Intra-procedural Data Flow Analysis

Backwards analyses

Hanne Riis Nielson and Flemming Nielson

e-mail: {riis, nielson}@imm.dtu.dk

Informatics and Mathematical Modelling
Technical University of Denmark

© Hanne Riis Nielson & Flemming Nielson
Copyright

This set of transparencies is mainly based on

Flemming Nielson, Hanne Riis Nielson, Chris Hankin:
Principles of Program Analysis.

They may be used by instructors for presentations based on the book but may not be copied and distributed electronically or by other means.

A handout version with four slides per page may be used for producing paper copies of the slides for students; the students may be charged no fee beyond reasonable production cost.

The handout version of the transparencies is available from the webpage
Summary: Forward analyses

\[ x := a \]
\[ A_\ast(\ell) \]
\[ A_\ast(\ell_1) \]
\[ A_\ast(\ell_k) \]

\[ A_\ast(\ell) = (A_\ast(\ell) \setminus \text{kill}_A(B^\ell)) \cup \text{gen}_A(B^\ell) \]

where \( B^\ell \in \text{blocks}(S_\ast) \)

\[ A_\ast(\ell) = \begin{cases} \epsilon_A & \text{if } \ell = \text{init}(S_\ast) \\ \bigcup \{ A_\ast(\ell') \mid (\ell', \ell) \in \text{flow}(S_\ast) \} & \text{otherwise} \end{cases} \]

where

<table>
<thead>
<tr>
<th>( \epsilon_A )</th>
<th>( \cup )</th>
<th>( \cap )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (x, ?) \mid x \in FV(S_\ast) }</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\bigcup | \bigcap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Live Variables Analysis

A variable is live at the exit from a label if there is a path from the label to a use of the variable that does not re-define the variable.

The aim of the Live Variables Analysis is to determine

For each program point, which variables may be live at the exit from the point.

Example

\[
\begin{align*}
&\text{point of interest} \\
&\downarrow \\
&[x:=2]; [y:=4]; [x:=1]; (\text{if } y>x \text{ then } [z:=y] \text{ else } [z:=y*y]); [x:=z]
\end{align*}
\]
The basic idea

\[ x := a \]

\[ LV_e(\ell) \]

Analysis information:

- \( LV_e(\ell) \): the variables that are live at the entry of block \( \ell \)
- \( LV_\ell(\ell) \): the variables that are live at at the exit of block \( \ell \)
- \( kill_{LV}(\ell) \): the variables that are killed by an elementary block
- \( gen_{LV}(\ell) \): the variables that are generated by an elementary block
**Analysis of elementary blocks**

**Definition** of the set of variables that are killed by an elementary block (i.e. they are no longer used in the future) and the set of variables that are generated by the block (they may be used in the future):

\[
\text{kill}_{LV}([x := a]) = \{x\} \\
\text{kill}_{LV}([\text{skip}]) = \emptyset \\
\text{kill}_{LV}([b]) = \emptyset \\
\text{gen}_{LV}([x := a]) = \text{FV}(a) \\
\text{gen}_{LV}([\text{skip}]) = \emptyset \\
\text{gen}_{LV}([b]) = \text{FV}(b)
\]
### Analysis of programs

\[
\begin{align*}
\text{LV}_\circ(\ell) &= \begin{cases} 
\text{Var}_* \\
\bigcup \{ \text{LV}_\circ(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_*) \}
\end{cases} & \text{if } \ell \in \text{final}(S_*) \\
\bigcup \{ \text{LV}_\circ(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_*) \} & \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
\text{LV}_\circ(\ell) &= (\text{LV}_\circ(\ell) \setminus \text{kill}_{\text{LV}}(B^t)) \cup \text{gen}_{\text{LV}}(B^t) \\
&\text{where } B^t \in \text{blocks}(S_*)
\end{align*}
\]
Intra-procedural Data Flow Analysis

**Example**

\[ x := 2 \]
\[ y := 4 \]
\[ x := 1 \]
\[ \text{if } y > x \text{ then } z := y \]
\[ \text{else } z := y \times y \]
\[ x := z \]

**Equations:**

\[ \text{LV}_0(1) = \text{LV}_* (1) \setminus \{ x \} \]
\[ \text{LV}_0(2) = \text{LV}_* (2) \setminus \{ y \} \]
\[ \text{LV}_0(3) = \text{LV}_* (3) \setminus \{ x \} \]
\[ \text{LV}_0(4) = \text{LV}_* (4) \cup \{ x, y \} \]
\[ \text{LV}_0(5) = (\text{LV}_* (5) \setminus \{ z \}) \cup \{ y \} \]
\[ \text{LV}_0(6) = (\text{LV}_* (6) \setminus \{ z \}) \cup \{ y \} \]
\[ \text{LV}_0(7) = \{ x \} \]
\[ \text{LV}_* (1) = \text{LV}_* (2) \]
\[ \text{LV}_* (2) = \text{LV}_* (3) \]
\[ \text{LV}_* (3) = \text{LV}_* (4) \]
\[ \text{LV}_* (4) = \text{LV}_* (5) \cup \text{LV}_* (6) \]
\[ \text{LV}_* (5) = \text{LV}_* (7) \]
\[ \text{LV}_* (6) = \text{LV}_* (7) \]
\[ \text{LV}_* (7) = \emptyset \]

