CS 4120/5120 Lecture 15 IR Lowering 28 Feb 2018

Lecturer: Andrew Myers

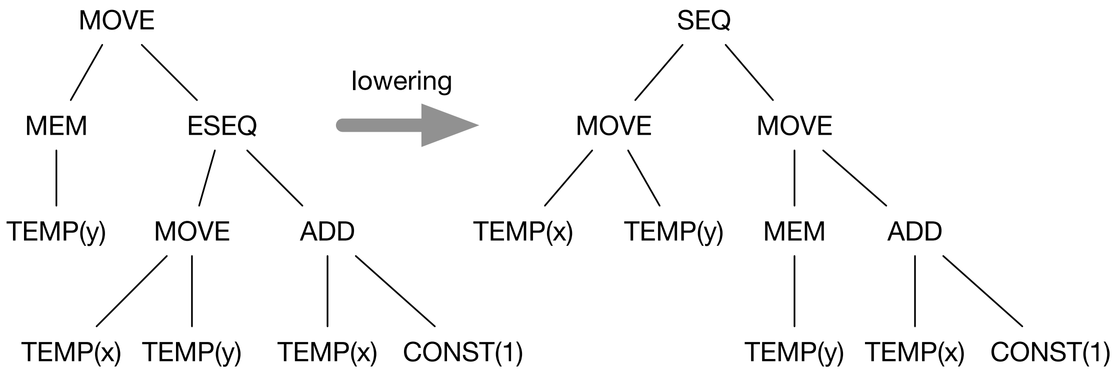
1 IR lowering

After doing the translations described thus far, we arrive at an IR version of the program code. However, this code is still not very assembly-like in various respects: it contains complex expressions and complex statements (because of **SEQ**), and statements inside expressions (because of **ESEQ**). Statements inside expressions means that an expression can cause side effects, and statements can cause multiple side effects. Another difference is the **CJUMP** statement can jump to two different places, whereas in assembly, a conditional branch instruction falls through to the next instruction if the condition is false.

To bring the IR closer to assembly we can flatten statements and expressions, resulting in a *canonical*, lower-level IR in which:

- There are no nested **SEQ**s; just one top-level sequence of lowering statements.
- There are no **ESEQ**s.
- Each statement contains at most one side effect (or call).
- The "false" target of a **CJUMP** always goes to the very next statement.
- All **CALL** nodes appear at the top of the tree, essentially as a kind of IR statement.

We will achieve this by lifting side effects in the IR syntax tree up to the top level. For example, consider the IR statement $\mathbf{MOVE}(\mathbf{MEM}(\mathbf{TEMP}(y)), \mathbf{ESEQ}(\mathbf{MOVE}(\mathbf{TEMP}(x), \mathbf{TEMP}(y)), \mathbf{ADD}(\mathbf{TEMP}(x), \mathbf{CONST}(1))))$. Note that it contains a side effect on the variable $\mathbf{TEMP}(x)$. If it is lowered, a reasonable result would be $\mathbf{MOVE}(\mathbf{TEMP}(x), \mathbf{TEMP}(y)); \mathbf{MOVE}(\mathbf{MEM}(\mathbf{TEMP}(y)), \mathbf{ADD}(\mathbf{TEMP}(x), \mathbf{CONST}(1)))$. Viewed as trees, the transformation can be seen to lift side effects to the top of the tree while ordering them in sequence:



2 Canonical IR

We'll express IR lowering as yet another syntax-directed translation. Unlike the previous translations, the source and target of the translation are both varieties of intermediate representation. We can describe the target language with a grammar that captures the restrictions listed above. First, since there are no nested **SEQ** nodes, the code becomes a linear sequence of other kinds nodes of nodes. For brevity, we will write this sequence as $s_1; s_2; \ldots; s_n$, essentially meaning the same thing as the input node $\mathbf{SEQ}(s_1, \ldots, s_n)$. The grammar for top-level statements is then:

$$
\begin{aligned}
s ::= \; & \mathbf{MOVE}(\mathit{dest}, e) \\
& \mid \mathbf{MOVE}(\mathbf{TEMP}(t), \mathbf{CALL}(f, e_1, \ldots, e_n)) \\
& \mid \mathbf{EXP}(\mathbf{CALL}(f, e_1, \ldots, e_n)) \\
& \mid \mathbf{JUMP}(e) \\
& \mid \mathbf{CJUMP}(e, l_1, l_2) \\
& \mid \mathbf{LABEL}(l) \\
& \mid \mathbf{RETURN}
\end{aligned}
$$

Expressions e are the same as before but may not include **ESEQ** or **CALL** nodes.

