Data-Flow Analysis

Data-flow analysis (DFA) provides information about
how the **execution of a program may manipulate its data**.

Many different problems can be formulated as **data-flow problems**, for example:

- Which assignments to variable **v** may influence a use of **v** at a certain program position?
- Is a variable **v** used on any path from a program position **p** to the exit node?
- The values of which expressions are available at program position **p**?

Data-flow problems are stated in terms of

- **paths through the control-flow graph** and
- **properties of basic blocks**.

Data-flow analysis provides information for **global optimization.**

Data-flow analysis does not know

- which input values are provided at run-time,
- which branches are taken at run-time.

Its results are to be interpreted **pessimistic**

Lecture Compilation Methods SS 2013 / Slide 218

Objectives:

Goals and ability of data-flow analysis

In the lecture:

- Examples for the use of DFA information are given.
- Examples for pessimistic information are given.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

Questions:

- What’s wrong about optimistic information?
- Why can pessimistic information be useful?

Data-Flow Equations

A data-flow problem is stated as a **system of equations** for a control-flow graph.

System of Equations for **forward problems** (propagate information along control-flow edges):

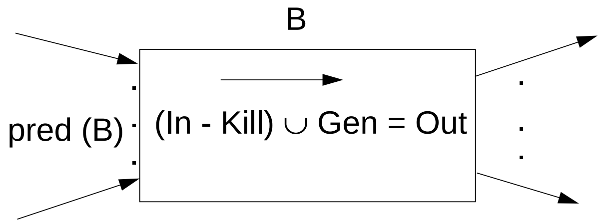
Example **Reaching definitions:**
A definiton **d** of a variable **v** reaches the begin of a block **B** if **there is a path** from **d** to **B** on which **v** is not assigned again.

In, Out, Gen, Kill represent analysis information:
sets of statements,
sets of variables,
sets of expressions
depending on the analysis problem

2 equations for each basic block:

$$\text{Out}(B) = f_B(\text{In}(B)) = \text{Gen}(B) \cup (\text{In}(B) - \text{Kill}(B))$$

$$\text{In}(B) = \underset{h \in \text{pred}(B)}{\Theta} \text{Out}(h)$$



In, Out **variables** of the system of equations for each block

Gen, Kill a pair of **constant sets** that characterize a block w.r.t. the DFA problem

Θ meet operator; e. g. $\Theta = \cup$ for „reaching definitions“, $\Theta = \cap$ for „available expressions“

Lecture Compilation Methods SS 2013 / Slide 219

Objectives:

A DFA problem is modeled by a system of equations

In the lecture:

- The equation pattern is explained.
- Equations are defined over sets.
- In this example: sets of assignment statements at certain program positions.
- The meet operator being the union operator is correlated to "there is a path" in the problem statement.
- Note: In this context a "definition of a variable" means an "assignment of a variable".

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

Questions:

- Explain the meaning of In(B)= {d1: x=5, d4: x=7, d6: y=a+1} for a particular block B.

Specification of a DFA Problem

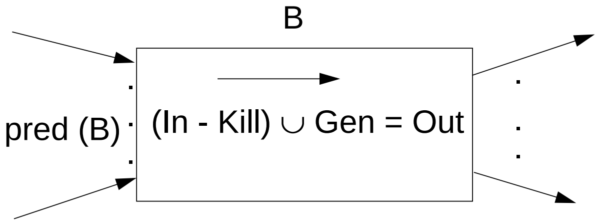
Specification of reaching definitions:

1. **Description**:
   A definiton **d** of a variable **v** reaches the begin of a block **B**
   if **there is a path** from **d** to **B** on which **v** is not assigned again.
2. It is a **forward problem**.
3. The **meet operator** is union.
4. The **analysis information** in the sets are assignments at certain program positions.
5. **Gen (B)**:
   contains all definitions **`d: v = e;`** in **B**,
   such that **v** is not defined after **d** in **B**.
6. **Kill (B)**:
   if **v** is assigned in **B**, then **Kill(B)**
   contains all definitions **`d: v = e;`**
   of blocks different from **B**.

2 equations for each basic block:

$$\mathrm{Out}\ (B) = f_B\ (\mathrm{In}\ (B)) = \mathrm{Gen}\ (B) \cup (\mathrm{In}\ (B) - \mathrm{Kill}\ (B))$$

$$\mathrm{In}\ (B) = \underset{h \in \mathrm{pred}(B)}{\Theta} \mathrm{Out}\ (h)$$



Lecture Compilation Methods SS 2013 / Slide 220

Objectives:

Specify a DFA problem systematically

In the lecture:

- The items that characterize a DFA problem are explained.
- The definition of Gen and Kill is explained.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

Questions:

- Why does this definition of Gen and Kill serves the purpose of the description in the first item?

