Lecture Notes on Liveness Analysis

15-411: Compiler Design
Frank Pfenning André Platzer

Lecture 4

1 Introduction

We will see different kinds of program analyses in the course, most of them for the purpose of program optimization. The first one, *liveness analysis*, is required for register allocation. A variable is *live* at a given program point if it will be used during the remainder of the computation, starting at this point. We use this information to decide if two variables could safely be mapped to the same register, as detailed in the last lecture.

Is this decidable? Is liveness decidable? Like many other properties of programs, liveness is undecidable if the language we are analyzing is Turing-complete. The approximation we describe here is standard, although its presentation is not. Chapter 10 of the textbook [App98] has a classical presentation.

2 Liveness by Backward Propagation

Consider a 3-address instruction applying a binary operator $\oplus$:

$$x \quad \leftarrow \quad y \oplus z$$

There are two reasons a variable may be live at this instruction. The first is immediate: if a variable (here: y and z) is used at an instruction, it is used in the computation starting from here. The second is slightly more subtle: since we execute the following instruction next, anything we determine is live at the next instruction is also live here. There is one exception to this second rule: because we assign to x, the value of x coming into this

instruction does not matter (unless it is y or z), even if it is live at the next instruction. In summary,

1. y and z are live at an instruction $x \leftarrow y \oplus z$.

2. u is live at $x \leftarrow y \oplus z$ if u is live at the next instruction and $u \neq x$.

Similarly, for an instruction $x \leftarrow c$ with a constant c, we find that u is live at this instruction if u is live at the next instruction and $u \neq x$.

As a last example, x is live at a return instruction return x, and nothing else is live there.

If we have a straight-line program, it is easy to compute liveness information by going through the program backwards, starting from the return instruction at the end. In that case, it is also precise rather than an approximation. As an example, one can construct the set of live variables at each line in this simple program bottom-up, using the two rules above.

Instructions			Live-in Variables
x_1	$\leftarrow$	1	$\cdot$
x_2	$\leftarrow$	$x_1 + x_1$	x_1
x_3	$\leftarrow$	$x_2 + x_1$	x_1, x_2
y_2	$\leftarrow$	$x_1 + x_2$	x_1, x_2, x_3
y_3	$\leftarrow$	$y_2 + x_3$	y_2, x_3
return		y_3	y_3

For example, looking at the 4th line, we see that x_1 and x_2 are live because of the first rule (they are used) and x_3 is live because it is live at the next instructions and different from y_2.

3 Liveness Analysis in Logical Form

Before we generalize to a more complex language of instructions, we try to specify the rules for liveness analysis in a symbolic form to make them more concise and to avoid any potential ambiguity. For this we give each instruction in a program a line number of *label*. If an instruction has label l, we write $l + 1$ for the label of the next instruction.

We also introduce the predicate live(l, x) which should be true when variable x is live at line l. We then turn the rules stated informally in English into logical rules.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{live}(l, y)\\ \mathsf{live}(l, z)\end{array}}\ L_1 \qquad \frac{\begin{array}{l}l : x \leftarrow y \oplus z\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_2$$

Here, the formulas above the line are premises of the inference rule and the formulas below the line are the conclusions. If all premises are true, we know all conclusions must be true. To the right of the line we write the name of the inference rule. For example, we can read rule L_1 as: "*If line l has the form $x \leftarrow y \oplus z$ then y is live at l and z is live at l.*"

This is somewhat more abstract than the backward propagation algorithm because it does not specify in which order to apply these rules. We can now add more rules for different kinds of instructions.

$$\frac{l : \mathsf{return}\ x}{\mathsf{live}(l, x)}\ L_3 \qquad \frac{\begin{array}{l}l : x \leftarrow c\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_4$$

If we only have binary operators, moves of constants into variables, and return instructions, then these four rules constitute a complete specification of when a variable should be live at any point in a program.

This specification also gives rise to an immediate, yet somewhat nondeterministic implementation. We start with a database of facts, consisting only of the original program, with each line properly labeled. Then we apply rules in an arbitrary order — whenever the premises are all in the database we add the conclusion to the database. Applying one rule may enable the application of another rule and so on, but eventually this process will not gain us any more information. At this point, we can still apply rules but all conclusions are already in the database of facts. We say that the database is *saturated*. Since the rules are a complete specification of our liveness analysis, by definition a variable x is deemed lived at line l if and only if the fact $\mathsf{live}(l, x)$ is in the saturated database.

