Dataflow Analysis

Iterative Data-flow Analysis and Static-Single-Assignment
Optimization And Analysis

- Improving efficiency of generated code
  - Correctness: optimized code must preserve meaning of the original program
  - Profitability: optimized code must improve code quality

- Program analysis
  - Ensure safety and profitability of optimizations
  - Compile-time reasoning of runtime program behavior
    - Undecidable in general due to unknown program input
    - Conservative approximation of program runtime behavior
    - May miss opportunities, but ensure all optimizations are safe

- Data-flow analysis
  - Reason about flow of values between statements
  - Can be used for program optimization or understanding
Control-Flow Graph

- Graphical representation of runtime control-flow paths
  - Nodes of graph: basic blocks (straight-line computations)
  - Edges of graph: flows of control
- Useful for collecting information about computation
  - Detect loops, remove redundant computations, register allocation, instruction scheduling...
- Alternative CFG: Each node contains a single statement
Building Control-Flow Graphs
Identifying Basic Blocks

- Input: a sequence of three-address statements
- Output: a list of basic blocks
- Method:
  - Determine each statement that starts a new basic block, including
    - The first statement of the input sequence
    - Any statement that is the target of a goto statement
    - Any statement that immediately follows a goto statement
  - Each basic block consists of
    - A starting statement S0 (leader of the basic block)
    - All statements following S0 up to but not including the next starting statement (or the end of input)

```
......
i := 0
s0: if i < 50 goto s1
    goto s2
s1: t1 := b * 2
    a := a + t1
    goto s0
s2: ...
```

Starting statements:
```
i := 0
S0,
goto S2
S1,
S2
```
Building Control-Flow Graphs

- Identify all the basic blocks
  - Create a flow graph node for each basic block
- For each basic block B1
  - If B1 ends with a jump to a statement that starts basic block B2, create an edge from B1 to B2
  - If B1 does not end with an unconditional jump, create an edge from B1 to the basic block that immediately follows B1 in the original evaluation order

```
......
i := 0
s0: if i < 50 goto s1
    goto s2
s1: t1 := b * 2
    a := a + t1
    goto s0
S2: ...
```

Diagram:
```
i := 0
S0: if i < 50 goto s1
  goto s2
s1: t1 := b * 2
    a := a + t1
    goto s0
S2: ...
```
Exercise:
Building Control-flow Graph

......
i = 0; z = x
while (i < 100) {
i = i + 1;
if (y < x) z=y;
A[i]=i;
}
....
Live Variable Analysis

- A data-flow analysis problem
  - A variable $v$ is live at CFG point $p$ iff there is a path from $p$ to a use of $v$ along which $v$ is not redefined
  - At any CFG point $p$, what variables are alive?

- Live variable analysis can be used in
  - Global register allocation
    - Dead variables no longer need to be in registers
  - SSA (static single assignment) construction
    - Dead variables don’t need $\emptyset$-functions at CFG merge points
  - Useless-store elimination
    - Dead variables don’t need to be stored back in memory
  - Uninitialized variable detection
    - No variable should be alive at program entry point
Computing Live Variables

- **Domain:**
  - All variables inside a function

- **Goal:** LiveIn(n) and LiveOut(n)
  - Variables alive at each basic block n

- For each basic block n, compute
  - UEVar(n)
    - vars used before defined
  - VarKill(n)
    - vars defined (killed by n)

- Formulate flow of data

\[
\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} \text{LiveIn}(m)
\]
\[
\text{LiveIn}(m) = \text{UEVar}(m) \cup (\text{LiveOut}(m) - \text{VarKill}(m))
\]

\[\Rightarrow\]
\[
\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) - \text{VarKill}(m)))
\]
Algorithm: Computing Live Variables

- For each basic block $n$, let
  - $UEVar(n)$ = variables used before any definition in $n$
  - $VarKill(n)$ = variables defined (modified) in $n$ (killed by $n$)

Goal: evaluate names of variables alive on exit from $n$

- $LiveOut(n)$ = $\bigcup \left( UEVar(m) \cup (LiveOut(m) - VarKill(m)) \right)_{m \in succ(n)}$

```plaintext
for each basic block $bi$
    compute $UEVar(bi)$ and $VarKill(bi)$
    $LiveOut(bi) := \emptyset$
for (changed := true; changed; )
    changed = false
for each basic block $bi$
    old = $LiveOut(bi)$
    $LiveOut(bi) = \bigcup \left( UEVar(m) \cup (LiveOut(m) - VarKill(m)) \right)_{m \in succ(bi)}$
    if ($LiveOut(bi) \neq old$) changed := true
```
Solution
Computing Live Variables

