~~The Case for Haskell~~ Ask More of Your Languages

Allele Dev (@queertypes)

July 22, 2014

Contact Me!

- Github: cabrera
- Twitter: @queertypes
- Blog: Read, Review, Refactor

What?

- Programming languages shape how we solve problems
- Types are a valuable tool for enforcing **and** communicating design
- Many mainstream languages are missing key ingredients for sound abstraction

Why the Change in Title?

- Not the message I want to share
  - "Us" vs. "Them" mentality is toxic
  - I want to unify, not divide.
  - We learn more together!
- Most concepts in this talk are applicable to all FP languages
  - Even those lacking statically checked types!

Generalizing This Talk

- Haskell is **my** medium for these concepts
- They're *largely* applicable to all of:
  - Rust
  - Ocaml
  - F#
  - F*
  - Standard ML
  - Idris
  - Swift
  - Scala
  - Elm
  - Purescript

Generalizing This Talk

- And with gradual typing or additional typing mechanisms
  - Racket + Types
  - Clojure + Types
  - Erlang + Types

Why Do Types Matter? (Briefly)

- They help prove our programs correct
- Complementary to testing
- They **communicate** intent and abstractions

Our First Type Error in a REPL

```
> 1 + "1"
```

Our First Type Error in a REPL

```
> 1 + "1"
    No instance for (Num [Char]) arising from a use of ‘+’
    In the expression: 1 + "1"
    In an equation for ‘it’: it = 1 + "1"
```

Our First Type Error in a REPL

```
> 1 + "1"
    No instance for (Num [Char]) arising from a use of ‘+’
    In the expression: 1 + "1"
    In an equation for ‘it’: it = 1 + "1"
```

- Statically checked, typed languages are WAT-resistant

More Information: Our First Type Error in a Source File

```haskell
-- wat-resistant.hs
main = print $ 1 + "1"
```

More Information: Our First Type Error in a Source File

```
alejandro@rainbow-generator:~:$ ghc wat-resistant.hs
[1 of 1] Compiling Main                ( wat-resistant.hs,
                                         wat-resistant.o )

wat-resistant.hs:1:18:
  No instance for (Num [Char]) arising from a use of ‘+’
  In the second argument of ‘($)’, namely ‘1 + "1"’
  In the expression: print $ 1 + "1"
  In an equation for ‘main’: main = print $ 1 + "1"
```

- 150ms to type check, and what did we learn?
  - Line and column number of the error: **1:18**
  - What the problem is: can't add a Num and a [Char]
  - With a trace zoning in on the problematic source

What Makes a Language Capable of This?

The Key Ingredients (with a Heavy Dose of Jargon)

- A solid type system
  - User-defined types
  - Parametric polymorphism
  - Product types
  - Sum types
  - Recursive types
- Optionally: a *great* type system
  - Higher kinds
  - Effect tracking
- Type inference: Hindley-Milner or better
- Pattern matching

User-defined Types

- A typedef that the compiler treats as distinct from the original type
  - A rough approximation: wrap all primitive types in C with structs

```
////////////////////////////////
struct PersonName {char* name;};
struct Address {char* name;};
struct ProcessId {int pid;};
////////////////////////////////

----------------------------
newtype PersonName = String
newtype Address = String
newtype ProcessId = Int
----------------------------
```

Parametric Polymorphism

- A powerful means to express generic functions
  - The good parts of c++ templates without the bad parts

```
-------------------------------
id :: a -> a
map :: (a -> b) -> [a] -> [b]
sort :: Ord a => [a] -> [a]
-------------------------------
```

Product Types

- Compile-time tuples, pairs of information
- Representable in most languages using structs/classes

```
--------------------------
swap :: (a, b) -> (b, a)
--------------------------
```

Sum Types

- A disjoint union, a series of alternates, compile-time enforced enum
- Not available in most main-stream languages

```haskell
data Maybe a = Just a | Nothing

data ROYGBIV =
    Red | Orange | Yellow | Green | Blue | Indigo | Violet

data LogMessage =
      UserLine String String
    | ErrorLine Int String
    | OtherLine String
```

Recursive Types

- Can express infinite data structures and/or hierarchies at compile-time

```haskell
data List a = List a | Nil
data BinaryTree a =
      Node a
    | Branch (BinaryTree a) (BinaryTree a)

-- great for collision detection
data QTree a =
      QLeaf a
    | QBranch (QTree a) (QTree a) (QTree a) (QTree a)
```

Higher Kinds

- Types that exist a level above types
  - Tell us what a valid type looks like
- Thinking in terms of shapes helps explain:
  - **Types** describe the *shape* of **values**
  - **Kinds** describe the *shape* of **types**

```
-------------------------------------
-- Maybe has kind: * -> *
-- Prevents nonsense like: Maybe Maybe
data Maybe a = Just a | Nothing

-- Either has kind: * -> * -> *
-- for laughs: Either Maybe Maybe
data Either a b = a | b
-------------------------------------
```

Effect Tracking

- A technique that arises from combining
  - parametric polymorphism
  - higher kinds
- Allows for compiler to detect when:
  - code would change state
  - interact with "world": disk/terminal/network I/O
- Enables safe software transactional memory

```
---------------------------------
printName :: String -> IO ()
printName name = print name

readPidFile :: String -> IO Int
readPidFile path = do
  handle <- openFile path ReadMode
  contents <- hGetContents handle
  return (read contents)
---------------------------------
```

