Compiler Design

Data Flow Analysis

Hwansoo Han
Control Flow Graph

- **What is CFG?**
  - Represents program structure for internal use of compilers
  - Used in various program analyses
  - Generated from AST or a sequential list of statements

- **Basic block**
  - To build CFG, first partition a program into *basic blocks*
  - Longest possible sequence of straight-line code
    - no branch-in/out in the middle of basic block
    - single entry at the beginning (label, join-point)
    - single exit at the end (can be branch or non-branch)
CFG Example

- **CFG = Basic blocks + Control transfers**
- **Unreachable code elimination**
  - DFS (or BFS) on the CFG can remove unreachable blocks
  - As a result, the declaration of `z` may also be removed.

```c
int test(int n)
{
    int x,y,z,w;
    if (n < 2)
        x = 3;
    else
        x = 0;
    goto l;
    z = x + 5;
    y = z * 3;
    l: w = x + 10;
    y = w * 2;
    return y;
}
```
Data Flow Problem

- **Applying data flow analysis on CFG**
  - Identify how data are manipulated/modified
  - Optimization relies on information quality acquired thru DFA
  - DFA is a process to solve the effects of statements, basic blocks or larger segments

- **Typical data flow problems**
  - Available expressions, reaching definitions
  - Aliases, live variables, copy propagation
  - Upward exposed uses (live-in for basic block)

Analyze data information at each point
Data Flow Analysis (DFA)

* Data flow information (DFI)
  - Abstract property of program data (variable, value, name)
  - At various points in the control flow

* Data flow equation (DFE)
  - Represents effects of statements, basic blocks, or program segments
  - Use systematic equations using following components
    1. sets or functions (\(in, out, gen, kill, f, g, \ldots\))
    2. operations (\(\circ, \cap, \cup, -\)) on sets/functions
  - To gather DFI, DFEs are solved by well-defined mathematical procedures.
DFA on Program Hierarchy

- **Local data flow analysis**
  - Composes effect of each statement within a basic block
  - Proceeds sequentially from the leader to the last statement in the basic block

- **Global data flow analysis**
  - Composes effect of each program segment (basic blocks, loops or intervals) at the segment boundaries in a CFG
  - Proceeds along the control transfer in a control flow graph
Local Data Flow Analysis

- **Transfer function** ($f_S$)
  - Represents effects of statement $S$ on DFI within a basic block.
  - Corresponding data flow equation
    \[
    \text{out}[S] = f_S(\text{in}[S])
    \]

- **Typical transfer function**
  \[
  f_S(\text{in}[S]) = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])
  \]
  - $\text{gen}[S]$: DFI newly generated by $S$
  - $\text{kill}[S]$: DFI that was originally in $\text{in}[S]$, but regenerated by $S$
Example: Local Reaching Definition

- **DFI of reaching definition**
  - Which variable definitions reach which points of uses?

- **DFE for each statement S.**
  - in[S] : set of definitions that *may* reach S
  - out[S] : set of definitions that are valid right after S
  - kill[S] : set of definitions that reach S but are killed because S assigns new values to the definitions
  - gen[S] : set of definitions that S newly assigns values to and not subsequently killed by itself

- Transfer function: \( f_S(\text{in}[S]) = \text{gen}[S] \cup (\text{in}[S] \setminus \text{kill}[S]) \)
Computing local reaching definitions

