CS 444
Assignment 5

Mark Rada (`marada`)
Riel Fuller (`r3fuller`)
Michael Forsyth (`mfforsyt`)

This document outlines the design of the code generation phase of the Joos 1W language implemented by our compiler to target the `i386` Linux platform as well as the `i386` OS X (darwin) platform.

Design

The design of code generation starts by generating static data needed for the Joos runtime, and then creating and running an instance of the `CodeGenerator` class for each compilation unit, as well as one instance for the `Array` pseudo-class.

The result is that a single assembly file is generated for each compilation unit, as well as one for `Array`, and a copy of our runtime library is also made. All output files are placed into `outptut/`.

Static Data

The Joos runtime requires various pieces of data that must can be statically determined at compile time and placed into the `data` section of the generated assembly code.

Type Checking

In order to support the 'instanceof' operator, down cast checking, and array element assignment, we are required to have hierarchy information available at runtime so that we can check perform a type check on the object. The required check can be paraphrased as "is the required type an ancestor of the type of the concrete data?", where the required type would be the right operand of 'instanceof', the cast type for a down cast, or the inner type of an array.

The first observation we made was that the requirement of these three runtime checks is very similar, the only exception being how `null` concrete data is handled. The first step in our solution is to assign a unique number to each compilation unit, we called this the `ancestor_number`. Then, we create a list for each concrete data type (each `Class` which is not abstract), which contains every possible type that an instance of that concrete class could be assigned. Finally, a null terminated list is output in the `data` section of the output file for the compilation unit containing the `ancestor_numbers`.

The list is accessed at runtime in the class data, which is reachable by concrete data via the concrete data's tag. To actually perform the checks, instructions for 'instanceof', down casting, and array element assignment are generated to check the `ancestor_number` by linearly scanning through the table.

In practice, class hierarchies are not very deep, and so these tables are quite small. For the marmoset test cases, the ancestor list is two or three entries. So we think this solution is reasonably fast and compact.

OK, interesting design.

Dynamic Dispatch

In Joos, static methods and constructors are monomorphic functions, and calling these functions can be done simply by knowing the label for the function and using the `call` instruction. However, instance methods are not always monomorphic, and require a runtime lookup to find the address of the method implementation.

Similar to runtime type checking, we assign a number to each concrete method signature declared for the program. However, we then also assign numbers to each abstract method signature which has a concrete implementation. This is to help with finding the correct method number during instruction generation when we may only have a reference to an abstract method implementation.

The number assigned to each method is then used as the index into a sparse table. Each instantiable type is given a table as part of the types static data. Empty entries in the table are given a null pointer. At runtime, a concrete object can reach the table via its tag.

Since Joos does not support any form of dynamic code loading, we would have liked to perform an analysis to check which instance methods were actually monomorphic, so that we could optimize those method calls and reduce the size of the vtable. However, we did not have time to implement this feature.OK

In fact, the assignment specifications and standard library do not provide any way for input to be given to a Joos program. The only output is the exit code and string output to standard output. The ultimate optimization would have been to interpret the Joos program in the compiler and generate assembly code that simply printed out any strings that the program would have printed, and exit with the appropriate code.Good observation.

Static Fields

Code generation for static fields is done in two steps. The first step is allocate space for the static value in the `data` section of the program. Each field is given a double word of space for storage for convenience. The default value for each static field is `00`, which is the correct default initialization value for Joos data no matter what type information is used to interpret the bits.

The second step is to generate a function that will be called during program initialization to initialize the field. However, static fields with no initializer will not have a function generated.

Each compilation unit has a single function which initializes its static fields in the order that they were declared in the source program. This single function is then called by the program's main function before calling the Joos `test()` method. Organizing initializers with this hierarchy was simply to keep information more compartmentalized.

String Literals

String literal objects are created for each string literal used in a compilation unit. String literals are deduplicated per compilation unit, but not globally. Our plan was to create a separate file for string literals which all compilation units would contribute to and allow full deduplication of string literals, but we did not have time to implement this feature.

Each string literal object is added to the `data` section of the compilation unit where it is used, and behaves like a normal object. It is given an object tag, which points to `java.lang.String` and contains an array reference to a `char[]` for the one field that a string declares. Directly after the string object, the array field is also declared.

