Data-flow analysis

MOV t32, arg1
MOV t33, arg2
ADD t34, t32, t33
MOV t35, arg3
MOV t36, arg4
ADD t37, t35, t36
MUL res1, t34, t37

Discovering information about how data (i.e. variables and their values) may move through a program.
Motivation

Programs may contain

• code which gets executed but which has no useful effect on the program’s overall result;

• occurrences of variables being used before they are defined; and

• many variables which need to be allocated registers and/or memory locations for compilation.

The concept of variable liveness is useful in dealing with all three of these situations.
Liveness

Liveness is a data-flow property of variables: “Is the value of this variable needed?” (cf. dead code)

```c
int f(int x, int y) {
    int z = x * y;
    :
    :
    :
    ?
    ?
    ?
    ?
    ?
```
Liveness

At each instruction, each variable in the program is either live or dead.

We therefore usually consider liveness from an instruction’s perspective: each instruction (or node of the flowgraph) has an associated set of live variables.

\[ \text{int } z = x * y; \]
\[ \text{return } s + t; \]

\[ \text{live}(n) = \{ s, t, x, y \} \]
Semantic vs. syntactic

There are two kinds of variable liveness:

- Semantic liveness
- Syntactic liveness
A variable $x$ is \textit{semantically} live at a node $n$ if there is some execution sequence starting at $n$ whose (externally observable) behaviour can be affected by changing the value of $x$.

\begin{verbatim}
int x = y * z;  x LIVE
:
return x;
\end{verbatim}
Semantic vs. syntactic

A variable $x$ is *semantically* live at a node $n$ if there is some execution sequence starting at $n$ whose (externally observable) behaviour can be affected by changing the value of $x$.

```c
int x = y * z; // x DEAD
::
x = a + b;
::
::
return x;
```
Semantic vs. syntactic

Semantic liveness is concerned with the execution behaviour of the program.

This is undecidable in general.
(e.g. Control flow may depend upon arithmetic.)
**Semantic vs. syntactic**

A variable is *syntactically* live at a node if there is a path to the exit of the flowgraph along which its value may be used before it is redefined.

Syntactic liveness is concerned with properties of the *syntactic structure* of the program.

Of course, this is decidable.

So what’s the difference?
Semantic vs. syntactic

```c
int t = x * y; t DEAD
if ((x+1)*(x+1) == y) {
    t = 1;
}
if (x*x + 2*x + 1 != y) {
    t = 2;
}
return t;
```

Semantically: one of the conditions will be true, so on every execution path \( t \) is redefined before it is returned. The value assigned by the first instruction is never used.
Semantic vs. syntactic

MUL t, x, y
ADD t32, x, #1
MUL t33, t32, t32
CMPNE t33, y, lab1
MOV t, #1
lab1:
MUL t34, x, x
MUL t35, x, #2
ADD t36, t34, t35
ADD t37, t36, #1
CMPEQ t37, y, lab2
MOV t, #2
lab2:
MOV res1, t
Semantic vs. syntactic

On *this* path through the flowgraph, \( t \) is not redefined before it’s used, so \( t \) is *syntactically* live at the first instruction.

Note that this path never actually occurs during execution.
Semantic vs. syntactic

So, as we’ve seen before, syntactic liveness is a computable approximation of semantic liveness.
Semantic vs. syntactic program variables

- Semantically live at $n$
- Semantically dead at $n$
Semantic vs. syntactic

syntactically live at $n$

imprecision
Semantic vs. syntactic

\[ \text{sem-live}(n) \subseteq \text{syn-live}(n) \]

Using syntactic methods, we safely overestimate liveness.
Live variable analysis

LVA is a *backwards* data-flow analysis: usage information from *future* instructions must be propagated backwards through the program to discover which variables are live.

```c
int f(int x, int y) {
    int z = x * y;
    :
    :
    print z;
    int a = z*2;
    if (z > 5) {
```
Live variable analysis

Variable liveness flows (backwards) through the program in a continuous stream.