Variants of DFA Problems

- **forward** problem:
  DFA information flows **along the control flow**
  In(B) is determined by Out(h) of the predecessor blocks

  backward problem (see C-2.23):
  DFA information flows **against the control flow**
  Out(B) is determined by In(h) of the successor blocks

- **union** problem:
  problem description: „there is a path“;
  meet operator is Θ **=** $\cup$
  solution: minimal sets that solve the equations

  intersect problem:
  problem description: „for all paths“
  meet operator is Θ **=** $\cap$
  solution: maximal sets that solve the equations

- **optimization information**: **sets of** certain statements, of variables, of expressions.

Further classes of DFA problems over general lattices instead of sets are not considered here.

Lecture Compilation Methods SS 2013 / Slide 221

Objectives:

Summary of the DFA variants

In the lecture:

- The variants of DFA problems are compared.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

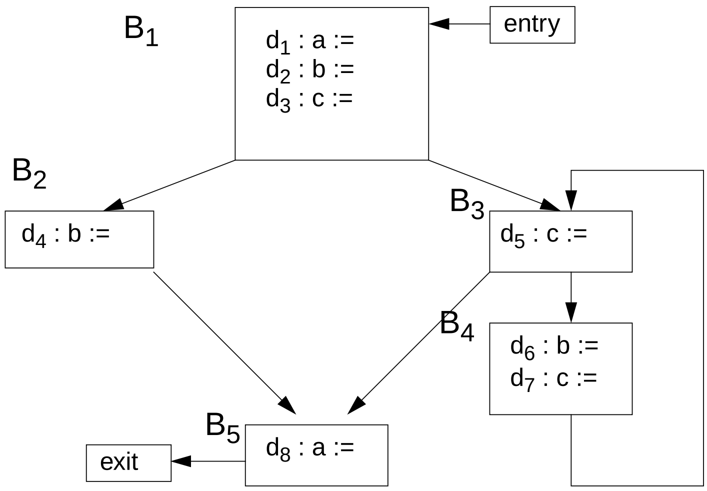
Questions:

- Explain the relation of the meet operator, the paths in the graph, and the maximal / minimal solutions.

Example Reaching Definitions

Gen (B):
contains all definitions **d: v = e;**
in **B**, such that **v** is not defined
after **d** in **B**.

Kill (B):
contains all definitions **d: v = e;**
in blocks different from **B**,
such that B has a definition of **v**.



Description of DFA-Problem

	Gen	Kill
B_1	d_1, d_2, d_3	d_4, d_5, d_6, d_7, d_8
B_2	d_4	d_2, d_6
B_3	d_5	d_3, d_7
B_4	d_6, d_7	d_2, d_3, d_4, d_5
B_5	d_8	d_1

DFA-Solution

In	Out
$\emptyset$	d_1, d_2, d_3
d_1, d_2, d_3	d_1, d_3, d_4
d_1, d_2, d_3, d_6, d_7	d_1, d_2, d_5, d_6
d_1, d_2, d_5, d_6	d_1, d_6, d_7
$d_1, d_2, d_3, d_4, d_5, d_6$	$d_2, d_3, d_4, d_5, d_6, d_8$

Lecture Compilation Methods SS 2013 / Slide 222

Objectives:

Understand the meaning of DFA sets

In the lecture:

- The example for C-2.20 is explained.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

Questions:

- Check that the In and Out sets solve the equations for the CFG.
- How can you argue that the solution is minimal?
- Add some elements to the solution such that it still solves the equations. Explain what such non-minimal solutions mean.

Iterative Solution of Data-Flow Equations

Input: the CFG; the sets Gen(B) and Kill(B) for each basic block B
Output: the sets In(B) and Out(B)

Algorithm:

```
repeat
   stable := true;
   for all B ≠ entry   {*}
   do begin
      for all V ∈ pred(B) do
         In(B):= In(B) Θ Out(V);
      oldout:= Out(B);
      Out(B):= Gen(B) ∪ (In(B)-Kill(B));
      stable:= stable and Out(B)=oldout
   end
until stable
```

Initialization

Union: empty sets

```
for all B do
begin
   In(B):=∅;
   Out(B):=Gen(B)
end;
```

Intersect: full sets

```
for all B do
begin
   In(B) := U;
   Out(B):=
      Gen(B)∪
          (U - Kill(B))
end;
```

Complexity: $O(n^3)$ with n number of basic blocks
$O(n^2)$ if $|$**pred (B)**$|$ ≤ **k << n** for all B

Lecture Compilation Methods SS 2013 / Slide 222b

Objectives:

Understand the iterative DFA algorithm

In the lecture:

The topics on the slide are explained. Examples are given.