- **Domain**
  - `a,b,c,d,e,f,m,n,p,q,r,s,t,u,v,w`

<table>
<thead>
<tr>
<th>UE var</th>
<th>Var kill</th>
<th>Live Out</th>
<th>Live Out</th>
<th>Live Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a,b</td>
<td>m,n</td>
<td>∅</td>
<td>a,b,c,d,f</td>
</tr>
<tr>
<td>B</td>
<td>c,d</td>
<td>p,r</td>
<td>∅</td>
<td>a,b,c,d</td>
</tr>
<tr>
<td>C</td>
<td>a,b,c,d</td>
<td>q,r</td>
<td>∅</td>
<td>a,b,c,d,f</td>
</tr>
<tr>
<td>D</td>
<td>a,b,f</td>
<td>e,s,u</td>
<td>∅</td>
<td>a,b,c,d</td>
</tr>
<tr>
<td>E</td>
<td>a,c,d,f</td>
<td>e,t,u</td>
<td>∅</td>
<td>a,b,c,d</td>
</tr>
<tr>
<td>F</td>
<td>a,b,c,d</td>
<td>v,w</td>
<td>∅</td>
<td>a,b,c,d</td>
</tr>
<tr>
<td>G</td>
<td>a,b,c,d</td>
<td>m,n</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>
Other Data-Flow Problems
Reaching Definitions

- Domain of analysis
  - The set of definition points in a procedure

- Reaching definition analysis
  - A definition point $d$ of variable $v$ reaches CFG point $p$ iff
    - There is a path from $d$ to $p$ along which $v$ is not redefined
    - At any CFG point $p$, what definition points can reach $p$?

- Reaching definition analysis can be used in
  - Build data-flow graphs: where each operand is defined
  - SSA (static single assignment) construction
    - An IR that explicitly encodes both control and data flow
Reaching Definition Analysis

- For each basic block \( n \), let
  - \( \text{DEDef}(n) \) = definition points whose variables are not redefined in \( n \)
  - \( \text{DefKill}(n) \) = definitions obscured by redefinition of the same name in \( n \)

- Goal: evaluate all definition points that can reach entry of \( n \)
  - \( \text{Reaches}_{\text{exit}}(m) = \text{DEDef}(m) \cup (\text{Reaches}_{\text{entry}}(m) - \text{DefKill}(m)) \)
  - \( \text{Reaches}_{\text{entry}}(n) = \bigcup \text{Reaches}_{\text{exit}}(m) \mid m \in \text{pred}(n) \)
Example

```c
void fee(int x, int y)
{
    int I = 0;
    int z = x;
    while (I < 100) {
        I = I + 1;
        if (y < x) z = y;
        A[I] = I;
    }
}
```

- Compute the set of reaching definitions at the entry and exit of each basic block through each iteration of the data-flow analysis algorithm
More About Dataflow Analysis

- Sources of imprecision
  - Unreachable control flow edges, array and pointer references, procedural calls

- Other data-flow programs
  - Very busy expression analysis
    - An expression $e$ is very busy at a CFG point $p$ if it is evaluated on every path leaving $p$, and evaluating $e$ at $p$ yields the same result.
    - At any CFG point $p$, what expressions are very busy?
  - Constant propagation analysis
    - A variable-value pair $(v, c)$ is valid at a CFG point $p$ if on every path from procedure entry to $p$, variable $v$ has value $c$
    - At any CFG point $p$, what variables have constants?
The Overall Pattern

- Each data-flow analysis takes the form
  
  \[
  \text{Input}(n) := \emptyset \quad \text{if } n \text{ is program entry/exit} \\
  := \Lambda \quad m \in \text{Flow}(n) \quad \text{Result}(m) \quad \text{otherwise} \\
  \text{Result}(n) = f_n (\text{Input}(n))
  \]

  - \(\Lambda\) is \(\cap\) or \(\cup\) (may vs. must analysis)
    - May analysis: properties satisfied by at least one path (\(\cup\))
    - Must analysis: properties satisfied by all paths (\(\cap\))

