Intra-Procedural Dataflow Analysis

Backward Analyses

Markus Schordan

Institut für Computersprachen
Technische Universität Wien

Live Variable 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.

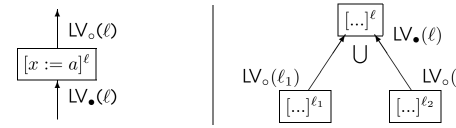
The aim of the Live Variables Analysis is to determine

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

$[y := 0]^0$; $[u := a{+}b]^1$; $[y := a{*}u]^2$; while $[y > u]^3$ do $[a := a + 1]^4$; $[u := a + b]^5$; $[x := u]^6$ od

dead dead

Basic Idea



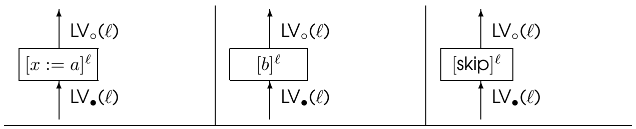
Analysis information: $\mathsf{LV}_\circ(\ell)$,$\mathsf{LV}_\bullet(\ell)$: $\mathsf{Lab}_\star \to \mathcal{P}(\mathsf{Var}_\star)$

- $\mathsf{LV}_\circ(\ell)$: the variables that are live at entry of block ℓ.
- $\mathsf{LV}_\bullet(\ell)$: the variables that are live at exit of block ℓ.

Analysis properties:

- Direction: backward
- May analysis with combination operator $\bigcup$

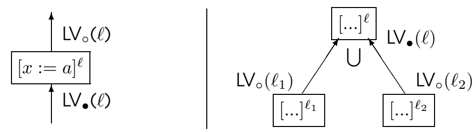
Analysis of Elementary Blocks



$$
\begin{aligned}
\mathsf{kill}_{\mathsf{LV}}([x := a]^\ell) &= \{x\}\\
\mathsf{kill}_{\mathsf{LV}}([\mathsf{skip}]^\ell) &= \emptyset\\
\mathsf{kill}_{\mathsf{LV}}([b]^\ell) &= \emptyset\\
\mathsf{gen}_{\mathsf{LV}}([x := a]^\ell) &= FV(a)\\
\mathsf{gen}_{\mathsf{LV}}([\mathsf{skip}]^\ell) &= \emptyset\\
\mathsf{gen}_{\mathsf{LV}}([b]^\ell) &= FV(b)
\end{aligned}
$$

$$\mathsf{LV}_\circ(\ell) = (\mathsf{LV}_\bullet(\ell) \backslash \mathsf{kill}_{\mathsf{LV}}(B^\ell)) \cup \mathsf{gen}_{\mathsf{LV}}(B^\ell) \quad \text{where } B^\ell \in \mathsf{blocks}(S_\star)$$

Analysis of the Program



$$\mathsf{LV}_\circ(\ell) = (\mathsf{LV}_\bullet(\ell) \backslash \mathsf{kill}_{\mathsf{LV}}(B^\ell)) \cup \mathsf{gen}_{\mathsf{LV}}(B^\ell) \quad \text{where } B^\ell \in \mathsf{blocks}(S_\star)$$

$$\mathsf{LV}_\bullet(\ell) = \begin{cases} \emptyset & : \text{ if } \ell = \mathsf{final}(S_\star) \\ \bigcup\{\mathsf{LV}_\circ(\ell') \mid (\ell', \ell) \in \mathsf{flow}^R(S_\star)\} & : \text{ otherwise} \end{cases}$$

Example

Program	$\mathsf{LV}_\bullet(\ell)$	$\mathsf{LV}_\circ(\ell)$	ℓ	$\mathsf{kill}_{\mathsf{LV}}(\ell)$	ge
$[y := 0]^0$;	{a, b}	{a, b}	0	{y}	∅
$[u := a{+}b]^1$;	{u, a, b}	{a, b}	1	{u}	[illegible]
$[y := a{*}u]^2$;	{u, a, b, y}	{u, a, b}	2	{y}	[illegible]
while$[y > u]^3$do	{a, b, y}	{u, a, b, y}	3	∅	[illegible]
$[a := a + 1]^4$;	{a, b, y}	{a, b, y}	4	{a}	[illegible]
$[u := a + b]^5$;	{u, a, b, y}	{a, b, y}	5	{u}	[illegible]
$[x := u]^6$ od	{u, a, b, y}	{u, a, b, y}	6	{x}	[illegible]
$[\mathsf{skip}]^7$	∅	∅	7	∅	∅

Dead Code Elimination (DCE)

An assignment $[x := a]^\ell$ is dead if the value of x is not used before it is redefined. Dead assignments can be eliminated.

