Overview

- context handling
- annotating the AST
  - attribute grammars
  - manual methods
- symbolic interpretation
- data-flow equations

Manual methods for analyzing the AST

- preparing the grounds for code generation
  - constant propagation
  - last-def analysis (reaching definitions)
  - live analysis
  - common subexpression elimination
  - dead-code elimination
- we need flow-of-control information

Threading the AST

- determine the control flow graph that records the successor(s) of AST nodes
- intermediate code
  
  \[
  \begin{align*}
  &\text{PUSH } b \\
  &\text{PUSH } b \\
  &\text{MUL} \\
  &\text{PUSH } 4 \\
  &\text{PUSH } a \\
  &\text{MUL} \\
  &\text{PUSH } c \\
  &\text{MUL} \\
  &\text{SUB} \\
  \end{align*}
  \]

Multiple successors

- problem: threading is built around single Last node pointer
  
  \[
  \begin{align*}
  &\text{IF} \\
  &\text{THEN} \\
  &\text{ELSE} \\
  \end{align*}
  \]
  
  condition

- solution: introduce join node

Threading the AST

- result is a post-order traversal of the AST
- global variable: Last node pointer

PROCEDURE Thread binary expression (Expr node pointer):
  Thread expression (Expr node pointer .left operand);
  Thread expression (Expr node pointer .right operand);
  // link this node to the dynamically last node
  SET Last node pointer .successor TO Expr node pointer;
  // make this node the new dynamically last node
  SET Last node pointer TO Expr node pointer;
Symbolic interpretation

- behaviour of code is determined by (the values of) variables
- simulate run-time execution at compile time
- attach a stack representation to each arrow in the control flow graph
- an entry summarizes all compile-time knowledge about the variable/constant

Symbolic interpretation

IF
THEN
ELSE
FI

Symbolic interpretation

IF
THEN
ELSE
FI

Exercise (5 min.)

- draw the control flow graph for
  while C do S od
  - propagate initial stack
  when C represents y>x
  and S stands for x:=7

Answers

Simple symbolic interpretation

- used in narrow compiler
- simple properties + simple control-flow
- example: detecting the use of uninitialized variables

```c
int foo(int n)
{
    int first;
    while (n-- > 0) {
        if (glob[n] == KEY)
            first = n;
    }
    return first;
}
```
**Tracking uninitialized variables**

- parameter declaration: add (ID:Initialized) tuple
- variable declaration: add (ID:Uninitialized) tuple
- expression: check status of used variables
- assignment: set tuple to (ID:Initialized)

- control statement
  - fork nodes: copy list
  - join nodes: merge lists

  ```
  merge(Init, Init) = Init
  merge(Uninit, Uninit) = Uninit
  merge(x, x) = Maybe
  ```

```c
int foo(int n)
{ int first;
  while (n-- > 0) {
    if (glob[n] == KEY)
      first = n;
  }
  return first;
}
```

**Simple symbolic interpretation**

- flow-of-control structures with one entry and one exit point (no GOTOs)
- the values of the property are severely constrained (see book)
- example: constant propagation

```c
int i = 0;
while (condition) {
  if (i>0) printf("loop reentered\n");
  i++;
  // i == 1
} // i == {0,1}
```

**Constant propagation**

- record the exact value iso Initialized

```c
int i = 0;
while (condition) {
  // i == 0
  if (i>0) printf("loop reentered\n");
  i++;
  // i == 1
} // i == {0,1}
```

- simple symbolic interpretation fails

**Full symbolic interpretation**

- maintain a property list for each entry point (label), initially set to empty
- traverse flow of control graph
  - L: merge current-list into list-L continue with list-L
  - jump L: merge current-list into list-L continue with the empty list
  - repeat until nothing changed

```
// i == 0
// i == 1
// empty
// i == ANY
// i == ANY
// i == ANY
```

**Data flow equations**

- "automated" full symbolic interpretation
- stack replaced by collection of sets
  - IN(N)
  - OUT(N)
- semantics are expressed by constant sets
  - KILL(N)
  - GEN(N)
- equations
  - IN(N) = \bigcup_{M \in \text{predecessor}[N]} OUT(M)
  - OUT(N) = IN(N) \setminus KILL(N) \cup GEN(N)

```c
n = n-1; IN
```

```
OUT
```
**Data flow equations**

