Handling Assignment

Comp 412

Copyright 2019, 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.
What is Left in Translation?

We have talked about the mechanism of translation. Next, we need to focus on the product, of translation.

Expressions
- Type-related issues in expressions
  - Mixed-type operations and value conversions.
- Assignment (including addressability)
- More operators in expressions
  - Boolean expressions, relational expressions, string operations

Control Flow
- Code shape and SDT strategies

Procedure and Function Calls
- Functionality: what, precisely, does a procedure call do?
- Implementation: how does the compiler emit code to do that?
Mixed-Type Expressions

What if the operands to an operation have different types?

• **Key observation:** the language must define the behavior

• Might simply be an error, detected during semantic elaboration
  – Require the programmer to “fix” it  
    *(insert a cast?)*

• Might require the compiler to insert an implicit conversion

Implicit conversions

• Compiler uses a conversion table to determine type of result
  – Convert arguments to result type & perform operation

• Most languages have symmetric & rational conversion tables

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th>Real</th>
<th>Double</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Real</td>
<td>Real</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
</tr>
</tbody>
</table>

Typical Conversion Table for Addition

*(note the symmetry)*
Mixed-Type Expressions

What if the operands to an operation have different types?

• **Key observation:** the language must define the behavior
• Might simply be an error, detected during semantic elaboration
  – Require the programmer to “fix” it (insert a cast?)
• Might require the compiler to insert an implicit conversion

Explicit conversions

• Many languages let the user specify explicit conversions, or *casts*
• Compiler needs a method to convert between types
  – Most of these methods are straightforward

Casts are conceptually more complex with objects

• We will return to this issue (briefly) when we discuss procedure calls

The key issue with mixed-type expressions is that the compiler must know the type of every subexpression.
Handling Assignment

PLs treat assignment as either a statement or an operator. The difference is whether it returns a value.

\[ x \leftarrow y \leftarrow z; \]

\( lhs \leftarrow rhs \)

- Evaluate \( rhs \) to a **value** (an rvalue)
- Evaluate \( lhs \) to a **location** (an lvalue)
  - **lvalue** is a register \( \Rightarrow \) move \( rhs \) into the register
  - **lvalue** is an address \( \Rightarrow \) store \( rhs \) into memory

Some values live in registers, others live in memory

- That decision is made during storage allocation
  - Unambiguous scalar values typically go into VRs
- Lifetime, storage class, & knowledge all factor into that decision

Unambiguous: a value is ambiguous if the running code can access it in multiple ways.

Examples:
- A pointer-based value
- A name whose address is taken (\&x in C)
- A call-by-reference parameter
Handling Assignment

Type information factors into the implementation of assignment

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

- What if \( \text{lhs} \) and \( \text{rhs} \) are distinct types?
- The programming language must specify the correct behavior.
- Typical rule is the same as an arithmetic operator
  - Evaluate \( \text{rhs} \) to its natural type
  - Convert \( \text{rhs} \) to the type of \( \text{lhs} \)
  - Language supplies a conversion table

### Typical Conversion Table for Assignment

<table>
<thead>
<tr>
<th>+</th>
<th>Integer</th>
<th>Real</th>
<th>Double</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Real</td>
<td>Real</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
</tr>
</tbody>
</table>
Handling Assignment

Type information factors into the implementation of assignment

\( \text{lhs} \leftarrow \text{rhs} \)

• What if the compiler cannot determine the type of \( \text{lhs} \) and/or \( \text{rhs} \) ?
  – Must store a runtime type tag with the value
  – Must emit code to compute types alongside operations
  – Must emit code to check compatibility and to perform conversions