**Analysis of elementary blocks:**

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{kill}_{\text{LV}} (\ell) )</th>
<th>( \text{gen}_{\text{LV}} (\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ x }</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>{ y }</td>
<td>\emptyset</td>
</tr>
<tr>
<td>3</td>
<td>{ x }</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>{ x, y }</td>
</tr>
<tr>
<td>5</td>
<td>{ z }</td>
<td>{ y }</td>
</tr>
<tr>
<td>6</td>
<td>{ z }</td>
<td>{ y }</td>
</tr>
<tr>
<td>7</td>
<td>{ z }</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

**Smallest solution:**

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{LV}_0 (\ell) )</th>
<th>( \text{LV}_* (\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>{ y }</td>
</tr>
<tr>
<td>3</td>
<td>\emptyset</td>
<td>{ x, y }</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>{ y }</td>
</tr>
<tr>
<td>5</td>
<td>{ y }</td>
<td>\emptyset</td>
</tr>
<tr>
<td>6</td>
<td>{ y }</td>
<td>\emptyset</td>
</tr>
<tr>
<td>7</td>
<td>{ z }</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Why smallest solution?

\[ \text{while } [x>1] \text{ do } [\text{skip}] \text{ od; } [x:=x+1] \]

Equations:
\[
\begin{align*}
LV_0(\ell) &= LV_0(\ell) \cup \{x\} \\
LV_0(\ell') &= LV_0(\ell') \\
LV_0(\ell'') &= \{x\} \\
LV_0(\ell) &= LV_0(\ell') \cup LV_0(\ell'') \\
LV_0(\ell') &= LV_0(\ell) \\
LV_0(\ell'') &= \{x\}
\end{align*}
\]

After some calculations: \( LV_0(\ell) = LV_0(\ell) \cup \{x\} \)

Many solutions to this equation: any superset of \( \{x\} \)
Dead Code Elimination

An assignment \( x := a \) is **dead** if the value of \( x \) is not used before it is redefined.

Dead assignments can be eliminated from the program.

**Example:**

\[
\begin{align*}
\end{align*}
\]

Live variables analysis enables a transformation into

\[
\begin{align*}
\end{align*}
\]
Faint variables

Consider the following program:

\[
\begin{align*}
\text{x} & := 1^1; \\
\text{x} & := \text{x} - 1^2; \\
\text{x} & := 2^3
\end{align*}
\]

Clearly \text{x} is dead at the exits from 2 and 3. But \text{x} is live at the exit of 1 even though its only use is to calculate a new value for a variable that turns out to be dead.

We shall say that a variable is a faint variable if it is dead or if it is only used to calculate new values for faint variables; otherwise it is strongly live. In the example \text{x} is faint at the exits from 1 and 2.

Exercise: Define a data flow analysis that detects strongly live variables.
Very Busy Expressions Analysis

An expression is very busy at the exit from a label if, no matter what path is taken from the label, the expression is always used before any of the variables occurring in it are redefined.

The aim of the Very Busy Expressions Analysis is to determine

For each program point, which expressions must be very busy at the exit from the point.

Example

point of interest

\[
\text{if } [a>b] \text{ then } ([x:=b-a]; [y:=a-b]) \text{ else } ([y:=b-a]; [x:=a-b])
\]
The basic idea

Analysis information:

- $\text{VB}_{\times}(\ell)$: the expressions that are very busy at the exit of block $\ell$
- $\text{VB}_0(\ell)$: the expressions that are very busy at the entry of block $\ell$
- $\text{kill}_{\text{VB}}(\ell)$: the expressions that are killed by an elementary block
- $\text{gen}_{\text{VB}}(\ell)$: the expressions that are generated by an elementary block
Analysis of elementary blocks

