Systeme hoher Sicherheit und Qualität
Universität Bremen, WS 2017/2018

Lecture 08:

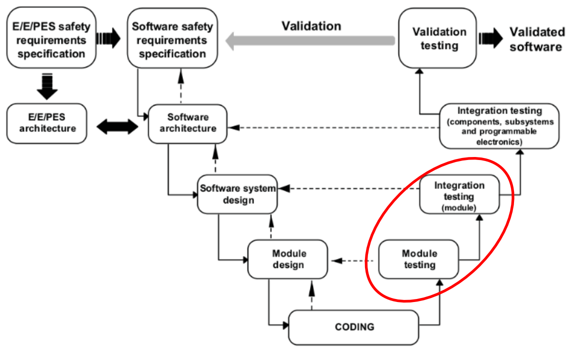
Static Program Analysis

Christoph Lüth, Dieter Hutter, Jan Peleska

Universität Bremen

Where are we?

Program Analysis in the Development Cycle



Static Program Analysis

- Analysis of run-time behaviour of programs **without executing them** (sometimes called static testing).
- Analysis is done for **all** possible runs of a program (i.e. considering all possible inputs).
- Typical questions answered:
  - Does the variable *x* have a constant value ?
  - Is the value of the variable *x* always positive ?
  - Are all pointer dereferences valid (or NULL)?
  - Are all arithmetic operations well-defined?
- These tasks can be used for **verification** or for **optimization** when compiling.

Usage of Program Analysis

Optimizing compilers
- Detection of sub-expressions that are evaluated multiple times
- Detection of unused local variables
- Pipeline optimizations

Program verification

Search for runtime errors in programs (program safety):
- Null pointer or other illegal pointer dereferences
- Array access out of bounds
- Exceptions which are thrown and not caught
- Division by zero
- Over/underflow of integers, rounding errors with floating point numbers
- Runtime estimation (worst-caste executing time, wcet)

In other words, **specific** verification **aspects.**

Program Analysis: The Basic Problem

Given a property P and a program p: $p \models P$ iff P holds for p

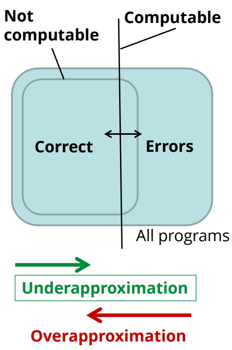
- Wanted: a terminating algorithm $\phi(p, P)$ which computes $p \models P$
  - ϕ is sound if $\phi(p, P)$implies $p \models P$
  - ϕ is complete if $\neg\phi(p, P)$ implies $\neg\, p \models P$
  - If ϕ is sound and complete then ϕ is a decision procedure

> The **basic problem** of static program analysis: virtually all interesting program properties are **undecidable**! (cf. Gödel, Turing)

- From the basic problem it follows that there are no sound and complete tools for interesting properties.
- Tools for interesting properties are either
  - sound (under-approximating) or
  - complete (over-approximating).

Program Analysis: Approximation

- **Under-approximation** is sound but not complete. It only finds correct programs but may miss out some.
  - Useful in **optimizing compilers;**
  - Optimization must preserve semantics of program, but is optional.
- **Over-approximation** is complete but not sound. It finds all errors but may find non-errors (false positives).
  - Useful in verification;
  - Safety analysis must find all errors, but may report some more.
  - Too high rate of false positives may hinder acceptance of tool.



Program Analysis Approach

- Provides **approximate** answers
  - yes / no / don't know or
  - superset or subset of values
- Uses an **abstraction** of program's behavior
  - Abstract data values (e.g. sign abstraction)
  - Summarization of information from execution paths e.g. branches of the if-else statement
- **Worst-case** assumptions about environment's behavior
  - e.g. any value of a method parameter is possible.
- Sufficient **precision** with good **performance.**

Analysis Properties: Flow Sensitivity

Flow-insensitive analysis

- Program is seen as an unordered collection of statements
- Results are valid for any order of statements
  e.g. $S_1 ; S_2$ vs. $S_2 ; S_1$
- Example: type analysis (inference)

Flow-sensitive analysis

- Considers program's flow of control
- Uses control-flow graph as a representation of the source
- Example: available expressions analysis

Analysis Properties: Context Sensitivity

Context-sensitive analysis

- Stack of procedure invocations and return values of method parameters
- Results of analysis of the method *M* depend on the caller of *M*

Context-insensitive analysis

- Produces the same results for all possible invocations of *M* independent of possible callers and parameter values.

