Verifikation von C-Programmen
Universität Bremen, WS 2014/15

Lecture 05 (19.11.2013)

Statische Programmanalyse

Christoph Lüth

Today: Static Program Analysis

- Analysis of run-time behavior 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 tasks
  - Does the variable *x* have a constant value ?
  - Is the value of the variable *x* always positive ?
  - Can the pointer *p* be null at a given program point ?
  - What are the possible values of the variable *y* ?
- These tasks can be used for verification (e.g. is there any possible dereferencing of the null pointer), or for optimisation when compiling.

Usage of Program Analysis

Optimising compilers

- Detection of sub-expressions that are evaluated multiple times
- Detection of unused local variables
- Pipeline optimisations

Program verification

- Search for runtime errors in programs
- Null pointer dereference
- Exceptions which are thrown and not caught
- Over/underflow of integers, rounding errors with floating point numbers
- Runtime estimation (worst-caste executing time, wcet; *AbsInt* tool)

Program Analysis: The Basic Problem

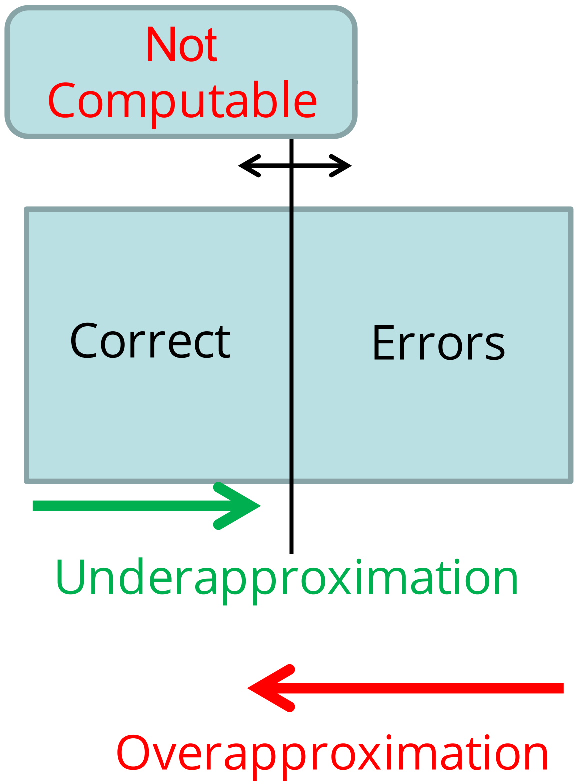
- Basic Problem:

> All interesting program properties are undecidable.

- Given a property P and a program p, we say $p \vDash P$ if a P holds for p. An algorithm (tool) ϕ which decides P is a computable predicate $\phi\colon p \to Bool$. We say:
  - ϕ is **sound** if whenever $\phi(p)$ then $p \vDash P$.
  - ϕ is **safe** (or **complete**) if whenever $p \vDash P$ then $\phi(p)$.
- From the basic problem it follows that there are no sound and safe tools for interesting properties.
  - In other words, all tools must either under- or overapproximate.

Program Analysis: Approximation

- **Underapproximation** only finds correct programs but may miss out some
  - Useful in optimising compilers
  - Optimisation must respect semantics of program, but may optimise.
- **Overapproximation** 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

Flow Sensitivity

Flow-sensitive analysis

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

Flow-insensitive analysis

- Program is seen as an unordered collection of statements
- Results are valid for any order of statements
  e.g. *S1* ; *S2* vs. *S2* ; *S1*
- Example: type analysis (inference)

Context Sensitivity

Context-sensitive analysis

- Stack of procedure invocations and return values of method parameters
  then 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

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

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 Very Simple Programming Language

- In the following, we use a very simple language with
  - Arithmetic operators given by
    $$a \ ::= x \mid n \mid a_1 \ op_a \ a_2$$
    with x a variable, n a numeral, op_a arith. op. (e.g. +, -, *)
  - Boolean operators given by
    $$b \ \coloneqq \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \, op_b \ b_2 \mid a_1 \, op_r \ a_2$$
    with op_b boolean operator (e.g. and, or) and op_r a relational operator (e.g. =, <)
  - Statements given by
    $$S \ ::= \ [x \coloneqq a]^l \mid [\text{skip}]^l \mid S_1; S_2 \mid \text{if } [b]^l \text{then } S_1 \text{else } S_2 \mid \text{while } [b]^l \text{do } S$$
- An Example Program:

```
[x := a+b]^1;
[y := a*b]^2;
while [y > a+b]^3 do ( [a:=a+1]^4; [x:= a+b]^5 )
```

