In the previous lecture we saw how to translate a high-level language into a tree IR representation. We dodged one feature of the language we are compiling: that some variables have a representation that is larger than one word. Many languages, such as C, have this feature. For efficiency, we would like to be able to represent such variables using registers (when enough registers are available) rather than putting their values into memory.

We'll define some additional syntax-directed translation functions to accomplish this part of the translation. Because our IR has no variables larger than one word, the main goal of this translation is to eliminate all variables with tuple type, flattening them into multiple single-word variables that can be used at the IR level.

1 The target

We continue to use an IR based on Appel's tree-structured IR:

$$\begin{aligned} s ::= \;& \mathbf{MOVE}(e_{dest}, e_{src}) \mid \mathbf{EXP}(e) \mid \mathbf{SEQ}(s_1, \ldots, s_n) \\ & \mid \mathbf{JUMP}(e) \mid \mathbf{CJUMP}(e, l_1, l_2) \mid \mathbf{LABEL}(l) \end{aligned}$$

$$\begin{aligned} e ::= \;& \mathbf{CONST}(i) \mid \mathbf{TEMP}(t) \mid OP(e_1, e_2) \\ & \mid \mathbf{MEM}(e) \mid \mathbf{CALL}(e_f, e_1, \ldots, e_n) \mid \mathbf{NAME}(l) \mid \mathbf{ESEQ}(s, e) \end{aligned}$$

$$OP ::= \mathbf{ADD} \mid \mathbf{SUB} \mid \mathbf{MUL} \mid \mathbf{DIV} \mid \mathbf{MOD} \mid \mathbf{SHL} \mid \mathbf{SHR} \mid \mathbf{ASHR}$$

2 A translation for tuples

We need a way to translate Iota$_9$ expressions with tuple type. Previously we developed a translation $\mathcal{E}[\![e]\!]$ that translates expressions whose value takes up one word (this included arrays; because arrays have variable length in Iota$_9$, we represent arrays as a pointer to the memory location where their elements are stored).

Let $w(t)$ mean the number of words needed to represent value of type t:

$$\begin{aligned} w(\texttt{int}) &= 1 \\ w(\texttt{bool}) &= 1 \\ w(t\texttt{[]}) &= 1 \\ w((t_1, t_2, \ldots, t_n)) &= \sum_{i \in 1..n} w(t_i) \qquad \text{(This is not necessarily equal to } n\text{!)} \end{aligned}$$

For expressions with tuple type $t = (t_1, \ldots, t_n)$, we need a syntax-directed translation that produces a result taking up $m = w(t)$ words:

> $\mathcal{T}[\![e, y_1, y_2, \ldots, y_m]\!]$ translates a term e of tuple type $(t_1, t_2, \ldots, t_n)$ to an IR *statement* that stores the representation of the result of e into destinations $y_1, y_2, \ldots, y_m$.

In each of the following translations, we assume we are translating an expression with type $t = (t_1, \ldots, t_n)$, and $w(t) = m$. We write $s_1; \ldots; s_k$ on the right-hand side as a shorthand for $\mathbf{SEQ}(s_1, \ldots, s_k)$.

We translate a tuple constructor recursively according to the following cases:

$$\mathcal{E}[\![(e_1,\ldots,e_n),y_1,\ldots,y_m]\!] = \mathbf{MOVE}(y_1,\mathcal{E}[\![e_1]\!]);\mathcal{E}[\![(e_2,\ldots,e_n),y_2,\ldots,y_m]\!] \qquad \text{(if } w(t_1)=1\text{)}$$
$$\mathcal{E}[\![(e_1,\ldots,e_n),y_1,\ldots,y_m]\!] = \mathcal{T}[\![e_1,y_1,\ldots,y_{w(t_1)}]\!];\mathcal{E}[\![(e_2,\ldots,e_n),y_{w(t_1)+1},\ldots,y_m]\!]$$

For translating a variable, we have the problem that a single variable $x : t$ might take up multiple variables at the IR level. We assume that the semantic analysis phase has associated fresh IR-level variables $x_1,\ldots,x_m$ with the source-level variable x. Then the translation is trivial. As we'll see, in many cases the **MOVE**s can be eliminated after code generation, by representing x_i and y_i with the same register.

$$\mathcal{T}[\![x,y_1,\ldots,y_m]\!] = \mathbf{MOVE}(y_1,x_1);\ldots;\mathbf{MOVE}(y_m,x_m)$$

