

Simple Code Generation

CS153: Compilers
Greg Morrisett

Code Generation

- PS3: Map Fish code to MIPS code
- Issues:
 - eliminating compound expressions
 - eliminating variables
 - encoding conditionals, short-circuits, loops

Source

```
datatype exp =
  Var of var
| Int of int
| Binop of exp * binop * exp
| Not of exp
| Or of exp * exp
| And of exp * exp
| Assign of var * exp
```

Target: MIPS

```
type label = string

datatype reg =
  R0 | R1 | R2 | ... | R31

datatype operand =
  Reg of reg
| Immed of Word32.word
```

MIPS continued

```
datatype inst =
  Add of reg * reg * operand
| Li of reg * Word32.word
| Slt of reg * reg * operand
| Beq of reg * reg * label
| Bgez of reg * label
| J of label
| La of reg * label
| Lw of reg * reg * Word32.word
| Sw of reg * reg * Word32.word
| Label of label | ...
```

Variables

Fish only has global variables.

These can be placed in the *data segment*.

```
.data
.align 0
x: .word 0
y: .word 0
z: .word 0
```

Variable Access

To compile $x = x+1$

```
la $3, x      ; load x's address
lw $2, 0($3) ; load x's value
addi $2,$2,1  ; add 1
sw $2, 0($3) ; store value in x
```

First Problem: Nested Expr's

- Source:
 - Binop(Binop(x,Plus,y),Plus,Binop(w,Plus,z))
- Target:
 - add rd, rs, rt

What should we do?

A Simple Strategy

- Given: Binop(A,Plus,C)
- translate A so that result ends up in a particular register (e.g., \$3)
- translate B so that result ends up in a different register (e.g., \$2)
- Generate: add \$2, \$3, \$2

Problems?

Strategy Fixed:

- Invariants:
 - results always placed in \$2
- Given: Binop(A,Plus,C)
- translate A
- save \$2 somewhere
- translate B
- Restore A's value into register \$3
- Generate: add \$2, \$3, \$2

For example:

Binop(Binop(x,Plus,y),Plus,Binop(w,Plus,z))

1. compute x+y, placing result in \$2
2. store result in a temporary t1
3. compute w+z, placing result in \$2
4. load temporary t1 into a register, say \$3
5. add \$2, \$3, \$2

Exp. Compilation

```
fun exp2mips(i:exp):inst list =
  case i of
  | Int j => [Li(R2, Word32.fromInt j)]
  | Var x => [La(R2,x), Lw(R2,R2,zero)]
  | Binop(i1,b,i2) =>
    let val t = new_temp()
    in
      (exp2mips i1) @ [La(R1,t), Sw(R2,R1,zero)]
      @ (exp2mips i2) @ [La(R1,t), Lw(R1,R1,zero)]
      @ (case b of
          Plus => [Add(R2,R2,Reg R1)]
          | ... => ...)
    end
  | Assign(x,e) => [exp2mips e] @
    [La(R1,x), Sw(R2,R1,zero)]
```

Statements

```

fun stmt2mips(s:F.stmt):inst list =
  case s of
    Exp e =>
      exp2mips e
  | Seq(s1,s2) =>
      (stmt2mips s1) @ (stmt2mips s2)
  | If(e,s1,s2) =>
      let val Else = new_label()
        val End = new_label()
      in
        (exp2mips b) @ [Beq(R2,R0,Else)] @
        (stmt2mips s1) @ [J End,Label Else] @
        (stmt2mips s2) @ [Label End]
      end
end

```

Statements Continued

```

| While(e,s) =>
  let val Test = new_label()
    val Top = new_label()
  in
    [J Test, Label Top] @
    (stmt2mips s) @
    [Label Test] @
    (exp2mips b) @
    [Bne(R2,R0,Top)]
  end
| For(e1,e2,e3,s) =>
  stmt2mips(Seq(Exp e1,While(e2,Seq(s,Exp e3))))

```

Lots of Inefficiencies:

- Compiler takes $O(n^2)$ time due to @.
- No constant folding.
- For "if (x == y) s1 else s2" we do a seq followed by a beq when we could just do a beq.
- For "if (b1 && b2) s1 else s2" we could just jump to s2 when b1 is false.
- Lots of la/lw and la/sw for variables.
- For e1 + e2, we always write out e1's value to memory when we could just cache it in a register.