- Flow\((n)\) is pred\((n)\) or succ\((n)\) (forward vs. backward flow)
  - Forward flow: data flow forward along control-flow edges.
    - Input\((n)\) is data entering \(n\), Result is data exiting \(n\)
    - Input\((n)\) is \(\emptyset\) if \(n\) is program entry
  - Backward flow: data flow backward along control-flow edges.
    - Input\((n)\) is data exiting \(n\), Result is data entering \(n\)
    - Input\((n)\) is \(\emptyset\) if \(n\) is program exit

- \(f_n\) is the transfer function associated with each block \(n\)
Iterative dataflow algorithm

for each basic block bi
    compute Gen(bi) and Kill(bi)
    Result(bi) := ∅
for (changed := true; changed; )
    changed = false
for each basic block bi
    old = Result(bi)
    Result(bi) =
        ∩ or ∪
        [m ∈ pred(bi) or succ(bi)]
        (Gen(m) ∪ (Result(m) - Kill(m)))
    if (Result(bi) != old)
        changed := true

- Iterative evaluation of result until a fixed point is reached
  - Always terminate?
    - If the results are bounded and grow monotonically, then yes; Otherwise, no.
    - Fixed-point solution is independent of evaluation order
  - What answer is computed?
    - Unique fixed-point solution
    - Meet-over-all-paths solution
  - How long does it take the algorithm to terminate?
    - Depends on traversing order of basic blocks
Traverse Order Of Basic Blocks

- Facilitate fast convergence to the fixed point
- Postorder traversal
  - Visits as many of a node’s successors as possible before visiting the node
  - Used in backward data-flow analysis
- Reverse postorder traversal
  - Visits as many of a node’s predecessors as possible before visiting the node
  - Used in forward data-flow analysis
Static Single Assignment form

- Data-flow analysis
  - Analyze data flow properties on control flow graph
  - Each analysis needs several passes over CFG

- Static Single Assignment form
  - Encode both control-flow and data-flow in a single IR
    - An intermediate representation
  - Each variable is assigned exactly once
    - Each use of variable has a single definition

- Steps:
  - Rename redefinition of variables
  - Use $\emptyset$-functions to merge conflicting definitions when paths meet
Construction Of SSA form

- Naïve algorithm: maximum SSA
  - Many extraneous $\emptyset$-functions are inserted
  - Need better algorithm to insert $\emptyset$-functions only when needed

(1) Insert $\emptyset$-functions
   for every basic block $b_i$ that has multiple predecessors
   for each variable $y$ used in $b_i$
     insert $\emptyset$-function $y = \emptyset(y, y, \ldots y)$,
     where each $y$ in $\emptyset$ corresponds to a predecessor

(2) Renaming
   Compute reaching definitions on CFG
   Each variable use has only one reachable definition
   Rename all definitions so that each defines a different name
   Rename all uses of variables according to its definition point
Dominators

- For each basic block \( y \)
  - \( x \) dominates \( y \) (\( x \in \text{Dom}(y) \)) if
    - \( x \) appears on all paths from entry to \( y \)
  - \( x \) strictly dominates \( y \) if
    - \( x \in \text{Dom}(y) \) and \( x \neq y \)
    - i.e. \( x \in \text{Dom}(y) \)-\{\( y \}\}
  - \( x \) immediately dominates \( y \) if
    - \( x \in \text{Dom}(y) \)
    - \( \forall z \in \text{Dom}(y) \), \( z \in \text{Dom}(x) \)
    - Written as \( x = \text{IDom}(y) \)

- Immediate dominators
  - \( \text{IDom}(F) = C \)
  - \( \text{IDom}(G) = A \)
  - \( \text{IDom}(D) = C \)
Where to insert $\emptyset$-functions

For variables defined in basic block $n$, which joint points in CFG need $\emptyset$-functions for them?

- A definition in $n$ forces a $\emptyset$-function just outside the region of CFG that $n$ dominates.
- A $\emptyset$-function must be inserted at each dominance frontier of $n$.

