CS415 Compilers

Code generation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

- Homework #7 will be posted this week.

- Project 2 is due on Monday, April 22. Removed GT and GEQ from language description, no need to change provided parse.y

- Final exam: May 14, noon – 3:00pm
  CONFLICTS?
A compiler is a lot of fast stuff followed by some hard problems

→ The hard stuff is mostly in code generation and optimization

→ For superscalars, its allocation & scheduling that is particularly important
The key code quality issue is holding values in registers

• When can a value be safely allocated to a register?
  → When only 1 name can reference its value (no aliasing)
  → Pointers, parameters, aggregates & arrays all cause trouble

• When should a value be allocated to a register?
  → When it is both safe & profitable

Encoding this knowledge into the IR (register-register model)

• Use code shape to make it known to every later phase
• Assign a virtual register to anything that can go into one
• Load or store the others at each reference

Relies on a strong register allocator
Recursive Treewalk vs. Ad-hoc SDT

int expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case $\times,\div,+,-$ :
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit(loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit(loadI, val(node), none, result);
            break;
    }
    return result;
}

Goal : Expr { $$ = $1; } ;
Expr: Expr PLUS Term
    { t = NextRegister();
        emit(add,$1,$3,t); $$ = t; }
    | Expr MINUS Term {…}
    | Term { $$ = $1; } ;
Term: Term TIMES Factor
    { t = NextRegister();
        emit(mult,$1,$3,t); $$ = t; ;
    | Term DIVIDES Factor {…}
    | Factor { $$ = $1; } ;
Factor: NUMBER
    { t = NextRegister();
        emit(loadI, val($1),none, t );
        $$ = t; }
    | ID
        { t1 = base($1);
            t2 = offset($1);
            t = NextRegister();
            emit(loadAO,t1,t2,t);
            $$ = t; }
Handling Assignment (just another operator)

\[ \text{ lhs } \leftarrow \text{ rhs } \]

**Strategy**
- Evaluate \( \text{ rhs } \) to a value (an rvalue)
- Evaluate \( \text{ lhs } \) to a location (memory address) (an lvalue)
  - \( \text{lvalue} \) is an address \( \Rightarrow \) store rhs
- If \( \text{ rvalue } \) & \( \text{lvalue} \) have different types
  - Evaluate \( \text{ rvalue } \) to its "natural" type
  - Convert that value to the type of \( \text{ lhs } \) value, if possible

Unambiguous scalars may go into registers (no aliasing)
Ambiguous scalars or aggregates go into memory (possible aliasing)
Handling Assignment

What if the compiler cannot determine the rhs’s type?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a _run-time_ check
- Add a _tag field_ to the data items to hold type information

Code for assignment becomes more complex

```plaintext
evaluate rhs
If lhs.type_tag ≠ rhs.type_tag then
    convert rhs to type(lhs) or
    signal a run-time error
lhs ← rhs
```

This is much more complex than if it knew the types
Handling Assignment

Compile-time type-checking
• Goal is to eliminate both the runtime check & the tag
• Determine, at compile time, the type of each subexpression
• Use compile-time types to determine if a run-time check is needed

Optimization strategy
• If compiler knows the type, move the check to compile-time
• Unless tags are needed for garbage collection, eliminate them
• If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)
Handling Assignment (with reference counting)

Garbage Collection

The problem with reference counting
• Must adjust the count on each pointer assignment
• Overhead is significant, relative to assignment

Code for assignment becomes

evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1

This adds 1+, 1-, 2 loads, & 2 stores

With extra functional units & large caches, this may become either cheap or free. What about power consumption?
How does the compiler handle $A[i,j]$?

First, must agree on a storage scheme

**Row-major order**
- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- $A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$

**Column-major order**
- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- $A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]$

**Indirection vectors**
- Vector of pointers to pointers to ... to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis
Laying Out Arrays

The Concept

Row-major order

Column-major order

Indirection vectors

These have distinct & different cache behavior
Computing an Array Address

Declaration: \( A[\text{low} .. \text{high}] \) of ...

\[ A[i] \]
- \( @A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: \( \text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1]) \)
Computing an Array Address

Declaration: A[low .. high] of ...

A[ i ]
- \( @A + (i - low) \times \text{sizeof}(A[1]) \)
- In general: base(A) + (i - low) \times \text{sizeof}(A[1])

int A[1:10] ⇒ low is 1
Make low 0 for faster access (saves a - )

Almost always a power of 2, known at compile-time ⇒ use a shift for speed
Computing an Array Address

Declaration: \( A[\text{low1 .. high1, low2 .. high2}] \) of ...

\[ A[i] \]
- \(@A + (i - \text{low}) \times \text{sizeof}(A[1])\)
- In general: \(\text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1])\)

What about \( A[i_1,i_2] \)?

**Row-major order, two dimensions**
\[ @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1]) \]

**Column-major order, two dimensions**
\[ @A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times \text{sizeof}(A[1]) \]

**Indirection vectors, two dimensions**
\[ *(A[i_1])[i_2] \quad \text{— where } A[i_1] \text{ is, itself, a 1-d array reference} \]
In row-major order
\[ @A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w \]

Which can be factored into
\[ @A + i \times (high_2 - low_2 + 1) \times w + j \times w \]
\[- (low_1 \times (high_2 - low_2 + 1) \times w) + (low_2 \times w) \]

If \( low_i, high_i, \) and \( w \) are known, the last term is a constant
Define \( @A_0 \) as
\[ @A - (low_1 \times (high_2 - low_2 + 1) \times w + low_2 \times w \]

And \( len_2 \) as \( (high_2 - low_2 + 1) \)

Then, the address expression becomes
\[ @A_0 + (i \times len_2 + j) \times w \]
Things to do and next class

Work on the project!

More code generation

Optimization: CSE

Procedure abstraction

Read EaC: Chapter 6.1 - 6.5