Formalizing a Framework for Dynamic Slicing of Program Dependence Graphs in Isabelle/HOL

Daniel Wasserrab and Andreas Lochbihler

Universität Karlsruhe (TH), Germany

August 19, 2008

Funded by DFG grant Sn11/10-1

Dynamic Slicing

```
1 sum := 0;
2 prod := 1;
3 while (i>0) {
4   sum := sum+i;
5   prod := prod*i;
6   i := i-1;
  }
7 out:=sum;
```

Task:

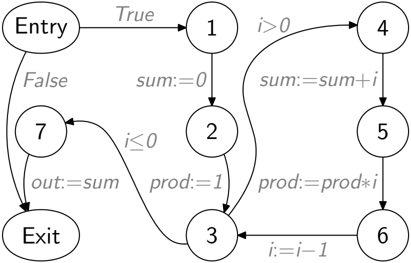
For a given program trace, find all statements that can have *influenced* the last statement s.

$\Rightarrow$ Values used/computed by s

$\Rightarrow$ Execution of s

Dynamic Slicing

```
1 sum := 0;
2 prod := 1;
3 while (i>0) {
4   sum := sum+i;
5   prod := prod*i;
6   i := i-1;
  }
7 out:=sum;
```

CFG: 

Task:
For a given program trace, find all statements that can have *influenced* the last statement *s*.

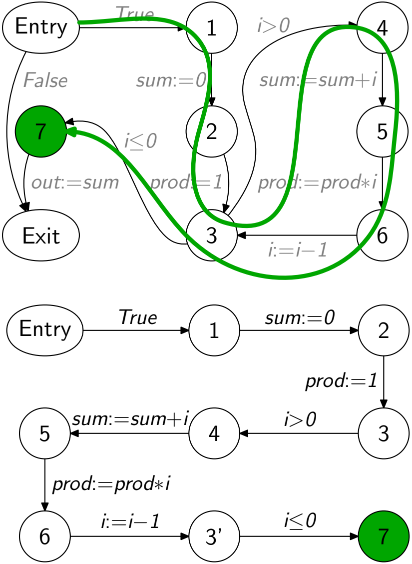
⇒ Values used/computed by *s*

⇒ Execution of *s*

Dynamic Slicing

```
1 sum := 0;
2 prod := 1;
3 while (i>0) {
4   sum := sum+i;
5   prod := prod*i;
6   i := i-1;
  }
7 out:=sum;
```

CFG:



Task:
For a given program trace, find all statements that can have *influenced* the last statement *s*.

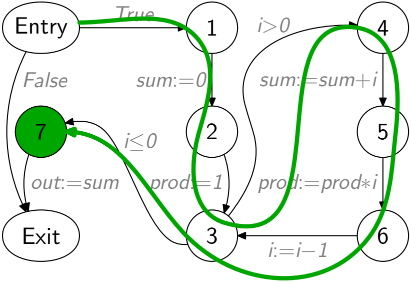
⇒ Values used/computed by *s*

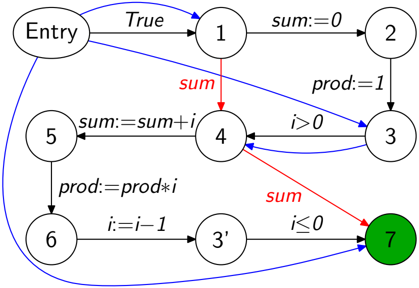
⇒ Execution of *s*

Dynamic Slicing

```
1 sum := 0;
2 prod := 1;
3 while (i>0) {
4   sum := sum+i;
5   prod := prod*i;
6   i := i-1;
  }
7 out:=sum;
```

CFG:





Task:
For a given program trace, find all statements that can have *influenced* the last statement *s*.

