The Jazz Language

By Ryan Ramsdell

Why Jazz

- Functional programming is great
- Haskell is a great FP language, but hard for beginners
  - “A monad is just a monoid in the category of endofunctors, what’s the problem?”
- Jazz
  - Takes imperative ideas and offers a compromise between them and purely functional concepts
  - Can still utilize the readability and optimization advantages of functional languages

Current Features

- ADTs
- Easy to understand syntax
- Fairly performant
- Strong, static typing
- Amazing type inference
- Immutable variables
- Pattern matching
- First class functions
- Curry-by-default
- No need to be a category theory expert

Future Features

- More polymorphism support
    - GADTs
    - Kinds
    - Rank-n-types
    - Existential quantification
    - And more...
- Incredibly performant
    - In the future, I will write an interpreter for Jazz in Haskell, and then bootstrap/self host the compiler and generate LLVM IR
- Intelligent lazy vs. strict evaluation selection
    - Avoids the “foldl vs foldr” problem
- Maybe some simple metaprogramming

Syntax

Basic Syntax

```
id :: a -> a
id = \(a) -> a
head :: [a] -> a
head = \([hd | rest]) -> hd
tail :: [a] -> [a]
tail = \([_ | tl]) -> tl

i = 0 + 1. -- The 0 + 1 will get optimized out into just "1"
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]

addTen = (+10)
-- Another way to implement addTen:
-- addTen = \(x) -> x + 10

addTenToAllInList = map addTen
newNums = addTenToAllInList nums

print! (addTen i)
print! newNums
print! $ (head newNums) == 12
-- Adding 10 to the first element of newNums and printing
print! $ addTen $ head newNums
```

Code Organization

```
module std::Data {
  module List {
    -- [a] is sugar for `List a`
    head :: [a] -> a
    head = \([hd | rest]) -> hd
  }
  module Foo {
    myFunc = undefined
  }
}

-- Inside another module
module MyProgram::MyModule {
  import std::Data::List
  import std

  doSomething = \(x, y) -> (head x, std::Data::Foo::myFunc y)
}
```

Data Classes & Traits

```
data Ordering {
  GT,
  LT,
  EQ
}

trait Ord(a) {
  (>) :: a -> a -> Bool
  (>) = \(lhs, rhs) -> (cmp lhs rhs) == GT

  (>=) :: a -> a -> Bool
  (>=) = \(lhs, rhs) -> contains [GT, EQ] (cmp lhs rhs)

  -- etc
}

impl Ord(Int)) {
  (>) = \(lhs, rhs) -> (#intCmp lhs rhs) == GT
  (>=) = \(lhs, rhs) -> contains [GT, EQ] (#intCmp lhs rhs)
}
```

Current Compiler Stack

- Parsed into an AST using megaparsec
- Desugaring
- Type inference
- Optimization
- JS code generation

Code Generation Sample

```
i = 0 + 1. -- The 0 + 1 will get optimized out into just "1"
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9].

addTen = (+10).
-- Another way to implement addTen:
-- addTen = \(x) -> x + 10

addTenToAllInList = map addTen.
newNums = addTenToAllInList nums.

print! (addTen i).
print! newNums.
print! $ (hd newNums) == 11.
print! $ (hd newNums) == 12.
-- Adding 10 to the first element of newNums and printing
print! $ addTen $ hd newNums.
```

```
// Standard library
const add = l => r => l + r
const subtract = l => r => l - r
const multiply = l => r => l * r
const divide = l => r => l / r
const map = f => xs => xs.map(f)
const hd = ([x]) => x
const tl = ([, ...xs]) => xs
const isEq = l => r => l == r

// Actual Program
let i = 1
let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
let addTen = (__partialInfixLambdaParam0) => add(__partialInfixLambdaParam0)(10)
let addTenToAllInList = map(addTen)
let newNums = addTenToAllInList(nums)
console.log(addTen(i))
console.log(newNums)
console.log(isEq(hd(newNums))(11))
console.log(isEq(hd(newNums))(12))
console.log(addTen(hd(newNums)))
```

The standard library functions that just map operators to named functions are needed because operators can't be called like functions in Javascript, and also because functions in JS are not curried by default. (e.g I can't call `+(1)(2)` or `+(1)(2)`