

Code Generation

Compiler Design

CSE 504

- 1 Syntax-Directed Code Generation
- 2 Machines
- 3 Expressions
- 4 Statements
- 5 Short-Circuit Code

Last modified: Fri Apr 08 2016 at 11:12:18 EDT

Version: 1.7 16:58:45 2016/01/29

Compiled at 11:16 on 2016/04/08



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:

$$E \longrightarrow E_1 + E_2 \quad \left\{ \begin{array}{l} \text{if } E_1.type \equiv E_2.type \equiv int \\ \quad E.type = int; \\ \text{else} \\ \quad E.type = float; \\ \end{array} \right\}$$

Code Generation via Syntax Directed Translation

Code Generation:

$$E \longrightarrow E_1 + E_2 \quad \left\{ \begin{array}{l} E.code = E_1.code \parallel \\ \quad E_2.code \parallel \\ \quad \text{"add"} \end{array} \right.$$

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.

$$S \longrightarrow \boxed{\text{load_immed } v} \longrightarrow v :: S$$

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

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

where $v = S[a]$.

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

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

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

- `add`: Add top two elements.

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

- `pop`: Remove top-most element.

$$v :: S \longrightarrow \boxed{\text{pop}} \longrightarrow S$$

Register Machines

- Machines with a (possibly unbounded) number of registers.
Let's call them $t_1, t_2, \dots$
- 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)

$$E \longrightarrow E_1 + E_2 \quad \{ E.code = E_1.code \parallel E_2.code \parallel \text{"add"} \}$$
$$E \longrightarrow \text{int} \quad \{ E.code = \text{"load_immed int.val"} \}$$
$$E \longrightarrow \text{id} \quad \{ E.code = \text{"load_immed id.addr"} \parallel \text{"load"} \}$$
$$E \longrightarrow \text{id} = E_1 \quad \{ E.code = \text{"load_immed id.addr"} \parallel E_1.code \parallel \text{"store"} \}$$

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

Variables and Addresses in Register Machines

- Abstract register machines have unbounded number of abstract registers.
- In the following, we assume all variables will be mapped to distinct abstract registers.
Each `id` has an attribute `id.addr` to represent this mapping.
- Not all program variables can be allocated in concrete registers.
E.g. variables representing formal parameters, variables that can be accessed from other threads, etc.
- Whenever possible, the registers of the abstract machine will be mapped to (a small, finite) set of registers of a concrete machine (e.g. MIPS) at a later phase.
- When abstract registers cannot be mapped concrete registers, separate “load/store” instructions will be generated to bring the values to and from memory.

Code Generation and Attributes (Register Machine)

$$\begin{aligned}
 E &\longrightarrow E_1 + E_2 && \{ E.t = \text{generate_new_temporary}(); \\
 & && E.code = E_1.code \parallel E_2.code \parallel \text{"add } E.t, E_1.t, E_2.t" \} \\
 E &\longrightarrow \text{int} && \{ E.t = \text{generate_new_temporary}(); \\
 & && E.code = [\text{"mov_immed } E.t, \text{int.val"}] \} \\
 E &\longrightarrow \text{id} && \{ E.t = \text{id.addr}; \\
 & && E.code = [] \} \\
 E &\longrightarrow \text{id} = E_1 && \{ E.t = E_1.t; \\
 & && E.code = E_1.code \parallel \text{"move id.addr, } E.t" \}
 \end{aligned}$$

- The above specifies the so-called “treewalk” code generation algorithm.
- If variables have to be stored in memory, the actions for last two rules with change:
 - $E \rightarrow \text{id}$ will generate a “load” instruction, and
 - $\text{id} = E$ will generate a “store” instruction as well.

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.
- Not all expressions make sense on the lhs of an assignment (e.g. constants, method calls).

The front end may enforce all or part of this restriction.

- 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:

$$E \rightarrow E + E$$
$$E \rightarrow L = E$$
$$E \rightarrow \text{int}$$
$$E \rightarrow L$$
$$L \rightarrow \text{id}$$
$$L \rightarrow L[E]$$

- 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 homeworks in this course, 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. Bounds Check
Register r_1 is the address of the heap segment. Register r_2 will then be set to the size of the segment.

Generating code for arrays:

Allocation

```
 $E \longrightarrow \text{new } T [ E_1 ]$   
    {  
       $E.t = \text{generate\_new\_temporary}();$   
       $E.code = E_1.code$   
        || "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  →  id      {
                L.t = L.at = id.addr;
                L.lcode = L.rcode = [ ];
                L.mem = reg;
                }

L  →  L1 [ E ] {
                L.at = generate_new_temporary();
                L.lcode = L1.rcode
                    || E.code
                    || "add L.at, L1.t, E.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

$$E \longrightarrow L \begin{cases} E.t = L.t; \\ E.code = L.rcode \\ \end{cases}$$

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

Generating code for arrays:

Assignments

