Compiler Construction

Chapter 7: Code Shape

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Code shape

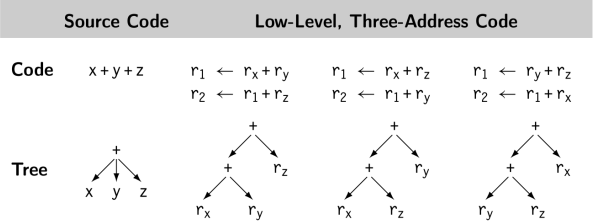
Translation choices

- Runtime speed
- Memory requirement
- Register optimization

For example, how should we translate the `switch` statement in C?

- A series of if-else
- Array access
- Hashing

Translation choices



■ **FIGURE 7.1** Alternate Code Shapes for **x + y + z**.

How should we translate `x+y+z` into 3-address codes?

- `r1 = rx + ry; r2 = r1 + rz;`
- `r1 = rx + rz; r2 = r1 + ry;`
- `r1 = ry + rz; r2 = r1 + rx`

What if the expression is `x+2+3`?

Arithmetic operators

- Base-offset for variables
- Immediate for constants
- Function call
- Automatic type conversion
- Assignment

Offset

Given an activation record with the base address in `r_arp`, we may load variable `a` into a temporary register `r_a` by

```
loadI    @a          -> r_1
loadAO   r_arp, r_1 -> r_a
```

Arithmetic operators

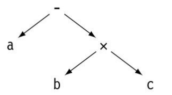

```
expr(node) {
    int result, t1, t2;
    switch(type(node)) {

        case ×, ÷, +, -:
            t1 ← expr(LeftChild(node));
            t2 ← expr(RightChild(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;

        case NAME:
            entry  ← STLookup(node);
            result ← ValueIntoReg(entry);
            break;

        case NUMBER:
            num   ← NumberFromNode(node);
            result ← NumberIntoReg(num);
            break;
    }
    return result;
}
```

(a) Treewalk Code Generator



(b) Abstract Syntax Tree for

a - b x c

loadI	@a	$\Rightarrow$	r_1
loadAO	r_{arp}, r_1	$\Rightarrow$	r_2
loadI	@b	$\Rightarrow$	r_3
loadAO	r_{arp}, r_3	$\Rightarrow$	r_4
loadI	@c	$\Rightarrow$	r_5
loadAO	r_{arp}, r_5	$\Rightarrow$	r_6
mult	r_4, r_6	$\Rightarrow$	r_7
sub	r_2, r_7	$\Rightarrow$	r_8

(c) Naive Code

■ **FIGURE 7.2** Simple Treewalk Code Generator for Expressions.

Other concerns in arithmetic operations

Commutativity, associativity, and number systems

- Common subexpression help improving the code quality
- However, floating-point operations should NOT be re-ordered

Function calls in an expression

- If the return value is put in a register,
- The change in evaluation order may have side effects to the call

Solution: activation record

Other concerns in arithmetic operations

Mixed-type expressions

- The compiler must insert the conversion code

+	int	float
int	int	float
float	float	float

Conversion Table for +

Assignment operator

1. Evaluate the right-hand side of the assignment to a **value**
2. Evaluate the left-hand side of the assignment to a **location**
3. Store the right-hand side value into the left-hand side location

- R-value: an expression is evaluated to a **value**
- L-value: an expression is evaluated to a **location**

Reducing demand for registers

```
loadI   @a        => r1
loadAO  rarp, r1  => r2
loadI   @b        => r3
loadAO  rarp, r3  => r4
loadI   @c        => r5
loadAO  rarp, r5  => r6
mult    r4, r6    => r7
sub     r2, r7    => r8
```

(a) Code from Fig. 7.2(c)

```
loadI   @a        => r1
loadAO  rarp, r1  => r1
loadI   @b        => r2
loadAO  rarp, r2  => r2
loadI   @c        => r3
loadAO  rarp, r3  => r3
mult    r2, r3    => r2
sub     r1, r2    => r2
```

(b) Code After Register Allocation

```
loadI   @c        => r1
loadAO  rarp, r1  => r2
loadI   @b        => r3
loadAO  rarp, r3  => r4
mult    r2, r4    => r5
loadI   @a        => r6
loadAO  rarp, r6  => r7
sub     r7, r5    => r8
```

(c) Evaluate b × c First

```
loadI   @c        => r1
loadAO  rarp, r1  => r1
loadI   @b        => r2
loadAO  rarp, r2  => r2
mult    r1, r2    => r1
loadI   @a        => r2
loadAO  rarp, r2  => r2
sub     r2, r1    => r1
```

(d) Code After Register Allocation

FIGURE 7.3 Rewriting a - b x c to Reduce Demand for Registers.