- Initialization variants are explained.
- The algorithm is explained.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.5, 8.2.6

Questions:

- How is the initialization related to the size of the solution for the two variants union and intersect?
- Why does the algorithm terminate?

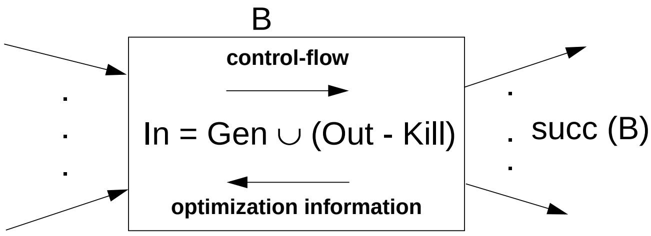
Backward Problems

System of Equations for **backward problems**
propagate information against control-flow edges:

2 equations for each basic block:

$$\mathrm{In}(B) = f_B(\mathrm{Out}(B)) = \mathrm{Gen}(B) \cup (\mathrm{Out}(B) - \mathrm{Kill}(B))$$

$$\mathrm{Out}(B) = \underset{h \in \mathrm{succ}(B)}{\Theta} \mathrm{In}(h)$$



Example **Live variables:**

1. Description: Is variable **v** alive at a given point **p** in the program, i. e. **is there a path** from **p** to the exit where **v** is used but not defined before the use?
2. backward problem
3. optimization information: sets of variables
4. meet operator: Θ **=** $\cup$ union
5. Gen (B): variables that are used in B, but not defined before they are used there.
6. Kill (B): variables that are defined in B, but not used before they are defined there.

Lecture Compilation Methods SS 2013 / Slide 223

Objectives:

Symmetry of forward and backward schemes

In the lecture:

The topics on the slide are explained. Examples are given.

- The equation pattern is explained.
- The DFA problem "live variables" is explained.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

Questions:

- How do you determine the live variables **within** a basic block?

Important Data-Flow Problems

1. **Reaching definitions:** A definiton **d** of a variable **v** reaches the beginning of a block **B** if there is a path from **d** to **B** on which **v** is not assigned again.
   DFA variant: forward; union; set of assignments
   Transformations: use-def-chains, constant propagation, loop invariant computations
2. **Live variables:** Is variable **v** alive at a given point **p** in the program, i. e. there is a path from **p** to the exit where **v** is used but not defined before the use.
   DFA variant: backward; union; set of variables
   Transformations: eliminate redundant assignments
3. **Available expressions:** Is expression **e** computed on every path from the entry to a program position **p** and none of its variables is defined after the last computation before **p**.
   DFA variant: forward; intersect; set of expressions
   Transformations: eliminate redundant computations
4. **Copy propagation:** Is a copy assignment **c: x = y** redundant, i.e. on every path from **c** to a use of **x** there is no assignment to **y**?
   DFA variant: forward; intersect; set of copy assignments
   Transformations: remove copy assignments and rename use
5. **Constant propagation:** Has variable **x** at position **p** a known value, i.e. on every path from the entry to **p** the last definition of **x** is an assignment of the same known value.
   DFA variant: forward; combine function; vector of values
   Transformations: substitution of variable uses by constants

Lecture Compilation Methods SS 2013 / Slide 224

Objectives:

Recognize the DFA problem scheme

In the lecture:

- The DFA problems and their purpose are explained.
- The DFA classification is derived from the description.
- Examples are given.
- Problems like copy propagation oftem match to code that results from other optimizing transformations.

Suggested reading:

Kastens / Übersetzerbau, Section 8.3

Questions:

- Explain the classification of the DFA problems.
- Construct an example for each of the DFA problems.

Algebraic Foundation of DFA

DFA performs computations on a **lattice (dt. Verband)** of values,
e. g. bit-vectors representing finite sets. It guarantees termination of computation and well-defined solutions. see [Muchnick, pp 223-228]

A **lattice L** is a set of values with two operations: $\cap$ **meet** and $\cup$ **join**

Required properties:

1. **closure**: $x, y \in L$ implies $x \cap y \in L$, $x \cup y \in L$
2. **commutativity**: $x \cap y = y \cap x$ and $x \cup y = y \cup x$
3. **associativity**: $(x \cap y) \cap z = x \cap (y \cap z)$ and $(x \cup y) \cup z = x \cup (y \cup z)$
4. **absorption**: $x \cap (x \cup y) = x = x \cup (x \cap y)$
5. unique elements **bottom** $\perp$, **top** T:
   $x \cap \perp = \perp$ and $x \cup T = T$

In most DFA problems only a **semilattice** is used with L, $\cap$, $\perp$ or L, $\cup$, T

