JVM ByteCode Interpreter written in Haskell (In under 1000 Lines of Code)

By Louis Jenkins

Presentation Schedule (≈15 Minutes)

- Discuss and Run the Virtual Machine first
  - <5 Minutes
- Syntax, Binding & Scope, Data Types, Control Flow, and Subprograms
  - <10 Minutes
  - +Code Snippets
- Future goals and plans
  - 1 minute

Virtual Machine

- Does
  - Accept and parse .class files
    - Can be generated by any JVM Language
      - Examples shown are generated from Scala and Java
  - Interpret a subset of ByteCode instructions
    - Loads, Stores, Arithmetic
  - Basic I/O support, Support for conditional expressions
    - ‘if…else if… else’, ‘for’, ‘while’
      - ‘for’ only supported in Java
        - Scala generates more complex bytecode

- Does Not
  - Contain a garbage collected heap
    - Variables exist on the stack only
  - Support side-effects
    - Only computations that operate purely on the operand stack and local variables work
      - I.E: An object that is duplicated on the stack are two different objects, and not a pointer to the heap (yet)
  - Support multi-threading
    - Monitors are not implemented
  - Have exception handling
    - Although relatively trivial to implement
  - Load the runtime
    - Relies on stubbed pseudo-implementations
      - I.E: $println$ uses Haskell’s built-in $putstrln$

Syntax

Currying, Function Declaration and Definition, Pattern Guards, and Function Calling

Currying

- The translation of an 'uncurried' function taking a tuple of arguments, into a sequence of functions taking only a single argument
- Example
  - $f(x, y) = z \equiv f: x \to (y \to z)$
  - The function f takes x as the input, and returns a function $f_x: y \to z$.
    - Note that the arrows are right associative, so $x \to (y \to z) \equiv x \to y \to z$

Function Syntax

- Functions take arguments as arrows
  - Considered the ‘Curried’ form of function application
  - Declaration arrows represent the types, but the names are decided in the definition
- Pattern Guards
  - Determine which function definition to call based on predicate
    - Represented with the ‘|’ character
- Functions arguments are passed sequentially
  - To disambiguate the function arguments, they can be wrapped in (…), or have the ‘$’ operator appended after the function.

```haskell
-- The main dispatcher logic
execute' :: StackFrame -> ByteCode -> IO ()
execute' frame bc
  -- NOP
  | bc == 0 = return ()
  -- Constants
  | bc >= 1 && bc <= 15 = constOp frame bc
  -- Push raw byte(s)
  | bc == 16 || bc == 17 =
    -- Special Case: 0x10 pushes a single byte, but 0x11 pushes a short
    (if bc == 16 then fromIntegral <$> getNextBC frame else getNextShort frame)
      >>= pushOp frame . fromIntegral
  -- Load from runtime constant pool
  | bc >= 18 && bc <= 20 =
    -- Special Case: 0x12 uses only one byte for index, while 0x13 and 0x14 use two
    (if bc == 18 then fromIntegral <$> getNextBC frame else getNextShort frame)
      >>= loadConstantPool env . fromIntegral >>= pushOp frame
  -- Loads
  | bc >= 21 && bc <= 53 = loadOp frame bc
  -- Stores
  | bc >= 54 && bc <= 86 = storeOp frame bc
  -- Special Case: 'dup' is used commonly but ignored, so we have to stub it
  | bc == 89 = return ()
  -- Math
  | bc >= 96 && bc <= 132 = mathOp frame bc
  -- Conditionals
  | bc >= 148 && bc <= 166 = cmpOp frame bc
  -- Goto: The address is the offset from the current, with the offset being
  -- the next two instructions. Since we advance the PC 2 (+1 from reading this
  -- instruction), we must decrement the count by 3 to correctly obtain the target.
  | bc == 167 = getNextShort frame >>= \jmp -> modifyPC frame (+ (jmp - 3))
  -- Return
  | bc == 177 = return ()
  -- Runtime Stubs
  | bc >= 178 || bc <= 195 = runtimeStub env frame bc
  | otherwise = error $ "Bad ByteCode Instruction: " ++ show bc
```

