Lecture Notes on Dataflow Analysis

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

Lecture 16
March 16, 2023

1 Introduction

In this lecture, we first extend liveness analysis to handle memory references and then consider *neededness analysis* which is similar to liveness and used to discover *dead code*. Both liveness and neededness are backward dataflow analyses. We then describe *reaching definitions*, a forward dataflow analysis which is an important component of optimizations such as constant propagation or copy propagation.

2 Memory References

Recall the rules specifying liveness analysis from Lecture 5.

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

We do not repeat the rules for extracting def, use, and succ from the program. They represent the following:

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

To model the store in our abstract assembly language, we add two new forms of instructions

- Load: $y \leftarrow M[x]$.
- Store: $M[x] \leftarrow y$.

All that is needed to extend the liveness analysis is to specify the def, use, and succ properties of these two instructions.

$$\frac{l : x \leftarrow M[y]}{\begin{array}{l}\mathsf{def}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_6 \qquad \frac{l : M[y] \leftarrow x}{\begin{array}{l}\mathsf{use}(l, x)\\ \mathsf{use}(l, y)\\ \mathsf{succ}(l, l+1)\end{array}}\ J_7$$

The rule J_7 for storing register contents to memory does not define any value, because liveness analysis does not track memory, only variables which then turn into registers. Tracking memory is indeed a difficult task and subject of a number of analyses of which *alias analysis* is the most prominent. We will consider this in a later lecture.

The two rules for liveness itself do not need to change. This is an indication that we refactored the original specification in a good way.

3 Dead Code Elimination

An important optimization in a compiler is *dead code elimination* which removes unneeded instructions from the program. Even if the original source code does not contain unnecessary code, after translation to a low-level language dead code often arises either just as an artifact of the translation itself or as the result of optimizations. We will see an example of these phenomena in Section 5; here we just use a small example.

In this code, we compute the factorial $x!$ of x. The variable x is live at the first line. This would typically be the case of an input variable to a program.

		Instructions	Live variables	
l_1	:	$p \leftarrow 1$	x	
l_2	:	$p \leftarrow p * x$	p, x	
l_3	:	$z \leftarrow p + 1$	p, x	(z not live @ l_4! Dead code?)
l_4	:	$x \leftarrow x - 1$	p, x	
l_5	:	if $(x > 0)$ then l_2 else l_6	p, x	
l_6	:	return p	p	

The only unusual part of the loop is the unnecessary computation of $p + 1$.

We may suspect that line 3 is dead code, and we should be able to eliminate it by replacing it with a nop instruction, which has no effect, or perhaps eliminate it entirely when we finally emit the code. The reason to suspect that line 3 is dead

code is that z is not live at the successor line. While this may be sufficient reason to eliminate the assignment here, this is not true in general. For example, we may have an assignment such as $z \leftarrow p/x$ which is required to raise an exception if $x = 0$, or if an overflow occurs, because the result is too large to fit into the allotted bits on the target architecture (division by -1). Another example is a memory reference such as $z \leftarrow M[x]$ which is required to raise an exception if the address x has actually not been allocated or is not readable by the executing process. We will come back to these exceptions in the next section. First, we discuss another phenomenon exhibited in the following small modification of the program above.

		Instructions	Live variables	
l_1	:	$p \leftarrow 1$	x, z	
l_2	:	$p \leftarrow p * x$	p, x, z	
l_3	:	$z \leftarrow z + 1$	p, x, z	(live but not needed)
l_4	:	$x \leftarrow x - 1$	p, x, z	
l_5	:	if $(x > 0)$ then l_2 else l_6	p, x, z	
l_6	:	return p	p	

Here we see that z is live in the loop (and before it) even though the value of z does not influence the final value returned. To see this yourself, note that in the first backwards pass we find z to be used at line 3. After computing p, x, and z to be live at line 2, we have to reconsider line 5, since 2 is one of its successors, and add z as live to lines 5, 4, and 3.