\[ S_{10} \quad x := y + 4 \]
\[ \text{kill}[S_{10}] = \{x_*, \text{gen}[S_{10}] = \{x_{S_{10}}\} \]
\[ \text{in}[S_{10}] = \text{in}[B] = \{v_{S_1}, x_{S_7}, y_{S_9}\} \]
\[ \text{in}[S_{11}] = \text{out}[S_{10}] = \{v_{S_1}, x_{S_{10}}, y_{S_9}\} \]
\[ S_{11} \quad y := y + v \]
\[ \text{kill}[S_{11}] = \{y_*, \text{gen}[S_{11}] = \{y_{S_{11}}\} \]
\[ \text{in}[S_{11}] = \text{out}[S_{10}] = \{v_{S_1}, x_{S_{10}}, y_{S_{11}}\} \]
\[ \text{in}[S_{12}] = \text{out}[S_{11}] = \{v_{S_1}, x_{S_{10}}, y_{S_{11}}\} \]
\[ S_{12} \quad z := x \cdot v \]
\[ \text{kill}[S_{12}] = \{z_*, \text{gen}[S_{12}] = \{z_{S_{12}}\} \]
\[ \text{in}[S_{12}] = \text{out}[S_{11}] = \{v_{S_1}, x_{S_{10}}, y_{S_{11}}, z_{S_{12}}\} \]
\[ \text{in}[S_{13}] = \text{out}[S_{12}] = \{v_{S_1}, x_{S_{10}}, y_{S_{11}}, z_{S_{12}}\} \]
\[ S_{13} \quad x := x - z \]
\[ \text{kill}[S_{13}] = \{x_*, \text{gen}[S_{13}] = \{x_{S_{13}}\} \]
\[ \text{in}[S_{13}] = \text{out}[S_{12}] = \{v_{S_1}, x_{S_{13}}, y_{S_{11}}, z_{S_{12}}\} \]
\[ \text{in}[S_{14}] = \text{out}[S_{13}] = \{v_{S_1}, x_{S_{13}}, y_{S_{11}}, z_{S_{12}}\} \]
\[ S_{14} \quad \text{goto L on } z > 0 \]
\[ \text{kill}[S_{14}] = \{\}, \text{gen}[S_{14}] = \{\} \]
\[ \text{out}[B] = \text{out}[S_{14}] = \{v_{S_1}, x_{S_{13}}, y_{S_{11}}, z_{S_{12}}\} \]
Global Data Flow Analysis

- **Transfer function**
  
  \[ out[X] = f_X(in[X]) \]
  
  - \( in[X] \): DFI given to the beginning of \( X \)
  - \( out[X] \): output DFI at the end of \( X \),
  - Collects outputs of predecessors
  
  \[ in[X] = \land_{P \in \text{Pred}[X]} out[P] \]

- **A typical form of a transfer function**
  
  \[ f_X(in[X]) = gen[X] \cup (in[X] - kill[X]) \]
  
  - \( gen[X] \): DFI newly generated within \( X \)
  - \( kill[X] \): DFI that was originally in \( in[X] \), but regenerated within \( X \)
Example: Global Reaching Definition

- **DFE components for each basic block B**
  - in[B] : set of definitions that *may* reach the beginning of B
  - out[B] : set of definitions that may reach the end of B
  - kill[B] : set of definitions that reach B but are killed by being assigned values in it
  - gen[B] : set of definitions that are newly assigned values in B and not subsequently killed in it

- Transfer function: \( f_B(\text{in}[B]) = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \)
Computing Reaching Definition

\[ S_{50} \ x := y + 4 \]
\[ S_{51} \ y := y + v \]
\[ S_{52} \ z := x \cdot v \]
\[ S_{53} \ x := x - z \]
\[ S_{54} \ \text{goto} \ L \ 	ext{on} \ z > 0 \]

\[ v_{S_{12}} \]
\[ x_{S_{27}} \]
\[ y_{S_{39}} \]

\[ v_{S_{12}} \]
\[ x_{S_{53}} \]
\[ y_{S_{51}} \]
\[ z_{S_{52}} \]

\[ \text{in}[B_4] \]
\[ \text{kill}[B_4] \]
\[ \text{gen}[B_4] \]

\[ \text{out}[B_4] = \text{gen}[B_4] \cup (\text{in}[B_4] - \text{kill}[B_4]) \]
Propagating DFI thru Control Flow

- Propagate DFI to a **join node**
  - A node \( u \) with multiple predecessors \( p_1, \ldots, p_n \)
  - Need to specify how to combine the DFI propagated from all its predecessors
  - Define a binary *meet* (or called *confluence*) operation \( \land \)
    - Find greatest lower bound (glb) on lattice

\[
in[u] = out[p_1] \land out[p_2] \land \ldots \land out[p_n] = \land_{i=1,n} out[p_i]
\]
Propagating Reaching Definitions