Partial order defined by meet, $x \subseteq y$: $x \cap y = x$ defined by join: $x \supseteq y$: $x \cup y = x$
(transitive, antisymmetric, reflexive)

Lecture Compilation Methods SS 2013 / Slide 224a

Objectives:

Recall algebraic structure lattice

In the lecture:

The topics on the slide are explained using examples of C-2.24b

Some DFA Lattices

2

Bool

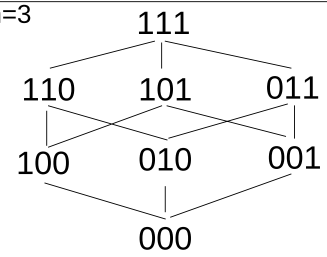
$\cap$ = and
$\cup$ = or

T = true
|
$\perp$ = false

1

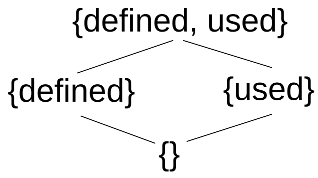
Bit-Vector $BV^{n=3}$

$\cap$ = bitwise and
$\cup$ = bitwise or



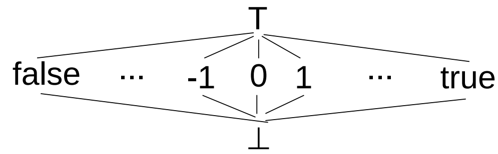
3

Variable usage



4

ICP Integer Constant Propagation Lattice



$n \cap \perp = \perp$ $n \cap n = n$ $n \cap m = \perp$ if $n \neq m$
$n \cup T = T$ $n \cup n = n$ $n \cup m = T$ if $n \neq m$

5

Range Lattice: $[lo, hi] \in (Z \cup \{-\infty, \infty\})^2$

$\perp$ = [ ] empty range, T = $[-\infty, \infty]$,
$x \subseteq y$: x is contained in y

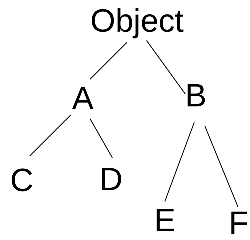
$\cap$: [l1, h1] $\cap$ [l2, h2] = x
let l = max (l1, l2),
h = min (h1, h2),
x = if h < l then $\perp$ else [l, h]

$\cup$: [l1, h1] $\cup$ [l2, h2] =
[min(l1, l2), max(h1, h2)]

6

Semilattice of types

$\cup$: $x \cup y$ = smallest common supertype of x and y



Lecture Compilation Methods SS 2013 / Slide 224b

Objectives:

Most important DFA lattices

In the lecture:

- The Examples are explained.
- A new lattice can be constructed by elementwise composition of simpler lattices; e.g. a bit-vector lattice is an n-fold composition of the lattice Bool.
- A new lattice may be constructed for a particular DFA problem.

Monotone Functions Over Lattices

The **effects of program constructs on DFA information** are described by functions over a suitable lattice,

e. g. the function for basic block B_3 on C-2.22:

$f_3(<x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8>) = <x_1\ x_2\ 0\ x_4\ 1\ x_6\ 0\ x_8> \in BV^8$

Gen-Kill pair encoded as function

f: L $\rightarrow$ L is a **monotone function** over the lattice L if
$\forall$ x, y $\in$ L: x $\subseteq$ y $\Rightarrow$ f(x) $\subseteq$ f(y)

Finite height of the lattice and **monotonicity** of the functions guarantee **termination** of the algorithms.

Fixed points z of the function f, with f(z) = z, is a solution of the set of DFA equations.

MOP: Meet over all paths solution is desired, i. e. the „best“ with respect to L

MFP: Maximum fixed point is computed by algorithms, if functions are monotone

If the functions f are additionally **distributive**, then **MFP = MOP**.
f: L $\rightarrow$ L is a **distributive function** over the lattice L if
$\forall$ x, y $\in$ L: f(x $\cap$ y) = f(x) $\cap$ f(y)

Lecture Compilation Methods SS 2013 / Slide 224c

Objectives:

DFA equations and monotone functions

In the lecture:

Understand solution of DFA equations as fixed point of monotone functions.

Variants of DFA Algorithms

Heuristic improvement:

Goal: propagate changes in the In and Out sets as fast as possible.
Technique: visit CFG nodes in topological order in the outer for-loop {*}.
Then the number of iterations of the outer repeat-loop is only determined by back edges in the CFG

Algorithm for backward problems:

Exchange In and Out sets symmetrically in the algorithm of C-2.22b.
The nodes should be visited in topological order as if the directions of edges were flipped.

