CSE 401/M501 – Compilers

x86-64, Running MiniJava,
Basic Code Generation and Bootstrapping

Hal Perkins & CSE 401/M501 staff

Spring 2023

Administrivia

- Final part of the project – codegen – out now
  - Due in a week(!), but this is shorter than semantics / type checking.
  - Biggest hurdle is getting started
    - Goal: get System.out.println(17) in main method working ~~by~~ before the weekend
  - Once that's done, look at writeup for a reasonable order to add codegen for different language features

Running MiniJava Programs

- To run a MiniJava program
  - Space needs to be allocated for a stack and a heap
  - `%rsp` and other registers need to have sensible initial values
  - We need some way to allocate storage (for `new`) and communicate with the outside world

Bootstrapping from C

- Idea: take advantage of the existing C runtime library
- Use a small C main program to call the MiniJava main method as if it were a C function
- C's standard library provides the execution environment and we can call C functions from compiled code for I/O, malloc, etc.

Assembler File Format

- Compiler output is an assembly language program (ascii .s)
- GNU syntax is roughly this (src/runtime/demo.s in project starter code is a runnable asm example, although not generated by a MiniJava compiler)

```
    .text                 # code segment
    .globl asm_main       # label at start of compiled static main
    <generated code>
asm_main:                 # start of compiled "main"
    ...
    .data
    <generated method tables>
    # repeat .text/.data as needed
    …
    end
```

External Names

- In a Linux environment, an external symbol is used as-is (`xyzzy`)
- In Windows and x86-64 MacOS, an external symbol xyzzy is written in asm code as `_xyzzy` (leading underscore)
- Your compiler needs to generate code that runs on attu using Linux conventions, but if you want to support the other as an option, feel free to add a compiler switch or something

Generating .asm Code

- Suggestion: isolate the actual compiler output operations in a handful of routines
  - Usual modularity reasons & saves some typing
  - Some possibilities

```
// write code string s to .asm output
void gen(String s) { … }
// write "op  src,dst" to .asm output
void genbin(String op, String src, String dst) { … }
// write label lbl to .asm output as "lbl:"
void genLabel(String lbl) { … }
```

  - A handful of these methods should do it

A Simple Code Generation Strategy

- Goal: quick 'n dirty correct code, "optimize" later if time
- Traverse AST primarily in execution order and emit code during the traversal
  - Codegen visitor might want to traverse the tree in ad-hoc ways depending on sequence that parts need to appear in the asm code
- Treat the x86-64 as a 1-register machine with a stack to hold additional intermediate values(!)
  - Ugly code, but will work – better later if there's time

(The?) Simplifying Assumption

- Store all values (reference, int, boolean) in 64-bit quadwords
  - Natural size for 64-bit pointers, i.e., object references (variables of class types)
  - C's "long" size for integers
    - Use `int64_t` or `uint64_t` in any C code that interacts with MiniJava generated code to guarantee size (declared in `<stdint.c>`)

Before Codegen Visitor Pass…

- Need an initial pass through class and method symbol tables to assign locations to variables
  - Method local variables: successive offsets in the stack frame relative to %rbp (-8, -16, …)
    - Also for parameters – place to store copies in stack frame
  - Object instance variables: successive offsets from the start of the object (+0 is vtable pointer, instance variables at +8, +16, …)
- This will also compute the size of each stack frame or object which is needed later

x86 as a Stack Machine

- Idea: Use x86-64 stack for expression evaluation with `%rax` as the logical "top" of the stack (initially empty)
- Invariant: Whenever an expression (or part of one) is evaluated at runtime, the generated code leaves the result in `%rax`
- If a value needs to be preserved while another expression is evaluated, push `%rax`, evaluate, then pop when first value is needed
  - Remember: ***always* pop what you push**
  - Will produce lots of redundant, but correct, code
- Examples below follow code shape examples, but with more details about code generation

Example: Generate Code for Constants and Identifiers

Integer constants, say 17

gen(movq $17,%rax)

- leaves value in %rax

Local variables (any type – int, bool, reference)

gen(movq varoffset(%rbp),%rax)

Instance variables (“this.var”)

gen(movq varoffset(%rdi),%rax)

Example: Generate Code for exp1 + exp2

Visit exp1

- generates code to evaluate exp1 with result in %rax

gen(pushq %rax)