Append:

Naïve append:

```

fun append nil y = y
  | append (h::t) y = h::(append t y)

```

Tail-recursive append:

```

fun revapp nil y = y
  | revapp (h::t) y = revapp t (h::y)
fun rev x = revapp x []
fun append x y = revapp (rev x) y

```

Accumulator-Based:

```

fun exp2mips' (i:exp) (a:inst list):inst list =
  case i of
  | Int w => Li(R2, Word32.fromInt w) :: a
  | Var x => revapp [La(R2,x), Lw(R2,R2,zero)] a
  | Binop(i1,b,i2) =>
    let val t = new_temp()
    in exp2mips' i1
      (revapp [La(R1,t), Sw(R2,R1,zero)]
        (exp2mips' i2
          (revapp [La(R1,t), Lw(R1,R1,zero)]
            ((case b of Plus => Add(R2,R2,Reg R1)
              | ... => ...))
            ::a))))
    end
  end
fun exp2mips (i:exp) = rev(exp2mips' i [])

```

Constant Folding: Take 1

```

fun exp2mips' (i:F.iexp) (a:inst list) =
  case i of
  | Int w => Li(R2, Word32.fromInt w) :: a
  | Binop(i1,Plus,Int 0) => exp2mips' i1 a
  | Binop(Int i1,Plus,Int i2) =>
    exp2mips' (Int (i1+i2)) a
  | Binop(Int i1,Minus,Int i2) =>
    exp2mips' (Int (i1-i2)) a
  | Binop(b,i1,i2) => ...

```

Why isn't this great? How can we fix it?

Conditional Contexts

Consider: if (x < y) then S1 else S2

```

    slt $2, $2, $3
    beq $2, ELSE
    S1
    j END
ELSE:
    S2
END:

```

Observation

- In most contexts, we want a value.
- But in a testing context, we jump to one place or another based on the value, and otherwise never use the value.
- In many situations, we can avoid materializing the value and thus produce better code.

For Example:

```

fun bexp2mips (e:exp) (t:label) (f:label) =
  case e of
  | Int 0 => [J f]
  | Int _ => [J t]
  | Binop (e1, Eq, e2) =>
    let val t = temp()
    in
      (exp2mips e1) @
      [La (R1, t), Sw (R2, R1, zero)] @
      (exp2mips e2) @
      [La (R1, t), Lw (R1, R1, zero),
       Bne (R1, R2, f), J t]
    end
  | _ => (exp2mips e1) @ [ Beq (R2, R0, f), J t ]

```

Global Variables:

We treated all variables (including temps) as if they were globals:

- set aside data with a label
- to read: load address of label, then load value stored at that address.
- to write: load address of label, then store value at that address.

This is fairly inefficient:

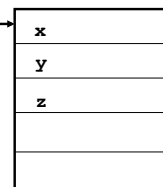
- e.g., $x+x$ involves loading x 's address twice!
- lots of memory operations.

Register Allocation:

- One option is to allocate a register to hold a variable's value:
 - Eliminates the need to load an address or do memory operations.
 - Will talk about more generally later.
 - Of course, in general, we can't avoid it when code has more (live) variables than registers.
- Can we at least avoid loading addresses?

Frames:

- Set aside one block of memory for all of the variables.
- Dedicate $\$30$ (aka $\$fp$) to hold the base address of the block.
- Assign each variable a position within the block ($x \rightarrow 0, y \rightarrow 4, z \rightarrow 8$, etc.)
- Now loads/stores can be done relative to $\$fp$:



Before and After:

`z = x+1`

```
lda $1,x
lw $2,0($1)
addi $2,$2,1
lda $1,z
sw $2,0($1)
```

```
lw $2,0($fp)
addi $2,$2,1
sw $2,8($fp)
```

Lowering

- Get rid of nested expressions before translating
 - Introduce new variables to hold intermediate results
 - Perhaps do things like constant folding
- For example, $a = (x + y) + (z + w)$ might be translated to:


```
t0 := x + y;
t1 := z + w;
a := t0 + t1;
```

12 instructions (9 memory)

```
t0 := x + y;    lw $v0, <xoff>($fp)
                lw $v1, <yoff>($fp)
                add $v0, $v0, $v1
                sw $v0, <t0off>($fp)
t1 := z + w;    lw $v0, <zoff>($fp)
                lw $v1, <woff>($fp)
                add $v0, $v0, $v1
                sw $v0, <t1off>($fp)
a := t0 + t1;   lw $v0, <t0off>($fp)
                lw $v1, <t1off>($fp)
                add $v0, $v0, $v1
                sw $v0, <aoff>($fp)
```

Still...

We're doing a lot of stupid loads and stores.

- We shouldn't need to load/store from temps!
- (Nor variables, but we'll deal with them later...)

So another idea is to use registers to hold the intermediate values instead of variables.

- Of course, we can only use registers to hold some number of temps (say k).
- Maybe use registers to hold first k temps?

For example:

```

t0 := x;      # load variable
t1 := y;      # load variable
t2 := t0 + t1; # add
t3 := z;      # load variable
t4 := w;      # load variable
t5 := t3 + t4; # add
t6 := t2 + t5; # add
a := t6;      # store result

```

Then: 8 instructions (5 mem!)

