Static Program Analysis
Part 4 – flow sensitive analyses

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University
Agenda

• Live variables analysis
• Available expressions analysis
• Very busy expressions analysis
• Reaching definitions analysis
• Constant propagation analysis
Liveness analysis

• A variable is *live* at a program point if its current value may be read in the remaining execution.

• This is clearly undecidable, but the property can be conservatively approximated.

• The analysis must only answer “*dead*” if the variable is really dead.
  – no need to store the values of dead variables.
A lattice for liveness

A subset lattice of program variables

```
var x, y, z;
x = input;
while (x>1) {
    y = x/2;
    if (y>3) x = x-y;
    z = x-4;
    if (z>0) x = x/2;
    z = z-1;
}
output x;
```

L = (2\{x,y,z\}, ⊆)

the trivial answer
The control flow graph

\[ z = x - 4 \]
\[ z > 0 \]
\[ z = z - 1 \]
\[ x = x/2 \]
\[ x = x - y \]
\[ y > 3 \]
\[ y = x/2 \]
\[ x > 1 \]
\[ x = \text{input} \]
\[ \text{var } x, y, z \]

output x
Setting up

• For every CFG node, v, we have a variable $\llbracket v \rrbracket$: 
  – the subset of program variables that are live at the program point before v

• Since the analysis is conservative, the computed sets may be too large

• Auxiliary definition:

  $JOIN(v) = \bigcup_{w \in succ(v)} \llbracket w \rrbracket$
Liveness constraints

• For the exit node:
  \[\langle \text{exit} \rangle = \emptyset\]

• For conditions and output:
  \[\langle \text{if } (E) \text{ } \rangle = \langle \text{output } E \rangle = \text{JOIN}(v) \cup vars(E)\]

• For assignments:
  \[\langle x = E \rangle = \text{JOIN}(v) \setminus \{x\} \cup vars(E)\]

• For variable declarations:
  \[\langle \text{var } x_1, \ldots, x_n \rangle = \text{JOIN}(v) \setminus \{x_1, \ldots, x_n\}\]

• For all other nodes:
  \[\langle v \rangle = \text{JOIN}(v)\]

\(vars(E) = \text{variables occurring in } E\)

right-hand sides are monotone since \(\text{JOIN}\) is monotone, and ...

Generated constraints

\[
\text{[var } \ x, y, z] = \text{[z=input]} \setminus \{x, y, z\}
\]
\[
\text{[x=input]} = \text{[x>1]} \setminus \{x\}
\]
\[
\text{[x>1]} = (\text{[y=x/2]} \cup \text{[output } \ x]) \cup \{x\}
\]
\[
\text{[y=x/2]} = (\text{[y>3]} \setminus \{y\}) \cup \{x\}
\]
\[
\text{[y>3]} = \text{[x=x-y]} \cup \text{[z=x-4]} \cup \{y\}
\]
\[
\text{[x=x-y]} = (\text{[z=x-4]} \setminus \{x\}) \cup \{x, y\}
\]
\[
\text{[z=x-4]} = (\text{[z>0]} \setminus \{z\}) \cup \{x\}
\]
\[
\text{[z>0]} = \text{[x=x/2]} \cup \text{[z=z-1]} \cup \{z\}
\]
\[
\text{[x=x/2]} = (\text{[z=z-1]} \setminus \{x\}) \cup \{x\}
\]
\[
\text{[z=z-1]} = (\text{[x>1]} \setminus \{z\}) \cup \{z\}
\]
\[
\text{[output } \ x] = \text{[exit]} \cup \{x\}
\]
\[
\text{[exit]} = \emptyset
\]
Least solution

Many non-trivial answers!

\[
\begin{align*}
[entry] &= \emptyset \\
[var \ x, y, z] &= \emptyset \\
[x=input] &= \emptyset \\
[x>1] &= \{x\} \\
[y=x/2] &= \{x\} \\
[y>3] &= \{x, y\} \\
[x=x-y] &= \{x, y\} \\
[z=x-4] &= \{x\} \\
[z>0] &= \{x, z\} \\
[x=x/2] &= \{x, z\} \\
[z=z-1] &= \{x, z\} \\
[output x] &= \{x\} \\
[exit] &= \emptyset
\end{align*}
\]
Optimizations

