

Code Shape, Part II
Expressions, Assignment, Addresses
Comp 412

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Road Map for Class



First, look at code shape

- Consider implementations for several language constructs

Then, consider code generation

- Selection, scheduling, & allocation *(order dictated by Lab 3)*
- Look at modern algorithms & modern architectures
- Lab 3 will give you insight into scheduling

If we have time, introduce optimization

- Eliminating redundant computations *(as with DAGs)*
- Data-flow analysis, maybe SSA-form

Start out with code shape for expressions ...



Generating Code for Expressions

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
 - 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

- 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
- ILOC has textual "memory tags" on loads, stores, & calls
- ILOC has a hierarchy of loads & stores (*see the digression*)

Relies on a strong register allocator



Generating Code for Expressions

```

expr(node) {
  int result, t1, t2;
  switch (type(node)) {
    case x,+,+,- :
      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;
}

```

The Concept

- Assume an **AST** as input & **ILOC** as output
- Use a **postorder treewalk** evaluator (*visitor pattern* in OOD)
 - › Visits & evaluates children
 - › Emits code for the op itself
 - › Returns register with result
- Bury complexity of addressing names in routines that it calls
 - › **base()**, **offset()**, & **val()**
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow



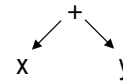
Generating Code for Expressions

```

expr(node) {
  int result, t1, t2;
  switch (type(node)) {
    case x,+,+,- :
      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 (loadl, val(node), none, result);
      break;
  }
  return result;
}

```

Example:



Produces:

```

expr("x") →
loadl    @x    ⇒ r1
loadAO   r_arp,r1 ⇒ r2
expr("y") →
loadl    @y    ⇒ r3
loadAO   r_arp,r3 ⇒ r4
NextRegister() → r5
Emit(add,r2,r4,r5) →
add      r2,r4 ⇒ r5

```



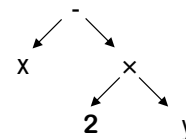
Generating Code for Expressions

```

expr(node) {
  int result, t1, t2;
  switch (type(node)) {
    case x,+,+,- :
      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 (loadl, val(node), none, result);
      break;
  }
  return result;
}

```

Example:



Produces:

```

loadl    @x    ⇒ r1
loadAO   r_arp,r1 ⇒ r2
loadl    2     ⇒ r3
loadl    @y    ⇒ r4
loadAO   r_arp,r4 ⇒ r5
mult     r3,r5 ⇒ r6
sub      R2,r6 ⇒ r7

```

Extending the Simple Treewalk Algorithm



More complex cases for IDENTIFIER

- What about values that reside in registers?
 - Modify the IDENTIFIER case
 - Already in a register \Rightarrow return the register name
 - Not in a register \Rightarrow load it as before, but record the fact
 - Choose names to avoid creating false dependences
- What about parameter values?
 - Many linkages pass the first several values in registers
 - Call-by-value \Rightarrow just a local variable with a negative offset
 - Call-by-reference \Rightarrow negative offset, extra indirection
- What about function calls in expressions?
 - Generate the calling sequence & load the return value
 - Severely limits compiler's ability to reorder operations

Extending the Simple Treewalk Algorithm



Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables
 - Original PL/I had asymmetric tables for BCD & binary integers

Typical
Table for
Addition

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

Extending the Simple Treewalk Algorithm



What about evaluation order?

