Code Generation

Compiler Design

CSE 504

Last modifled: Wed Apr 08 2015 at 16:10:34 EDT
Version: 1.6 15:28:43 2015/01/25
Compiled at 16:12 on 2015/04/08

Code Generation

- Intermediate code generation: Abstract (machine independent) code.
- Code optimization: Transformations to the code to improve time/space performance.
- Final code generation: Emitting machine instructions.

Syntax Directed Translation

Interpretation:

$$E \longrightarrow E_1 + E_2 \quad \{\ E.val := E_1.val + E_2.val;\ \}$$

Type Checking:

$$E \longrightarrow E_1 + E_2 \quad \{$$

if $E_1.type \equiv E_2.type \equiv int$

$$E.type = int;$$

else

$$E.type = float;$$

$$\}$$

Code Generation via Syntax Directed Translation

Code Generation:

$$E \longrightarrow E_1 + E_2 \quad \{ \; E.code = E_1.code \parallel E_2.code \parallel \text{“add”} \; \}$$

Stack Machines

Simplified translation, but fewer opportunities for optimization.

- Machine Configuration:
  - Contents of stack; each element of the stack is a cell of some standard size (e.g. 32 bits).
  - Registers: Program counter, Stack pointer, (more later)
- Stack representation:
  - "[ ]" to represent empty stack
  - "$v_1::S$" to represent a stack whose top element is v_1 and the remainder of the stack is S.
  - "$S[i]$" represents the value at the i-th element of stack S (counting from the base, not top, of the stack).

Stack Machine Instructions

- `load_immed` v: Push v on stack.

$$S \quad \text{—}\boxed{\texttt{load\_immed v}}\longrightarrow \quad v :: S$$

- `load`: Load value from given address to top of stack.

$$a :: S \quad \text{—}\boxed{\texttt{load}}\longrightarrow \quad v :: S$$

  where $v = S[a]$.

- `store`: Store a given value to a given address.

$$v :: a :: S \quad \text{—}\boxed{\texttt{store}}\longrightarrow \quad v :: T$$

  where T is same as S except $T[a] = v$.

- `add`: Add top two elements.

$$v_2 :: v_1 :: S \quad \text{—}\boxed{\texttt{add}}\longrightarrow \quad v_1 + v_2 :: S$$

- `pop`: Remove top-most element.

$$v :: S \quad \text{—}\boxed{\texttt{pop}}\longrightarrow \quad S$$

Register Machines

- Machines with a (possibly unbounded) number of registers. Lets call them t_1, t_2, ...
- Separate *Heap* space for dynamically allocated objects.
- Instructions:
  - `move` t_1, t_2: Move value from register t_2 to t_1.
  - `move_immed` t_1, i: move literal constant i to a register t_1.
  - `add` t_1, t_2: Add values of t_1 and t_2, store it back in t_1.

Code Generation and Attributes (Stack Machine)

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \texttt{ + } E_2 & \{ \\
 & & & E.code = E_1.code \parallel E_2.code \parallel \\
 & & & \quad \text{“}\texttt{add}\text{”} \\
 & & & \} \\
E & \longrightarrow & \texttt{int} & \{\ E.code = \text{“}\texttt{load\_immed int}.val\text{”}\ \} \\
E & \longrightarrow & \texttt{id} & \{\ E.code = \text{“}\texttt{load id}.addr\text{”}\ \} \\
E & \longrightarrow & \texttt{id = } E_1 & \{ \\
 & & & E.code = \text{“}\texttt{load\_immed id}.addr\text{”} \parallel \\
 & & & \quad E_1.\text{code} \parallel \\
 & & & \quad \text{“}\texttt{store}\text{”} \\
 & & & \}
\end{array}
$$

`id`.*addr* is the address of cell allocated for `id`.

Variables and Addresses in Register Machines

- Since the register machine has unbounded number of registers, all local variables and function parameters will be stored in registers.
- Each program variable is mapped to a distinct abstract register.
  Each `id` has an attribute `id`.*addr* to represent this mapping.
- For final code generation, the registers of the abstract machine will be mapped to (a small, finite) set of registers of a concrete machine (e.g. MIPS).
- Clearly not all abstract machine registers may have a corresponding concrete machine register.
  Such abstract registers will be **spilled** to cells on a stack on the concrete machine.