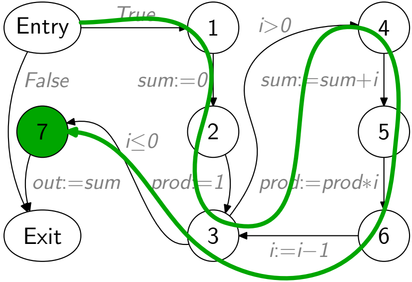
values ⇒ data dependences

execution ⇒ control dependences

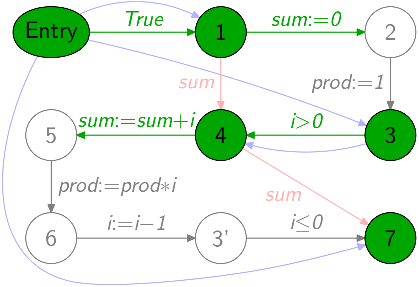
Dynamic Slicing

```
1 sum := 0;
2 prod := 1;
3 while (i>0) {
4   sum := sum+i;
5   prod := prod*i;
6   i := i-1;
  }
7 out:=sum;
```

CFG:



Slice:



Task:

For a given program trace, find all statements that can have *influenced* the last statement *s*.

values ⇒ data dependences

execution ⇒ control dependences

Slicing captures influence

Influence is

- defined in terms of semantics,
- approximated by data and control dependence

Correctness property for slicing:
No other statements affect the values computed at the slicing criterion (or its execution).

Applications of slicing exploit this property:

- Debugging
- Compiler technology
- Software security
- ...

Aim

Previous correctness proofs suffer from

- being only for while language
- depending on specific program languages
- not being machine-checked
- having to be redone for every new programming language

but slicing algorithms are independent of the programming language

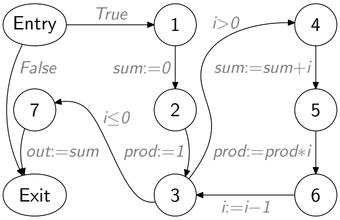
Goal:

Show that no node outside the slice has any semantic influence

1. independent of specific programming languages
2. as modular as possible
3. in Isabelle/HOL

Module: Control Flow Graph

The *control flow graph (CFG)* is the abstract program representation:



Nodes: Set *valid-node* and special nodes *Entry*, *Exit*

Edges: Edge $a \in$ *valid-edge* between *src* a and *trg* a.

Semantics: *kind* labels edges with state predicates or transfer functions

Instantiate for specific programming languages to get:

Paths: $n \ {-as\rightarrow*}\ n'$ runs from n to n' via edges *as*

Execution: *transfer* (*kind* a) s executes a's transfer functions on state s, *pred* (*kind* a) s checks if s satisfies a's predicate; *transfers* and *preds* fold these over lists

Control dependence: n *controls* n' *via as*
Standard (static) control dependence and ($n \ {-as\rightarrow*}\ n'$)

Modelling effects

Model effect of transfer functions and evaluation of predicates:

Def n set of locations that *n*'s edges can affect

Use n set of locations that *n*'s edges can depend on

sval retrieves the location's value in a state

Assume: They correctly model the semantics of edge labels

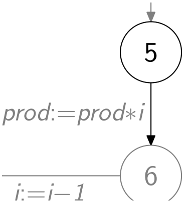
Example:

States Mappings from $\{i, prod, sum\}$ to $\mathbb{Z}$

sval Function application

Use 5 $\{i, prod\}$

Def 5 $\{prod\}$



Well-formedness constraints for modelling effects

1. Affected locations are in *Def*

$$\frac{a \in \mathit{valid\text{-}edge} \qquad V \notin \mathit{Def}\ (\mathit{src}\ a)}{\mathit{sval}\ (\mathit{transfer}\ (\mathit{kind}\ a)\ s)\ V = \mathit{sval}\ s\ V}$$