- The meet operation in the reaching definitions problem is union.

```plaintext
S_{50} x := y+4
S_{51} y := y+v
S_{52} z := x*v
S_{53} x := x-z
S_{54} goto L on z>0

S_{40} read(x)
S_{41} v := z*v
S_{42} goto L on x=0

B_3

S_{40} x_S{40}
S_{41} v_S{12}
S_{42} z_S{30}
S_{41} x_S{27}
y_S{39}
v_S{12}

v_S{12}

in[B_4]

B_4

S_{55} x := y+4
S_{51} y := y+v
S_{52} z := x*v
S_{53} x := x-z
S_{54} goto L on z>0

S_{55} x_S{55}
y_S{51}
z_S{52}
v_S{41}

in[B_8] = out[B_4]

B_4

S_{55} x := y+4
S_{51} y := y+v
S_{52} z := x*v
S_{53} x := x-z
S_{54} goto L on z>0

S_{55} x_S{55}
y_S{51}
z_S{52}
v_S{41}


B_8

S_{70} L: t := z+x
S_{71} s := y*v
...

S_{70} ...

B_6

S_{70} ...
```
Iterative Data Flow Analysis

- Given $G = (N, E, S)$, forward data-flow problem’s DFE
  
  $\text{in}[B] = \text{Init}$ \hspace{1cm} (Init = $\Phi$ for Reaching Definition)
  
  $\text{out}[B] = f_B ( \text{in}[B] ) = \text{gen}[B] \cup ( \text{in}[B] - \text{kill}[B] )$

- A typical algorithm for iterative data flow analysis

  ```
  for each program segment $X$ do
    initialize $\text{out}[X]$;
  od
  while changes to any $\text{out}[X]$ occur do
    for each segment $X$ in some order do
      $\text{in}[X] = \land \text{out}[P_X]$; \hspace{1cm} \text{all predecessors } P_X \text{ of } X
      $\text{out}[X] = \text{gen}[X] \cup (\text{in}[X] - \text{kill}[X])$;
    od
  od
  ```
Live Variables Problem

- **Dataflow problem**
  
  “Find which variables are live in each basic block!”

- **Definition of “live”**
  
  - A variable $v$ is *live* at point $p$, if the value of $v$ is used along some path in CFG starting from $p$.
  - Otherwise, $v$ is *dead*. 
A Framework For Live Variables

- **Backward dataflow problem**
  - DFI propagates backward the flow of control
  - DFI in live variables problem is a set of ‘variables’:
    
    \[ V = \{ v \mid v \subseteq \text{a set of all variables in the program} \} \]

- **Transfer function and meet operator**
  - a variable is live at the beginning of B
    - if used in B before redefinition (local definition)
    - or live at the end of B and is not redefined in B
      
      \[ \text{in}[B] = f_B(\text{out}[B]) = \text{use}[B] \cup (\text{out}[B] - \text{def}[B]) \]
  - a variable is live at the end of B if it is live at the beginning of any of its successors.
    
    \[ \text{out}[B] = \bigcup \text{in}[S] \rightarrow \bigwedge = \bigcup \]
    
    \[ S \in \text{succ}(B) \]
Forward vs. Backward Data Flow

**Forward dataflow equation**

\[
\begin{align*}
\text{in}[\ast] &= \text{Init}, \\
\text{in}[B] &= \bigwedge_{P \in \text{pred}(B)} \text{out}[P] \\
\text{out}[B] &= f_B( \text{in}[B] ) = \text{gen}[B] \cup ( \text{in}[B] - \text{kill}[B] )
\end{align*}
\]

**Backward dataflow equation**

\[
\begin{align*}
\text{out}[\ast] &= \text{Init}, \\
\text{out}[B] &= \bigwedge_{S \in \text{succ}(B)} \text{in}[S] \\
\text{in}[B] &= f_B( \text{out}[B] ) = \text{gen}[B] \cup ( \text{out}[B] - \text{kill}[B] )
\end{align*}
\]
Backward Data Flow Analysis

To handle backward data flow analysis,

1. Build a reverse CFG $G^R=(N,E^R)$ from the original CFG $G=(N,E)$ s.t. for each $(u,v) \in E$, there exists $(v,u) \in E^R$.

