Translation into Stack/Register Machine

Outline

We first show that the AST of an arithmetic expression can be translated into a register machine.

After that, we give examples of how a function with local variables can be translated into a stack/register machine.

The machine uses registers for the evaluation of expressions, and the stack for storing local variables.

The stack machine is reasonably close to LLVM (a frequently used intermediate representation for C, C^{++} and `Rust`) that will appear later in the course, while at the same time it is not yet too complicated.

For the moment we use an untyped machine model, which only has `double`.

Output of Parser

The parser produces ASTs for expressions, which we have to translate. For example, the expression `3 + 2 * - ( x + 4 )` results in the following AST:

$$+(\,3, \star(\,2, -(\,+(\,x, 4\,))))\,.$$

In structured notation:

```
+
    3
    *
        2
        -
            +
                x
                4
```

Register Machine

We assume that we have an unbounded number of registers, which we call $\alpha_1, \alpha_2, \ldots$

For the moment, registers always have type `double`.

We allow instructions of the following forms:

$\alpha \leftarrow c$: Put the value of constant c in α.

$\alpha \leftarrow \textbf{load}\ v$: Load the value of local variable v into α.

$\alpha \leftarrow \textbf{op}\ \alpha_1$: Apply unary operator **op** on register α_1, and put the result in α.

$\alpha \leftarrow \textbf{op}\ \alpha_1,\ \alpha_2$: Apply binary operator **op** on α_1, α_2, and put the result in α.

In general, registers should not be reassigned. (The exact rules follow later with SSA.)

In order to translate `3 + 2 * - ( x + 4 )`, one needs to walk through the AST in left-right, depth-first order, and translate each node into a statement. The result is:

$$
\begin{aligned}
\alpha_1 &\leftarrow 3.0 \\
\alpha_2 &\leftarrow 2.0 \\
\alpha_3 &\leftarrow \text{load } x \\
\alpha_4 &\leftarrow 4.0 \\
\alpha_5 &\leftarrow \text{sum } \alpha_3,\ \alpha_4 \\
\alpha_6 &\leftarrow \text{minus } \alpha_5 \\
\alpha_7 &\leftarrow \text{mult } \alpha_2,\ \alpha_6 \\
\alpha_8 &\leftarrow \text{sum } \alpha_1,\ \alpha_7
\end{aligned}
$$

We give the translation algorithm a bit later in these slides.

Stack for Local Variables

Local variables are created and destroyed in LIFO order:

```
{
   double a = 3.0;
   {
      double b = 4.0;
      double c = b + b;
   } // c,b go out of scope.
} // a goes out of scope.
```

In order to create and destroy local variables, we use a stack.

Treatment of Variables

We introduce some commands for treatment of local variables:

alloc $\#n$: Create space for n doubles on the stack.

dealloc $\#n$: Remove the space for n doubles from the stack.

$\alpha \leftarrow$ load $\#i$: Load the variable that is currently at i-th position of the stack into register α.

store α, $\#i$: Write α into the variable that is currently at i-th position of the stack.

Calling a Function

When a function is called, we assume that the caller first creates a space for the return value on stack, and after that, pushes the arguments on the stack.

When the function starts, it can create more space for additional local variables.

When it returns, it writes the return value into the dedicated space, and deallocates the local variables and parameters.

Example: Iteratively Computing Factorial

Consider the following program for computing $n! = n(n-1)(n-2)\cdots 1$.

```
double fact( double n )
{
   double res = 1.0;
   while( n != 0 )
   {
      res = res * n;
      n = n - 1;
   }
   return res;
}
```

Translation of `fact` into Stack/Register Machine

Just before `fact` is called, stack layout is:

offset #0	offset #1
n	(space for return value)

At the beginning of `fact`, we create space for the local variable `res`, after which the stack layout is:

offset #0	offset #1	offset #2
res (unitialized)	n	(space for return value)

After that, `fact` computes the value, which results in:

offset #0	offset #1	offset #2
res ($n!$)	n	(space for return value)

We copy #0 into #2, deallocate 2 positions, and return.

Translation of `fact`

Create variable `res` and initialize it with 1 :

$$\text{alloc}\,\#1$$

$$\alpha_0 \leftarrow 1.0$$

$$\text{store}\ \alpha_0,\ \#0$$

The `while` loop:

$$
\begin{aligned}
\text{loop}: \quad & \alpha_1 \leftarrow \text{load}\,\#1 \\
& \alpha_2 \leftarrow 0.0 \\
& \text{ifeq } \alpha_1,\ \alpha_2,\ \text{goto exit} \\
& \alpha_3 \leftarrow \text{load}\,\#0 \\
& \alpha_4 \leftarrow \text{load}\,\#1 \\
& \alpha_5 \leftarrow \text{mult}\,\alpha_3,\ \alpha_4 \\
& \text{store } \alpha_5,\ \#0 \\
& \alpha_6 \leftarrow \text{load}\,\#1 \\
& \alpha_7 \leftarrow 1.0 \\
& \alpha_8 \leftarrow \text{minus } \alpha_6,\ \alpha_7 \\
& \text{store } \alpha_8,\ \#1 \\
& \text{goto loop}
\end{aligned}
$$

On exit, we copy local variable `res` into the result, and clean up `res` and `n`:

$$\begin{array}{ll} \text{exit}: & \alpha_9 \leftarrow \text{load}\,\#0 \\ & \text{dealloc}\ \#2 \\ & \text{store}\ \alpha_9,\ \#0 \\ & \text{return} \end{array}$$

Recursive Implementation of $n!$

```
double fact( double n )
{
   if( n == 0 )
      return 1.0;
   else
      return n * fact( n - 1 );
}
```