Code Generation and Attributes (Register Machine)

$E \longrightarrow E_1$ `+` E_2 {
$E.t = generate\_new\_temporary();$
$E.code = E_1.code \parallel E_2.code$
$\parallel$ "`add` $E.t$, $E_1.t$, $E_2.t$"
}

$E \longrightarrow$ `int` {
$E.t = generate\_new\_temporary();$
$E.code =$ ["`mov_immed` $E.t$, `int`$.val$"]
}

$E \longrightarrow$ `id` {
$E.t =$ `id`$.addr;$
$E.code =$ [ ]
}

$E \longrightarrow$ `id =` E_1 {
$E.t = E_1.t;$
$E.code = E_1.code$
$\parallel$ "`move` `id`$.addr$, $E.t$"
}

References

Some languages allow variables to have *locations* (i.e. addresses) and a variable may **refer** to another variable's location.

- Languages such as C/C++ permit programmers to obtain location of arbitrary variables (using the "address-of" operation, "&"), and dereference locations (i.e. access the value stored at an address, using "*").
- When translating a C-like language, every variable should be potentially allocated on stack; it can be mapped to a register if there is no operation that takes its address.
- Languages such as Java give locations only to objects and arrays. All variables are stack-allocated.
  The location of a variable itself is not accessible to the program.

l- and *r*-values

```
i = i + 1;
```

- *r*-**value**: actual value of the expression
- *l*-**value**: for expressions associated with specific memory addresses, the location where the value of the expression is stored.

- Some expressions (e.g. `i`) have both *l*- and *r*-values.
- Other expressions (e.g. `5`) have only an *r*-value.
- Some expression's values may be (at least temporarily) stored in locations, but those locations may not have a meaning in terms of the program, and we consider them also to have only *r*-values (e.g. `i+1`).

Code Generation for *L*-expressions

An *L*-expression is one which has an *l*-value.

- Roughly speaking *L*-expressions are those that may occur on the lhs of an assignment.
- In Proto(2), the only *L*-expressions were identifiers.
- In Proto(3), *L*-expressions include array access.
- For compiling assignments, we will use additional attributes for *L*-expressions (other than $L.t$ and $L.code$, which all expressions have.

Arrays

Consider expression grammar changed as follows:

$$
\begin{array}{lcl}
E & \rightarrow & E \texttt{ + } E \\
E & \rightarrow & L \texttt{ = } E \\
E & \rightarrow & \texttt{int} \\
E & \rightarrow & L \\
 & & \\
L & \rightarrow & \texttt{id} \\
L & \rightarrow & L\texttt{[}E\texttt{]}
\end{array}
$$

- L represents simple identifiers as well as array expressions.
- The index of an array expression can be any arbitrary expression (including an array expression itself)

  Example: `x[y[i]]`
- The base of an array expression is an identifier or another array expression.

  Example: `(x[i])[j]`
- LHS of an assignment can be an array expression.

Addresses and Allocation

- For Proto, we'll use Java-like convention of keeping variables in stack/registers, and arrays (and later, objects) on heap.
- For heap access, we use the following intermediate code instructions:
  - `hstore` a, r: store value to a heap cell.
    Register a has the address of the cell in heap, and register r has the value to be stored.
  - `hload` r, a: load value from a heap cell.
    Register a has the address of the cell in heap, and register r is the destination for the load.
  - `halloc` r_1, r_2: allocate a segment of heap cells.
    Register r_2 contains the number of cells to allocate. Register r_1 will then be set to the base address of the allocated heap cells.
  - ?? `hsize` r_1, r_2: get size of a heap segment.
    Register r_1 is the address of the heap segment. Register r_2 will then be set to the size of the segment.

Bounds Check

Generating code for arrays: Allocation

$$E \longrightarrow \texttt{new T [} E_1 \texttt{]}$$

{
$E.t = \mathit{generate\_new\_temporary}();$
$E.\mathit{code} = E_1.\mathit{code} \parallel$ "`halloc` $E.t$, $E_1.t$"
}

- E_1 will be an integer-valued expression that specifies the number of elements in the array to allocate.
- Type T is ignored (at least, for now).
- E, then, is a reference to the newly allocated array.
- If bounds check is needed, additional book-keeping info needs to be maintained with the array.

  ... allocate $n + 1$ cells, and use the zero-th cell to store the length!