Type Inference

- The compiler makes **conservative** attempts to automatically fill in type information
  - If there's no inheritance/sub-typing, the answer is guaranteed to be exact
- Bare minimum you have to do: annotate top-level declaration

Pattern Matching

- Write your functions against the shape of the data
  - Better than a switch statement
  - Less verbose than if-else-if-else chains
- Works with recursive types, too! (not shown)

```haskell
data Move = Rock | Paper | Scissor
data Outcome = Win | Lose

rockPaperScissor :: Move -> Move -> Outcome
rockPaperScissor Rock Scissor = Win
rockPaperScissor Scissor Paper = Win
rockPaperScissor Paper Rock = Win
rockPaperScissor _ _ = Lose
```

Example Data Type: JSON

```haskell
-- json.hs
data JValue =
    JString String
  | JNumber Double
  | JBool Bool
  | JNull
  | JObject [(String, JValue)]
  | JArray [JValue]

render :: JValue -> String
render (JString s) = s
render (JNumber i) = show i
render JNull = "null"
render (JObject o) = "{" ++ obj o ++ "}"
  where obj [] = ""
        obj ps = intercalate "," (map renderObj ps)
        renderObj (k,v) = show k ++ ":" ++ render v
```

Example Data Type: JSON

- Can do nifty things like:

```
$ ghci json.hs
> render (JObject [("cat", JNumber 10)])
> "{\"cat\": 10}"
```

Example Data Type: JSON

- . . . and if we ask the compiler to warn us extensively:

```
$ ghci -Wall json.hs
[1 of 1] Compiling Main             ( json.hs, interpreted )

json.hs:12:1: Warning:
    Pattern match(es) are non-exhaustive
      In an equation for ‘render’:
          Patterns not matched:
              JBool _
              JArray _
```

Example: Refactoring Rock, Paper, Scissors

- Notice in our previous version we excluded the possibility of a Draw outcome
- Let's fix that!

Example: Refactoring Rock, Paper, Scissors

```haskell
-----------------------------------------
data Move = Rock | Paper | Scissor
data Outcome = Win | Lose

rockPaperScissor :: Move -> Move -> Outcome
rockPaperScissor Rock Scissor = Win
rockPaperScissor Scissor Paper = Win
rockPaperScissor Paper Rock = Win
rockPaperScissor _ _ = Lose
-----------------------------------------
```

Example: Refactoring Rock, Paper, Scissors

```haskell
-----------------------------------------
data Move = Rock | Paper | Scissor deriving Eq
data Outcome = Win | Lose | Draw

rockPaperScissor :: Move -> Move -> Outcome
rockPaperScissor Rock Scissor = Win
rockPaperScissor Scissor Paper = Win
rockPaperScissor Paper Rock = Win
rockPaperScissor left right | left == right = Draw
rockPaperScissor _ _ = Lose
-----------------------------------------
```

Types Matter - Take Two

- **Iteratively** prove your programs correct: Curry-Howard style
- Stephanie Weirich elaborates this eloquently:
  - Lightweight, machine-checked, and ubiquituous verification
  - Wonderful for refactoring safely
- Just as important, types communicate to others **unambiguously**
  - What you mean and what your abstractions look like
- Altogether, they show and preserve the **shape** of your program

The Art and Math of Abstraction

- Art: personal, fluid, experimental
- Math: collective, rigid, proven
- Both are important!
- Type systems enable and amplify both the math and the art

The Art and Math of Abstraction

```haskell
-- art: how do you encode your swirlies?
type Radius = Int
type Density = Int
data Swirlies = Swirlies Radius Density

-- math: use Monoid for combining Swirlies
-- monoid: an identity element
--         and an associative binary operator
--   (think: x + 0 = x, (+) is operator, 0 is identity)
instance Monoid Swirlies where
    mempty = Swirlies 0 1
    mappend (Swirlies lrad ldens) (Swirlies rrad rdens) =
        Swirlies (lrad + rrad) (rdens * ldens)
```

Thoughts on Productive Criticism of a Programming Language

- A programming language is the frontend for your type system
  - Judge a language based on its ability to support your abstractions at compile-time

However!

- Must also account for:

However!

- Must also account for:
  - social considerations: community, diversity, communication, etc.

However!

- Must also account for:
  - social considerations: community, diversity, communication, etc.
  - systems considerations: real-time, performance, etc.

However!

- Must also account for:
  - social considerations: community, diversity, communication, etc.
  - systems considerations: real-time, performance, etc.
  - enterprise considerations: hiring, platform support, etc.

What Does the Future Hold?

- Type system advancements:
  - Dependent types: values at the type level
  - Linear types: capture use-only-once, close-after-open, RAII at type level
  - Smarter gradual typing, gradual effect systems
- Smarter tools:
  - Search engines for functions: hoogle hayoo
  - Editors: Lamdu
  - Code generation: Rest, Ivory
- Better abstractions:
  - functional reactive programming
  - parsing
  - crypto
  - performance
- Better module systems
- More learning resources
- Better communities

Ask More of Your Languages

- Together, in pursuit of software that works

Ask More of Your Languages

- Together, in pursuit of software that works
- Individually, as you explore what makes technology exciting for you

Ask More of Your Languages

- Together, in pursuit of software that works
- Individually, as you explore what makes technology exciting for you
- Let's ask more of **our** languages

Ask More of Your Languages

- Together, in pursuit of software that works
- Individually, as you explore what makes technology exciting for you
- Let's ask more of **our** languages
- Towards a better tomorrow

Thank You!