Hierarchical algorithms, interval analysis:

Regions of the CFG are considered nodes of a CFG on a higher level.
That abstraction is recursively applied until a single root node is reached.
The Gen, Kill sets are combined in upward direction;
the In, Out sets are refined downward.

Lecture Compilation Methods SS 2013 / Slide 226

Objectives:

Overview on DFA algorithms

In the lecture:

- The variants of the algorithm of C-2.25 are explained.
- The improvement is discussed.
- The idea of hierarchical approaches is explained.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.5, 8.2.6

Questions:

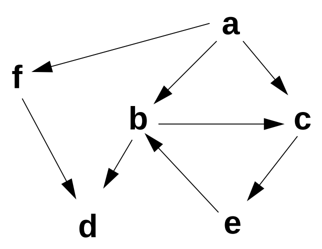
- For a backward problem the blocks could be considered in reversed topological order. Why is that not a good idea?

Program Analysis: Call Graph (context-insensitive)

Nodes: defined functions

Arc g -> h: function g contains a call h(),
i. e. a call g() **may** cause the execution of a call h()

```
void a ()  {...b()...c()...f()...}
void b ()  {...d()...c()...}
void c()   {...e()...}
void d()   {...}
void e()   {...v++;...b()...}
void f()   {...d()...}
```



Analysis of structure:
b, c, e are recursive;
a, d, f are non-recursive

Propagation of properties:
assume a call e() may **modify a global variable** v
then calls a(), b(), c() may indirectly cause modification of v

```
v = f(); cnt = 0; while(...){...b(); cnt += v;}
```


Lecture Compilation Methods SS 2013 / Slide 227

Objectives:

Understand call graphs

In the lecture:

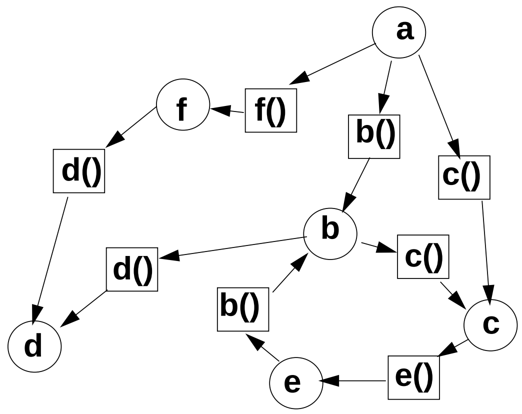
- Structural abstraction of call relation,
- Structural properties, e. g. reachability,
- Simplified implementation of non-recursive functions, of functions without calls, of functions that are never called.
- Propagation of information along call paths.
- Description of function behaviour, e. g. no side-effect on global variables.

Program Analysis: Call Graph (context-sensitive)

Nodes: defined functions and calls (bipartite)

Arc g -> h: function g contains a call h(),i.e a call g() **may** cause the execution of a call h()
or call g() leads to function g

```
void a ()  {...b()...c()...f()...}
void b ()  {...d()...c()...}
void c()   {...e()...}
void d()   {...}
void e()   {...v++;...b()...}
void f()   {...d()...}
```



Calls of the same function in different contexts are distinguished by **different nodes**, e.g. the call of c in a and in b.

Analysis can be **more precise** in that aspect.

Lecture Compilation Methods SS 2013 / Slide 227a

Objectives:

Understand context-sensitive call graphs

In the lecture:

Distinguish context-insensitive and context-sensitive call graphs

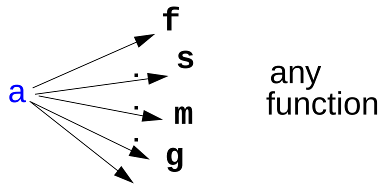
Calls Using Function Variables

Contents of **function variables is assigned at run-time**.

Static analysis does not know (precisely) which function is called.

Call graph has to assume that **any function may be called**.

```
void a()
  {...(*h)(0.3, 27)...}
```



Analysis for a better approximation of potential callees:

only those functions which

1. **fit to the type** of h
2. **are assigned** somewhere in the program
3. can be derived from the **reaching definitions** at the call

```
void m (int j) {...}

void g (float x, int i) {...}

...k = m;... f(g); ...

void a()
  {  void (*h)(float,int) = g;
     ...
     if(...) h = s;

     ...(*h)(0.3, 27)...
  }
```


Lecture Compilation Methods SS 2013 / Slide 228

Objectives:

Approximate call targets

In the lecture:

- Explain the approximation techniques using the example.
- Relate the problem to dynamically bound method calls.