Translation of `fact` (recursive variant)

$\alpha_1 \leftarrow \text{load } \#0$

$\alpha_2 \leftarrow 0.0$

$\text{ifneq } \alpha_1, \alpha_2, \text{ goto } L_1$

$\alpha_3 \leftarrow 1.0$

$\text{store } \alpha_3, \#1$

$\text{dealloc } \#1$

return

Translation of `n * fact( n - 1 )`:

$$
\begin{aligned}
L_1: \quad & \alpha_4 \leftarrow \text{load}\,\#0 \\
& \alpha_5 \leftarrow \text{load}\,\#0 \\
& \alpha_6 \leftarrow 1.0 \\
& \alpha_7 \leftarrow \text{minus}\ \alpha_5,\ \alpha_6 \\
& \text{alloc}\ \#2 \\
& \text{store}\ \alpha_7,\ \#0 \\
& \text{call fact} \\
& \alpha_8 \leftarrow \text{load}\ \#0 \\
& \text{dealloc}\ \#1
\end{aligned}
$$

The result is now in α_8.

Compute the final multiplication `n*fact(n-1)` and return the result:

$\alpha_9 \leftarrow$ mult α_4, α_8

store α_9, #1

dealloc #1

return

Translation Function

We define a function
register **translate**(AST t)

It returns a register name α, and emits code that puts the value of t into α. We assume that **translate** has access to a container that contains the variable declarations.

Start by getting a new register name α.

- If t is a constant c, then emit $\alpha \leftarrow c$ and return α.
- If t is a variable v, then find out where v is located on the stack. Assume that the position is n. Emit $\alpha \leftarrow$ **load** $\#n$ and return α.

- If the AST has form $\mathbf{op}(t_1)$, then let $\alpha_1 = \mathbf{translate}(t_1)$. Emit $\alpha \leftarrow \mathbf{op}\ \alpha_1$, and return α.
- If the AST has form $\mathbf{op}(t_1, t_2)$, then let $\alpha_1 = \mathbf{translate}(t_1)$ and let $\alpha_2 = \mathbf{translate}(t_2)$. Emit $\alpha \leftarrow \mathbf{op}\ \alpha_1, \alpha_2$, and return α.
- Function **translate** can be easily generalized to operators of higher arity, but these are rare.

- If the AST has form $f(t_1, \ldots, t_n)$, where f is a function that is not built-in, first call

$$\alpha_1 = \textbf{translate}(t_1), \ldots, \alpha_n = \textbf{translate}(t_n).$$

After that, emit the following:

alloc $\#(n+1)$	create space for parameters and return value
store $\alpha_1, \#\,0$	
$\cdots$	store parameters in memory
store $\alpha_n, \#\,n$	
call f	
$\alpha \leftarrow$ load #0	load result in α
dealloc #1	

Some Possible Optimizations

Remove load/store sequences of the following form:

$$\text{store } \alpha_5, \#1$$
$$\alpha_6 \leftarrow \text{load } \#1$$

Remove repeated loads of the following form:

$$\alpha_4 \leftarrow \text{load } \#0$$
$$\alpha_5 \leftarrow \text{load } \#0$$

In general, try to detect and remove recomputations.

Try to move variables from the stack to registers as much as possible. (hard to detect)

Some More Possible Optimizations

Try to change computation order to minimize register use.

Try to identify the result position with a local variable in functions (RVA). For example, the first factorial function moves `res` into the result position when it returns. There is no reason, why `res` cannot be the result position from the beginning. It saves memory and one unnecessary copy.

Another Example: GCD

Consider the following program for computing the greatest common divisor of `n1` and `n2`:

```
double gcd( double d1, double d2 )
{
   while( d2 != 0 )
   {
      double d3 = d1 % d2;
      d1 = d2;
      d2 = d3;
   }
   return d1;
}
```

Translation of `gcd`

Before start, stack lay out is:

offset #0	offset #1	offset #2
d_1	d_2	(space for return value)

start : $\alpha_0 \leftarrow \text{load}\,\#1$

$\alpha_1 \leftarrow 0.0$

ifeq α_0, α_1, goto exit

alloc #1

We are now in the while loop, and just created `d3`:

offset #0	offset #1	offset #2	offset #3
d_3	d_1	d_2	(space for return value)

$$\alpha_2 \leftarrow \text{load}\,\#1$$

$$\alpha_3 \leftarrow \text{load}\,\#2$$

$$\alpha_4 \leftarrow \text{mod}\ \alpha_2,\ \alpha_3$$

$$\text{store}\ \alpha_4,\ \#0$$

$$\alpha_5 \leftarrow \text{load}\,\#2$$

$$\text{store}\,\alpha_5, \#\,1$$

$$\alpha_6 \leftarrow \text{load}\,\#0$$

$$\text{store}\,\alpha_6, \#\,2$$

Variable `d3` goes out of scope, and we restart the loop.

$$\begin{array}{ll} & \text{dealloc}\,\#1 \\ & \text{goto start} \end{array}$$

$$\begin{array}{ll} \text{exit}: & \text{load}\,\#0 \\ & \text{dealloc}\,\#2 \\ & \text{store}\,\alpha_8, \#0 \end{array}$$

Summary

We gave an intermediate representation. It is fairly realistic, but a bit simplified.

In reality, operations have different types, and registers have different types. (Only simple types, like `bool`,`char`,`int`,`double`.)

This means that values on the stack will have different sizes. It makes calculations of offsets a bit harder.

Later in the course, we will use a normal form called SSA.

If you look closely at how function calls are translated, you will understand why many C/C^{++} compilers evaluate arguments of function calls backwards.