Dynamic Data

The design of non-static data follows from choices made for static data.

Object Layout

The concrete data for an object begins with a tag, which points to static data for the class. The fields of an object are located after the tag and are organized in the obvious way. Fields are ordered first by the class hierarchy ordering and then by declaration order. This allows for polymorphism to work correctly and provides an efficient strategy for accessing field data by determining offsets at compilation time.

Each field is allotted a double word of storage space to make offset calculations simple, though we would have preferred to pack objects more efficiently if time had permitted.

Array Objects

Since there is no class declaration for an array, the data layout of array objects is not handled by the same code as for other objects. However, it does have a compatible layout in order to support the various things that an object must do at runtime, such as type checking.

An array contains, in order, a tag pointer for static data generated for the array pseudo-class, a tag pointer to the static data for the inner type of the array, length field of the array, and then finally the array data.

Since arrays contain an inner type, runtime type checks for arrays need to handle array checks specially to know where the tag for the inner type is located. As well, since an array's inner type can be a Joos basic type, the type checking logic must handle that specially.

For arrays of some basic type, the inner tag pointer is a pointer to unaddressable memory. On most operating systems, including Linux and OS X, the first page of memory is unaddressable, and the first page is usually 4kB in size. So, we use small values, close to 0, to represent specific basic type arrays. OK

The implementation of array element reading, writing, allocation, and type checking is written in assembly and included as part of the Joos runtime.

Assembly Skeleton Files

In order for compilation units to call each other's methods and access each others data, we needed a label generating scheme that allowed different instances of `CodeGenerator` to easily determine a label coming from another `CodeGenerator`.

Compilation units - classes and interfaces - were given a label name which was simply the units fully qualified name prefixed with `@`. Since the fully qualified name must be unique within a program, this seemed like a natural choice. The `@` prefix allows us to have a separator before the label which will not have collisions with Joos/Java identifiers.

Java allows `$`, `_`, and `.` in qualified identifier names, so we avoided using those characters as separators. However, NASM also allows `#`, `@`, , and `?` in label names, so we made thorough use of these characters.

Each field, constructor, and method label is prefixed with the compilation unit name, and then one of the separators, followed by the simple name of entity.

For example, fields are separated by `?`, so the `foo` field of `java.lang.bar` class would have the label `@java.lang.bar?foo`.

Similar to fields, methods use ? as their separator. However, as methods can be overloaded, method labels include the signature for the method. For example, a method named `foo` for class `java.lang.bar`, which takes one parameter of type `java.lang.String` would have the label `@java.lang.bar foo @java.lang.String`.

Constructor labels are generated in the same way as method labels, except that the simple name for a constructor is substituted with `@` just to make it easier for humans to tell the labels apart during debugging.

By using `@` as a prefix, we can prepend anything before a generated label for use internal to the compiler, such as `vtable_@java.lang.String`, as a shortcut for accessing the virtual function table for `java.lang.String`.

Literal string labels are simply of the form `string#UUID`, where `UUID` is a UUID generated at compile time. UUID's are extremely unlikely to collide, so we think it was a safe choice for making the label unique in a global context. Though literal strings are limited to the file in which they were declared, we did originally plan on global deduplication of string literals. OK

Program Startup

Program startup is as required by the assignment specification. All static fields are initialized, according to specification, and then a `call` to the `public static int test()` method is made.

The program will exit with the `int` returned from the `test()` method.

Conventions

Each method, and static field initializer, follows the same calling convention and organizational structure.

Calling Convention

The calling convention implemented by our compiler follows the general structure of a caller-save pattern. First, we push registers with live variables onto the stack, then we push the arguments onto the stack in the order which they appear in the source code, then we push the receiver (`this`) if the method is not static. We then lookup the method pointer as described above, and finally we call the method.

When the method returns, we pop arguments off the stack and have the result in the `eax` register.

Method Layout

Each method and static field initializer has a `.prologue` and `.epilogue` label. In the prologue, we push `ebp` onto the stack and `mov ebp, esp`, to setup the function.

In the epilogue, we reverse these instructions by first performing `mov esp, ebp` and then popping `ebp` from the stack, and finally `ret`urning.

