COMP 520 - Compilers

Lecture 19 (April 18 - 23)

**Dataflow Analysis**

- 4/18 Please pick up from back of class
  - WA4 (single sheet) on register allocation - due Tue 4/23 at start of class.
- 4/23 Please try exercise available on class web site
  - Exercise will not be graded, but will be reviewed in class Thu 4/25
Announcements Thu Apr 18

• Final project submission due Fri April 26
  – Accepted through Sun Apr 28
  – No new functionality
    • get credit for functionality missing previously
    • possibility for extra credit from additional capabilities
  – See handout on the web

• Short written assignment due Tue April 23 start of class
  – WA4 register allocation for expressions
Data Flow Analysis

- Topics
  - Determining properties of programs with multiple execution paths
  - Control flow graphs
  - Dataflow equations and their solution
  - Dataflow frameworks
  - Sample applications
Analysis of programs

• Determine properties of programs
  – true for all possible executions
  – irrespective of input values
  – of use in program analysis and code optimization
  – … but an undecidable problem, in general

• Example
  – “dead code” elimination
    • find unused computations
      – introduced by programmer or compiler
    • delete them from program
      – without changing program meaning
    • which statements are “dead” and can be removed?
      – a statement is “dead” if it performs an assignment that can not influence the value of any variable at termination

Program

(1) x = y + 1;
(2) y = 2 * z;
(3) x = y + z;
(4) z = 1;
(5) z = x;
Analysis of programs

- Determine properties of programs
  - true for all possible executions
  - irrespective of input values
  - of use in program analysis and code optimization
  - … but an undecidable problem, in general

- Example
  - “dead code” elimination
    - find unused computations
      - introduced by programmer or compiler
    - delete them from program
      - without changing program meaning
    - which statements are “dead” and can be removed?
      - a statement is “dead” if it performs an assignment that can not influence the value of any variable at termination

Program

1. \( x = y + 1; \)
2. \( y = 2 \times z; \)
3. \( x = y + z; \)
4. \( z = 1; \)
5. \( z = x; \)
Analysis in presence of control structures

• Add alternative construct to example
  – suppose we don’t know the value of d (e.g. it is an input)
  – is statement (4) dead?
    • there are no possible executions where this value of z could be used
  – is statement (1) dead?
    • in some possible execution (where d == false), its final value of x may be used later

Program
(1) \( x = y + 1; \)
(2) \( y = 2 * z; \)
(3) if (d) \( x = y + z; \)
(4) \( z = 1; \)
(5) \( z = x; \)

Program
(1) \( x = y + 1; \)
(2) \( y = 2 * z; \)
(3) if (d) \( x = y + z; \)
(4) \( z = 1; \)
(5) \( z = x; \)
Analysis in presence of control structures

- Add repetitive control construct to example
  - we don’t know the value of c or d or z
  - is statement (2) dead?
    - No, in some execution, the value of x may be used in line 7
  - is statement (5) dead?
    - No, in some executions, value from z = 1 may be used in the next iteration!

Program
1. while (c) {
2.  x = y + 1;
3.  y = 2 * z;
4.  if (d)  x = y + z;
5.  z = 1;
6. }
7. z = x;

Program
1. while (c) {
2.  x = y + 1;
3.  y = 2 * z;
4.  if (d)  x = y + z;
5.  z = 1;
6. }
7. z = x;
Control flow and optimization

• Optimization requires analysis
  – dead code elimination: need to know if values are possibly used in a program

• Required information
  – not explicit in the program
  – must be computed statically (i.e. at compile time)
  – must consider all potential run-time executions

• Control flow complicates analysis
  – different executions may follow different paths through the program
Control Flow Graphs

- **Control Flow Graph (CFG)**
  - directed graph representation of computation and control flow in a program
  - framework for static analysis of programs

- Control constructs are reduced to (conditional) jumps
  - like flow charts

- **Nodes are basic blocks of tuple code operations**
  - straight line tuple code
    - no jumps except possibly in the last tuple in the block
    - no tuple in a basic block is the target of any jump program-wide
      - except possibly the first tuple in the block

- Directed edges represent possible flow of control from one block to others
  - there may be multiple incoming/outgoing edges for each block
**CFG Example**

**Program**

\[ x = x - 2; \]
\[ y = 2 \times z; \]
\[ \text{if (c) \{ } \]
\[ \quad x = x + 1; \]
\[ \quad y = y + 1; \]
\[ \} \]
\[ \text{else \{ } \]
\[ \quad x = x - 1; \]
\[ \quad y = y - 1; \]
\[ \} \]
\[ z = x + y; \]

**Control Flow Graph**

- **B₁**: \[ x = x - 2; \]
  \[ y = 2 \times z; \]
  \[ \text{if (c)} \]