This may seem like an unreasonable expensive way to compute liveness, but in fact it can be quite efficient, both in theory and practice.

In theory, we can look at the rules and determine their theoretical complexity by (a) counting so-called *prefix firings* of each rule, and (b) bounding

the size of the completed database. We will return to prefix firings, a notion due to McAllester [McA02], in a later lecture. Bounding the size of the completed database is easy. We can infer at most $L \cdot V$ distinct facts of the form $\mathsf{live}(l, x)$, where L is the number of lines and V is the number of variables in the program. Counting prefix firings does not change anything here, and we get a theoretical complexity of $O(L \cdot V)$ for the analysis so far.

In practice, we can implement logical rules more efficiently than traditional techniques by using Binary Decision Diagrams (BDD's). Whaley, Avots, Carbin, and Lam [WACL05] have shown scalability of global program analyses using inference rules, transliterated into so-called Datalog programs. Unfortunately, there is no Datalog library that we can easily tie into our compilers, so while we specify and analyze the structure of our program analyses via the use of inference rules, we generally do not implement them in this manner. Instead, we use other implementations that follow the ideas that are identified precisely and concisely by the logical rules. Because our logical rules identify the fundamental principles, this presentation makes it easier to understand what the important things of liveness analysis are. This also helps capturing the implementation-independent commonality among different styles of implementation. We will see throughout this whole course, that logical rules can capture many other important concepts in a similarly simple way.

4 Loops and Conditionals

The nature of liveness analysis changes significantly when the language permits loops. This will also be the case for most other program analyses.

Here, we add two new forms of instructions, and unconditional jump l : goto l', and a conditional branch l : if $(x$? $c)$ goto l', where "?" is a relational operator such as equality or inequality.

We now discuss how liveness analysis should be extended for these two forms of instructions. A variable u is live at l : goto l' if it is live at l'. We capture this with the following inference rule, which is the only rule pertaining to goto

$$\frac{l : \mathsf{goto}\ l' \qquad \mathsf{live}(l', u)}{\mathsf{live}(l, u)}\ L_5$$

When executing a conditional branch l : if $(x$? $c)$ goto l' we have two potential successor instructions: we may go to the next $l + 1$ if the

condition is false or to l' if the condition is true. In general, we will not be able to predict at compile time whether the condition will be true or false and usually it will sometimes be true and sometimes be false during the execution of the program. Therefore we have to consider a variable live at l if it is live at the possible successor $l+1$ or it is live at the possible successor l'. Also, the instruction uses x, so x is live. Summarizing this as rules we obtain

$$\frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\mathsf{live}(l, x)}\ L_6 \qquad \frac{\begin{array}{l} l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l' \\ \mathsf{live}(l+1, u) \end{array}}{\mathsf{live}(l, u)}\ L_7 \qquad \frac{\begin{array}{l} l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l' \\ \mathsf{live}(l', u) \end{array}}{\mathsf{live}(l, u)}\ L_8$$

These rules are straightforward enough, but if we have backwards branches we will not be able to analyze in a single backwards pass. As an example to illustrate this point, we will use a simple program for calculating the greatest common divisor of two positive integers. We assume that at the first statement labeled 1, variables x_1 and x_2 hold the input, and we are supposed to calculate and return $\gcd(x_1, x_2)$.

		Instructions	Live variables, initially
1	:	$\mathsf{if}\ (x_2 = 0)\ \mathsf{goto}\ 8$	
2	:	$q \leftarrow x_1 / x_2$	
3	:	$t \leftarrow q * x_2$	
4	:	$r \leftarrow x_1 - t$	
5	:	$x_1 \leftarrow x_2$	
6	:	$x_2 \leftarrow r$	
7	:	$\mathsf{goto}\ 1$	
8	:	$\mathsf{return}\ x_1$	

If we start at line 8 we see x_1 is live there, but we can conclude nothing (yet) to be live at line 7 because nothing is known to be live at line 1, the target of the jump. After one pass through the program, listing all variables we know to be live so far we arrive at:

		Instructions	Live variables, after pass 1
1	:	if $(x_2 = 0)$ goto 8	x_1, x_2
2	:	$q \leftarrow x_1/x_2$	x_1, x_2
3	:	$t \leftarrow q * x_2$	x_1, x_2, q
4	:	$r \leftarrow x_1 - t$	x_1, x_2, t
5	:	$x_1 \leftarrow x_2$	x_2, r
6	:	$x_2 \leftarrow r$	r
7	:	goto 1	$\cdot$
8	:	return x_1	x_1

At this point, we can apply the rule for goto to line 7, once with variable x_1 and once with x_2, both of which are now known to be live at line 1. We list the variables that are now further to the right, and make another pass through the program, applying more rules.

			Live-in variables,		
		Instructions	after pass 1	after pass 2	saturate
1	:	if $(x_2 = 0)$ goto 8	x_1, x_2		
2	:	$q \leftarrow x_1/x_2$	x_1, x_2		
3	:	$t \leftarrow q * x_2$	x_1, x_2, q		
4	:	$r \leftarrow x_1 - t$	x_1, x_2, t		
5	:	$x_1 \leftarrow x_2$	x_2, r		
6	:	$x_2 \leftarrow r$	r	x_1	
7	:	goto 1	$\cdot$	x_1, x_2 (from 1)	
8	:	return x_1	x_1		

At this point our rules have saturated and we have identified all the live variables at all program points. From this we can now build the interference graph and from that proceed with register allocation.

The algorithm which saturates the inference rules implies that a variable is designated live at a given line only if we have definitive reason to believe it might be live. Consider the program

```
1 : u1 ← 1
2 : y ← y * x
3 : z ← y + y          // z not used, redundant, still interference graph
4 : x ← x − u1
5 : if (x > 0) goto 2
6 : return y
```

which has a redundant assignment to z in line 3. Since z is never used, z is not found to be live anywhere in this program. Nevertheless, unless we eliminate line 3 altogether, we have to be careful to note that z interferes with x, u_1, and y because those variables are live on line 4. If not, z might be assigned the same register as x, y, or u_1 and the assignment to z would overwrite one of their values.

In the slightly different program

```
1 : u1 ← 1
2 : y ← y * x
3 : z ← z + z          // z live but never needed
4 : x ← x − u1
5 : if (x > 0) goto 2
6 : return y
```

the variable z will actually be inferred to be live at lines 1 through 5. This is because it is used at line 3, although the resulting value is eventually ignored. To capture redundancy of this kind is the goal of *dead code elimination* which requires *neededness analysis* rather than liveness analysis. We will present this in a later lecture.

5 Refactoring Liveness

Figure 1 has a summary of the rules specifying liveness analysis.

This style of specification is precise and implementable, but it is rather repetitive. For example, L_2 and L_4 are similar rules, propagating liveness information from $l + 1$ to l, and L_1, L_3 and L_6 are similar rules recording the usage of a variable. If we had specified liveness procedurally, we would try to abstract common patterns by creating new auxiliary procedures. But what is the analogue of this kind of restructuring when we look at specifications via inference rules? The idea is to identify common concepts and distill them into new predicates, thereby abstracting away from the individual forms of instructions.

Here, we arrive at three new predicates.

1. $\mathsf{use}(l, x)$: the instruction at l *uses* variable x.
2. $\mathsf{def}(l, x)$: the instruction at l *defines* (that is, writes to) variable x.
3. $\mathsf{succ}(l, l')$: the instruction executed after l *may be* l'.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{live}(l, y)\\ \mathsf{live}(l, z)\end{array}}\ L_1 \qquad \frac{\begin{array}{l}l : x \leftarrow y \oplus z\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_2$$

$$\frac{l : \mathsf{return}\ x}{\mathsf{live}(l, x)}\ L_3 \qquad \frac{\begin{array}{l}l : x \leftarrow c\\ \mathsf{live}(l+1, u)\\ x \neq u\end{array}}{\mathsf{live}(l, u)}\ L_4$$

$$\frac{\begin{array}{l}l : \mathsf{goto}\ l'\\ \mathsf{live}(l', u)\end{array}}{\mathsf{live}(l, u)}\ L_5$$

$$\frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\mathsf{live}(l, x)}\ L_6 \qquad \frac{\begin{array}{l}l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'\\ \mathsf{live}(l+1, u)\end{array}}{\mathsf{live}(l, u)}\ L_7 \qquad \frac{\begin{array}{l}l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'\\ \mathsf{live}(l', u)\end{array}}{\mathsf{live}(l, u)}\ L_8$$

