CS 4120 Lecture 14 Syntax-directed translation 28 September 2009
Lecturer: Andrew Myers

We want to translate from a high-level programming into an intermediate representation (IR). This lecture introduces *syntax-directed translation* as a concise way to formally describe the translation process.

1 The target

We are using an IR based on Appel's tree-structured IR:

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

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

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

We've written these as a grammar, but the grammar also stands for an abstract syntax tree representation of the IR.

2 Translation spec

We want to implement a translation from high-level AST nodes to these IR nodes, something like the following methods:

```
class Stmt {
    /** Return an IR statement node implementing this AST node. */
    IRStmtNode translate() {
    }
}
class Expr {
    /** Return an IR expression node implementing this AST node. */
    IRExprNode translate() {
    }
}
```

We will formally describe these two methods with translation functions $\mathcal{S}[\![s]\!]$ and $\mathcal{E}[\![e]\!]$. If s is an Iota$_9$ statement, $\mathcal{S}[\![s]\!]$ is its translation into an IR statement node that has the same effect. If e is a Iota$_9$ expression,

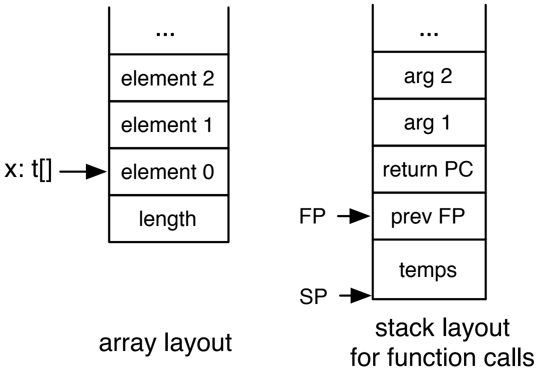

Figure 1: Memory layouts

$\mathcal{E}[\![e]\!]$ is its translation into an IR expression node that has the same side effects and evaluates to the same value (or more precisely, the correct IR-level representation of the same value). We will define these two translation functions recursively.

When developing translations, the key is to clearly define the specification of the translation, and to choose the right specification. Once the specification is right, the translation pretty much writes itself. Think of the specification as a contract that must be guaranteed by the actual translation chosen, but can rely on the contract being satisfied by any recursive uses of the same specification.

3 Memory layout

The IR is considerably lower-level than the source level: it eliminates all higher-level data structures and control structures, making memory accesses and jumps explicit. This means we have to make some decisions about how built-in types are represented in memory, as depicted in Figure 1.

Arrays need to store their own length, to be able to implement the `length` operation. We'll represent a value of array type as a reference to element 0, and store the length just before that in memory.

Functions will be implemented using the C calling conventions for IA-32 (Intel 32-bit) processors: the arguments can be accessed relative to the frame pointer register, which we will write as $\mathbf{TEMP}(FP)$ or just FP in the IR. The first argument is at address $FP + 8$, the second at $FP + 12$, and so on.

To keep things simple for now, we ignore expressions with tuple type. Because the result of these expressions is larger than a single, word-sized IR value, we can't translate them directly to IR expressions.

4 Translating expressions

We define the translation of expression e by considering each of the possible syntactic forms that e can take. For each form, there is a single rule that can be chosen to translate the expression. Therefore translation does not involve any searching; it is syntax-directed, just as the type systems we wrote earlier were.

$$\begin{array}{ll}
\mathcal{E}[\![n]\!] = \mathbf{CONST}(n) & \\
\mathcal{E}[\![x]\!] = \mathbf{TEMP}(x) & \text{(where x is a local variable name)} \\
\mathcal{E}[\![x]\!] = \mathbf{MEM}(\mathbf{NAME}(x)) & \text{(where x is a global variable name)} \\
\mathcal{E}[\![x]\!] = \mathbf{MEM}(\mathbf{ADD}(\mathbf{TEMP}(FP), \mathbf{CONST}(4 * i + 4))) & \text{(where x is the } i\text{th function parameter)}
\end{array}$$

Different kinds of variables are translated differently. Here we've chosen for function parameters to store them in the stack location where arguments are pushed as part of the function calling conventions.

Alternatively we could use a **TEMP**, and start each function with a *prologue* that copies all the stack locations into **TEMP** nodes.