3 Translation functions

We express the lowering transformation using two syntax-directed translation functions, $\mathcal{L}[\![s]\!]$ and $\mathcal{L}[\![e]\!]$:

$\mathcal{L}[\![s]\!]$ translates an IR statement s to a sequence $s_1; \ldots; s_n$ of canonical IR statements that have the same effect. We write $\mathcal{L}[\![s]\!] = s_1; \ldots; s_n$, or as a shorthand, $\mathcal{L}[\![s]\!] = \vec{s}$.

$\mathcal{L}[\![e]\!]$ translates an IR expression e to a sequence of canonical IR statements $\vec{s}$ that have the same effect, and an expression e' that has the same value if evaluated after the whole sequence of statements $\vec{s}$. We write $\mathcal{L}[\![e]\!] = \vec{s}\,;\, e'$ to denote this. Notice that this semicolon is not a "real" semicolon in the language; it is merely there to separate the two results of the translation, and the semicolon symbol is chosen to emphasize the sequential ordering of the two parts.

Given these translation functions, we can apply $\mathcal{L}[\![s]\!]$ to the IR for each function body to obtain a linear sequence of IR statements representing the function code. This will get us much closer to assembly code for each function.

Note that these translations will not ensure the fall-though property for **CJUMP**'s. That property will be enforced in a subsequent phase.

4 Lowering expressions

Our goal is to convert an expression into one that has no side effects, the side effects being factored out into a sequence of statements that are hoisted to the top level of the generated code. If we use $\bullet$ to represent an empty sequence of statements, expressions that already have no side effects are trivial to lower. We can write these translations as an inference rule:

$$\frac{e = \mathbf{CONST}(i) \lor e = \mathbf{NAME}(l) \lor e = \mathbf{TEMP}(t)}{\mathcal{L}[\![e]\!] = \bullet; e}$$

For other simple expressions, we just hoist the statements out of subexpressions:

$$\frac{\mathcal{L}[\![e]\!] = \vec{s}; e'}{\mathcal{L}[\![\mathbf{MEM}(e)]\!] = \vec{s}; \mathbf{MEM}(e')}$$

$$\frac{\mathcal{L}[\![e]\!] = \vec{s}; e'}{\mathcal{L}[\![\mathbf{JUMP}(e)]\!] = \vec{s}; \mathbf{JUMP}(e')}$$

$$\frac{\mathcal{L}[\![e]\!] = \vec{s}; e'}{\mathcal{L}[\![\mathbf{CJUMP}(e, l_1, l_2)]\!] = \vec{s}; \mathbf{CJUMP}(e', l_1, l_2)}$$

Since we can hoist statements, we can eliminate **ESEQ** nodes completely:

$$\frac{\mathcal{L}[\![s]\!] = \vec{s} \quad \mathcal{L}[\![e]\!] = \vec{s'}; e'}{\mathcal{L}[\![\mathbf{ESEQ}(s, e)]\!] = \vec{s}; \vec{s'}; e'}$$

Call nodes must be hoisted too because they can cause side effects. And the side effects of computing arguments must be prevented from changing the computation of other arguments:

$$\frac{\mathcal{L}[\![e_i]\!] = \vec{s_i}; e_i' \quad {}^{\forall i \in 0..n}}{\begin{array}{ll}\mathcal{L}[\![\mathbf{CALL}(e_0, e_1, \ldots, e_n)]\!] = & \vec{s_0}; \\ & \mathbf{MOVE}(\mathbf{TEMP}(t_0), e_0'); \\ & \vec{s_1}; \\ & \mathbf{MOVE}(\mathbf{TEMP}(t_1), e_1'); \\ & \ldots \\ & \vec{s_n}; \\ & \mathbf{MOVE}(\mathbf{TEMP}(t_n), e_n'); \\ & \mathbf{MOVE}(\mathbf{TEMP}(t), \mathbf{CALL}(t_0, t_1, \ldots, t_n)); \\ & \quad \mathbf{TEMP}(t)\end{array}}$$