3 Tuple declarations

The Iota$_9$ language has the unusual feature of tuple declarations, which appear as statements in the language. Earlier we introduced a translation for statements, which we must extend to handle tuple declarations of the form $(d_1,\ldots,d_n) = e$. Note that each d_i may be either a declaration $x_i : t_i$ or $\_$. By appealing to yet another translation function $\mathcal{D}[\![\ldots]\!]$ that binds the components of a tuple to a list of declarations, we can translate a tuple declaration straightforwardly:

$$\mathcal{S}[\![(d_1,\ldots,d_n) = e]\!] = \mathcal{T}[\![e,x_1,\ldots,x_m]\!];\mathcal{D}[\![(d_1,\ldots,d_n),x_1,\ldots,x_m]\!]$$

The specification of this new translation function is as follows:

> $\mathcal{D}[\![(d_1,\ldots,d_n),x_1,x_2,\ldots,x_m]\!]$ is an IR statement that binds the variables appearing in declarations $d_1,\ldots,d_n$ to the corresponding parts of the information contained in variables $x_1,\ldots,x_m$.

Again, we define this translation recursively.

$$\mathcal{D}[\![(\_,d_2,\ldots,d_n),x_1,x_2,\ldots,x_m]\!] = \mathcal{D}[\![(d_2,\ldots,d_n),x_{1+w(t_1)},x_{2+w(t_1)},\ldots,x_m]\!]$$
$$\mathcal{D}[\![(y_1:t_1,d_2,\ldots,d_n),x_1,x_2,\ldots,x_m]\!] = \mathbf{MOVE}(y_1,x_1);\mathcal{D}[\![(d_2,\ldots,d_n),x_2,\ldots,x_m]\!] \qquad \text{(where } w(t_1)=1\text{)}$$
$$\mathcal{D}[\![(y_1:t_1,d_2,\ldots,d_n),x_1,x_2,\ldots,x_m]\!] = \mathbf{MOVE}(y_{11},x_1);\ldots;\mathbf{MOVE}(y_{1k});\mathcal{D}[\![(d_2,\ldots,d_n),x_{k+1},x_{k+2},\ldots,x_m]\!]$$
$$\text{(where } k = w(t_1) \text{ and } y \text{ is represented by IR variables } y_{11},\ldots,y_{1k}\text{)}$$

4 Functions

We need to extend the function translations from earlier to handle passing and returning tuples. In case we are passing a tuple and returning a word-sized argument, we use the $\mathcal{T}$ translation to set up the arguments:

$$\mathcal{E}[\![f(e)]\!] = \mathcal{T}[\![e,x_1,\ldots,x_{w(t')}]\!];\mathbf{CALL}(f,x_1,\ldots,x_{w(t')})$$
$$\text{(where } f : t'\text{->}t \text{ and } t' = (t'_1,\ldots,t'_n) \text{ and } w(t)=1\text{)}$$

For a function that *returns* a tuple, we need to decide on a calling convention. We'll assume that a function that returns a tuple gets at the IR level an extra argument that comes before any of the source-level argument. This argument will be the address in memory of a block of words big enough to store the representation of the tuple to be returned. Since such a function returns a tuple, we will only use it with the $\mathcal{T}$ translation. We will assume we can access the stack pointer using the IR expression $\mathbf{TEMP}(SP)$.

$$\begin{aligned}\mathcal{T}[\![f(e), y_1, \ldots, y_m]\!] = \;& \mathbf{MOVE}(\mathbf{TEMP}(SP), \mathbf{SUB}(SP, \mathbf{CONST}(w(t))));\\ & \mathbf{CALL}(f, \mathbf{TEMP}(SP), \mathcal{E}[\![e]\!]);\\ & \mathbf{MOVE}(y_1, \mathbf{MEM}(\mathbf{TEMP}(SP)));\\ & \mathbf{MOVE}(y_2, \mathbf{MEM}(\mathbf{ADD}(\mathbf{TEMP}(SP), 4)));\\ & \ldots\\ & \mathbf{MOVE}(y_m, \mathbf{MEM}(\mathbf{ADD}(\mathbf{TEMP}(SP), 4 * m - 4)))\\ & \quad \text{(where } f : t' \text{->} t \text{ and } w(t') = 1)\end{aligned}$$

What if both the arguments and the return value are tuples? We just combine the previous two translations. This is left as an exercise for the reader.

5 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.

6 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:

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

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

7 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.