Analysis: Live Variables Analysis

Transformation: For each $[x := a]^\ell$ in $S_\star$ with $x \notin \mathsf{LV}_\bullet(\ell)$ (i.e. dead) eliminate $[x := a]^\ell$ from the program.

Example:

Before:

$[y := 0]^0$; $[u := a{+}b]^1$; $[y := a{*}u]^2$; while $[y > u]^3$ do $[a := a + 1]^4$; $[u := a + b]^5$; $[x := u]^6$ od

After:

$[u := a{+}b]^1$; $[y := a{*}u]^2$; while $[y > u]^3$ do $[a := a + 1]^4$; $[u := a + b]^5$; od

Example: Combining Optimizations

Example:
$[x := a{+}b]^1$; $[y := a{*}x]^2$; while $[y > a{+}b]^3$ do $[a := a + 1]^4$; $[x := a + b]^5$ od

Common Subexpression Elimination gives

$[u := a{+}b]^{1'}$; $[x := u]^1$; $[y := a{*}x]^2$; while $[y > u]^3$ do $[a := a + 1]^4$; $[u := a + b]^{5'}$; $[x := u]^5$ od

Copy Propagation gives

$[u := a{+}b]^{1'}$; $[y := a{*}u]^2$; while $[y > u]^3$ do $[a := a + 1]^4$; $[u := a + b]^{5'}$; $[x := u]^5$ od

Dead Code Elimination gives

$[u := a{+}b]^1$; $[y := a{*}u]^2$; while $[y > u]^3$ do $[a := a + 1]^4$; $[u := a + b]^5$; od

What are the results for other optimization sequences?

Faint Variables

Consider the following program consisting of three statements:

$[x := 1]^1$; $[x := 2]^2$; $[y := x]^3$;

Clearly x is dead at the exit from 1 and y is dead at the exit of 3
is live at the exit of 2 although it is only used to calculate a new
for y that turns out to be dead.

We shall say that a variable is a faint variable if it is dead or if it i
used to calculate new values for faint variables; otherwise it is st
live.

Very Busy Expressions Analsysis

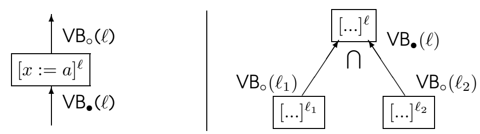
An expression is very busy at the exit from a label if, no matter what path is taken from the label, the expression is always used before any of the variables occurring in it are redefined.

The aim of the Very Busy Expression Analysis is to determine

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

if $[a > b]^1$ then ($[x := b{-}a]^2$; $[y := a{-}b]^3$) else ($[y := b{-}a]^4$; $[x := a{-}b]^5$)

Basic Idea



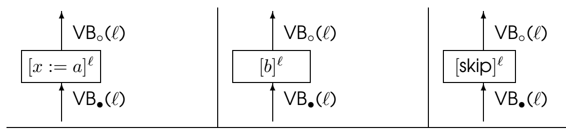
Analysis information: $\mathsf{VB}_\circ(\ell), \mathsf{VB}_\bullet(\ell) : \mathsf{Lab}_\star \to \mathcal{P}(\mathsf{AExp}_\star)$

- $\mathsf{VB}_\circ(\ell)$: the expressions that are very busy at entry of block ℓ.
- $\mathsf{VB}_\bullet(\ell)$: the expressions that are very busy at exit of block ℓ.