- push exp1 onto stack

Visit exp2

- generates code for exp2; result in %rax

gen(popq %rdx)

- pop left argument into %rdx; clean up stack

gen(addq %rdx,%rax)

- perform the addition; result in %rax

Example: var = exp; (1)

Assuming that var is a local variable

Visit node for exp

- Generates code to eval exp and leave result in %rax

gen(movq %rax,offset_of_variable(%rbp))

Similar code if var is part of an object, but use pointer to the object instead of %rbp

Example: var = exp; (2)

If var is a more complex expression (object or array reference, for example)

visit var

gen(pushq %rax)

- push lvalue (address) of variable or object containing variable onto stack

visit exp

- leaves rhs value in %rax

gen(popq %rdx)

gen(movq %rax,appropriate_offset(%rdx))

Example: Generate Code for obj.f(e1,e2,…en)

In principal the code should work like this:

Visit obj

- leaves reference to object in %rax

gen(movq %rax,%rdi)

- "this" pointer is first argument

Visit e1, e2, …, en. For each argument,

- gen(movq %rax,%correct_argument_register)

generate code to load method table pointer located at 0(%rdi) into some register, probably %rax

generate call instruction with indirect jump

Method Call Complications

- Big one: code to evaluate any argument might clobber argument registers (i.e., computing an argument value might require a method call)
  – Possible strategy to cope on next slides, but feel free to do something better
- And more: clobbers *current* method's %rdi (this ptr)
  – Save it on method entry; reload after call (or on every use)
- Other one: what if a method has too many parameters?
  – OK for CSE 401/M501 to assume all methods have ≤ 5 parameters plus "this" – do better if you want

Method Calls in Parameters

- Suggestion to avoid trouble:
  - Evaluate parameters and push them on the stack
  - Right before the call instruction, pop the parameters into the correct registers
- But....

Stack Alignment (1)

- Above ~~idea~~ hack works provided we don't call a method while an odd number of parameter values are pushed on the stack!
  - (violates 16-byte alignment on method call…)
- We have a similar problem if an odd number of intermediate values are pushed on the stack when we call a function while evaluating an expression
  - (We might get away with it if it only involves calls to our own generated, not library, code, but it would be wrong* to do that)

    *i.e., might "work", but it's not the right way to solve the problem

Stack Alignment (2)

- Workable solution: keep a counter in the code generator of how much has been pushed on the stack. If needed, emit extra gen(pushq %rax) to push a useless value and align the stack before generating a call instruction
  - Be sure to pop it after!!
- Another (cleaner, but more work) solution: make stack frame big enough and use movq instead of pushq to store arguments and temporaries
  - Will need extra bookkeeping to keep track of how much to allocate and how temps are used and where they are in the stack frame

Sigh…

- Multiple registers for method arguments is a big win compared to pushing on the stack, but complicates our life since we do not have a fancy register allocator
- Feel free to do better than this simple push/pop scheme – but remember, simple and works wins over fancy and not finished or broken

Code Gen for Method Definitions

- Generate label for method

  Classname$methodname:

- Generate method prologue

  push %rbp, copy %rsp to %rbp, subtract frame size (multiple of 16) from %rsp

- Visit statements in order
  - Method epilogue is normally generated as part of a return statement (details shortly)
  - In MiniJava the return is generated after visiting the method body to generate its code

Registers again…

- Method parameters are in registers
- But code generated for methods also will be using registers, even if there are no calls to other methods
- So how do we avoid clobbering parameters?
- Suggestion: Allocate space in the stack frame and save copies of all parameter registers on method entry. Use those copies as local variables when you need to reference a parameter.

Example: return exp;

- Visit exp; this leaves result in %rax where it should be
- Generate method epilogue (copy %rbp to %rsp, pop %rbp) to unwind the stack frame; follow with ret instruction
  - Can use leave instead of movq/popq to unwind the stack, but the separate instructions might be a little easier to debug if something isn’t right

Control Flow: Unique Labels

- Needed in code generator: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, …)
  - Improvement: a set of methods that generate different kinds of labels for different constructs (can really help readability of the generated code)
    - (while1, while2, while3, …; if1, if2, …; else1, else2, …; endif1, endif2, … .)

