Control Flow Analysis

- Necessary to enable global optimizations beyond basic blocks
- Basis for data-flow analysis
- Reconstruction of if-then-else, loops from MIR or low-level source code
- Identification of basic blocks for routines

### 1. Dominator-based analysis

#### Detecting basic blocks

- Basic block (BB) = max. sequence of instructions that can be entered only via the first of them, and left only via the last of them.
- First instruction ("leader") of a BB: either
  - Entry point of a procedure
  - Branch target, or
  - Call instruction

#### Extended basic blocks

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

#### Predecessor BBs of a BB

\[
\{ e \in N : \exists b \in E \text{ such that } (b,e) \in E \} = \text{Pred}(e)
\]

#### Successor BBs of a BB

\[
\{ e \in N : \exists b \in E \text{ such that } (e,b) \in E \} = \text{Succ}(e)
\]
Graph-theoretic concepts of control-flow analysis (1)

**depth-first search (dfs):**
- recursively explores descendants of an node before any of its siblings
- first explores descendants of a node in a directed graph

**dfs-number:**
- order in which dfs enters nodes

**tree edges:**
- edges followed by dfs via recursive calls

**non-tree edges:**
- classified as:
  - forward edges
  - back edges
  - cross edges

**dom: d1 dom d2**
- if every possible execution path from d2 to entry includes d1

**strict dominance:**
- d1 sdom d2 if d1 dom d2 and d1 b d2

**postdominance:**
- p1 postdom p2 if every possible execution path from p2 to exit includes p1

Dominance, immediate dominance, strict dominance

not unique depending on ordering of descendants

Dis-free: order in which dfs explores nodes

Depth-first search (dfs): recursively explore descendants of a node before any of its siblings

Given: flow graph G = (N,E) with nodes N and edges E

```
\( \text{if } \text{dom } q \text{ and there is no } c \in N \text{ with } \text{dom } c \text{ and } \text{dom } q \)
\( \text{i.e. } \text{immediately dominates} \) 
```

```
\( \text{idom } d \) is unique for each \( d \in N \)
```

```
\( \text{dom is reflexive, transitive, antisymmetric} \) 
\( \text{partial order on } N \)
```

```
\( \text{if } v \text{ is reachable from entry via } G \text{ then } \) 
\( \text{entry } \text{ dominates } v \) 
```

```
\( \text{for any } q \in N \text{, } \text{q is a root of the dominator tree} \) 
```

Computing dominators

Algorithm 2 [Lengauer/Tarjan'79]

Based on depth-first search and path compression
time $O(e \log n)$ or $O(e \cdot \alpha)$

see [Muchnick pp.185–190]

Loops and Strongly Connected Components

Edge $(w',w)$ is a back edge if $w \in \text{dom}(w')$.
Identifying the natural loop of a back edge:

Algorithm [Muchnick Fig. 7.21]

Reducibility of flow graphs:
Reducible flow graphs are well-structured (loops properly nested)

+ for technical reasons, add a pre-header

Interval analysis
for both CFA and DFA

Example: T1-T2 analysis [Ullman,73] see [Muchnick Fig. 7.30]

Properties of Natural Loops

Reducibility of flow graphs

Intuitively: a flow-graph is reducible if there are no jumps into the middle of loops (e.g., go to's)

In reducible flow graphs are well-structured (loops properly nested)

Interval analysis for both CFA and DFA

+ divide flowgraph into regions (e.g., loops in CFA)
+ repeatedly collapse a region to an abstract node

+ abstract flowgraph
+ + strongly connected component (SCC)

SCC's can be computed with Tarjan's algorithm (extension of dfs)
in time O(n + e)

S is strongly connected component (SCC)

from every other node in V,F there exists an edge in E,
where every node in V,F is reachable in S

= strongly connected component (SCC)

Each natural loop is a SCC.

or one is nested in the other

+ two natural loops are either disjoint
+ for technical reasons, add a pre-header [Muchnick Fig. 7.22]

Identifying the natural loop of a back edge:

can be made reducible by replacing nodes.
Structured Analysis

CFA/DFA follows the hierarchical construction of the source program by applying grammar rules (productions) ...

Equations etc. for dataflow analysis can be pre-formulated...

Example: Reaching Definitions

Definition \( d \) of variable \( v \):

\[ d \vdash v \]

\( d \) reaches a point in the CFG if there is a path from \( d \) to \( p \) in the CFG excluding \( \{d,p\} \) ...

\( d \) reaches \( p \) definition of variable \( v \):

\[ d \vdash p \]

Conservative approximation to global information on dataflow properties that are relevant for optimizations ...

FORWARD vs. BACKWARD, iterative vs. interval-based vs. structured...

Extensions to handle arbitrary flowgraphs...

FORWARD vs. BACKWARD, iterative vs. interval-based vs. structured...

Extensions to handle arbitrary flowgraphs...

FORWARD vs. BACKWARD, iterative vs. interval-based vs. structured...

FORWARD vs. BACKWARD, iterative vs. interval-based vs. structured...

