Lecture Notes on Instruction Selection

15-411: Compiler Design
Frank Pfenning and Jan Hoffmann

Lecture 2
August 31, 2017

1 Introduction

In this lecture we discuss the process of instruction selection, which typcially turns some form of intermediate code into a pseudo-assembly language in which we assume to have infinitely many registers called "temps". We next apply register allocation to the result to assign machine registers and stack slots to the temps before emitting the actual assembly code. Additional material regarding instruction selection can be found in the textbook [**?**, Chapter 9].

2 A Simple Source Language

In this lecture, we use a simple source language where a program is just a sequence of assignments terminated by a return statement. The right-hand side of each assignment is a simple arithmetic expression. Later in the course we describe how the input text is parsed and translated into some intermediate form. Here we assume we have arrived at an intermediate representation where expressions are still in the form of trees and we have to generate instructions in pseudo-assembly. We call this form *IR Trees* (for "Intermediate Representation Trees").

We describe the possible IR trees in a kind of pseudo-grammar, which should not be read as a description of the concrete syntax, but the recursive structure of the data.

Programs	$\vec{s}$	::=	$s_1, \ldots, s_n$	sequence of statements
Statements	s	::=	$x = e$	assignment
		|	return e	return, always last
Expressions	e	::=	c	integer constant
		|	x	variable
		|	$e_1 \oplus e_2$	binary operation
Binops	$\oplus$	::=	$+ \mid - \mid * \mid / \mid \ldots$	

3 Abstract Assembly Target Code

For our simple source, we use an equally simple target. Our target language has fixed registers and also arbitrary variables, called here *temps*. We allow variables x with the same name to appear both in expressions in IR trees and as instruction operands.

Programs	$\vec{i}$	::=	$i_1, \ldots, i_n$	
Instructions	i	::=	$d \leftarrow s$	move
		|	$d \leftarrow s_1 \oplus s_2$	binary operation
		|	ret	
Operands	d, s	::=	r	register
		|	c	integer constant (immediate)
		|	t	temp (variable)

We use d to denote operands of instructions that are *destinations* of operations and s for *sources* of operations. There are some restrictions. In particular, immediate operands cannot be destinations. More restrictions arise when memory references are introduced. For example, it may not be possible for more than one operand to be a memory reference.

4 Maximal Munch

The simplest algorithm for instruction selection proceeds top-down, traversing the input tree and recursively converting subtrees to instruction sequences. For this to work properly, we either need to pass down or return a way to refer to the result computed by an instruction sequence. In lecture, it was suggest to pass down a *destination* for the result of an operation. We therefore have to implement a function

cogen(d, e)	a sequence of instructions implementing e, putting the result into destination d.

e	$\mathsf{cogen}(d, e)$	proviso
c	$d \leftarrow c$	
x	$d \leftarrow x$	
$e_1 \oplus e_2$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 \oplus t_2$	(t_1, t_2 fresh)

If our target language has more specialized instructions we can easily extend this translation by matching against more specialized patterns and matching against them first. For example: if we want to implement multiplication by the constant 2 with a left shift, we would add one or two patterns for that.

e	$\mathsf{cogen}(d, e)$	proviso
c	$d \leftarrow c$	
x	$d \leftarrow x$	
$2 * e$	$\mathsf{cogen}(t, e), d \leftarrow t \mathsf{<<} 1$	(t fresh)
$e * 2$	$\mathsf{cogen}(t, e), d \leftarrow t \mathsf{<<} 1$	(t fresh)
$e_1 \oplus e_2$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 \oplus t_2$	(t_1, t_2 fresh)

Since $*$ is a binary operation (that is, $\oplus$ can be $*$), the patterns for e now need to be matched in the listed order to avoid ambiguity and to obtain the intended more efficient implementation. If we always match the deepest pattern first at the root of the expression, this algorithm is called *maximal munch*. This is also a first indication where the built-in pattern matching capabilities of functional programming languages can be useful for implementing compilers.

