Lecture Notes on Instruction Selection

15-411: Compiler Design
Frank Pfenning

Lecture 2
September 3, 2015

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 [App98, Chapter 9].

2 A Simple Source Language

We use a very 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
		$\mid$	return e	return, always last
Expressions	e	$::=$	c	integer constant
		$\mid$	x	variable
		$\mid$	$e_1 \oplus e_2$	binary operation
Binops	$\oplus$	$::=$	$+ \mid - \mid * \mid / \mid \ldots$	

3 Abstract Assembly Target Code

For our very 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
		$\mid$	$d \leftarrow s_1 \oplus s_2$	binary operation
		$\mid$	ret s	return
Operands	d, s	$::=$	r	register
		$\mid$	c	immediate (integer constant)
		$\mid$	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 Top-down Instruction Selection

The simplest algorithm for instruction selection, sometimes called *maximal munch*, 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 we passed down a *destination* for the result of evaluating an expression. 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	$d \leftarrow x$	
$e_1 \oplus e_2$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 \oplus t_2$	$(t_1, t_2$ new$)$

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

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. 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.

s	$\mathsf{cogen}(s)$	proviso
$x = e$	$\mathsf{cogen}(x, e)$	
$\mathsf{return}\ e$	$\mathsf{cogen}(t, e), \mathsf{ret}\ t$	$(t$ new$)$

Now a sequence of statements constituting a program is just translated by appending the sequences of instructions resulting from their translations. This algorithm is correct, quite 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

$$\begin{array}{rl}
 & \mathsf{cogen}(z = (x+3) * (y-5)), \mathsf{cogen}(\mathsf{return}\ z) \\
= & \mathsf{cogen}(z, (x+3) * (y-5)), \mathsf{cogen}(t_0, z), \mathsf{ret}\, t_0 \\
= & \mathsf{cogen}(t_1, x+3), \mathsf{cogen}(t_2, y-5), z \leftarrow t_1 * t_2, t_0 \leftarrow z, \mathsf{ret}\, t_0 \\
= & \mathsf{cogen}(t_3, x), \mathsf{cogen}(t_4, 2), t_1 \leftarrow t_3 + t_4, \\
 & \mathsf{codge}(t_5, y), \mathsf{cogen}(t_6, 5), t_2 \leftarrow t_5 - t_6, z \leftarrow t_1 * t_2 \\
 & t_0 \leftarrow z, \mathsf{ret}\, t_0
\end{array}$$

After one more step, we obtain the following program

$$\begin{array}{lcl}
t_3 & \leftarrow & x \\
t_4 & \leftarrow & 3 \\
t_1 & \leftarrow & t_3 + t_4 \\
t_5 & \leftarrow & y \\
t_6 & \leftarrow & 5 \\
t_2 & \leftarrow & t_5 - t_6 \\
z & \leftarrow & t_1 * t_2 \\
t_0 & \leftarrow & z \\
\mathsf{ret}\, t_0 & &
\end{array}$$

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 new)

and for statements

s	$\check{s}$	proviso
$x = e$	$\check{e}, x \leftarrow \hat{e}$	
$\mathsf{return}\ e$	$\check{e}, t \leftarrow \hat{e}, \mathsf{ret}\, t$	(t new)

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

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

we obtain

$$\begin{array}{lll} t_1 & \leftarrow & x + 3 \\ t_2 & \leftarrow & y - 5 \\ t_3 & \leftarrow & t_1 * t_2 \\ z & \leftarrow & t_3 \\ t & \leftarrow & z \\ \mathsf{ret}\, t & & \end{array}$$

Therefore, it seems like straightforward top-down recursive instruction selection, whether we pass destinations down or source operands up, naturally introduces some extra move instructions. In the first translation, it is easy to add further instructions to avoid generating unnecessary moves. For example:

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

Proceeding along these lines can easily 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 of the form $t \leftarrow s$. 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:

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

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:

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

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.

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 "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.

9 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.

If our three-address instruction already has the form $d \leftarrow d \oplus s$, then it is already in 2-address form, and if our three-address instruction has the form $d \leftarrow s \oplus d$ and $\oplus$ is commutative ($+$ or $\times$), then we can use commutativity to put it into three-address form.

As long as d and s_2 are *different*, then we can implement any three-address instruction as 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

If we look back at the instruction selection algorithm we described considered, we can observe that s_2 will always be different than d in the 3-address code we emit, and stop here. However, once we introduce code transformations such as copy propogation which could potentially change this invariant, then we need to detect the case where d and s_2 are the same and use extra temps to correctly translate such instructions.

[1]In this class, the most primitive form of instruction we will consider is GNU assembly language, so we will use the convention that the destination of an operation comes last (also called "AT&T syntax") 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. We want to use specific registers for as little time as possible, a point we will consider more in the upcoming lectures. Consider integer division, as an example. On the left is the simple three-address form. In the middle is a reasonable approximation in two-address form. On the right is the actual x86 assembly.

3-address form	2-address form (approx.)	x86-64 assembly
$d \leftarrow s_1 \,/\, s_2$	`%eax` $\leftarrow s_1$	`MOVL` $s_1,$ `%eax`
		`CDQ`
	`%edx` $\leftarrow$ `%eax` $\%\ s_2$	`IDIVL` s_2
	`%eax` $\leftarrow$ `%eax` $/\ s_2$	
	$d \leftarrow$ `%eax`	`MOVL` `%eax`$, d$

The `IDIVL` s_2 instruction divides the 64-bit number represented by [`%edx`, `%eax`] by s_2, storing the quotient in `%eax` and the remainder in `%edx`. The second x86-64 instruction, `CDQ` (also called `CLTD`), has the effect of sign-extending s_1 into a 64-bit quantity represented by the pair of registers [`%edx`, `%eax`]. A less-fancy alternative would have been "`MOVL` s_2, `%edx`" followed by "`SARL` \$31, `%edx`."

The `IDIVL` instruction will raise a division by zero exception when s_2 is 0, or if there is an overflow (if we divide the smallest 32 bit integer in two's complement representation, -2^{31}, by -1). Fortunately, the same behavior is also specified for the source languages we compile in this course.

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

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