- solved through iteration (closure algorithm)
- data-flow nodes may not change the stack
  - individual statements
  - basic blocks
- example: tracking uninitialized variables
  - properties: I x, M x, U x
  - GEN(x = expr.) = {I x}
  - KILL(x = expr.) = {U x, M x}

```
int foo(int n) {
    int first;
    L: n = n - 1;
    if (n >= 0) {
        if (glob[n] == KEY)
            first = n;
        goto L;
    }
    return first;
}
```

**Iterative solution**

- initialization: set all sets to empty
- iterate from top to bottom

```
IN(N) = \bigcup M \in \text{predecessor}[N] \text{ OUT}(M)
OUT(N) = \text{IN}(N) \setminus \text{KILL}(N) \cup \text{GEN}(N)
```

**Efficient data-flow equations**

- limit to (set of) on/off properties
  - set union = bitwise OR
  - set difference = bitwise AND NOT
- combine statements into basic blocks
  - KILL[S₁;S₂] = KILL[S₁] ∪ KILL[S₂]
  - GEN[S₁;S₂] = GEN[S₂] ∪ (GEN[S₁] \ KILL[S₂])
- sort data-flow graph
  - depth-first traversal (break cycles!)

**Live analysis**

- a variable is live at node N if the value it holds is used on some path further down the control-flow graph; otherwise it is dead
- useful information for register allocation
- information must flow "backwards" (up) through the control-flow graph
  - difficult for symbolic interpretation
  - easy for data flow equations

**Solving data-flow equations**

- forwards
  - IN(N) = \bigcup M \in \text{predecessor}[N] \text{ OUT}(M)
  - OUT(N) = \text{IN}(N) \setminus \text{KILL}(N) \cup \text{GEN}(N)

- backwards
  - OUT(N) = \bigcup M \in \text{successor}[N] \text{ IN}(M)
  - IN(N) = \text{OUT}(N) \setminus \text{KILL}(N) \cup \text{GEN}(N)
Exercise (5 min.)

• determine the KILL and GEN sets for the property "V is live here" for the following statements

<table>
<thead>
<tr>
<th>statement S</th>
<th>KILL</th>
<th>GEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v = a ⊕ b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v = M[i]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M[i] = v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f(a_1, ... , a_n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v = f(a_1, ... , a_n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if a&gt;b then goto L_1 else goto L_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exercise (7 min.)

• draw the control-flow graph for the following code fragment

```c
double average(int n, double v[])
{
    int i;
    double sum = 0.0;
    for (i=0; i<n; i++) {
        sum += v[i];
    }
    return sum/n;
}
```

• perform backwards live analysis using the KILL and GEN sets from the previous exercise

Answers

1

2

3

4

summary(N) = \bigcup_{N \in \text{successor}(N)} \text{IN}(M)

\text{IN}(N) = \text{OUT}(N) \setminus \text{KILL}(N) \cup \text{GEN}(N)

statement GENKILL

<table>
<thead>
<tr>
<th>statement</th>
<th>KILL</th>
<th>GEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary

manual methods for annotating the AST

• threading
• symbolic interpretation
• data-flow equations

<table>
<thead>
<tr>
<th>method</th>
<th>granularity</th>
<th>direction</th>
<th>symbolic interpretation</th>
<th>data-flow equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack-based simulation</td>
<td>AST node</td>
<td>one-pass</td>
<td>simple</td>
<td>IN, OUT, GEN, KILL sets</td>
</tr>
<tr>
<td></td>
<td>basic block</td>
<td>iterative</td>
<td>full</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>forwards &amp; backwards</td>
<td>data-flow equations</td>
<td></td>
</tr>
</tbody>
</table>

Homework

• study sections:
  * 3.2.4  interprocedural data-flow analysis
  * 3.2.5.1 live analysis by symbolic interpretation

• assignment 1:
  * replace yacc with LLgen
  * deadline April 9 08:59

• print handout for next week [blackboard]