Control Flow: Tests

- Recall that the context for compiling a boolean expression is:
  - Label or address of jump target
  - Whether to jump if true or false
- So the visitor for a boolean expression should receive this information from the parent node

Example: while(exp) body

- Assuming we want the test at the bottom of the generated loop…

  gen(jmp testLabel)

  gen(bodyLabel:)

  visit body

  gen(testLabel:)

  visit exp (condition) with target=bodyLabel and sense="jump if true"

Example: exp1 < exp2

- Similar to other binary operators
- Difference: surrounding (parent) context is a target label and whether to jump if true or false
- Code

```
visit exp1
gen(pushq %rax)
visit exp2
gen(popq %rdx)
gen(cmpq %rdx,%rax)
gen(condjump targetLabel)
```

  - appropriate conditional jump depending on sense of test

Boolean Operators

&& (and || if you add it)

- Create label(s) needed to skip around the two parts of the expression
- Generate subexpressions with appropriate target labels and conditions

!exp

- Generate exp with same target label, but reverse the sense of the condition

Reality check

- Lots of projects in the past have evaluated all booleans to get 1 or 0, then tested that value for control flow
- Would be nice to do better (as above), but "simple and works…"

- (And we need to be able to generate the 0/1 anyway for storable boolean expressions)

Join Points

- Loops and conditional statements have join points where execution paths merge
- Generated code must ensure that machine state will be consistent regardless of which path is taken to get there
  - i.e., the paths through an if-else statement must not leave a different number of values pushed onto the stack
  - If we want a particular value in a particular register at a join point, both paths must put it there, or we need to generate additional code to move the value to the correct register
- With our simple 1-accumulator model of code generation, this should usually be true without needing extra work; with better use of registers it becomes a bigger issue
  - With more registers, would need to be sure they are used consistently at join point regardless of how we get there

Bootstrap Program

- The bootstrap is a tiny C program that calls your compiled code as if it were an ordinary C function
- It also contains some functions that compiled code can call as needed
  - MiniJava "runtime library"
  - Add to this if you like
    - Sometimes simpler to generate a call to a new library routine instead of generating in-line code
    - Suggestion: do this for "exit if subscript out of bounds" check
- File: src/runtime/boot.c in project starter code

Bootstrap Program Sketch

```
#include <stdio.h>
extern void asm_main();  /* compiled code */
/* execute compiled program */
void main( ) { asm_main(); }
/* write x to standard output */
void put(int64_t x) { … }
/* return a pointer to a zeroed-out block of memory at
   least nBytes large (or null on failure) */
char* mjcalloc(size_t nBytes) { return calloc(1,nBytes); }
```

Main Program Label

- Compiler needs special handling for the publicstaticvoid `main` method label
  - Label must be the same as the one declared extern in the C bootstrap program and declared .globl in the .s asm file
  - asm_main used above
    - Could be changed, but probably no point
    - Why not “main”? (Hint: where is the real `main`?)

Interfacing to “Library” code

- Trivial to call “library” functions
- Evaluate parameters using the regular calling conventions
  - But no “this” parameter since we’re calling C code
- Generate a call instruction using the “library” function label
  - (External names need leading _ in Windows, OS X)
  - Linker will hook everything up

System.out.println(exp)

MiniJava's "print" statement

```
<compile exp; result in %rax>
movq    %rax,%rdi    # load argument register
call    put          # call external put routine
```

- If the stack is not properly 16-byte aligned when call is executed, calls to external C or library code can cause a runtime error (*will* cause error halt on x86-64 MacOS)

If you want to run code on an Intel Mac…

- Compiled code should work on a mac, but need to deal with a few things:
  - External labels need to start with `_` (e.g., `_put`)
  - `%rsp` *must* be 16-byte aligned when `call` is executed (should be anyway, but Linux will probably let you get away with 8-byte alignment)
  - Addressing modes: assembler might reject `leaq label,%rax`. Use `leaq label(%rip),%rax` instead (explicit base reg.; also works fine on Linux)
  - Hard to run gdb on a mac. Use clang/lldb instead
  - New annoyance on MacOS Ventura: may need to include `.align 8` in assembler code before each vtable to stop linker complaints
- And be ***sure*** that things run on attu/cse vm Linux in your final version!!! (*No* external `_labels`)

And That's It…

- We've now got enough on the table to complete the compiler code generator
- Past & Future Attractions
  - Lower-level IR and control-flow graphs
  - Mid part of compiler (optimizations)
  - Back end (industrial-strength instruction selection, scheduling, and register allocation)