Intermediate Code Generation

Basic Approach and Application to Assignment and Expressions

Array Expressions

Boolean Expressions

A compiler is a lot of fast stuff followed by some hard problems

– The hard stuff is mostly in Code Generation and Optimization
– For super-scalars, its Allocation & Scheduling that counts

Three-Address Instructions

• Assignments:
  - \( x = y \cdot op \cdot z \) (binary operator)
  - \( x = y \) (copy)
  - \( x = y[i] \) and \( x[i] = y \) (array indexing assignments)
  - \( x = phi \cdot y \cdot z \) (Static Single Assignment instruction)

• Memory Operations:
  - \( x = *y; x = &y \) and \( *x = y \) for assignments via pointer variables.

The hard stuff is mostly in Code Generation and Optimization

– For super-scalars, it’s Allocation & Scheduling that counts
SDT to Three Address Code

- Attributes for the Non-Terminals, E and S
  - Location (in terms of temporary variable) of the value of an expression: E.place
  - The Code that Evaluates the Expressions or Statement: E.code
  - Markers for beginning and end of sections of the code S.begin, S.end

- Semantic Actions in Productions of the Grammar
  - Functions to create temporaries
  - newtemp
  - labels
  - newlabel
  - Auxiliary functions to enter symbols and consult types corresponding to declarations in a symbol table.
  - Generate the code we use the emit function gen which creates a list of instructions to be emitted later and can generate symbolic labels corresponding to next instruction of a list.
  - Use of append function on lists of instructions.
  - Synthesized and Inherited Attributes

Assignment Statements

\[ E = \text{call}(\text{call}(E_1,E_2),E_3); \]
\[ E = \text{call}(E_1,E_2,E_3); \]
\[ E = \text{call}(E_1,E_2,E_3,E_4); \]
\[ E = \text{call}(E_1,E_2,E_3,E_4,E_5); \]
\[ E = \text{call}(E_1,E_2,E_3,E_4,E_5,E_6); \]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[ 1 \times E \]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

\[ \text{Sid} = \text{EE \times id} \]
\[ \text{id} = a \]
\[ \text{id} = b \]
\[ \text{id} = x \]
\[ \text{id} = c \]
\[ \text{id} = d \]
\[ \text{id} = e \]
\[ \text{id} = f \]

\[ t1 = e \times f; \]
\[ t2 = c \times d; \]
\[ t3 = t2 - t1; \]
\[ t4 = a \times b; \]
\[ t5 = t4 + t3; \]
\[ x = t5; \]

Reusing Temporary Variables

- Temporary Variables
  - Short lived
  - Used for evaluation of expressions
  - Clutter the symbol table

- Change the newtemp function
  - Keep track of when a value created in a temporary is used
  - Use a counter to keep track of the number of active temporaries
  - When a temporary is generated by newtemp, increment counter
  - When a temporary is used in an expression, decrement counter
  - Initialize counter to zero (0)

- Alternatively, can be done as a post-processing pass...

Code Generation for Array Accesses

- Questions:
  - What is the base type of A?
  - What are the dimensions of A?

- Answers:
  - Use symbol table
  - Check array layout

\[ x = A[y,z]; \]

Laying Out Arrays

The Concept

Row-major order

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Column-major order

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Indirection vectors

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Computing an Array Address

\[ A[i] \]

- baseA + (i - low) \times \text{sizeof}(\text{baseType}(A))
- In general: base(A) + (i - low) \times \text{sizeof}(\text{baseType}(A))

What about \( A[i,j] \)?

Row-major order, two dimensions:
baseA + (i - low) \times (high2 - low2 + 1) + (j - low2) \times \text{sizeof}(\text{baseType}(A))

Column-major order, two dimensions:
baseA + (j - low2) \times (high1 - low1 + 1) + (i - low1) \times \text{sizeof}(\text{baseType}(A))

Indirection vectors, two dimensions:

\[ A[i][j] \] where \( A[i] \) is itself, a 1-d array reference

Address Calculation for \( A[i_1,i_2,\ldots,i_m] \)

- Addressing generalizes to
  \[
  \text{baseA} + ((i_1-\text{low}_1) \times \text{high}_1, \ldots, (i_m-\text{low}_m) \times \text{high}_m) \times w
  \]
  \[
  + \text{baseA} - ((0, \ldots, 0), \text{high}_1-\text{low}_1, \ldots, \text{high}_m-\text{low}_m) \times w
  \]
  \[
  \text{where } \text{low}_j = \text{high}_j - \text{low}_j + 1 \text{ and } w = \text{sizeof}(	ext{baseType}(A))
  \]

