6.035 Final Project

Yasyf Mohamedali, Jack Serrino
{yasyf, jserrino}@mit.edu
Wednesday, May 17th, 2017

Overview

Over the past semester, we've put considerable effort into designing and implementing an interpreter for the MITScript language. This document will outline the design of our interpreter, as well as go into detail about the optimizations we have employed to make given source files execute as quickly as possible.

Interpreter Design

Components

Our interpreter has six main components:

1. MITScript-to-bytecode compiler (referred to as BCC or Bytecode Compiler), designed to take in MITScript source files, and compile them to a bytecode representation.
2. The bytecode virtual machine, designed to execute compiled bytecode functions directly and produce a program's output.
3. The heap manager, used to allocate any objects and collect any garbage a program creates.
4. The bytecode-to-IR compiler, designed to take in compiled bytecode functions and produce an internal representation useful for optimization and producing machine code.
5. The IR optimizer, designed to optimize and remove any redundancy or inefficiency in the IR representation
6. The IR-to-machine-code compiler, used to produce machine executable code, directly executed on the host machine.

When running in unoptimized mode, the interpreter will only use components 1, 2, and 3. A source file will be compiled to bytecode by 1, and then will be executed by 2 to produce the desired result, while having its heap managed by 3. Note this will happen all at once, without further serialization/deserialization of the code.

In fully optimized mode, all six components are used. First, a source file will be compiled to bytecode by 1, and then each bytecode function will be individually compiler to the IR representation by 4. This IR will be optimized by 5 and passed to 6, which will generate executable machine code. To run the compiled assembly, a helper implemented in 2 will setup the correct call frame and jump directly into the machine code, beginning execution. When things need to be allocated, 3 will be periodically called for allocations and garbage collection.

Currently, we have made the decision that every function is compiled to machine code just-in-time. However, the VM is designed such that we could use an arbitrary predicate for making this choice.

Bytecode Changes

We've made a few of small changes to the bytecode since Lab 3.

Firstly, we've changed **Goto** and **If** behavior to jump to labels instead of jumping to relative offsets. This included adding **Label** instructions for the purpose of identifying where instruction pointers should jump to without knowing IR and ASM instruction deltas ahead of time. This reduced the accounting needed in later compilation steps and allowed us to represent multiple IR instructions with one BC line.

Secondly, we've added a **GarbageCollect** instruction that is inserted at the beginning of every while loop and after every function call statement. This made it easy for us to reason about the behavior of garbage collection, since in these scenarios we can ensure that no intermediate values will be stored on the operand stack or in registers.

Finally, we added a **ThrowUninitialized** instruction that allows uninitialized variable exceptions detected by the BCC to be passed through and lazily raised when the appropriate line is eventually executed.

IR Design

Our IR has the following regular instructions:

Assign, Store, Add, Sub, Mul, Div, Gt, Geq, Eq, And, Or, Not, Neg, Call, AllocClosure, Return, OutputLabel, Jump, CondJump, CallHelper, CallAssert, Fork

The following instructions are generated in optimization passes:

ForceLoad, IntAdd, FastEq, ShortJump, Noop

Each instruction maps almost directly from the bytecode instructions, often augmenting or performing extra analysis. Each also maps almost directly onto a set of machine code instructions.

Generating IR

To generate a list of instructions for the IR, we perform a linear scan through the instructions of a bytecode function, transforming instructions one by one as necessary. This is possible to do in one scan since we can emulate the stack machine of the program as we go. Since the state of the stack is guaranteed to be the same before and after an if statement, and before and after a while loop, we don't have to worry about following branches.

The IR operand stack operates on temps (temporary, one-use variables), which can be generated from various operations, or loaded from a BC operand: Var, Glob, Const, Ref, Deref, Function, or RetVal. Note that a Deref is the result of dereferencing a reference, whereas a Ref simply sets a temp to the reference itself. Assigning a temp from RetVal causes the temp to assume the return value of the last function or helper call.

The IR generation phase also inserts naive assertions before every operation that requires it. Many of these are later optimized out.

Generating Machine Code

To generate machine code, we go through the list of IR instructions, and output the set of machine instructions that correspond for each IR instruction. Certain instructions, like read_global or alloc_record, call out to helpers implemented in C++. Most operations, however, are implemented directly in machine code, only reaching into the C++ layer to throw an exception or call a new function.. We also perform some bookkeeping to see which machine code registers are being used as scratch space, as well as output necessary x64 machine code to conform to the Linux calling convention.