The Control Flow Graph

- We define some functions on the abstract syntax:
  - The initial label (entry point) $\text{init}: S \rightarrow Lab$
  - The final labels (exit points) $\text{final}: S \rightarrow \mathbb{P}(Lab)$
  - The elementary blocks $\text{block}: S \rightarrow \mathbb{P}(Blocks)$ where an elementary block is
    - an assignment [x:= a],
    - or [skip],
    - or a test [b]
  - The control flow $\text{flow}: S \rightarrow \mathbb{P}(Lab \times Lab)$ and reverse control $\text{flow}^{R}: S \rightarrow \mathbb{P}(Lab \times Lab)$.
- The **control flow graph** of a program S is given by
  - elementary blocks $\text{block}(S)$ as nodes, and
  - flow(S) as vertices.

Labels, Blocks, Flows: Definitions

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

init($[x := a]^l$) = l
init($[skip]^l$) = l
init(S_1; S_2) = init(S_1)
init(if $[b]^l$ then S_1 else S_2) = l
init(while $[b]^l$ do S) = l

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

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

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

labels(S) = { l | $[B]^l \in$ blocks(S)}
FV(a) = free variables in a
Aexp(S) = nontrivial subexpressions of S

Another Example

P = $[x := a+b]^1$; $[y := a*b]^2$; while $[y > a+b]^3$ do ($[a:=a+1]^4$; $[x:= a+b]^5$)

init(P) = 1

final(P) = {3}

blocks(P) =

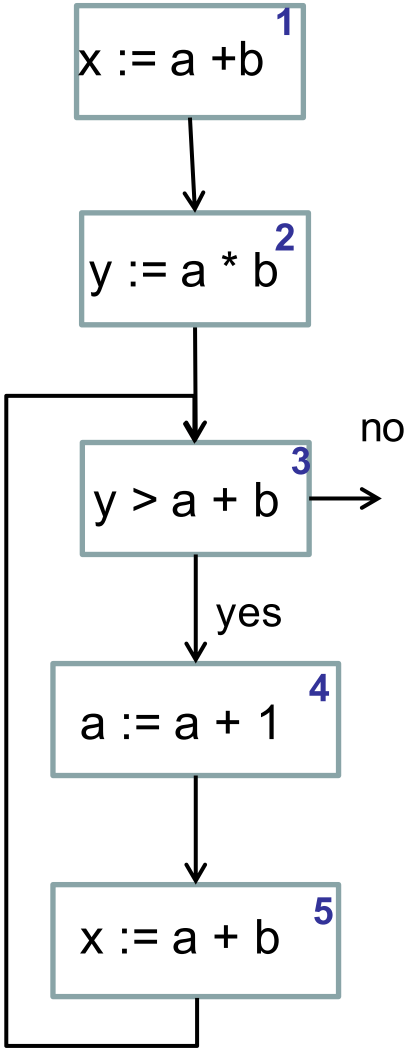
{ $[x := a+b]^1$, $[y := a*b]^2$, $[y > a+b]^3$, $[a:=a+1]^4$, [x:= a+b] }

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)

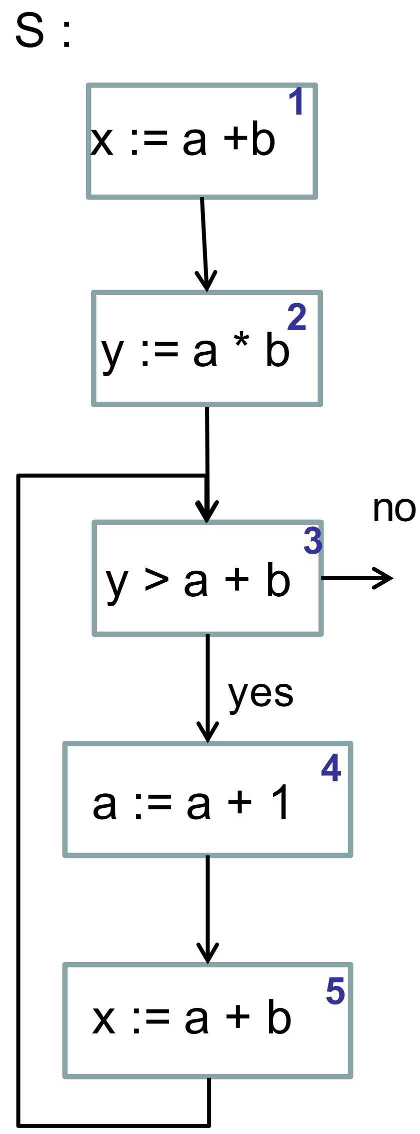
FV(a + b) = {a, b}



Available Expression Analysis

- The avaiable expression analysis will determine:

For each program point, which expressions must have already been computed, and not later modified, on all paths to this program point.



Available Expression Analysis

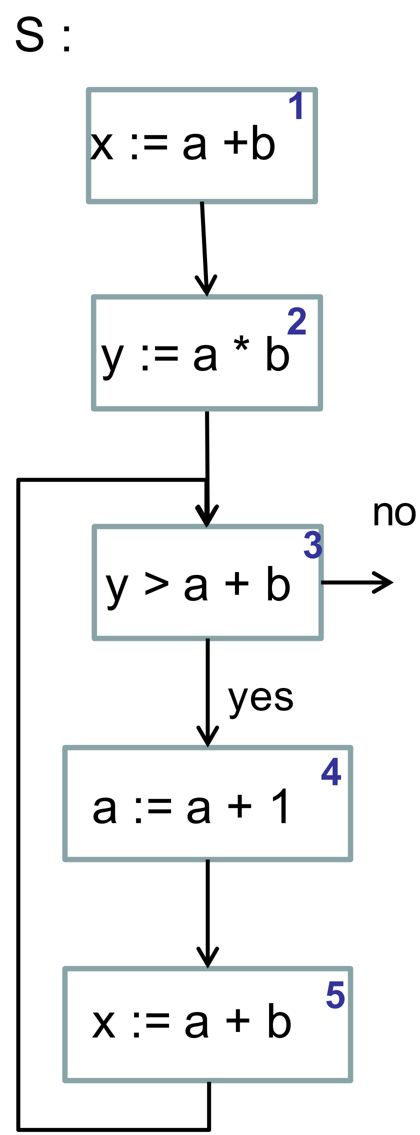
gen($[x := a]^l$) = { a‘ ∈ Aexp(a) | x∉FV(a‘) }
gen($[skip]^l$) = ∅
gen($[b]^l$) = Aexp(b)

kill($[x := a]^l$) = { a‘ ∈ Aexp(S) | x ∈ FV(a‘) }
kill($[skip]^l$) = ∅
kill($[b]^l$) = ∅

$AE_{in}(l) = \emptyset$, if $l \in init(S)$ and
$AE_{in}(l) = \bigcap \{AE_{out}(l') \mid (l', l) \in flow(S) \}$, otherwise
$AE_{out}(l) = (AE_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	kill(l)	gen(l)
1		
2		
3		
4		
5		

l	AE_{in}	AE_{out}
1		
2		
3		
4		
5		



Available Expression Analysis

$gen([x := a]^l) = \{ a' \in Aexp(a) \mid x \notin FV(a') \}$
$gen([skip]^l) = \emptyset$
$gen([b]^l) = Aexp(b)$

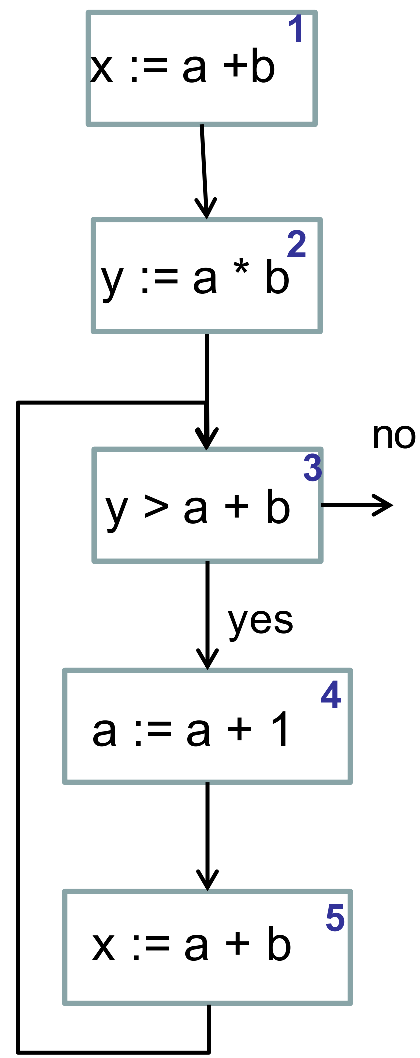
$kill([x := a]^l) = \{ a' \in Aexp(S) \mid x \in FV(a') \}$
$kill([skip]^l) = \emptyset$
$kill([b]^l) = \emptyset$