Binding & Scope Rules

Unlimited Extent, Lambdas, Lazy Evaluation, Thunks, and more…

Binding & Scoping Rules

- Referential Transparency
  - Variables defined are immutable
    - With some exceptions…
  - Since they are immutable, their outputs are always deterministic
  - Variables have Unlimited Extent
    - They exist for as long as they are referenced
      - Even variables of lambdas
- Lazy-Evaluation
  - Computations are delayed inside of ‘thunks’
    - Thunks contain ‘lazy’ computations that are only evaluated when needed.
- Immutability
  - All data is immutable, with some exception
    - The IO Monad needs side-effects to interact with the ‘RealWorld’
      - I.E: Printing to the console is a side-effect
    - ‘IORef’, ‘STRef’, ‘MVar’, ‘Tvar’, etc., all can maintain references to immutable to data that can be changed to point something else
      - Special Case: Software Transactional Memory
      - Underlying data is still immutable

```haskell
{- |
  Constructs a stack frame from the passed method. Each stack frame keeps track
  of it's local variables, operand stack, and code segment, which is composed
  of the instructions and the current program counter.
-}
createFrame :: Method -> IO StackFrame
createFrame meth = createFrame' >>= newIORef
  where
    createFrame' :: IO Stack_Frame
    createFrame' = newIORef ([] :: [Operand]) >>= \opstack -> newIORef 0 >>=
      \pc -> (createLocals . method_locals) meth >>= \locals ->
      return Frame {
        local_variables = locals,
        operand_stack = opstack,
        code_segment = Code {
          byte_code = method_code meth,
          program_counter = pc
        }
      }
        where
          createLocals :: Word16 -> IO [Local_Variable]
          createLocals n
            | n == 0 = return []
            | n > 0 = (:) <$> newIORef (VReference 0) <*> createLocals (n - 1)
            | otherwise = error $ "Error while attempting to create locals: n=" ++ show n

{- | Obtain the reference to the PC of this stack frame -}
getPC :: StackFrame -> IO (IORef Word32)
getPC frame = program_counter . code_segment <$> readIORef frame

{- | Obtain the PC of this stack frame -}
getPC' :: Integral a => StackFrame -> IO a
getPC' frame = getPC frame >>= \f -> fromIntegral <$> readIORef f
```

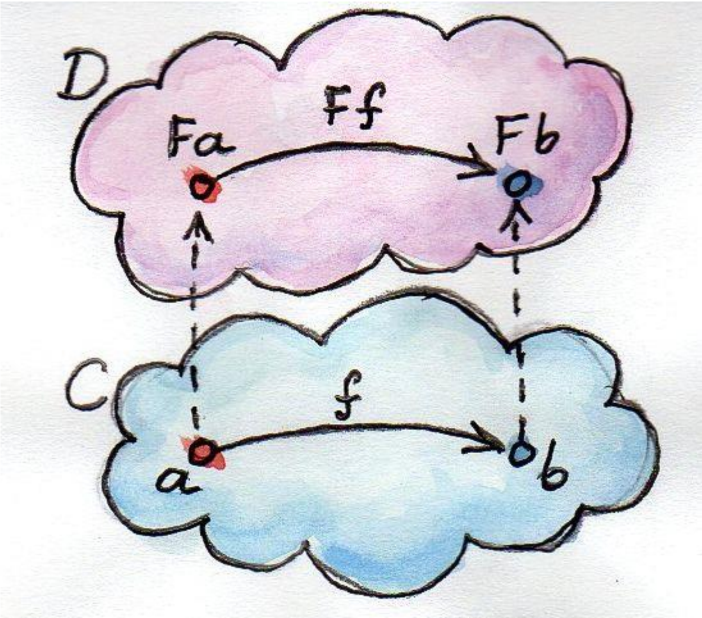
Control Flow

Functors, Applicative Functors, ‘Lazy’ Recursion and Evaluation, and Monads

Functors - Simplified

- A container for values that allow mapping of each of it’s values from one ‘category’ to another.
  - Category: Collection of Objects
    - I.E: Sets
- Example: Adding some constant to all elements in a list
  - $(+1) < \$ > \ [1..100] \equiv [2..101]$