Binary operations are surprisingly tricky. A naive first cut at a lowering translation would be to simply hoist the side effects of both expressions to the top level:

$$\frac{\mathcal{L}[\![e_1]\!] = \vec{s_1}; e_1' \quad \mathcal{L}[\![e_2]\!] = \vec{s_2}; e_2'}{\mathcal{L}[\![OP(e_1, e_2)]\!] = \vec{s_1}; \vec{s_2}; OP(e_1', e_2')}$$

This naive translation has some nice features: beyond its simplicity, it also keeps the expressions e_1' and e_2' together as part of a larger expression, which helps with the quality of later code generation.

However, there's a problem with this naive translation: it reorders the evaluation of $\vec{s_2}$ and e_1'. Let us assume that the language, like Java, dictates that expressions are evaluated in left-to-right order. If executing statements $\vec{s_2}$ changes the value that e_1' computes, the lowering translation will change the behavior of the code. For example, e_2 might use a function call that changes the contents of an array that e_1 reads from.

Therefore, the rule above can only be used if the result of evaluating the expression $OP(e_1, e_2)$ does not depend on the order in which e_1 and e_2 are evaluated. In this case, we say that e_1 and e_2 *commute*:

$$\frac{\mathcal{L}[\![e_1]\!] = \vec{s_1}; e_1' \quad \mathcal{L}[\![e_2]\!] = \vec{s_2}; e_2' \quad e_1 \text{ and } e_2 \text{ commute}}{\mathcal{L}[\![OP(e_1, e_2)]\!] = \vec{s_1}; \vec{s_2}; OP(e_1', e_2')}$$

If they don't commute, the solution is to evaluate e_1 first, store its result into a fresh temporary, and only then evaluate e_2:

$$\frac{\mathcal{L}[\![e_1]\!] = \vec{s_1}; e_1' \quad \mathcal{L}[\![e_2]\!] = \vec{s_2}; e_2'}{\mathcal{L}[\![OP(e_1, e_2)]\!] = \vec{s_1}; \mathbf{MOVE}(\mathbf{TEMP}(t_1), e_1'); \vec{s_2}; OP(\mathbf{TEMP}(t_1), e_2')}$$

We usually prefer the first rule when it is safe to use it, because it keeps the subexpressions e_1' and e_2' together. Breaking code code into more and smaller statements may eliminate opportunities to generate efficient code. For example, keeping expressions may enable the constant-folding optimization. As we'll see later when we do instruction selection, big expression trees are helpful for generating efficient assembly code, because it enables choosing more powerful assembly-language instructions that compute more of the IR tree at once.

5 Lowering statements

Recall that the lowering translation lifts all statements up into a single top-level sequence of statements.

Hence, we translate a sequence of statements by lowering each statement in the sequence to its own sequence of statements, and concatenating the resulting sequences into one big sequence:

$$\mathcal{L}[\![\mathbf{SEQ}(s_1,\ldots,s_n)]\!] = \mathcal{L}[\![s_1]\!];\ldots;\mathcal{L}[\![s_n]\!]$$

To lower an **EXP** node, we just throw away the expression, because that is what **EXP** does. We write the translation rule as an inference rule:

$$\frac{\mathcal{L}[\![e]\!] = \vec{s}; e'}{\mathcal{L}[\![\mathbf{EXP}(e)]\!] = \vec{s}}$$

For statements such as **JUMP** and **CJUMP**, we flatten the expression to obtain a sequence of statements that are done before the jump:

$$\frac{\mathcal{L}[\![e]\!] = \vec{s}; e'}{\mathcal{L}[\![\mathbf{JUMP}(e)]\!] = \vec{s}; \mathbf{JUMP}(e')} \qquad \frac{\mathcal{L}[\![e]\!] = \vec{s}; e'}{\mathcal{L}[\![\mathbf{CJUMP}(e,l_1,l_2)]\!] = \vec{s}; \mathbf{CJUMP}(e',l_1,l_2)}$$

