CS 444 - Assignment 5

dburgoyn
udimitri
x2fang

Section 001
2015-04-01

This document describes the design of the code generation phase of our compiler; discusses challenges that we encountered during this phase and how we tried to overcome them; and explains the testing that we did before submitting to Marmoset.

1 Conventions and planning

1.1 Type IDs

We assigned a unique Type ID to every type in the program; this is an integer used to identify the inner type of an array of primitive types, as well as to determine the concrete type of heap-allocated objects at runtime. Type IDs were allocated using the following convention:

Type	**ID**
`boolean`	-1
`byte`	-2
`char`	-3
`short`	-5
`int`	-6
`null` literal	-7
T`[]`	0
T	1 + T's topological order

1.2 Object layout

We store instances of both primitive and reference types in 32-bit values, even when fewer bits could be used. For example, the `boolean` value `true` is represented as `0x00000001`, the `null` literal is represented as a 32-bit zero pointer, and object references are represented by 32-bit pointers.

1.2.1 Primitive conversions

When converting between integral types, either implicitly (due to assignment, binary operations, or unary negation) or explicitly (due to casts), we performed sign extension (for widening conversions) or truncation (for narrowing conversions) as appropriate. The `movsx` and `movzx` instructions were used to perform widening conversions. When widening a `char` to an `int`, we pad the upper 16 bits of the `int` with zeros rather than performing sign extension, since `char`s are unsigned 16-bit values.

1.2.2 Reference type layout

A heap-allocated object with runtime type T holds the Type ID of T in the object's first 32 bits.

Consider the case where $T = U$`[]`. If U is a primitive type, the next 32 bits of the object hold the Type ID of U. If U is a reference type, the next 32 bits of the object hold a pointer to U's subtype table (see below). In either case, the third 32-bit value in an object of type $T = U$`[]` is an integer containing the length of the array. If the length of the array is n, there are n subsequent 32-bit entries in the object, each corresponding to an entry in the array.

If T is a class type, then an object of type T has as many 32-bit values following its Type ID as it has non-static fields (both declared and inherited). The fields inherited from T's superclass appear first, in the same order that they would appear in a concrete instance of T's superclass. The fields which were declared in T follow in the order in which they appear in the source code.

Creation of arrays and other objects at runtime begins with a call to `__malloc` with the appropriate size placed in `eax`. The object's Type ID is then placed in the first 32 bits of the allocated memory. Furthermore if we are allocating an array type, we populate the second two fields in its header as described above. Then, in either case, we zero out the rest of the object. If we are creating a non-array reference type, then we additionally invoke the appropriate constructor code, passing the newly-allocated object's address as the `this` argument.

1.3 Virtual function tables and runtime type hierarchy

For each declared type in T in the program, and for every subtype S of T. we generate a vtable $V(T, S)$ which provides a mapping from methods that could possibly be called on a pointer of static type T to the concrete implementation of those methods by S. The mapping is defined as follows: let $m_0, m_1, ..., m_k$ be the list of all methods that could possibly be called on a pointer of static type T. Then $[V(T, S) + 4i]$ (the i^{th} entry of $V(T, S)$) points to S's implementation of m_i.

For each declared type in T in the program, we also generate a subtype table $STT(T)$ to evaluate casts and answer `instanceof` queries. If S is a subtype of T with Type ID i, then $[STT(T) + 4i]$ is a pointer to $V(T, S)$. If S is not a subtype of T, then $[STT(T) + 4i]$ is a null pointer.

When invoking a non-static method m_i on an object of static type T and runtime type S, we look up the object's runtime Type ID in $STT(T)$ and obtain a pointer to $V(T, S)$. We then use i to index into this vtable, obtaining a pointer to the correct implementation of the method, and invoke it.

1.4 Parameter passing and storage

When invoking a non-static method or constructor, we push a pointer to the receiver (the `this` pointer) onto the stack, followed by the arguments to the method or constructor (if any) in the order in which the corresponding parameters were declared. The method or constructor is responsible for popping all provided arguments off the stack before it returns.

We use the CS241-style calling convention where all needed registers are backed up by the caller before evaluation of a subexpression and subsequently restored by the caller. All code is otherwise free to use any registers for any purpose, except `esp` and `ebp` which are used for managing local variable storage.

1.5 Local variable storage

Local variables are stored on the stack. We used the frame concept facilitated by the x86 instructions `enter` and `leave` to organize local variables by creating a frame of appropriate size for each scope element in the program. We wrote a `Frame` class with convenience functions for creating new frames; leaving the current frame; returning from an arbitrarily-nested frame; declaring and retrieving local variables; and retrieving the `this` pointer from the stack, if it is expected to exist. This mostly localized the arithmetic involved with local variable management and made debugging this part of our compiler relatively easy.

1.6 Main entry point

After generating the assembly code for the first compilation unit's `public static int test()` method, we generate a global label `_start`. This code first calls the string literal initializer (`strlit_init`), followed by all static field initializers, and then calls the `public static int test()` method. Once this method returns, its return value is passed to the `sys_exit` system call.