Figure 1: Summary: Rules specifying liveness analysis (non-refactored)

Now we split the set of rules into two. The first set analyzes the program and generates the use, def and succ facts. We run this first set of rules to saturation. Afterwards, the second set of rules employs these predicates to derive facts about liveness. It does not refer to the program instructions directly—we have abstracted away from them.

We write the second program first. It translates the following two, informally stated rules into logical language:

1. If a variable is used at l it is live at l.
2. If a variable is live at a possible next instruction and it is not defined at the current instruction, then it is live at the current instruction.

$$\frac{\mathsf{use}(l, x)}{\mathsf{live}(l, x)}\ K_1 \qquad \frac{\begin{array}{l}\mathsf{live}(l', u)\\ \mathsf{succ}(l, l')\\ \neg\mathsf{def}(l, u)\end{array}}{\mathsf{live}(l, u)}\ K_2$$

Here, we use $\neg$ to stand for negation, which is an operator that deserves more attention when using saturation via logic rules. For this to be well-defined we need to know that def does not depend on live. Any implementation must first saturate the facts about def before applying any rules concerning liveness, because the absence of a fact of the form $\mathsf{def}(l, -)$ does not imply that such a fact might not be discovered in a future inference—unless we first saturate the def predicate. Here, we can easily first apply all rules that could possibly conclude facts of the form $\mathsf{def}(l, u)$ exhaustively until saturation. If, after saturation with those rules ($J_1 \ldots J_5$ below), $\mathsf{def}(l, u)$ has not been concluded, then we know $\neg\mathsf{def}(l, u)$, because we have exhaustively applied all rules that could ever conclude it. Thus, after having saturated all rules for $\mathsf{def}(l, u)$, we can saturate all rules for $\mathsf{live}(l, u)$. This simple saturation in stages would break down if there were a rule concluding $\mathsf{def}(l, u)$ that depends on a premise of the form $\mathsf{live}(l', v)$, which is not the case.

We return to the first set of rules. It must examine each instruction and extract the use, def, and succ predicates. We could write several subsets of rules: one subset to generate def, one to generate use, etc. Instead, we have just one rule for each instruction with multiple conclusions for all required predicates.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{use}(l, z)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_1 \qquad \frac{l : \mathsf{return}\ x}{\mathsf{use}(l, x)}\ J_2 \qquad \frac{l : x \leftarrow c}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_3$$

$$\frac{l : \mathsf{goto}\ l'}{\mathsf{succ}(l, l')}\ J_4 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\begin{array}{l}\mathsf{use}(l, x)\\ \mathsf{succ}(l, l')\\ \mathsf{succ}(l, l+1)\end{array}}\ J_5$$

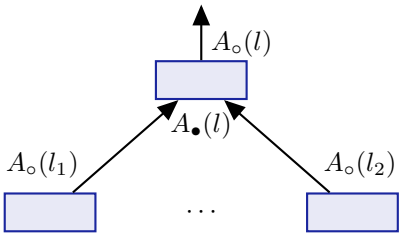
It is easy to see that even with any number of new instructions, this specification can be extended modularly. The main definition of liveness analysis in rules K_1 and K_2 will remain unchanged and captures the essence of liveness analysis.

The theoretical complexity does not change, because the size of the database after each phase is still $O(L \cdot V)$. The only point to observe is that even though the successor relation looks to be bounded by $O(L \cdot L)$, there can be at most two successors to any line l so it is only $O(L)$.

6 Control Flow

Properties of the control flow of a program are embodied in the succ relation introduced in the previous section. The *control flow graph* is the graph whose vertices are the lines of the program and with an edge between l and l' whenever $\mathrm{succ}(l, l')$. It captures the possible flows of control without regard to the actual values that are passed.

