CS179E Phase 2: Intermediate Code Generation

by Brandon Yi

Requirements and Specifications

This phase of the project implements an intermediate code generator. The program takes a correctly typechecked MiniJava program (Typechecking is done in Phase 1) and prints a program in the Vapor language to standard out. The resulting program in the Vapor language behaves the same as the input MiniJava file. Furthermore, the Vapor program checks if an array access is out of bounds.

Design

Overview

There are two main data structures involved in the process of creating Vapor intermediate code: the Symbol table from Phase 1 and a new structure called a V_Table. For an explanation of how the Symbol table is constructed, refer to the Phase 1 documentation.

After the Symbol table and the V_Tables have been constructed, a visitor is used to traverse a syntax tree that was built in Phase 1. The visitor has enough information to do its job just by looking at the data stored in the Symbol table and the V_Table.

V_Table

A V_Table or virtual table contains all of the functions which are members of a certain class. Each class in the program gets its own V_Table. Every time a class instance is created, the V_Table is copied into the new class instance. The V_Table is an essential part of enabling MiniJava to perform dynamic dispatching. If a class extends another class, the class which is doing the extending gets a copy of the entries in the parent's V_Table.

The V_Table is simply a `const` structure that is provided by the Vapor language. Here is an example of a V_Table:

```
const A_vtable
  :A_test
  :A_these
  :A_are
  :A_function
  :A_labels
```

This V_Table contains a label to the functions which are accessible to the A class.

As mentioned above, when a class instance is created, the associated V_Talbe is copied into the class instance. This is how the V_Table is copied in the Vapor language:

```
const A_vtable
  :A_test

const A
  :A_vtable

...

func SomeFunc(this)
  t.0 = HeapAllocZ(4)
  [t.0] = :A
```

Additionally, a function, say `A_test`, can be called in this manner:

```
func SomeFunc(this)
  t.0 = HeapAllocZ(4)
  [t.0] = :A
  t.1 = [t.0]
  t.1 = [t.1]
  t.1 = [t.1 + 0]
  t.2 = call t.1(t.0) # Result of A_test is stored in a temp var
```

The line `t.1 = [t.1 + 0]` is storing the address of the 0th element in `A_vtable` into `t.0`. This visitor is able to calculate the offset of each function label because this information was precomputed before running the visitor. This is done using a minor data structure called a class record.

Class Record

A class record is a helper data structure which provides the visitor with critical information like the offsets of function labels, the offsets of fields and the size of the class structure. In particular, the size of the class structure is calculated by multiplying the number of fields in the class by 4 and then adding 4 to the whole quantity.

Testing and Verification

Along with the basic tests provided by this course, I added a few more special cases (All MiniJava test files can be found in `./src/test/resources/part2_input`). These cases include:

Test file	Description
ArrayLength.java	This test checks if the `<arr>.length` statement returns the length of the array. It does.
BoolFunc.java	This checks various boolean statements to see if the correct results are returned.
Call.java	A more simple function call test.
ParamAccess.java	This test checks if the memory of class objects are correctly isolsated from other instances of classes.
Var.java	A more simple test for checking the accessibility of local variables in functions.

While this project does use `gradle`, you must supply the test files to the program manually and use a Vapor interpreter to verify the results.