Code Shape

- **Definition**
  - All those nebulous properties of the code that impact performance & code “quality”
  - Includes code, approach for different constructs, cost, storage requirements & mapping, & choice of operations

- **Code shape is the end product of many decisions**
  - Code shape influences algorithm choice & results
  - Code shape can encode important facts, or hide them

- **Expose as much derived information as possible**
**Code Shape (example 1)**

- **An example**

  \[ x + y + z \rightarrow t1 \]
  \[ x + y \rightarrow t1 \]
  \[ x + z \rightarrow t1 \]
  \[ y + z \rightarrow t1 \]
  \[ t1 + z \rightarrow t2 \]
  \[ t1 + y \rightarrow t2 \]
  \[ t1 + z \rightarrow t2 \]

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

- **The “best” shape for \( x+y+z \) depends on contextual knowledge**

  - There may be several conflicting options
    - May need to consider resolution of data type (e.g. float)
    - May need to consider common subexpression elimination

  Addition is commutative & associative for integers
Code Shape (example 2)

- **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
    - Less dense set of conditions, and too many for linear search
    - Uniform (log n) cost
  - Implement it as a jump table
    - Lookup address in a table & jump to it
    - Need a dense set of conditions to search
    - Uniform (constant) cost

- **Compiler must choose the best implementation strategy**
Program Construct

- Expressions
- Types
  - Scalar variable,
  - Array, structure
- Control statements
  - Conditionals (if-then-else, switch)
  - Loops
- Function calls
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**
  - Assign a virtual register to anything that can go into register
    - Relies on a strong register allocator
  - Load or store the others at each reference
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
  > 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
Example (1)

```c
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;
}
```

Example:

```
\begin{center}
\begin{tikzcd}
+ \arrow{dr} & \arrow{dl} \\
\text{x} & \text{y}
\end{tikzcd}
\end{center}
```

Produces:

```
expr("x") →
loadl @x \Rightarrow r1
loadAO r_{\text{arp}}, r1 \Rightarrow r2
expr("y") →
loadl @y \Rightarrow r3
loadAO r_{\text{arp}}, r3 \Rightarrow r4
NextRegister() \rightarrow r5
emit(add, r2, r4, r5) →
add r2, r4 \Rightarrow r5
```
Example (2)

```c
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;
}
```

Example:
```
x
  __________
 |           |
 |           |
 |           |
 | _________ |
 \         / \
 \        / \
 \_______ /   \
     x x
```

Generates:
```
loadl @x ⇒ r1
loadAO r0, r1 ⇒ r2
loadl 2 ⇒ r3
loadl @y ⇒ r4
loadAO r0, r4 ⇒ r5
mult r3, r5 ⇒ r6
sub r2, r6 ⇒ r7
```
Extending the Simple Treewalk (1)

- **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 (anti-, output-)

- **What about parameter values?**
  - Many linkages pass the first several values in registers
  - Accessing parameters
    - Call-by-value ⇒ just a local variable with negative 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 (2)

- **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

<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>
Extending the Simple Treewalk (3)

- **What about evaluation order?**
  - Can use commutativity & associativity to improve code
  - This problem is truly hard

- **What about order of evaluating operands?**
  - 1\textsuperscript{st} operand must be preserved while 2\textsuperscript{nd} is evaluated
  - Takes an extra register for 1\textsuperscript{st} operand
  - Should evaluate more demanding operand expression first
Handling Assignment

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

- **Treat as just another operator**
  - Evaluate \( \text{rhs} \) to a *value* (an *rvalue*)
  - Evaluate \( \text{lhs} \) 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*

- **Location**
  - Register: scalars with no aliases
  - Memory: scalars with possible aliases, aggregates such as structures, arrays
Generating Code in the Parser

- **Need to generate an initial IR form**
  - ASTs vs. 3-address code
    - Generate an AST
    - Use AST for some high-level, near-source work (type checking, optimization),
    - Then traverse AST and emit a lower-level IR

- **The big picture**
  - Recursive algorithm really works bottom-up
    - Actions on non-leaves occur after children are done
  - Can encode the 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 vs. Recursive Treewalk

Goal: \[ \text{Expr} \{ $$ = $1; \} ; \]
Expr: \[ \text{Expr PLUS Term} \]
\[ \{ t = \text{NextRegister}(); \]
\[ \quad \text{emit}(\text{add},$1,$3,t);$ $$ = t; \} \]
\| \text{Expr MINUS Term } \{...\}
\| \text{Term} \{ $$ = $1; \} ; \]
Term: \[ \text{Term TIMES Factor} \]
\[ \{ t = \text{NextRegister}(); \]
\[ \quad \text{emit}(\text{mult},$1,$3,t);$ $$ = t; \} ; \]
\| \text{Term DIVIDES Factor } \{...\}
\| \text{Factor} \{ $$ = $1; \} ; \]
Factor: \[ \text{NUMBER} \]
\[ \{ t = \text{NextRegister}(); \]
\[ \quad \text{emit}(\text{loadI},\text{val}($1),\text{none}, t ); \]
\[ \quad $$ = t; \} \]
\| \text{ID} \]
\[ \{ t1 = \text{base}($1); \]
\[ \quad t2 = \text{offset}($1); \]
\[ \quad t = \text{NextRegister}(); \]
\[ \quad \text{emit}(\text{loadAO},t1,t2,t); \]
\[ \quad $$ = t; \} \]