$AE_{in}(l) = \emptyset$, if $l \in init(S)$ and
$AE_{in}(l) = \bigcap \{ AE_{out}(l') \mid (l', l) \in flow(S) \}$, otherwise
$AE_{out}(l) = (AE_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	kill(l)	gen(l)
1	$\emptyset$	{a+b}
2	$\emptyset$	{a*b}
3	$\emptyset$	{a+b}
4	{a+b, a*b, a+1}	$\emptyset$
5	$\emptyset$	{a+b}

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

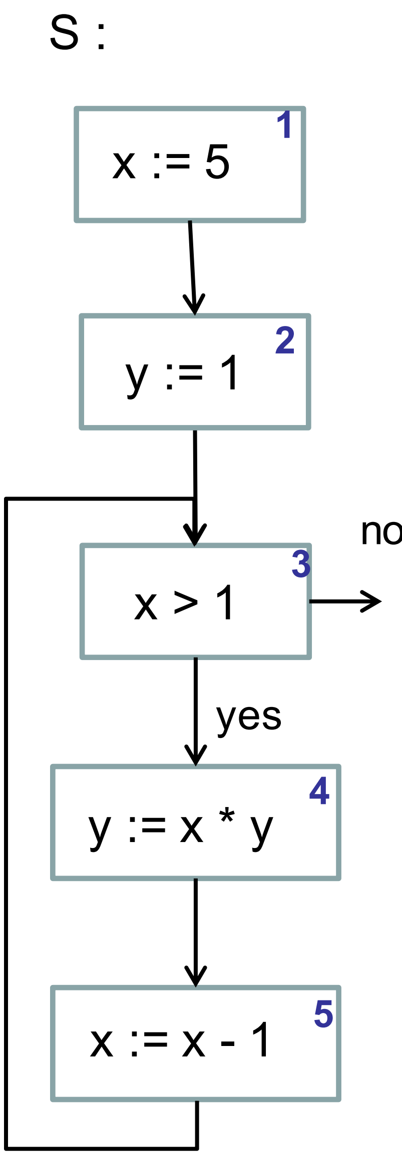
S :



Reaching Definitions Analysis

- Reaching definitions (assignment) analysis determines if:

> An assignment of the form $[x := a]^l$ may reach a certain program point k if there is an execution of the program where x was last assigned a value at l when the program point k is reached



Reaching Definitions Analysis

gen([x :=a]^l) = { (x, l) }
gen([skip]^l) = $\emptyset$
gen([b]^l) = $\emptyset$

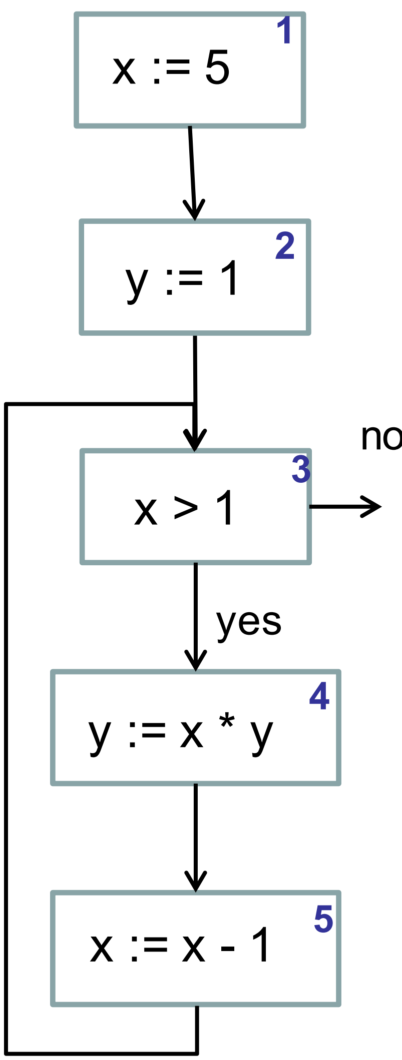
kill([skip]^l) = $\emptyset$
kill([b]^l) = $\emptyset$
kill([x :=a]^l) = { (x, ?) } $\cup$ { (x, k) | B^k is an assignment to x in S }

