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

  • \( f(x, y) = z \equiv f : x \rightarrow (y \rightarrow z) \)
  
  • The function \( f \) takes \( x \) as the input, and returns a function \( f_x : y \rightarrow z \).
    • Note that the arrows are right associative, so \( x \rightarrow (y \rightarrow z) \equiv x \rightarrow y \rightarrow z \)
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.
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
    • ‘IORref’, ‘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
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]
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?
    • (+) <$> Just 2 ≡ (Just (2 +) :: Just (Int → Int))
    • fmap requires (a → b), not f (a → b) as the mapping function
  • Applicative does exactly that
    • (+) <$> Just 2 <*> Just 2 = Just 4
Monads

- A type of functor that allows “chaining” operations.
  - “Chaining” operations can be done using “bind”, represented as >>=
  - Allows you to form “pipelines” of instructions
    - Simulate side-effects

- Example: Processing User Input
  - \( \text{get } \gg= \text{process } \gg= \text{write} \)
    - Obtain the input String with get
      - \( \text{get} :: \text{IO String} \)
        - \( m = \text{IO}, a = \text{String} \)
    - Process the input String with process
      - \( \text{process} :: \text{String} \rightarrow \text{IO String} \)
        - \( m = \text{IO}, a = \text{String}, b = \text{String} \)
    - Write the processed String with write
      - \( \text{write} :: \text{String} \rightarrow \text{IO()} \)
        - \( m = \text{IO}, a = \text{String}, b = () \)
  - 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!
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
  - $fibs = 0 : 1 : \text{zipWith (+) fibs (tail fibs)}$
  - $\text{take } n \text{ fibs}$
  - Result of each call to $fibs$ is stored as evaluated inside of a thunk. The function used ($\text{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c]$) applies the function to the head of both lists (I.E. The last two values evaluated). $\text{take}$ will force it to evaluate only up to $n$ times and collect the result.
Data Types

Type-Classes and deriving/instantiating them
Data Types

• **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
    • \( f(x, y) = z \equiv f : x \rightarrow (y \rightarrow z) \equiv f : x \rightarrow y \rightarrow z \)
  • Application: \( f(x) \equiv f_x : y \rightarrow z \)
    • ‘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...
  • \((+): Int \rightarrow Int \rightarrow Int\)
    • \((+): Int \rightarrow Int\)
      • \((+): Int \rightarrow Int\)
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 \( \textit{getNext} \), which returns a \textit{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 ‘\( \cdot \)’ operator denotes function composition.
    \( (g \cdot f)(x) \equiv (g \circ f)(x) = g(f(x)) \)
  • Pronounced \( g \) “after” \( f \) of \( x \)
  • \( (\cdot) : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \)
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!