Lecture Notes on Liveness Analysis

15-411: Compiler Design
Frank Pfenning, André Platzer, Rob Simmons, and Jan Hoffmann

Lecture 5
September 12, 2017

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 used 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 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, by which we mean *live just before the instruction is executed* (we'll also say *live-in*, which is slightly less ambiguous). The first reason 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 $\mathsf{return}\ x$, 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.

$\uparrow$

	Instructions			Live-in Variables
$l_1:$	x_1	$\leftarrow$	1	$\cdot$
$l_2:$	x_2	$\leftarrow$	$x_1 + x_1$	x_1
$l_3:$	x_3	$\leftarrow$	$x_2 + x_1$	x_1, x_2
$l_4:$	y_2	$\leftarrow$	$x_1 + x_2$	x_1, x_2, x_3
$l_5:$	y_3	$\leftarrow$	$y_2 + x_3$	y_2, x_3
$l_6:$	ret	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 or *label*. If an instruction has label l, we write $l + 1$ for the label of the next instruction.

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

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

This way of writing rules down is called an inference rule: if all premises (the facts above the line) are true, we know all conclusions (the facts below the line) must be true. We will use these rules as very concise way of stating what we mean by liveness analysis: the complete liveness analysis for a program is all the facts $\mathsf{live}(l, x)$ that can be proven true based on those inference rules.

It's also possible to look at these rules as describing the *computation* we use to determine liveness. Because of line l_5 and rule L_1, we compute the following facts:

$\mathsf{live}(l_5, y_2)$, $\mathsf{live}(l_5, x_3)$

Using rule L_1 again on lines l_3 and l_4, we add more facts to our list of facts:

$\mathsf{live}(l_3, x_2)$, $\mathsf{live}(l_3, x_1)$
$\mathsf{live}(l_4, x_1)$, $\mathsf{live}(l_4, x_2)$
$\mathsf{live}(l_5, y_2)$, $\mathsf{live}(l_5, x_3)$

We can also use the second rule, L_2, along with the fact $l_4 : y_2 \leftarrow x_1 + x_2$ and the fact, present above, that $\mathsf{live}(l_5, x_3)$, to derive $\mathsf{live}(l_4, x_3)$. We cannot use rule L_2 to determine $\mathsf{live}(l_4, y_2)$, though, because y_2 takes the place of both x and u in the rule L_2, and they are not distinct.

$\mathsf{live}(l_3, x_2)$, $\mathsf{live}(l_3, x_1)$
$\mathsf{live}(l_4, x_1)$, $\mathsf{live}(l_4, x_2)$, $\mathsf{live}(l_4, x_3)$
$\mathsf{live}(l_5, y_2)$, $\mathsf{live}(l_5, x_3)$

While we could use L_2 to derive $\mathsf{live}(l_3, x_1)$ and $\mathsf{live}(l_3, x_2)$, we *already* derived those facts using rule L_1, so we don't need to derive them again. It doesn't matter *how* we derive something; we just have to derive it some way or another.

This way of thinking about liveness is more abstract than the backward propagation algorithm. It does not specify in which order to apply these rules. We can now add more rules for different kinds of instructions.

$$\frac{l : \mathsf{ret}\ x}{\mathsf{live}(l, x)}\ L_3 \qquad \frac{\begin{array}{l} l : x \leftarrow c \\ \mathsf{live}(l+1, u) \\ x \neq u \end{array}}{\mathsf{live}(l, u)}\ L_4 \qquad \frac{l : x \leftarrow y}{\mathsf{live}(l, y)}\ L_5 \qquad \frac{\begin{array}{l} l : x \leftarrow y \\ \mathsf{live}(l+1, u) \\ x \neq u \end{array}}{\mathsf{live}(l, u)}\ L_6$$

If we only have binary operators, moves, and return instructions, then these six 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 set 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 set of known facts, we add the conclusion to the set as well. Applying one rule may enable the application of another rule and so on, but eventually this process will not gain us any more information, so we will be left with a set of facts that we cannot grow any further:

$\mathsf{live}(l_2, x_1)$
$\mathsf{live}(l_3, x_2)$, $\mathsf{live}(l_3, x_1)$
$\mathsf{live}(l_4, x_1)$, $\mathsf{live}(l_4, x_2)$, $\mathsf{live}(l_4, x_3)$
$\mathsf{live}(l_5, y_2)$, $\mathsf{live}(l_5, x_3)$
$\mathsf{live}(l_6, y_3)$

At this point, we can still apply rules but all conclusions are already in the set of facts (which we'll also refer to as a *database*). We say that the database is *saturated*. Since the rules are a complete specification of our liveness analysis: a variable x is deemed lived at line l if and only if the fact $\mathsf{live}(l, x)$ is in the saturated database.

