6.S081 Spring 2018

Assignment 3: Virtual Machine & Assignment 4: Garbage Collection

Anelise Newman, Stef Ren, Kimberli Zhong

Virtual Machine

The virtual machine design has two main components: a *compiler* and an *interpreter*. The compiler is responsible for performing static analysis and translating MITScript source code into bytecode, while the interpreter executes the bytecode.

Design Decisions

Compiler

First, our MITScript parser from Assignment 2 parses the MITScript text and creates an abstract syntax tree (AST) in memory.

The next step is to create a symbol table representation that can be used to properly scope variables during bytecode generation. The output of the `SymbolTableBuilder` is a vector of `SymbolTables`, added in the order of their traversal by the visitor. `SymbolTables` map variable names to variable description objects or `VarDescs`, which contain information such as if the variable is global, local, or free, and if it is referenced in any child frames. Each `SymbolTable` represents a specific frame and contains a entry for each variable used in that frame, as well as a pointer to the parent frame . This representation is somewhat redundant because it contains multiple entries for the same variable when that variable is used in multiple scopes. However, it is an effective representation because it allows us to easily describe situations where the same variable has different roles in different functions (for instance, it may be free in one function and local in another). It also provides for a very quick and easy lookup when we are generating bytecode and need to find a complete descriptor for a variable in the context of a particular frame. All of the static analysis used to build the symbol tables is conducted ahead of time before bytecode is generated so that the two processes are easier to separate and reason about.

The symbol table is built in two stages. First, a `SymbolTableBuilder`, implemented as a visitor, traverses the code. It keeps track of variables that are marked as global or that are assigned within a function (these correspond to global and local variables, respectively), as well as any variable that is accessed. It creates a partially-filled-in symbol table for each function containing these global and local variables. In the second stage, for each frame, the `VarDesc` for every variable that was accessed is resolved via pointer-chasing, and local variables that are referred to in child frames are marked as ref vars. .

Then, a main `BytecodeCompiler` class implements the visitor pattern to generate the bytecode. When the compiler visits a function declaration, it uses the corresponding symbol table to preload the `Function` representation's local vars, local ref vars, free vars, and names arrays. Each node visited by the Visitor generates bytecode by adding to a growing list of instructions. We decided not to use a CFG representation because we found that it added a lot of complexity to our code and made it harder to debug. In particular, building up a CFG via the visitor interface turned out to be complicated.

Interpreter

The Interpreter's main state is a set of stack frames, where each frame is represented by a `Frame` object. Each frame contains its own opstack represented as a stack of `Values`. `Values` in the program can be `Constants`, `Records`, `Functions`, `Closures`, or `ValuePtrs`. The `Frame` also maintains a mapping from variable names to `ValuePtr` objects.

One of the more challenging parts of implementing the interpreter was figuring out how to properly represent variables so that when a variable was reassigned, the change could be reflected in the interpreter without chasing down every reference to that variable. We accomplished this using a `ValuePtr` object, which is basically just a wrapper that points to another `Value` that can be manipulated as a value itself. Each variable corresponds to one `ValuePtr`. When we change the value of a variable, this corresponds to changing the pointer contained in its `ValuePtr`. We can then pass a reference to this `ValuePtr` anywhere this variable is required, such as a frame's mapping from variable names to `ValuePtrs`. For instance, executing `load_ref` or `store_ref` corresponds to looking up the reference variable's name and following its `ValuePtr` to resolve its value. We also use `ValuePtrs` to implement reference passing between frames, since we can pass the `ValuePtr` object to the next frame's variable map. If any frame changes the value at the indicated pointer, it will be changed in all frames.

The interpreter runs in a loop, executing bytecode instructions and manipulating its frame and operation stack according to the instruction specifications. It takes in parsed bytecode instructions from either the compiler or the bytecode parser. Each instruction is passed to a switch block that dispatches it to the code for the corresponding instruction. As specified in the bytecode spec, the interpreter will jump to other dependent functions or to other points in the instruction list according to `goto` and `if` instructions.

Testing

A variety of staff-provided and manually written tests are located in the `tests/vm` folder. We wrote three testing scripts:

- `test_compiler.sh` tests the compiler's ability to produce correct bytecode output
- `test_vm.sh` tests the interpreter's ability to parse bytecode files and generate correct program output

- `test_interpreter.sh` checks the entire pipeline, starting from MITScript parsing all the way out to program execution

Most of our custom `test_compiler.sh` tests are taken from the custom tests written for Assignment 2, since they test edge cases with interpreter behavior. We also wrote a suite of unit tests for the bytecode instructions, making sure we cover every possible instruction. Our test scripts optionally take in a string regex argument used to run any test whose filename matches the regex.

Challenges/Difficulties

Our project is currently not failing any of our test cases. We ran into a few ambiguities for which we made assumptions, however.

- Printing a record is under-specified because the order in which to print fields is not set, so our results may differ from the staff results.
- The staff-provided bytecode lexer's string-matching regex did not support escaped backslashes or tab characters, so we had to update that to pass our special character test case.
- The sample bytecode output that came with the assignment's provided tests

Contributions

Compiler: Anelise & Kim
Interpreter: Steph & Kim
Debugging: Collective