• Variables $y$ and $z$ are never simultaneously live
  $\Rightarrow$ they can share the same variable location

• The value assigned in $z = z - 1$ is never read
  $\Rightarrow$ the assignment can be skipped

```plaintext
var x,yz;
x = input;
while (x>1) {
    yz = x/2;
    if (yz>3) x = x-yz;
    yz = x-4;
    if (yz>0) x = x/2;
}
output x;
```

• better register allocation
  • a few clock cycles saved
Time complexity (for the naive algorithm)

• With \( n \) CFG nodes and \( k \) variables:
  – the lattice \( L^n \) has height \( k \cdot n \)
  – so there are at most \( k \cdot n \) iterations

• Subsets of Vars (the variables in the program) can be represented as bitvectors:
  – each element has size \( k \)
  – each \( \cup, \setminus \) = operation takes time \( O(k) \)

• Each iteration uses \( O(n) \) bitvector operations:
  – so each iteration takes time \( O(k \cdot n) \)

• Total time complexity: \( O(k^2n^2) \)

• Exercise: what is the complexity for the worklist algorithm?
Agenda

- Live variables analysis
- **Available expressions analysis**
- Very busy expressions analysis
- Reaching definitions analysis
- Constant propagation analysis
Available expressions analysis

• A (nontrivial) expression is *available* at a program point if its current value has already been computed earlier in the execution

• The approximation generally includes *too few* expressions
  – the analysis can only report “*available*” if the expression is definitely available
  – no need to re-compute available expressions (e.g. common subexpression elimination)
A lattice for available expressions

A reverse subset-lattice of nontrivial expressions

```javascript
var x, y, z, a, b;
z = a + b;
y = a * b;
while (y > a + b) {
    a = a + 1;
    x = a + b;
}
```

\[ L = \left( 2^{\{a+b, a*b, y>a+b, a+1\}}, \subseteq \right) \]
Reverse subset lattice

\[
\emptyset
\]

\[
\begin{align*}
\{a+b\} & \quad \{a*b\} & \quad \{y>a+b\} & \quad \{a+1\} \\
\{a+b, a*b\} & \quad \{a+b, y>a+b\} & \quad \{a+b, a+1\} & \quad \{a*b, y>a+b\} & \quad \{a*b, a+1\} & \quad \{y>a+b, a+1\} \\
\{a+b, a*b, y>a+b\} & \quad \{a*b, y>a+b, a+1\} & \quad \{a+b, y>a+b, a+1\} & \quad \{a*b, y>a+b, a+1\} \\
\{a+b, a*b, y>a+b, a+1\}
\end{align*}
\]

the trivial answer
The flow graph

var x, y, z, a, b

z = a + b

y = a * b

y > a + b

a = a + 1

x = a + b
Setting up

• For every CFG node, v, we have a variable $\left[ v \right]$:
  – the subset of program variables that are available at the program point after v

• Since the analysis is conservative, the computed sets may be too small

• Auxiliary definition:

$$JOIN(v) = \bigcap_{w \in pred(v)} \left[ w \right]$$
Auxiliary functions

• The function $X \downarrow x$ removes all expressions from $X$ that contain a reference to the variable $x$

• The function $\text{exps}(E)$ is defined as:
  – $\text{exps}(\text{intconst}) = \emptyset$
  – $\text{exps}(x) = \emptyset$
  – $\text{exps}(\text{input}) = \emptyset$
  – $\text{exps}(E_1 \text{ op } E_2) = \{E_1 \text{ op } E_2\} \cup \text{exps}(E_1) \cup \text{exps}(E_2)$
    but don’t include expressions containing input
Availability constraints

• For the *entry* node:
  \[
  \llbracket \text{entry} \rrbracket = \emptyset
  \]

• For conditions and output:
  \[
  \llbracket \text{if } (E) \rrbracket = \llbracket \text{output } E \rrbracket = JOIN(v) \cup \text{exps}(E)
  \]