Now the translation of statements is straightforward. We write $\mathsf{cogen}(s)$ for the sequence of instructions implementing statement s. We assume that there is a special return register r_{ret} so that a return statement is translated to a move into the return register following by a return instruction.

s	$\mathsf{cogen}(s)$
$x = e$	$\mathsf{cogen}(x, e)$
$\mathsf{return}\ e$	$\mathsf{cogen}(r_{\mathrm{ret}}, e), \mathsf{ret}$

Now a sequence of statements constituting a program is just translated by appending the sequences of instructions resulting from their translations. Maximal munch is easy to implement (especially in a language with pattern matching) and gives acceptable results in practice.

5 A Simple Example

Let's apply our translation to a simple program

$$z = (x + 3) * (y - 5), \mathsf{return}\ z$$

Working through code generation and always working on the left subtree before the right now, we obtain

$$\begin{array}{ll} & \mathsf{cogen}(z = (x+3) * (y-5)), \mathsf{cogen}(\mathsf{return}\ z) \\ = & \mathsf{cogen}(z, (x+3) * (y-5)), \mathsf{cogen}(r_{\mathrm{ret}}, z), \mathsf{ret} \\ = & \mathsf{cogen}(t_1, x+3), \mathsf{cogen}(t_2, y-5), z \leftarrow t_1 * t_2, r_{\mathrm{ret}} \leftarrow z, \mathsf{ret} \\ = & \mathsf{cogen}(t_3, x), \mathsf{cogen}(t_4, 2), t_1 \leftarrow t_3 + t_4, \\ & \mathsf{codge}(t_5, y), \mathsf{cogen}(t_6, 5), t_2 \leftarrow t_5 - t_6, z \leftarrow t_1 * t_2 \\ & r_{\mathrm{ret}} \leftarrow z, \mathsf{ret} \end{array}$$

After one more step, we obtain the following program

$$\begin{array}{lcl} t_3 & \leftarrow & x \\ t_4 & \leftarrow & 3 \\ t_1 & \leftarrow & t_3 + t_4 \\ t_5 & \leftarrow & y \\ t_6 & \leftarrow & 5 \\ t_2 & \leftarrow & t_5 - t_6 \\ z & \leftarrow & t_1 * t_2 \\ r_{\mathrm{ret}} & \leftarrow & z \\ \mathsf{ret} & & \end{array}$$

6 Generating "Better" Code

From the example we see that the resulting program has a lot of redundant move instructions. We can eliminate the redundancy in several ways, all of which are prototypical for many of the choices you will have to make while writing your compiler.

1. We can completely redesign the translation algorithm so it generates better code.
2. We can keep the basic structure of the translation but add special cases to avoid introducing some glaring redundancies in the first place.
3. We can keep the translation the same and apply optimizations subsequently to eliminate redundancies.

Let's work through the options.

Instead of passing down a destination, we can have the translation generate and return a source operand which can be used to refer to the value of the expression. Here is what this might look like. We write $\check{e}$ (read: "*down* e") for the sequence of instructions generated for e and $\hat{e}$ (read: "*up* e") as the source operand we can use

to refer to the result.

e	$\check{e}$	$\hat{e}$	proviso
c	$\cdot$	c	
x	$\cdot$	x	
$e_1 \oplus e_2$	$\check{e}_1, \check{e}_2, t \leftarrow \hat{e}_1 \oplus \hat{e}_2$	t	(t fresh)

and for statements

s	$\check{s}$
$x = e$	$\check{e}, x \leftarrow \hat{e}$
return e	$\check{e}, r_{\text{ret}} \leftarrow \hat{e}$, ret

In this formulation, it seems fewer moves are generated from expressions, but we pay for that with explicit moves for assignment and return statements because we cannot pass the left-hand side of the assignment or the return register as an argument to the translation. Working through this new translation for the same program

$$z = (x + 3) * (y - 5), \text{return } z$$

we obtain

$$\begin{array}{lcl} t_1 & \leftarrow & x + 3 \\ t_2 & \leftarrow & y - 5 \\ t_3 & \leftarrow & t_1 * t_2 \\ z & \leftarrow & t_3 \\ r_{\text{ret}} & \leftarrow & z \\ \text{ret} & & \end{array}$$

We observe that straightforward recursive instruction selection, whether we pass destinations down or source operands up, naturally introduces some extra move instructions.