$m \in DF(n)$ iff

1. $n$ dominates a predecessor of $m$.
   \[ \exists q \in \text{preds}(m) \text{ s.t. } n \in \text{Dom}(q) \]
2. $n$ does not strict dominate $m$.
   \[ m \notin \text{Dom}(n) - \{n\} \]
Example: Constructing SSA

```c
void fee(int x, int y)
{
    int I = 0;
    int z = x;
    while (I < 100) {
        I = I + 1;
        if (y < x) z = y;
        A[I] = I;
    }
}
```
Reconstructing Executable Code

- SSA form is not directly executable on machines
  - Must rewrite $\emptyset$-functions into copy instructions
    - Need to split incoming edges of each $\emptyset$-function
    - Need to break cycles in $\emptyset$-function references
  - Rewriting made complex by SSA transformations
    - All phi functions of the same join point need to be evaluated concurrently
Appendix: Very Busy Expressions

- Domain of analysis
  - Set of expressions in a procedure
  - An expression e is very busy at a CFG point p if it is evaluated on every path leaving p, and evaluating e at p yields the same result.
  - At any CFG point p, what expressions are very busy?

- If an expression e is very busy at p, we can evaluate e at p and then remove all future evaluation of e.
  - Code hoisting --- reduces code space, but may lengthen live range of variables

- For each basic block n, let
  - UEExpr(n) = expressions used before any operands being redefined in n
  - ExprKill(n) = expressions whose operands are redefined in n

Goal: evaluate very busy expressions on exit from n
  - VeryBusy(n) = \( \bigcup \{ UEExpr(m) \cap (VeryBusy(m) - ExprKill(m)) \} \)
    \[ m \in \text{succ}(n) \]
Appendix: Constant Propagation

- Domain of analysis
  - Set of variable-value pairs in a procedure
  - A pair (v, c) is valid at a CFG point p if on every path from procedure entry to p, variable v has value c.
  - (v, _): v has undefined value; (v, ⊥): v has unknown value; (v, ci): v has a constant value ci

- If a variable v always has a constant value c at point p, the compiler can replace uses of v at p with c
  - Allows specialization of code based on value cz

- For each basic block n,
  - Evaluate all variable-value pairs valid on entry to n
  \[
  \text{Constants}(n) = \bigwedge \text{Fm}(\text{Constants}(m)) \quad m \in \text{preds}(n)
  \]
  where \(\bigwedge\): pair-wise meet of var-val pairs
  \[
  \text{Fm}(\text{Constants}(m)) : \text{var-val pairs on exit from m}
  \]
Constant Propagation
Local Sets And Meet-over-all-paths

- For each basic block \( n \),
  \[ \text{Constants}(n) = \bigwedge \text{Fm} (\text{Constants}(m)) \]
  where \( \text{Fm}(\text{Constants}(m)) \) is var-val pairs on exit from \( m \)

\[
(v, c_1) \wedge (v, c_2) = \begin{cases} 
  (v, c_1) & \text{if } c_1 = = c_2; \\
  (v, \bot) & \text{otherwise}
\end{cases}
\]

- Compute \( \text{Fm}(\text{input}) \)
  Let \( m = S_1, S_2, ..., S_k \)
  for each \( i = 1, ..., k \)
  If \( S_i \) is \( x := y \)
    Suppose \( (x,c_1), (y,c_2) \in \text{input} \)
    \( \text{input} = (\text{input} - \{(x,c_1)\}) \cap \{(x,c_2)\} \)
  If \( S_i \) is \( y \text{ op } z \)
    Suppose \( (x,c_1), (y,c_2), (z,c_3) \in \text{input} \)
    \( \text{input} = (\text{input} - \{(x,c_1)\}) \cap \{(x,c_2 \text{ op } c_3)\} \)
\[
c_2 \text{ op } c_3 = \begin{cases} 
  \text{Constant} & \text{if } c_2, c_3 \text{ are constants} \\
  \bot & \text{otherwise}
\end{cases}
\]
More On Constant Propagation

- Termination of constant propagation
  - Iterative data-flow algorithms are guaranteed to terminate if the result sets are bounded and grow monotonically.
  - Constant propagation does not have a bounded result set --- the set of all constant values is infinite
  - However, each variable-value pair can be updated at most twice. So constant propagation is guaranteed to terminate

- Using constant propagation to specialize code
  - Constant folding: evaluate integer expressions at compile time instead of runtime
  - Eliminate unreachable code: if a conditional test is always false, the entire branch can be removed
  - Enable more precision in other program analysis. E.g., knowing the bounds of loops can eliminate superfluous reordering constraints.
Appendix: Computing Dominators

- **Domain of analysis**
  - Set of basic blocks in a procedure
  - A basic block \( x \) dominates basic block \( y \) in CFG if \( x \) appears on all paths from entry to \( y \)
  - At any CFG node \( y \), what basic blocks dominate \( y \)?

