Design with Types! (In Haskell)

Andres Löh

2021-10-29 —

Well-Typed
The Haskell Consultants

About me

- Master's degree in Mathematics (University of Konstanz) 2000
- PhD (Utrecht University) 2004 on "Generic Haskell"
- Lecturer at Utrecht University 2007–2010
- Partner at Well-Typed 2010–now

Well-Typed

- Founded 1998 by Duncan Coutts, Ian Lynagh, and Björn Bringert.
- Haskell consulting (development, advice, support, training).
- Currently $\sim$20 people working full-time in Europe, USA, Canada.
- Clients in various countries of the world (most work done remotely).

This talk

- Haskell
- Type system
- Design with types

Haskell

Haskell

- Is a standardized language.
- Designed by committee, actually designed by the community.
- First version 1990.
- Usable, stable version: Haskell 1998.
- Current standard: Haskell 2010 (but many extensions in active use).
- Main implementation: GHC (Glasgow Haskell Compiler) – Simon Peyton Jones at Microsoft Research Cambridge and many contributors, including several people from Well-Typed.

Haskell features

Technical:

- easy to define datatypes
- high abstraction level
- strong type system
- separation of effectful and pure computations
- very versatile

Social:

- large helpful community
- culture of solving problems properly
- open-source (BSD) by default
- vast amount of libraries in central repository (Hackage)

Abstraction

C / Java

```
int total = 0;
for (int i = 0; i < lst.length; i ++) {
  total = total + 3 * lst[i];
}
```

Haskell

```
total = sum (map (3 *) lst)
```

Example taken from Brent Yorgey's UPenn Haskell intro.

Abstraction

C / Java

```
int total = 0;
for (int i = 0; i < lst.length; i ++) {
  total = total + 3 * lst[i];
}
```

Haskell

```
total = sum (map (3 *) lst)
```

Functions such as `sum` or `map` are normal library functions, it's easy to define your own variants.

Example taken from Brent Yorgey's UPenn Haskell intro.

Static types with type inference

- Haskell is statically typed.
- Type errors are reported at compile time.
- Type annotations are mostly optional and can be inferred.
- Support for polymorphism.

Static types with type inference

- Haskell is statically typed.
- Type errors are reported at compile time.
- Type annotations are mostly optional and can be inferred.
- Support for polymorphism.

Example:

```
test (p, x) =
  if
    p x
  then
    x
  else
    0
```

Static types with type inference

- Haskell is statically typed.
- Type errors are reported at compile time.
- Type annotations are mostly optional and can be inferred.
- Support for polymorphism.

Example:

```
test :: (Int -> Bool, Int) -> Int   -- inferred if not specified
test (p, x) =
  if
    p x
  then
    x
  else
    0
```

Effects

C / Java

```
int add0(int x, int y) {
  return x + y;
}
```

Effects

C / Java

```
int add0(int x, int y) {
  return x + y;
}
```

```
int add1(int x, int y) {
  launch_missiles(now);
  return x + y;
}
```

Effects

C / Java

```
int add0(int x, int y) {
  return x + y;
}
```

```
int add1(int x, int y) {
  launch_missiles(now);
  return x + y;
}
```

Both functions have the same type!

Effects

Haskell

```haskell
add0 :: Int -> Int -> Int
add0 x y = x + y
```

Effects

Haskell

```haskell
add0 :: Int -> Int -> Int
add0 x y = x + y
```

```haskell
add1 :: Int -> Int -> IO Int
add1 x y = launch_missiles >> return (x + y)
```

Effects

Haskell

```haskell
add0 :: Int -> Int -> Int
add0 x y = x + y
```

```haskell
add1 :: Int -> Int -> IO Int
add1 x y = launch_missiles >> return (x + y)
```

Effectful computations are tagged by the type system!

Being explicit

By marking the presence of side effects explicitly with `IO`, the **absence** of such a marker guarantees that a piece of code is **definitely free of side effects**.

Effects in Haskell's types

Fine-grained control about effects by choosing the right type:

	`A`	some type, no effect
`IO`	`A`	IO, exceptions, random numbers, concurrency, ...
`Gen`	`A`	random numbers only
`ST s`	`A`	mutable variables only
`STM`	`A`	software transactional memory log variables only
`State s`	`A`	(persistent) state only
`Error`	`A`	exceptions only
`Signal`	`A`	time-changing value

New effect types can be defined. Effects can be combined.

User-defined datatypes

(Record) Types

```
data Point a = MkP {px :: a, py :: a}
```

(Record) Types

```haskell
data Point a = MkP {px :: a, py :: a}
```

```haskell
p1 :: Point Int
p1 = MkP {px = 3, py = 5}
```

(Record) Types

```haskell
data Point a = MkP {px :: a, py :: a}
```

```haskell
p1 :: Point Int
p1 = MkP {px = 3, py = 5}
```

```haskell
p2 = MkP 3 5
```

(Record) Types

```haskell
data Point a = MkP {px :: a, py :: a}
```

```haskell
p1 :: Point Int
p1 = MkP {px = 3, py = 5}
```

```haskell
p2 = MkP 3 5
```

```haskell
p3 :: Point Double
p3 = MkP {px = 2.5, py = 7.3}
```

(Enumeration) Types

```
data Direction = North | West | South | East
data Tetromino = I | O | T | S | Z | J | L
data Weekday   = Mo | Tu | We | Th | Fr | Sa | Su
```