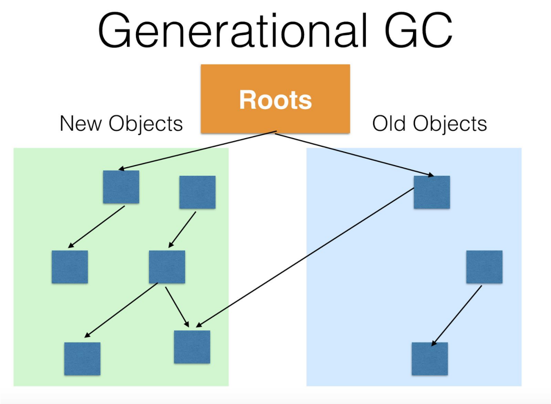
General Optimizations

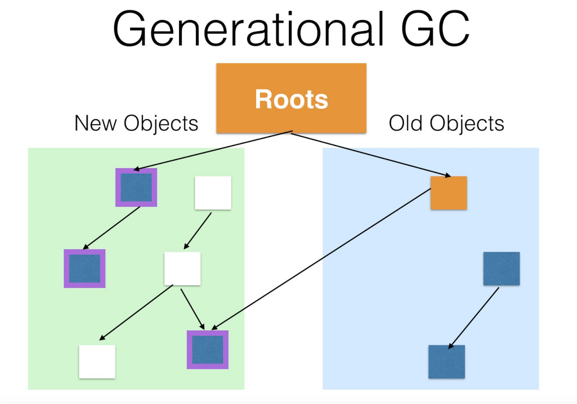
We've implemented a couple of optimizations upon the six main components of our system to improve the runtime speed. This section will detail Non-IR optimizations.

Generational Garbage Collection

One problem with a naive mark and sweep algorithm is that both phases take a long time, since you have to iterate through every live object in the entire set. A generational garbage collector solves this problem by taking advantage of the fact that most objects that are allocated are very short lived, which means they are allocated and freed all in one cycle of the collector. If this is the case, we can save time in our collector by doing a "good enough" job just collecting all of these low hanging fruits, only stopping to collect everything once enough objects pile-up.

We implemented a 2-generation garbage collection algorithm, and saw decent speedups in our program execution time. Our collector works as follows. Whenever an object is allocated, it's added to the "New objects" heap. When objects in the the "old objects" heap, change to point to an object in the new object's heap, we add that object to a set of temporary cross-generation roots. To perform a collection, we just perform mark and sweep on the new objects starting from the initial set of roots and the set of temporary roots, collecting objects from the new generation heap that aren't marked. Finally, we "move" objects from the new heap to the old heap. By doing it with generations, we save us a lot of time since we only have to mark and sweep over the new objects.





Hybrid Linear Scan Register Allocator

To improve the runtime of our assembly code, we worked to keep objects in registers as often as possible. To accomplish this, we implemented a two-stage register allocator which first uses the Linear Scan technique (https://pdos.csail.mit.edu/papers/toplas-linearscan.ps) to effeciently distribute the bulk of the registers (r12 - r15, r8 - r11), followed by a greedy allocator to distribute the remaining registers as required operands and scratch space.

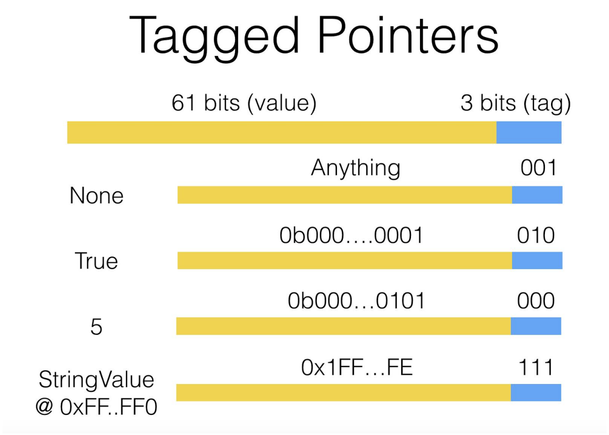
Each Temp and Var is tagged with the register (if any) it has been allocated by the first phase, so that the second phase can spill these operands to main memory as infrequently as possible. As of right now, for simplicity, we still allocate enough space for each such operand to be spilled to a static location on the function's stack.

We made the decision to only have the Linear Scan allocator tag operands with registers that are never required arguments to x64 operations, to avoid conflicts and the complexity of shuffling registers around at runtime. While we explored graph-colouring-based approaches to optimal register allocation, we settled on the solution that gave us the best tradeoff between optimality, memory efficiency, and performance. Given that we are building a JIT compiler, a high fixed cost at compilation time is undesirable and we sought to avoid extraneous overhead at all costs.

