Lecture Notes on Instruction Selection

15-411: Compiler Design
Frank Pfenning*

Lecture 2

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.

*Edited by André Platzer

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

3 Abstract Assembly Code Target

For our very simple source, we use an equally simple target. Our target language has fixed registers and also arbitrary temps, which it shares with the IR trees.

Programs	$\vec{i}$	::=	$i_1, \ldots, i_n$	
Instructions	i	::=	$d \leftarrow s$	
		|	$d \leftarrow s_1 \oplus s_2$	
Operands	d, s	::=	r	register
		|	c	immediate (integer constant)
		|	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. We define two functions (which are computed together):

$\check{e}$	a sequence of instructions implementing e	"write down code"
$\hat{e}$	operand which refers to the value computed by e	"get value up"

e	$\check{e}$	$\hat{e}$
c	$\cdot$	c
t	$\cdot$	t
$e_1 \oplus e_2$	$\check{e}_1,\ \check{e}_2,\ t \leftarrow \hat{e}_1 \oplus \hat{e}_2$	t (t 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. We also add a pattern that implements multiplication by the constant 7 with a shift and subtract. These optimizations are called strength reduction, because they reduce the strength of the operators, which can save time, sometimes even at the expense of extra instructions.

e	$\check{e}$	$\hat{e}$
c	$\cdot$	c
t	$\cdot$	t
$2 * e$	$\check{e},\ t \leftarrow \hat{e} \ll 1$	t (t new)
$e * 2$	$\check{e},\ t \leftarrow \hat{e} \ll 1$	t (t new)
$7 * e$	$\check{e},\ t \leftarrow \hat{e} \ll 3,\ t \leftarrow t - \hat{e}$	t (t new)
$e * 7$	$\check{e},\ t \leftarrow \hat{e} \ll 3,\ t \leftarrow t - \hat{e}$	t (t new)
$e_1 \oplus e_2$	$\check{e}_1,\ \check{e}_2,\ t \leftarrow \hat{e}_1 \oplus \hat{e}_2$	t (t new)

Since $*$ is a binary operation (that is $\oplus$ can be $*$), the patterns for e now need to be matched in order so as to avoid ambiguity and to obtain the intended more efficient implementation. Which one do we choose? If we always match the deepest pattern first at the root of the expression, this algorithm is called *maximal munch*. This also is 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 $\check{s}$ for the sequence of instructions implementing statement s ("write down"). We assume that there is a special return register r_{ret} so that a return instruction is translated to a move into this return register.

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

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 Optimal Instruction Selection

If we have a good cost model for instructions, we can often find better translations than maximal munch 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, even though it is an interesting topic.

6 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:

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 the destination 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 special register uses short, so they interfere with other registers as little as possible, as explained in Lecture 3 on register allocation. As an example, we consider integer division. 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`
		`CLTD`
	`%eax` $\leftarrow$ `%eax` $/\; s_2$	`IDIVL` s_2
	||`%edx` $\leftarrow$ `%eax` $\%\; s_2$	
	$d \leftarrow$ `%eax`	`MOVL` `%eax`, d

Here, `CLTD` sign-extends `%eax` into `%edx`. In the Intel Instruction Set Reference, this instruction is called `CDQ`. This is one of relatively few places where the Intel and GNU assembler names of instructions differ. 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`. Note that 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 by -1).

7 General Principle by Example

More general theories and principles have been developed for code generation. There is a whole tool tower including code generator generators and code generator generator generators. The basic ideas are already visible in the previous sections. Yet, let's make consider the issues from a more systematic perspective now.

Abstractly, we here represent programs in our intermediate language as trees of operations on terms (formally: objects in a term algebra). In particular, we assume every operation has only one result. The basic observation due to Weingart 1973 is that every instruction covers a part of such a program tree. Instruction selection is successful when the whole program tree has been covered by instructions this way (but without overlap!). Now

the basic operations we need to do are those of term rewriting, i.e., successively rewriting a term that describes our program into a term describing the series of machine instructions. And we do this term rewriting using rules that describe which program pattern can be implemented by which instructions.

For example, we could have the following competing rules for turning an immediate into a register and for adding registers or adding addresses and registers with different cost each

rule	condition	emit	cost
$imm \rightsquigarrow reg$	if $imm = 0$	XORL reg reg	1
$imm \rightsquigarrow reg$		MOVL imm, reg	2
$+(reg_1, reg_2) \rightsquigarrow reg_2$		ADDL reg_1, reg_2	2
$+(imm(reg_1), reg_2) \rightsquigarrow reg_2$		ADDL reg_1(imm), reg_2	4

With this, there is a common representation as term rewriting, and the questions are whether to do bottom-up rewriting or not and whether to respect and optimize or ignore cost and so on.

8 Extensions

In general, there can 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.

Quiz

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.