### Program Analysis

**Syntactic analysis** derives information about a program's structure (typically static).

**Semantic analysis** derives information about the properties of dynamic computations of a program (operations, data).

- Determine validity of a program (e.g., type correctness)
- Understand behavior of a program for debugging, maintenance, verification, testing
- Transform (representation of) a program, preserving its semantics
- For debugging, maintenance, verification, testing
- Understand behavior of a program (e.g., type correctness)
- Derive information about the properties of dynamic computations

### Mechanisms for Semantic Analysis

<table>
<thead>
<tr>
<th>Type of Analysis</th>
<th>Static Analysis</th>
<th>Dynamic Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time of performing analysis</td>
<td>Compile time</td>
<td>Runtime</td>
</tr>
<tr>
<td>Target platform given</td>
<td>Module view</td>
<td>Whole program given</td>
</tr>
<tr>
<td>Program input known</td>
<td>Whole program given</td>
<td>Partial (profile information known)</td>
</tr>
</tbody>
</table>

#### Formal Semantics-Based Approaches

- Model checking
- Other approaches
  - Map abstract values back to concrete domain
  - Perform computations on abstract domains
  - Map concrete domain of values to abstract domain

#### Abstract Interpretation

Map concrete domain of values to abstract domain
Perform computations on abstract domains
Map abstract values back to concrete domain

#### Inference Systems

- Set of axioms + inductive / compositional rules of inference
- Infer properties by repeatedly discovering the premises that are satisfied, and by invoking applicable rules of inference
- Source by iteratively solving related constraints together.
- Structured constraints: exploit temporal or spatial structure of data
- Unstructured constraints: unstructured — more powerful

#### Constraint Resolution Systems

- Transform (representation of) a program, preserving its semantics
- For debugging, maintenance, verification, testing
- Understand behavior of a program (e.g., type correctness)
- Derive information about the properties of dynamic computations

### Time of Performing Analysis

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Compile Time</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Platform</td>
<td>Module View</td>
<td>Whole Program Given</td>
</tr>
<tr>
<td>Program Input</td>
<td>Whole Program Given</td>
<td>Partial (Profile Information Known)</td>
</tr>
</tbody>
</table>
1. Dominator-based analysis
   + construct the flow graph / basic block graph
   + identify basic blocks of a routine from MIR or low-level source code
   + reconstruction of if-then-else loops
   + basis for data-flow analyses
   + necessary to enable global optimizations beyond basic blocks

2. Interval analysis
   (recursive)

3. Structural analysis
   (recursive)

Flow analysis
Propagate data flow information along a flow graph
Solve a Flow Analysis Problem:
   Determine the flow of scalar values; Build Data Flow Graph
   Build a Control Flow Graph
   Determine the control structure of the program

Control Flow Analysis
+ necessary to enable global optimizations beyond basic blocks
+ basis for data-flow analysis
+ reconstruction of if-then-else, loops
+ identify basic block of a routine from MIR or low-level source code
+ necessary to enable global optimizations beyond basic blocks

Motivation
In the sequel, we focus on static analysis.

Hybrid forms
acceptable overhead (time, space)
required precision
availability of needed information
choice governed by

Static vs. dynamic analysis

Time of performing analysis (2)

I. DO: 
   \begin{align*}
   y & = 1, \quad c + a \rightarrow c \\
   q & \rightarrow q + a \\
   \end{align*}
Control Flow Analysis – Running Example

Detection of Basic Blocks

- \( B(B) \) = max. sequence of consecutive statements (IR or target level)
- First instruction ("leader") of a BB: either + entry point of a procedure or + call instruction
- Last instruction ("exit") of a BB: either + branch target, or + exit point of a procedure, or + last instruction of a basic block (BB)
- Exception-based control transfer not considered here

Terminology: in Muchnick'97 called control-flow graph (CFG) whereas our CFG (statement level) is there called a "flowchart".