Analysis of Object-Oriented Programs

Aspects specific for object-oriented analysis:

1. **hierarchy of classes and interfaces**
   specifies a complex **system of subtypes**
2. **hierarchy of classes and interfaces**
   specifies **inheritance and overriding** relation for methods
3. **dynamic method binding**
   for method calls **`v.m(...)`** the **callee is determined at run-time**
   good object-oriented style relies on that feature
4. **many small methods** are typical object-oriented style
5. **library use and reuse of modules**
   complete program contains many **unused classes and methods**

Static predictions for dynamically bound method calls
are essential for most analyses

Lecture Compilation Methods SS 2013 / Slide 229

Objectives:

Overview on oo analysis issues

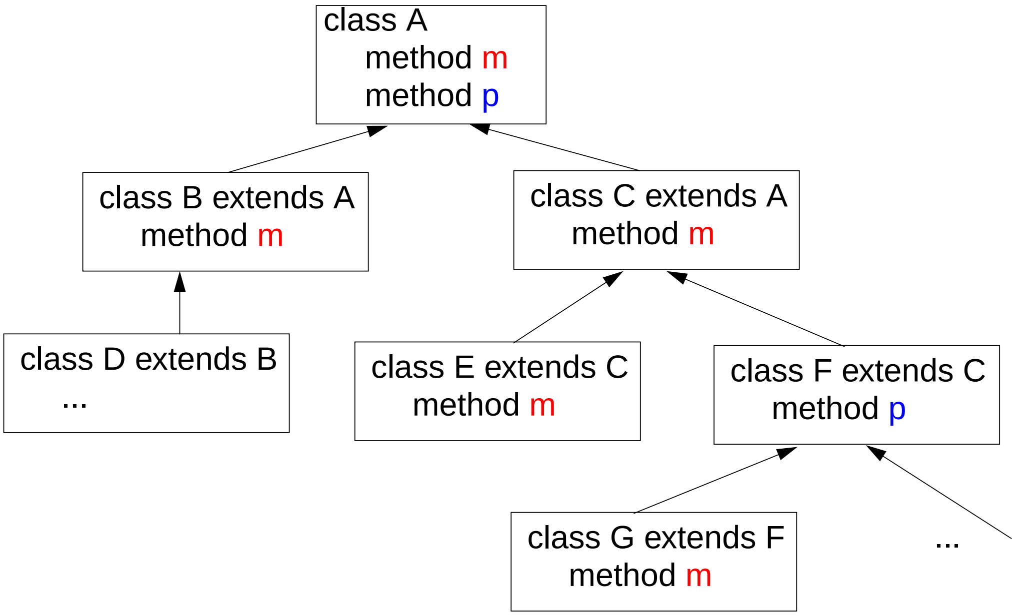
In the lecture:

- Role of class hierarchy for program analysis.
- Role of dynamic method binding for program analysis.

Class Hierarchy Graph

Node: **class or interface**

Arc a -> b: a is **subclass of** b or a **implements interface** b



Lecture Compilation Methods SS 2013 / Slide 230

Objectives:

Example for further consideration

In the lecture:

Recall central OO language properties:

- class hierarchy and typing,
- typed variables and method calls v.m(),
- inheritance of methods,
- overriding of methods,
- dynamically bound calls

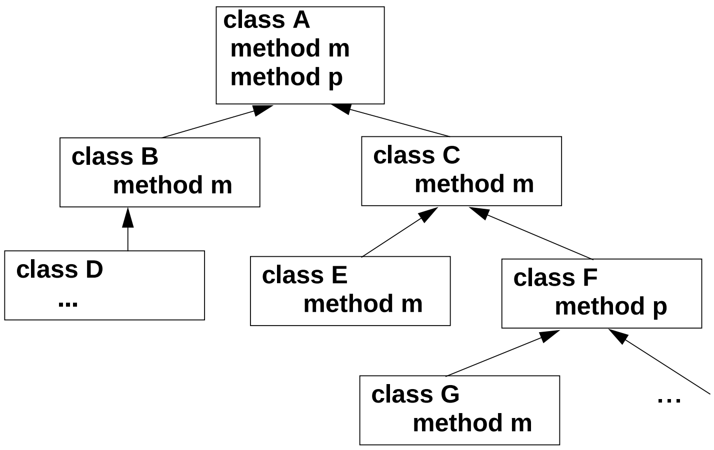
Assignments:

Recall the above mentioned language properties for Java and C++.

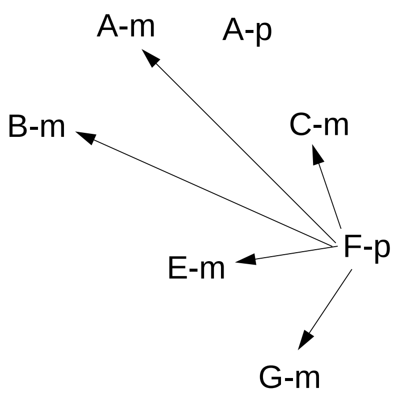
Object-Oriented Call Graph

Node: **implemented method**,
identified by class name, method name: X-a

Arc X-a -> Y-b: method **X-a contains a call** v.b(...) that **may be bound to Y-b**



Call graph for F-p containing v.m(...)