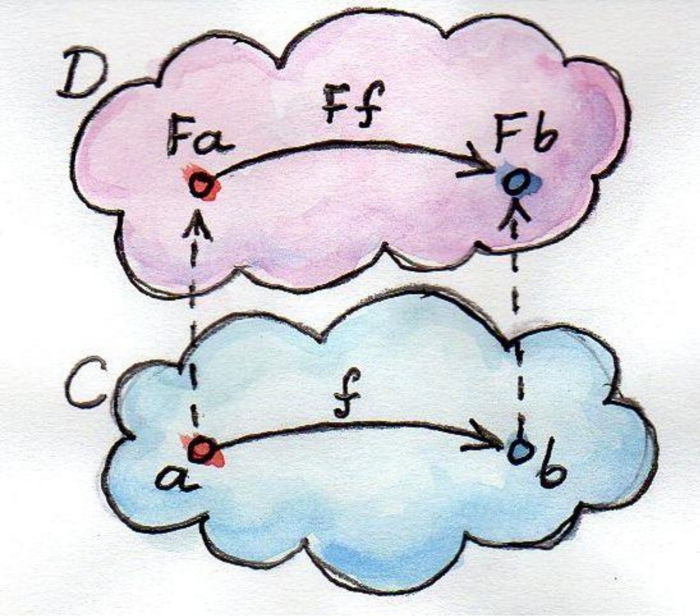
```haskell
class Functor f where
  -- Example implementation...
  -- fmap :: (a -> b) -> Just a -> Just b
  -- fmap f (Just x) = Just (f x)
  fmap :: (a -> b) -> f a -> f b

-- The infix operator for 'fmap', which is used
-- between both arguments. Example usage...
-- (+1) <$> Just 1
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```



Applicative Functors - Simplified

- A type of functor that allows partial applications
  - Partial Applications of Functions discussed later
- Why?
  - What if we want to add two functors together?
    - $(+) < \$ > Just\ 2 \equiv \left(Just\ (2\ +) :: Just\ (Int\ \rightarrow Int)\right)$
    - $fmap$ requires $(a \rightarrow b)$, not $f\ (a \rightarrow b)$ as the mapping function
  - Applicative does exactly that
    - $(+) < \$ > Just\ 2 <*> Just\ 2 = Just\ 4$

```haskell
class (Functor f) => Applicative f where
  -- Wraps the given value in the functor 'f'
  pure  :: a -> f a
  -- Similar to 'fmap', except the mapping function
  -- is held inside of a functor. Example implementation...
  -- (Just f) <*> (Just x) = Just (f x)
  (<*>) :: f (a -> b) -> f a -> f b
```



Monads

- A type of functor that allows "chaining" operations.
  - "Chaining" operations can be done using "bind", represented as $\gg=$
  - Allows you to form "pipelines" of instructions
    - Simulate side-effects
- Example: Processing User Input
  - $get \gg= process \gg= write$
    - Obtain the input $String$ with get
      - $get :: IO\ String$
      - $m = IO, a = String$
    - Process the input $String$ with $process$
      - $process :: String \to IO\ String$
      - $m = IO, a = String, b = String$
    - Write the processed $String$ with $write$
      - $write :: String \to IO(\,)$
      - $m = IO, a = String, b = (\,)$
  - How is this different from normal imperative programming?
    - There are no side-effects. The $String$ in each step is never mutated, but it appears as if it did!

```haskell
class Monad m where
  -- Takes the value 'a' from the monad 'm'
  -- and returns the result of the mapping
  -- function. Example Implementation...
  -- (Just x) >>= f = f x
  (>>=) :: m a -> (a -> m b) -> m b

  -- Just like '>>=' excepts it discards the result...
  -- This is needed because expressions are evaluated
  -- lazily, and this will force it's evaluation.
  -- Default implementation...
  -- x >> y = x >>= \_ -> y
  (>>) :: m a -> m b -> m b

  -- Same as 'pure' from Applicative
  return :: a -> m a

  -- For errors. Default implementation...
  -- fail = error
  fail :: String -> m a
```

Control Flow (Recursion)

- Recursion
  - Any and all ‘iteration’ is performed through recursion
    - Why?
      - Iteration requires mutation of some variable
        - All variables are immutable
- Infinite recursion is actually ‘safe’
  - Used to produce infinite data streams
    - Recursive calls only called when needed
- Example: Obtain first n Fibonacci Numbers
  - $fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$
  - $take\ n\ fibs$
  - Result of each call to $fibs$ is stored as evaluated inside of a thunk. The function used $(zipWith :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c])$ applies the function to the head of both lists (I.E: The last two values evaluated). $take$ will force it to evaluate only up to n times and collect the result.

