Static Program Analysis
Part 4 – flow sensitive analyses

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

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

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

```javascript
var x,y,z;
x = 27;
y = input,
z = 2*x+y;
if (x<0) { y=z-3; } else { y=12 }
output y;
```

```javascript
var x,y,z;
x = 27;
y = input;
z = 2*x+y;
if (0) { y=z-3; } else { y=12 }
output y;
```

```javascript
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:

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

- Constant propagation analysis
- **Live variables analysis**
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables 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 powerset lattice of program variables

\[ L = (\mathcal{P}\{x, y, z\}, \subseteq) \]

```
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;
```
The control flow graph

\[ x = \text{input} \]

\[ x > 1 \rightarrow y = x/2 \rightarrow y > 3 \rightarrow x = x - y \]

\[ \text{var } x, y, z \]

\[ z = x - 4 \rightarrow z > 0 \rightarrow x = x/2 \]

\[ z = z - 1 \]

\[ \text{output } x \]
Setting up

• For every CFG node, v, we have a variable \([v]\):
  – the set 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:

\[ \text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} [w] \]
Liveness constraints

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

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

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

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

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

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

right-hand sides are monotone since JOIN is monotone, and ...
Generated constraints

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

Many non-trivial answers!

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

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

• The value assigned in $z = z - 1$ is never read
  ⇒ 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

- Constant propagation analysis
- Live variables analysis
- **Available expressions analysis**
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables 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 powerset lattice of nontrivial expressions

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

$$L = (\mathcal{P}\{a+b, a*b, y>a+b, a+1\}), \supseteq$$
Reverse powerset lattice

∅

- {a+b}
- {a*b}
- {y>a+b}
- {a+1}

{a+b, a*b} {a+b, y>a+b} {a+b, a+1} {a*b, y>a+b} {a*b, a+1} {y>a+b, a+1}

{a+b, a*b, y>a+b} {a+b, a*b, a+1} {a+b, y>a+b, a+1} {a*b, y>a+b, a+1}

{a+b, a*b, y>a+b, a+1}

the trivial answer
The control flow graph

```plaintext
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 $\llbracket v \rrbracket$:
  – the set of expressions that are available at the program point after v

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

• Auxiliary definition:

$$JOIN(v) = \bigcap_{w \in \text{pred}(v)} \llbracket w \rrbracket$$
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 $\text{input}$
Availability constraints

• For the entry node:
  \[ \langle entry \rangle = \emptyset \]

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

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

• For any other node \( v \):
  \[ \langle v \rangle = \text{JOIN}(v) \]
Generated constraints

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

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

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

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- **Very busy expressions analysis**
- Reaching definitions analysis
- Initialized variables 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
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;
Setting up

• For every CFG node, $v$, we have a variable $\llbracket v \rrbracket$:
  – the set of expressions that are very busy at the program point before $v$

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

• Auxiliary definition:

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

• For the exit node:
  \[[exit]\] = \emptyset

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

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

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

- Constant propagation analysis
- Live variables analysis
- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Initialized variables 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 powerset lattice of assignments

\[ L = (\mathcal{P}(\{x=\text{input}, y=x/2, x=x-y, z=x-4, x=x/2, z=z-1\}), \subseteq) \]

```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:
  \[[ x = E ]\] = \( \text{JOIN}(v) \downarrow x \cup \{ x = E \} \)

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

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

• The function \( X \downarrow x \) removes assignments to \( x \) from the set \( X \)
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
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><strong>forward</strong></th>
<th><strong>backward</strong></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>
Agenda

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

• Compute for each program point those variables that have \textit{definitely} been initialized in the past
• (Called \textit{definite assignment} analysis in Java and C#)
• \(\Rightarrow\) \textit{forward must analysis}
• Reverse powerset lattice of all variables

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

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