• For assignments:
  \[
  \llbracket x = E \rrbracket = (JOIN(v) \cup \text{exps}(E))\downarrow x
  \]

• For any other node *v*:
  \[
  \llbracket v \rrbracket = JOIN(v)
  \]
Generated constraints

\[
\begin{align*}
\llbracket \text{entry} \rrbracket &= \emptyset \\
\llbracket \text{var } x, y, z, a, b \rrbracket &= \llbracket \text{entry} \rrbracket \\
\llbracket z = a + b \rrbracket &= \text{exps}(a + b) \downarrow z \\
\llbracket y = a \times b \rrbracket &= (\llbracket z = a + b \rrbracket \cup \text{exps}(a \times b)) \downarrow y \\
\llbracket y > a + b \rrbracket &= (\llbracket y = a \times b \rrbracket \cap \llbracket x = a + b \rrbracket) \cup \text{exps}(y > a + b) \\
\llbracket a = a + 1 \rrbracket &= (\llbracket y > a + b \rrbracket \cup \text{exps}(a + 1)) \downarrow a \\
\llbracket x = a + b \rrbracket &= (\llbracket a = a + 1 \rrbracket \cup \text{exps}(a + b)) \downarrow x \\
\llbracket \text{exit} \rrbracket &= \llbracket y > a + b \rrbracket
\end{align*}
\]
Least solution

\[
\begin{align*}
\text{[entry]} & = \emptyset \\
\text{[var } x, y, z, a, b]\text{]} & = \emptyset \\
\text{[z=a+b]} & = \{a+b\} \\
\text{[y=a*b]} & = \{a+b, a*b\} \\
\text{[y>a+b]} & = \{a+b, y>a+b\} \\
\text{[a=a+1]} & = \emptyset \\
\text{[x=a+b]} & = \{a+b\} \\
\text{[exit]} & = \{a+b\}
\end{align*}
\]

Again, many nontrivial answers!
Optimizations

• We notice that $a+b$ is available before the loop
• The program can be optimized (slightly):

```javascript
var x, y, x, a, b, aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
a = a+1;
aplusb = a+b;
x = aplusb;
}
```
Agenda

- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Constant propagation analysis
Very busy expressions analysis

• A (nontrivial) expression is *very busy* if it will definitely be evaluated before its value changes

• The approximation generally includes *too few* expressions
  – the answer “*very busy*” must be the true one
  – very busy expressions may be pre-computed (e.g. loop hoisting)

• Same lattice as for available expressions
Setting up

• For every CFG node, $v$, we have a variable $⟦v⟧$:
  – the subset of program variables that are very busy at the program point before $v$

• Since the analysis is conservative, the computed sets may be too small

• Auxiliary definition:

$$JOIN(v) = \bigcap_{w \in succ(v)} ⟦w⟧$$
Very busy constraints

- For the *exit* node:
  \[ \text{⟦exit⟧} = \emptyset \]

- For conditions and output:
  \[ \text{⟦if } (E) \text{⟧} = \text{⟦output } E \text{⟧} = \text{JOIN}(v) \cup \text{exps}(E) \]

- For assignments:
  \[ \text{⟦x = E⟧} = \text{JOIN}(v) \downarrow x \cup \text{exps}(E) \]

- For all other nodes:
  \[ \text{⟦v⟧} = \text{JOIN}(v) \]
An example program

```javascript
var x, a, b;
x = input;
a = x - 1;
b = x - 2;
while (x > 0) {
    output a*b - x;
    x = x - 1;
}
output a*b;
```

The analysis shows that $a \times b$ is very busy
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x > 0) {
    output a*b-x;
    x = x-1;
}
output a*b;

var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb = a*b;
while (x > 0) {
    output atimesb-x;
    x = x-1;
}
output atimesb;
Agenda

- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Constant propagation analysis
Reaching definitions analysis

• The *reaching definitions* for a program point are those assignments that may define the current values of variables

• The conservative approximation may include *too many* possible assignments
A lattice for reaching definitions