To simplify method layout, each object and basic type is allocated a double word of storage on the stack.

Substance

Method bodies, fields initializers, and constructors are first compiled to an intermediate representation before being fully compiled to x86. We refer to this representation as Single Static Assignment form, though this is a misnomer - we do not actually transform any assignments to SSA. Did you plan to use SSA if you had time?

SSA code is organized in two levels. At the lower level, we have `Instructions`. Each instruction may consume SSA variables and produces at most one SSA variable. A SSA variable is created for each intermediate value in an expression and is immutable. We define a SSA variable by its instruction, forming a tree, but for convenience we also assign it a number which is unique to the method body. Operations may only act on SSA variables - to access, say, a local variable, we first load it into an SSA variable using a `Get` instruction.

At the higher level, we create a graph of `FlowBlocks` for representing control flow. A `FlowBlock` consists of an array of SSA instructions that form its body, as well as a `continuation` that points to the next block(s) in the graph, or returns an SSA variable.

Short-circuiting operators are also represented as `FlowBlocks`, with a special Φ instruction (called `Merge`) for "choosing" the result of a branch. We implement this in assembly by ensuring that a block followed by a `Merge` always places its result in `eax`.

The purpose of creating the intermediate representation was twofold. First, it allowed us to break away from our existing AST code in favour of a simpler form. Unfortunately, much of our code for previous compilation stages depends on the presence of concrete nodes in the syntax tree, which we never fully simplified. These concrete nodes would have significantly complicated code generation.Good.

The second purpose was anticipating register allocation. It is relatively simple to perform liveness analysis on SSA variables by traversing the flow graph backwards. Given liveness information, we could determine the necessary size of the stack frame and choose locations for values ahead of time, simplifying some of the details of code generation. However, this was not implemented due to time constraints.
Yes, register allocation gets easier if you have SSA form. It's too bad that you did not have time.

Testing

Testing for code generation was done by implementing unit tests, example Joos programs, and by using the available public tests from Marmoset.

SSA compilation testing was done with unit testing by writing small programs and testing that the output matched exactly with what was expected. Test programs were created for each type of SSA instruction.

Good

The standard library `runtime.s` was ported to OS X. As well, our entire runtime was written to support both `i386` Linux and `i386` OS X. The compiler will compile for the platform that it is executed from, and tests platform using the `uname` program. The only challenge that we had with OS X support was discovering the correct linker incantation to link with the system C library. Linking with the C library was necessary in order to call `malloc` as OS X does not support the system calls which was used by the Linux runtime library provided in the standard library. Our workaround is to use `clang`, the C compiler for OS X, which will make the appropriate calls to the linker.

The Joos runtime was written while SSA compilation was still being implemented, and so we could not test the library by compiling programs. Our strategy was to create mock Joos objects in assembly, and test the individual functions by passing the mock objects. It was simple to perform rudimentary checks for boolean values and null pointers in assembly, and exit with an appropriate exit code.

In addition to checking the result of a runtime function, we implemented a function for printing full strings to standard output, by using the example from the standard library runtime. This function allowed us to output explicit exception class names when exiting with an exception, instead of just exiting with exit code 13. However, this functionality is disabled in the submitted code just in case it would interfere with expected output.

For system level testing, we wrote example Joos programs to test edge cases in expected side effect behaviour and exception behaviour. These tests included testing that constructors initialize fields and call the super constructor in the correct order, and array bounds checking tests. We also wrote some basic programs, such as bubble sort, to exercise various other components of code generation.

The test programs were added to the collection of public tests available from Marmoset. We relied on Marmoset test for assignment 5, and some from previous assignments that were relevant to assignment 5, to fully exercise our code generator.

However, the Marmoset test harness required various changes in order to manage files that were output by the compiler. Namely, we had to change the compiler to allow the output directory to be configurable, with a default value of `output/`, so that the test suite was able parallelize compilation of test cases correctly. Interestingly, we found that tests that ran extremely slowly are hidden by all the tests which run quickly due to the parallelization.

You have made interesting and reasonable design decisions in your compiler. You have tested it thoroughly.
I wish that I could give you more time for the project. The document is very clear, organized, and detailed.
Presentation: 20/20
Content: 10/10
Total: 30/30