Some statements we can leave alone:

$$\mathcal{L}[\![\mathbf{LABEL}(l)]\!] = \mathbf{LABEL}(l)$$

The **MOVE** statement is another tricky case. It is useful to consider the different kinds of destinations separately. The translation is simple when the destination is a **TEMP**:

$$\frac{\mathcal{L}[\![e]\!] = \vec{s'}; e'}{\mathcal{L}[\![\mathbf{MOVE}(\mathbf{TEMP}(x), e)]\!] = \vec{s'}; \mathbf{MOVE}(\mathbf{TEMP}(x), e')}$$

However, what about a memory location used as a destination; that is, a statement $\mathbf{MOVE}(\mathbf{MEM}(e_1), e_2)$. Assuming that the side effects of the subexpressions e_1 and e_2 are to be performed in left-to-right order, we'd like the following translation:

$$\frac{\mathcal{L}[\![e_1]\!] = \vec{s'_1}; e'_1 \quad \mathcal{L}[\![e_2]\!] = \vec{s'_2}; e'_2}{\mathcal{L}[\![\mathbf{MOVE}(\mathbf{MEM}(e_1), e_2)]\!] = \vec{s'_1}; \vec{s'_2}; \mathbf{MOVE}(\mathbf{MEM}(e'_1), e')}$$

However, this translation moves the effects of computing e_2 before the computation of the location of the memory to be updated, so it is only correct if the computation of e_2 does not affect the destination of the **MOVE**. In fact, the reason why the rule for a **MOVE** into a **TEMP** was correct was the same: the destination couldn't be affected by the expression being moved. (Note that it's fine for the expression to affect the *contents* of the destination; we are only concerned whether it changes the *location* of the destination.) We can exploit this insight to write a single rule that combines the two previous rules for **MOVE**:

$$\frac{\mathcal{L}[\![\mathit{dest}]\!] = \vec{s'_1}; \mathit{dest}' \quad \mathcal{L}[\![e_2]\!] = \vec{s'_2}; e'_2 \quad e_2 \text{ does not affect the location of } \mathit{dest}}{\mathcal{L}[\![\mathbf{MOVE}(\mathit{dest}, e_2)]\!] = \vec{s'_1}; \vec{s'_2}; \mathbf{MOVE}(\mathit{dest}', e')} \ (\textsc{Move-Commuting})$$

Note that we haven't defined what it means to lower a destination. That is because IR expressions have been crafted in such a way that we can reuse the translation for lowering expressions.

Of course, this translation is not safe in general. The problem is the same as with the translation of binary expressions: if the statements $\vec{s'}$ change the location of *dest*, the translated program does something different from the original. In this case, we need a (generally less efficient) rule that uses a temporary to save the address into which the **MOVE** is going to place the result of e_2:

$$\frac{\mathcal{L}[\![e_1]\!] = \vec{s'_1}; e'_1 \quad \mathcal{L}[\![e_2]\!] = \vec{s'_2}; e'_2 \quad t \text{ is fresh}}{\mathcal{L}[\![\mathbf{MOVE}(\mathbf{MEM}(e_1), e_2)]\!] = \vec{s'_1}; \mathbf{MOVE}(\mathbf{TEMP}(t), e'_1); \vec{s'_2}; \mathbf{MOVE}(\mathbf{MEM}(\mathbf{TEMP}(t)), e'_2);} \ (\textsc{Move-General})$$

6 Examples

To see that the lowering translation works, let's consider the example from above, which we can write a bit more compactly by representing $\mathbf{TEMP}$ nodes as their names and $\mathbf{CONST}$ names as their values: $\mathbf{MOVE}(\mathbf{MEM}(y), \mathbf{ESEQ}(\mathbf{MOVE}(x, y), \mathbf{ADD}(x, 1)))$. The lowering of this expression is computed using MOVE-COMMUTING:

$$\mathcal{L}[\![\mathbf{MOVE}(\mathbf{MEM}(y), \mathbf{ESEQ}(\mathbf{MOVE}(x, y), \mathbf{ADD}(x, 1)))]\!] = \vec{s_1'}; \vec{s_2'}; \mathbf{MOVE}(\mathit{dest}', e')$$

where

$$\mathcal{L}[\![\mathbf{MEM}(y)]\!] = \vec{s_1'}; \mathit{dest}' = \bullet; \mathbf{MEM}(y)$$
$$\mathcal{L}[\![\mathbf{ESEQ}(\mathbf{MOVE}(x, y), \mathbf{ADD}(x, 1))]\!] = \vec{s_2'}; e_2' = \mathbf{MOVE}(x, y); \mathbf{ADD}(x, 1)$$

