Fast Computation of Arbitrary Control Dependencies

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall

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 $a$ ($a, b > 0$)

\[
\begin{align*}
1 : & \quad \text{quo} = 0; \\
2 : & \quad \text{r} = \text{a}; \\
3 : & \quad \text{while} (\text{b} \leq \text{r}) \{ \\
4 : & \quad \text{quo} = \text{quo} + 1; \\
5 : & \quad \text{r} = \text{r} - \text{b}; \\
6 : & \quad \text{if} (\text{r} \neq 0) \{ \\
7 : & \quad \text{res} = 0; \\
8 : & \quad \text{else} \{ \\
9 : & \quad \text{res} = 1; \\
\}
\end{align*}
\]

Euclidean division of $a$ by $b$ is the remainder equal to 0?

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 $a$ ($a, b > 0$)

Original program $p$

```
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;
}
```

Slice $q$ w.r.t. line 8

```
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;
}
```
On a structured language

if (l: b) {
  ...
  l\textsubscript{then}: stmt;
  ...
} else {
  ...
  l\textsubscript{else}: stmt;
  ...
}

while (l: b) {
  ...
  l\textsubscript{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 \( u_2 \)

```plaintext
start

L1: quo=0

L2: r=a

L3: while (b<=r)

L4: quo=quo+1

L5: r=r-b

L6: if(r!=0)

L7: res=0

L8: res=1

end
```

Jean-Christophe Léchenet Fast Computation of Arbitrary Control Dependencies April 19th, 2018 6 / 24
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 $u_2$.

start

L1: quo=0

L2: r=a

L3: while (b<=r)

L4: quo=quo+1

L5: r=r-b

L6: if(r!=0)

L7: res=0

L8: res=1

end

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

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

\[
\begin{align*}
\text{start} & \\
L1: \quad \text{quo}=0 & \\
L2: \quad r=a & \\
L3: \quad \text{while } (b<=r) & \\
L4: \quad \text{quo}=\text{quo}+1 & \\
L5: \quad r=r-b & \\
L6: \quad \text{if}(r!=0) & \\
L7: \quad \text{res}=0 & \\
L8: \quad \text{res}=1 & \\
\text{end} & \\
\end{align*}
\]

L5 is present on all paths L4 \(\leadsto\) end, but not on L6 \(\leadsto\) 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

- $\textit{obs}(u)$: set of first-reachable nodes \textbf{(observables)} from $u$ in $V'$
Definitions

- $\text{obs}(u)$: 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 $V'$

\[
\begin{align*}
\text{obs}(u) &= \{u_1, u_2\} \\
\text{obs}(v) &= \{u_1, u_2\} \\
\text{obs}(v_1) &= \{u_2\} \\
\text{obs}(v_2) &= \{u_2\} \\
\text{obs}(u_1) &= \{u_1\} \\
\text{obs}(u_2) &= \{u_2\}
\end{align*}
\]
Definitions

- $\text{obs}(u)$: 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

\[
\begin{align*}
\text{obs}(u) &= \{u_1, u_2\} \\
\text{obs}(v) &= \{u_1, u_2\} \\
\text{obs}(v_1) &= \{u_2\} \\
\text{obs}(v_2) &= \{u_2\} \\
\text{obs}(u_1) &= \{u_1\} \\
\text{obs}(u_2) &= \{u_2\}
\end{align*}
\]
Definitions

- \( \text{obs}(u) \): 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 common internal node except their origin

Point of divergence closest to \( V' \)

\[
\text{obs}(u) = \{u_1, u_2\}X \\
\text{obs}(v) = \{u_1, u_2\}X \\
\text{obs}(v_1) = \{u_2\} \checkmark \\
\text{obs}(v_2) = \{u_2\} \checkmark \\
\text{obs}(u_1) = \{u_1\} \checkmark \\
\text{obs}(u_2) = \{u_2\} \checkmark 
\]
Definitions

- \( \text{obs}(u) \): 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. \( \text{control-closure}(V') = V' \cup \{ \text{\( V' \)-deciding vertices} \} \)

\[
\text{obs}(u) = \{u_1, u_2\}
\]
\[
\text{obs}(v) = \{u_1, u_2\}
\]
\[
\text{obs}(v_1) = \{u_2\}
\]
\[
\text{obs}(v_2) = \{u_2\}
\]
\[
\text{obs}(u_1) = \{u_1\}
\]
\[
\text{obs}(u_2) = \{u_2\}
\]
Definitions

• \( \text{obs}(u) \): 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. \( \text{control-closure}(V') = V' \cup \{ \text{\( V' \)-deciding vertices} \} \)

\[
\begin{align*}
\text{obs}(u) &= \{ v \} \\
\text{obs}(v) &= \{ v \} \\
\text{obs}(v_1) &= \{ u_2 \} \\
\text{obs}(v_2) &= \{ u_2 \} \\
\text{obs}(u_1) &= \{ u_1 \} \\
\text{obs}(u_2) &= \{ u_2 \}
\end{align*}
\]
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

\[ V' = \{u_1, u_3\} \]

\[ u_0, u_2 \text{ and } u_4 \text{ are } V'\text{-deciding too} \]
Running example

\[ V' = \{ u_1, u_3 \} \]