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

$RD_{in}(\ l\ )$ = { (x, ?) | x $\in$ FV(S)} , if $l \in$ init(S) and
$RD_{in}(\ l\ ) = \bigcup \{RD_{out}(\ l'\ ) \mid (l', l) \in flow(S)\}$, otherwise
$RD_{out}(\ l\ ) = (\ RD_{in}(\ l\ ) \setminus kill(B^l)\ ) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	RD_{in}	RD_{out}
1		
2		
3		
4		
5		

S :



Reaching Definitions Analysis

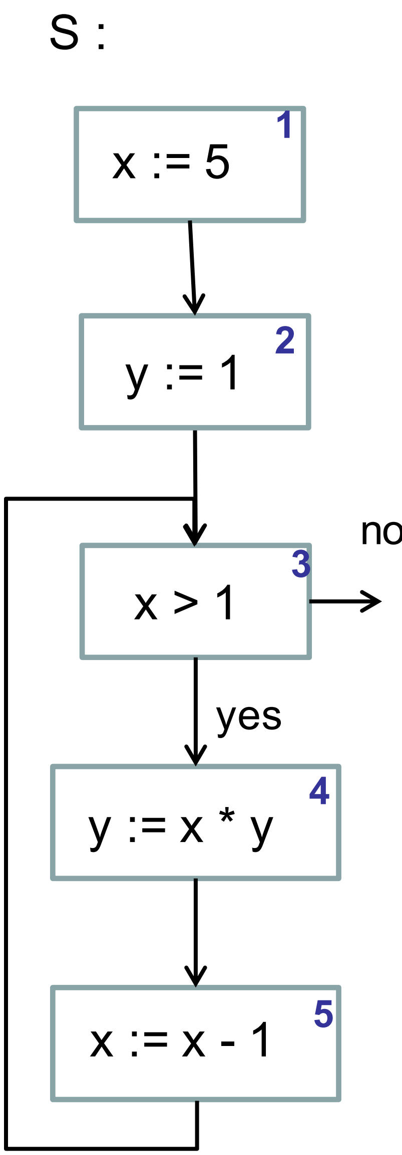
gen([x :=a]^l) = { (x, l) }
gen([skip]^l) = ∅
gen([b]^l) = ∅

kill([skip]^l) = ∅
kill([b]^l) = ∅
kill([x :=a]^l) = { (x, ?) } ∪ { (x, k) | B^k is an assignment to x in 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)}

$RD_{in}(l) = \{ (x, ?) \mid x \in FV(S) \}$, if $l \in init(S)$ and
$RD_{in}(l) = \bigcup \{ RD_{out}(l') \mid (l', l) \in flow(S) \}$, otherwise
$RD_{out}(l) = (RD_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

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)}

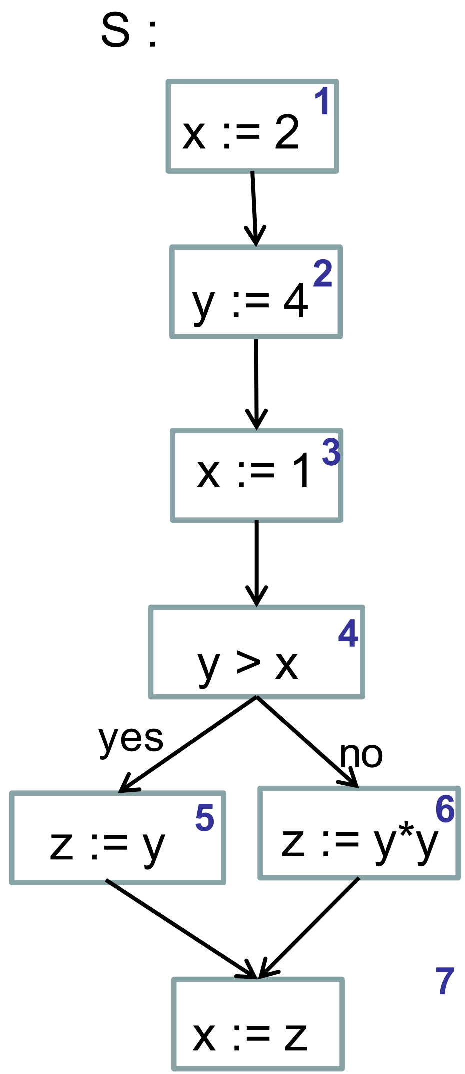


Live Variables Analysis

- 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 live at the exit from that point.