Analysis properties:

- Direction: backward
- Must analysis with combination operator $\bigcap$

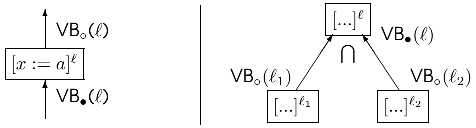
Analysis of Elementary Blocks



$$
\begin{array}{rcl}
\mathsf{kill}_{\mathsf{VB}}([x := a]^\ell) & = & \{a' \in \mathsf{AExp}_\star \mid x \in FV(a')\} \\
\mathsf{kill}_{\mathsf{VB}}([\mathsf{skip}]^\ell) & = & \emptyset \\
\mathsf{kill}_{\mathsf{VB}}([b]^\ell) & = & \emptyset \\
\mathsf{gen}_{\mathsf{VB}}([x := a]^\ell) & = & \mathsf{AExp}(a) \\
\mathsf{gen}_{\mathsf{VB}}([\mathsf{skip}]^\ell) & = & \emptyset \\
\mathsf{gen}_{\mathsf{VB}}([b]^\ell) & = & \mathsf{AExp}(b)
\end{array}
$$

$\mathsf{VB}_\circ(\ell) \;=\; (\mathsf{VB}_\bullet(\ell) \backslash \mathsf{kill}_{\mathsf{VB}}(B^\ell)) \cup \mathsf{gen}_{\mathsf{VB}}(B^\ell)$ where $B^\ell \in$ bloc

Analysis of the Program



$$\mathrm{VB}_\circ(\ell) = (\mathrm{VB}_\bullet(\ell)\backslash \mathrm{kill}_{\mathrm{VB}}(B^\ell)) \cup \mathrm{gen}_{\mathrm{VB}}(B^\ell) \quad \text{where } B^\ell \in \mathrm{blocks}(S_\star)$$

$$\mathrm{VB}_\bullet(\ell) = \begin{cases} \emptyset & : \text{ if } \ell = \mathrm{final}(S_\star) \\ \bigcap\{\mathrm{VB}_\circ(\ell') \mid (\ell', \ell) \in \mathrm{flow}^R(S_\star)\} & : \text{ otherwise} \end{cases}$$

Example

if $[a > b]^1$ then ($[x := b{-}a]^2$; $[y := a{-}b]^3$) else ($[y := b{-}a]^4$; $[x := a{-}b]^5$)

ℓ	VB$_\bullet(\ell)$	VB$_\circ(\ell)$
1	{a−b, b−a}	{a−b, b−a}
2	{a−b}	{a−b, b−a}
3	$\emptyset$	{a−b}
4	{a−b}	{a−b, b−a}
5	$\emptyset$	{a−b}

ℓ	kill$_{\mathrm{VB}}(\ell)$	gen$_{\mathrm{VB}}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	{b−a}
3	$\emptyset$	{a−b}
4	$\emptyset$	{b−a}
5	$\emptyset$	{a−b}

Code Hoisting

Code hoisting finds expressions that are always evaluated follo
some point in the program regardless of the execution path – a
moves them to the earliest point (in execution order) beyond w
they would always be executed.

Before:
if $[a > b]^1$ then ($[x := b{-}a]^2$; $[y := a{-}b]^3$) else ($[y := b{-}a]^4$; [x := a

After:
$[t1 := a{-}b]^0$; $[t2 := b{-}a]^{0'}$;
if $[a > b]^1$ then ($[x := t2]^2$; $[y := t1]^3$) else ($[y := t2]^4$; $[x := t1]^5$)

Summary of Classical Analyses

Analysis	may	must
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Analysis	may	must
Combination Op.	$\cup$	$\cap$
Solution of equ.	smallest	largest

Analysis	Extremal labels set	Abstract flow graph
Forward	$\{\mathrm{init}(S_\star)\}$	$\mathrm{flow}(S_\star)$
Backward	$\mathrm{final}(S_\star)$	$\mathrm{flow}^R(S_\star)$

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 Framework Examples

- Dual available expressions determines for each program po
which expressions may not be available when execution re
that point (forward may analysis)
- Copy analysis determines whether there on every executio
from a copy statement $x := y$ to a use of x there are no
assignments to y (forward must analysis).
- Dominators determines for each program point which prog
points are guaranteed to have been executed before the c
one is reached (forward must analysis).
- Upwards exposed uses determines for a program point, wh
of a variable are reached by a particular definition (assignm
(backward 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 that 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 unitialized determines for each program point which variables have dubious values: a variable has a dubious value if either it is not initialized or its value depends on variables with dubious values.

References

- Material for this 3rd lecture

  `www.complang.tuwien.ac.at/markus/optub.html`
- Book

  Flemming Nielson, Hanne Riis Nielson, Chris Hankin:

  Principles of Program Analysis.

  Springer, (2nd edition, 452 pages, ISBN 3-540-65410-0), 2005.
  - Chapter 2 (Data Flow Analysis)