Zen and the Art of Haskell Types

Lucian Mogosanu

01.07.2014

Questions

- What are types?
- Why do we need types/type checking?
- How do we make a friend of Haskell types?

What are types?

Loose definition:

- Semantic annotations for data structures
  - "Bob is a person"
  - "x is an integer"
  - "g is a graph"
- ... but also for algorithms
  - "f is a function"
  - "p is a logical proposition"
  - "(another) p is a program"

Digression

What types does your PC understand?

Digression

What types does your PC understand?

- Integers (sometimes with a sign)
- Memory addresses
- Instructions

What types does your PC understand? (2)

- Consider the following sequence of instructions:

```
1: r0 <- 0
2: r1 <- val@r0 ; get val from mem addr in r0
```

- … or the following

```
1: r0 <- 0
2: goto instr@r0 ; jump to instr at mem addr in r0
```

- What are the semantics of these programs?

What types does your PC understand? (3)

- Consider the following sequence of instructions:

```
1: r0 <- 0
2: r1 <- val@r0
```

- 0 is a “special” **address**
  - but an ordinary **integer**!
- Accessing arbitrary addresses can cause programs to
  - fail
  - (*or worse*) misbehave
  - and (*even worse*) help the NSA spy on you.

So, why do we need type checking?

- We're careless
- We need a tool to whip our fingers when we go wrong
- We need to be able to reason about our programs
  - "Lightweight formal verification"

Type Signatures

- In set theory, we would say, e.g. $-2 \in \mathbb{Z}$
- In Haskell, we say:

```haskell
-2 :: Integer
```

Basic Haskell Types

```haskell
3 :: Int
120894192912541294198294982 :: Integer
3.14 :: Float
'a' :: Char
"The cake is a lie" :: String
True :: Bool
```

Compound Types: Lists

```
[3, 4, 5] :: [Int]
['a', 'b'] :: [Char] -- String
[[3,4], []] :: [[Int]]
```

Compounde Types: Lists (2)

- What happens if we write:

```
[3, 4, 'a', 'b'] :: [Int]
```

Compounde Types: Lists (2)

- What happens if we write:

```
[3, 4, 'a', 'b'] :: [Int]
```

- The type checker will scream:

```
Couldn't match expected type `Int' with actual type `Char'
In the expression: 'a'
In the expression: [3, 4, 'a', 'b'] :: [Int]
In an equation for `it': it = [3, 4, 'a', ....] :: [Int]
```

- **expected** type: `Int`
- **actual** type: `Char`

Compound Types: Pairs/Tuples

```
() :: ()
(3, "The cake is a lie") :: (Int, String)
(3, True, 3.14, "It's July") :: (Int, Bool, Float, [Char])
 (('a', 5), "A String") :: ((Char, Int), String)
```

Functions and Type Variables (1)

```
f x = x
```

```
f :: Int -> Int
```

Why not:

```
f True
```

Functions and Type Variables (2)

```
f x = x
```

```
f :: a -> a
```

- Capital letter $\rightarrow$ proper type
- Small letter $\rightarrow$ type variable (stands for multiple types)

We say that `f` is **polymorphic**

- Single implementation
- Multiple types

Functions and Type Variables (3)

- Functions are curried

```
a -> b -> c
a -> (b -> c) -- same as above
(a -> b) -> c -- different type
```

Types as documentation (1)

- Which functions have type `a -> a`?
  - excluding dummy cases
    - `f x = error "undefined"`
    - `f x = undefined`
    - `f x = f x`
  - want the simplest possible expression
    - `f x = head [...]`

Types as documentation (1)

- Which functions have type `a -> a`?
  - excluding dummy cases
    - `f x = error "undefined"`
    - `f x = undefined`
    - `f x = f x`
  - want the simplest possible expression
    - `f x = head [...]`
- The natural solution:

```
f x = x
```

Types as documentation (2)

- Which functions have type `(a, b) -> a`?