• Wow. That sounds expensive.
  – It is. That is why PLs should have sound & complete type systems

```
evaluate rhs
if type(lhs) \neq rhs.tag then
  convert rhs to type(lhs) or throw an exception
lhs \leftarrow 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

Alternative is to allow unsafe programs to execute. (Bad idea)
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 and should *design* the language so all checks are static
Handling Assignment

For any lhs that is not in a register, the compiler must emit code to access its value. It needs an access method for each value.

Many interesting values cannot be kept in a register

• Strings, arrays, structures, objects
• Global and static values
• Known values that are too large

Aggregate Value (arrays, structures/records, strings, objects)

• Compiler needs to know a starting address
• Compiler needs a method to compute an offset
• Compiler needs to understand size & type

From this information, it emits code to compute the runtime address

Factor | ( Expr )
| number
| Reference

The grammar might combine all those cases in “Reference”
Access Method for an Array Reference

The General Scheme for an Array Reference

• Ensure base address is available
• Compute the offset within the array
• Add these two values

\[
A[i] = @A + (i - low) \times \text{sizeof}(A[1])
\]

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

@A is the base address of A.
Depending on how A is declared, @A may be
• an offset from the pointer to the procedure’s data area (its ARP),
• an offset from some global label, or
• an arbitrary address.
The first two are compile time constants.

\( \text{low} \) | Lower bound on index
\hline

Color Code: Invariant Varying
Computing an Address for an Array Element

A[ i ]

• @A + ( i – low ) x sizeof(A[1])

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

Note, for the record, that a naïve vector reference turns into a subtract, a shift, and an address-offset load (e.g., loadAO) — three operations before optimization.

If low is zero, the subtract goes away.
This sequence is amazingly cheap.
Scheme to Generate Code For A[i,j]?

**Compiler must generate the runtime address of the element A[i,j]**

- The compiler needs to know where A begins
  - Storage layout and addressability
    - *assume that @A is the address of A’s first element*
  - Tie between compile-time knowledge and runtime behavior

- The compiler must have a plan for the internal layout of A
  - Programming language usually dictates array element layout
  - Three common choices
    - *Row-major order*
    - *Column-major order*
    - *Indirection vectors*

- And a method to calculate the address of an array element
  - *General scheme: compute offset; add base Address; issue load (rvalue) or store (lvalue)*

**The Concept**

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td></td>
</tr>
</tbody>
</table>

A
Array Layout

**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]

**Storage Layout**

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

**Stride One Access**

```
for ( i = 0; i < n; i++)
  for ( j = 0; j < n; j++)
    A[i][j] = 0;
```

Declared arrays in most languages that have contiguous arrays.

Python adds a row at a time ...

Stride one access: successive references in the loop are to successive locations in the level one (or L1) data cache.

Stride one access maximizes spatial reuse. It plays well with hardware strategies that always fetch the next cache line (Intel).

Hardware prefetch units try to catch the non-stride-one access patterns.
Array Layout

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]

Storage Layout

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

Stride One Access