Call graph: **any method m** may be bound to that call in F-p
(compare to function variables)
analysis yields better approximations

Lecture Compilation Methods SS 2013 / Slide 231

Objectives:

Understand the call graph problem

In the lecture:

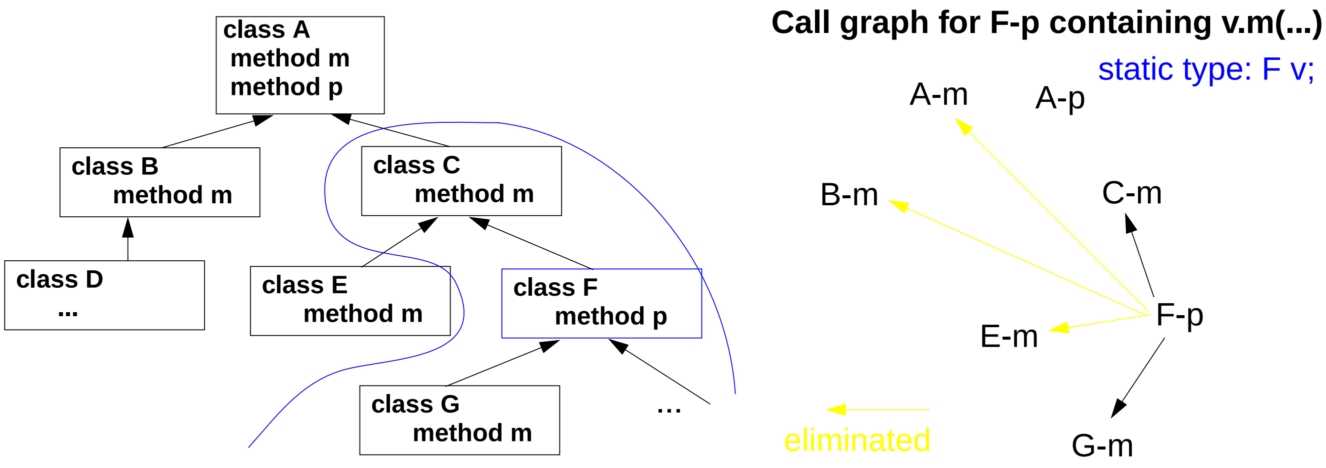
The topics on the slide are explained. using the example.

Call Graphs Constructed by Class Hierarchy Analysis (CHA)

The call graph is reduced to a set of **reachable methods** using the **class hierarchy** and the **static type of the receiver** expression in the call:

If a method F-p **is reachable** and
if it contains a **dynamically bound call v.m(...)** and
T is the static type of v,

then every method **m that is inherited by T or by a subtype of T is also reachable**, and arcs go from F-p to them.



Lecture Compilation Methods SS 2013 / Slide 232

Objectives:

In the lecture:

The CHA method is explained using the example.

Refined Approximations for Call Graph Construction

Class Hierarchy Analysis (CHA): (see C-2.32)

Rapid Type Analysis (RTA):

As CHA, but only methods of those classes C are considered which are instantiated (`new C()`) in a reachable method.

Reaching Type Analysis:

Approximations of run-time types is propagated through a graph: nodes represent variables, arcs represent copy assignments.

Declared Type Analysis:
one node T represents all variables declared to have type T

Variable Type Analysis:
one node V represents a single variable

Points-to Analysis:

Information on object identities is propagated through the control-flow graph

Lecture Compilation Methods SS 2013 / Slide 233

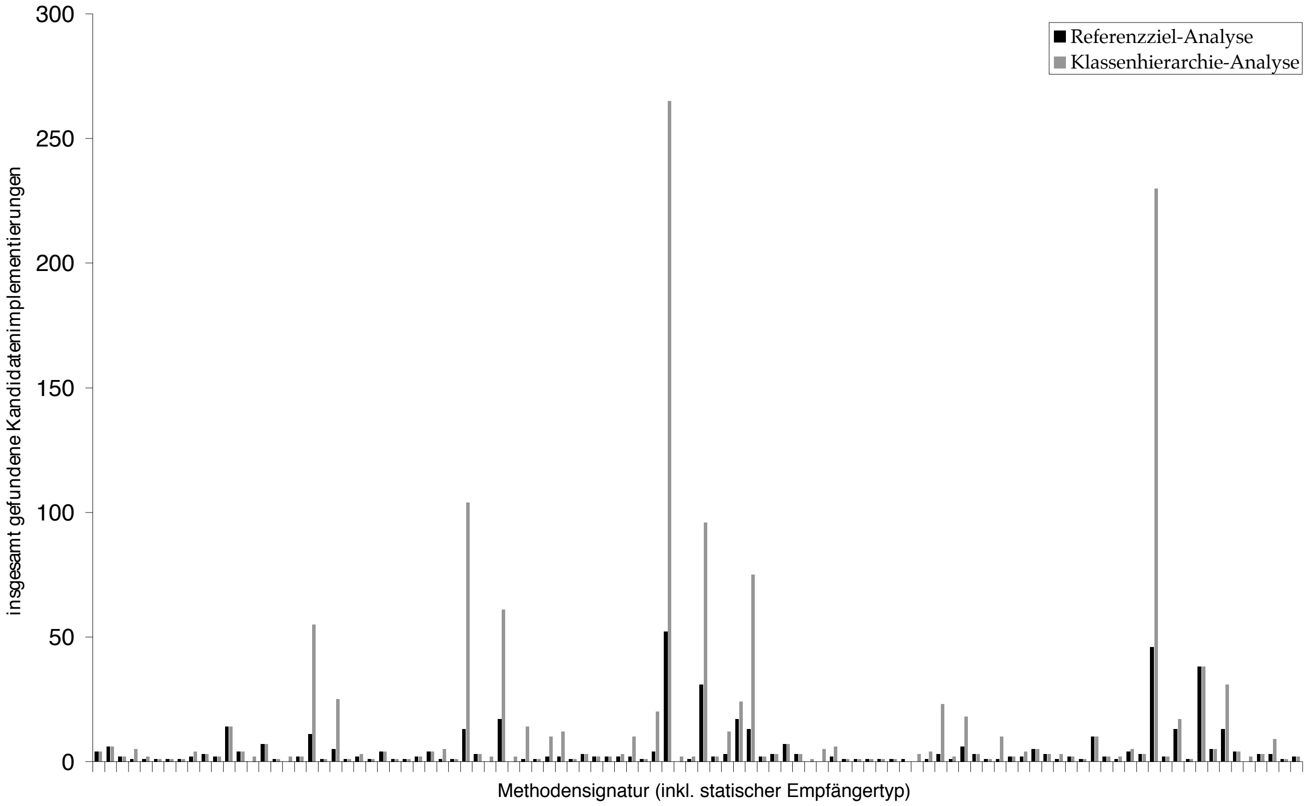
Objectives:

Powerful OO type analyses

In the lecture:

The methods are explained using small examples.

Results of Analysis of Dynamically Bound Calls



Lecture Compilation Methods SS 2013 / Slide 234

Objectives:

Effects on call identification

In the lecture:

The topics on the slide are explained. Examples are given.

- A pair of bars characterizes the number of method implementations, that may be bound to a set of calls having a particular type characteristics.
- Compare the results for CHA and points-to analysis.

Modules of a Toolset for Program Analysis

analysis module	purpose	category
ClassMemberVisibility	examines visibility levels of declarations	visualization
MethodSizeStatistics	examines length of method implementations in bytecode operations and frequency of different bytecode operations	
ExternalEntities	histogram of references to program entities that reside outside a group of classes	
InheritanceBoundary	histogram of lowest superclass outside a group of classes	
SimpleSetterGetter	recognizes simple access methods with bytecode patterns	
MethodInspector	decomposes the raw bytecode array of a method implementation into a list of instruction objects	auxiliary analysis
ControlFlow	builds a control flow graph for method implementations	fundamental analyses
Dominator	constructs the dominator tree for a control flow graph	
Loop	uses the dominator tree to augment the control flow graph with loop and loop nesting information	
InstrDefUse	models operand accesses for each bytecode instruction	
LocalDefUse	builds intraprocedural def / use chains	
LifeSpan	analyzes lifeness of local variables and stack locations	
DefUseTypeInfo	infers type information for operand accesses	analysis of incomplete programs
Hierarchy	class hierarchy analysis based on a horizontal slice of the hierarchy	
PreciseCallGraph	builds call graph based on inferred type information, copes with incomplete class hierarchy	
ParamEscape	transitively traces propagation of actual parameters in a method call (escape = leaves analyzed library)	
ReadWriteFields	transitive liveness and access analysis for instance fields accessed by a method call	

Table 0-1. Analysis plug-ins in our framework

[Michael Thies: Combining Static Analysis of Java Libraries with Dynamic Optimization, Dissertation, Shaker Verlag, April 2001]

Lecture Compilation Methods SS 2013 / Slide 235

Objectives:

See analysis methods provided by a tool

In the lecture:

Some modules are related to methods presented in this lecture.

Questions:

Which modules implement a method that is presented in this lecture?