CS444 Assignment 5 Design Document

An, Qin Kuan, Wei Heng McIntosh, Daniel David

April 8, 2020

Overview

For code generation phase, the compiler is organized in a standard way: a top-down traversal of the AST is performed. Each AST node emits appropriate assembly code and in most cases recursively calls "code generate" method of child nodes.

Stack Layout and Function Call

To utilize machine instructions "push" and "pop" to the maximum extent, the notion of a "stack" in the generated assembly code matches x86 convention: grow from higher to lower addresses, and stack pointer points at the lowest address in memory containing valid data.

To perform a function call with n arguments, the argument list is evaluated from left to right as required by Java Language Specification, and the arguments are pushed onto the stack in that order as well. For non-static methods, the implicit "this" parameter is treated as the very first argument, and is evaluated and pushed normally.

After all arguments have been pushed onto the stack, the address of the procedure is figured out and the "call" assembly instruction is used to branch to the procedure. For static methods, since they don't exhibit polymorphic behaviors the exact procedure to invoke can be determined at compile time by simply jumping to the corresponding label for that static method. For non-static methods, this involves looking up the method selector table associated with the runtime object and will be explained later.

"call" instruction additionally pushes the next instruction pointer onto the stack before branching, therefore upon entry of the target procedure the top of the stack is no longer the rightmost parameter, which is at esp+4 instead. The first thing that all procedures do upon entry is to push the old frame pointer (ebp) onto the stack and overwrite it with the stack pointer (esp). Note that at this point there is a gap of 8 bytes between the frame pointer and the rightmost argument and special care needs to be taken when generating assembly code to access arguments.

For local variables, they are pushed onto the stack as they appear and popped as they go out of scope. Each procedure maintains a record of all local variables that ever existed, and their index. This index is used when generating assembly code to access local variables through the frame pointer. Note that this design minimizes work to preallocate stack space at the cost of complicated variable access since function parameters are "above" the frame pointer and local variables "below". Depending on the case, different index calculation has to be used.

Finally, the procedure returns to caller by undoing its preamble: overwrites the stack pointer (esp) with the frame pointer (ebp) and pop the old frame pointer off the stack. Now the stack pointer points at the next instruction pointer. "ret" instruction is used to return to caller.

Object Layout

Naming of Labels

For naming of labels for methods, we mimic what has been already setup for native methods to reduce some special cases. The keyword "NATIVE" is dropped for non-native methods. The label is a concatenation of canonical name of type, ".", name of method, then followed by canonical name of parameter types separated by underscore, due to method overloading.

For static fields, they are named canonical name of type followed by a dot and the name of the field.

For type information, the corresponding label is the canonical name followed by "_typeinfo". This is used by ClassInstanceCreation and ArrayInstanceCreation to fill in type tag correctly.

The above are all the labels that need to be globally unique since they are declared global. For labels of branches for conditional statements and constants, since they are only used within the declaring assembly source file, they are named after arbitrary string and numerical counters.

Non-static Method Resolution

To resolve non-static method invocation at runtime, a method selector table is computed for each concrete class at compile time and stored in the type information structure in the data section of the assembly code. Instead of maintaining two separate interface table and virtual function table, we merged them in a single table. Each uniquely declared non-static method is assigned a global index across all types. When generating assembly code to index the method selector table, the global index of the method is used as an offset into the table. By default, all function pointers in a method selector table is assigned null. Each concrete class then overwrites methods in this table with its implementations by looking at its contain set computed earlier. In some cases, the overriding relationship between methods is established in parent classes, and a concrete implementation may be overriding multiple methods upstream and this case needs to be handled.

String-Related Functionalities

For string constants, they are layed out in the data section of the executable as a runtime array of characters (chars[]). They have proper fields (class tag, length, etc.) and data filled out at compiled time, and the content appears identical to a runtime char[]. When a string literal appears in a Java program, the String constructor with a character array parameter is invoked with a pointer to the corresponding string constant as the argument. The disruption to runtime code and special cases required is minimized using this approach.

Casting and InstanceOf

Testing

We primarily relied on the provided test cases for assignment 5 for testing. We also used test cases for assignment 3 extensively for code generation since they deal with class hierarchy and interface method a lot, which are not test thoroughly by assignment 5 test cases. Our automated test facility resides entirely in the compiler itself. After generating assembly code implementing the Java program, the compiler is able to perform assemble, link and run step if configured to do so. This is achieved via spawning nasm and ld processes and passing them the right arguments. The compiler then, acting very similar to a debugger like gdb, spawns the linked program and retrieves its return value.

One particular powerful gain through this setup is the ability to obtain crash dumps easily. Normally when an executable is run from the command line, the printout of crashes is not very helpful since it only shows the high level cause (Segmentation Fault/Floating Point Exception) and nothing else. With a custom written program (the compiler) spawning all the hundred test programs, we can use Linux's ptrace API to obtain register values and memory contents at the point of crash and print them out. What we get is a complete test result and crash dumps for all test programs. This provides a direct feedback on efforts of bug fixing and makes iteration time much quicker. We also utilized addr2line utility in the compiler to print out line number of assembly source code at point of crash. This minimized labor work of invoking tools manually and we can focus on fixing bugs fast.