FORWARD vs. BACKWARD, iterative vs. interval-based vs. structured...
Example: Computing reaching definitions
RDin

(1) = def. reaches entry of B

initially, RDin B

00000000

for

MUST-Reach we initialize RDin B

00000000

for all B

Effect of a node B in CFG on definitions d reaching it:
B either kills d or preserves d or generates d

PRSV

B

00010100

(0 = kill/generate, 1 = preserve)

GEN

B

00100000

(1 = generate)

RDout

B

00100000

(1 = def. reaches end of B)

we initialize RDout B

00000000

for all B

Why does this work?

iterate applying the equations to RDin, RDout until no more changes occur.

Algorithm:

\[
A \; (d)^{RDin} \; \cap \; B = (d)^{RDin} \\
\text{set equation:}
\]

\[
A \; (d)^{RDout} \; \cup \; B = (d)^{RDout} \\
\text{direct equation:}
\]

Example: Reaching definitions

Example: Compiling reaching definitions

Example: Bitvector lattice

\[ L_{BV}^3 \]

= bitwise AND,

= bitwise OR

1 1 1
1 1 0
1 0 1
0 1 1
0 0 0

"#" = 101
"&" = 001

partial order:

\[ x \leq y \text{ iff } x \leq y \]

(transitive, antisymmetric, reflexive)

+ meet operation

x \wedge y

join operation

x \vee y

(commutativity)

associativity)

distributivity)

(1) for all \( x, y, z \in L \):

(2) for all \( x, y \in L \):

(3) for all \( x, y, z \in L \):

(4) there are two unique elements of \( L \):

(5) often also distributivity of \( \wedge \), \( \vee \), \( \neg \) given

For all possible executions of the procedure representing abstract properties of variables, expressions etc.,

+ set of values

algebraic structure lattice \( (4, \cup, \cap) \)

Example: Lattices

}\n
\]

Setup of initial state of the program, e.g., initial entry of program

\[ RD_{RD}^i = \{ (00000000) \} \]

\[ RD_{GE}^i = \{ (00000100) \} \]

\[ RD_{PSV}^i = \{ (001001000) \} \]

Effect of a node B in CFG on definitions reaching it:

for all B

unless read is initial entry of B

Initial, RDout entry of B

for MUST-Reach

\[ RD_{RD}^f = \{ (00000000) \} \]

\[ RD_{GE}^f = \{ (00001000) \} \]

\[ RD_{PSV}^f = \{ (001001000) \} \]

Example: Bitvector lattice

\[ L_{BV}^3 \]

= bitwise AND,

= bitwise OR

1 1 1
1 1 0
1 0 1
0 1 1
0 0 0

"#" = 101
"&" = 001

partial order:

\[ x \leq y \text{ iff } x \leq y \]

(transitive, antisymmetric, reflexive)

+ meet operation

x \wedge y

join operation

x \vee y

(commutativity)

associativity)

distributivity)

(1) for all \( x, y, z \in L \):

(2) for all \( x, y \in L \):

(3) for all \( x, y, z \in L \):

(4) there are two unique elements of \( L \):

(5) often also distributivity of \( \wedge \), \( \vee \), \( \neg \) given

For all possible executions of the procedure representing abstract properties of variables, expressions etc.,

+ set of values

algebraic structure lattice \( (4, \cup, \cap) \)

Example: Lattices

\[ L_{BV}^3 \]

= bitwise AND,

= bitwise OR

1 1 1
1 1 0
1 0 1
0 1 1
0 0 0

"#" = 101
"&" = 001

partial order:

\[ x \leq y \text{ iff } x \leq y \]

(transitive, antisymmetric, reflexive)

+ meet operation

x \wedge y

join operation

x \vee y

(commutativity)

associativity)

distributivity)

(1) for all \( x, y, z \in L \):

(2) for all \( x, y \in L \):

(3) for all \( x, y, z \in L \):

(4) there are two unique elements of \( L \):

(5) often also distributivity of \( \wedge \), \( \vee \), \( \neg \) given

For all possible executions of the procedure representing abstract properties of variables, expressions etc.,

+ set of values

algebraic structure lattice \( (4, \cup, \cap) \)

Example: Lattices

\[ L_{BV}^3 \]

= bitwise AND,

= bitwise OR

1 1 1
1 1 0
1 0 1
0 1 1
0 0 0

"#" = 101
"&" = 001

partial order:

\[ x \leq y \text{ iff } x \leq y \]

(transitive, antisymmetric, reflexive)

+ meet operation

x \wedge y

join operation

x \vee y

(commutativity)

associativity)

distributivity)

(1) for all \( x, y, z \in L \):

(2) for all \( x, y \in L \):

(3) for all \( x, y, z \in L \):

(4) there are two unique elements of \( L \):

(5) often also distributivity of \( \wedge \), \( \vee \), \( \neg \) given

For all possible executions of the procedure representing abstract properties of variables, expressions etc.,