Example: Detecting Basic Blocks; Basic Block Graph

\[
\{ \pi \text{ entry} \in \mathcal{N} \} = (\pi) \text{ entry} \quad \text{entry} \leftarrow \text{ex}it \quad \text{entry} \leftarrow \text{ex}it
\]

whereas \( \text{CFG (statement level)} \) is there called a "flowchart".
Extended Basic Blocks (EBB)

- **Extended basic block (EBB)**: a sequence of instructions beginning with a leader that contains no join nodes other than (maybe) its first node.

- **Extended blocks** are useful for some optimizations, e.g., instruction scheduling.

- **Algorithm for computing the EBBs of a CFG**:
  - Single entry, multiple exits, tree-like internal control flow
  - The control follows a join node (maybe) its first node
  - Adjacent to instructions beginning with a leader

Example: Extended Basic Blocks

- Motivation: Finding Loops
  - Programs spend most of the execution time in loops.

- Loop unrolling, loop parallelization, software pipelining, ...
Graph-theoretic concepts of control-flow analysis (2)

Dominance, immediate dominance, strict dominance

\[ q \neq p \Rightarrow \text{dom} q \Rightarrow \text{dom} p \]

**Graph-theoretic concepts of control-flow analysis (2)**

- **Dominance**
  - Every path from the entry node to \( q \) passes through \( d \).
  - Every path from the entry node to \( p \) passes through \( d \).
  - \( d \) dominates \( q \) and \( p \).

- **Immediate Dominance**
  - \( q \) immediately dominates \( p \) if \( q \) dominates \( p \) and there is no \( c \in N \setminus \{q\} \) with \( \text{dom} c \neq \text{dom} p = \text{dom} q \).

- **Strict Dominance**
  - \( q \) strictly dominates \( p \) if \( q \) dominates \( p \) and \( q \) immediately dominates \( p \).

**Example:** DFS-tree, edge classification [Muchnick Fig. 7.9]

- **Preorder Traversal**
  - Visit each node \( q \in N \) process \( q \) before its descendants.
  - Not unique in DFS-order.

- **Postorder Traversal**
  - Visit each node \( q \in N \) process \( q \) after its descendants.
  - Unique in DFS-order.
Computing dominators

Algorithm 2

[Lengauer/Tarjan’79]
based on depth-first search and path compression

\[
O\left(\alpha(n)\cdot\log^* n\right)
\]

see [Muchnick pp.185–190]
Properties of Natural Loops

Fortechnicalreasons,addapre-header

\[ \text{Muchnick Fig. 7.22} \]

Twonaturalloopsareeitherdisjoint
ornesonestackedintheother.

EachnaturalloopisaSCC.

**Stronglyconnectedcomponent (SCC)**

\[ \begin{align*}
S &= \{ s \in N : s \in R_{\text{N}} \} \\
E &= \{ (s, r) : s, r \in S, (s, r) \in E_{\text{N}} \}
\end{align*} \]

whereeverynodein\( S \)isreachablein\( S \),

EachnaturalloopisaSCC.

ForbothCFAandDFA

Reducibilityofflowgraphs

Intuitively:aflowgraphis

Reducible

ifthereare

Reducibleflowgraphsarewell-structured(loopsproperlynested)

Reducibleflowgraphsarewell-structured(loopsproperlynested)

Intervalanalysis

forbothCFAandDFA

\[ \text{Muchnick Fig. 7.30, 7.31} \]

\[ \text{Muchnick Fig. 7.32} \]

T1-T2analysis

\[ \text{Ullman’73} \]

Example:T1-T2analysis\[ \text{Ullman’73} \],see\[ \text{Muchnick Fig. 7.30} \]

\[ \Rightarrow \]

\[ \Rightarrow \]

Reductionofflowgraphs

Intervalanalysis

\[ \text{Tarjan’72} \]

Reducibilityofflowgraphs

Intuitively:aflow-graph-is

Reducible

ifthereare

Reducibleflowgraphsarewell-structured(loopsproperlynested)

Reducibleflowgraphsarewell-structured(loopsproperlynested)