Intra- vs. Inter-procedural Analysis

Intra-procedural analysis

- Single function is analyzed in isolation.
- Maximally pessimistic assumptions about parameter values and results of procedure calls.

Inter-procedural analysis

- Procedure calls are considered.
- Whole program is analyzed at once.

Data-Flow Analysis

Focus on questions related to values of variables and their lifetime

Selected analyses:

- **Available expressions (forward analysis)**
  - Which expressions have been computed already without change of the occurring variables (optimization) ?
- **Reaching definitions (forward analysis)**
  - Which assignments contribute to a state in a program point? (verification)
- **Very busy expressions (backward analysis)**
  - Which expressions are executed in a block regardless which path the program takes (verification) ?
- **Live variables (backward analysis)**
  - Is the value of a variable in a program point used in a later part of the program (optimization) ?

A Simple Programming Language

- **Arithmetic** expressions:

  $$a \;::=\; x \mid n \mid a_1\; op_a\; a_2$$

  - Arithmetic operators: $op_a \in \{+,-,*,/\}$
- **Boolean** expressions:

  $$b \;:=\; \text{true} \mid \text{false} \mid \text{not } b \mid b_1 op_b\; b_2 \mid a_1 op_r\; a_2$$

  - Boolean operators: $op_b \in \{and, or\}$
  - Relational operators: $op_r \in \{=,<,\leq,>,\geq,\neq\}$
- **Statements**:

  $$S ::= [x := a]^l \mid [\textbf{skip}]^l \mid S1; S2 \mid \textbf{if}\, [b]^l\;\; S1 \;\textbf{else}\; S2 \mid \textbf{while}\; [b]^l\, S$$

- Note this abstract syntax, operator precedence and grouping statements is not covered. We can use { and } to group statements, and (and) to group expressions.

Computing the Control Flow Graph

- To calculate the CFG, we define some functions on the abstract syntax S :
  - The initial label (entry point)
    init: $S \rightarrow Lab$

    $init([x \coloneqq a]^l) = l$
    $init([skip]^l) = l$
    $init\ (S_1; S_2) = init\ (S_1)$
    $init\ (if\ [b]^l\ \{S_1\}\ else\ \{S_2\} = l$
    $init\ (while\ [b]^l\ \{\ S\ \} = l$

  - The final labels (exit points)
    final: $S \rightarrow \mathbb{P}(Lab)$

    $final([x \coloneqq a]^l) = \{l\}$
    $final\ ([skip]^l) = \{l\}$
    $final\ (S_1; S_2) = final\ (S_2)$
    $final(if\ [b]^l\ \{\ S_1\} else\ \{S_2\})$
    $= final\ (S_1) \cup final\ (S_2)$
    $final(while\ [b]^l\ \{S\}\ ) = \{l\}$

  - The elementary blocks
    $blocks: S \rightarrow \mathbb{P}(Blocks)$ where an elementary block is an assignment [x:= a], or [skip], or a test [b]

    $blocks([x \coloneqq a]^l) = \{[x \coloneqq a]^l\}$
    $blocks([skip]^l) = \{\ [skip]^l\}$
    $blocks(S_1; S_2) = blocks(S_1) \cup blocks(S_2)$
    $blocks(if\ [b]^l\ \{S_1\}\ else\ \{S_2\})$
    $= \{[b]^l\ \} \cup blocks(S_1) \cup blocks(S_2)$
    $blocks(while\ [b]^l\ \{S\}) = \{[b]^l\} \cup blocks(S)$

Computing the Control Flow Graph

- The control flow $\quad$ flow: $S \rightarrow \mathbb{P}(Lab \times Lab)$
  and reverse control $\quad$ $\text{flow}^{\text{R}}: S \rightarrow \mathbb{P}(Lab \times Lab)$

  $flow\ ([x \coloneqq a]^l) = \emptyset$
  $flow\ ([skip]^l) = \emptyset$
  $flow\ (S_1; S_2) = flow\ (S_1) \cup flow\ (S_2) \cup \{(l, init(S_2)) \mid l \in final(S_1)\}$
  $flow\ (if\ [b]^l\ \{S_1\}\ else\ \{S_2\ \}\ ) = flow\ (S_1) \cup flow(S_2) \cup \{(l, init(S_1)), (l, init(S_2))\}$
  $flow\ (while\ ([b]^l\ \{S\}\ ) = flow(S) \cup \{(l, init\ (S))\} \cup \{(l', l) | l' \in final(S)\}$