2. Updates use only declared locations

$$\frac{a \in \mathit{valid\text{-}edge} \qquad \forall\, V {\in} \mathit{Use}\ (\mathit{src}\ a).\ \mathit{sval}\ s\ V = \mathit{sval}\ s'\ V \qquad V \in \mathit{Def}\ (\mathit{src}\ a)}{\mathit{sval}\ (\mathit{transfer}\ (\mathit{kind}\ a)\ s)\ V = \mathit{sval}\ (\mathit{transfer}\ (\mathit{kind}\ a)\ s')\ V}$$

3. Predicates depend only on used locations

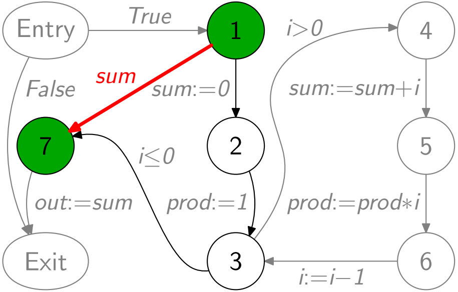
$$\frac{a \in \mathit{valid\text{-}edge} \qquad \forall\, V {\in} \mathit{Use}\ (\mathit{src}\ a).\ \mathit{sval}\ s\ V = \mathit{sval}\ s'\ V}{\mathit{pred}\ (\mathit{kind}\ a)\ s = \mathit{pred}\ (\mathit{kind}\ a)\ s'}$$

Data dependence

Dynamic data dependence

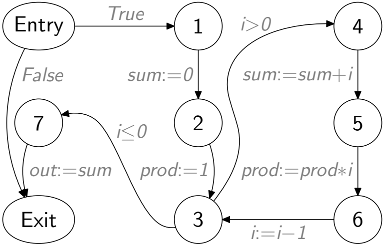
n influences V in n' via as:

$V \in Def\ n$	n defines location V
$V \in Use\ n'$	n' uses V, and
n $-as\rightarrow*$ n'	Nodes inside *as* do not define V inbetween.



Program dependence graph

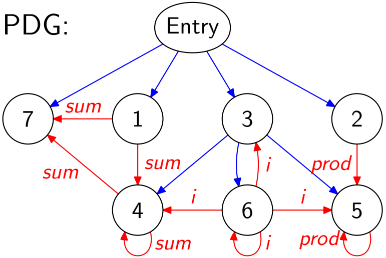
Combine control and data dependences in the *program dependence graph (PDG)* to get dependence paths $n \ -as{\rightarrow_d}* \ n'$



Dynamic PDG / slicing:

- *Remember* CFG paths in dependence edges
- Match program trace with path information

PDG:



Static PDG / slicing:

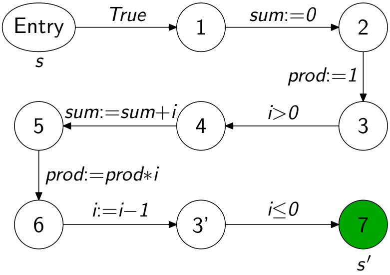
- *Abstract* from CFG paths in dependence edges

$\Rightarrow$ Reachability analysis on the PDG

- Overapproximates dynamic slices

Formal correctness statement

1. Take an executable program trace $n \ -as\!\rightarrow\!*\ n'$ with initial state s and final state $s' = \mathit{transfers}\ (\mathit{kinds}\ \mathit{as})\ s$.
2. Compute dynamic slice bs for as
3. For all nodes not in bs, replace outgoing transfer functions with no-ops and predicates with True, to get as'.