- Can use commutativity & associativity to improve code
- This problem is truly hard

Local rather than global

Commuting operands at a single operation is much easier

- 1st operand must be preserved while 2nd is evaluated
- Takes an extra register for 2nd operand
- Should evaluate more demanding operand expression first

(Ershov in the 1950's, Sethi in the 1970's)

Taken to its logical conclusion, this creates Sethi-Ullman scheme for register allocation

[See Bibliography in EaC]

Generating Code in the Parser



Need to generate an initial IR form

- Chapter 4 talks about ASTs & ILOC
- Might generate an AST, use it for some high-level, near-source work such as type checking and optimization, then traverse it and emit a lower-level IR similar to ILOC for further optimization and code generation

The Big Picture

- Recursive algorithm really works bottom-up
 - Actions on non-leaves occur after children are done
- Can encode same basic structure into *ad-hoc* SDT scheme
 - Identifiers load themselves & stack virtual register name
 - Operators emit appropriate code & stack resulting VR name
 - Assignment requires evaluation to an *lvalue* or an *rvalue*

Ad-hoc SDT versus a Recursive Treewalk



<pre> expr(node) { int result, t1, t2; switch (type(node)) { case x, ÷, +, - : 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 (load, val(node), none, result); break; } return result; } </pre>	<pre> Goal : Expr { \$\$ = \$1; }; Expr: Expr PLUS Term { t = NextRegister(); emit(add,\$1,\$3,t); \$\$ = t; } Expr MINUS Term {...} Term: Term TIMES Factor { t = NextRegister(); emit(mult,\$1,\$3,t); \$\$ = t; }; Term DIVIDES Factor {...} Factor: Factor { \$\$ = \$1; }; NUMBER ID { t = NextRegister(); emit(load, val(\$1), none, t); \$\$ = t; } </pre>
---	---

Handling Assignment (just another operator)



$lhs \leftarrow rhs$

Strategy

- Evaluate *rhs* to a **value** (an *rvalue*)
- Evaluate *lhs* to a **location** (an *lvalue*)
 - *lvalue* is a register \Rightarrow move *rhs*
 - *lvalue* is an address \Rightarrow store *rhs*
- If *rvalue* & *lvalue* have different types
 - Evaluate *rvalue* to its "natural" type
 - Convert that value to the type of **lvalue*

Unambiguous scalars go into registers
 Ambiguous scalars or aggregates go into memory

Let hardware sort out the addresses!



Handling Assignment

What if the compiler cannot determine the type of the rhs?

- This is a property of the language & the specific program
- For type-safety, compiler must insert a run-time check
 - Some languages & implementations ignore safety (*bad idea*)
- Add a *tag* field to the data items to hold type information
 - Explicitly check tags at runtime

Code for assignment becomes more complex

```

evaluate rhs
if type(lhs) ≠ rhs.tag
  then
    convert rhs to type(lhs) or
    signal a run-time error
lhs ← rhs

```

Choice between conversion & a runtime exception depends on details of language & type system

Much more complex than static checking, plus costs occur at runtime rather than compile time



Handling Assignment

Compile-time type-checking

- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

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

Can *design* the language so all checks are static



Handling Assignment with Reference Counts

Reference counting is an incremental strategy for implicit storage deallocation *(alternative to batch collectors)*

- Simple idea
 - Associate a count with each heap allocated object
 - Increment count when pointer is duplicated
 - Decrement count when pointer is destroyed
 - Free when count goes to zero
- Advantages
 - Bounded costs incurred at predictable times
 - Useful in real-time applications, user interfaces
 - Counts will be in cache, ILP may reduce expense
- Disadvantages
 - Bounded costs on every pointer assignment



Handling Assignment with Reference Counts

Implementing reference counts

- Must adjust the count on each pointer assignment
- Extra code on every counted (e.g., pointer) assignment

Code for assignment becomes

```

evaluate rhs
lhs->count--
lhs ← addr(rhs)
rhs->count++
if (rhs->count = 0)
  free rhs
  
```

Likely hits in L1 cache

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

With extra functional units & large caches, the overhead may become either cheap or free ...





How does the compiler handle $A[i,j]$?

