Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 09 (07-12-2015)

Static Program Analysis

Christoph Lüth Jan Peleska Dieter Hutter

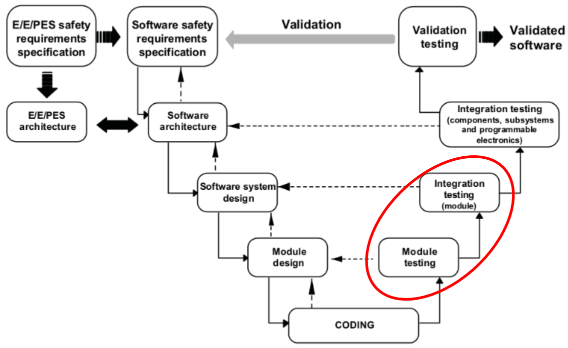
Universität Bremen

Where are we?

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.

Program Analysis in the Development Cycle



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

Program Analysis: The Basic Problem

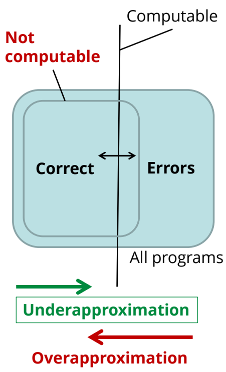
- Basic Problem:

  All interesting program properties are undecidable.

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

Our Simple Programming Language

- In the last lecture, we introduced a very simple language with a C-like syntax.
- Synposis:

Arithmetic operators given by

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

Boolean operators given by

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

$$op_b \in \{and, or\}, op_r \in \{=,<,\leq,>,\geq,\neq\}$$

Statements given by

S ::=

$$[x := a]^l \mid [skip]^l \mid S_1; S_2 \mid if\ [b]^l\ \{S_1\}\ else\ \{S_2\} \mid while\ [b]^l\ \{S\}$$

Computing the Control Flow Graph

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

Labels, Blocks, Flows: Definitions

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

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

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

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

$labels(S) = \{l \mid [B]^l \in blocks(S)\}$

$FV(a)$ = free variables in a

$Aexp(S)$ = non-trival 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)}

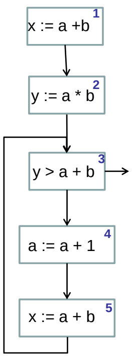
flow^R(P) = {(2, 1), (3, 2), (4, 3), (5, 4), (3, 5)}

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

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

FV(P) = {a, b, x, y}

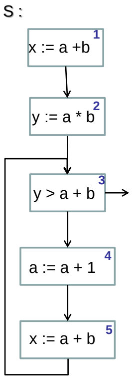
Aexp(P) = {a+b, a*b, a+1}



Available Expression Analysis

- The available expression analysis will determine:

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



Available Expression Analysis

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

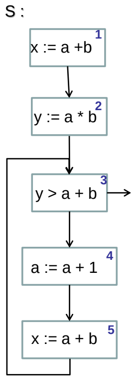
kill([x :=a]^l) = $\{a' \in Aexp(S) \mid x \in FV('a)\}$
kill([skip]^l) = ∅
kill([b]^l) = ∅

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

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

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

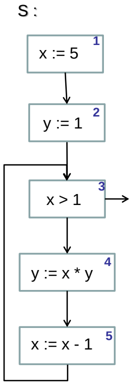
l	AE_{in}	AE_{out}
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}



Reaching Definitions Analysis

- Reaching definitions (assignment) analysis determines if:

An assignment of the form [x := a]^l 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) = ∅
gen([b]^l) = ∅

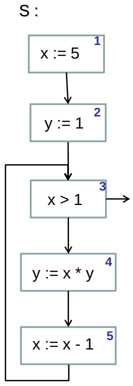
l	kill(B^l)	gen(B^l)
1	{(x,?), (x,1),(x,5)}	{(x, 1)}
2	{(y,?), (y,2),(y,4)}	{(y, 2)}
3	∅	∅
4	{(y,?), (y,2),(y,4)}	{(y, 4)}
5	{(x,?), (x,1),(x,5)}	{(x, 5)}

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

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

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

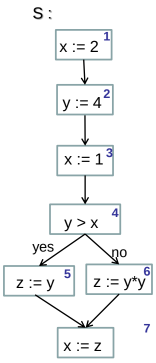


Live Variables Analysis

- 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) = ∅
gen([b]^l) = $FV(b)$

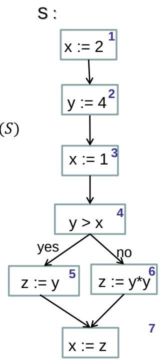
kill([x :=a]^l) = $\{x\}$
kill([skip]^l) = ∅
kill([b]^l) = ∅

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

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

l	kill(*l*)	gen(*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

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

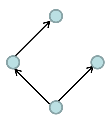
With:

- □ is either ⋃ or ⋂
- **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 $B^l \in blocks(S)$

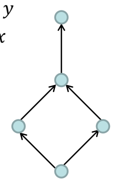
Backward analysis: **Flow** = flow^R, • = IN, ∘ = OUT
Forward analysis: **Flow** = flow, • = OUT, ∘ = IN

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



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. Let F be the set of transfer functions of the form
  $f_l : L \rightarrow L$ with l being a label
- Knowledge transfer is monotone
  - $\forall x, y. x \sqsubseteq y \Longrightarrow f_l(x) \sqsubseteq f_l(y)$
- Space F of transfer functions
  - F contains all transfer functions f_l
  - F contains the identity function id: $\forall x \in M. id(x) = x$
  - F is closed under composition: $\forall f, g \in F. (g \circ f) \in F$

The Generalized Analysis

- $\text{Analysis}_\circ(l) = \sqcup \{\text{Analysis}_\bullet(l') \mid (l', l) \in Flow(S)\} \sqcup \{\iota'_E\}$
  with $\iota'_E = \begin{cases} EV & \text{if } l \in E \\ \bot & \text{otherwise} \end{cases}$
- $\text{Analysis}_\bullet(l) = f_l(\text{Analysis}_\circ(l))$

With:

- L property space representing data flow information with $(L, \sqsubseteq)$ 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)