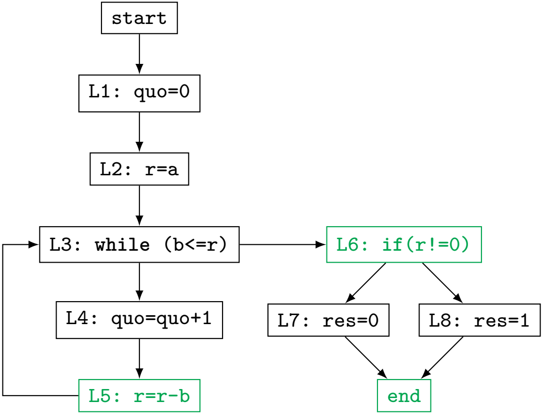
Contribution summary:

- Formalization in Coq of:
    - a theory of control dependence on finite directed graphs
    - an algorithm computing closure under control dependence
- Design of an optimization of this algorithm
- Proof in Why3 of this new algorithm
- Experiments confirm the new algorithm outperforms Danicic's method

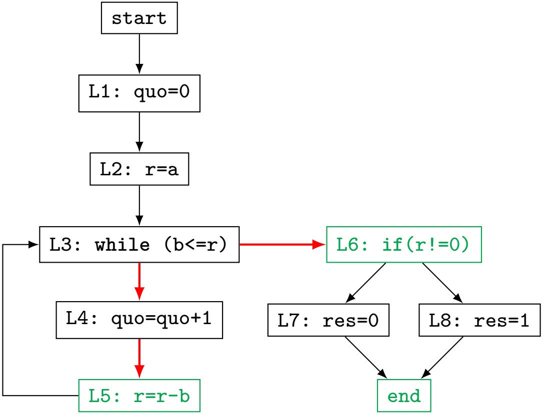
Future work:

- Integrate this work in a generic program slicer
- Formalize the other concepts in [Danicic et al., 2011]

Weak control-closure on euclidean division



Weak control-closure on euclidean division



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