6.S081 Spring 2018

Assignment 3: Virtual Machine & Assignment 4: Garbage Collection

Anelise Newman, Stef Ren, Kimberli Zhong

Assignment 3: 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.

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 had slightly different instructions, but we ascertained that our compiler preserved the same functionality.
- A few of the staff-supplied bytecode tests had reserved bytecode keywords as identifiers, such as `add` and `eq`. We changed these tests appropriately to generate parseable bytecode.
- Currently, our compiler allocates a new `Constant` for each of its appearances in the source code. For example, the statement `a = 1 + 1;` will result in one `Integer` allocated for each integer in the expression. Although this is not optimal in terms of memory usage, we decided to make this tradeoff for implementation simplicity for now.

Contributions

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

Assignment 4: Garbage Collection

We implemented a mark-and-sweep garbage collector using two main data types: `Collectable`, a base class for all values we want to be able to delete, and `CollectedHeap`, which keeps track of allocated `Collectables` and performs garbage collection.

Currently, our allocator only tracks the size of objects allocated during the interpretation phase. Originally, we toyed with also using the allocator to create the `Functions` and `Constants` generated in the compilation phase, but we decided to cut it out because those generated values can b

Data Types

All of our `Value` objects as well as our `Frames` derive from a base `Collectable` class. The `Collectables` metadata contains a boolean flag used to keep track of their reachability. `Collectables` also implement two functions. The `follow()` function calls `markSuccessors` on all `Collectable` objects to which the current `Collectable` has a pointer. The `getSize()` function returns an intelligent estimate of the total memory allocated for that object, taking into account heap-allocated data structures like maps and vectors.

`CollectedHeap` aggregates the items in the heap by keeping a linked list of all allocated `Collectables`. It provides `allocate` constructors which a) allocate a new `Collectable`, b) store a pointer to it in the linked list, c) update its tally of total memory usage, and d) return a pointer to the new object. It also keeps track of the total number of allocated items and the approximate memory used. `CollectedHeap` also has a method to increment the `currentSizeBytes` value used to track the total memory usage, useful for when data structures such as `maps` and `vectors` add members.

Each object “owns” the memory used for its own metadata and internal data structures, but does not count space allocated for children `Collectables` it points to. For instance, the size of the `Value` pointed to by a `ValuePtr` is not included in its total, and a `Record` is responsible for the memory it requires to store its own keys and pointers, but is not responsible for the memory allocated for the values themselves.

We track the sizes of our different objects using the `getSize()` instance method defined for `Collectables`. For simple datatypes like `None`, `Boolean`, `Integer`, and `ValuePtr`, `getSize()` returns a constant value corresponding to the size of its struct. For datatypes containing members that are heap-allocated, like `Strings`, `Records`, `Functions`, and `Closures`, the `getSize()` function also iterates through those objects, counting the size of their elements. For example, `String`'s `getSize()` method includes the length of the `string` stored as its member.

Garbage collection process

The garbage collection process consists of a mark phase and then a sweep phase.

Marking

`Collectables` are marked by setting an instance boolean field in the `markSuccessors()` function. The garbage collector is given a `rootset` which consists of the current stack of frames. We recursively mark all allocated objects that can be found from this `rootset`. Each frame marks its function. `Functions` mark their constants, functions, local variables, and local reference variables. `Records` also call mark on their elements.

If the garbage collector reaches an item that is already marked, it does not recurse on that item, since we should have already recursed on it.

Sweeping

The garbage collector iterates through the linked list of allocated objects. If any objects are not marked, it deletes them. Otherwise, it will mark the object as unmarked so that its state is reset for the next mark and sweep operation.

When a `Collectable` object is being deleted, we use the `delete` C++ operator to destruct it and its members. For things like the `std::maps` that map `string` variable names to `ValuePtrs*`, destructing the containing struct will destruct the `string` keys as well as the `ValuePtrs*`, but not the `ValuePtrs` themselves. The `ValuePtrs` of the defunct `Frame` are deallocated separately in the mark and sweep process, since they are also contained in the linked list of allocated objects. However, their target `Constants` may not be deleted—this is intended behavior, as it's possible that two frames store `ValuePtrs` to the same underlying `Constant`, and deallocating one frame should only deallocate the `ValuePtr` wrapper.

Triggering garbage collection

After every instruction, the CollectedHeap will check whether or not the `currentSizeBytes` exceeds `maxSizeBytes/2`. If so, garbage collection will be initiated.

Even though our theoretical threshold for triggering garbage collection is half of `maxSizeBytes`, we include this constant factor of 2 to account for overhead in the compilation process and allocation of `std` data structures (when we create `maps`, `vectors`, etc. with the `new` keyword, our allocator tallies the size of their contents instead of the data structure capacity).

Testing

We wrote stress tests that checked variable, record, and function frame deallocation and used valgrind's Massif heap profiling tool to verify memory usage. Some stress tests isolate functionality by e.g. repeatedly garbage collecting only `Records` or `Frames`. The memory usage graphs for some of our custom stress tests are included below. As can be seen in the graphs, we consistently stay below 4 MB usage.