First, must agree on a storage scheme

Row-major order

(most languages)

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

(Fortran)

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

(Java)

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

A	1,1	1,2	1,3	1,4
	2,1	2,2	2,3	2,4

These can have distinct & different cache behavior

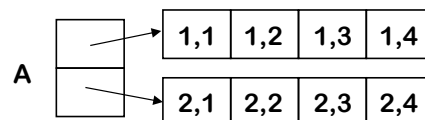
Row-major order

A	1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4
---	-----	-----	-----	-----	-----	-----	-----	-----

Column-major order

A	1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4
---	-----	-----	-----	-----	-----	-----	-----	-----

Indirection vectors





Computing an Array Address

$A[i]$

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



Computing an Array Address

$A[i]$

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

$\text{int } A[1:10] \Rightarrow \text{low is } 1$
Make low 0 for faster
access (saves a -)

Almost always a power of
2, known at compile-time
 \Rightarrow use a shift for speed



Computing an Array Address

$A[i]$

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

What about $A[i_1, i_2]$?

This stuff looks expensive!
Lots of implicit +, -, x ops

Row-major order, two dimensions

$$@A + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times sizeof(A[1])$$

Column-major order, two dimensions

$$@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times sizeof(A[1])$$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

e.g., $@A + (i_1 - low) \times sizeof(A[1])$



Optimizing Address Calculation for $A[i, j]$

In row-major order

where $w = sizeof(A[1,1])$

$$@A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w$$

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

$$@A_0 + (i \times len_2 + j) \times w$$

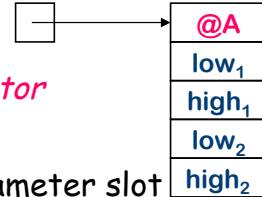
Compile-time constants



Array References

What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters



- Need dimension information \Rightarrow build a *dope vector*
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

Some improvement is possible

- Save len_i and low_i rather than low_i and $high_i$
- Pre-compute the fixed terms in prologue sequence

What about call-by-value?

- Most c-b-v languages pass arrays by reference
- This is a language design issue



Array References

What about $A[12]$ as an actual parameter?

If corresponding parameter is a scalar, it's easy

- Pass the address or value, as needed
- Must know about both formal & actual parameter
- Language definition must force this interpretation

What is corresponding parameter is an array?

- Must know about both formal & actual parameter
- Meaning must be well-defined and understood
- Cross-procedural checking of conformability

\Rightarrow Again, we're treading on language design issues



Array References

What about variable-sized arrays?

Local arrays dimensioned by actual parameters

- Same set of problems as parameter arrays
- Requires dope vectors (or equivalent)
 - dope vector at fixed offset in activation record

⇒ Different access costs for textually similar references

This presents a lot of opportunity for a good optimizer

- Common subexpressions in the address polynomial
- Contents of dope vector are fixed during each activation
- Should be able to recover much of the lost ground

⇒ Handle them like parameter arrays



Example: Array Address Calculations in a Loop

```
DO J = 1, N
  A[I,J] = A[I,J] + B[I,J]
END DO
```

A, B are declared as conformable
floating-point arrays
floatsize is sizeof(A[1,1])

- **Naïve:** Perform the address calculation twice

```
DO J = 1, N
  R1 = @A0 + (J × len1 + I) × floatsize
  R2 = @B0 + (J × len1 + I) × floatsize
  MEM(R1) = MEM(R1) + MEM(R2)
END DO
```

Code generated by a
translator will almost
certainly work this way.
(treewalk code generator)
Imagine a 5 point stencil:

$$A[I,J] = 0.2 * (A[I-1,J] + A[I,J] + A[I+1,J] + A[I,J-1] + A[I,J+1])$$

Example: Array Address Calculations in a Loop



```
DO J = 1, N
  A[I,J] = A[I,J] + B[I,J]
END DO
```

- **More sophisticated:** Move common calculations out of loop

```
R1 = I × floatsize
c = len1 × floatsize ! Compile-time constant
R2 = @A0 + R1
R3 = @B0 + R1
DO J = 1, N
  a = J × c
  R4 = R2 + a
  R5 = R3 + a
  MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

Example: Array Address Calculations in a Loop