- Application: dead code elemination.



Live Variables Analysis

gen([x :=a]^l) = FV(a)
gen([skip]^l) = $\emptyset$
gen([b]^l) = FV(b)

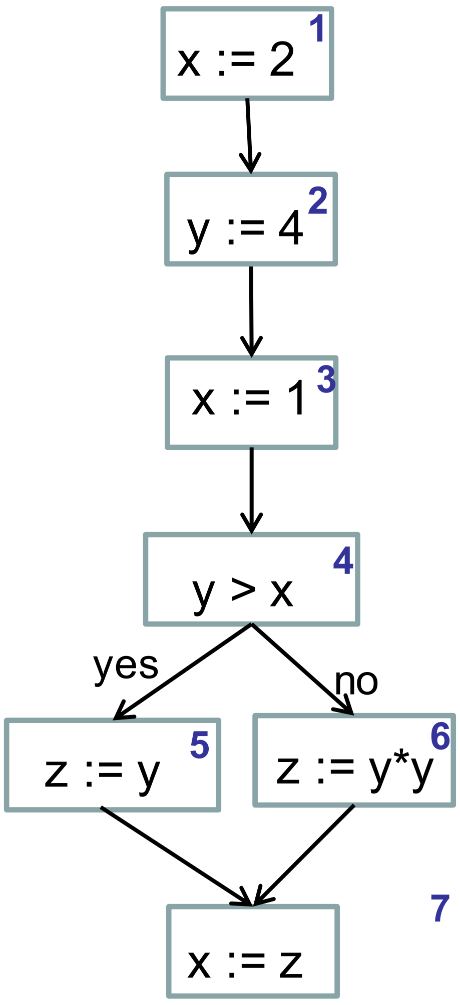
kill([x :=a]^l) = {x}
kill([skip]^l) = $\emptyset$
kill([b]^l) = $\emptyset$

$LV_{out}(l) = \emptyset$, if $l \in$ final(S) and
$LV_{out}(l) = \bigcup \{LV_{in}(l') \mid (l', l) \in flow^R(S) \}$, otherwise
$LV_{in}(l) = (LV_{out}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in$ blocks(S)

l	kill(*l*)	gen(*l*)
1		
2		
3		
4		
5		
6		
7		

l	LV_{in}	LV_{out}
1		
2		
3		
4		
5		
6		
7		

S :



Live Variables Analysis

gen([x :=a]^l) = FV(a)
gen([skip]^l) = $\emptyset$
gen([b]^l) = FV(b)

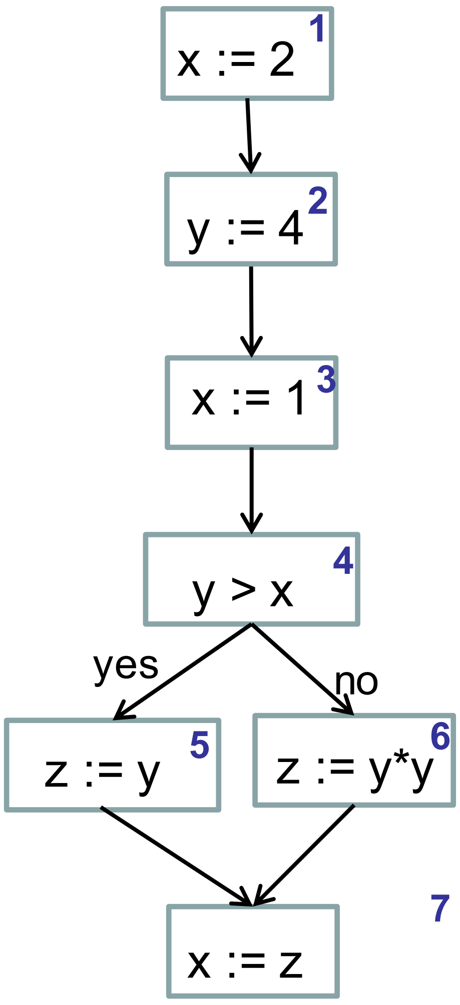
kill([x :=a]^l) = {x}
kill([skip]^l) = $\emptyset$
kill([b]^l) = $\emptyset$