Let us consider the first translation again to expore Option 2. It is easy to add further instructions to avoid generating unnecessary moves. For example:

e	$\text{cogen}(d, e)$	proviso
c	$d \leftarrow c$	
t	$d \leftarrow t$	
$c \oplus e_2$	$\text{cogen}(t_2, e_2), d \leftarrow c \oplus t_2$	(t_2 fresh)
$x \oplus e_2$	$\text{cogen}(t_2, e_2), d \leftarrow s \oplus t_2$	(t_2 fresh)
$e_1 \oplus c$	$\text{cogen}(t_1, e_1), d \leftarrow t_1 \oplus c$	(t_1 fresh)
$e_1 \oplus x$	$\text{cogen}(t_1, e_1), d \leftarrow t_1 \oplus x$	(t_1 fresh)
$\cdots$	$\cdots$	
$e_1 \oplus e_2$	$\text{cogen}(t_1, e_1), \text{cogen}(t_2, e_2), d \leftarrow t_1 \oplus t_2$	(t_1, t_2 fresh)

One can see that this can lead to an explosion in the size of the translation code, especially once source and target become richer languages. Also, are we really sure

that we have now eliminated the undesirable redundancies? In the table above not yet, unless we introduce even more special cases (say, for an operation applied to a variable and a constant). Generally speaking, our advice is to keep code generation and other transformations as simple as possible, using clear and straightforward translations that are easy to understand. This, however, means that even for this very small pair of source and target language, it is worthwhile to consider how we might eliminate moves.

7 The First Two Optimizations

The first two optimizations are aimed at eliminating moves, $t \leftarrow s$. There are two special cases: s could be a constant, or s could be a temp. We first consider the case $t \leftarrow c$ for a constant c. We would like to replace occurrences of t by c in subsequent instructions, an optimization that is called *constant propagation*. However, we can not replace all occurrence of t. Consider, for example:

$$\begin{array}{lllll} 1: & t & \leftarrow & 5 \\ 2: & x & \leftarrow & t-4 \\ 3: & t & \leftarrow & x+7 \\ 4: & z & \leftarrow & t-1 \end{array}$$

In line 3, we store the value of $x + 7$ in t, so t may no longer refer to 5. So it would be incorrect to replace t in line 4 by the constant 5. So we stop with the replacement of t by 5 when we reach an instruction that redefines t.

The second case is an assignment $t \leftarrow x$, just moving a value from one temp to another. Again, we would like to replace occurrences of t by x, an optimization called *copy propagation*. The condition is slightly more complicated than for constant propagation. Consider, for example:

$$\begin{array}{lllll} 1: & t & \leftarrow & x \\ 2: & x & \leftarrow & y-4 \\ 3: & z & \leftarrow & t+7 \end{array}$$

We cannot replace the occurrence of t in line 3 by x, because x now potentially holds a different value than it did in line 1. So we have to stop replacement of t by x at an assignment to either t or x. In this case, it is also not possible to remove the move in line 1 since we are still accessing t in line 3.

We can simplify these conditions. For example, if t is indeed a true, fresh temporary variable introduced in our translation, then we assign to it only once. So we don't even need to check if it is assigned to again. However, if there are variables in the source program which are assigned to more than once, the condition still has to be checked.

8 Static Single Assignment Form

The conditions on the two optimization in the previous sections are not too onerous, but once the language and our optimizations become more complex, so do the conditions. As we have seen, they can be drastically simplified if we know that every variable will be assigned to only once. The idea now is to transform the program into this form, called *static single assignment* (SSA) to simplify the optimizations. This has emerged as a de facto standard representation in modern compilers and is used by tools such as the LLVM. In this lecture we only see a first, very simple version of it.