Putting the pieces together, we have $\mathbf{MOVE}(x, y); \mathbf{MOVE}(\mathbf{MEM}(y), \mathbf{ADD}(x, 1))$, as expected.

What if the original statement had updated $\mathbf{MEM}(x)$ instead of $\mathbf{MEM}(y)$? In that case, the computation of the $\mathbf{MOVE}$ expression *would* have changed the destination of the $\mathbf{MOVE}$, so we'd use the MOVE-GENERAL rule:

$$\begin{aligned}&\mathcal{L}[\![\mathbf{MOVE}(\mathbf{MEM}(x), \mathbf{ESEQ}(\mathbf{MOVE}(x, y), \mathbf{ADD}(x, 1)))]\!]\\ &\quad = \vec{s_1'}; \mathbf{MOVE}(\mathbf{TEMP}(t), e_1'); \vec{s_2'}; \mathbf{MOVE}(\mathbf{MEM}(\mathbf{TEMP}(t)), e_2');\\ &\quad = \mathbf{MOVE}(t, x); \mathbf{MOVE}(x, y); \mathbf{MOVE}(\mathbf{MEM}(t), \mathbf{ADD}(x, 1))\end{aligned}$$

7 Commuting statements and expressions

The rules for OP and $\mathbf{MOVE}$ both rely on interchanging the order of a statement s and an expression e. This can be done safely when the statement cannot alter the value of the expression. There are two ways in which this could happen: the statement could change the value of a temporary variable used by the expression, and the statement could change the value of a memory location used by the expression. It is easy to determine whether the statement updates a temporary used by the expression, because temporaries have unique names. Memory is harder because two memory location can be *aliases*. If the statement s uses a memory location $\mathbf{MEM}(e_1)$ as a destination, and the expression reads from the memory location $\mathbf{MEM}(e_2)$, and e_1 might have the same value as e_2 at run time, we cannot safely interchange the operations.

A simple, conservative approach is to assume that all $\mathbf{MEM}$ nodes are potentially aliases, so a statement and an expression that both use memory are never reordered. We can do better by exploiting the observation that two nodes of the form $\mathbf{MEM}(\mathbf{TEMP}(t) + \mathbf{CONST}(k_1))$ and $\mathbf{MEM}(\mathbf{TEMP}(t) + \mathbf{CONST}(k_2))$ cannot be aliases if $k_1 \neq k_2$ [1].

To do a good job of reordering memory accesses, we want more help from analysis of the program at source level. Otherwise there is not enough information available at the IR level for alias analysis to be tractable.

A simple observation we can exploit is that if the source language is strongly typed, two $\mathbf{MEM}$ nodes with different types cannot be aliases. If we keep around source-level type information for each $\mathbf{MEM}$ node, which we might denote as $\mathbf{MEM}_t(e)$, then these type annotations t can help identify opportunities to reorder operations. Accesses to $\mathbf{MEM}_t(e_1)$ and to $\mathbf{MEM}_{t'}(e_2)$ cannot conflict if t is not compatible with t'.

Sophisticated compilers incorporate some form of *pointer analysis* to determine which memory locations might be aliases. The typical output of pointer analysis is a label for each distinct $\mathbf{MEM}$ node, such that accesses to differently labeled $\mathbf{MEM}$ nodes cannot interfere with each other. The label could be as simple as an integer index, where $\mathbf{MEM}_i$ and $\mathbf{MEM}_j$ cannot be aliases unless $i = j$. We'll see how to do such a pointer analysis in a later lecture.

[1] We also need to know that the language run-time system will never map two virtual addresses to the same physical address.