- **T**
  \[ B₂ : x = x + 1; \]
  \[ y = y + 1; \]

- **F**
  \[ B₃ : x = x - 1; \]
  \[ y = y - 1; \]

- **B₄**: \[ z = x + y; \]
Basic Blocks

- Sequence of consecutive statements such that
  - control enters only at the beginning of the sequence
    - control may come from any of the predecessor blocks
  - control leaves only at the end of the sequence
    - control may transfer to any of the successor blocks
  - no branching into or out of the middle of basic blocks!
    - easy to insure in modern “structured” languages

\[
\begin{align*}
x &= x - 2; \\
y &= 2 \times z; \\
\text{if} \ (c)
\end{align*}
\]
CFG models all potential program executions

- Potential execution
  - path in graph
    - from start block (indegree 0)
    - to end block (outdegree 0)
  - possible paths
    - \( B_1 \ B_2 \ B_4 \)
    - \( B_1 \ B_3 \ B_4 \)
  - some executions may be infeasible
    - why?

\[
\begin{align*}
B_1 & : \ x = x - 2; \\
      & \quad y = 2 \times z; \\
      & \quad \text{if (c)} \\
B_2 & : \ x = x + 1; \\
      & \quad y = y + 1; \\
B_3 & : \ x = x - 1; \\
      & \quad y = y - 1; \\
B_4 & : \ z = x + y;
\end{align*}
\]
CFG Example 2

Program
(1) while (c) {
(2)   x = y + 1;
(3)   y = 2 * z;
(4)   if (d)  x = y + z;
(5)   z = 1;
(6)   }
(7)   z = x;
Building the CFG

- Construct CFG by traversal of the AST
  - each statement type generates one or more nodes
  - nodes with straight line control flow can be merged

- Level of the CFG
  - high level, e.g. statements with arbitrary expressions
  - low level
    - tuple code

- Low level view is most useful for many optimizations
  - register allocation
  - common subexpression elimination

- What about functions and procedures?
  - a set of CFGs, one for each function/procedure
  - *global* dataflow analysis: interprocedural data flow analysis
**Dataflow Analysis on CFG**

- **Live variable analysis**
  - Variable \( v \) is *live* at a program point \( i \) if there is a path from \( i \) to a *use* of \( v \)
  - There is a program point at the start and end of every line of the tuple code
  - What are the live variables at each program point?

- **Method**
  - Let \( L_i \) be the set of variables live at program point \( i \)
  - Define a rule that relates \( L_i \) to \( L_{i+1} \)
    - \( L_i \) may determine \( L_{i+1} \) or vice versa
Derive rules for computing $L_i$

- **rule for a statement $S$**
  
  $$L_i = (L_{i+1} - \text{def}[S]) \cup \text{use}[S]$$
  
  $v$ is live at program point $i$ if
  - $v$ is live at $i+1$ and is not defined by $S$
  - $v$ is used in $S$

- **rule for a basic block $B$**
  
  $$L_{\text{out}(B)} = \bigcup_{B' \in \text{succ}(B)} L_{\text{in}(B')}$$

- **Examples**
  - **statement** “$y = 2 \times z$”
    - $L_4 = (L_5 - \{y\}) \cup \{z\}$
  - **basic block**
    - $L_6 = L_7 \cup L_9$
Simplify the rules for the given problem

\[
L_1 = L_2 \cup \{c\}
\]
\[
L_2 = L_3 \cup L_{11}
\]
\[
L_3 = (L_4 - \{x\}) \cup \{y\}
\]
\[
L_4 = (L_5 - \{y\}) \cup \{z\}
\]
\[
L_5 = L_6 \cup \{d\}
\]
\[
L_6 = L_7 \cup L_9
\]
\[
L_7 = (L_8 - \{x\}) \cup \{y,z\}
\]
\[
L_8 = L_9
\]
\[
L_9 = (L_{10} - \{z\})
\]
\[
L_{10} = L_1
\]
\[
L_{11} = (L_{12} - \{z\}) \cup \{x\}
\]
\[
L_{12} =
\]

```
if (c)
  x = y + 1;
  y = 2 * z;
  if (d)
    x = y + z;
    T
    L1
    T
    L3
    F
    L4
    F
    L5
    L6
    L7
    L8
    L9
    L10
  F
  L2
  L11
  L12
```
Solving dataflow equations

• A set of dataflow equations F has a unique solution if
  – the domain D of the equations has a partial order ⊆ with a least element and greatest element
  – all equations in F are monotonic
    • If \( X \subseteq Y \) then \( F(X) \subseteq F(Y) \) meaning for each \( f \in F \) we have \( f(X) \subseteq f(Y) \)
  – all chains \( X_1 \subseteq X_2 \subseteq \ldots \) in D are finite and have a least upper bound