  $flow^R(S) = \{(l', l) |\ (l, l') \in flow(S)\}$

- The **control flow graph** of a program S is given by
  - elementary blocks $block(S)$ as nodes, and
  - *flow(S)* as vertices.
- Additional useful definitions

  $labels(S) = \{l \mid [B]^l \in blocks(S)\}$
  $FV(a)$ = free variables in a
  $Aexp(S)$ = non-trivial subexpressions in S (variables and constants are trivial)

An Example Program

P = [x := a+b][1]; [y := a*b][2]; while [y > a+b][3] { [a:=a+1][4]; [x:= a+b][5] }

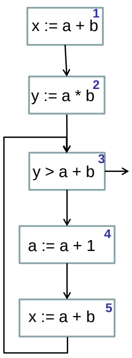
init(P) = 1
final(P) = {3}

blocks(P) =
{ [x := a+b][1], [y := a*b][2], [y > a+b][3], [a:=a+1][4], [x:= a+b][5]}

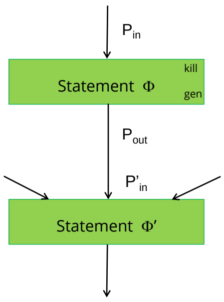
flow(P) = {(1, 2), (2, 3), (3, 4), (4, 5), (5, 3)}
flow^R(P) = {(2, 1), (3, 2), (4, 3), (5, 4), (3, 5)}

labels(P)= {1, 2, 3, 4, 5)

FV(a+b) = {a, b}
FV(P) = {a, b, x, y}
Aexp(P) = {a+b, a*b, a+1}



Program Analysis CFG : General Idea



Locally for each statement:

Relationship between P_{in} and P_{out} :

- kill : part of P_{in} that is invalidated by Φ
- gen : additional part that is generated by Φ

$P_{out} = (P_{in} \setminus \text{kill}) \cup \text{gen}$

Globally for each link:

$P'_{in} = \cup P_{out}$ (or $\cap P_{out}$)

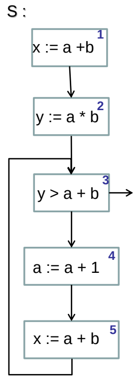
We obtain constrains for the P_{out} and P_{in} for all statements and links!
Solve CSP by a constraint solver.

Available Expression Analysis

- The available expression analysis will determine for each program point:

  which non-trivial expressions have been already computed in prior statements (and are still valid)

  „Caching of expressions“



Available Expression Analysis

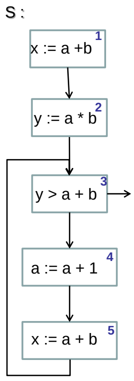
gen([x :=a]^l)	=	*{ exp ∈ Aexp(a) | x ∉ FV(exp) }*
gen([skip]^l)	=	∅
gen([b]^l)	=	*Aexp(b)*
kill([x :=a]^l)	=	*{ exp ∈ Aexp(S) | x ∈ FV(exp) }*
kill([skip]^l)	=	∅
kill([b]^l)	=	∅

$$AE_{in}(l) = \begin{cases} \emptyset, \text{ if } l \in init(S) \\ \cap \{ AE_{out}(l') \mid (l',l) \in flow(S)\}, \text{ otherwise} \end{cases}$$

$$AE_{out}(l) = \left(AE_{in}(l) \setminus kill(B^l)\right) \cup gen(B^l), \text{ where } B^l \in blocks(S)$$

l	$kill(B^l)$	$gen(B^l)$
1	∅	{a+b}
2	∅	{a*b}
3	∅	{a+b}
4	{a+b, a*b, a+1}	∅
5	∅	{a+b}

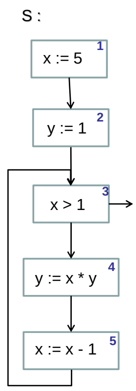
l	AE_{in}	AE_{out}
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}



Reaching Definitions Analysis

- Reaching definitions (assignment) analysis determines if:
  - An assignment of the form *[x := a]^l* reaches a program point *k*

    if **there is** an execution path where *x* was last assigned at *l* when the program reaches *k*