This example shows that liveness is not precise enough to eliminate even simple redundant instructions such as the one in line 3 above.

4 Neededness

To recognize that assignments as in the previous example program are indeed redundant, we need a different property we call *neededness*. We will structure the specification in the same way as we did for liveness: we analyze each instruction and extract the properties that are necessary for neededness to proceed without further reference to the program instructions themselves.

The crucial first idea is that some variables are needed because an instruction they are involved in may have an *effect*. Let's call such variable *necessary*. Formally, we write $\mathsf{nec}(l, x)$ to say that x is necessary at instruction l. We use the notation $\oslash$ for a binary operator which may raise an exception, such as division or the modulo operator. For our set of instructions considered so far, the following are places

where variables are necessary because of the possibility of effects.

$$\frac{l : x \leftarrow y \oslash z}{\begin{array}{l}\mathsf{nec}(l, y)\\ \mathsf{nec}(l, z)\end{array}}\ E_1 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f}{\mathsf{nec}(l, x)}\ E_2$$

$$\frac{l : \mathsf{return}\ x}{\mathsf{nec}(l, x)}\ E_3 \qquad \frac{l : y \leftarrow M[x]}{\mathsf{nec}(l, x)}\ E_4 \qquad \frac{l : M[x] \leftarrow y}{\begin{array}{l}\mathsf{nec}(l, x)\\ \mathsf{nec}(l, y)\end{array}}\ E_5$$

Here, x is flagged as necessary at a return statement because that is the final value returned, and a conditional branch because it is necessary to test the condition. The effect here is either the jump to l_t, or a jump to l_f. We noted that x was necessary in the instruction $y \leftarrow M[x]$ because x could be an invalid memory address, leading to an exception. However, if the language is *memory safe* (and C0 is), and we guard memory references with appropriate checks (such as bounds checks for arrays), then we can loosen this definition and possibly eliminate the memory load instruction if y is not needed at $l + 1$. This means that our optimizations must take special care to avoid transforming a safe program into an unsafe one.

A side remark: on many architectures including the x86 and x86-64, apparently innocuous instructions such as $x \leftarrow x + y$ have an effect because they set the condition code registers. This makes optimizing unstructured machine code quite difficult. However, in compiler design we have a secret weapon: we only have to optimize the code that we generate! Put a different way, the language that we are analyzing here is an *intermediate language*. Most compiler writers don't have the freedom to change the meaning of source language programs that they receive, and almost no compiler writers have the freedom to change the meaning of the assembly programs they generate, but you have a great deal of freedom in deciding what is or is not allowed in your intermediate languages.

Now that we have extracted when variables are immediately *necessary* at any given line, we have to exploit this information to compute neededness. We write $\mathsf{needed}(l, x)$ if x is needed at l. The first rule captures the motivation for designing the rules for necessary variables.

$$\frac{\mathsf{nec}(l, x)}{\mathsf{needed}(l, x)}\ N_1$$

This seeds the neededness relation and we need to consider how to propagate it. Our second rule is an exact analogue of the way we propagate liveness.

$$\frac{\begin{array}{l}\mathsf{needed}(l', u)\\ \mathsf{succ}(l, l')\\ \neg\mathsf{def}(l, u)\end{array}}{\mathsf{needed}(l, u)}\ N_2$$

The crucial rule is the last one. In an assignment $x \leftarrow y \oplus z$ the variables y and z are needed if x is needed in the remaining computation. If x cannot be shown to be needed, then y and z are not needed if $\oplus$ is an effect free operation. Abstracting away from the particular instruction, we get the following:

$$\frac{\begin{array}{l}\mathsf{use}(l, y)\\ \mathsf{def}(l, x)\\ \mathsf{succ}(l, l')\\ \mathsf{needed}(l', x)\end{array}}{\mathsf{needed}(l, y)}\ N_3$$

