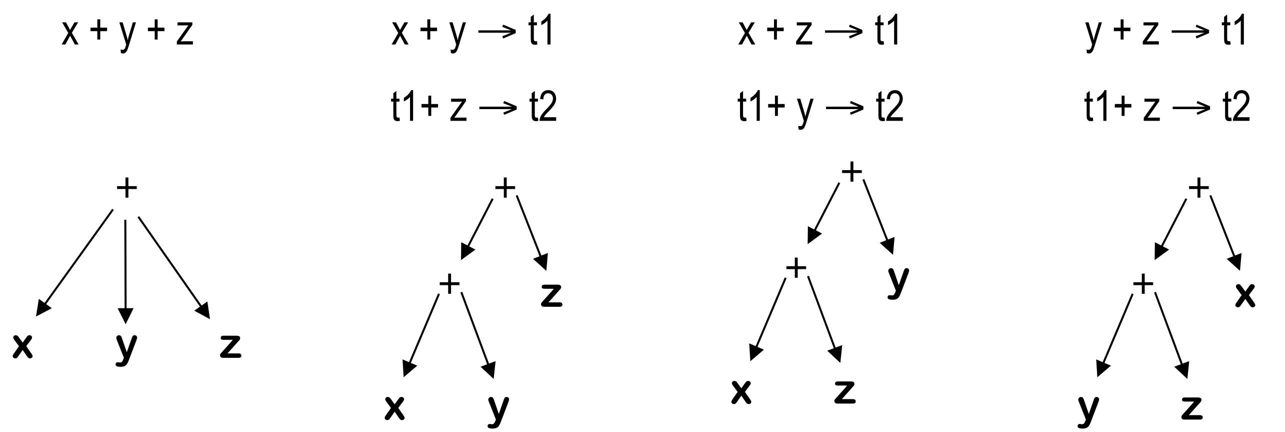
Code Shape I
Expressions & Arrays

Code Shape



- What if x is 2 and z is 3?
- What if y+z is evaluated earlier?

Addition is commutative & associative for integers

The "best" shape for x+y+z depends on contextual knowledge

→ There may be several conflicting options

Code Shape

Another example -- the case statement

- Implement it as cascaded if-then-else statements
  - → Cost depends on where your case actually occurs
  - → O(number of cases)
- Implement it as a binary search
  - → Need a dense set of conditions to search
  - → Uniform (log n) cost
- Implement it as a jump table
  - → Lookup address in a table & jump to it
  - → Uniform (constant) cost

Compiler must choose best implementation strategy

No amount of massaging or transforming will convert one into another

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

Relies on a strong register allocator

Generating Code for Expressions

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

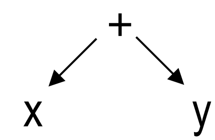
The concept

- **Use a simple treewalk evaluator**
- **Bury complexity in routines it calls**
  - *base*(), *offset*(), & *val*()
- **Implements expected behavior**
  - **Visits & evaluates children**
  - **Emits code for the op itself**
  - **Returns register with result**
- **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 ×,÷,+,- :
           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;
 }
```

Example:



Produces:

expr("x") →

loadI	**@x**	**⇒ r1**
loadAO	**r0, r1**	**⇒ r2**

expr("y") →

loadI	**@y**	**⇒ r3**
loadAO	**r0, r3**	**⇒ r4**

expr("+") →

NextRegister() → r5

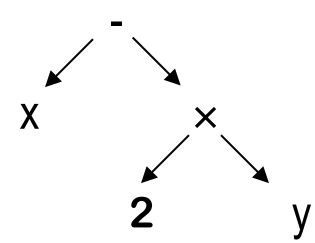
emit(add,r2,r4,r5) →

add	**r2, r4**	**⇒ r5**

Generating Code for Expressions

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

Example:



Generates:

loadI	**@x**	**⇒ r1**
loadAO	**r0, r1**	**⇒ r2**
loadI	**2**	**⇒ r3**
loadI	**@y**	**⇒ r4**
loadAO	**r0,r4**	**⇒ r5**
mult	**r3, r5**	**⇒ r6**
sub	**r2, r6**	**⇒ r7**

Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values in registers?
  - Modify the **IDENTIFIER** case
  - Already in a register ⇒ return the register name
  - Not in a register ⇒ 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 ⇒ just a local variable with "funny" offset
  - Call-by-reference ⇒ needs an 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

Typical Addition Table

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

Handling Assignment (just another operator)

lhs ← *rhs*

Strategy

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

Let hardware sort out the addresses !

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

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

```
evaluate rhs
if type(lhs) ≠ rhs.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 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

Can design the language so all checks are static

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 have distinct & different cache behavior

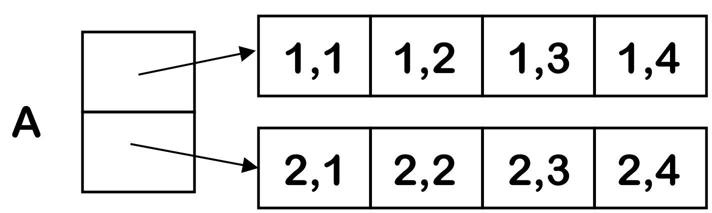
Row-major order

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 - low) x sizeof(A[1])
- In general: base(A) + (i - low) x sizeof(A[1])

Computing an Array Address

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x 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

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x 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

Optimizing Address Calculation for A[i,j]

In row-major order

where $w = \text{sizeof}(A[1,1])$

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

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

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

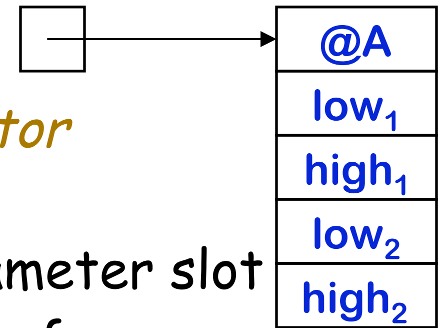
Compile-time constants

Array References

What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Need dimension information ⇒ 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



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 if corresponding parameter is an array?

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

⇒ Again, we're treading on language design issues

Example: Array Address Calculations in a Loop

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

- Naïve: Perform the address calculation twice

DO J = 1, N

$R1 = @A_0 + (J \times len_1 + I) \times floatsize$

$R2 = @B_0 + (J \times len_1 + I) \times floatsize$

MEM(R1) = MEM(R1) + MEM(R2)

END DO

Example: Array Address Calculations in a Loop

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

- Sophisticated: Move comon calculations out of loop

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