Reaching Definitions Analysis

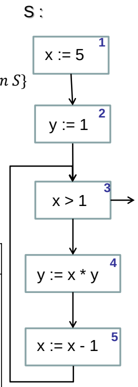
gen([x :=a]^l) = $\{ (x,l)\}$ *kill([skip]^l)* = ∅
gen([skip]^l) = ∅ *kill([b]^l)* = ∅
gen([b]^l) = ∅ *kill([x :=a]^l)* =
$\{(x,?)\} \cup \{ (x,k) \mid B^k \text{ is an assigment in } S\}$

$$RD_{in}(l) = \begin{cases} \{ (x,?) \mid x \in FV(S)\} \text{ if } l \in init(S) \\ \cup \{ RD_{out}(l') \mid (l',l)\} \in flow(S) \text{ otherwise} \end{cases}$$

$$RD_{out}(l) = \left(RD_{in}(l) \setminus kill(B^l)\right) \cup gen(B^l) \text{ where } B^l \in blocks(S)$$

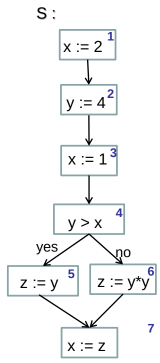
l	$kill(B^l)$	$gen(B^l)$
1	{(x,?), (x,1),(x,5)}	{(x, 1)}
2	{(y,?), (y,2),(y,4)}	{(y, 2)}
3	∅	∅
4	{(y,?), (y,2),(y,4)}	{(y, 4)}
5	{(x,?), (x,1),(x,5)}	{(x, 5)}

l	RD_{in}	RD_{out}
1	{(x,?), (y,?)}	{(x,1), (y,?)}
2	{(x,1), (y,?)}	{(x,1), (y,2)}
3	{(x,1), (x,5), (y,2), (y,4)}	{(x,1), (x,5), (y,2), (y,4)}
4	{(x,1), (x,5), (y,2), (y,4)}	{(x,1), (x,5),(y,4)}
5	{(x,1), (x,5),(y,4)}	{(x,5),(y,4)}



Live Variables Analysis

- A variable x is **live** at some program point (label *l*) if there exists if there exists a path from *l* to an exit point that does not change the variable
- Live Variables Analysis determines:
  - for each program point, which variables *may* be still live at the exit from that point.
- Application: dead code elemination.



Live Variables Analysis

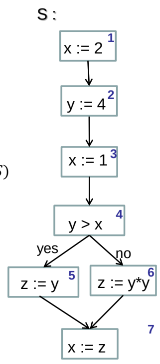
gen([x :=a]^l) = $FV(a)$ *kill([x :=a]^l)* = $\{x\}$
gen([skip]^l) = ∅ *kill([skip]^l)* = ∅
gen([b]^l) = $FV(b)$ *kill([b]^l)* = ∅

$$LV_{out}(l) = \begin{cases} \emptyset \quad \text{if } l \in final(S) \\ \cup \{LV_{in}(l') \mid (l',l) \in flow^R(S)\} \text{ otherwise} \end{cases}$$

$$LV_{in}(l) = \left(LV_{out}(l) \setminus kill(B^l)\right) \cup gen(B^l) \text{ where } B^l \in blocks(S)$$

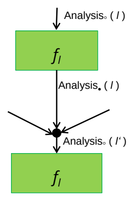
l	$kill(B^l)$	$gen(B^l)$
1	{x}	∅
2	{y}	∅
3	{x}	∅
4	∅	{x, y}
5	{z}	{y}
6	{z}	{y}
7	{x}	{z}

l	LV_{in}	LV_{out}
1	∅	∅
2	∅	{y}
3	{y}	{x, y}
4	{x, y}	{y}
5	{y}	{z}
6	{y}	{z}
7	{z}	∅



First Generalized Schema

- $\text{Analysis}_{\circ}(l) = \begin{cases} \mathbf{EV} \text{ if } l \in \mathbf{E} \\ \square\{\text{Analysis}_{\bullet}(l') \mid (l',l) \in \mathbf{Flow}(S)\} \text{ otherwise} \end{cases}$
- $\text{Analysis}_{\bullet}(l) = f_l(\text{Analysis}_{\circ}(l))$



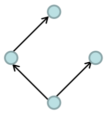
With:

- **EV** is the initial / final analysis information
- **E** is either {init(S)} or final(S)
- □ is either ⋃ or ⋂
- **Flow** is either flow or $flow^R$
- f_l is the transfer function associated with $B^l \in blocks(S)$

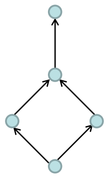
Forward analysis: **Flow** = flow, • = OUT, ∘ = IN
Backward analysis: **Flow** = $flow^R$, • = IN, ∘ = OUT