Intervalanalysis

forbothCFAandDFA

\[ \text{Muchnick Fig. 7.30, 7.31} \]

\[ \text{Muchnick Fig. 7.32} \]

T1-T2analysis

\[ \text{Ullman’73} \]

\[ \text{Muchnick Fig. 7.30, 7.31} \]
Structured Analysis

CFA/DFA follows the hierarchical construction of the source program

---

Control-Flow Analysis: Summary

- Basic blocks, extended basic blocks
- Loop detection
- Dominator-based CFA (standard compiler technology)
- Interprocedural CFA (executing compiler)
- Interprocedural analysis
- Conservative approximation to global information on data flow properties

---

Data Flow Analysis

Conservative approximation to global information on data flow properties

- MAY-problems vs. MUST-problems
- Which definitions of \( \text{var} \) may be relevant for this use?
- Has \( \text{var} \) always the same constant value at this point?
- REACHING DEFINITIONS

---

Structural Analysis

- Extensions to handle arbitrary programs
- For each source language construct
- Equations etc. for dataflow analysis can be pre-formulated
- Every region has an entry point
- By applying grammar rules (productions)

---

Interprocedural

Forward vs. Backward, Iterative vs. Iterative-based vs. Structured...
Example: Reaching definitions

- **Definition d** of variable v:
  - d: v

  S
  S
  S
  d reaches a point p in CFG if there is a path $d \rightarrow p$ in CFG (excluding d, p) that contains no kill of d (= reassignment of v)

- **Undecidable in the formal sense!**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Conservative approximation**: MAY-REACH or MUST-REACH, depending on the application:
  - A basic block B kills all definitions d that write any variable v defined in B.
  - A definition d that is not killed by a basic block B is preserved by B.

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example: Reaching Definitions with bitvector representation**

```
Example: Reaching Definitions

[Muchnick Fig. 7.3]

Basic Block

1
m
in node 1

2
f0
in node 2

3
f1
in node 3

4
i
in node 4

5
f2
in node 8

6
f0
in node 9

7
f1
in node 10

8
i
in node 11

 GEN

 B1

 PRSV

 B1

 RDN

 B1

 DEFINITION

 REACH

 ENTRY

 GEN

 B1

 PRSV

 B1

 RDN

 B1

 DEFINITION

 REACH

 ENTRY

```
Example: Reaching definitions

set equation:
\[ RD_{out} = B \]
\[ GEN = B \]
\[ RD_{in} = B \]
\[ PRSV = B \]
\[ 9 = B \]

bit vector equation:
\[ RD_{out} = B \]
\[ GEN = B \]
\[ X = RD_{in} \]
\[ PRSV = B \]
\[ 9 = B \]

Example: Muchnick p.222

Why does this work?

Algorithm: Iterative computation of reaching definitions

until no more changes occur

\[ RD^{m+1} = RD \]
\[ RD^{m+1} = RD \]
\[ RD^{m+1} = RD \]

for all other \( b \)

for MAY-Reach we initialize

for MUST-Reach we initialize

set equation:
\[ RD_{in} = B \]
\[ Z = \text{Pred} > B ? RD_{out} \]
\[ 9 = B \]

forwardflow problem:

propagating information in direction from entry towards exit

Example: bitvector lattice

Bitvector lattice:
\[ L_{BV} \]
\[ = \text{bitwise AND} \]
\[ = \text{bitwise OR} \]

Example: Dataflow analysis: Lattices

algebraic structure

lattice

set of values representing abstract properties of variables, expressions, etc.

meet operation

join operation

(1) for all \( x, y \in L \) \( x \lor y = y \lor x \) \( \lor \) (commutativity)

(2) for all \( x, y, z \in L \) \( x \lor (y \lor z) = (x \lor y) \lor z \) \( \lor \) (associativity)

(3) for all \( x, y, z \in L \) \( x \lor y \lor z = x \lor y \lor z \) \( \lor \) (closure)

(4) there are two unique elements of \( L \) \( \bot \) \( \lor \) (bottom)

(5) often also distributivity of \( \land \) over \( \lor \)

Example: Reading definitions

Example: Control and Data Flow Analysis


ADVANCED COMPILER CONSTRUCTION — Control and Data Flow Analysis.