We describe the algorithm for converting a program to SSA informally, by example. Consider the following program:

$$\begin{array}{lrcl} 1: & t & \leftarrow & 5 \\ 2: & x & \leftarrow & t-4 \\ 3: & t & \leftarrow & t+x \\ 4: & z & \leftarrow & t-1 \end{array}$$

We traverse the program line by line, maintaining a current version number for each temp. When we see a temp for the first time, we assign it version 0 and replace subsequent occurrences as operands by this version. If it is defined by an instruction (here this means that is assigned a new value), then we increase the generation number. After we carry this out on the code above we obtain the following SSA form:

$$\begin{array}{lrcl} 1: & t_0 & \leftarrow & 5 \\ 2: & x_0 & \leftarrow & t_0-4 \\ 3: & t_1 & \leftarrow & t_0+x_0 \\ 4: & z_0 & \leftarrow & t_1-1 \end{array}$$

This new program will in general use more temps, corresponding to different generations of the original temps, but it will always perform the same computations. Now we can optimize simply by replacement since every temp is defined only once.

When we introduce loops and conditionals into our language, the SSA form becomes somewhat more complicated, but it is still simpler than the conditions we would other have to check for our optimizations. In the presence of loops, more example, even if there is only a single instruction $t \leftarrow \ldots$ for a given temp t, it may be assigned to every time around the loop. That's why it is called *static* single assignment form: in the program text (statically) there is only one assignment to each temp, but while executing the program (dynamically) the temp may still assume many different values.

We encourage you to introduce SSA form into your compilers as early as possible in the semester, in order to avoid significant restructuring when the time comes to implement serious optimizations.

9 "Optimal" Instruction Selection

If we have a good cost model for instructions, we can often find better translations if we apply dynamic programming techniques to construct instruction sequences of minimal cost, from the bottom of the tree upwards. In fact, one can show that we get "optimal" instruction selection in this way if we start with tree expressions.

On modern architectures it is very difficult to come up with realistic cost models for the time of individual instructions. Moreover, these costs are not additive due to features of modern processors such as pipelining, out-of-order execution, branch predication, hyperthreading, etc. Therefore, optimal instruction selection is more relevant when we optimize code size, because then the size of instructions is not only unambiguous but also additive. Since we do not consider code-size optimizations in this course, we will not further discuss optimal instruction selection.

10 x86-64 Considerations

Assembly code on the x86 or x86-64 architectures is not as simple as the assumptions we have made here, even if we are only trying to compile straight-line code. One difference is that the x86 family of processors has two-address instructions, where one operand will function as a source as well as destination of an instruction, rather than three-address instructions as we have assumed above. Another is that some operations are tied to specific registers, such as integer division, modulus, and some shift operations. We briefly show how to address such idiosyncracies.

To implement a three-address instruction we replace it by a move and a two-address instruction. For example:[1]

3-address form	2-address form	x86-64 assembly
$d \leftarrow s_1 + s_2$	$d \leftarrow s_1$	`MOVL` s_1, d
	$d \leftarrow d + s_2$	`ADDL` s_2, d

Here we use the GNU assembly language conventions where the destination of an operation comes last, rather than the Intel assembly language format where it comes first.

In order to deal with operations tied to particular registers we have to make similar transformations. It is important to keep the live range of these registers short, so they interfere with other registers as little as possible, as explained in Lecture 3 on register allocation.

[1] Careful: as we noticed during Lecture 3, the shown translation from 3-address to 2-address form requires some conditions unless the source is in SSA form. What are these conditions?

11 Extensions

In general, there will be interdependencies of instruction selection and register allocation. The register allocation depends on which instructions are executed, especially for special instructions on x86-64. Also some of the analysis needed for register allocation may depend on the selected instructions. Conversely, however, optimal instructions may depend on the register assignment. For these and similar reasons, recent advanced compilers, especially those following the so-called SSA intermediate representation combine register allocation and code generation into a joint phase.

Questions

1. How can you implement the data structures for an intermediate representation as defined in this lecture?
2. What are the advantages of working with a 3-address intermediate representation compared to a 2-address representation and vice versa?
3. What is the advantage and disadvantage of using macro expansion for instruction selection, i.e., to associate exactly one instruction sequence to each individual piece of the intermediate language?
4. Why do many CPUs provide such an asymmetric set of instructions? Why do they not just provide us with all useful instructions and no special register requirements?

References