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.
- 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.

2 Canonical IR

We'll express this IR lowering as yet another syntax-directed translation. Unlike the previous translations, the source and target of the translation are both varieties of IR. We can describe the target language with a grammar. 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 $s_1; s_2; \ldots; s_n$, equivalent to source-level $\mathbf{SEQ}(s_1, \ldots, s_n)$. The grammar for top-level statements is then:

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

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:

> $\mathcal{L}[\![s]\!]$ 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.

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.

4 Lowering statements

We translate a sequence of statements by lowering each statement, and concatenating the results:

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

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:

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

A tricky case is **MOVE**. We'd like the following translation, but it's not safe in general:

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

The problem is that if the statements $\vec{s'}$ change the value of e', the meaning of the translated program is not the same as the original. We'll say that e' *commutes* with s' if s' can be evaluated either before or after e' with the same effect and the same value resulting from e'. The **MOVE** rule above can only be used if s' and e' are guaranteed to commute.

What if they don't commute? In that case we can capture the value of e' in a fresh temporary. This is not as good a translation in general, because the simpler **MOVE** node at the end may eliminate some opportunities to generate efficient code. As we'll see later when we do instruction selection, big expression trees are helpful for generating efficient assembly code.

$$\frac{\mathcal{L}[\![e]\!] = \vec{s}; e' \quad \mathcal{L}[\![dest]\!] = \vec{s'}; dest' \quad e' \text{ and } s' \text{ do not commute}}{\mathcal{L}[\![\mathbf{MOVE}(dest, e)]\!] = \vec{s}; \mathbf{MOVE}(\mathbf{TEMP}(t), e'); \vec{s'}; \mathbf{MOVE}(dest', t)}$$

Deferring the question of how to decide when expressions and statements commute, let's look at how to lower IR expressions.

5 Lowering expressions

Recall that 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. If we use $\bullet$ to represent an empty sequence of statements, expressions that already have no side effects are trivial to lower:

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

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:

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

Binary operations create the same commutation problem we saw with **MOVE**:

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

Call nodes must be hoisted too because they can cause side effects. And we have to prevent the side effects of computing arguments from changing the computation of other arguments.

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

6 Commuting statements and expressions

The rules for OP and **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 **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 **MEM** nodes with different types cannot be aliases. If we keep around source-level type information for each **MEM**

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

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 **MEM** node, such that accesses to differently labeled **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.