CSE P 501 – Compilers

x86-64, Running MiniJava,
Basic Code Generation and Bootstrapping
Hal Perkins
Winter 2016
Administrivia

- Codegen part of project posted now
  - Final push deadline Sun. 3/13, 11pm – no extensions
  - Project report due next night (will post info later)
  - Project meetings Tue. & Wed. 3/15-16 – scheduling later
  - Grading of previous parts – long story...
- HW4 out (now | soon) [pick one]
  - Review problems on analysis/opt/IRs/SSA
  - Due Monday night, 2/29 – no extensions; solutions posted next day
- Exam next Thur. 3/3 @UW and @Microsoft, 6:30-8:00
  - Topic list & old exams on web now
  - OK to use textbook + slides + topic list + 2 handouts (x86-64, MiniJava grammar) during exam; laptops ok if only used to access items linked from exam resources or topics pages
Codegen/Execution for MiniJava

- Build on Codeshape lectures

- More specifics here about execution environment and compiler code gen routines
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 (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 source file (.s)
- GNU syntax is roughly this (asm samples included in starter src/runtime directory, links to gcc/as refs. on project page)

```assembly
# comments start with # and go to end of the line
.text                      # code segment
.globl asm_main             # start of compiled static main
<generated code>            # repeat .text/.data as needed
asm_main:                   # start of compiled “main”
    ...
.data
    <generated method tables>
    # repeat .text/.data as needed
    ...
end
```

UW CSE P 501 Winter 2016
External Names

- In a Linux environment, an external symbol is used as-is (xyzzy)
- In Windows and OS X, an external symbol xyzzy is written in asm code as _xyzzy (leading underscore)
- Adapt to whatever environment you’re using
  - but what you turn in should run on CSE linux using the Linux conventions
  - Exceptions? Check with course staff first
Generating .asm Code

- Suggestion: isolate the actual compiler output operations in a handful of routines
  - Modularity & saves some typing
  - Possibilities
    
    ```java
    // 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 L to .asm output as “L:”
    void genLabel(String L) { ... }
    ```

- 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 in visitor methods
  – Visitor may traverse the tree in ad-hoc ways depending on the sequence in which parts need to be written to the code assembler file
• Treat the x86 as a 1-register machine with a stack for 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, e.g., object references (variables of class types)
  – C’s “long” size for integers
    • Better to use int64_t or uint64_t in C code to guarantee size (declared in <stdint.h>)
x86 as a Stack Machine

- Idea: Use x86-64 stack for expression evaluation with %rax as the “top” of the stack
- 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 some more details about code generation
Example: Generate Code for Constants and Identifiers

Integer constants, say 17

\[ \text{gen(movq } $17,\%rax) \]
- leaves value in \%rax

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

\[ \text{gen(movq varoffset(\%rbp),\%rax)} \]
Example: Generate Code for $exp1 + exp1$

Visit $exp1$
- generate code to evaluate $exp1$ with result in %rax
  gen(pushq %rax)
  - push $exp1$ onto stack

Visit $exp2$
- generate 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: \texttt{var = exp; (1)}

Assuming that \texttt{var} is a local variable:

Visit node for \texttt{exp}

- Generates code to eval \texttt{exp} and leave result in \%rax

\texttt{gen(movq \%rax,offset_of_variable(\%rbp))}
Example: var = exp; (2)

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

visit var
    gen(pushq %rax)
    • push reference to (i.e., 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 \texttt{obj.f(e1,e2,...,en)}

In principal the code should work like this:

\begin{itemize}
  \item Visit \texttt{obj} \hfill leaves reference to object in \%rax
  \item \texttt{gen(movq \%rax,\%rdi)} \hfill “this” pointer is first argument
\end{itemize}

Visit \texttt{e1, e2, ..., en}. For each argument,

\begin{itemize}
  \item \texttt{gen(movq \%rax,correct_argument_register)}
\end{itemize}

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., method call in some parameter value)
  — Possible strategy to cope on next slides, but feel free to do something better

• Other one: what if a method has too many parameters?
  — OK for P 501 to assume that all methods have \( \leq 5 \) 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 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 align stack before generating a call instruction
  – Be sure to pop it after!!

• Another solution: make stack frame big enough and use `movq` instead of `pushq` to store arguments and temporaries
  – Will need some extra bookkeeping to keep track of how much to allocate and how temps are used
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 broken
Code Gen for Method Definitions

- Generate label for method
  
  
  \[
  \text{Classname}\$\text{methodname}:
  \]

- Generate method prologue
  
  \[
  \begin{aligned}
  &\text{Push } %rbp, \text{ copy } %rsp \text{ to } %rbp, \text{ subtract frame size from } \\
  &\text{ }
  \end{aligned}
  \]

- Visit statements in order
  
  - Method epilogue is normally generated as part of each return statement (next)
  
  - 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
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, ...; fi1, fi2, ... .)
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: \texttt{exp1 < exp2}

- Similar to other binary operators
- Difference: context is a target label and whether to jump if true or false
- Code
  - visit \texttt{exp1}
  - \texttt{gen(pushq \%rax)}
  - visit \texttt{exp2}
  - \texttt{gen(popq \%rdx)}
  - \texttt{gen(cmpq \%rdx,\%rax)}
  - \texttt{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 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...”
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 words 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 a 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
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
  - Mini “runtime library”
  - Add to this if you like
    - Sometimes simpler to generate a call to a hand-written library routine instead of generating in-line code – implementer tradeoff
# 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 block of memory at least nBytes large (or null on failure) */
char* mjmalloc(size_t nBytes) { return calloc(1, nBytes); }
Main Program Label

- Compiler needs special handling for the `public static void 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
• Generate a call instruction using the 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 kept 16-byte aligned, calls to external C or library code can cause a runtime error (will cause error halt on OS X)
And That’s It...

• We’ve now got enough on the table to complete the compiler project

• Coming Attractions
  – Back end (instruction selection and scheduling, register allocation)
  – and more...
    (including an exam)