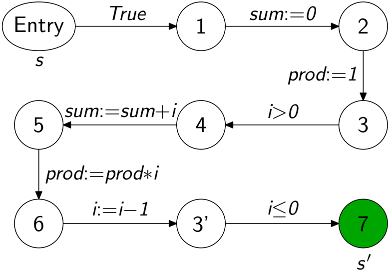
Then $\mathit{preds}\ (\mathit{kinds}\ \mathit{as}')\ s$,
i.e. as' is executable,

and in the resulting state
$s'' = \mathit{transfers}\ (\mathit{kinds}\ \mathit{as}')\ s$:
$\mathit{sval}\ s'\ V = \mathit{sval}\ s''\ V$ for all $V \in \mathit{Use}\ n'$

Proof: Induction on as

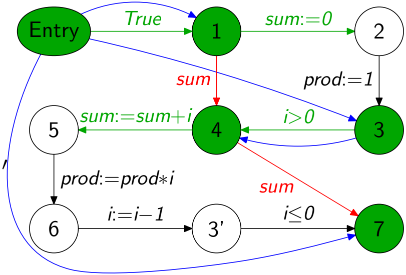
Formal correctness statement

1. Take an executable program trace $n \xrightarrow{as}{}^{*}\, n'$ with initial state s and final state $s' = \mathit{transfers}\ (\mathit{kinds}\ as)\ s$.
2. Compute dynamic slice bs for as
3. For all nodes not in bs, replace outgoing transfer functions with no-ops and predicates with *True*, to get as'.



Then $\mathit{preds}\ (\mathit{kinds}\ as')\ s$,
i.e. as' is executable,

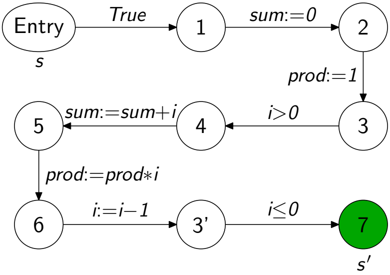
and in the resulting state
$s'' = \mathit{transfers}\ (\mathit{kinds}\ as')\ s$:
$\mathit{sval}\ s'\ V = \mathit{sval}\ s''\ V$ for all $V \in \mathit{Use}\ n'$



Proof: Induction on *as*

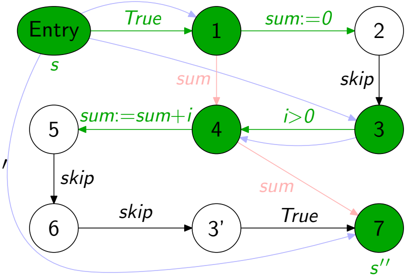
Formal correctness statement

1. Take an executable program trace $n \ -as\rightarrow* \ n'$ with initial state s and final state $s' = \mathit{transfers}\ (\mathit{kinds}\ \mathit{as})\ s$.
2. Compute dynamic slice bs for as
3. For all nodes not in bs, replace outgoing transfer functions with no-ops and predicates with *True*, to get as'.



Then $\mathit{preds}\ (\mathit{kinds}\ \mathit{as}')\ s$,
i.e. as' is executable,

and in the resulting state
$s'' = \mathit{transfers}\ (\mathit{kinds}\ \mathit{as}')\ s$:
$\mathit{sval}\ s'\ V = \mathit{sval}\ s''\ V$ for all $V \in \mathit{Use}\ n'$



Proof: Induction on as

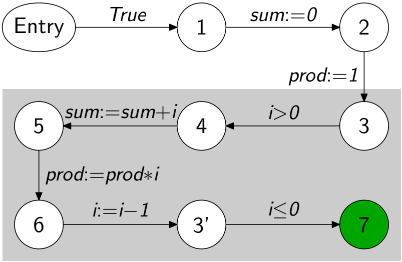
Formal correctness statement

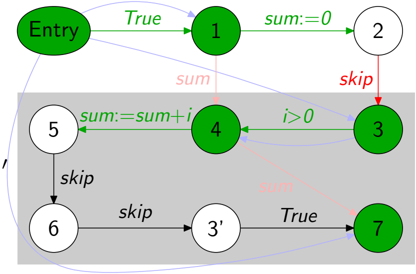
1. Take an executable program trace $n \,-as\!\rightarrow\!*\; n'$ with initial state s and final state $s' = \mathit{transfers}\ (\mathit{kinds}\ as)\ s$.
2. Compute dynamic slice bs for as
3. For all nodes not in bs, replace outgoing transfer functions with no-ops and predicates with *True*, to get as'.