```plaintext
for ( j = 0; j < n; j++)
  for ( i =0; i < n; i++)
    A[i][j] = 0;
```

The Concept

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

All arrays in FORTRAN
Array Layout

Indirection Vectors
• Vector of pointers to pointers to ... to values
• Takes much more space, trades indirection for arithmetic
• Not amenable to analysis

Storage Layout

Stride One Access
No reference pattern guarantees stride one access in cache, unless rows are contiguous

Arrays in Java
```java
int **array;
```
in C
```
```
Computing an Address for an Array Element

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 \( A[i_1,i_2] \) ?

\[ \text{Row-major order, two dimensions} \]

\[ @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1,1]) \]

\begin{array}{|c|c|c|c|}
\hline
i_1 & i_2 & \text{low}_1 & \text{high}_1 \\
\hline
1 & 1 & 1 & 5 \\
2 & 1 & 1 & 5 \\
3 & 1 & 1 & 5 \\
4 & 1 & 1 & 5 \\
5 & 1 & 1 & 5 \\
\hline
\end{array}

\begin{array}{|c|c|c|}
\hline
i_1 & i_2 & \text{low}_2 & \text{high}_2 \\
\hline
1 & 1 & 1 & 4 \\
2 & 1 & 1 & 4 \\
3 & 1 & 1 & 4 \\
4 & 1 & 1 & 4 \\
5 & 1 & 1 & 4 \\
\hline
\end{array}

\( (i - \text{low}_1) \) is 3
\( (\text{high}_2 - \text{low}_2 + 1) \) is 4
3 x 4 is 12
\( i_2 - \text{low}_2 \) is 2

Combined, the 2 terms take us to the start of \( A[4,3] \)

Address computation took 3 adds, 3 subtracts, 1 multiply and 1 shift
Cheap in comparison to the register-save costs of a function call.
Array Layout

Row-Major Order
• Lay out as a sequence of consecutive rows
• Rightmost subscript varies fastest

Storage Layout

Address Polynomial for $A[i,j]$

$$@A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1,1])$$

Why does C start arrays indices at zero?
To avoid those subtractions

This polynomial follows *Horner’s rule* for evaluation.
Array Layout

**Column-Major Order**
- Lay out as a sequence of columns
- Leftmost subscript varies fastest

**Storage Layout**

\[
\begin{array}{cccccccc}
A & 1,1 & 2,1 & 1,2 & 2,2 & 1,3 & 2,3 & 1,4 & 2,4 \\
\end{array}
\]

**Address Polynomial for $A[i,j]$**

@A + \((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1\) \times \text{sizeof}(A[1,1])

This polynomial follows *Horner’s rule* for evaluation.
Array Layout

Indirection Vectors

• Vector of pointers to pointers to ... to values
• Takes much more space, trades indirection for arithmetic
• Not amenable to analysis

Storage Layout

Address Polynomial for \( A[i,j] \)

\[ *(A[i_1])[i_2] \] — where each of the \([i]\)’s is, itself, a 1-d array reference

Back when systems supported 1 or 2 cycle “indirect load” operations, this scheme was efficient. It replaces a multiply & an add with an indirect load.
Generating Better Code for $A[i,j]$  

In row-major order

\[
@A + ((i - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + j - \text{low}_2) \times w
\]

Can be re-distributed to

\[
@A + (i - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w
\]

Which can be re-distributed to

\[
@A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w
\]

\[-(\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) - (\text{low}_2 \times w)\]

If $\text{low}_i$, $\text{high}_i$, and $w$ are known, the last term is a constant.

Define $@A_0$ as

\[ @A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w - \text{low}_2 \times w) \]

And $\text{len}_2$ as $(\text{high}_2 - \text{low}_2 + 1)$

Then, the address expression becomes

\[ @A_0 + (i \times \text{len}_2 + j) \times w \]
How Did We Know To Do That?

Accumulated wisdom

Certain simple optimizations are performed as the quadruples are constructed. For example, an integer multiplication by a constant power of two (e.g., I*4 or I*16) is replaced by a left-shift operation. An exponentiation involving an integer constant power (e.g., A**9) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, -(B-C) becomes C-B. Finally, constants employed in a subscript expression [for example, each number 7 in A(7,1-7,1+7)] are often extracted from the subscript, evaluated as constant offsets from the start of the subscripted array, and combined into an aggregate constant offset which does not require computation during execution.

(p. 662; description of the “old” compiler, e.g., Medlock and Lowry)

The standard FORTRAN compiler evaluates subscripted array references in six steps. First, numerical constants embedded in the subscripts are extracted, evaluated, and combined into an aggregate constant subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the subscript and combined to form the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript. For an aggregate constant subscript in the range 0-4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction. Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element. This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction.

(p. 665; describing the new compiler)

Accumulated wisdom

Certain simple optimizations are performed as the quadruples are constructed. For example, an integer multiplication by a constant power of two (e.g., \(I \times 4\) or \(I \times 16\)) is replaced by a left-shift operation. An exponentiation involving an integer constant power (e.g., \(A^{**9}\)) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, \((B - C)\) becomes \(C - B\).

Finally, constants employed in a subscript expression (for example, each number 7 in \(A(7,1-7,1+7)\)) are often extracted from the subscript, evaluated as constant offsets from the start of the subscripted array, and combined into an aggregate constant offset during execution.

(p. 662; description of the "old" compiler, e.g., Medlock and Lowry)

The standard FORTRAN compiler evaluates subscripted array references in six steps. First, numerical constants embedded in the subscript are extracted, evaluated, and combined into an aggregate constant subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript. For an aggregate constant subscript in the range 0-4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction. Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element. This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction.

(p. 665, describing the new compiler)

Algebraic Reassociation of Expressions

This same kind of algebraic manipulation can improve other expressions

• Use the commutative, associative, & distributive properties
• Can achieve goals such as reducing demand for registers, exposing more common subexpressions, exposing more loop-invariant expressions for code motion, and reducing total operation count

Automating this kind of manipulation is harder than it looks.

... the symbolic coding generated is at least comparable to the results of hand coding. Other examples, however, could disclose the limitations of the algorithm. Its inability to apply the associative laws may result in unnecessary mode conversions and the storage of partial results in computing sums or products of quantities of unlike modes. Its inability to recognize equivalent subexpressions containing subscripted variables is a more serious drawback, and more nearly intrinsic to the algorithm. Finally, no provision has been made to recognize integral constant exponents. Most existing compilers waste time extravagantly by using exp(2 x ln(x)) to compute x ↑ 2.


No algorithm for this problem was published until the 1990s.