+ set of values

algebraic structure lattice \( (4, \cup, \cap) \)
Lattices: Monotonicity, height, termination

An element \( f \) in a lattice \( L \) is monotone iff for any \( x, y \) in \( L \):

- \( x \leq y \Rightarrow f(x) \leq f(y) \)
- \( f(x) \leq f(y) \Rightarrow x \leq y \)

The height of \( L \) is defined as the length of the longest strictly ascending chain in \( L \):

\[ \text{height}(L) = \max \{ n \mid \exists \text{an ascending chain } x_1 \leq x_2 \leq \ldots \leq x_{n+1} \} \]

In general, a solution to the data flow equations:

- Is a function \( f \) (possibly chosen arbitrarily)
- \( L \) is a fixed point of \( f \)

Meet-over-all-paths (MOP) solution for forward problems:

1. Begin with initial information \( \text{Init} \) at entry.
2. Apply composition of flow functions along all possible paths from entry to every CFG node.
3. Compose the results using the meet operator.

For backward problems:

1. Begin with the meet of all initial information at exit.
2. Compose the results using the meet operator.

Fixed points:

Fixed points of a mapping \( f \) are elements \( z \) in \( L \) such that \( f(z) = z \).

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

\[ \text{MOP}(f) = \text{Meet} \left( \bigcup_{\text{paths}} \text{Flow}(f) \right) \]

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

\[ \text{MOP}(f) = \text{Meet} \left( \bigcup_{\text{paths}} \text{Flow}(f) \right) \]
Fixpointsolutions of the dataflow equations

\[ M_{\text{F}} \] (maximal w.r.t. &
most information and still
the „real“ solution)

A solution \( \in \{0,1\}^* \) of the dataflow equations is \( \text{safe} \) iff \( \in \)
\( B \) &\( M_{\text{F}} \) \( \subseteq \)

\[ \{ M \in \mathbb{B}^* : \text{all flow functions are distributive over } \} \]

\[ \text{Iterative Data Flow Analysis} \]

\( \text{given: CFG} \quad G \quad N \) & \( \mathcal{E} \), Lattice \( \mathcal{L} \) & \( \text{dataflow equations} \)

\[ \begin{align*}
\text{for } \mathcal{B} & \quad \mathcal{P}_{\text{Pred}} \mathcal{B} \quad \mathcal{P}_{\text{out}} \\
\text{otherwise} & \quad = \mathcal{B} \quad \mathcal{P}_{\text{Pred}} \mathcal{B} \quad \mathcal{P}_{\text{out}} \\
\text{entry} & = g \\
\{ (g)w \}_{\mathcal{B}} \cup \mathcal{I}_{\mathcal{B}} \quad \{ (g)w \}_{\mathcal{B}} \cup \mathcal{I}_{\mathcal{B}} \\
\text{for } \mathcal{B} & \quad \mathcal{P}_{\text{Pred}} \mathcal{B} \quad \mathcal{P}_{\text{out}} \\
\text{otherwise} & \quad = \mathcal{B} \quad \mathcal{P}_{\text{Pred}} \mathcal{B} \quad \mathcal{P}_{\text{out}} \\
\text{entry} & = g \\
\{ (g)w \}_{\mathcal{B}} \cup \mathcal{I}_{\mathcal{B}} \quad \{ (g)w \}_{\mathcal{B}} \cup \mathcal{I}_{\mathcal{B}} \\
\end{align*} \]
Worklist algorithm
+ maintain a worklist of blocks $B$ whose predecessors' in values have changed in the last iteration
+ worklist contains initially all BB's (except entry)
+ iterate applying the dataflow equations until no more changes occur
Observation: maximal effect on forwarding information
+ start with reverse postorder
+ queue as worklist

Example: [Muchnick Fig. 8.17] useful in global register allocation
used in global register allocation
$\text{Web} \; \forall \text{variable } v$
implemented as lists

DU-chains, UD-chains, webs
sparse representation of dataflow information about variables:

Data Flow Analysis: Summary
+ gather global information about dataflow properties
+ correct under-/overestimation depending on intended transformations
+ propagation over the CFG, iterative analysis
+ lattice theory: monotonicity + finite height $\iff$ termination of iteration of iteration
+ various problems and methods
+ correct under- / overestimation depending on intended transformations
+ gather global information about dataflow properties

Data Flow Analysis, further topics and outlook:
+ further DFA methods (interval/structural analysis)
+ array dataflow analysis
+ DFA for pointers and heap data structures
+ DU/UD-chains, webs
+ SSA form
+ DU/UD/chain + webs
+ various problems and methods
+ correct under-/overestimation depending on intended transformations
+ gather global information about dataflow properties

Worklist algorithm
+ start with reverse postorder
+ queue as worklist
+ iterate applying the dataflow equations until no more changes occur
+ $A + Z \iff \text{iteration for a (sub-) CFG with a back edges (Hecht/Ollmann75)}$
+ $A + Z \iff \text{iteration for a (sub-) CFG with a back edges (Hecht/Ollmann75)}$