The backward propagation algorithm we started with can be seen as a particular way of applying these rules: we apply both L_1 and L_2 using line l_6, then line l_5, then line l_4, and so on.

This may seem like an unreasonably expensive way to compute liveness. 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 may 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 the bound $L \cdot V$ on the number of iterations.

In practice, there are a number of ways logical rules and saturation can be implementation efficiently. One uses 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. See Smaragdakis and Bravenboer's work on Doop [SB10] for a different technique. 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 the important issues of liveness analysis. 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 concise and straightforward 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 $(x \ ? \ c)$ then l_t else l_f, 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

$$\frac{\begin{array}{l} l : \mathsf{goto}\ l' \\ \mathsf{live}(l', u) \end{array}}{\mathsf{live}(l, u)}\ L_7$$

When executing a conditional branch l : if $(x \ ? \ c)$ then l_t else l_f we have two potential successor instructions: we may go to the next l_t if the condition is true or to l_f if the condition is false. 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 either l_t or l_f. Also, the instruction uses x, so x is live. Summarizing this as rules, we obtain:

$$\frac{l : \mathsf{if}\ (x \ ? \ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f}{\mathsf{live}(l, x)}\ L_8$$

$$\frac{\begin{array}{l} l : \mathsf{if}\ (x \ ? \ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f \\ \mathsf{live}(l_t, u) \end{array}}{\mathsf{live}(l, u)}\ L_9 \qquad \frac{\begin{array}{l} l : \mathsf{if}\ (x \ ? \ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f \\ \mathsf{live}(l_f, u) \end{array}}{\mathsf{live}(l, u)}\ L_{10}$$

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
l_1	:	if $(x_2 \neq 0)$ then l_2 else l_8	
l_2	:	$q \leftarrow x_1 / x_2$	
l_3	:	$t \leftarrow q * x_2$	
l_4	:	$r \leftarrow x_1 - t$	
l_5	:	$x_1 \leftarrow x_2$	
l_6	:	$x_2 \leftarrow r$	
l_7	:	goto l_1	
l_8	:	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
l_1	:	if $(x_2 \neq 0)$ then l_2 else l_8	x_1, x_2
l_2	:	$q \leftarrow x_1/x_2$	x_1, x_2
l_3	:	$t \leftarrow q * x_2$	x_1, x_2, q
l_4	:	$r \leftarrow x_1 - t$	x_1, x_2, t
l_5	:	$x_1 \leftarrow x_2$	x_2, r
l_6	:	$x_2 \leftarrow r$	r
l_7	:	goto l_1	$\cdot$
l_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
l_1	:	if $(x_2 \neq 0)$ then l_2 else l_8	x_1, x_2		
l_2	:	$q \leftarrow x_1/x_2$	x_1, x_2		
l_3	:	$t \leftarrow q * x_2$	x_1, x_2, q		
l_4	:	$r \leftarrow x_1 - t$	x_1, x_2, t		
l_5	:	$x_1 \leftarrow x_2$	x_2, r		
l_6	:	$x_2 \leftarrow r$	r	x_1	
l_7	:	goto l_1	$\cdot$	x_1, x_2 (from 1)	
l_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. Now we can build the interference graph and from that proceed with register allocation.

5 Building the Interference Graph

Remember our key observation for interference: two temps need to interfere (and thereby be assigned to two different registers) if they must hold two different values at the same time. Liveness gives us information about *when* we care about the value in a temp.

As a first approximation, we can try saying that two registers interfere when their *live ranges overlap*, which we could try to capture with the following rule:

$$\frac{\mathsf{live}(l, x) \quad \mathsf{live}(l, y) \quad x \neq y}{\mathsf{inter}(x, y)}$$

This is close, but it ignores an important point: the way we are capturing liveness information, there may not be *any* liveness facts about a particular temp. In the following code, we would derive $\mathsf{live}(l_2, a)$ and $\mathsf{live}(l_3, a)$, but we would derive no liveness facts about b.