\[
\text{expr(node)} \{
\text{int result, t1, t2; }
\text{switch (type(node)) } \{
\text{case } \times,\div,+,\,- : }
\text{t1} \leftarrow \text{expr(left child(node))};
\text{t2} \leftarrow \text{expr(right child(node))};
\text{result} \leftarrow \text{NextRegister}();
\text{emit}(\text{op(node)},t1,t2,\text{result});
\text{break};
\text{case IDENTIFIER: }
\text{t1} \leftarrow \text{base(node)};
\text{t2} \leftarrow \text{offset(node)};
\text{result} \leftarrow \text{NextRegister}();
\text{emit}(\text{loadAO},t1,t2,\text{result});
\text{break};
\text{case NUMBER: }
\text{result} \leftarrow \text{NextRegister}();
\text{emit}(\text{loadI},\text{val(node)},\text{none},\text{result});
\text{break};
\text{break};
\text{break};
\text{}}
\text{return result; }
\text{}}
**Handle Arrays - \( A[i,j] \)**

**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
- Everything is 1D vector \( \Rightarrow \) conceptually simple design
- 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

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

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

- **Row-major order, two dimensions**
  - @A + ((i_1 - low_1) x (high_2 - low_2 + 1) + i_2 - low_2) x sizeof(A[1])

- **Column-major order, two dimensions**
  - @A + ((i_2 - low_2) x (high_1 - low_1 + 1) + i_1 - low_1) x sizeof(A[1])

- **Indirection vectors, two dimensions**
  - *(A[i_1])[i_2]* — where \( A[i_1] \) is, itself, a 1-d array reference

Expensive! Require two memory references

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

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

Expensive! Require two memory references
Optimizing Address Calculation for $A[i,j]$

❖ In row-major order
  - $@A + (i - low_1)(high_2 - low_2 + 1) \times w + (j - low_2) \times w$

❖ 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,
  Define $@A_0 = @A - (low_1 \times (high_2 - low_2 + 1) \times w + low_2 \times w$
  $len_2 = (high_2 - low_2 + 1)$

❖ Then, the address expression becomes
  $@A_0 + (i \times len_2 + j) \times w$

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

Compile-time constants
Array-valued Parameters

- Some languages rely on compiler to pass necessary information at function calls

- 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

- 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 call-by-value languages pass arrays by reference
  - This is a language design issue
Array Address Calculations in a Loop

DO J = 1, N
END DO

- Naïve: Perform the address calculation twice

DO J = 1, N
    R1 = @A_0 + (J x len_1 + I ) x floatsize
    R2 = @B_0 + (J x len_1 + I ) x floatsize
    MEM(R1) = MEM(R1) + MEM(R2)
END DO
Array Address Calculations in a Loop

DO J = 1, N
END DO

❖ Move common calculations out of loop

R1 = I x floatsize

\[ c = \text{len}_1 \times \text{floatsize} \] ! Compile-time constant

R2 = @A_0 + R1
R3 = @B_0 + R1

DO J = 1, N
    a = J x c
    R4 = R2 + a
    R5 = R3 + a
    MEM(R4) = MEM(R4) + MEM(R5)
END DO
Array Address Calculations in a Loop

DO J = 1, N
END DO

📍 Convert multiply to add (Operator Strength Reduction)

R1 = I x floatsize

\[ c = \text{len}_1 \times \text{floatsize} \] ! Compile-time constant

R2 = @A_0 + R1 ; R3 = @B_0 + R1

DO J = 1, N
    R2 = R2 + c
    R3 = R3 + c
    MEM(R2) = MEM(R2) + MEM(R3)
END DO
Boolean & Relational Values (1)

- **Numerical representation**
  - Assign values to TRUE and FALSE
  - Use hardware AND, OR, and NOT operations
  - Use comparison to get a boolean from a relational expression

- **Examples**

  If \((x < y)\)
  
  then \(stmt_T\)
  
  else \(stmt_F\)

  \[
  \begin{align*}
  &\text{becomes} \\
  &L_T: \text{stmt}_T \\
  &L_F: \text{stmt}_F \\
  &L_E: \ldots\text{other stmts}\ldots
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{cmp}_{LT} \quad r_x, r_y \Rightarrow r_1 \\
  &\text{cbr} \quad r_1 \rightarrow L_T, L_F \\
  &\text{br} \quad \rightarrow L_E
  \end{align*}
  \]
Boolean & Relational Values (2)

*Positional (implicit) representation*
- ISA uses a condition code
- Compare provides enough info for every possible relation
- Must use a conditional branch to interpret result of compare