```
E  →  L = E1
      {
        E.t = E1.t;
        if L.mem == reg
            assigncode = "move L.at, E1.t";
        else
            assigncode = "hstore L.at, E1.t";
        E.code = L.lcode
                || E1.code
                || assigncode;
      }
```

Addresses

- So far, we've considered a *flat* memory: a collection of cells accessed via “absolute” addresses.
- In general, most cells in memory will be accessible using a “relative” address:
 - ... represented as a pair (*base*, *offset*)
 - ... *base* is an address in memory relative to which all cells of a particular kind are accessed
 - ... *offset* is an integer that represents the address of a cell relative to its *base*.
 - ... *base* addresses may be known only at run-time.
 - ... *offset* is calculated or known at compile time.
- Relative addressing is very common:
 - global variables (relative to the base of static area)
 - local variables (relative to activation record)
 - fields of objects and structures (relative to object's base)
 - elements of arrays (relative to array's base)

Accessing structures

- Each identifier is associated with its *offset*:
id.offset
- The *base* for an identifier is determined based on its kind:
 - For local variables and formal parameters, *base* is the activation record pointer, **ARP** (determined at runtime)
depending on the implementation this could be a fixed offset from the ARP.
 - For fields in objects (i.e. instances) and fields in structures the base is the object/structure pointer.
 - For global variables, the base is the *static area*.
 - For field names in classes (i.e. “static” fields) the base is the area for this class in the *static area*.

Generating code for field/structure accesses

```
L  →  E.id  {  
    L.at = generate_new_temporary();  
    L.offset = generate_new_temporary();  
    L.base = E.t  
    L.lcode = E.rcode  
        || "move_immed L.offset, id.offset"  
        || "add L.at, L.base, L.offset"  
    L.t = generate_new_temporary();  
    L.rcode = L.lcode  
        || "hload L.t, L.at"  
    L.mem = relative;  
}
```

Code Generation for Statements

$$\begin{aligned} Ss &\longrightarrow S \ Ss_1 \quad \left\{ \begin{array}{l} Ss.code = S.code \\ \quad \parallel Ss_1.code; \end{array} \right. \\ &\hspace{15em} \} \\ Ss &\longrightarrow \epsilon \quad \{ Ss.code = [] \} \\ S &\longrightarrow E \ ; \quad \left\{ \begin{array}{l} S.code = E.code; \end{array} \right. \\ &\hspace{15em} \} \end{aligned}$$

Conditional Statements

```
S  →  if E, S1, S2 {  
      elselabel = get_new_label();  
      endlabel = get_new_label();  
      S.code =      E.code  
      ||           "beq E.t, 0, elselabel"  
      ||           S1.code;  
      ||           "jmp endlabel"  
      || "elselabel:"  
      ||           S2.code;  
      || "endlabel:"  
  }
```

Conditional Statements and Continuations

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

```
S  →  if E, S1, S2  {  
      S.begin = get_new_label();  
      S1.end = S2.end = S.end;  
      S.code = "S.begin:"  
              || E.code  
              || "beq E.t, 0, S2.begin"  
              || S1.code  
              || S2.code;  
      }
```

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.

$$Ss \longrightarrow S Ss_1 \quad \left\{ \begin{array}{l} Ss_1.end = Ss.end; \\ S.end = \textit{fallthrough}; \\ Ss.code = \dots \end{array} \right\}$$
$$S \longrightarrow E ; \quad \left\{ \begin{array}{l} \text{if } S.end == \textit{fallthrough} \\ \quad \textit{next} = [] \\ \text{else} \\ \quad \textit{next} = \textcolor{violet}{\text{"jump } S.end\text{"}} \\ S.code = \dots \\ \quad \parallel \textit{next}; \end{array} \right\}$$

Code Generation for Boolean Expressions

$$E \longrightarrow E_1 \ \&\& \ E_2 \quad \left\{ \begin{array}{l} E.t = \text{generate_new_temporary}(); \\ E.code = E_1.code \\ \quad \parallel E_2.code \\ \quad \parallel \text{"and } E.t, E_1.t, E_2.t"; \end{array} \right\}$$

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

$$E \longrightarrow E_1 \ \&\& \ E_2 \quad \left\{ \begin{array}{l} E.t = \text{generate_new_temporary}(); \\ \text{skip} = \text{generate_new_label}(); \\ E.code = E_1.code \\ \quad \parallel \text{"move } E.t, E_1.t" \\ \quad \parallel \text{"beq } E.t, 0, \text{skip}" \\ \quad \parallel E_2.code \\ \quad \parallel \text{"move } E.t, E_2.t" \\ \quad \parallel \text{"skip:"}; \end{array} \right\}$$

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

$$\begin{aligned} E &\longrightarrow E_1 \ \&\& \ E_2 \quad \left\{ \begin{array}{l} E_1.fail = E.fail; \\ E_2.fail = E.fail; \\ E_1.success = get_new_label(); \\ E_2.success = E.success; \\ E.code = E_1.code \parallel \\ \qquad \qquad \qquad "E_1.success:" \parallel \\ \qquad \qquad \qquad E_2.code \end{array} \right\} \\ E &\longrightarrow ! \ E_1 \quad \left\{ \begin{array}{l} E_1.fail = E.success; \\ E_1.success = E.fail; \\ E.code = E_1.code \end{array} \right\} \\ E &\longrightarrow true \quad \left\{ \begin{array}{l} E.code = "jmp, E.success" \end{array} \right\} \end{aligned}$$

Short-circuit code for Conditional Statements

```
S  →  if E, S1, S2 {  
      S.begin = get_new_label();  
      S1.end = S2.end = S.end;  
      E.success = S1.begin;  
      E.fail = S2.begin;  
      S.code = "S.begin:" ||  
               E.code ||  
               S1.code ||  
               S2.code;  
    }
```

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.