**Definition** of the expressions that are killed by an elementary block (i.e. they values are no longer guaranteed to be needed in the future) and the assignments that are generated by the block (their values are now guaranteed to be needed in the future):

\[
\begin{align*}
\text{kill}_{VB}(\{x := a\}^\ell) &= \{a' \in \text{AExp}_* \mid x \in FV(a')\} \\
\text{kill}_{VB}(\{\text{skip}\}^\ell) &= 0 \\
\text{kill}_{VB}(\{b\}^\ell) &= 0 \\
\text{gen}_{VB}(\{x := a\}^\ell) &= \text{AExp}(a) \\
\text{gen}_{VB}(\{\text{skip}\}^\ell) &= 0 \\
\text{gen}_{VB}(\{b\}^\ell) &= \text{AExp}(b)
\end{align*}
\]
Analysis of programs

\[
\begin{align*}
\text{VB}_b(\ell) &= \begin{cases} 
\emptyset & \text{if } \ell \in \text{final}(S_*) \\
\bigcap \{ \text{VB}_b(\ell') | (\ell', \ell) \in \text{flow}^R(S_*) \} & \text{otherwise}
\end{cases} \\
\text{VB}_s(\ell) &= (\text{VB}_b(\ell) \setminus \text{kill}_{\text{VB}}(B^\ell)) \cup \text{gen}_{\text{VB}}(B^\ell) \\
\text{where } B^\ell &\in \text{blocks}(S_*)
\end{align*}
\]
Example

if $a > b$ then $(x := b-a ; y := a-b)$ else $(y := b-a ; x := a-b)$

Equations:

\[
\begin{align*}
\text{VB}_1(1) &= \text{VB}_1(1) \\
\text{VB}_2(2) &= \text{VB}_2(2) \cup \{b-a\} \\
\text{VB}_3(3) &= \{a-b\} \\
\text{VB}_4(4) &= \text{VB}_4(4) \cup \{b-a\} \\
\text{VB}_5(5) &= \{a-b\}
\end{align*}
\]

Analysis of elementary blocks:

\[
\begin{array}{c|c|c}
\ell & \text{kill}_\ell(\ell) & \text{gen}_\ell(\ell) \\
\hline
1 & \emptyset & \emptyset \\
2 & \emptyset & \{b-a\} \\
3 & \emptyset & \{a-b\} \\
4 & \emptyset & \{b-a\} \\
5 & \emptyset & \{a-b\}
\end{array}
\]

\[
\begin{array}{c|c|c}
\ell & \text{VB}_\ell(1) & \text{VB}_\ell(2) \\
\hline
1 & \{a-b, b-a\} & \{a-b, b-a\} \\
2 & \{a-b, b-a\} & \{a-b\} \\
3 & \{a-b\} & \emptyset \\
4 & \{a-b, b-a\} & \{a-b\} \\
5 & \{a-b\} & \emptyset
\end{array}
\]

Largest solution:

\[
\begin{array}{c|c|c}
\ell & \text{VB}_\ell(1) & \text{VB}_\ell(2) \\
\hline
1 & \{a-b, b-a\} & \{a-b, b-a\} \\
2 & \{a-b, b-a\} & \{a-b\} \\
3 & \{a-b\} & \emptyset \\
4 & \{a-b, b-a\} & \{a-b\} \\
5 & \{a-b\} & \emptyset
\end{array}
\]
Why largest solution?

while \( [x>1] \) do \([\text{skip}]''\) od; \([x:=x+1]''\)

Equations:

\[
\begin{align*}
\text{VB}_\bullet(\ell) &= \text{VB}_\bullet(\ell) \\
\text{VB}_\bullet(\ell') &= \text{VB}_\bullet(\ell') \\
\text{VB}_\bullet(\ell'') &= \{x+1\} \\
\text{VB}_\bullet(\ell) &= \text{VB}_\bullet(\ell') \cap \text{VB}_\bullet(\ell'') \\
\text{VB}_\bullet(\ell') &= \text{VB}_\bullet(\ell) \\
\text{VB}_\bullet(\ell'') &= \emptyset
\end{align*}
\]

After some simplifications: \( \text{VB}_\bullet(\ell) = \text{VB}_\bullet(\ell) \cap \{x+1\} \)

Two solutions to this equation: \( \{x+1\} \) and \( \emptyset \)
Code Hoisting

Code hoisting finds expressions that are always evaluated following some point in the program regardless of the execution path — and moves them to the latest point beyond which they would always be executed.

Example:

\[
\text{if } [a>b] \text{ then } ([x := b-a]; [y := a-b]) \text{ else } ([y := b-a]; [x := a-b])
\]

The Very Busy Expressions analysis enables a transformation into

\[
[t_1 := b-a]; [t_2 := b-a];
\]

\[
\text{if } [a>b] \text{ then } ([x := t_1]; [y := t_2]) \text{ else } ([y := t_1]; [x := t_2])
\]
The classical analyses revisited