Test `good01-stressWhile`

```
--------------------------------------------------------------------------------
Command:            ../../vm/mitscript -mem 4 -s good01-stressWhile.mit
Massif arguments:   (none)
--------------------------------------------------------------------------------


    MB
3.557^                                                           ::
     | #       ::       :@@             :           :      :     :        :
     | #       :        :@             ::           :      :     :        :
     | #       :        :@         @   ::   :       :    :::     :      @ :
     | #       :      :::@ :       @   ::   :       :  ::: :  @  :      @ :  :
     | #       : :    : :@ :       @   ::   :       :  : : :  @  : :  : @ :  :
     | #  :    : :    : :@ :       @   ::   :       :: : : :::@  : :  : @ :  :
     | #:::    : :    : :@ :    :  @   ::   :       :: : : :: @  : :: : @ :: :
     | #: :    : : :: : :@ :    :  @   ::   :       :: : : :: @::: :: : @ :: :
     | #: :    : : :: : :@ :    :: @:  ::   :       :: : : :: @: : :::: @::: :
     | #: :    : : :: : :@ :    :: @:  ::   ::      :: : : :: @: : :::: @::: :
     | #: ::   : : :: : :@ :    :: @:  ::@@ ::      :: : : :: @: : :::::@::: :
     | #: ::   : : :: : :@ :    :: @:  ::@  ::      :: : : :: @: : :::::@:::::
     | #: ::   : : :: : :@ :    :: @:  ::@  ::::    :: : : :: @: : :::::@:::::
     | #: ::   : : :: : :@ :    :: @:  ::@  :::    ::: : : :: @: : :::::@:::::
     | #: ::   : :::: : :@ :    :: @:  ::@  :::    ::: : : :: @: : :::::@:::::
     | #: :::::: :::: : :@ ::   :: @:  ::@  :::  ::::: : : :: @: : :::::@:::::
     | #: ::: :: :::: : :@ :::: :: @:  ::@ :::: :: ::::: : :: @: : :::::@:::::
     | #: ::: :: :::: : :@ :::  :: @:  ::@ :::: :: ::::: : :: @: : :::::@:::::
     | #: ::: :: :::::: :@ ::: ::: @:::::@ :::: :: ::::: : :: @: : :::::@:::::
   0 +----------------------------------------------------------------------->Gi
     0                                                                   10.30
```

Test `good02-stressRecursive`

```
--------------------------------------------------------------------------------
Command:            ../../vm/mitscript -mem 4 -s good02-stressRecursive.mit
Massif arguments:   (none)
--------------------------------------------------------------------------------


    MB
3.594^      #
     |      #                      ::                             :
     |      #                      :       :      @               :
     |      #                      :       :      @       :       :      ::
     |     @#      ::      :       :       :      @       :      ::      :
     |     @#      :       :       :       :      @       :      ::      :
     |     @#      :       :      ::      ::      @       :      ::     @:
     |     @#      :       :      ::      ::     :@       ::    :::     @:
     |    :@#    :::       :      ::      ::     :@       ::    :::     @:
     |    :@#    : :     :::      ::      ::     :@::   @@::    :::    :@:
     |    :@#    : :     : :      ::    ::::     :@:    @ ::    :::    :@:
     |   ::@#    : :     : ::   ::::    : ::     :@:    @ ::    :::    :@:
     |   ::@#    : :    :: ::   : ::    : ::   @@:@:   :@ ::    :::   ::@:
     |   ::@#    : :    :: ::   : ::   :: ::   @ :@:   :@ ::    :::   ::@:   :
     |  @::@#  ::: : :  :: ::  :: ::   :: ::   @ :@:   :@ ::  :::::   ::@:   :
     |  @::@#  : : : :  :: ::  :: ::   :: ::   @ :@:   :@ ::  : :::  :::@: : :
     |  @::@#  : : : :  :: ::  :: ::   :: ::  :@ :@:   :@ ::  : :::  :::@: : :
     |  @::@#  : : : :@@:: ::  :: :: @@:: ::  :@ :@:   :@ ::  : ::::::::@: :::
     |  @::@#  : : : :@ :: :::::: :: @ :: ::  :@ :@: :::@ ::::: :::: :::@: :::
     |  @::@#::: : : :@ :: ::: :: :: @ :: :::::@ :@: : :@ ::: : :::: :::@: :::
   0 +----------------------------------------------------------------------->Gi
     0                                                                   2.608
```

We also used valgrind's leak checking tool on small unit test cases to find memory leaks that were not properly deallocated by the garbage collector, since initially the Massif output files showed that we had large amounts of memory being leaked, as described below.

Challenges & Difficulties

We struggled for a long time with memory leaking consistently throughout the garbage collection process, resulting in total memory usage for our stress tests at the level of hundreds of MBs, way more than the total size of all possible allocated program data.

We squashed a number of small bugs: converting the escaped string in the `String::toString()` method created a dangling pointer, as did initializing a `map` in `Record`. It turns out that the largest memory leak resulted from an oversight with our `delete` operators inside the sweeping process: if its operand is a pointer to an object, `delete` normally calls the object's destructor only if it's defined virtually. In our code, the destructor for `Collectable` was not defined virtually, even though destructors for `Value` and `Frame` were. Since `Value` and `Frame` inherit from `Collectable`, our virtual destructors for those classes were deleted, and calling `delete` on any of them with this bug did not result in memory actually being freed by the program.

Contributions

Allocator: Anelise & Stef
Interpreter: Stef & Kim
Debugging: Collective