Each instruction has an effect on the liveness information as it flows past.
Live variable analysis

An instruction makes a variable live when it *references* (uses) it.
Live variable analysis

\[ a = b \times c; \quad \text{REFERENCE } b, c \]
\[ d = e + 1; \quad \text{REFERENCE } e \]
\[ \text{print } f; \quad \text{REFERENCE } f \]
Live variable analysis

An instruction makes a variable dead when it defines (assigns to) it.
Live variable analysis

\[
\begin{align*}
a &= 7; & \text{DEFINE } a \\
b &= 11; & \text{DEFINE } b \\
c &= 13; & \text{DEFINE } c
\end{align*}
\]
Live variable analysis

We can devise functions \( \text{ref}(n) \) and \( \text{def}(n) \) which give the sets of variables referenced and defined by the instruction at node \( n \).

\[
\begin{align*}
\text{ref}( x = 3 ) &= \{ \} & \text{ref}( \text{print } x ) &= \{ x \} \\
\text{def}( x = 3 ) &= \{ x \} & \text{def}( \text{print } x ) &= \{ \} \\
\text{ref}( x = x + y ) &= \{ x, y \} \\
\text{def}( x = x + y ) &= \{ x \}
\end{align*}
\]
Live variable analysis

As liveness flows backwards past an instruction, we want to modify the liveness information by *adding* any variables which it references (they become live) and *removing* any which it defines (they become dead).

\[
\begin{align*}
\text{ref( print } x \text{ ) } &= \{ x \} \\
\text{def( } x = 3 \text{ ) } &= \{ x \}
\end{align*}
\]
Live variable analysis

If an instruction both references and defines variables, we must remove the defined variables before adding the referenced ones.

\[
\begin{align*}
\{ x, y, z \} \\
x &= x + y \\
\{ x, z \}
\end{align*}
\]

\[
\begin{align*}
def(x = x + y) &= \{ x \} \\
ref(x = x + y) &= \{ x, y \}
\end{align*}
\]
Live variable analysis

So, if we consider in-live\((n)\) and out-live\((n)\), the sets of variables which are live immediately before and immediately after a node, the following equation must hold:

\[
in\text{-live}(n) = \left( out\text{-live}(n) \setminus \text{def}(n) \right) \cup \text{ref}(n)\]

Live variable analysis

\[
in-live(n) = \left( out-live(n) \setminus \text{def}(n) \right) \cup \text{ref}(n)
\]

\[
\begin{align*}
in-live(n) &= \left( \{ x, z \} \setminus \{ x \} \right) \cup \{ x, y \} \\
&= \{ z \} \cup \{ x, y \} = \{ x, y, z \}
\end{align*}
\]

\[
\begin{align*}
\text{out-live}(n) &= \{ x, z \} \\
\text{def}(n) &= \{ x \} \\
\text{ref}(n) &= \{ x, y \}
\end{align*}
\]
Live variable analysis

So we know how to calculate \( \text{in-live}(n) \) from the values of \( \text{def}(n) \), \( \text{ref}(n) \) and \( \text{out-live}(n) \). But how do we calculate \( \text{out-live}(n) \)?

\[
\begin{align*}
in-live(n) &= (\text{out-live}(n) \setminus \text{def}(n)) \cup \text{ref}(n) \\
\end{align*}
\]

\[
\begin{cases}
x = x + y \\
\end{cases}
\]

\[
\begin{align*}
\text{out-live}(n) &= ?
\end{align*}
\]
Live variable analysis

In straight-line code each node has a unique successor, and the variables live at the exit of a node are exactly those variables live at the entry of its successor.
Live variable analysis

\[ z = x * y; \]

\(\text{in-live}(m) = \{ s, t, x, y \}\)

\(\text{out-live}(m) = \{ s, t, z \}\)

\(\text{in-live}(n) = \{ s, t, z \}\)

\(\text{out-live}(n) = \{ z \}\)

\(\text{in-live}(o) = \{ z \}\)