Partial Order

- $L = (M, \sqsubseteq)$ is a partial order iff
  - Reflexivity: $\forall x \in M. x \sqsubseteq x$
  - Transitivity: $\forall x, y, z \in M. x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - Anti-symmetry: $\forall x, y \in M. x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$



- Let $L = (M, \sqsubseteq)$ be a partial order, $S \subseteq M$
  - $y \in M$ is upper bound for S ($S \sqsubseteq y$) iff $\forall x \in S. x \sqsubseteq y$
  - $y \in M$ is lower bound for S ($y \sqsubseteq S$) iff $\forall x \in S. y \sqsubseteq x$
  - Least upper bound $\bigsqcup X \in M$ of $X \subseteq M$:
    - $X \sqsubseteq \bigsqcup X \land \forall y \in M. X \sqsubseteq y \Rightarrow \bigsqcup X \sqsubseteq y$
  - Greatest lower bound $\sqcap X$ of $X \subseteq M$:
    - $\sqcap X \sqsubseteq X \land \forall y \in M. y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$



Lattice

A **lattice** ("Verband") is a partial order L = (M, $\sqsubseteq$) such that

(1) $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq L$

(2) Unique greatest element $\top = \bigsqcup L$
(3) Unique least element $\bot = \sqcap L$

(1) Alternatively (for finite M), binary operators $\sqcup$ and $\sqcap$ ("meet" and "join") such that
$x, y \sqsubseteq x \sqcup y$ and $x \sqcap y \sqsubseteq x, y$

Transfer Functions

- Transfer functions to propagate information along the execution path (i.e. from input to output, or vice versa)
- Let $L = (M, \sqsubseteq)$ be a lattice. Let F be the set of transfer functions of the form
  $f_l : M \rightarrow M$ with l being a label
- Knowledge transfer is monotone
  - $\forall x, y.\ x \sqsubseteq y \Longrightarrow f_l(x) \sqsubseteq f_l(y)$
- Space $\boldsymbol{F}$ of transfer functions
  - $\boldsymbol{F}$ contains all transfer functions f_l
  - $\boldsymbol{F}$ contains the identity function id $\forall x \in M.\ id(x) = x$
  - $\boldsymbol{F}$ is closed under composition $\forall f, g \in \boldsymbol{F}.\ (g \circ f) \in \boldsymbol{F}$

The Generalized Analysis

- Analysis$_\circ$(l) = $\bigsqcup$ {Analysis$_\bullet$(l') | (l', l) ∈ F} ⊔ { ι_E^l }

  with $\iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{otherwise} \end{cases}$

- Analysis$_\bullet$(l) = f_l(Analysis$_\circ$(l))

With:

- M property space representing data flow information with $(M, \sqsubseteq)$ being a lattice
- A space F of transfer functions f_l and a mapping f from labels to transfer functions in F
- F is a finite flow (i.e. $flow$ or $flow^R$)
- ι is an extremal value for the extremal labels E (i.e. $\{init(S)\}$ or $final(S)$)

Instances of Framework

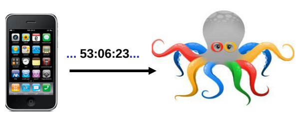
	Available Expr.	Reaching Def.	Live Vars.
M	$\mathcal{P}$(AExpr)	$\mathcal{P}$(Var x L)	$\mathcal{P}$(Var)
$\sqsubseteq$	$\supseteq$	$\subseteq$	$\subseteq$
$\sqcup$	$\cap$	$\cup$	$\cup$
$\bot$	AExpr	$\emptyset$	$\emptyset$
ι	$\emptyset$	{(x, ?) | x ∈ FV(S)}	$\emptyset$
E	{ init(S) }	{ init(S) }	final(S)
F	flow(S)	flow(S)	flow^R(S)
$\boldsymbol{F}$	{ f : $M \rightarrow M$ | ∃ m_k, m_g. f(m) = (m \ m_k) ∪ m_g }		
f_l	f_l(m) = (m \ kill(B^l)) ∪ gen(B^l) where B^l ∈ blocks(S)		