Arithmetic is straightforward to translate. Notice that we have to use the translation function $\mathcal{E}[\![e]\!]$ recursively on the subexpressions.[1]

$$\begin{aligned}
\mathcal{E}[\![e_1 + e_2]\!] &= \mathbf{ADD}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!]) \\
\mathcal{E}[\![e_1 - e_2]\!] &= \mathbf{SUB}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!]) \\
\mathcal{E}[\![e_1 * e_2]\!] &= \mathbf{MUL}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!]) \\
\mathcal{E}[\![e_1 / e_2]\!] &= \mathbf{DIV}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!]) \\
\mathcal{E}[\![e_1 \% e_2]\!] &= \mathbf{MOD}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!])
\end{aligned}$$

We'll assume that we have the types of expressions available to disambiguate array accesses from function applications. Given this, we can translate array indexing and function calls:

$$\begin{aligned}
\mathcal{E}[\![e_1\ e_2]\!] &= \mathbf{MEM}(\mathbf{ADD}(\mathcal{E}[\![e_1]\!], \mathbf{LSHIFT}(\mathcal{E}[\![e_2]\!], \mathbf{CONST}(2)))) && \text{(where } e_1 : t[\,]\text{)} \\
\mathcal{E}[\![f(e_1, \ldots, e_n)]\!] &= \mathbf{CALL}(\mathbf{NAME}(f), \mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!]) && \text{(where } f : t\texttt{->}t'\text{)}
\end{aligned}$$

For example, consider what happens when we translate the source expression `gcd(x, y-x)`. Using the rules we have so far, we get:

$$\mathcal{E}[\![\texttt{gcd}(\texttt{x}, \texttt{y} - \texttt{x})]\!] = \mathbf{CALL}(\mathbf{NAME}(\texttt{gcd}), \mathbf{TEMP}(\texttt{x}), \mathbf{SUB}(\mathbf{TEMP}(\texttt{y}), \mathbf{TEMP}(\texttt{x})))$$

5 Translating statements

The translation function $\mathcal{S}[\![s]\!]$ translates a statement s into a an IR statement.

$$\begin{aligned}
\mathcal{S}[\![x = e]\!] &= \mathbf{MOVE}(\mathcal{E}[\![x]\!], \mathcal{E}[\![e]\!]) \\
\mathcal{S}[\![e_1\ e_2 = e_3]\!] &= \mathbf{MOVE}(\mathcal{E}[\![e_1\ e_2]\!], \mathcal{E}[\![e_3]\!])
\end{aligned}$$

For statements that involve control, we need to generate labels and use **JUMP** statements. The statement labels should be *fresh*: not used anywhere else in the translation.

$$\begin{aligned}
\mathcal{S}[\![\texttt{if}\ (e)\ s]\!] = \mathbf{SEQ}(&\mathbf{CJUMP}(\mathcal{E}[\![e]\!], l_t, l_f), && \text{(Translation with } \texttt{else} \text{ is similar)} \\
&\mathbf{LABEL}(l_t), \mathcal{S}[\![s]\!], \\
&\mathbf{LABEL}(l_f))
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![\texttt{while}\ (e)\ s]\!] = \mathbf{SEQ}(&\mathbf{LABEL}(l_h), \mathbf{CJUMP}(\mathcal{E}[\![e]\!], l_e, l_t), \\
&\mathbf{LABEL}(l_t), \mathcal{S}[\![s]\!], \\
&\mathbf{JUMP}(\mathbf{NAME}(l_h)), \\
&\mathbf{LABEL}(l_e))
\end{aligned}$$

---

[1] Because it is always used on proper subterms on the right-hand side, the definition of $\mathcal{E}[\![e]\!]$ is a well-founded inductive definition.

6 Translating function definitions

We assume that there is a special $\mathbf{TEMP}$ for return values for functions, which we will write $\mathbf{TEMP}(RV)$ or just RV. Suppose that a function is written as $f(x_1\!:\!t_1, \ldots, x_n\!:\!t_n) = s$. The function body s is translated to an IR statement as:

$$\begin{aligned}&\mathbf{SEQ}(\mathbf{LABEL}(f),\\ &\qquad \mathcal{S}[\![s]\!],\\ &\qquad \mathbf{LABEL}(f_{epilogue}))\end{aligned}$$

The idea is that when the function is ready to return, the code will jump to the label $f_{epilogue}$, which is the function epilogue. It will do what is necessary with RV to get the result back to the caller. Therefore we translate `return` as follows:

$$\mathcal{S}[\![\texttt{return}]\!] = \mathbf{JUMP}(f_{epilogue})$$

$$\begin{aligned}\mathcal{S}[\![\texttt{return}\ e]\!] = \mathbf{SEQ}(&\mathbf{MOVE}(\mathbf{TEMP}(RV), \mathcal{E}[\![e]\!]),\\ &\mathbf{JUMP}(f_{epilogue}))\end{aligned}$$