We see that neededness analysis is slightly more complex than liveness analysis: it requires three rules instead of two, and we need the new concept of a variable necessary for an instruction due to effects. We can restructure the program slightly and could unify the formulas $\mathsf{nec}(l, x)$ and $\mathsf{needed}(l, x)$. This is mostly a matter of taste and modularity. Personally, I prefer to separate local properties of instructions from those that are propagated during the analysis, because local properties are more easily re-used. The specification of neededness is actually an example of that: it re-uses $\mathsf{use}(l, x)$ in rule N_3 which we first introduced for liveness analysis. If we had structured liveness analysis so that the rules for instructions generate $\mathsf{live}(l, x)$ directly, it would not have worked as well here.

We can now perform neededness analysis on our example program. We have indexed each variable with the numbers of all rules that can be used to infer that they are needed (N_1, N_2, or N_3).

		Instructions	Needed variables
l_1	:	$p \leftarrow 1$	x^2
l_2	:	$p \leftarrow p * x$	$p^3, x^{2,3}$
l_3	:	$z \leftarrow z + 1$	p^2, x^2
l_4	:	$x \leftarrow x - 1$	p^2, x^3
l_5	:	if $(x > 0)$ then l_2 else l_6	$p^2, x^{1,2}$
l_6	:	return p	p^1

At the crucial line l_3, z is defined but not needed on line l_4, and consequently it is not needed at line l_3 either.

Since the right-hand side of $z \leftarrow z + 1$ does not have an effect, and z is not needed at any immediate successor line, this statement is dead code and can be optimized away.

5 Optimization Example

The natural direction for both liveness analysis and neededness analysis is to traverse the program backwards. In this section we present another important anal-

ysis whose natural traversal directions is *forward*. As motivating example for this kind of analysis we use an array access with bounds checks.

In our source language C0 we will have an assignment $x = A[0]$ where A is an array. We also assume there are (assembly language) variables n with the number of elements in array A, variable s with the size of the array elements, and a with the base address of the array. We might then translate the assignment to the following code:

$$\begin{array}{lcl} l_1 & : & i \leftarrow 0 \\ l_2 & : & \mathsf{if}\ (i < 0)\ \mathsf{then\ error\ else}\ l_3 \\ l_3 & : & \mathsf{if}\ (i \geq n)\ \mathsf{then\ error\ else}\ l_4 \\ l_4 & : & t \leftarrow i * s \\ l_5 & : & u \leftarrow a + t \\ l_6 & : & x \leftarrow M[u] \\ l_7 & : & \mathsf{return}\ x \end{array}$$

The last line is just to create a live variable x. We notice that line 2 is redundant because the test will always be false. Computationally, we can figure this out in two steps. First we apply *constant propagation* to replace $(i < 0)$ by $(0 < 0)$ and then apply *constant folding* to evaluate the comparison to 0 (representing falsehood). Line 3 is necessary unless we know that $n > 0$. Line 4 performs a redundant multiplication: because i is 0 we know t must also be 0. This is an example of an arithmetic optimization similar to constant folding. And now line 5 is a redundant addition of 0 and can be turned into a move $u \leftarrow a$, again a simplification of modular arithmetic.

At this point the program has become

$$\begin{array}{lcl} l_1 & : & i \leftarrow 0 \\ l_2 & : & \mathsf{goto}\ l_3 \\ l_3 & : & \mathsf{if}\ (i \geq n)\ \mathsf{then\ error\ else}\ l_4 \\ l_4 & : & t \leftarrow 0 \\ l_5 & : & u \leftarrow a \\ l_6 & : & x \leftarrow M[u] \\ l_7 & : & \mathsf{return}\ x \end{array}$$