Accessing parameter values

Call-by-value

- Similar to local variable, using AR

Call-by-reference

- Save the address (pointer) into the AR. The retrieval needs 2 dereferencing steps
- Passing parameter d

```
loadI  @d          -> r1
loadAO r_arp, r_1 -> r_2
load   r_2         -> r3
```

Access methods for values

- Variables stored in a register
- Variables stored in memory: `base + offset`
- Local variables: `r_arp + offset`
- Local variables of surrounding scopes: level information e.g. from access links/global display
- Static and global variables: addresses based on labels e.g. `loadI <label> -> r_i`
- Variables passed as parameters
- Variables stored in the heap
- Access methods for aggregates e.g. objects, structs, vectors, strings

Structure

- C's `struct`
- Pascal's `record`
- Object's record

Structure layouts

Linkedlist

Structure Layout Table

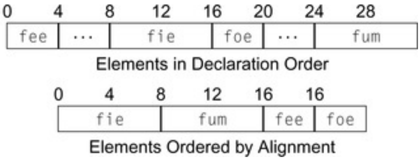
Name	Length	1^{st} Element
`node`	8	•
...	...	•

Structure Element Table

Name	Length	Offset	Type	Next
`node.value`	4	0	`int`	•
`node.next`	4	4	`struct node *`	•
...	...	...	...	...

Structure layout

```
struct example {
  int fee;
  double fie;
  int foe;
  double fum;
};
```



Object reference

```
Class Point {
    public int x, y;
    private int z;
    public void draw() {...};
    public void move() {...};
}
```

```
Class ColorPoint extends Point {
    private Color c;
    public void draw() {...};
    public void setc( Color x )
        { this.c = x };
}
```

(a) Class Definitions

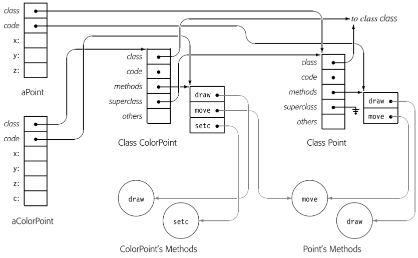
Class: Point
Superclass: *none*

x	int	public
y	int	public
z	int	private
draw	void()	public
move	void()	public

Class: ColorPoint
Superclass: Point

c	Color	private
draw	void()	public
setc	void()	public

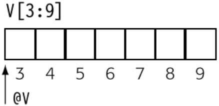
(b) Corresponding Scope Tables



(c) Object Records for Point, ColorPoint, aPoint, and aColorPoint

■ **FIGURE 6.3** Object Layout, Linking, and Inheritance.

Vector addressing



Vector Layout

```
subI    r_i 3         r_1 // (i - lower bound)
multI   r_1 4         r_2 // x element length (4)
add     r_@v    r_2   r_3 // address of v[i]
load    r_3           r_v[i] // get the value of v[i]
```

- The false zero of a vector `v` is the address where `v[0]` would be

The location of `v[i]` is $(i - low) \times w$ where

- low is the lower bound
- w is the element length
- Then, `v[0] = v - low * w`

Storing and accessing arrays

Base and offset calculation

Example: accessing a[4] whose element in a requires 4 bytes

```
loadI   @a_0          r_a0    # base
multI   r_i,    4     r_2     # offset size
addI    r_a0,   r_2   r_3     # add offset
load    r_3           r_ai
```

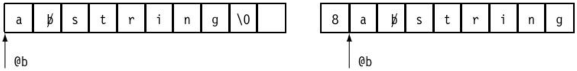
However, the the first index is not 0, we need to adjust the offset calculation

Multiplication to addition

If the offset size `w` in `r_i * w` is a power of two, we can use **shift** operation instead

- Fewer processing cycle
- E.g., `lshiftI r_i 2 -> r_2` is equivalent with `r_i * 4`

String representation



String with Null Termination

String with Explicit Length Field

String assignment

`a[1] = b[2];`

Generated codes depend on the target machine instruction set

```
loadI  @b -> r_b
cloadI r_b, 2 -> r_2
loadI @a -> r_a
cstoreAI r_2 -> r_a, 1
```

Or

```
loadI  0x0000FF00  ⇒ rC2    // mask for 2nd char
loadI  0xFF00FFFF  ⇒ rC124  // mask for chars 1, 2, & 4
loadI     @b          ⇒ r@b   // address of b
load      r@b         ⇒ r1    // get 1st word of b
and       r1, rC2     ⇒ r2    // mask away others
lshiftI   r2, 8       ⇒ r3    // move it over 1 byte
loadI     @a          ⇒ r@a   // address of a
load      r@a         ⇒ r4    // get 1st word of a
and       r4, rC124   ⇒ r5    // mask away 2nd char
or        r3, r5      ⇒ r6    // put in new 2nd char
store     r6          ⇒ r@a   // put it back in a
```