• For live variables problem
  – domain D = all subsets of \( \{t_1, \ldots, t_n\} \) where \( t_1, \ldots, t_n \) are the program variables
  – the partial order is the subset relation
    – Least element = \( \{\} \subseteq \{t_1\} \subseteq \{t_1, t_2\} \subseteq \ldots \subseteq \{t_1, \ldots, t_n\} = \) greatest element
    – check that the equations are monotonic
      • \( L_i = (L_{i+1} - \text{def}[S]) \cup \text{use}[S] \)  \( F(X) = (X - \text{def}[S]) \cup \text{use}[S] \)
      • \( L_{\text{out}(B)} = \bigcup_{B' \in \text{succ}(B)} L_{\text{in}(B')} \)  \( F(X_1, \ldots, X_k) = X_1 \cup \ldots \cup X_k \)
  – D is finite so all chains have a L.U.B.
Solving dataflow equations

- **Algorithm**
  - Initialize value at each program point to the least element of $D$.
  - Iteratively re-evaluate rules (in any order) until a fixpoint for all program points is reached.

- **The algorithm must terminate**
  - because every chain has a least upper bound.
    - but some evaluation orders terminate faster than others.

- **The solution $S$ satisfies $F(S) = S$ for every rule**
  - It is also guaranteed to be the least solution.
    - for any other solution $S'$ we can prove $S \subseteq S'$. 

Solution: Initialization

\[ L_1 = L_2 \cup \{c\} \]
\[ L_2 = L_3 \cup L_{11} \]
\[ L_3 = (L_4 - \{x\}) \cup \{y\} \]
\[ L_4 = (L_5 - \{y\}) \cup \{z\} \]
\[ L_5 = L_6 \cup \{d\} \]
\[ L_6 = L_7 \cup L_9 \]
\[ L_7 = (L_8 - \{x\}) \cup \{y,z\} \]
\[ L_8 = L_9 \]
\[ L_9 = (L_{10} - \{z\}) \]
\[ L_{10} = L_1 \]
\[ L_{11} = (L_{12} - \{z\}) \cup \{x\} \]
\[ L_{12} = \]

\[ L_1 = \{\} \]
\[ L_2 = \{\} \]
\[ L_3 = \{\} \]
\[ L_4 = \{\} \]
\[ L_5 = \{\} \]
\[ L_6 = \{\} \]
\[ L_7 = \{\} \]
\[ L_8 = \{\} \]
\[ L_9 = \{\} \]
\[ L_{10} = \{\} \]
\[ L_{11} = \{\} \]
\[ L_{12} = \{\} \]
Iteration 1

$L_1 = L_2 \cup \{c\}$
$L_2 = L_3 \cup L_{11}$
$L_3 = (L_4 - \{x\}) \cup \{y\}$
$L_4 = (L_5 - \{y\}) \cup \{z\}$
$L_5 = L_6 \cup \{d\}$
$L_6 = L_7 \cup L_9$
$L_7 = (L_8 - \{x\}) \cup \{y, z\}$
$L_8 = L_9$
$L_9 = (L_{10} - \{z\})$
$L_{10} = L_1$
$L_{11} = (L_{12} - \{z\}) \cup \{x\}$
$L_{12} = \{}$

$L_1 = \{x, y, z, c, d\}$
$L_2 = \{x, y, z, d\}$
$L_3 = \{y, z, d\}$
$L_4 = \{z, d\}$
$L_5 = \{y, z, d\}$
$L_6 = \{y, z\}$
$L_7 = \{y, z\}$
$L_8 = \{}$
$L_9 = \{}$
$L_{10} = \{}$
$L_{11} = \{x\}$
$L_{12} = \{}$
Iteration 2

L_1 = L_2 U \{c\}
L_2 = L_3 U L_{11}
L_3 = (L_4 - \{x\}) U \{y\}
L_4 = (L_5 - \{y\}) U \{z\}
L_5 = L_6 U \{d\}
L_6 = L_7 U L_9
L_7 = (L_8 - \{x\}) U \{y,z\}
L_8 = L_9
L_9 = (L_{10} - \{z\})
L_{10} = L_1

L_{11} = (L_{12} - \{z\}) U \{x\}
L_{12} =

if (c)
  x = y + 1;
y = 2 * z;
if (d)
  x = y + z;

L_1 = \{x,y,z,c,d\}
L_2 = \{x,y,z,c,d\}
L_3 = \{y,z,c,d\}
L_4 = \{x,z,c,d\}
L_5 = \{x,y,z,c,d\}
L_6 = \{x,y,z,c,d\}
L_7 = \{y,z,c,d\}
L_8 = \{x,y,c,d\}
L_9 = \{x,y,c,d\}
L_{10} = \{x,y,z,c,d\}
L_{11} = \{x\}
L_{12} = \{\}
Itaration 3: Fixpoint!