Types as documentation (2)

- Which functions have type `(a, b) -> a`?
- A single solution:

```
f (x, y) = x
```

Types as documentation (3)

- What are the types of:

```
3
3.0
2 + 3
2 + 3.0
2 / 3
```

Types as documentation (3)

- What are the types of:

```
3
3.0
2 + 3
2 + 3.0
2 / 3
```

- Signatures of `(+)`, `(/)`

Consulting Types in GHCi

- :t expression

Hayoo, Hoogle

- hoogle
  `http://www.haskell.org/hoogle`
  `https://www.fpcomplete.com/hoogle`
- hayoo
  `http://holumbus.fh-wedel.de/hayoo/hayoo.html`

Programmer defined Types (1)

- How is a `String` a synonym of `[Char]`?

```
type String = [Char]

type Point = (Double, Double)
type Size = (Double, Double)
type Vector = (Double, Double)

area :: Size -> Double
area (x, y) = x * y

p :: Point
p = (3.14, 4.2)

area p
```

Programmer defined Types (2)

- Type synonyms help only at lexical level
- Compiler sees the base type and works with it

Programmer defined Types (3)

- How do we build our own types?

```
data Colour = Red | Green | Blue | ...
```

- "The type Colours can have the values Red, Green, Blue, and so on"

```
data Bool = True | False
```

- A type is a collection of **constructors**
- Constructors start with capital letters

Programmer defined Types (4)

- Constructors can have **fields**

```
data Person
    = Male String Int -- name and age
    | Female String String Int -- name, maiden name, age
```

- Constructors are **functions**

```
Female :: String -> String -> Int -> Person
```

```
Male "Winston Smith" :: Int -> Person
```

Programmer defined Types (5)

- Types can be defined in terms of type variables
  - **parametric types**
- Lists are parametric: `[a]`

```
data Container a = Empty | Holding a
```

Interesting Haskell Types

```haskell
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Trees (Recursive Data Types)

```haskell
data BinaryLeafTree a =
      BLTLeaf a
    | BLTNode (BinaryLeafTree a) (BinaryLeafTree a)
data BinaryTree a =
      BTLeaf a
    | BTNode (BinaryTree a) a (BinaryTree a)
data RoseTree a =
      RTLeaf a
    | RTNode a [RoseTree a]
```

Working with Programmer defined Types

```haskell
data Container a = Empty | Holding a

isEmpty :: Container a -> Bool
isEmpty Empty = True
isEmpty _ = False
-- isEmpty (Holding _ ) = False

place :: Container a -> a -> Container a
place Empty x = Holding x
place _ _ = error "Container already full"
```

Space optimization

Scenario:

- My type has one constructor with one field, e.g.:

```
data OneFieldOnly a = Constructor a
type OneFieldOnly a = a
```

Space optimization

Scenario:

- My type has one constructor with one field, e.g.:

```
data OneFieldOnly a = Constructor a
type OneFieldOnly a = a
```

- **data** – needs too much extra-memory (constructor tags, thunk tags)
- **type** – doesn't offer type safety
- **newtype** – have the cookie and eat it too

```
newtype OneFieldOnly a = Constructor a
```

Smart Constructors

```
newtype Nat = MkNat { fromNat :: Int }
```

- `MkNat` → can take negative values
- **Smart constructor** → function that performs extra checks on input

```
toNat :: Int -> Nat
```

Record Types (1)

```
data Person = P String String String
    Int Int Person Person
```

- Which one is the father's name? (can use type synonyms)
- What happens if we:
  - Change field order by mistake?
  - Add another parameter?

Record Types (2)

```haskell
data Person = P
    { name :: String
    , address :: String
    , nationality :: String
    , age :: Int
    , numberOfChildren :: Int
    , father :: Person
    , mother :: Person
    }
```

- Each field is a function

```haskell
name :: Person -> String
P :: String -> String -> String -> Int -> Int ->
    Person -> Person -> Person
```

Typeclass Constraints (1)

```
data RoseTree a =
    RTLeaf a
  | RTNode a [RoseTree a]
```

```
> :t RTNode 3 []
> RTNode 3 [] == RTNode 3 []
> RTNode 3 []
```

Typeclass Constraints (2)

- We must "enrol" `RoseTree` into `Show` and `Eq`

```
class Show a where
    show :: a -> String

class Eq a where
    (==) :: a -> a -> Bool
```

Typeclass Constraints (3)

- Haskell can automatically derive most typeclass instances:

```
data RoseTree a =
    RTLeaf a
  | RTNode a [RoseTree a]
  deriving Show
```

Typeclass Constraints (4)

- Final definition:

```
data RoseTree a =
    RTLeaf a
  | RTNode a [RoseTree a]
  deriving (Show, Eq)
```