Lecture 17: High-level code generation

David Hovemeyer

October 31, 2022

601.428/628 Compilers and Interpreters

High-level code generation

These slides present a few thoughts/recommendations about high-level code generation

Correctness, not efficiency

The goal of high-level code generation is to correctly represent the operations that each function will execute

You do *not* need to be concerned with generating *efficient* code

Storage allocation

- Each local variable and parameter should have storage allocated (add fields to `Symbol` to describe where this storage is)
- A scalar (integer and pointer) local variable whose address is never taken can have a virtual register (vreg) as its storage
- All other local variables (especially arrays and struct instances) will require storage in memory (in the stack frame)
    - The `StorageCalculator` class is intended to help you determine storage requirements (size and offset) for local variables allocated in a block of memory (i.e., the local variable area in the stack frame)

Annotate nodes with Operands

- To generate code for an expression node, annotate it with an `Operand` indicating the storage location where the value of the expression can be found
- For rvalues, this should be a freshly-allocated temporary register
- For lvalues, this could be a virtual register (if one has been allocated as the storage for a scalar local variable), or a memory reference operand (for a local variable allocated storage in memory, a pointer dereference, or a field reference)

Assignment

```
generated code for left subexpression
generated code for right subexpression

mov_sfx lhs_location, rhs_value
```

sfx is an operand size suffix (`b`, `w`, `l`, `q`) based on type of value being assigned.

lhs_location is the operand specifying the storage location for the left hand side lvalue.

rhs_value is a temporary virtual register storing the computed value of the expression on the right hand side. (This can also be the `Operand` representing the overall result of the assignment.)

Binary operators

```
generated code for left subexpression
generated code for right subexpression

op_sfx vrn, lhs_location, rhs_location
```

op_sfx is the high-level opcode corresponding to the operator. E.g., `add_l` if adding 32-bit integer values.

`vrn` is a temporary virtual register. This will also be the `Operand` representing the evaluation of the overall expression.

lhs_location and *rhs_location* are the temporary virtual registers holding the results of evaluating the left and right subexpressions.

Pointer operations

Idea: an lvalue whose storage is in memory is represented by an operand of the form $\boxed{\texttt{(vr}n\texttt{)}}$; i.e., vr*n* is a virtual register being used as a pointer, and the operand is a memory reference using this pointer.

Address-of: Taking the address of this lvalue means changing $\boxed{\texttt{(vr}n\texttt{)}}$ to $\boxed{\texttt{vr}n}$. I.e., we just want the pointer.

Dereference: If $\boxed{\texttt{vr}n}$ is a pointer, then dereferencing the pointer converts the operand to $\boxed{\texttt{(vr}n\texttt{)}}$. I.e., we want to refer to the memory the pointer is pointing to.

Arrays, subscript operations

Code for subscript operation:

generated code to find address of first element
generated code to find index

```
mul_q vroff, vridx, $eltsize
add_q vreltaddr, vrbase, vroff
```

vr*idx* is result of evaluating expression computing index. vr*off* is temporary virtual register to hold computed element offset. *eltsize* is the array element size. vr*base* is the virtual register holding the pointer to the first element of the array. vr*eltaddr* is the temporary virtual register holding the pointer to the element. The operand representing the element is $\boxed{(\texttt{vr}\mathit{eltaddr})}$.

Structs, field references

Similar to arrays. Location of struct instance is indicated by virtual register pointing to beginning of struct instance. For each field, you will need to know the *offset* of the field (from the beginning of the struct instance.)