$l_1 : a \leftarrow 7$
$l_2 : b \leftarrow 3$
$l_3 : \mathsf{ret}\ a$

Nevertheless, it is critical that a and b interfere: if they do not, we could assign them both to the register `%eax`, ending up with a non-equivalent program that returns 3 instead of 7.

$l_1 :$ `%eax` $\leftarrow 7$
$l_2 :$ `%eax` $\leftarrow 3$
$l_3 :$ ret `%eax`

Remark: Overlapping live ranges is the method for constructing the interference graph that is used by Hack [Hac07] to show that interference graphs of programs in SSA from are chordal. This method is sound for programs in which dead code has been eliminated before register allocation (see the lecture on dataflow analysis). In the above program, b is not needed in the code and the instruction $l_2 : b \leftarrow 3$ can be eliminated.

We could do something artificial to solve this problem, like saying that $l : x \leftarrow c$ automatically forces x to be live on line $l + 1$. Instead, we'll rephrase the problem a bit. The languages we are using, C0, always define variables before they are used, so if two destinations have overlapping live ranges, the definition of one destination must fall *inside* the live range of the other. When we write to a destination, we need to make sure that this destination interferes with everything that we *still need* after the write. The values we still need on the next line are precisely the destinations that are live after the assignment.

Therefore, if we write to a destination, we must mark it as interfering with all the destinations that are *live-out* of that instruction. We don't ever want to talk about live-out destinations, so we instead refer to all the destinations that are *live-in*

to the next line.

$$\frac{\begin{array}{l} l : x \leftarrow y \oplus z \\ \mathsf{live}(l+1, u) \\ x \neq u \end{array}}{\mathsf{inter}(x, u)}\ I_1 \qquad \frac{\begin{array}{l} l : x \leftarrow y \\ \mathsf{live}(l+1, u) \\ x \neq u \end{array}}{\mathsf{inter}(x, u)}\ I_2 \qquad \frac{\begin{array}{l} l : x \leftarrow c \\ \mathsf{live}(l+1, u) \\ x \neq u \end{array}}{\mathsf{inter}(x, u)}\ I_3$$

This re-phrasing of interference avoids the problem with our original description of overlapping live ranges. It also lets us make a further optimization: when we have a move $t \leftarrow s$, we do *not* need to create an interference edge between t and s: if these two destinations end up being the same register, this is not a problem. In fact, it's a good thing! In that case, we can just get rid of the move entirely. This optimization corresponds to adding the premise $u \neq y$ to the inference rule I_2 above.

$$\frac{\begin{array}{l} l : x \leftarrow y \\ \mathsf{live}(l+1, u) \\ x \neq u \\ x \neq y \end{array}}{\mathsf{inter}(x, u)}\ I_2'$$

Remark: If you use the optimized rule I_2' then programs in SSA from can have non-chordal graphs.

It may be apparent, looking at I_1, I_2, and I_3 above, that there's a lot of redundancy in our presentation. This specification will only become more redundant (and more error prone) as we add more ways of writing to destinations, like reading from memory, calculating a memory offset, or reading an array's length.

6 Refactoring Liveness

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

This style of specification, like the rules for calculating interference we saw above, is rather repetitive. For example, L_2, L_4 and L_5 are very similar rules, propagating liveness information from $l + 1$ to l, and L_1, L_3 and L_7 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.

$$\frac{l : x \leftarrow y \oplus z}{\mathsf{live}(l, y) \quad \mathsf{live}(l, z)}\ L_1 \qquad \frac{l : x \leftarrow y \oplus z \quad \mathsf{live}(l+1, u) \quad x \neq u}{\mathsf{live}(l, u)}\ L_2 \qquad \frac{l : \mathsf{ret}\ x}{\mathsf{live}(l, x)}\ L_3$$

$$\frac{l : x \leftarrow c \quad \mathsf{live}(l+1, u) \quad x \neq u}{\mathsf{live}(l, u)}\ L_4 \qquad \frac{l : x \leftarrow y}{\mathsf{live}(l, y)}\ L_5 \qquad \frac{l : x \leftarrow y \quad \mathsf{live}(l+1, u) \quad x \neq u}{\mathsf{live}(l, u)}\ L_6 \qquad \frac{l : \mathsf{goto}\ l' \quad \mathsf{live}(l', u)}{\mathsf{live}(l, u)}\ L_7$$

$$\frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f}{\mathsf{live}(l, x)}\ L_8$$

$$\frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f \quad \mathsf{live}(l_t, u)}{\mathsf{live}(l, u)}\ L_9 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f \quad \mathsf{live}(l_f, u)}{\mathsf{live}(l, u)}\ L_{10}$$