Tagged Pointers

A huge performance optimization that we made early on was the addition of Tagged Pointers. We noticed that all of the pointers to objects in our Value class where 8-byte aligned, and that meant that the last three bits of any pointer was free to store extra data. The end result of our tagged pointers implementation meant that all integers, booleans, and nones never had to be directly allocated, since they would always be resting inside our value class either on the stack or inside a ReferenceValue. This gave an immense speedup since we no longer had to allocate tiny amounts of memory to do a simple addition, or call a helper to do a type check.

Our tagged pointers representation is shown below:



Value Type and Tag					
Integer	None	Boolean	StringConstant	StringValue	ReferenceValue/ClosureValue/ RecordValue/etc.
000	001	010	011	111	100

We made the conscious decision to set integer value's tag to be 000, since it would make addition/subtraction/multiplication/division 1 instruction (sometimes 2) instead of 2 instructions (sometimes 4). Since arithmetic is a incredibly core component to every program, it's important that these instructions that are likely to be part of the inner loop are fast.

We also made the conscious design decision to make StringConstant and StringValue's tags very similar, that way we could just check if the last two bits are 1 to see if it's any type of string. Similarly, the 3rd least significant bit being 1 indicates that it's a heap-managed pointer.

Other/Minor Optimizations

We also saw speedups by doing the following optimizations:

1. Reducing the binary size
    a. How?
        i. By moving things away from .h files into .cpp files

ii. By using less templated functions
iii. By only linking the parts of each library we needed

b. Effect?
   i. With a smaller binary, our code will have a small maximum resident set size, which allowed us to garbage collect less often, speeding the program

2. Reducing the size of ClosureValues, RecordValues, ReferenceValues, etc.
   a. How?
      i. Removing dependencies such as “CollectedHeap”
      ii. By removing C++ standard library containers if possible
   b. Effect?
      i. With our value classes using less memory, we garbage collect less often, speeding up the program.
3. Eliminating string lookups
   a. How?
      i. The original bytecode had programs look up strings in a names array, which would then be used as an index into a map to find the values of references and locals. We eliminated this, and used the indexes directly from the bytecode as the index into an array
   b. Effect?
      i. This speeds up program execution since programs won’t need to hash a string. Also, this reduces program memory usage, since we don’t have an extra hash table per stack frame.
4. Pre-computing argument-to-local mapping
   a. How?
      i. Some arguments that are passed in maybe be local reference vars, while some may be normal locals. The values passed in as arguments to a function need to be set properly, so we need a mapping from argument name to index in the correct array. Instead of computing this mapping each time we call a function, we compute the mapping once at compile time, and reuse it multiple times
   b. Effect?
      i. Program execution is much faster since we don’t have to construct a mapping as often.
5. Store all variables on the stack
   a. How?
      i. We used to store variables in an std::vector because of the nice auto-expanding properties, but then we realized we know the number of locals ahead of time. So, we put all locals and pointers to local references in a big array on the stack.
   b. Effect?

i. This saves execution time since it's very inexpensive to create stack space, and it's memory efficient since we aren't allocating inside an std::vector

Ineffective Optimizations

Not all of the optimizations we tried were effective.

One optimization that had promise but ended up being detrimental was the concept of a "String Tree". The motivation was that we wanted string concatenation to be cheap O(1) instead of O(n), since concatenation is the only operation you can perform on strings. Furthermore, you only need to read through the entire string on == or on printing. The idea was you'd allocate a new type of StringValue which just had pointers to the left and right objects that were part of the string, and it would lazily calculate the string when needed. Where this fell short was in garbage collection -- instead of having to mark one object for strings, we'd have to mark everything down the tree. This gave pretty large slowdowns in most cases, so we disabled the "optimization".

Another optimization we made was storing pointers to strings inside maps instead of the actual std::string. If each map inside a record value stores a pointer instead of an std::string then we could save memory by not duplicating keys in each map. This could especially be powerful since most maps are clones of each other in a typical program - each map has different values but the same keys. In practice, this gave a very small memory benefit, and slowed down execution greatly, so we disabled this optimization as well.

IR and Machine Code Optimizations

We have the luxury of working with an IR representation in which every operand is a Temp which is used once and discarded. We exploit this to be able to do analysis for optimizations on the linear representation of the IR instructions, instead of building a CFG. We traverse the lattice formed by the flow of data into and out of temps, iteratively transferring properties from source to destination until convergence or a static halting point. Many of the optimizations below rely on this process.

Constant Folding

We propagate the property of being derived from a constant value or expression through our lattice. This starts with constant literals and is transferred through any arithmetic or logical operation, as well as loads and stores involving local Vars. At every pass, we statically evaluate any expression whose operands are all tagged as constant, replacing the instruction with the result of the expression.

This optimization resulted in many fewer IR instructions for constant literal-heavy programs.