\(\text{print } s + t; \)
Live variable analysis

In general, however, each node has an arbitrary number of successors, and the variables live at the exit of a node are exactly those variables live at the entry of any of its successors.
Live variable analysis

\[
m: \begin{cases} x = 17; \\ \{ x, z \} \end{cases}
\]

\[
n: \begin{cases} y = 19; \\ \{ x, z \}, \{ y, z \} \end{cases}
\]

\[
o: \begin{cases} s = x \times 2; \\ \{ s, z \} \end{cases}
\]

\[
p: \begin{cases} t = y + 1; \\ \{ t, z \} \end{cases}
\]
Live variable analysis

So the following equation must also hold:

\[ \text{out-live}(n) = \bigcup_{s \in \text{succ}(n)} \text{in-live}(s) \]
Data-flow equations

These are the data-flow equations for live variable analysis, and together they tell us everything we need to know about how to propagate liveness information through a program.

\[
in-live(n) = \left( out-live(n) \setminus def(n) \right) \cup ref(n) \\
out-live(n) = \bigcup_{s \in succ(n)} in-live(s)
\]
Data-flow equations

Each is expressed in terms of the other, so we can combine them to create one overall liveness equation.

\[
live(n) = \left( \left( \bigcup_{s \in succ(n)} live(s) \right) \setminus def(n) \right) \cup ref(n)
\]
Algorithm

We now have a formal description of liveness, but we need an actual algorithm in order to do the analysis.
“Doing the analysis” consists of computing a value $\text{live}(n)$ for each node $n$ in a flowgraph such that the liveness data-flow equations are satisfied.

A simple way to solve the data-flow equations is to adopt an iterative strategy.
Algorithm

def x, y
{
}

def z
{
    x, y
}

ref x
{
    x, y, z
}

ref y
{
    y, z
}

ref z
{
    z
}

✗
Algorithm

```python
def x, y:
    { }

def z:
    { x, y }

ref x:
    { x, y, z }

ref y:
    { x, y, z }

ref z:
    { z }
```
Algorithm

for i = 1 to n do live[i] := {}
while (live[] changes) do
    for i = 1 to n do
        live[i] :=
            \(\bigcup_{s \in \text{succ}(i)} \text{live}[s] \setminus \text{def}(i)\)
            \(\cup\ \text{ref}(i)\)
Algorithm

This algorithm is guaranteed to terminate since there are a finite number of variables in each program and the effect of one iteration is monotonic.

Furthermore, although any solution to the data-flow equations is safe, this algorithm is guaranteed to give the smallest (and therefore most precise) solution.

(See the Knaster-Tarski theorem if you’re interested.)
Algorithm

Implementation notes:

• If the program has $n$ variables, we can implement each element of `live[ ]` as an $n$-bit value, with each bit representing the liveness of one variable.

• We can store liveness once per basic block and recompute inside a block when necessary. In this case, given a basic block $n$ of instructions $i_1, ..., i_k$:

$$\text{live}(n) = \left( \bigcup_{s \in \text{succ}(n)} \text{live}(s) \right) \setminus \text{def}(i_k) \cup \text{ref}(i_k) \cdots \setminus \text{def}(i_1) \cup \text{ref}(i_1)$$
Safety of analysis

- Syntactic liveness safely overapproximates semantic liveness.

- The usual problem occurs in the presence of address-taken variables (cf. labels, procedures): ambiguous definitions and references. For safety we must
  - overestimate ambiguous references (assume all address-taken variables are referenced) and
  - underestimate ambiguous definitions (assume no variables are defined); this increases the size of the smallest solution.
Safety of analysis

MOV x, #1
MOV y, #2
MOV z, #3
MOV t32, #&x
MOV t33, #&y
MOV t34, #&z

ɗ

STI t35, #7
def(m) = 
ref(m) = { t35 }

LDI t36, t37
def(n) = { t36 }
ref(n) = { t37, x, y, z }
Summary

• Data-flow analysis collects information about how data moves through a program

• Variable liveness is a data-flow property

• Live variable analysis (LVA) is a backwards data-flow analysis for determining variable liveness

• LVA may be expressed as a pair of complementary data-flow equations, which can be combined

• A simple iterative algorithm can be used to find the smallest solution to the LVA data-flow equations