```
DO J = 1, N
  A[I,J] = A[I,J] + B[I,J]
END DO
```

- **Very sophisticated:** Convert multiply to add

```
R1 = I × floatsize
c = len1 × floatsize ! Compile-time constant
R2 = @A0 + R1 ; R3 = @B0 + R1
DO J = 1, N
  R2 = R2 + c
  R3 = R3 + c
  MEM(R2) = MEM(R2) + MEM(R3)
END DO
```

J is now bookkeeping

A good compiler
would rewrite the
end-of-loop test to
operate on R2 or R3

(Linear function test
replacement)



Representing and Manipulating Strings

Character strings differ from scalars, arrays, & structures

- Fundamental unit is a character
 - Typical sizes are one or two bytes
 - Target ISA may (or may not) support character-size operations
- Set of supported operations on strings is limited
 - Assignment, length, concatenation, translation (?)
- Efficient string operations are complex on most RISC ISAs
 - Ties into representation, linkage convention, & source language

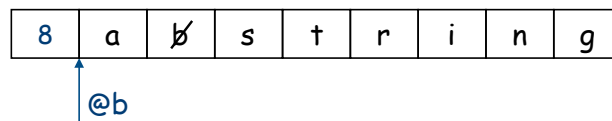
Subword data



Representing and Manipulating Strings

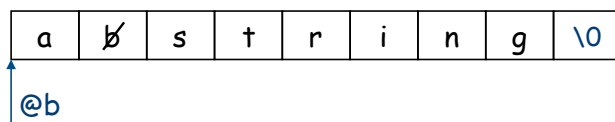
Two common representations

- Explicit length field



Length field may take more space than terminator

- Null termination



- Language design issue

– Fixed-length versus varying-length strings (1 or 2 length fields)



Representing and Manipulating Strings

Each representation as advantages and disadvantages

Operation	Explicit Length	Null Termination
Assignment	Straightforward	Straightforward
Checked Assignment	Checking is easy	Must count length
Length	$O(1)$	$O(n)$
Concatenation	Must copy data	Length + copy data

Unfortunately, null termination is almost considered normal

- Hangover from design of C
- Embedded in OS and API designs



Manipulating Strings

Single character assignment

- With character operations
 - Compute address of rhs, load character
 - Compute address of lhs, store character
- With only word operations *(>1 char per word)*
 - Compute address of word containing rhs & load it
 - Move character to destination position within word
 - Compute address of word containing lhs & load it
 - Mask out current character & mask in new character
 - Store lhs word back into place



Manipulating Strings

Multiple character assignment

Two strategies

1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

- With character operations

1. Easy to generate; inefficient use of resources
2. Harder to generate; better use of resources

Requires explicit code to check for buffer overflow (⇒ length)

- With only word operations

1. Lots of complication to generate; inefficient at runtime, too
2. Fold complications into end case; reasonable efficiency

Source & destination aligned differently
⇒ much harder cases for word operations



Manipulating Strings

Concatenation

- String concatenation is a length computation followed by a pair of whole-string assignments
 - Touches every character
- Exposes representation issues
 - Is string a descriptor that points to text?
 - Is string a buffer that holds the text?
 - Consider $a \leftarrow b || c$
 - Compute $b || c$ and assign descriptor to a ?
 - Compute $b || c$ into a temporary & copy it into a ?
 - Compute $b || c$ directly into a ?
- What about call $fee(b || c)$?

Manipulating Strings



Length Computation

- Representation determines cost
 - Explicit length turns `length(b)` into a memory reference
 - Null termination turns `length(b)` into a loop of memory references and arithmetic operations
- Length computation arises in other contexts
 - Whole-string or substring assignment
 - Checked assignment (buffer overflow)
 - Concatenation
 - Evaluating call-by-value actual parameter or concatenation as an actual parameter