```haskell
runtimeStub :: Runtime_Environment -> StackFrame -> ByteCode -> IO ()
runtimeStub env frame bc
:: Runtime_Environment -> StackFrame -> ByteCode -> IO ()
  | bc == 178 = void $ getNextShort frame
  -- invokevirtual: (append, println). NOTE: MUST HAVE ONLY ONE PARAMETER ELSE UNDEFINED
  | bc == 182 = getNextShort frame >>= \method_idx -> (readIORef . current_class) env
    >>= \c -> case methodName c method_idx of
      "append" -> ((\x y -> VString $ show y ++ show x) <$> popOp frame <*> popOp frame) >>= pushOp frame
      "println" -> popOp frame >>= print -- Defer I/O to Haskell
      "toString" -> return () -- StringBuilder object is already a 'VString'
      _ -> error "Bad Method Call!"
  -- Invokespecial is used to call <init>, but we don't deal with that... yet
  -- Invokestatic is also used Scala compatibility, which we can safely ignore
  | bc == 183 || bc == 184 = void $ getNextShort frame
  -- Laziness: 'new' must refer to StringBuilder... otherwise it's undefined anyway
  | bc == 187 = getNextShort frame >> pushOp frame (VString "")
  | otherwise = error $ "Bad ByteCode Instruction: " ++ show bc
      where
        methodName :: Class -> Word16 -> String
        methodName clazz method_idx = let
          cpool = constant_pool clazz
          method_ref = cpool !! fromIntegral method_idx
          name_and_type = cpool !! fromIntegral (name_and_type_index method_ref)
          utf8_name = cpool !! fromIntegral (name_index name_and_type)
          in show . utf8_bytes $ utf8_name

{- | Loads from local_variables to operand_stack -}
loadOp :: StackFrame -> ByteCode -> IO ()
loadOp frame bc
  -- Loads with the index as the next bytecode instruction
  | bc >= 21 && bc <=25 = getNextBC frame >>= getLocal' frame >>= pushOp frame
  -- Loads with a constant index (I.E: ILOAD_0 to ILOAD_3 have indice 0 to 3 respectively)
  | bc >= 26 && bc <= 45 = getLocal' frame ((bc - 26) `mod` 4) >>= pushOp frame
  | otherwise = error $ "Bad ByteCode Instruction: " ++ show bc

{- | Stores from operand_stack to local_variables -}
storeOp :: StackFrame -> ByteCode -> IO ()
storeOp frame bc
  -- Stores with the index as the next bytecode instruction
  | bc >= 54 && bc <= 58 = popOp frame >>= \op -> getNextBC frame >>= \idx -> putLocal frame idx op
    >> when (bc == 55 || bc == 57) (putLocal frame (idx + 1) (VReference 0))
  -- Stores with a constant index (I.E: ISTORE_0 to ISTORE_3 have indice 0 to 3 respectively)
  | bc >= 59 && bc <= 78 = let idx = ((bc - 59) `mod` 4) in
    popOp frame >>= putLocal frame idx
    -- Special Case: Double word-sized variables, such as 'long' and 'double' must
    -- take up two slots. We fit both types in a single slot, but the compiler generates
    -- ByteCode that are sensitive to these invariants, so we insert a dummy null reference
    -- in the second slot to restore that balance.
    >> when ((bc >= 63 && bc <= 66) || (bc >= 71 && bc <= 74))
    ((putLocal frame $ idx + 1) (VReference 0))
```

Data Types

Type-Classes and deriving/instantiating them

Data Types

```haskell
-- Wrap Java native primivitve types in Haskell types
data Value = VInt Int | VLong Integer| VFloat Float | VDouble Dou
  | VReference Object | VString String deriving (Eq, Ord)

instance Num Value where
  (+) (VInt x) (VInt y) = VInt (x + y)
  (+) (VLong x) (VLong y) = VLong (x + y)
  (+) (VFloat x) (VFloat y) = VFloat (x + y)
  (+) (VDouble x) (VDouble y) = VDouble (x + y)
  (+) _ _ = error "Bad Op: Addition"

  (-) (VInt x) (VInt y) = VInt (x - y)
  (-) (VLong x) (VLong y) = VLong (x - y)
  (-) (VFloat x) (VFloat y) = VFloat (x - y)
  (-) (VDouble x) (VDouble y) = VDouble (x - y)
  (-) _ _ = error "Bad Op: Subtraction"

