Designing Type-Safe Haskell APIs

Presented by: Michael Snoyman

What is type safety?

- Combination of **static** and **strong** typing
- Static typing: type errors caught compile time
- Strong typing: express invariants in the types

Strong/weak vs dynamic/static

What does `"hello" + 1` do?

	Strong	Weak
Static	Haskell Compile-time error	C "ello"
Dynamic	Python Run-time error	Javascript "hello1"

Static typing

- Compare to dynamic (e.g., Python, Ruby)
- Mostly a binary choice
- Even in statically typed languages, some dynamic typing exists (RTTI, reflection, Typeable)

Strong typing

- Compare to weak (e.g., Perl, Javascript)
- Not a binary choice: there's a large spectrum
- Not just a language feature: libraries and programming style have a strong impact
- Some languages make strong typing easier

Weakly typed Haskell

```
isAdult :: String -> IO ()
isAdult input
    | read age < 18 = putStrLn $ name ++ " is not an adult"
    | otherwise     = putStrLn $ name ++ " is an adult"
  where
    [name, age] = words input

main = do
    isAdult "Alice 25"
    isAdult "Bob 17"
    isAdult "Chris Johnson 17"
```

Motivation

Full blog post explaining *why* this matters:

http://www.yesodweb.com/blog/2012/08/webinar-oreilly

tl;dr: Reliably catch bugs at compile time instead of run time, when it's cheaper to fix.

Basics

Cheap newtypes

Bad:

```
verifyUser :: String -> String
           -> IO Bool
```

Better:

```
newtype Username = Username String
newtype Password = Password String
data Validity = Valid | Invalid
verifyUser :: Username
           -> Password
           -> IO Validity
```

Use the right datatype

- Use Map instead of assoc list
    - if order doesn't matter
- Use Set instead of list
- Don't be afraid to combine them

```
Map Username (Set Permission)
```

Make sure to use the right kind of union, e.g.:

Express invariants in types

- User must provide phone number, or email address, or both

Bad: `(Maybe Phone, Maybe Email)`

Good:

```
data ContactInfo
    = OnlyPhone Phone
    | OnlyEmail Email
    | PhoneAndEmail Phone Email
```

Use the right libraries

```
type FilePath = String
```

No type safety at all.

Use: `system-filepath`

Similarly: text, bytestring, blaze-html, ...

The Strings Issue

OverloadedStrings

- Makes it cheap to create newtypes
- Simple literal syntax for ByteString, HTML
- Replace `String` with improved `Text`
- Separate type for XML names
  - Compare to the Java solution: double the methods!
- Downside: no compile time checking
  - Not a huge problem in practice
  - Can always use QuasiQuotes instead

text versus bytestring

- Need to explicitly state character encoding
- Works as a tool for explanation

```
encodeUtf8 "שלום" =
    "\215\169\215\156\215\149\215\157"

putStrLn (encodeUtf8 "שלום") -- compile time error
```

blaze-html

- Automatic entity escaping (avoids XSS)
- Explicit functions to avoid escaping
- Newtypes like Textarea have special features

```
renderHtml "<unsafe>" == "&lt;unsafe;&gt"

renderHtml $ preEscapedToMarkup "<b>Hello!</b>" == "<b>Hello!</b>"

renderHtml $ toHtml $ Textarea "Hello\nWorld" == "Hello<br>World"
```

Going too far

- Ascii data is neither Text nor ByteString
- Idea: create a newtype!
- Result: lots of complaints, too difficult to use
- Lesson learned: sometimes safer != better

Type tricks, extensions

Phantom data types

- Problem: all database keys look the same
- Solution: use a phantom

```
data Person  = Person  Name
data Vehicle = Vehicle Make Model

newtype Key table = Key Int
type PersonKey  = Key Person
type VehicleKey = Key Vehicle

get :: Key table -> DB (Maybe table)
```

GADTs and data kinds

```
-- Name, age, and ID. Don't actually use bare Ints
-- like that in practice!
data Person = Person String Int Int

-- Automatic promotion: Sortable is a kind, constructors
-- are types
data Sortable = IsSortable | NotSortable

data PersonField value (s :: Sortable) where
    PersonName :: PersonField String NotSortable
    PersonAge  :: PersonField Int    IsSortable
    PersonId   :: PersonField Int    NotSortable
```

GADTs and data kinds (2)

```
data PersonFilter where
    (:==) :: Eq value => PersonField value s -> value
                      -> PersonFilter
    (:/=) :: Eq value => PersonField value s -> value
                      -> PersonFilter

data PersonSort where
    Asc  :: Ord value => PersonField value IsSortable
                      -> PersonSort
    Desc :: Ord value => PersonField value IsSortable
                      -> PersonSort
```

GADTs and data kinds (3)

```
query :: [PersonFilter] -> [PersonSort]
      -> [Person] -> [Person]

query [PersonName :== "Alice"] [] -- correct
query [PersonName :/= "Alice"] [] -- also fine
query [PersonName :== True]    [] -- compile error
query [] [Asc PersonAge]          -- no problem
query [] [Desc PersonId]          -- not sortable!
```

Type parameters

- Read a list of employees, some have IDs
- Assign IDs to employees without

```
newtype EmployeeId = EmployeeId Int
data Employee eid  = Employee Name eid

readEmployees  :: FilePath -> IO [Employee (Maybe EmployeeId)]
assignId       :: Employee (Maybe EmployeeId) -> IO (Employee EmployeeId)
writeEmployees :: FilePath -> [Employee EmployeeId] -> IO ()

readEmployees inFile >>= mapM assignId >>= writeEmployees outFile
```

Keep it general

- Program to typeclasses when possible
- Use `Monad m` instead of `IO`
    - Won't accidentally perform actions
    - Code reuse
- `Monoid` covers a lot of use cases too
- Downside: more confusing error messages

Examples from Yesod

The boundary issue

- You lose all type safety when interacting with the outside world
- Solution: keep everything strongly typed
- Render at the last moment
- Parse to strong types immediately

Example: type-safe URLs

- Every route in a web app == value of a type
- Requested path gets converted to value immediately
  - If it can't be converted, send a 404 "not found"
- Render to text at the last minute
- We can introspect on these values
  - Permissions
  - Breadcrumbs
  - Request body limiting
- Compiler prevents us from generating invalid

Typeclasses state requirements

- Simple example: `MonadIO`
- In Yesod:
  - Tells us which messages need to be translated (`RenderMessage`)
  - State Javascript deps (e.g., `YesodJquery`)

Type families

- State a relationship between two types
- In Yesod: type-safe URLs and web app
- Combines nicely with typeclasses

```
class RenderUrl url where
    renderUrl :: url -> Text

type family Route app

runApp :: RenderUrl (Route app) => app -> IO ()
```