$LV_{out}(l) = \emptyset$, if $l \in final(S)$ and
$LV_{out}(l) = \bigcup \{LV_{in} (l') \mid (l', l) \in flow^R(S) \}$, otherwise
$LV_{in} (l) = (LV_{out}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	kill(*l*)	gen(*l*)
1	{x}	$\emptyset$
2	{y}	$\emptyset$
3	{x}	$\emptyset$
4	$\emptyset$	{x, y}
5	{z}	{y}
6	{z}	{y}
7	{x}	{z}

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

S :



First Generalized Schema

- $\text{Analyse}_\circ(l) = \textbf{EV}$, if $l \in \textbf{E}$ and
- $\text{Analyse}_\circ(l) = \sqcup \{ \text{Analyse}_\bullet(l') \mid (l', l) \in \textbf{Flow}(S) \}$, otherwise
- $\text{Analyse}_\bullet(l) = \mathbf{f}_l(\text{Analyse}_\circ(l))$

With:

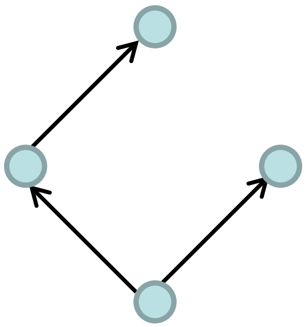
- $\sqcup$ is either $\bigcup$ or $\bigcap$
- EV is the initial / final analysis information
- Flow is either flow or flow^R
- E is either {init(S)} or final(S)
- f_l is the transfer function associated with $B^l \in \text{blocks}(S)$

Backward analysis: $F = \text{flow}^R$, $\bullet$ = IN, $\circ$ = OUT

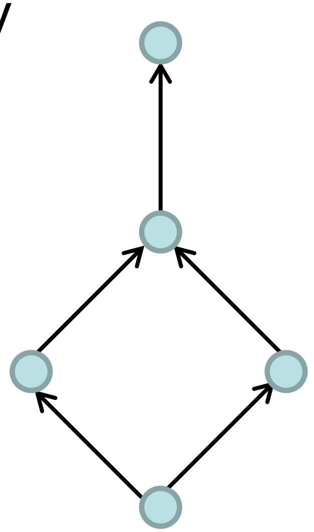
Forward analysis: F = flow, $\bullet$ = OUT, $\circ$ = IN

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. $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - Anti-symmetry: $\forall$ x,y $\in$ M. $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$



- 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 $\sqcup$X $\in$ M of X $\subseteq$ M :
    - $X \sqsubseteq \sqcup X \wedge \forall y \in M : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$
  - Greatest lower bound $\sqcap$X $\in$ M of X $\subseteq$ M:
    - $\sqcap X \sqsubseteq X \wedge \forall y \in M : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$



Lattice

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

- $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq M$
- Unique greatest element $\top = \sqcup M = \sqcap \emptyset$
- Unique least element $\bot = \sqcap M = \sqcup \emptyset$

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. Set F of transfer functions of the form
  $f_l : L \to L$ with l being a label

- Knowledge transfer is monotone
  - $\forall$ x,y. $x \sqsubseteq y \Rightarrow f_l(x) \sqsubseteq f_l(y)$

- Space F of transfer functions
  - F contains all transfer functions f_l
  - F contains the identity function id, i.e. $\forall x \in M.\ id(x) = x$
  - F is closed under composition, i.e. $\forall f,g \in F.\ (f \circ g) \in F$

The Generalized Analysis

- $Analyse_{\circ}(l) = \bigsqcup \{ Analyse_{\bullet}(l') \mid (l', l) \in Flow(S) \} \sqcup \iota^{l}_{E}$
  with $\iota^{l}_{E} = EV$ if $l \in E$ and
  $\iota^{l}_{E} = \bot$ otherwise
- $Analyse_{\bullet}(l) = f_{l}(Analyse_{\circ}(l))$

With:

- L property space representing data flow information with (L, $\sqcup$) being a lattice
- Flow is a finite flow (i.e. flow or $flow^{R}$)
- EV is an extremal value for the extremal labels E (i.e. {init(S)} or final(S))
- transfer functions f_{l} of a space of transfer functions *F*

Summary

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