Classical Dataflow Analysis

Dr. Barbara G. Ryder
Cs 6304
Fall 2015
Lecture 1 – Outline

• Classical Dataflow Analysis
  • Control flow graphs, Reaching definitions, Live uses of variables, Available Expressions
  • Dataflow equations (transfer functions)
  • References: optimization chapter of compiler textbooks
Compilation Process

Optimization is a semantics-preserving transformation
Static (compile-time) Analysis

• Semantic analysis of code to ensure correctness of machine independent optimization
  - Optimizing Fortran compiler - IBM Backus late 1960’s
• Classical dataflow problems defined on Fortran serve as simple examples of defining and solving dataflow problems
• Assume knowledge of internal program representations of code
  – Rooted, digraphs: control flow graph (of a function), call graph (program calling structure)
sum = 0

```
do 10 i = 1, n
10  sum = sum + a(i) * a(i)  ! original Fortran
```

1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.

sum = 0; initialize loop counter
loop test, check for limit
a[i]

a[i]
a[i] * a[i]
increment sum
increment loop counter
Control Flow Graph (CFG)

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. T
Optimized Control Flow Graph (CFG)

Optimizations enabled by dataflow analysis extracting info about reads and writes - data dependences

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
Some Classical Data Flow Problems

• Reaching definitions, Live uses of variables, Available expressions, used historically for low-level code optimizations
• Def-use and use-def chains, built from Reach and Live provide semantic basis for data dependence analysis
• Available expressions enable common subexpression elimination
Reaching Definitions

• **Definition** A statement which may change the value of a variable

• A definition of a variable $x$ at node $k$ reaches node $n$ if there is a definition-clear path from $k$ to $n$.

![Diagram showing the concept of reaching definitions]

$x = \ldots$

$n \ldots = x$
Live Uses of Variables

• **Use** Appearance of a variable as an operand of a 3 address statement

• A use of a variable $x$ at node $n$ is *live on exit* from node $k$ if there is a definition-clear path for $x$ from $k$ to $n$. 

![Diagram of live uses of variables](image)
Def-use Relations

- Use-def chain links an use to a definition that reaches that use →
- Def-use chain links a definition to an use that it reaches →

\[
x = \ldots
\]

\[
\ldots = x
\]
Constant Propagation

\[ = 5 \times i + 3 \]

\[ = i \times 2 \]

\[ = 5 \times i + 3 = 8 \]

same constant
different constants
Reaching Definitions Equations

Reach(j) = \bigcup \{ Reach(m) \cap \text{pres}(m) \cup \text{dgen}(m) \} \quad m \in \text{Pred}(j)

where:

- \text{pres}(m) is the set of defs preserved through node \( m \)
- \text{dgen}(m) is the set of defs generated at node \( m \)
- \text{Pred}(j) is the set of immediate predecessors of node \( j \)
Live Uses Equations

\[ \text{Live}(j) = \bigcup \{ \text{Live}(m) \cap \text{pres}(m) \cup \text{ugen}(m) \} \]

where \( m \in \text{Succ}(j) \)

- \( \text{pres}(m) \) is the set of uses preserved through node \( m \) (these will correspond to variables whose defs are preserved)
- \( \text{ugen}(m) \) is the set of uses generated at node \( m \)
- \( \text{succ}(j) \) is the set of immediate successors of node \( j \)
Available Expressions

• An expression $X \text{ op } Y$ is available at program point $n$ if EVERY path from program entry to $n$ evaluates $X \text{ op } Y$, and after every evaluation prior to reaching $n$, there are NO subsequent assignments to $X$ or $Y$. 
Global Common Subexpressions
Available Expressions Equations

\[ \text{Avail}(j) = \bigcap \{ \text{Avail}(m) \cap \text{epres}(m) \cup \text{egen}(m) \} \]
\[ m \in \text{Pred}(j) \]

where:
- \( \text{epres}(m) \) is the set of expressions preserved through node \( m \)
- \( \text{egen}(m) \) is the set of (downwards exposed) expressions generated at node \( m \)
- \( \text{pred}(j) \) is the set of immediate predecessors of node \( j \)
### Classical Dataflow Problems

<table>
<thead>
<tr>
<th></th>
<th>May Problems</th>
<th>Must Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward Problems</strong></td>
<td>Reaching Defs</td>
<td>Available Exprs</td>
</tr>
<tr>
<td><strong>Backward Problems</strong></td>
<td>Live Uses of Variables</td>
<td>Very Busy Expressions</td>
</tr>
</tbody>
</table>
Dominators and Natural Loops

- A **dominator** of a node \( x \) in a rooted digraph is a node \( y \) such that all paths from the root to \( x \) must pass through \( y \).
- A node \( x \) can have many dominators. There is one dominator \( y \) such that there are no other dominators on a path from \( y \) to \( x \). Then \( y \) is \( x \)'s immediate dominators.
- Dominators and spanning trees can define natural loops on a rooted digraph.
How to find the loops on this graph?
node 1 dominates node 7

Example

Dominator Tree
Example

Dominator Tree
Example

Loops
(10,7): \{7,8,10\}
(7,4): \{4,5,6,7,8,10\}
(4,3)(8,3): \{3,4,5,6,7,8,10\}
(9,1):
\{1,2,3,4,5,6,7,8,9,10\}
How to find dominators of CFG, \( G=(N,E,\rho) \)? Use fixed point iteration (justification later)

\[
D(\rho) = \{\rho\}
\]

for \( n \in N-\{\rho\} \) do

\[
D(n) = N;
\]

while changes to any \( D(n) \) occur do

for \( n \in N-\{\rho\} \) do

\[
D(n) = \{n\} \cup \bigcap D(p)
\]

\( p \in \text{pred}(n) \)
Dominators

- Algorithm terminates since at every step some set $D(k)$ becomes smaller; this cannot occur indefinitely, so loop terminates.
- Invariant: Node $k$ is parent of node $n$ in the dominator tree, if node $k$ is the immediate dominator of $n$. 