Page 41
Lattices: Monotonicity, height; termination

$f : L \rightarrow L$

is monotone iff

$\forall x, y \in L : f(x) \leq f(y)$

height of $L$ is the length of the longest strictly ascending chain:

$\text{height}(L) = \max \{ |\{ x \in L | x \leq y \} | : y \in L \}$

finite height + monotonicity $\Rightarrow$ termination of the fixpoint iteration

Meet overall paths (1)

Meet-over-all-paths (MOP) solution to data flow equations (for forward problems):

- Begin with initial information at entry
- Apply composition of flow functions along all possible paths from entry to each CFG node
- Compose the results by the meet operator:

\[
\text{MOP} = \bigwedge \text{paths from entry to CFG node}
\]

Example: $f : BY \rightarrow Y$ with $f(0) = 0$ and $f(1) = 1$

solution to a set of data flow equations $\leftarrow$

in general not unique

fixed point of a function $f : L \rightarrow L$

Flow functions must be monotone:

\[
\langle x_1 \cdots x_n \rangle = \langle x_1 \cdots x_n \rangle f
\]

\[
\leq_p \text{ generates } d, \text{ kills } r
\]

$\langle x_1 \cdots x_n \rangle$

\[
\text{e.g. reaching definitions:}
\]

$\langle x \rangle \leftarrow L : f$

similarly for backward problems
MOP is the best solution that we could compute statically.

The most precise solution is \( X^P \), but MOP (B) = \( X^H \).

Consider the following artificial construction:

\( A \) solution in is safe if \( B \) \( \subseteq MOP \) \( A \).

Meet over all paths (1)

Meet over all remaining paths to \( B \).

Meet over all paths \( B \) possibly taken at run time.

\( X^H \) is not statically known.

Because the exact subset of the paths really taken at run time

MOP may not be the most precise solution.

\[ \text{Meet all \( g \) in} \quad \{ (g, y) \mid \forall x \in \mathcal{F} (x, g) \subseteq g \} \]

Theorem (Khalil, 1973)

If there exists a fixed point, then there exists a safe solution.

MOP is the best solution that we could compute statically.

Meet over all remaining paths to \( B \).

Meet over all paths \( B \) possibly taken at run time.

\( X^H \) is not statically known.

Because the exact subset of the paths really taken at run time

MOP may not be the most precise solution.

\[ \text{Meet all \( g \) in} \quad \{ (g, y) \mid \forall x \in \mathcal{F} (x, g) \subseteq g \} \]

Theorem (Khalil, 1973)

If there exists a fixed point, then there exists a safe solution.

Meet over all remaining paths to \( B \).

Meet over all paths \( B \) possibly taken at run time.

\( X^H \) is not statically known.

Because the exact subset of the paths really taken at run time

MOP may not be the most precise solution.

\[ \text{Meet all \( g \) in} \quad \{ (g, y) \mid \forall x \in \mathcal{F} (x, g) \subseteq g \} \]
Iterative DFA: Worklist algorithm (1)

Maintain a worklist of blocks whose predecessors’ in values have changed in the last iteration.

Observation: maximal effect on forwarding information

iteration 1

- start with reverse postorder
- queue as worklist

Algorithm

Iterate applying the dataflow equations until no more changes occur

Partial Redundancy analysis

\[ \text{Forward, biliveness (1 bit per use of a variable)} \]

\[ \text{Backward, biliveness (1 bit per definition of an expression)} \]

Liveness analysis

\[ \text{Forward, biliveness (1 bit per definition of an expression)} \]

\[ \text{Available expressions} \]

Survey of data flow problems (1)

classified by:

- information to be computed:
  - Reaching definitions
  - Available expressions
  - Live variables