The subset lattice of assignments

$$L = (2\{x=\text{input}, y=x/2, x=x-y, z=x-4, x=x/2, z=z-1\}, \subseteq)$$

```javascript
var x, y, z;
x = input;
while (x > 1) {
y = x/2;
    if (y>3) x = x-y;
    z = x-4;
    if (z>0) x = x/2;
    z = z-1;
}
output x;
```
Reaching definitions constraints

• For assignments:
  \[ \lfloor x = E \rfloor = JOIN(v) \downarrow x \cup \{ x = E \} \]

• For all other nodes:
  \[ \lfloor v \rfloor = JOIN(v) \]

• Auxiliary definition:
  \[ JOIN(v) = \bigcup_{w \in \text{pred}(v)} \lfloor w \rfloor \]

• The function \( X \downarrow x \) removes assignments to \( x \) from \( X \)
Def-use graph

Reaching definitions define the def-use graph:
- like a CFG but with edges from *def* to *use* nodes
- basis for *dead code elimination* and *code motion*

```
x=input
x>1
y=x/2
y>3
x=x-y
x=x/2
z=x-4
z>0
z=z-1
output x
```
Forward vs. backward

• A *forward* analysis:
  – computes information about the *past* behavior
  – examples: available expressions, reaching definitions

• A *backward* analysis:
  – computes information about the *future* behavior
  – examples: liveness, very busy expressions
May vs. must

• A *may* analysis:
  – describes information that is *possibly* true
  – an *over*-approximation
  – examples: liveness, reaching definitions

• A *must* analysis:
  – describes information that is *definitely* true
  – an *under*-approximation
  – examples: available expressions, very busy expressions
## Classifying analyses

<table>
<thead>
<tr>
<th></th>
<th>forward</th>
<th>backward</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>may</strong></td>
<td>example: reaching definitions</td>
<td>example: liveness</td>
</tr>
<tr>
<td></td>
<td>$[[v]]$ describes state after $v$</td>
<td>$[[v]]$ describes state before $v$</td>
</tr>
<tr>
<td></td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} [[w]] = \bigcup_{w \in \text{pred}(v)} [[w]]$</td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} [[w]] = \bigcup_{w \in \text{succ}(v)} [[w]]$</td>
</tr>
<tr>
<td><strong>must</strong></td>
<td>example: available expressions</td>
<td>example: very busy expressions</td>
</tr>
<tr>
<td></td>
<td>$[[v]]$ describes state after $v$</td>
<td>$[[v]]$ describes state before $v$</td>
</tr>
<tr>
<td></td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} [[w]] = \bigcap_{w \in \text{pred}(v)} [[w]]$</td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} [[w]] = \bigcap_{w \in \text{succ}(v)} [[w]]$</td>
</tr>
</tbody>
</table>
Initialized variables analysis

• Compute for each program point those variables that have *definitely* been initialized in the *past*
• (Called *definite assignment* analysis in Java and C#)
• ⇒ *forward must analysis*
• Reverse subset lattice of all variables

\[
JOIN(v) = \bigcap_{w \in \text{pred}(v)} \llbracket w \rrbracket
\]

• For assignments: \( \llbracket x = E \rrbracket = JOIN(v) \cup \{x\} \)
• For all others: \( \llbracket v \rrbracket = JOIN(v) \)
Agenda

- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Constant propagation analysis
var x, y, z;
x = 27;
y = input;
z = 2*x+y;
if (x<0) { y=z-3; } else { y=12 }
output y;

var x, y, z;
x = 27;
y = input;
z = 54+y;
if (0) { y=z-3; } else { y=12 }
output y;

var y;
y = input;
output 12;
Constant propagation analysis

- Determine variables with a constant value
- Flat lattice:
Constraints for constant propagation

• Essentially as for the Sign analysis...

• Abstract operator for addition:

\[ \overline{+}(n,m) = \begin{cases} 
\bot & \text{if } n = \bot \lor m = \bot \\
T & \text{else if } n = T \lor m = T \\
n+m & \text{otherwise}
\end{cases} \]