- **For each basic block \( n \)**
  - \( \text{Dom}(n) = \{n\} \cup (\cap \text{Dom}(m)) \) \( m \in \text{preds}(n) \)
  - \( \text{IDom}(n) \) = the block in \( \text{Dom}(n) \) with smallest RPO sequence number
    - Each basic block \( n \) has a single \( \text{IDom}(n) \)
    - Can use \( \text{IDom} \) relation to build a dominator tree
Computing Dominance Frontiers

for each CFG node \( n \)
\[
\text{DF}(n) = \emptyset
\]
for each CFG node \( n \)
if \( n \) has multiple predecessors
for each predecessor \( p \) of \( n \)
runner := \( p \)
while runner ≠ \text{IDom}(n)
\[
\text{DF}(\text{runner}) := \text{DF}(\text{runner}) \cup \{n\}
\]
runner := \text{IDom}(runner)

Dominance tree:
Inserting $\emptyset$-Functions (skip)

Finding global names:

<table>
<thead>
<tr>
<th>Globals := $\emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each variable $x$</td>
</tr>
<tr>
<td>Blocks$(x)$ := $\emptyset$</td>
</tr>
<tr>
<td>for each block $b_i$: S1, S2, ..., Sk</td>
</tr>
<tr>
<td>VarKill := $\emptyset$</td>
</tr>
<tr>
<td>for $j = 1$ to $k$</td>
</tr>
<tr>
<td>let $S_j$ be $x := y \text{ op } z$</td>
</tr>
<tr>
<td>if $y \notin$ VarKill then</td>
</tr>
<tr>
<td>Globals :=Globals $\cup$ ${y}$</td>
</tr>
<tr>
<td>if $z \notin$ VarKill then</td>
</tr>
<tr>
<td>Globals :=Globals $\cup$ ${z}$</td>
</tr>
<tr>
<td>VarKill := VarKill $\cup$ ${x}$</td>
</tr>
<tr>
<td>Blocks$(x)$ := Blocks$(x)$ $\cup$ ${b}$</td>
</tr>
</tbody>
</table>

Inserting $\emptyset$-functions:

for each name $x \in$ Globals

| WorkList := Blocks$(x)$ |
| for each block $b \in$ WorkList |
| for each block $d$ in DF$(b)$ |
| insert a $\emptyset$-function for $x$ in $d$ |
| WorkList := WorkList $\cup$ $\{d\}$ |
Renaming After $\emptyset$-Insertion(skip)

Main

for each name $x \in \text{Globals}$
  counter[$x$] := 0
  stack[$x$] := 0
 Rename (n0)

Rename (n0)

for each $x := \emptyset(...) \in \text{bi}$
  rename $x$ as NewName($x$)

for each operation $x := y \ op \ z$ in $\text{bi}$
  rewrite $y$ as top(stack[$y$])
  rewrite $z$ as top(stack[$z$])
  rewrite $x$ as NewName($x$)

for each $m \in \text{succ}(\text{bi})$
  fill in $\emptyset$-function parameters in $m$

for each $n$ such that $\text{bi} = \text{IDom}(n)$
  Rename($n$)

for each operation $x := y \ op \ z$ in $\text{bi}$ and each $x := \emptyset(...) \in \text{bi}$
  pop(stack[$x$])

Create new name:

NewName($x$)

  $i := \text{counter}[x]$
  counter[$x$] := counter[$x$] + 1
  push $x_i$ onto stack[$x$]
 return $x_i$

Recursive renaming:

Rename($\text{bi}$)

  for each $x := \emptyset(...) \in \text{bi}$
    rename $x$ as NewName($x$)

  for each operation $x := y \ op \ z$ in $\text{bi}$
    rewrite $y$ as top(stack[$y$])
    rewrite $z$ as top(stack[$z$])
    rewrite $x$ as NewName($x$)

  for each $m \in \text{succ}(\text{bi})$
    fill in $\emptyset$-function parameters in $m$

  for each $n$ such that $\text{bi} = \text{IDom}(n)$
    Rename($n$)

  for each operation $x := y \ op \ z$ in $\text{bi}$ and each $x := \emptyset(...) \in \text{bi}$
    pop(stack[$x$])