Lecture Notes: Optimization

17-363/17-663: Programming Language Pragmatics (Fall 2021)
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

1 Local Value Numbering

The Local Value Numbering optimization goes through a basic block one statement at a time, producing a new basic block that is equivalent to the old one, but optimized. As we go along, we build up a map that tracks which virtual registers hold a particular value. We assume the basic block is made up of a series of instructions, and every instruction that writes to a register writes to a unique virtual register.

Consider the following program:

$$
\begin{array}{l}
v1 := x \\
v2 := 3 \\
v3 := v1 + v2 \\
v4 := 1 \\
v5 := x \\
v6 := 2 \\
v7 := v4 + v6 \\
v8 := v5 + v7 \\
v9 := v8 - v3 \\
y := v9 \\
v10 := v3 * v6 \\
\text{return } v10
\end{array}
$$

If we apply the rules described in the textbook section 17.3.2, we get the following answer:

$$
\begin{array}{l}
v1 := x \\
v3 := v1 + 3 \\
v10 := v3 << 1 \\
y := 0 \\
\text{return } v10
\end{array}
$$

The final map will read as follows:

key	value
x	v1
v2	3
v1 + 3	v3
v4	1
v5	v1
v6	2
v7	3
v8	v3
v9	0
y	v9
v3 << 1	v10

2 Live Variable Analysis

Live variable analysis can be summarized with a set of flow functions that describe how individual instructions affect the dataflow information. A flow function takes as input an instruction and input set of dataflow information, which in this case is the set of live variables immediately after the instruction. It produces updated dataflow information, in this case the set of live variables immediately before the instruction. Here are the flow functions for live variable analysis:

$$
\begin{aligned}
f_{LV}[\![x := e]\!](\sigma) &= \text{if } x \in \sigma \text{ then } (\sigma - \{x\}) \cup \mathit{uses}(e) \text{ else } \sigma \\
f_{LV}[\![x := f(y)]\!](\sigma) &= (\sigma - \{x\}) \cup \mathit{uses}(e) \\
f_{LV}[\![\text{goto } n]\!](\sigma) &= \sigma \\
f_{LV}[\![\text{if } \mathit{cond} \text{ goto } n]\!](\sigma) &= \sigma \cup \mathit{uses}(e) \\
f_{LV}[\![\text{return } e]\!](\sigma) &= \mathit{uses}(e)
\end{aligned}
$$

In the flow functions, σ is the set of live variables. $\mathit{uses}(e)$ means all the variables used in e. The return value rule initializes the dataflow information with the set of variables used in the returned expression. The rule for arbitrary (side-effect-free) expressions does nothing if the assigned variable is not live, because we anticipate the expression will be removed by dead code analysis. But if the assigned variable is live, we remove it from the set and we add all the variables used in the right-hand side. Function calls may have side effects, so we assume their arguments are live regardless of whether the variable holding result of the call is live or not.

Consider the following program:

$start$:

$y := x$
$z := 1$

$header$:

if $y = 0$ goto 7

$body$:

$z := z * y$
$y := y - 1$
$x := x + 1$
goto 3

end :

$y := 0$
return z

We can analyze this code as follows, showing the code below and above each basic block, for each time the block is analyzed. Remember that live variables is a backwards analysis, so we start by analyzing end, and we analyze each block bottom up, from the "below/after" information to the "above/before" information.

block	below	above
end	$\varnothing$	$\{z\}$
header	$\{z\}$	$\{y, z\}$
body	$\{y, z\}$	$\{y, z\}$
header	$\{y, z\}$	$\{y, z\}$
start	$\{y, z\}$	$\{x\}$

Notice that we have to analyze header twice, because the input analysis information (from below the basic block) changes. In particular, we analyze the header the first time with input $\{z\}$, but after we analyze the body we find that the information above body can be $\{y, z\}$. We take the union of this information with the information above end, to get $\{y, z\}$ as the new input information below the header. Thus we need to analyze the header again to see if a different result is produced given the new input information that y is also live coming into the header. It turns out this does not change the output (above) information for the header block. Thus the other basic blocks only need to be analyzed once.

If we use this live variable information to optimize the program with dead code elimination, we will remove the instructions $y := 0$ and $x := x + 1$ because both are side-effect free instructions that write to a variable that is not live at that program point.