An alternative form of presentation of liveness analysis is shown in dataflow equation form in Figure 2. This equation system plays the role of the logic rules from the previous sections.



$$A_\circ(l) = (A_\bullet(l) \setminus \mathit{kill}(l)) \cup \mathit{gen}(l)$$

$$A_\bullet(l) = \bigcup_{l \mapsto l_i} A_\circ(l_i)$$

$$A_\circ(\mathit{final}) = c$$

Statement l	$\mathit{gen}(l)$	$\mathit{kill}(l)$
final	$A_\bullet(\mathit{final}) = \emptyset$	
$x \leftarrow y \oplus z$	$\{y, z\}$	$\{x\}$
$x \leftarrow c$	$\emptyset$	$\{x\}$
goto l'	$\emptyset$	$\emptyset$
return x	$\{x\}$	$\emptyset$
if $(x > c)$ **goto** l'	$\{x\}$	$\emptyset$
l' :	$\emptyset$	$\emptyset$

Figure 2: Dataflow analysis for live-in $A_\circ(l)$ and live-out $A_\bullet(l)$ variables

The textbook [App98] recommends an explicit representation of the control flow graph, together with the use of *basic blocks* to speed up analysis. A *basic block* is a simple fragment of straight-line code that is always entered at the beginning and exited at the end. That is

- the first statement may have a label,
- the last statement terminates the control flow of the current block (with a goto, conditional branch, or a return), and
- all other statements in between have no labels (entry points) and no gotos or conditional branches (exit points).

From a logical perspective, basic blocks do not change anything, because they just accumulate a series of simple statements into one compound code block. Hence, it is not clear if a logical approach to liveness and other program analyses would actually benefit from basic block representations. But depending on the actual implementation technique, basic blocks can help surprisingly much, because the number of nodes that need to be considered in each analysis is reduced somewhat. Basic blocks basically remove trivial control flow edges and assimilate them into a single basic block, exposing only more nontrivial control flow edges. Basic blocks are an example of an engineering decision that looks like a no-op, but can still pay off. They are also quite useful for SSA intermediate language representations and LLVM code generation.

Control flow information can be made more precise if we analyze the possible values that variables may take. Since control flow critically influences other analyses in a similar way to liveness analysis, it is almost universally important. Our current analysis is not sensitive to the actual values of variables. Even if we write

$$
\begin{array}{lcll}
l & : & x \leftarrow 0 & \\
l+1 & : & \mathsf{if}\ (x < 0)\ \mathsf{goto}\ l+3 & \\
l+2 & : & \mathsf{return}\ y & \\
l+3 & : & \mathsf{return}\ z & \text{// unreachable in this program due to values}
\end{array}
$$

we deduce that both y and z may be live at $l+1$ even though only return y can actually be reached. This and similar patterns may seem unlikely, but in fact they arise in practice in at least two ways: as a result of other optimizations and during array bounds checking. We may address this issue in a later lecture.

7 Summary

Liveness analysis is a necessary component of register allocation. It can be specified in two logical rules which depend on the control flow graph, $\mathsf{succ}(l, l')$, as well as information about the variables used, $\mathsf{use}(l, x)$, and defined, $\mathsf{def}(l, x)$, at each program point. These rules can be run to saturation in an arbitrary order to discover all live variables. On straight-line programs, liveness analysis can be implemented in a single backwards pass, on programs with jumps and conditional branches some iteration is required until no further facts about liveness remain to be discovered. Liveness analysis is an example of a *backward dataflow analysis*; we will see more analyses with similar styles of specifications throughout the course.

Quiz

1. Can liveness analysis be faster if we execute it out of order, i.e., not strictly backwards?
2. Is there a program where liveness analysis gives imperfect information?
3. Is there a class of programs where this does not happen? What is the biggest such class?
4. Can you refactor the presentation of the dataflow equations formulation for liveness analysis in similar ways how we have refactored the logic rules formulation liveness analysis?
5. Suppose the definition in Figure 2 were altered as follows:

   $$A_\circ(l) = (A_\bullet(l) \cup gen(l)) \setminus kill(l)$$

   Would liveness analysis still work? Could it work at all? Is there a way to fix it? If so, how, if not, why not?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002. http://dx.doi.org/10.1145/581771.581774.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K.Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 97–118. Springer LNCS 3780, November 2005.