Asmar Khalid, ShengMin Zhang, Freddie Feng
Group 12

CS 444 - Compiler Construction

Assignment Five Design Doc

This document succeeds the assignment four design document.

Overview

This last phase of our compiler, code generation was cobbled together in about three days. As a result, some of our design decisions may seem poorly thought out, and our code incomprehensible. Both of the former statements are true. Glad you're being honest...

For each class being compiled, our compiler will output an x86 assembly file with the fully qualified name. It will also output a common library (_lib.s) file which is widely used. For lack of time, all files import all labels exported, and export all labels defined. OK

The visitor pattern is useful when we do not have to handle every node in the AST. However, since each AST node (with some very minor exceptions such as an *EmptyStatement*) are required to generate x86, we opted for a more explicit design. Each Ast node is wrapped into a scala implicit class which has a *generate* method. Every AST node implements this method with their own custom code generation logic, calling children's *generate* when necessary.

Assembly Code Generation

Instead of generating assembly codes directly as strings (which is error-prone), we built an abstract layer that we use to generate assembly code. We generate objects that implement Good. *AssemblyLine* (a representation of an x86 instruction), which then will output the contents to an OutputStream. We also leveraged Scala's operator overloading to provide familiar syntax for writing "type-safe" assembly. The following is a snippet that demonstrates this facility: Great!

```
appendText(
      push(Ecx) :# "Save this",
      mov(Ecx, Eax) :# "Inject reference to new this",
      mov(Eax, at(Eax)) :# "Move selector table into eax",
      mov(Eax, at(Eax + selectorIndex * 4)) :# "Load method declaration into Eax",
      call(Eax) :# "Call method. Returns arguments in eax",
      pop(Ecx) :# "Retrieve this",
      add(Esp, 4 * invocation.arguments.size) :# "Pop arguments off stack"
)
```

Generates the following:

```
push ecx    ; Save this
mov ecx, eax    ; Inject reference to new this
mov eax, [eax]    ; Move selector table into eax
mov eax, [eax + 4]    ; Load method declaration into Eax
call eax    ; Call method. Returns arguments in eax
pop ecx    ; Retrieve this
add esp, 0    ; Pop arguments off stack
```

Common Library

The common library (_lib.s) is an automatically generated file which provides common functionality across classes. Examples of some functions provided here are null checks, and operators. Additionally, each primitive’s array subtype table (to be explained later) is located within the data segment of the common library.

Table Metadata

Each method is given a unique index.

A selector index table is a mapping from a method index to an implementation. This is needed for dynamic dispatching.

For example, if class A overrides the object equals method (of index i), then the selector index table at index i will have a pointer to A’s implementation of equals. If there were more objects along the hierarchy that had the equals method, those indices of the table would similarly point to A’s implementation.OK. Do you use the SIT even for methods that are not declared in interfaces?

Each object type is given a unique index.

A subtype table is generated for each object. Index i of this table is 1 if and only if object is a subtype type i. It is 0 otherwise. This is obviously useful for cast and instanceof checks. Conversely, subtype tables are not generated for primitive types, and so special care is needed to handle casts regarding primitives.

Object Representation

Objects are stored on the heap. They are allocated 4 bytes per instance field, plus eight bytes. At offset zero bytes from the object, we store a pointer to the object’s selector index table. At offset four bytes from the object, we store a pointer to the object’s subtype table. The first instance field (if it exists) is located at an offset of eight bytes from the object, and so on.

Fields are ordered consistently with the object hierarchy. Since interfaces do not have fields, this ordering can remain consistent in complex hierarchies. This ordering is generated using a lazy evaluation strategy, by adding field declarations from the parent class into a *LinkedHashMap*, and then adding (or updating for overriding) entries based on the current classes fields.

Array Representation

Arrays are represented very similarly to objects. However, at offset eight we store the array length field,~~ and at offset of 12 we store the element type's subtype table.~~ Array elements are located at offset ~~16~~ 12 and above.

Arrays have a separate subtype table since their hierarchies are treated differently. A reference to the element type is required to check for invalid array stores due to array covariance. Good. Therefore, on an array store, we check the element types subtype table against the right hand side's runtime type.

Array Covariance

Since arrays in Joos are covariant, clever subverting of the type checker can introduce nonsensical behaviour. Because of this, we are required to check that the right hand of an assignment expression is a subtype of the array element runtime type. Unfortunately, due to time constraints, we left this feature unimplemented. OK

Quirks

The minimum integer value -2147483648 is tokenized as a negation of the positive integer 2147483648. Since 2147483648 does not fit in 32-bit two's complement, we convert the token to the minimum integer value -2147483647. The following negation (which inevitably exists) will again yield -2147483647, as desired.

I think you mean -2147483648, don't you? (This is invariant under negation.)

String concatenation using the + operator maps both arguments to *String.valueOf* to convert them into strings (if one is not already). Then, it will call the *concat(String)* instance method on the left string with the right string as its argument.

Testing

We used a very test-driven approach to tackling this assignment. We started generating code for very simple ASTs (such as just one return statement) and writing tests to validate the output. Additionally, we wrote a *very sophisticated* bash script to run these test cases through all stages of our compiler, assembler, linker and finally execute them to validate output. This helped us to monitor regressions when developing. Great.

Additionally, we also imported the marmoset tests and ran the same script on them. Although, we did not even run these until we had already sufficient tests. We mainly used the marmoset tests to patch edge cases in our code or bad assumptions we were making about the Joos language.

Appendix

Data Representation

- Uninitialized values, that is, numeric primitives, false, and null assume the value zero.
- Values are stored as 32-bits.
- true is the value 1
- Primitives are not autoboxed into references

Program Invariants

- Expression nodes will return the result of their evaluation in *eax*. The address of the resulting value (if the expression was an lvalue) will be placed in *edx*.
- *ecx* contains a reference to *this* object when the context permits
- Before an instance method invocation, the current value of *ecx* (that is the current *this*) will be placed on the stack, after all parameters. *ecx* will then be set to the method owner instance so that *this* references are valid. *ecx* will be restored after the function returns.
- Parameters are passed left-to-right and are below the frame pointer
- Local variables are located above the frame pointer
- *ebx, esi,* and *edi* are callee save registers. (In fact, *esi* and *edi* are unused)

As a group, we collectively enjoyed our time spent learning about compiler theory, learning scala, implementing the various stages of the compiler, and finally seeing real computable results. I guess we also learned that language designers love to include really silly language features. 10/10 would take the course again, after a long hiatus. Great, glad you liked it!

Under the time pressure, you made good design decisions leading to a solid compiler. The document is organized, clear, and detailed.
Presentation: 20/20
Content: 10/10
Total: 30/30