*Example:*

If \( x < y \) then \( \text{stmt}_T \) becomes \( \text{L}_T: \text{stmt}_T \) \( \text{br} \rightarrow \text{L}_E \) else \( \text{stmt}_F \) \( \text{L}_F: \text{stmt}_F \) \( \text{L}_E: \ldots \text{other stmts} \ldots \)

**Condition codes**
- are an architect’s hack
- allow ISA to avoid some comparisons
- complicates code for simple cases
Conditional Move & Predication

- Conditional move & Predication both simplify if-then-else

<table>
<thead>
<tr>
<th>Example</th>
<th>OTHER ARCHITECTURAL VARIATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; y)</td>
<td></td>
</tr>
<tr>
<td>then a ← c + d</td>
<td>Conditional Move</td>
</tr>
<tr>
<td>else a ← e + f</td>
<td>Predicated Execution</td>
</tr>
<tr>
<td>comp</td>
<td>cmp_LT</td>
</tr>
<tr>
<td>r_x, r_y ⇒ cc_1</td>
<td>r_x, r_y ⇒ r_1, r_2</td>
</tr>
<tr>
<td>add</td>
<td>(r_1) add</td>
</tr>
<tr>
<td>r_c, r_d ⇒ r_1</td>
<td>r_c, r_d ⇒ r_a</td>
</tr>
<tr>
<td>add</td>
<td>(r_2) add</td>
</tr>
<tr>
<td>r_e, r_f ⇒ r_2</td>
<td>r_e, r_f ⇒ r_a</td>
</tr>
<tr>
<td>i2i_LT</td>
<td>cc_1, r_1, r_2 ⇒ r_a</td>
</tr>
</tbody>
</table>

- Both versions avoid the branches
- Both are shorter than CCs or Boolean-valued compare
- Are they better?
Control Flow

- **If-then-else**
  - Follow model for evaluating relationals & booleans with branches

- **Branching versus predication (e.g., IA-64)**
  - Frequency of execution
    - Uneven distribution ⇒ do what it takes to speed common case
  - Amount of code in each case
    - Unequal amounts means predication may waste issue slots
  - Another control flow inside *then-block* or *else-block*
    - Any branching activity within the block complicates the predicates and makes branches attractive
Control Flow - loop

- **Loops**
  - Evaluate condition before loop (if needed)
  - Evaluate condition after loop
  - Branch back to the top (if needed)

- **Merges post-test with last block of loop body**
  - Fill the delay slot of post-test branch with instructions in last block of loop body

- **while, for, do, & until all fit this basic model**
Loop Implementation Code

for (i = 1; i < 100; i++) {
    body
}

next statement

Initialization

Pre-test

Post-test

for (i = 1; i < 100; i++) {
    loadI 1 \rightarrow r_1
    loadI 1 \rightarrow r_2
    loadI 100 \rightarrow r_3
    cmp_GE r_1, r_3 \rightarrow r_4
    cbr r_4 \rightarrow L_2, L_1

    L_1: body
        add r_1, r_2 \rightarrow r_1
        cmp_LT r_1, r_3 \rightarrow r_5
        cbr r_5 \rightarrow L_1, L_2

    L_2: next statement
Many modern languages include a break
- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement

Translates into a jump
- Targets statement outside control-flow construct
- Creates multiple-exit construct
- Skip in loop goes to next iteration
Control Flow - case

- Case Statements
  1. Evaluate the controlling expression
  2. Branch to the selected case
  3. Execute the code for that case
  4. Branch to the statement after the case (use break)

- Parts 1, 3, & 4 are well understood, part 2 is the key

- Strategies
  - Linear search (nested if-then-else constructs)
  - Build a table of case expressions & binary search it
  - Directly compute an address (requires dense case set)
Procedure Call – register save

- **If \( p \) calls \( q \), one of them must**
  - Preserve register values in ARs (caller-saves vs. callee-saves)

- **Space tradeoff**
  - Pre-call & post-return occur on every call
  - Prolog & epilog occur once per procedure
  - Moving operations into prolog/epilog saves space

- **As register sets grow, save/restore code does, too**
  - Each register costs 2 operations
  - Optimizations
    - Hardware support for save/restore or storeM/loadM
    - Use a library routine for save/restore to save code space
Procedure Call - parameters

- Evaluating parameters
  - Call by reference ⇒ evaluate parameter to an lvalue
  - Call by value ⇒ evaluate parameter to an rvalue & store it
  - Aggregates, arrays, & strings are usually call-by-reference

- Procedure-valued parameters
  - Must pass starting address of procedure
  - With access links, need the lexical level as well
    - Procedure value is a tuple <level, address>
    - Precall inserts the appropriate code to fetch level and adjust access link
Summary

- **Expression**
  - Treewalk with unlimited number of virtual registers
  - Use registers for multiple uses of same variable

- **Array references**
  - Address calculations for element accesses
  - Optimize array references within loop

- **Boolean & relational value**
  - Numerical representation
  - Positional (implicit) representation $\Rightarrow$ use condition code

- **Control flow**
  - Patterns for If-then-else, Loop, & Case

- **Procedure call**
  - Register save, Parameter passing