- direction of information flow:
  - Forward, backward, bidirectional

- lattices used:
  - Const. Propagation analysis

Survey of data flow problems (2)

Copy-propagation analysis

Constant-propagation analysis

Forward, biliveness

Copy-propagation analysis

Backward, biliveness

Liveness analysis

Available expressions

Partial Redundancy analysis

\[ \text{Upwardsexposed uses} \]

\[ \text{Copy-propagation analysis} \]

\[ \text{Forward, biliveness (1 bit per copy assignment)} \]

\[ \text{Copy-propagation analysis} \]

\[ \text{Backward, biliveness (1 bit per use of a variable)} \]

\[ \text{Live variables} \]

\[ \text{Available expressions} \]

\[ \text{Forward, biliveness (1 bit per definition of an expression)} \]

\[ \text{Partial Redundancy analysis} \]

\[ \text{Available expressions} \]

\[ \text{Forward, biliveness (1 bit per definition of an expression)} \]

\[ \text{Available expressions} \]

\[ \text{Forward, biliveness (1 bit per use of a variable)} \]

\[ \text{Available expressions} \]

\[ \text{Reverse postorder} \]

\[ \text{Queue as worklist} \]

[Hecht/Ullman'75]

[Muchnick Fig. 8.6]

[Munchel FG: 8.6] Algorithm: Worklist Algorithm (2)

Advancing/Flushing/Barrier[92] Lazy Code Motion

[Morel/Bernos/91] Bidirectional, biliveness (1 bit per expression computation)

[PU03]

[PU03]
Available Expressions
An expression, say $x+y$, is available at a point $p$ if:
1. Every path from the start node to $p$ evaluates $x+y$, and
2. After the last evaluation prior to reaching $p$, there are no subsequent assignments to $x$ or $y$.

We say that a basic block kills expression $x+y$ if it may assign $x$ or $y$ and does not subsequently recompute $x+y$.

Copy-propagation analysis
Every expression, say $x+y$, is available at a point $p$. If there are no subsequent assignments to $x$ or $y$.

Upward exposed uses
If $x$ may assign $x$ or $y$, then kill $x+y$.

Live variables
Backward, bitvector (1 bit per use of a variable)

Live variables
Backward, bitvector (1 bit per copy assignment)

Copy-propagation analysis
Forward, bitvector (1 bit per copy assignment)
Partial Redundancy Elimination

Example: [Munchik Fig. 8.17]

useful in global register allocation
= maximal union of intersecting DU-chains for a variable
Web
implemented as lists
DU-chain connects a use to all definitions that may reach it
UD-chain connects a definition to all uses it may reach
sparse representation of dataflow information about variables:

[Web Construction Example]

[Dhamdhere,92] - PRe made easy
[Knoop/Rüthing/Steffen,92] - "Lazy Code Motion"
[Morel/Renvoise,91] - bidirectional, bidirectional, bidirectional
(1 bit per expression computation)

[DU,UD-chains, Webs]

implemented as lists
UD-chain connects a use to all definitions that may reach it
DU-chain connects a definition to all uses it may reach
sparse representation of dataflow information about variables:

[Web Construction Example]
Data Flow Analysis: Summary

- Gather global information about dataflow properties
- Correct under-/overestimation depending on intended transformations
- Propagation over the CFG, iterative analysis
- Various problems and methods
- SSA form
- DU/UD chains, webs
- Array data flow analysis [Feautrier'91, Maydan/Hennessy/Lam'91]
- Further DPA methods (interval/structural analysis)
- SSA form
- Generators for DPA analyzers, e.g. Shatil [Feautrier'91], SSA form
- DFA for pointers and heap data structures
- Further topics and outlook:
  - Transfer functions:
  - Monotonicity + finite height = termination of iterative analysis
  - Correct under-/overestimation depending on intended transformations
  - Gather global information about dataflow properties

[Feautrier'91], [Maydan/Hennessy/Lam'91]

DU/UD chains, webs