The classical analyses compute information in a set of the form \( \mathcal{P}(D) \) for some finite set \( D \) and the equations have the general form:

\[
A_\circ(\ell) = \begin{cases} 
\ell & \text{if } \ell \in E \\
\bigcup \{ A_\bullet(\ell') \mid (\ell', \ell) \in F \} & \text{otherwise}
\end{cases}
\]

\[
A_\bullet(\ell) = (A_\circ(\ell) \setminus \text{kill}_\ell) \cup \text{gen}_\ell
\]

where \( A_\circ, A_\bullet : \text{Lab} \to \mathcal{P}(D) \) and

- \( F \) is \( \text{flow}(S_\ast) \) or \( \text{flow}^R(S_\ast) \) and \( E \) is \( \{\text{init}(S_\ast)\} \) or \( \text{final}(S_\ast) \),
- \( \bigcup \) is \( \bigcap \) or \( \bigcup \) on \( \mathcal{P}(D) \) and \( \epsilon \subseteq D \) specifies the initial/final information,
- \( \text{kill}_\ell \subseteq D \) and \( \text{gen}_\ell \subseteq D \) specify the information invalidated by/generated by \( B^\ell \in \text{blocks}(S_\ast) \).
## Examples

<table>
<thead>
<tr>
<th>Available Expressions</th>
<th>Reaching Definitions</th>
<th>Very Busy Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}(D)$</td>
<td>$\mathcal{P}(\text{AExp}_s)$</td>
<td>$\mathcal{P}(\text{Var}_s \times \text{Lab}_s)$</td>
<td>$\mathcal{P}(\text{Var}_s)$</td>
</tr>
<tr>
<td>$\cup_i \emptyset$</td>
<td>$\cap\emptyset$</td>
<td>$\cup {(x,?)</td>
<td>x \in \text{FV}(S_s)}$</td>
</tr>
<tr>
<td>$E$</td>
<td>${\text{init}(S_s)}$</td>
<td>${\text{init}(S_s)}$</td>
<td>$\text{final}(S_s)$</td>
</tr>
<tr>
<td>$F$</td>
<td>$\text{flow}(S_s)$</td>
<td>$\text{flow}(S_s)$</td>
<td>$\text{flow}^R(S_s)$</td>
</tr>
</tbody>
</table>

$$f_{\ell}(X) = (X \setminus \text{kill}_\ell) \cup \text{gen}_\ell \text{ where } B^{\ell} \in \text{blocks}(S_s) \text{ (and } X \subseteq D)$$
Forward versus backwards analyses

- The **forward analyses** have $F$ to be $\text{flow}(S_i)$ and then $A_e$ concerns entry conditions and $A_o$ concerns exit conditions.
  - Reaching definitions analysis
  - Available expressions analysis
- The **backward analyses** have $F$ to be $\text{flow}^R(S_i)$ and then $A_o$ concerns exit conditions and $A_e$ concerns entry conditions.
  - Very busy expressions analysis
  - Live variables analysis
May versus must analyses

- When $\bigcup$ is $\bigcup$ we require the smallest sets that solve the equations and we are able to detect properties satisfied by at least one execution path to (or from) the entry (or exit) of a label; the analysis is a may-analysis.
  - Reaching definitions analysis
  - Live variables analysis

- When $\bigcup$ is $\bigcap$ we require the greatest sets that solve the equations and we are able to detect properties satisfied by all execution paths reaching (or leaving) the entry (or exit) of a label; the analysis is a must-analysis.
  - Available expressions analysis
  - Very busy expressions analysis
**Bit vectors**

The classical analyses operate over elements of $\mathcal{P}(D)$ where $D$ is a finite set.

The elements can be represented as bit vectors: Each element of $D$ can be assigned a unique bit position $i$ ($1 \leq i \leq n$). A subset $S$ of $D$ is then represented by a vector of $n$ bits:

- if the $i$'th element of $D$ is in $S$ then the $i$'th bit is 1
- if the $i$'th element of $D$ is not in $S$ then the $i$'th bit is 0

Then we have efficient implementations of

- **set union** as logical or
- **set intersection** as logical and
More bit vector frameworks

- **Upwards exposed uses** determines for a program point, what uses of a variable are reached by a particular definition (assignment)
  — a backwards may analysis
- **Copy analysis** determines whether there on every execution path from a copy statement \( x := y \) to a use of \( x \) there are no assignments to \( y \)
  — a forward must analysis
- **Dominators** determines for each program point which program points are guaranteed to have been executed before the current one is reached
  — a forward must analysis
- **Dual available expressions** determines for each program point which expressions may not be available when execution reaches that point
  — a forward may analysis
Non-bit vector frameworks

- **Constant propagation** determines for each program point whether or not a variable has a constant value whenever execution reaches that point.

- **Detection of signs analysis** determines for each program point the possible signs of the values of the variables may have whenever execution reaches that point.

- **Faint variables** determines for each program point which variables are faint: a variable is faint if it is dead or it is only used to compute new values of faint variables.

- **May be uninitialised** determines for each program point which variables have dubious values: a variable has a dubious value if either it is not initialised or its value depend on variables with dubious values.