Generating code for arrays: LHS

$L \longrightarrow$ `id` {

$L.t = L.at =$ `id`$.addr$;

$L.lcode = L.rcode = [\,]$;

$L.mem = reg$;

}

$L \longrightarrow L_1$ `[` E `]` {

$L.at = generate\_new\_temporary()$;

$L.lcode = L_1.rcode$

$\|\ E.code$

$\|$ "`mul` $L.at$, $E.t$, 4"

$\|$ "`add` $L.at$, $L.at$, $L_1.t$";

$L.t = generate\_new\_temporary()$;

$L.rcode = L.lcode$

$\|$ "`hload` $L.t$, $L.at$";

$L.mem = heap$;

}

- $L.t$: Register holding L's value.
- $L.at$: Register holding L's address.
- $L.lcode$: Code for evaluating L's address.
- $L.rcode$: Code for evaluating L's value.
- Note: no bounds check!

Generating code for arrays: RHS

$$E \longrightarrow L \quad \{ \; E.t = L.t; \; E.code = L.rcode \; \}$$

Example expression:

```
(i + a[i])
   + b[i][j]
```

With i.$addr = t_1$,
j.$addr = t_2$,
a.$addr = t_3$,
b.$addr = t_4$.

```
// i's rcode (empty)
// a[i]'s rcode
mul t5, t1, 4
add t5, t5, t3
aload t6, t5
// i+a[i]'s code
add t7, t1, t6
// b[i][j]'s rcode:
//     b[i]'s rcode
mul t8, t1, 4
add t8, t8, t4
aload t9, t8
//   use b[i] as base:
mul t10, t2, 4
add t10, t10, t8
aload t11, t10
// add b[i][j] to prev result
add t12, t7, t11
```

Generating code for arrays: Assignments

$$
\begin{array}{lll}
E & \longrightarrow & L = E_1 \\
 & \{ & E.t = E_1.t; \\
 & & \text{if } L.mem == reg \\
 & & \quad assigncode = \text{“}\texttt{move } L.at, E_1.t\text{”}; \\
 & & \text{else} \\
 & & \quad assigncode = \text{“}\texttt{hstore } L.at, E_1.t\text{”}; \\
 & & E.code = L.lcode \\
 & & \quad \| E_1.code \\
 & & \quad \| assigncode; \\
 & \} &
\end{array}
$$

Code Generation for Statements

$$
\begin{array}{lcll}
Ss & \longrightarrow & S \;\; Ss_1 & \{ \\
 & & & Ss.code = S.code \\
 & & & \qquad \| \; Ss_1.code; \\
 & & & \} \\
Ss & \longrightarrow & \epsilon & \{ \; Ss.code = [\,] \} \\
 & & & \\
S & \longrightarrow & E \; ; & \{ \\
 & & & S.code = E.code; \\
 & & & \}
\end{array}
$$

Conditional Statements

$$
\begin{array}{lll}
S & \longrightarrow & \texttt{if}\ E,\ S_1,\ S_2 \quad \{ \\
& & \quad \mathit{elselabel} = \mathit{get\_new\_label}(); \\
& & \quad \mathit{endlabel} = \mathit{get\_new\_label}(); \\
& & \quad S.\mathit{code} = \quad\quad E.\mathit{code} \\
& & \qquad\qquad \| \quad\quad \text{“}\texttt{beq}\ E.t,\ 0,\ \mathit{elselabel}\text{”} \\
& & \qquad\qquad \| \quad\quad S_1.\mathit{code}; \\
& & \qquad\qquad \| \quad\quad \text{“}\texttt{jmp}\ \mathit{endlabel}\text{”} \\
& & \qquad\qquad \| \text{“}\mathit{elselabel}\text{:”} \\
& & \qquad\qquad \| \quad\quad S_2.\mathit{code}; \\
& & \qquad\qquad \| \text{“}\mathit{endlabel}\text{:”} \\
& & \quad \}
\end{array}
$$

Conditional Statements and Continuations

$S.end$: label to jump after S is executed completely.

$$
\begin{array}{lll}
S & \longrightarrow & \texttt{if}\ E,\ S_1,\ S_2 \quad \{ \\
 & & \quad S.begin = get\_new\_label(); \\
 & & \quad S_1.end = S_2.end = S.end; \\
 & & \quad S.code = \text{“}S.begin\text{:”} \\
 & & \qquad \| \ E.code \\
 & & \qquad \| \ \text{“beq } E.t, \ 0, \ S_2.begin\text{”} \\
 & & \qquad \| \ S_1.code \\
 & & \qquad \| \ S_2.code; \\
 & & \}
\end{array}
$$

Continuations

Attributes of a statement that specify where control will flow to **after** the statement is executed.

- Analogous to the *follow* sets of grammar symbols.
- In deterministic languages, there is only one continuation for each statement.
- Can be generalized to include local variables whose values are needed to execute the following statements:

  Uniformly captures call, return *and* **exceptions**.