7 Booleans and control flow

We might be tempted to translate the boolean conjunction operation as follows:

$$\mathcal{E}[\![e_1 \& e_2]\!] = \mathbf{AND}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!]) \qquad \text{(BAD!)}$$

This doesn't work because we need the & operator to short-circuit if the first argument is false. We can code this up explicitly using control flow, but the result is rather verbose:

$$\begin{aligned}\mathcal{E}[\![e_1 \& e_2]\!] = \mathbf{ESEQ}(\mathbf{SEQ}(&\mathbf{MOVE}(\mathbf{TEMP}(x), 0),\\ &\mathbf{CJUMP}(\mathcal{E}[\![e_1]\!], l_1, l_f),\\ &\mathbf{LABEL}(l_1), \mathbf{CJUMP}(\mathcal{E}[\![e_2]\!], l_2, l_f),\\ &\mathbf{LABEL}(l_2), \mathbf{MOVE}(\mathbf{TEMP}(x), 1)\\ &\mathbf{LABEL}(l_f)),\\ \mathbf{TEMP}(x))\ &\end{aligned}$$

Imagine applying this translation to the statement `if` $(e_1 \& e_2)$ s. We will obtain a big chunk of IR that combines all the IR nodes from this translation with those of `if`. Instead, here is a much shorter translation of `if` $(e_1 \& e_2)$ s:

$$\begin{aligned}&\mathbf{CJUMP}(\mathcal{E}[\![e_1]\!], l_1, l_f)\\ &\mathbf{LABEL}(l_1), \mathbf{CJUMP}(\mathcal{E}[\![e_2]\!], l_2, l_f)\\ &\mathcal{S}[\![s]\!]\\ &\mathbf{LABEL}(l_f)\end{aligned}$$

How can we get an efficient translation like this one? The key is to observe that the efficient translation never records the value of e_1 and e_2. Instead, it turns their results into immediate control flow decisions. The fact that we arrive at label l_1 means that e_1 is true. Because e_1 and e_2 appear only as children of **CJUMP** nodes, we will be able to generate more efficient code when we translate IR to assembly, too.

8 Translating booleans to control flow

Therefore, we develop a third syntax-directed translation, which we call $\mathcal{C}$ for "control". The specification of our translation is as follows:

> $\mathcal{C}[\![e, t, f]\!]$ is an IR statement that has all the side effects of e, and then jumps to label t if e evaluates to false, and to label f if e evaluates to true.

Given this spec, we can easily translate simple boolean expressions:

$$\begin{aligned}
\mathcal{C}[\![\texttt{true}, t, f]\!] &= \textbf{JUMP}(t) \\
\mathcal{C}[\![\texttt{false}, t, f]\!] &= \textbf{JUMP}(f) \\
\mathcal{C}[\![e, t, f]\!] &= \textbf{CJUMP}(\mathcal{E}[\![e]\!], t, f) && \text{(We use this rule if no other rule applies.)}
\end{aligned}$$

The payoff comes for boolean operators, e.g.:

$$\mathcal{C}[\![e_1 \& e_2, t, f]\!] = \textbf{SEQ}(\mathcal{C}[\![e_1, l_1, f]\!], \textbf{LABEL}(l_1), \mathcal{C}[\![e_2, l_2, f]\!])$$

We can then translate `if` and `while` more efficiently by using the $\mathcal{C}[\![\cdot]\!]$ translation instead:

$$\begin{aligned}
\mathcal{S}[\![\texttt{if}\ (e)\ s]\!] = \textbf{SEQ}(&\mathcal{C}[\![e, l_1, l_2]\!], \\
&\textbf{LABEL}(l_1), \mathcal{S}[\![s]\!], \\
&\textbf{LABEL}(l_2))
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![\texttt{while}\ (e)\ s]\!] = \textbf{SEQ}(&\textbf{LABEL}(l_h) \\
&\mathcal{C}[\![e, l_1, l_e]\!], \\
&\textbf{LABEL}(l_1), \mathcal{S}[\![s]\!], \\
&\textbf{JUMP}(l_h), \\
&\textbf{LABEL}(l_e))
\end{aligned}$$

Now when we translate `if` $(e_1 \& e_2)\ s$, the result is compact and efficient.