\[ L_1 = L_2 \cup \{c\} \]
\[ L_2 = L_3 \cup L_{11} \]

\[ L_3 = (L_4 - \{x\}) \cup \{y\} \]
\[ L_4 = (L_5 - \{y\}) \cup \{z\} \]
\[ L_5 = L_6 \cup \{d\} \]
\[ L_6 = L_7 \cup L_9 \]

\[ L_7 = (L_8 - \{x\}) \cup \{y,z\} \]
\[ L_8 = L_9 \]
\[ L_9 = (L_{10} - \{z\}) \]
\[ L_{10} = L_1 \]

\[ L_{11} = (L_{12} - \{z\}) \cup \{x\} \]
\[ L_{12} = \]

\[ x = y + 1; \]
\[ y = 2 \times z; \]
\[ \text{if (d)} \]

\[ z = 1; \]

\[ x = y + z; \]
\[ \text{if (c)} \]

\[ z = x; \]
\[ T \quad F \]

\[ L_1 = \{x,y,z,c,d\} \]
\[ L_2 = \{x,y,z,c,d\} \]
\[ L_3 = \{y,z,c,d\} \]
\[ L_4 = \{x,y,z,c,d\} \]
\[ L_5 = \{x,y,z,c,d\} \]
\[ L_6 = \{x,y,z,c,d\} \]
\[ L_7 = \{y,z,c,d\} \]
\[ L_8 = \{x,y,c,d\} \]
\[ L_9 = \{x,y,c,d\} \]
\[ L_{10} = \{x,y,z,c,d\} \]
\[ L_{11} = \{x\} \]
\[ L_{12} = \{\} \]
Generalization

• Live variable analysis and detection of dead code are related
  – An assignment statement \( x = \ldots \) is \textit{dead} if \( x \) is \textit{not live} at the completion of the statement

• Other examples
  – Uninitialized variables
  – Common subexpressions (available expressions)
  – Dynamic type determination

• Data flow analysis framework
  – a common framework for many compiler analyses
  – forward and backward equations (information flow)
  – any path vs all path equations
    • may happen on some execution, must happen on all executions
Applications of dataflow Analysis

1. Global register allocation
   - Solve live variables at all points in a method body
     • Variable $t_i$ is live at program point $j$ if $t_i \in L_j$

   - Construct interference graph $G=(V,E)$
     • $V = \{t_1, \ldots, t_n\}$
     • $(t_i, t_j) \in E$ if $\exists_{1 \leq k \leq n}$ ($t_i \in L_k$ and $t_j \in L_k$)

   - Use graph coloring heuristic algorithm to color graph and assign registers

   - This optimization is performed by most all optimizing compilers and is particularly effective when combined with method inlining
Applications of dataflow Analysis

2. Dynamic class type inference for instance variables
   - What is the domain of values
     • Sequence of program variables with a declared class type
       - \([c_1, \ldots, c_n]\)
     • Possible values for \(c_i\)
       - Suppose \(c_i\) is declared to be of type \(A\)
         and \(A\) has subclasses \(A_1, \ldots, A_k\)
       - Ordering of values for \(c_i\)

   Rules
   - The functions are defined in the \textit{forward} direction to track the dynamic type of program variables with a declared class type
     • \([c_1, \ldots, c_n]\) \texttt{“} \(c_i = \text{new } A_j()\)” \texttt{”} \(\Rightarrow \) \([c_1, \ldots, c_{i-1}, A_j, c_{i+1}, \ldots, c_n]\)
     • \([c_1, \ldots, c_n]\) \texttt{“} \(c_i = c_j\)” \texttt{”} \(\Rightarrow \) \([c_1, \ldots, c_{i-1}, c_j, c_{i+1}, \ldots, c_n]\)

   We need to know the dynamic type along \textit{all} paths reaching a program point so
     • \(\ln_B = \bigcap_{B' \in \text{pred}(B)} \text{out}_{B'}\)
     • \(\bot \cap c = \bot, \quad c \cap d = \text{if } (c == d) \text{ then } c \text{ else } T, \quad c \cap T = T\)
Example: dynamic type inference in Java

- **Types**
  - A, B with B subclass of A
  - foo() is redefined in B

- **Instance variables**
  - a, b

- **Dataflow values**
  - \([\text{dynT}(a), \text{dynT}(b)]\)

- **All path calculation**
  - \(L_7 = L_4 \cap L_6 = [A, B] \cap [B, B] = [T, B]\)

```
A a = new A();
A b = new B();
if (e) a.foo();
b.foo();
```

- **Dynamic call since dynamic type of “a” can vary**
- **Monomorphic call of foo() in class B**