Sequence and Continuation

- Most frequently, the continuation of a statement will simply be its succeeding statement.
- We will use a special label "*fallthrough*" to denote this.

$$
\begin{array}{lcll}
Ss & \longrightarrow & S\ Ss_1 & \{ \\
 & & & \quad Ss_1.end = Ss.end; \\
 & & & \quad S.end = fallthrough; \\
 & & & \quad Ss.code = \ldots \} \\
 & & & \\
S & \longrightarrow & E\ ; & \{ \\
 & & & \quad \text{if } S.end == fallthrough \\
 & & & \qquad next = [\,] \\
 & & & \quad \text{else} \\
 & & & \qquad next = \text{“jmp } S.end\text{”} \\
 & & & \quad S.\text{code} = \ldots \\
 & & & \qquad \| \; next; \}
\end{array}
$$

Code Generation for Boolean Expressions

$$E \longrightarrow E_1 \;\&\&\; E_2 \quad \{ \quad E.t = \mathit{generate\_new\_temporary}(); \; E.\mathit{code} = E_1.\mathit{code} \parallel E_2.\mathit{code} \parallel \text{“}\texttt{and } E.t, E_1.t, E_2.t\text{”}; \quad \}$$

- The above code evaluates E_2 regardless of the value of E_1.
- Short circuit code: evaluate E_2 only if needed.

$$E \longrightarrow E_1 \;\&\&\; E_2 \quad \{ \quad E.t = \mathit{generate\_new\_temporary}(); \; \mathit{skip} = \mathit{generate\_new\_label}(); \; E.\mathit{code} = E_1.\mathit{code} \parallel \text{“}\texttt{move } E.t, E_1.t\text{”} \parallel \text{“}\texttt{beq } E.t, \texttt{0}, \mathit{skip}\text{”} \parallel E_2.\mathit{code} \parallel \text{“}\texttt{move } E.t, E_2.t\text{”} \parallel \text{“}\mathit{skip}\texttt{:”}; \quad \}$$

Generating Shortcircuit Code

Use two continuations for each boolean expression:

- $E.success$: where control will go when expression in E evaluates to *true*.
- $E.fail$: where control will go when expression in E evaluates to *false*.

Both continuations are *inherited* attributes.

Shortcircuit Code for Boolean Expressions

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \texttt{ \&\& } E_2 & \{ \quad E_1.fail = E.fail; \\
 & & & \quad E_2.fail = E.fail; \\
 & & & \quad E_1.success = get\_new\_label(); \\
 & & & \quad E_2.success = E.success; \\
 & & & \quad E.code = E_1.code \parallel \\
 & & & \qquad \text{"}E_1.success\text{:"} \parallel \\
 & & & \qquad E_2.code \quad \} \\
E & \longrightarrow & \texttt{! } E_1 & \{ \quad E_1.fail = E.success; \\
 & & & \quad E_1.success = E.fail; \\
 & & & \quad E.code = E_1.code \quad \} \\
E & \longrightarrow & \texttt{true} & \{ \quad E.code = \text{“}\texttt{jmp, } E.success\text{”} \quad \}
\end{array}
$$

Short-circuit code for Conditional Statements

$$
\begin{array}{lll}
S & \longrightarrow & \texttt{if}\ E,\ S_1,\ S_2 \quad \{ \\
 & & \quad S.begin = get\_new\_label(); \\
 & & \quad S_1.end = S_2.end = S.end; \\
 & & \quad E.success = S_1.begin; \\
 & & \quad E.fail = S_2.begin; \\
 & & \quad S.code = \text{“}S.begin\text{:”} \parallel \\
 & & \qquad\qquad E.code \parallel \\
 & & \qquad\qquad S_1.code \parallel \\
 & & \qquad\qquad S_2.code; \\
 & & \}
\end{array}
$$

Continuations and Code Generation

Continuation of a statement is an inherited attribute.

It is not an L-inherited attribute!

Code of statement is a synthesized attribute, but is dependent on its continuation.

Backpatching: *Make two passes to generate code.*

1. Generate code, leaving "holes" where continuation values are needed.
2. Fill these holes on the next pass.

What's left?

After intermediate code is generated,

- **Optimize** intermediate code using target machine-independent techniques.
  Examples:
  - constant propagation
  - loop-invariant code motion
  - dead-code elimination
  - strength reduction
- **Generate** final machine code
  Perform target machine-specific optimizations.