Then $\mathit{preds}\ (\mathit{kinds}\ as')\ s$,
i.e. as' is executable,

and in the resulting state
$s'' = \mathit{transfers}\ (\mathit{kinds}\ as')\ s$:
$\mathit{sval}\ s'\ V = \mathit{sval}\ s''\ V$ for all $V \in \mathit{Use}\ n'$

Proof: Induction on *as* fails!





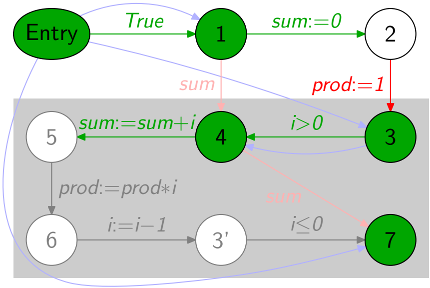
Dependent live variables

Live variable analysis (LVA):
What variables (locations) are used in the trace before being defined again?

Dependent live variables (DLV):
Consider *Def*/*Use* sets of non-slice nodes to be empty for LVA

Induction invariant:

1. s_1 and s_2 agree on the (current) set of DLV
2. Execute the original and sliced trace one step each for s_1 and s_2
3. Then, the resulting states agree on the (new) DLV set again



For the trace [*3*, *4*, *5*, *6*, *3*, *7*]:

Live variables: *i*, *prod*, *sum*

Dependent live variables: *i*, *sum*

Strengthened correctness statement for slicing

1. Take an executable program trace $n \ -as\!\rightarrow\!*\ n'$ with initial state s_1 and final state $s_1{}' = \mathit{transfers}\ (\mathit{kinds}\ \mathit{as})\ s_1$.
2. Let s_2 agree with s_1 on DLV of *as*.
3. Compute dynamic slice *bs* for *as*
4. For all nodes not in *bs*, replace outgoing transfer functions with no-ops and predicates with *True*, to get *as*′.

Then $\mathit{preds}\ (\mathit{kinds}\ \mathit{as}')\ s_2$, i.e. *as*′ is executable,

and in the resulting state $s_2{}' = \mathit{transfers}\ (\mathit{kinds}\ \mathit{as}')\ s_2$:
$\mathit{sval}\ s_1{}'\ V = \mathit{sval}\ s_2{}'\ V$ for all $V \in \mathit{Use}\ n'$

$$\frac{\begin{array}{c} n \ -as\!\rightarrow\!*\ n' \qquad bs \preceq_b bs' \qquad \textit{slice-path as} = bs \\ \textit{select-edge-kinds as bs} = es \qquad \textit{select-edge-kinds as bs}' = es' \qquad \textit{preds } es'\ s' \\ \forall\, V\ xs.\ (V,\ xs,\ as) \in \textit{dependent-live-vars } n' \longrightarrow \textit{sval } s\ V = \textit{sval } s'\ V \end{array}}{\textit{preds es s} \qquad \forall\, V {\in} \textit{Use } n'.\ \textit{sval}\ (\textit{transfers es s})\ V = \textit{sval}\ (\textit{transfers } es'\ s')\ V}$$

Summary

Framework for dynamic slicing based on CFGs/PDGs

- Generic correctness proof
- Instantiable for specific programming languages
- Highly modularized

Context: **Quis custodiet project**

- Generic framework for slicing in Isabelle/HOL
  - Different control dependences $\surd$
  - Static intraprocedural slicing $\surd$
  - Static interprocedural slicing future work
  - Instantiated for a While language $\surd$
  - Realistic languages (Jinja, CoreC++) future work
- Verifying software security analyses / algorithms