2. Apply the same algorithm that is applied to forward data flow problems.
Iterative Algorithm For Liveness

- **Iterative backward data flow analysis to compute live variables**

  for each basic block B do
  
in[B] = use[B];        // $f_B(\emptyset) = \text{use}[B] \cup (\emptyset - \text{def}[B])$ where
  
od                     // $\emptyset$ is the initial DFI of live variables
  
  while changes to any in[B] occur do
  
  for each X in some order do
  
  out[B] = $\cup$ in[sB];
  
  all successors sB of B
  
  in[B] = use[B] $\cup$ (out[B] - def[B]);
  
  od
  
  od
Code Example

Initial Condition (in[] = use[])

out[B₁] = {} in[B₁] = {u,v,w}
out[B₂] = {} in[B₂] = {x,y}
out[B₃] = {} in[B₃] = {y}
out[B₄] = {} in[B₄] = {x}
out[B₅] = {} in[B₅] = {x}

After 1st iteration (postorder)

out[B₅] = {x,y} in[B₅] = {x,y}
out[B₃] = {x,y} in[B₃] = {x,y}
out[B₄] = {x,y} in[B₄] = {x,y}
out[B₂] = {x,y} in[B₂] = {x,y}
out[B₁] = {x,y} in[B₁] = {u,v,w}

After 2nd iteration (stabilized)

out[B₅] = {x,y} in[B₅] = {x,y}
out[B₃] = {x,y} in[B₃] = {x,y}
out[B₄] = {x,y} in[B₄] = {x,y}
out[B₂] = {x,y} in[B₂] = {x,y}
out[B₁] = {x,y} in[B₁] = {u,v,w}

PostOrder: B₅,B₃,B₄,B₂,B₁

use[B₁] = {u,v,w} def[B₁] = {x,y,t}
use[B₂] = {x,y} def[B₂] = {x,y}
use[B₃] = {y} def[B₃] = {t}
use[B₄] = {x} def[B₄] = {x}
use[B₅] = {x} def[B₅] = { }
How Fast Is The Iterative Algorithm?

- **Execution time**
  - One pass visits all nodes (basic blocks)
  - Number of visits to nodes: $O(n^2)$ algorithm
    - length of longest acyclic path on CFG
  - Merge straight basic blocks to reduce number of nodes

- **Bit vector**
  - Instead of set operations, use bit vector and bit operations
    - reaching definition: 1 bit for 1 variable
    - available expression: 1 bit for 1 definition of expression

- **Work-list algorithm**
  - Only visit nodes which have changed in[X]
    - *i.e.* successors of changed out[X]

- **Visiting order**
  - Post-order, reverse post-order (DFS)
Work List Iterative Algorithm

- **Typical Iterative algorithm implements worklist**

```plaintext
for each \( x \in V \) do
    \( \text{OUT}(x) = \text{GEN}(x) \cup (\text{IN}(x) - \text{KILL}(x)) \); \hspace{1cm} // \text{IN}(x) = "initial value"
od

\text{worklist} \leftarrow \text{all } x \in V

while \text{worklist} \neq \emptyset do
    remove a node \( x \) from \text{worklist}
    oldout(x) = \text{OUT}(x);
    \text{IN}(X) = \bigwedge_{P \in \text{pred}(X)} \text{OUT}(P);
    \text{OUT}(X) = \text{GEN}(x) \cup (\text{IN}(x) - \text{KILL}(x)); \hspace{1cm} // = \text{GEN} \mid (\text{IN} \& \neg \text{KILL})
    if oldout(x) \neq \text{OUT}(x) then
        \text{worklist} \leftarrow \text{worklist} \cup \text{succ}(x);
    od
```
Visiting Order

- **To avoid unnecessary work**
  - Limit the number of visits by visiting a node roughly after all its predecessors
  - Post-order for backward problem
  - Reverse post-order for forward problem (reverse PO ≈ DFS)

```
main()        // Post-Order
  count = 1;
  DFS-visit(root);

DFS-visit(n)
  mark n as “visited”
  foreach s ∈ succ(n) not yet visited
    DFS-visit(s);
  PostOrder(n) = count++;
```

post-order:  d e b f g c a
reverse PO:  a c g f b e d
Summary

❖ Data flow analysis on CFG
  - Data flow information
  - Data flow equation
    - \( in[B] = \bigwedge_{P \in \text{Pred}[B]} out[P] \)
  - Define a meet operation for join points

❖ Faster Iterative Algorithm for DFA
  - Bit vector instead of set operation
  - Work list
  - Visiting order