Limitations of Data Flow Analysis

- The general framework of data flow analysis treats all outgoing edges **uniformly**. This can be a problem if conditions influence the property we want to analyse.
- Example: show no division by 0 can occur.
- Property space:
  - $M_0 = \{\bot, \{0\}, \{1\}, \{0,1\}\}$ (ordered by inclusion)
  - $M = Loc \rightarrow M_0$ (ordered pointwise)
  - $app_\sigma(t) \in M_0$ „approximate evaluation" of t under $\sigma \in M$
  - $cond_\sigma(b) \in M$ strengthening of $\sigma \in M$ under condition b
  - $gen\ [x = a] = \sigma[x \mapsto app_\sigma(a)]$
  - Kill needs to distinguish wether cond'n holds:
    $kill[b]_\sigma^{if} = cond_\sigma(b)$ $kill[b]_\sigma^{then} = cond_\sigma(!\ b)$
- This leads us to **abstract interpretation.**

Program Analysis for Information Flow Control

Confidentiality as a property of dependencies:



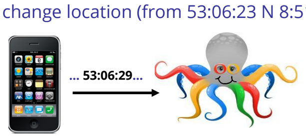
- The GPS data 53:06:23 N 8:51:08 O is confidential.
- The information on the GPS data must not leave Bob's mobile phone
- First idea: 53:06:23 N 8:51:08 O does not appear (explicitly) on the output line.
  - too strong, too weak
- Instead: The output of Bob's smart phone does not **depend** on the GPS setting
  - Changing the location (e.g. to 53:06:29 N 8:51:04 O) will not change the observed output of Bob's smart phone

Note: Confidentiality is formalized as a notion of dependability.

Confidentiality as Dependability

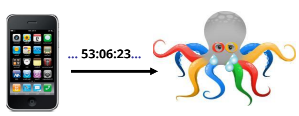
Confidential action:

change location (from 53:06:23 N 8:51:08 O) to 53:06:29 N 8:51:04 O



Insecure system:
output 53:06:29 depends on GPS data

Secure System:
output 53:06:23 does not depend on GPS data

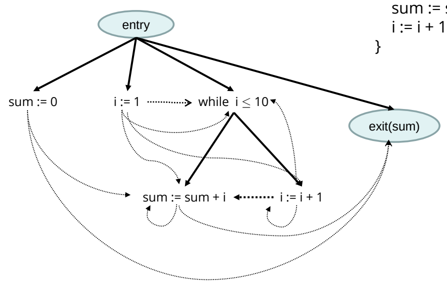


Program Slicing

- Which parts of the program compute the message ?
- Do these parts contain GPS data ?
  - If yes: GPS data influence message (data leak)
  - If no: message is independent of GPS data
- Program Dependence Graph
  - Nodes are statements and conditions of a program
  - Links are either
    - Control dependences (similar to CFG)
    - Data flow dependences (connecting assignment with usage of variables)

Example

→ Control dependences
⇢ Data flow dependences



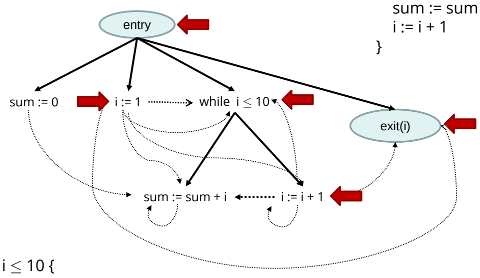

```
sum := 0;
i := 1;
while i ≤ 10 {
   sum := sum + i;
   i := i + 1
}
```


Backward Slice

- Let G be a program dependency graph and
- S be subset of nodes in G
- Let $n \Rightarrow m := n \rightarrow m \vee n \dashrightarrow m$
- Then, the backward slice BS(G, S) is a graph G' with
  - $N(G') = \{ n \mid n \in N(G) \wedge \exists m \in S.\ n \Rightarrow^* m \}$
  - $E(G') = \{n \rightarrow m \mid n \rightarrow m \in E(G) \wedge n, m \in N(G') \} \cup$
    $\{n \dashrightarrow m \mid n \dashrightarrow m \in E(G) \wedge n, m \in N(G') \}$
- Backward slice BS(G, S) computes same values for variables occurring in S as G itself

Example

→ Control dependences
⇢ Data flow dependences




```
sum := 0;
i := 1;
while i ≤ 10 {
   sum := sum + i;
   i := i + 1
}
```

BS:

```
i := 1;
while i ≤ 10 {
   i := i + 1
}
```


Summary

- Static Program Analysis is the analysis of run-time behavior of programs without executing them (sometimes called static testing)
- Approximations of program behaviors by analyzing the program's CFG
- Analysis include
  - available expressions analysis
  - reaching definitions
  - live variables analysis
  - program slicing
- These are instances of a more general framework
- These techniques are used commercially, e.g.
  - AbsInt aiT (WCET)
  - Astrée Static Analyzer (C program safety)