Static Program Analysis
Part 4 – flow sensitive analyses

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

- 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;
  – available expression may not be re-computed.
A lattice for available expressions

A reverse subset-lattice of nontrivial expressions

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

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

\[
\begin{align*}
\{} & \{} \\
\{a+b\} & \{a*b\} & \{y>a+b\} & \{a+1\} \\
\{a+b, a*b\} & \{a+b, y>a+b\} & \{a+b, a+1\} & \{a*b, y>a+b\} & \{a*b, a+1\} & \{y>a+b, a+1\} \\
\{a+b, a*b, y>a+b\} & \{a+b, a*b, a+1\} & \{a+b, y>a+b, a+1\} & \{a*b, y>a+b, a+1\} \\
\{a+b, a*b, y>a+b, a+1\}
\end{align*}
\]
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 $\llbracket v \rrbracket$:
  – 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:

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

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

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

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

\[ entry \] = \{
\]
\[ var \ x, y, z, a, b \] = \[ entry \]
\[ z = a + b \] = \[ \text{exps}(a + b) \downarrow z \]
\[ y = a \times b \] = (\[ z = a + b \] \cup \[ \text{exps}(a \times b) \]) \downarrow y
\[ y > a + b \] = (\[ y = a \times b \] \cap \[ x = a + b \]) \cup \[ \text{exps}(y > a + b) \]
\[ a = a + 1 \] = (\[ y > a + b \] \cup \[ \text{exps}(a + 1) \]) \downarrow a
\[ x = a + b \] = (\[ a = a + 1 \] \cup \[ \text{exps}(a + b) \]) \downarrow x
\[ exit \] = \[ y > a + b \]
Least solution

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

• 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

• 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 \( \textit{before} \) \( v \)

• Since the analysis is conservative, the computed sets may be \textit{too small}

• Auxiliary definition:

\[
JOIN(v) = \bigcap_{w \in \text{succ}(v)} [w]
\]
Very busy constraints

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

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

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

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

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*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

- 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
The subset lattice of assignments

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

```plaintext
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 \( X \)
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

z=x-4

z>0

x=x-y

x=x/2

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>$\llbracket v \rrbracket$ describes state after $v$</td>
<td>$\llbracket v \rrbracket$ describes state before $v$</td>
</tr>
<tr>
<td></td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket$</td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} \llbracket w \rrbracket = \bigcup_{w \in \text{succ}(v)} \llbracket w \rrbracket$</td>
</tr>
<tr>
<td><strong>must</strong></td>
<td>example: available expressions</td>
<td>example: very busy expressions</td>
</tr>
<tr>
<td></td>
<td>$\llbracket v \rrbracket$ describes state after $v$</td>
<td>$\llbracket v \rrbracket$ describes state before $v$</td>
</tr>
<tr>
<td></td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket = \bigcap_{w \in \text{pred}(v)} \llbracket w \rrbracket$</td>
<td>$\text{JOIN}(v) = \bigcup_{w \in \text{succ}(v)} \llbracket w \rrbracket = \bigcap_{w \in \text{succ}(v)} \llbracket w \rrbracket$</td>
</tr>
</tbody>
</table>
Initialized variables analysis

• Compute for each program point those variables that have *definitely* been initialized in the *past*

• ⇒ *forward must analysis*

• Reverse subset lattice of all variables

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

• \([\text{entry}] = \{\}\]

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

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

- Available expressions analysis
- Very busy expressions analysis
- Reaching definitions analysis
- Constant propagation 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:

$$\oplus(n,m) = \begin{cases} 
\bot & \text{if } n=\bot \lor m=\bot \\
? & \text{else if } n=? \lor m=? \\
n+m & \text{otherwise}
\end{cases}$$