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

3. $\mathsf{succ}(l, l')$: the instruction executed after l *may be* l'.

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 ($J_1 \ldots J_6$ 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.

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{use}(l, z)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_1 \qquad \frac{l : \mathsf{return}\ x}{\mathsf{use}(l, x)}\ J_2 \qquad \frac{l : x \leftarrow c}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_3 \qquad \frac{l : x \leftarrow y}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_4$$

$$\frac{l : \mathsf{goto}\ l'}{\mathsf{succ}(l, l')}\ J_5 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f}{\begin{array}{l}\mathsf{use}(l, x)\\ \mathsf{succ}(l, l_t)\\ \mathsf{succ}(l, l_f)\end{array}}\ J_6$$

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 $O(L)$.

7 Implementing Liveness by Line or by Variable

When we implement the analysis, in the absence of a saturating datalog engine, we have to decide how to compute the information on the given program directly.

One option is to use sets of live variables associated with each line of the program. We start with the empty set everywhere and then walk backward along the control flow edges (from l' to l if $\mathsf{succ}(l, l')$) computing the current approximation to the live-in set for line l from the live-in set for line l' and the variables defined at l. We must take care that if a line has multiple predecessors, we explore all the alternatives. We stop on any particular branch when the new live set computed for a line is equal to the one already stored there. We refer to this as a line-oriented traversal, which we used earlier when we walked through an example. The line-oriented traversal has the disadvantage of potentially redundant set operations when loops are traversed multiple times, and it can be extremely inefficient as a result.

Alternatively, we can perform a variable-oriented traversal. When we arrive at a line and see which variables are used there. For each such variable, we see if it is already live, in which case we do nothing. If it is not live, we declare it so and then walk backwards along the control flow edges propagating only the information for the single variable, stopping if it is already known to be live when we reach a line, or if it is defined at that line. This strategy actually achieves the $O(L \cdot V)$ bound, because for each variable we traverse at most $O(L)$ lines.

Here is how this would work in our example

		Instructions
l_1	:	if $(x_2 \neq 0)$ then l_2 else l_8
l_2	:	$q \leftarrow x_1/x_2$
l_3	:	$t \leftarrow q * x_2$
l_4	:	$r \leftarrow x_1 - t$
l_5	:	$x_1 \leftarrow x_2$
l_6	:	$x_2 \leftarrow r$
l_7	:	goto l_1
l_8	:	return x_1

We start at line 8 and propagate the liveness of x_1 to lines 1, 7, 6, in this order. We stop at 6 because x_1 is defined in line 5. The resulting liveness info is recorded in the column labeled (8). Line 7 does not use any variables, line 6 uses r which propagates only to line 5 (see column (6)). Line 5 uses x_2 which propagates around the loop, etc. At the end, all variables used by lines 1 and 2 are already known to be live, so no further propagation takes place.

↑		Instructions	(8)	(6)	(5)	(4)	(4)	(3)
1	:	if $(x_2 \neq 0)$ then l_2 else l_8	x_1		x_2			
2	:	$q \leftarrow x_1/x_2$			x_2	x_1		
3	:	$t \leftarrow q * x_2$			x_2	x_1		q
4	:	$r \leftarrow x_1 - t$			x_2	x_1	t	
5	:	$x_1 \leftarrow x_2$		r	x_2			
6	:	$x_2 \leftarrow r$	x_1	r				
7	:	goto l_1	x_1		x_2			
8	:	return x_1	x_1					

8 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.

Questions

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?

References

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

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[SB10] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, pages 245–251, Oxford, UK, March 2010. Springer LNCS 6702. Revised selected papers.

[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.