Now we notice that line 4 is dead code because t is not *needed*. We can also apply *copy propagation* to replace $M[u]$ by $M[a]$, which now makes u not needed so we can apply *dead code elimination* to line 4. We observe that l_2 has the same meaning as nop in this particular program, since it is a goto that points to the very next line. Finally, we can again apply *constant propagation* to replace the *only* remaining occurrence of

i in line 3 by 0 followed by *dead code elimination* for line 1 to obtain

$$
\begin{array}{lcl}
l_1 & : & \mathsf{nop} \\
l_2 & : & \mathsf{nop} \\
l_3 & : & \mathsf{if}\ (0 \geq n)\ \mathsf{then\ error\ else}\ l_4 \\
l_4 & : & \mathsf{nop} \\
l_5 & : & \mathsf{nop} \\
l_6 & : & x \leftarrow M[a] \\
l_7 & : & \mathsf{return}\ x
\end{array}
$$

which can be quite a bit more efficient than the first piece of code. Of course, when emitting machine code we can delete the nop operations to reduce code size.

One important lesson from this example is that many different kinds of optimizations have to work in concert in order to produce efficient code in the end. What we are interested in for this lecture is what properties we need for the code to ensure that the optimization are indeed applicable.

We return to the very first optimization of constant propagation. We replaced the test $(i < 0)$ with $(0 < 0)$. This looks straightforward, but what happens if some other control flow path can reach the test? For example, we can insert an increment and a conditional to call this optimization into question.

$$
\begin{array}{lcl}
l_1 & : & i \leftarrow 0 \\
l_2 & : & \mathsf{if}\ (i < 0)\ \mathsf{then\ error\ else}\ l_3 \\
l_3 & : & \mathsf{if}\ (i \geq n)\ \mathsf{then\ error\ else}\ l_4 \\
l_4 & : & t \leftarrow i * s \\
l_5 & : & u \leftarrow a + t \\
l_6 & : & x \leftarrow M[u] \\
l_7 & : & \mathsf{return}\ x
\end{array}
\qquad
\begin{array}{lcl}
l_1 & : & i \leftarrow 0 \\
l_2 & : & \mathsf{if}\ (i < 0)\ \mathsf{then\ error\ else}\ l_3 \\
l_3 & : & \mathsf{if}\ (i \geq n)\ \mathsf{then\ error\ else}\ l_4 \\
l_4 & : & t \leftarrow i * s \\
l_5 & : & u \leftarrow a + t \\
l_6 & : & x \leftarrow M[u] \\
\mathbf{l_7} & : & i \leftarrow i + 1 \\
\mathbf{l_8} & : & \mathsf{if}\ (i < n)\ \mathsf{then}\ l_2\ \mathsf{else}\ l_9 \\
l_9 & : & \mathsf{return}\ x
\end{array}
$$

Even though lines 1–6 have not changed, suddenly we can no longer replace $(i < 0)$ with $(0 < 0)$ because the second time line 2 is reached, i is 1. With arithmetic reasoning we may be able to recover the fact that line 2 is redundant, but pure constant propagation and constant folding is no longer sufficient.

What we need to know for copy propagation is that the definition of i in line 1 is the only definition of i that can reach line 2. This is true in the program on the left, but not on the right since the definition of i at line 7 can also reach line 2 if the condition at line 9 is true.

6 Reaching Definitions

We say a definition $l : x \leftarrow \ldots$ *reaches* a line l' if there is a path of control flow from l to l' during which x is not redefined. Because variables in SSA form are

never redefined, this property is only useful for code not in SSA form. In logical language:

- $\mathsf{reaches}(l, x, l')$ if the definition of x at l reaches l' (especially x has not been redefined since).

We only need two inference rules to define this analysis. The first states that a variable definition reaches any immediate successor. The second expresses that we can propagate a reaching definition of x to all successors of a line l' we have already reached, unless this line also defines x.

$$\frac{\mathsf{def}(l, x) \quad \mathsf{succ}(l, l')}{\mathsf{reaches}(l, x, l')}\ R_1 \qquad \frac{\mathsf{reaches}(l, x, l') \quad \mathsf{succ}(l', l'') \quad \neg\mathsf{def}(l', x)}{\mathsf{reaches}(l, x, l'')}\ R_2$$

Analyzing the original program on the left, we see that the definition of i at line 1 reaches lines 2–7, and this is (obviously) the only definition of i reaching lines 2 and 4. We can therefore apply the optimizations sketched above.