- Notice that each little statement can be directly translated to MIPS instructions:

```

t0 := x;      --> lw $t0,<xoff>($fp)
t1 := y;      --> lw $t1,<yoff>($fp)
t2 := t0 + t1; --> add $t2,$t0,$t1
t3 := z;      --> lw $t3,<zoff>($fp)
t4 := w;      --> lw $t4,<woff>($fp)
t5 := t3 + t4; --> add $t5,$t3,$t4
t6 := t2 + t5; --> add $t6,$t2,$t5
a := t6;      --> sw $t6,<aoff>($fp)

```

Recycling:

- Sometimes we can recycle a temp:

```

t0 := x;      t0 taken
t1 := y;      t0,t1 taken
t2 := t0 + t1; t2 taken (t0,t1 free)
t3 := z;      t2,t3 taken
t4 := w;      t2,t3,t4 taken
t5 := t3 + t4; t2,t5 taken (t3,t4 free)
t6 := t2 + t5; t6 taken (t2,t5 free)
a := t6;      (t6 free)

```

Tracking Available Temps:

Hmmm. Looks a lot like a stack...

```

t0 := x;      t0
t1 := y;      t1,t0
t0 := t0 + t1; t0
t1 := z;      t1,t0
t2 := w;      t2,t1,t0
t1 := t1 + t2; t1,t0
t1 := t0 + t1; t1
a := t1;      <empty>

```

Finally, consider:

$(x+y) * x$

```
t0 := x;           # loads x
t1 := y;
t0 := x+y;
t1 := x;           # loads x again!
t0 := t0*t1;
```

Good Compilers: (not this proj!)

Introduces temps as described earlier:

- It lowers the code to something close to assembly, where the number of resources (i.e., registers) is made explicit.
- Ideally, we have a 1-to-1 mapping between the lowered intermediate code and assembly code.

Performs an analysis to calculate the *live range* of each temp:

- A temp t is live at a program point if there is a subsequent read (use) of t along some control-flow path, without an intervening write (definition).
- The problem is simplified for *functional* code since variables are never re-defined.

Interference Graphs:

From the live-range information for each temp, we calculate an *interference graph*.

- Temps t_1 and t_2 *interfere* if there is some program point where they are both live.
- We build a graph where the nodes are temps and the edges represent interference.
- If two temps interfere, then we cannot allocate them to the same register.
- Conversely, if t_1 and t_2 do not interfere, we can use the same register to hold their values.

Register Coloring

- Assign each node (temp) a register such that if t_1 interferes with t_2 , then they are given distinct colors.
 - Similar to trying to "color" a map so that adjacent countries have different colors.
 - In general, this problem is NP complete, so we must use heuristics.
- Problem: given k registers and $n > k$ nodes, the graph might not be colorable.
 - Solution: spill a node to the stack.
 - Reconstruct interference graph & try coloring again.
 - Trick: spill temps that are used infrequently and/or have high interference degree.

Example:

```

a := (x+y)*(x+z)

t0 := x
t1 := y
t2 := z
t3 := t0+t1
t4 := t0+t2
t5 := t3*t4
a := t5
    
```

Diagram illustrating the live ranges for variables t_0 through t_5 in the code snippet. The live ranges are indicated by colored brackets on the right side of the code:

- live range for t_1 (red)
- live range for t_0 (blue)
- live range for t_2 (green)
- live range for t_3 (yellow)
- live range for t_4 (purple)
- live range for t_5 (black)

Graph:

```

a := (x+y)*(x+z)

t0 := x
t1 := y
t2 := z
t3 := t0+t1
t4 := t0+t2
t5 := t3*t4
a := t5
    
```

Diagram illustrating the control flow graph for the code snippet. The nodes are t_0 through t_5 and a . The edges represent the flow of control between these nodes.

Coloring:

```

a := (x+y)*(x+z)

t0 := x
t1 := y
t2 := z
t3 := t0+t1
t4 := t0+t2
t5 := t3*t4
a := t5
    
```

Diagram illustrating the control flow graph for the code snippet. The nodes are t_0 through t_5 and a . The nodes t_1 , t_2 , t_3 , and t_4 are shaded gray, indicating they are not yet colored.

Coloring:

```

a := (x+y)*(x+z)

t0 := x
t1 := y
t2 := z
t3 := t0+t1
t4 := t0+t2
t5 := t3*t4
a := t5
    
```

Diagram illustrating the control flow graph for the code snippet. The nodes are t_0 through t_5 and a . The nodes t_1 , t_2 , t_3 , and t_4 are colored yellow, pink, green, and purple respectively, indicating they are now colored.

Assignment:

```

a := (x+y)*(x+z)

t0 := x
t1 := y
t2 := z
t3 := t0+t1
t4 := t0+t2
t5 := t3*t4
a := t5

```

Legend:

- t0
- t1
- t2
- t3

Rewrite:

```

a := (x+y)*(x+z)

t0 := x
t1 := y
t2 := z
t3 := t0+t1
t0 := t0+t2
t0 := t3*t0
a := t0

```

Legend:

- t0
- t1
- t2
- t3

Generate Code

```

a := (x+y)*(x+z)

t0 := x    --> lw $t0,<xoff>($fp)
t1 := y    --> lw $t1,<yoff>($fp)
t2 := z    --> lw $t2,<zoff>($fp)
t3 := t0+t1 --> add $t3,$t0,$t1
t0 := t0+t2 --> add $t0,$t0,$t2
t0 := t3*t0 --> mul $t0,$t3,$t2
a := t0    --> sw $t0,<aoff>($fp)

```