String assignment

```
                loadI     @b        ⇒ r@b
                loadAI    r@b,-4    ⇒ r1          // get b's length
                loadI     @a        ⇒ r@a
                loadAI    r@a,-4    ⇒ r2          // get a's length
                cmp_LT    r2,r1     ⇒ r3          // will b fit in a?
                cbr       r3        → Lsov,L1     // raise overflow
            L1: loadI     0         ⇒ r4          // counter
a = b;          cmp_LT    r4,r1     ⇒ r5          // more to copy?
                cbr       r5        → L2,L3
            L2: cloadAO   r@b,r4    ⇒ r6          // get char from b
                cstoreAO  r6        ⇒ r@a,r4      // put it in a
                addI      r4,1      ⇒ r4          // increment offset
                cmp_LT    r4,r1     ⇒ r7          // more to copy?
                cbr       r7        → L2,L3
            L3: storeAI   r1        ⇒ r@a,-4      // set length
```

String assignment

```
t1 = a;
t2 = b;
do {
  *t1++ = *t2++;
} while (*t2 != '\0')
```

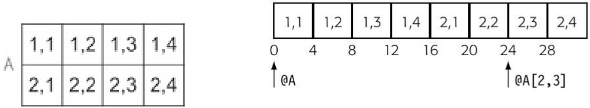
```
      loadI   @b      => r@b      // get pointers
      loadI   @a      => r@a
      loadI   NULL    => r1       // terminator
      cload   r@b     => r2       // get next char
L1:   cstore  r2      => r@a      // store it
      addI    r@b,1   => r@b      // bump pointers
      addI    r@a,1   => r@a
      cload   r@b     => r2       // get next char
      cmp_NE  r1,r2   => r4
      cbr     r4      -> L1,L2
L2:   nop                         // next statement
```

Other string operations

- String concatenation
- String length

Array storage layout

Row-major and column-major choices



Let

- `low1` be the first index of the first dimension
- `low2` be the first index of the second dimension
- `high1` be the first dimension's upper bound
- `high_2` be the second dimension's upper bound
- `len_k = high_k - low_k + 1` be the size of dimension `k`
- `w` be the size of each element

The location of `A[i,j]` is

`A + (i - low_1) * len_2 * w + (j - low_2) * w`

Array addressing

We can simplify

```
A + (i - low_1) * len_2 * w + (j - low_2) * w
```

into

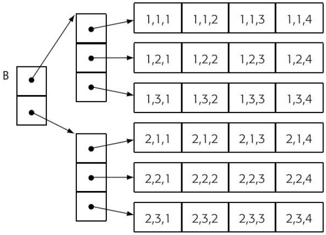
```
A + i * len_2 * w - low_1 * len_2 * w + j * w - low_2 * w
= A + (i * len_2 * w) + (j * w) - (low_1 * len_2 * 2 - low_2 * w)
= A + (i * len_2 * w) + (j * w) + A_0
= A_0 + (i * len_2 + j) * w
```

If `i, j` are in `r_i, r_j` then

```
loadI   A_0           -> r_A0 // false zero
multI   r_i    len_2  -> r_1  // i * len_2
add     r_1    r_j    -> r_2  // + j
multI   r_2    4      -> r_3  // *w, w = 4
loadA0  r_A0   r_3    -> r_A  // A[i,j]
```

High dimensional array

Indirection address



■ **FIGURE 7.4** Indirection Vectors in Row-Major Order for B[1:2,1:3,1:4].

To access B[i,j,k]

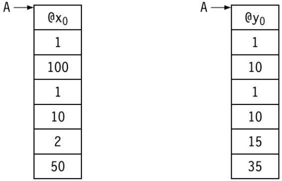
```
loadI   B_0          -> r_B0 // false zero
multI   r_i    4     -> r_1  // pointer size = 4
loadAO  r_B0   r_1   -> r_2  // Address of B[i]
multI   r_j    4     -> r_3  // pointer size = 4
loadAO  r_2    r_3   -> r_4  // Address of B[i,j]
muliT   r_k    4     -> r_5  // w = 4
loadAO  r_4    r_5   -> r_b  // B[i,j,k]
```

Dope vectors

A descriptor for an actual parameter array

```
program main;
  begin;
    declare x(1:100,1:10,2:50),
      y(1:10,1:10,15:35) float;
    call fee(x);
    call fee(y);
  end main;
procedure fee(A)
  declare A(*,*,*) float;
      ...
  end fee;
```

(a) PL/I Code That Passes Whole Arrays



(b) Dope Vector for the First Call Site

(c) Dope Vector for the Second Call Site

FIGURE 7.5 Dope Vectors.

Range check

A program that access an out-of-bound element is not well formed.

Naive range checking

```
for i in 1 .. n
    for j in 1 .. m
        if low_1 <= i <= high_1 and low_2 <= j <= high_2 then
            access a[i,j]
        else
            throw OutOfBoundException
```

Optimized version

```
if low_1 <= 1 <= and n <= high_1 and low_2 <= 1 and m <= high_2 then
    for i in 1 .. n
        for j in 1 .. m
            access a[i,j]
else
    throw OutOfBoundException
```

We can move the range check out of the two loops