Type Propagation

We propagate hints about the type of an operand, limited in scope to Ints, Bools, and Strings. This allows us to make several optimizations down the line, including removing unnecessary runtime type assertions. We use a bitvector to store these type hints, ORing new hints in each time.

Type Specialization

Once we are certain of the type of a Temp (it only has one option that has been hinted), we can attempt to specialize certain operations (such as Add for Ints or Eq for Ints/Bools) so that they can be executed in assembly without calling a helper.

Copy-On-Write

We included an optimization which retained values which were written to a variable and then subsequently read back out. Instead of potentially reading and writing from main memory, the values are kept in temps which are ideally register-allocated. This optimization is currently disabled as it conflicts with our register allocation scheme.

Temp and Var Liveness

In order for subsequent optimizations, including register allocation to occur, we must have an accurate sense of the "live range" for each Temp and Var. Thus each optimization pass includes one pass through the code, examining where each Temp and Var is used and setting the start and end of the live range for each.

Dead Temps

After passes through all the other optimizations, we replace assignments from Temps to Temps by simply resolving the aliases. We then find any unused temps and add them to an obsolete list, which is later processed by deleting all checks and references to said Temp.

Dead Variable Assignment

Local Vars which are written to but never read from are simply removed from the IR. This is one example of several instances where we remove an operation that has no observable effect on the final output of a program.

Noop Generation and Removal

Resizing vectors of instructions is an O(n) operation in C++. Thus, to avoid this overhead, optimizations simply replace instructions they would like to delete with a Noop singleton. We then occasionally scan the instruction list between passes, copying all existing instructions to a new list but excluding any Noops.

Short Jumps

In order to maintain conservative correctness, our IR compiler by default outputs Jump instructions that get translated to x64 near jumps. However, in specific cases, we can make do with the more efficient short jump instead. Thus one of our optimization passes goes through and makes this replacement as conservatively as possible. This optimization provided a negligible speedup.

Implementation Issues

While our final interpreter can run any valid MITScript program, we encountered many implementation challenges along the way.

Perhaps the chief problem from the memory side of optimizations was determining the best time to garbage collect. This is because 1) estimating how much memory a program is using for everything is difficult on linux, and 2) garbage collection patterns on very similar programs can often be very different. As an example, there was one point in our interpreter where the following first program used LESS memory than the second one.

First:

```
    i = 0;

    iterations = intcast(input());

    while (i < iterations) {
        x = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" + "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        i = i + 1;
    }
```

Second:

```
    i = 0;

    iterations = intcast(input());

    while (i < iterations) {
        x = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" + "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        i = i + 1;
    }
```

This sort of behavior is extremely counterintuitive and hard to track down, and developing accurate estimates for the overhead of the program took significant time and testing.

Another problem we ran into was an issue with keeping everything that is referenceable in the root set. This includes local reference vars and local variables, as well as the operand stack. While executing the bytecode vm, we can add the operand stack directly to our root set, but when executing in assembly mode, there is no stack - these are stored directly in registers most of the time. To combat this, we developed a garbage collection instruction that we inserted at key points in our program (where we ensured the stack was empty or no temps contained collectable things). Before a garbage collect, we flush the necessary variables to memory so that they aren't stale when the GC performs the mark and sweep operation.

The last, and perhaps hardest, issue we faced when writing this compiler was debugging incorrect or slow assembly code. Because this code is generated at runtime, traditional debugging tools, like GDB, can only be so effective. We relied heavily on the step feature in GDB to step through certain functions, as well as our own debugging tools like our assembly pretty printer to diagnose and correct problems, but the process was extremely slow.

Tooling

In the process of creating our interpreter, we developed a lot of tooling to increase our own development speed.

The most useful set of tooling we created was a Makefile and folder structure that allowed us to compile each portion of our interpreter independently from the other portions. This made it easy to visually inspect bytecode/IR/ASM, program correctness checkers, etc. The folder layout of our project is designed for modularity.

We also built in a series of non-optimizing flags into our interpreter so we could analyze its behavior at runtime. The most notable flags are --memory-usage and --memory trace, which show memory usage statistics, and a trace of each allocation and deallocation, respectively. We also built a corresponding allocation analyzer in python, helpful in assessing the performance our collector.

Another set of tools we built was this snapshot system, which was used to measure the performance of our interpreter over time, as we implemented new improvements. We periodically saved snapshots of our interpreter, and we ran all the snapshots against a common benchmark to note improvements and to see if our optimizations were effective.

Division of Work

While both of us contributed greatly to all sections on the code, towards the end we started to specialize some. Jack mostly worked on memory-related optimizations, while Yasyf mostly worked on ASM/IR related optimizations. However, both team members were directly involved in debugging and evaluating the other’s work. It’s hard to define a clear boundary where we split the work, since we each worked on the different sections as needed.