**Live-Variable Analysis**

For each program point, and each variable approximately compute whether the variable is:

* definitely not *live*: its current value will not be used in the future, or
* it is possibly live: its value may be used in the future

The precise notion of liveness is given by program semantics, as follows.

**Definition:** a variable $x$is **dynamically live** in a concrete program state $(u_i,w_i)$, if there exists a control-flow graph execution   
\begin{displaymath}
   (u_i,w_i),
   s_i,
   (u_{i+1},w_{i+1}),
   s_{i+1},
   \ldots,
   (u_n,w_n),
   s_n,
\end{displaymath}  
(where $u_k$are control-flow graph nodes, $w_k$are values of variables), such that

* the statement $s_n$**reads** the value of $x$(i.e. it is a test mentioning $x$or an assignment statement with $x$on the right-hand side)
* none of the statements $s_i, s_{i+1}, \ldots, s_{n-1}$**writes** to $x$(i.e. is an assignment of the form $x = ...$)

We design a data-flow analysis that, for each program point $u$, computes **static liveness** information for program variables, given by the set of variables $live(u)$.

**Correctness statement for liveness analysis:** if $x \notin live(u)$then for every program execution that reaches a state $(u,w)$, the variable $x$is dynamically live at point $u$.

NOTE: if a static analysis says that the variable is live at program point, the variable may or may not be dynamically live

* but if we know that it is **not** statically live, this is useful information, we know that it is not dynamically alive

Consider a sequence of instructions:

|  |  |
| --- | --- |
| **code** | **live variables** |
|  | {z} |
| x = 42 |  |
|  | {x,z} |
| y = x + 3 |  |
|  | {x,y,z} |
| z = y + z |  |
|  | {x} |
| y = 3 + x |  |
|  | {} |

Applications:

* allocating space for variables (e.g. register allocation for CPU)
  + if variable not used in future, we can store another variable in the same address
* an alternative to initialization analysis: must be initialized if it will be used
  + if variable is used in the future before being assigned, it must be initialized now
  + we can do initialization check by checking that no variable is live at CFG entry

Liveness is naturally computed using **backwards data-flow analysis**

Consider the program execution backwards

* execution is very non-deterministic (e.g. x=3 goes into all values of x)
* mathematically equally well-defined
* introduce an additional state bit to variable, mark it “used” when it is used
* if a state is reached in backward execution where it is used, then it “will be used”

Corresponding backward analysis:

* variable uses flow towards their initializations
* edges in data-flow analysis are interpreted in the opposite way
* analysis starts from the exit point

Final state at the exit point

* no variable is live - no more statements at the end, so no future uses
* this is also the bottom of the lattice

**Pointwise Representation**

For each variable, store its liveness:

1. (potentially) live (top)
2. not live (bottom)

Bottom: map each variable to bottom

Join: pointwise join

* bottom $\sqcup$top = top
* $m_1 \sqcup m_2 = \lambda i. m_1(i) \sqcup m_2(i)$

**An Alternative Representation**

The set of potentially live variables, instead of

* $m : \mbox{Var} \to \{\top, \bot\}$consider
* $m' \subseteq \mbox{Var}$, given by $m' = \{ x \mid m(x)=\top \}$
* this is just a different notation for the same thing

Bottom: empty set

Top: set of all variables

Join: union

**Semilattice for Live-Variable Analysis**

(Semilattice is like [lattice](http://lara.epfl.ch/web2010/sav08:lattices) but need not have meet.)

Elements are sets $S$of live variables

* we assume that variables have been renamed according to scoping rules

Join is union $\cup$

**Transfer Functions for Live-Variable Analysis**

For each statement st in CFG, we introduce sets of variables

* use(st) denote variables used in statement
* def(st) denote variables overwritten in st

For [SimpleCFG.scala](http://lara.epfl.ch/web2010/compilation:simplecfg.scala) we have

|  |  |  |
| --- | --- | --- |
| **st** | **use(st)** | **def(st)** |
| x = y op z | {y,z} | {x} |
| x = y | {y} | {x} |
| Assume[y relOp z] | {y,z} | {} |
| print(y) | {y} |  |

Examples:

|  |  |  |
| --- | --- | --- |
| **st** | **use(st)** | **def(st)** |
| x = y + 1 | {y} | {x} |
| x = x + 1 | {x} | {x} |

In ordinary execution, statement

* first uses variables in use(st) to compute some value
* then assigns this value to variables in def(st)

In backward execution, statement

* marks def(st) as not live
* marks use(st) as live

in that order

Transfer function   
\begin{displaymath}
   transFun(st,S) = (S \setminus d{}ef(st)) \cup use(st)
\end{displaymath}

We have seen **forward analyses**, where for each point $v$, we have:   
\begin{displaymath}
    facts(v) = facts(v) \sqcup \bigsqcup_{e.v2=v} transFun(e.lab, e.v1) \ \ \ (F)
\end{displaymath}  
in backward analysis, we instead have:

\begin{displaymath}
    facts(v) = facts(v) \sqcup \bigsqcup_{e.v1=v} transFun(e.lab, e.v2) \ \ \ (B)
\end{displaymath}

**References**

* [Tiger book](http://lara.epfl.ch/web2010/cc09:tiger_book), Chapter 10, Chapter 17 (page 358)