1.7 String literals

We collect all string literals in the program during a prior pass through the AST. When generating `java.lang.String.s`, we generate a number of public labels on uninitialized memory blocks, which will eventually store `java.lang.String` objects corresponding to string literals. We also generate file-private labels corresponding to the `char[]`s which will back the string literals. The string literal initializer called by `_start` invokes the `java.lang.String(char[])` constructor on each string literal, passing it the corresponding `char[]` as the argument. Usages of string literals in the program are resolved by referencing these labels.

1.8 Label name conventions

The following are the rules to generate all possible labels in the x86 code. Note that the names of array types are mangled using @ instead of `[]` since brackets are not legal in `nasm` labels. Some labels, like those used to evaluate comparison expressions (where we need to do `cmp` and then a conditional jump), will never be reused, so these labels are just randomly generated with a human-readable prefix, and are not declared as global.

1.8.1 Subtype table entries

The prefix `st_`, followed by the canonical name of the table owner.

Example: `st_java.lang.String`

1.8.2 Virtual table entries

The prefix `vt_`, followed by the canonical name of the supertype, followed by `#`, followed by the canonical name of the subtype.

Example: `vt_java.lang.Object#java.lang.String`

1.8.3 Methods

Use prefix `sm_` for static methods, `im_` for non-static methods, and `call_` for non-static method dispatchers, followed by the canonical name of the implementing type, followed by `#`, followed by the method name. If the method has parameters, the canonical names of their types, preceded by `#`, follow in order.

Example 1: `sm_java.lang.System#gc`
Example 2: `sm_java.lang.System#exit#int`
Example 3: `im_java.lang.Object#toString`
Example 4: `call_java.util.PrintStream#println#java.lang.String`

1.8.4 Constructors

The prefix `ctor_`, followed by the canonical name of the type. If the constructor has parameters, the canonical names of their types, preceded by `#`, follow in order.

Example 1: `ctor_java.lang.Object`
Example 2: `ctor_java.io.PrintStream#java.io.File#java.lang.String`
Example 3: `ctor_java.lang.String#char@`

1.8.5 Static fields

The prefix `sf_`, followed by the canonical name of the containing type, followed by `#`, followed by the field name.

Example: `sf_java.lang.System#out`

1.8.6 Initializers

Use the prefix `si_` for static initializers, and `ii_` for non-static initializers, followed by the canonical name of the type.

Example 1: `si_java.lang.System`
Example 2: `ii_CodeGeneration.Frame`

1.8.7 String literals

The prefix `strlit_`, followed by the index of the string literal in `Program.allStringLiterals`.

Example: `strlit_12`

2 Implementation

We implemented code generation in our compiler by adding the method `generateCode` to the `ASTNode` class. This method generates the code corresponding to an `ASTNode`, recursing on the node's children if appropriate, and prints the generated code to an output stream. The output stream is wrapped by an object of type `AsmWriter`, which contains convenience methods related to generating `nasm` instructions and comments. `AsmWriter` also serves to make our generated code more human-readable by controlling indentation and other whitespace.

The `Program` class' implementation of `generateCode` orchestrates the creation of output streams and ensures that the generated assembly code for each type declared in the program is placed in its own, appropriately-named file. `Program` is also responsible for generating special code needed to handle `String` literals and generating the main entry point into the compiled program.

The `Compiler` class builds a vtable and subtype table for every declared type in the program, verifies the existence of a main entry point in the first compilation unit, then invokes `Program`'s implementation `generateCode` to begin the code generation process. Optionally, our new `Assembler` class can then assemble and link the generated assembly files with the standard library, and our `Runner` class can execute the generated binary and report its exit status.

3 Testing

We expanded on the test fixture we used in previous assignments. Previously, "positive" tests were expected to compile, while "negative" tests were not. In this assignment, however, all tests are expected to compile, assemble and link without issue, and the exit status of the compiled binary indicates success or failure. We wrote code to automate the assembly, linking and execution process, and to clean up the compilation artifacts between tests. We used this test fixture to run the public Marmoset tests, as well as several of our own tests.

4 Challenges

We had to revisit the layout convention for arrays due to an oversight early in the design phase. Previously, all arrays held the Type ID of their inner type in their second 32-bit value. This resulted in a problem during assignment of array elements, since we needed to access the subtype table of the runtime inner type of the array and had no mechanism to do this. The current array layout convention is one way of addressing this problem.

Most of the challenges we faced during this phase of our compiler occurred during testing. The first binary that our compiler produced would crash the machine it was run on due to a malformed loop in the string literal initialization code, making our initial testing quite slow since we had to reboot the development machine between tests. We eventually used `gdb` to find and resolve the issue. All other known problems with our generated code were subsequently resolved using `gdb`.

We attempted to make our test fixture capture and display the output of compiled test binaries as they executed, but were unable to successfully implement this feature.