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

end

return \( W \)
Danicic’s method to compute control closure

\[
\begin{align*}
\text{begin} & \\
W & \leftarrow V' \\
\text{while} & \text{ there exists a node } u \text{ that is } V' \text{-deciding } \text{ do} \\
& \text{ add all such nodes to } W \\
\text{end} & \\
\text{return } W \\
\text{end}
\end{align*}
\]

Formally:
\[1 \leq |\text{obs}(v)| < |\text{obs}(u)|\]
Danicic’s method to compute control closure

begin
\[ W \leftarrow V'; \]
while \( \text{there exists a node } u \text{ that is } V'-\text{deciding} \) do
\[ \text{add all such nodes to } W \]
end

return \( W \)
end

Formally:
\[ 1 \leq |\text{obs}(v)| < |\text{obs}(u)| \]

with strictly more observables in \( W \) than one of its children \( v \)

\( u \) is a rich parent
\( v \) is a poor child
Rich parent illustrated

\[ \text{obs}(u) = \{u_1, u_2\} \times \]
\[ \text{obs}(v) = \{u_1, u_2\} \times \]
\[ \text{obs}(v_1) = \{u_2\} \checkmark \]
\[ \text{obs}(v_2) = \{u_2\} \checkmark \]
Rich parent 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 illustrated

rich parent (observes $u_1$ and $u_2$)

poor child (observes $u_2$)

poor child (observes $u_2$)

obs($u$) = \{u_1, u_2\} \times
obs(v) = \{u_1, u_2\} \times
obs(v_1) = \{u_2\} \checkmark
obs(v_2) = \{u_2\} \checkmark
Rich parent illustrated

- Rich parent (observes $u_1$ and $u_2$)
- Poor child (observes $u_2$)
- Poor child (observes $u_2$)

$\text{obs}(u) = \{u_1, u_2\}$
$\text{obs}(v) = \{u_1, u_2\}$
$\text{obs}(v_1) = \{u_2\}$
$\text{obs}(v_2) = \{u_2\}$
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 \(1 \leq |\text{obs}(v)| < |\text{obs}(u)|\)

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

\[
\begin{align*}
\{ u_6 \} & \quad \{ u_6 \} \\
\{ u_2 \} & \quad \{ u_0 \} \\
\{ u_1 \} & \quad \{ u_1 \} \\
\{ u_3 \} & \quad \{ u_3 \} \\
\{ u_4 \} & \quad \{ u_4 \} \\
\{ u_0 \} & \quad \{ u_0 \} \\
\end{align*}
\]
Danicic’s algorithm on an example

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

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 brief history of control dependence

Danicic’s theory of control dependence

A new optimized algorithm

Conclusion

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

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
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

$V'$

$u_1$

$u_2$

$u$

$v$

$v_1$

$v_2$

observes $u_1$

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 $u_1$
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 \(u_3 \in obs(u), u_3 \not\in obs(v)\)

identify rich parent/poor child with witness \(u_3\)
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 \(u_0\)

does not observe \(u_0\)
The optimized algorithm on an example

Iteration 3c: update $W$

- add $u_2$
- add $u_4$
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
- 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

start

L1: quo=0

L2: r=a

L3: while (b<=r)

L4: quo=quo+1

L5: r=r-b

L6: if(r!=0)

L7: res=0

L8: res=1

end
Weak control-closure on euclidean division

start

L1: quo=0

L2: r=a

L3: while (b<=r)

L4: quo=quo+1

L5: r=r-b

L6: if(r!=0)

L7: res=0

L8: res=1

end
Graph theory

<table>
<thead>
<tr>
<th></th>
<th>Alt-Ergo (1.30)</th>
<th>CVC4 (1.5)</th>
<th>Coq (8.6.1)</th>
<th>Eprover (2.0)</th>
<th>Z3 (4.5.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>10</td>
<td>14</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Min time (s)</td>
<td>0</td>
<td>0.02</td>
<td>0.27</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>Max time (s)</td>
<td>0.01</td>
<td>0.67</td>
<td>0.37</td>
<td>0.44</td>
<td>0</td>
</tr>
<tr>
<td>Avg time (s)</td>
<td>0.01</td>
<td>0.083</td>
<td>0.3</td>
<td>0.093</td>
<td>N/A</td>
</tr>
</tbody>
</table>

+ 1 axiom (but proved in the Coq formalization)
## Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Alt-Ergo (1.30)</th>
<th>CVC4 (1.5)</th>
<th>Coq (8.6.1)</th>
<th>Eprover (2.0)</th>
<th>Z3 (4.5.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong></td>
<td>233</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td><strong>Min time (s)</strong></td>
<td>0.01</td>
<td>0.08</td>
<td>0.32</td>
<td>0.08</td>
<td>0.34</td>
</tr>
<tr>
<td><strong>Max time (s)</strong></td>
<td>3.96</td>
<td>0.83</td>
<td>0.76</td>
<td>2.35</td>
<td>3.18</td>
</tr>
<tr>
<td><strong>Avg time (s)</strong></td>
<td>0.18</td>
<td>0.46</td>
<td>0.48</td>
<td>0.72</td>
<td>1.76</td>
</tr>
</tbody>
</table>