First term can be computed using the recurrence:

\[
\alpha_i = \text{low}_i
\]

\[
\alpha_{i_1} \times \alpha_{i_2} \times \ldots \alpha_{i_m} = \text{low}_m
\]

at the end multiply by \( w \) and add compile-time constant term

\[ \text{SDT for Addressing Arrays Elements} \]

- Three Attributes
  - base: just the name or base address of the array
  - offset: the index value into the array
  - dim: the number of dimensions

- Use the Recurrence to Compute Offset
  \[
  \text{offset}_m = \text{offset}_{m-1} \times n_m + \text{low}_m
  \]
  - At the end multiply by \( w = \text{sizeof(\text{baseType}(A))} \)
  - Add the compile-time constant term
  - Keep track of which dimension at each level

- Use the auxiliary function \( n_m = \text{numElm}(\text{dim}); \text{of}(\text{dim}) \) as the number of elements along the \( m \)-th dimension of \( A \)
Some improvement is possible

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, and low, rather than low and high
- 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

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

Some improvement is possible:
- Save len, and low, rather than low and high
- 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 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 sub-expressions in the address polynomial

• Contents of dope vector are fixed during each activation

⇒ Handle them like parameter arrays

Array Address Calculations in a Loop

DO J = 1, N
END DO

• Naïve: Perform the address calculation twice

DO J = 1, N
  R1 = baseA + (J x len1 + I) x floatsize
  R2 = baseB + (J x len1 + I) x floatsize
  MEM(R1) = MEM(R1) + MEM(R2)
END DO

• Sophisticated: Move common calculations out of loop

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

• Very sophisticated: Convert multiply to add (Operator Strength Reduction)

R1 = I x floatsize
c = len1 x floatsize ! Compile-time constant
R2 = baseA + R1
R3 = baseB + R1
DO J = 1, N
  R2 = R2 + c
  R3 = R3 + c
  MEM(R2) = MEM(R2) + MEM(R3)
END DO

SDT Scheme for Boolean Expressions

• Two Basic Code Generation Flavors
  – Use boolean and, or, not instructions (like arithmetic)

  – Control flow (or positional code) defines true or false of predicate

• Arithmetic Evaluation
  – Simpler to generate code as just eagerly evaluate the expression
  – Associate “1” or “0” with outcome of predicates and combine with logic instr.

  – Use the same SDT scheme explained for arithmetic operations.

• Control Flow Evaluation (short circuit evaluation)
  – More efficient in many cases

  – Complications
  • Need to know Address to Jump To for Some Cases
  • Solution: Two Additional Attributes

  – *nextstat (Inherited) Indicates the next location to be generated

  – *laststat (Synthesized) Indicates the last location filled

  – As code is generated the attributes are filled with the correct value

Arithmetic Scheme: Grammar and Actions

E = false E.place = newtemp()
E.code = {gen(E.place = 0)}
E.nextstat = E.laststat + 1

E = true E.place = newtemp()
E.code = {gen(E.place = 1)}
E.nextstat = E.laststat + 1

E = (E) E.place = E.place()
E.code = {gen(E.place = E.place)}
E.nextstat = E.laststat
E.laststat = E.laststat

E = not E E.place = newtemp()
E.code = append(E.code, gen(E.place = not E.place)
E.nextstat = E.laststat
E.nextstat = E.laststat

E = end E E.place = newtemp()
E.code = {gen(E.place = E.place)}
E.nextstat = E.laststat
E.nextstat = E.laststat
Arithmetic Scheme: Grammar and Actions

\[
E \rightarrow E_1 \text{ or } E_2 \\
E \rightarrow \text{newtemp}() \text{,} \ E \cdot \text{place} = \text{E\.place} \text{,} \ E \cdot \text{code} = \text{E\.code} \\
E \cdot \text{nextstat} = E \cdot \text{nexstat} \text{,} \ E \cdot \text{laststat} = E \cdot \text{laststat} + 1
\]

\[
E \rightarrow E_1 \text{ and } E_2 \\
E \rightarrow \text{newtemp}() \text{,} \ E \cdot \text{place} = \text{E\.place} \text{,} \ E \cdot \text{code} = \text{E\.code} \\
E \cdot \text{nextstat} = E \cdot \text{nexstat} \text{,} \ E \cdot \text{laststat} = E \cdot \text{laststat} + 1
\]

Boolean Expressions: Example

```
00: if a < b goto 03
01: t1 = 0
02: goto 04
03: t1 = 1
04: if c < d goto 07
05: t2 = 0
06: goto 08
07: t2 = 1
08: if e < f goto 11
09: t3 = 0
10: goto 12
11: t3 = 1
12: t4 = t2 and t3
13: t5 = t1 or t4
```

### Summary

- **Intermediate Code Generation**
  - Using Syntax-Directed Translation Schemes
  - Expressions and Assignments
  - Array Expressions and Array References
  - Boolean Expressions (Arithmetic Scheme)