  (*) (VInt x) (VInt y) = VInt (x * y)
  (*) (VLong x) (VLong y) = VLong (x * y)
  (*) (VFloat x) (VFloat y) = VFloat (x * y)
  (*) (VDouble x) (VDouble y) = VDouble (x * y)
  (*) _ _ = error "Bad Op: Multiplication"

  fromInteger = VInt . fromIntegral

instance Real Value where
  toRational (VInt x) = toRational x
  toRational (VLong x) = toRational x
  toRational (VFloat x) = toRational x
  toRational (VDouble x) = toRational x
  toRational _ = error "Bad Op: toRational"

instance Enum Value where
  toEnum _ = error "Bad Op: toEnum"
  fromEnum _ = error "Bad Op: fromEnum"
```

- Type Classes
  - Constructs that define methods
    - Even arithmetic operators are methods
  - Can sometimes be automatically derived
    - Only if the objects they are composed of all are instances of it
  - Can be used for type constraints of polymorphic functions
    - Specify that the generic type must implement the listed types
  - Have 'data constructors'
    - Remember: Same as a normal function
    - Can have 'field selectors'
    - Can have a 'default' values of undefined
      - Defined as ⊥, or 'bottom'
        - Also used for non-terminating functions and runtime errors
      - All types have this value in common
  - Can be instantiated by data types
    - Must implement required methods

Subprograms and Parameter Passing

Partial Applications of Functions (in theory and practice)

Partial Application of Functions (in Theory)

- Applying an argument to a function taking more than one argument, resulting in a function taking one less argument
  - Remember Currying
    - $f(x, y) = z \equiv f\colon x \to (y \to z) \equiv f\colon x \to y \to z$
  - Application: $f(x) \equiv f_x\colon y \to z$
    - 'Applying' x to f will result in a function f_x that takes the remaining arguments…
- In Haskell, all function arguments are applied this way!
  - Since all variables have unlimited extent, applied arguments are always safe to use!
- Example: The addition/plus binary operator…
  - $(+) :: Int \to Int \to Int$
    - $(+)\ 1\ :: Int \to Int$
      - $(+)\ 1\ 1\ :: Int$

Subprograms and Parameter Passing

- Partial Application of Functions (in Practice)
  - Data Constructors for a type are just functions, and like such can be partially applied
  - With a combination of the results from $getNext^*$, which returns a $Parser\ Word^*$, that result can be passed to the data constructor through application.
- Why is this important?
  - Arguments can be passed from functors
  - Arguments can also be passed by value
  - Cuts out the amount of boilerplate
- Functions Composition
  - The '.' operator denotes function composition.
  - $(g.f)(x) \equiv (g \circ f)(x) = g\big(f(x)\big)$
    - Pronounced g "after" f of x
  - $(.) :: (b \to c) \to (a \to b) \to (a \to c)$

```haskell
{-
  ClassFile Parser
-}

parseClassFile :: ByteString -> ClassFile
parseClassFile = evalState $ getNextInt >> getNextShort >> getNextShort -- Discard magic and version infor
  >> parseConstants >>= \cp -> ClassFile cp <$> getNextShort <*> getNextShort <*> getNextShort
  <*> parseInterfaces <*> parseFields cp <*> parseMethods cp <*> parseAttributes cp


parseConstants :: Parser [CP_Info]
parseConstants = getNextShort >>= \n -> replicateM (fromIntegral n - 1) parseConstant
  >>= \cp -> return $ Dummy_Info : cp
    where
      parseConstant = do
        t <- getNextByte
        case t of
          7 -> Class_Info t <$> getNextShort
          9 -> Fieldref_Info t <$> getNextShort <*> getNextShort
          10 -> Methodref_Info t <$> getNextShort <*> getNextShort
          11 -> InterfaceMethodref_Info t <$> getNextShort <*> getNextShort
          8 -> String_Info t <$> getNextShort
          3 -> Integer_Info t <$> getNextInt
          4 -> Float_Info t <$> getNextInt
          5 -> Long_Info t <$> getNextInt <*> getNextInt
          6 -> Double_Info t <$> getNextInt <*> getNextInt
          12 -> NameAndType_Info t <$> getNextShort <*> getNextShort
          1 -> Utf8_Info t <$> parseUTF8
            where
              parseUTF8 = getNextShort >>= \n -> UTF8_Bytes <$> getNextBytes (fromIntegral n)
          15 -> MethodHandle_Info t <$> getNextByte <*> getNextShort
          16 -> MethodType_Info t <$> getNextShort
          18 -> InvokeDynamic_Info t <$> getNextShort <*> getNextShort
          _ -> undefined
```

Virtual Machine – Plans and Goals

- Implement a Heap that takes advantage of Haskell’s GC
- Implement all ByteCode Instructions
  - Bootstrap Classloading
  - Monitors
  - Exception Handling
- Refactor, Refactor, Refactor…
  - Needs vast improvements!