In the program on the right hand side, the definition of i at line 7 also reaches lines 2–8 so neither optimization can be applied.

Inspection of rule R_2 confirms the intuition that reaching definitions are propagated *forward* along the control flow edges. Consequently, a good implementation strategy starts at the beginning of a program and computes reaching definitions in the forward direction. Of course, saturation in the presence of backward branches means that we may have to reconsider earlier lines, just as in the backwards analysis.

A word on complexity: we can bound the size of the saturated database for reaching definitions by L^2, where L is the number of lines in the program. This is because each line defines at most one variable (or, in realistic machine code, a small constant number). Counting prefix firings (which we have not yet discussed) does not change this estimate, and we obtain a complexity of $O(L^2)$. This is not quite as efficient as liveness or neededness analysis (which are $O(L \cdot V)$), so we may need to be somewhat circumspect in computing reaching definitions.

7 Summary

We have extended the ideas behind liveness analysis to neededness analysis which enables more aggressive dead code elimination. Neededness is another example of a program analysis proceeding naturally backward through the program, iterating through loops.

We have also seen reaching definitions, which is a forward dataflow analysis necessary for a number of important optimizations such as constant propagation

or copy propagation. Reaching definitions can be specified in two rules and do not require any new primitive concepts beyond variable definitions ($\mathsf{def}(x, l)$) and the control flow graph ($\mathsf{succ}(l, l')$), both of which we already needed for liveness analysis.

Another important observation from the need for dataflow analysis information during optimization is that dataflow analysis may have to be rerun after an optimization transformed the program. Rerunning all analysis exhaustively all the time after each optimization may be time-consuming. Adapting the dataflow analysis information during optimization transformations is sometimes possible as well, but correctness is less obvious.

For an alternative approach to dataflow analysis via *dataflow equations*, see the textbook [App98], Chapters 10.1 and 17.1–3. Notes on implementation of dataflow analyses are in Chapter 10.1–2 and 17.4. Generally speaking, a simple iterative implementation with a library data structure for sets which traverses the program in the natural direction should be efficient enough for our purposes. We would advise against using bitvectors for sets. Not only are the sets relatively sparse, but bitvectors are more time-consuming to implement. An interesting alternative to iterating over the program, maintaining sets, is to do the analysis one variable at a time (see the remark on page 216 of the textbook). The implementation via a saturating engine for Datalog is also interesting, yet a bit more difficult to tie into the infrastructure of a complete compiler. The efficiency gain noted by Whaley et al. [WACL05] becomes only critical for interprocedural and whole program analyses rather than for the intraprocedural analyses we have presented so far.

Questions

1. Why does or liveness analysis not track memory? Should it?
2. Why is neededness different from liveness? Could we reuse part of one analysis for the other? Should we?
3. Why should it be a problem if a single dataflow analysis is slow? We only run it once, don't we?
4. How can the def/use/succ information be made accessible conveniently in a programming language? Does it improve the structure of the code if we do that?
5. Should our intermediate representation have an explicit representation of the control flow graph? What are the benefits and downsides?
6. Why should we care about dead code elimination? Nobody writes dead code down anyways, because that'd be a waste of time.

7. Where do the arithmetic optimizations alluded to in this lecture play a role in compiling? When are they important?

8. Suppose $x = y/z$ is computed but x never used later. That would make the statement not needed and dead code if it wasn't for the fact that the division could have side effects. So it is needed. But what would liveness analysis do about it? How does this impact register allocation? What is the interplay with the special register requirements of integer division?

References

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

[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.