(Enumeration) Types

```
data Direction = North | West | South | East
data Tetromino = I | O | T | S | Z | J | L
data Weekday   = Mo | Tu | We | Th | Fr | Sa | Su
```

```
renderWeekday :: Weekday -> String
renderWeekday wd =
  case wd of
    Mo -> "Monday"
    Tu -> "Tuesday"
    We -> "Wednesday"
    Th -> "Thursday"
    Fr -> "Friday"
    Sa -> "Saturday"
    Su -> "Sunday"
```

Domain-specific Booleans

```
generateReport ::
  (Bool, Bool, InputData) -> Report
```

Domain-specific Booleans

```haskell
generateReport ::
  (Bool, Bool, InputData) -> Report
```

```haskell
data Logging   = EnableLogging | DisableLogging
data Verbosity = IncludeExplanations | Regular
```

```haskell
generateReport ::
  (Logging, Verbosity, InputData) -> Report
```

Domain-specific “Booleans”

```
data Logging   = EnableLogging | DisableLogging
data Debug     = DebugOff | DebugOn
data Verbosity = IncludeExplanations | Regular
```

```
generateReport ::
  (Logging, Debug, Verbosity, InputData) -> Report
```

Domain-specific “Booleans”

```haskell
data Logging   = EnableLogging | DisableLogging
data Debug     = DebugOff | DebugOn
data Verbosity = IncludeExplanations | Regular
```

```haskell
generateReport ::
  (Logging, Debug, Verbosity, InputData) -> Report
```

What about this combination?

```haskell
DebugOn
DisableLogging
```

Domain-specific “Booleans”

```
data LogLevel   = None | Normal | Debug

data Verbosity = IncludeExplanations | Regular
```

```
generateReport ::
  (LogLevel, Verbosity, InputData) -> Report
```

Optionality

```haskell
data Talk =
  MkTalk
    { title    :: String
    , speaker  :: String
    , abstract :: String
    , duration :: Int
    }
```

Well-Typed

Optionality

```haskell
data Talk =
  MkTalk
    { title    :: String
    , speaker  :: String
    , abstract :: String
    , duration :: Int
    }
```

Rule

"For each talk, we need to know the title, the name of the speaker, and optionally an abstract and an estimated duration in minutes."

Optionality

```
data Talk =
  MkTalk
    { title    :: String
    , speaker  :: String
    , abstract :: String
    , duration :: Int
    }
```

Rule

"For each talk, we need to know the title, the name of the speaker, and **optionally** an abstract and an estimated duration in minutes."

Expressing optionality

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

Expressing optionality

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

Values of type `Int` :

```

```

```
...
-3
-2
-1
0
1
2
3
...
```

Expressing optionality

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

Values of type `Maybe Int` :

```
Nothing
```

```
...
Just (-3)
Just (-2)
Just (-1)
Just 0
Just 1
Just 2
Just 3
...
```

Applying optionality

```
data Talk =
  MkTalk
    { title    :: String
    , speaker  :: String
    , abstract :: String
    , duration :: Int
    }
```

Applying optionality

```haskell
data Talk =
  MkTalk
    { title    :: String
    , speaker  :: String
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

Being explicit

By marking the presence of optionality with `Maybe`, the **absence** of such a marker guarantees that a value is **definitely there**.

`Maybe` versus `null`

In languages with `null`,

- nearly everything can be `null`,
- we can never be certain something is not `null`,
- is is easy to forget to check for `null`.

`Maybe` versus `null`

In languages with `null`,

- nearly everything can be `null`,
- we can never be certain something is not `null`,
- is is easy to forget to check for `null`.

By using `Maybe`,

- we know for certain whether a value is optional or not,
- the type system forces us to handle the `Nothing` case,
- we do not have to worry about `Nothing` for non-optional values.

Allowing multiple occurrences

```
data Talk =
  MkTalk
    { title    :: String
    , speaker  :: String
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

What if we want to allow multiple speakers?

Allowing multiple occurrences

```
data Talk =
  MkTalk
    { title    :: String
    , speakers :: List String
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

Allowing multiple occurrences

```haskell
data Talk =
  MkTalk
    { title    :: String
    , speakers :: [String]  -- special syntax for lists
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

Allowing multiple occurrences

```
data Talk =
  MkTalk
    { title    :: String
    , speakers :: [String]  -- special syntax for lists
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

Lists can have arbitrarily many elements.

Allowing multiple occurrences

```
data Talk =
  MkTalk
    { title    :: String
    , speakers :: [String]  -- special syntax for lists
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

Lists can have arbitrarily many elements.

Examples:

```
[]
["Andres"]
["Edsko", "Thomas", "Adam"]
```

Do we really want to allow talks with 0 speakers?

Allowing multiple occurrences

```haskell
data Talk =
  MkTalk
    { title          :: String
    , primarySpeaker :: String
    , otherSpeakers  :: [String]
    , abstract       :: Maybe String
    , duration       :: Maybe Int
    }
```

Allowing multiple occurrences

```
data Talk =
  MkTalk
    { title    :: String
    , speakers :: NonEmptyList String
    , abstract :: Maybe String
    , duration :: Maybe Int
    }
```

```
data NonEmptyList a =
  MkNonEmptyList
    { first  :: a
    , others :: [a]
    }
```

Well-Typed

More concepts: a choice between two types

```haskell
data Either a b =
     Left  a
   | Right b
```

A value of type `Either Int String` is

- either an `Int` (tagged with `Left`)
- or a `String` (tagged with `Right`).

More concepts: one or the other or both

```
data OneOrBoth a b =
    OnlyLeft  a
  | OnlyRight b
  | Both      a b
```

A value of type `OneOrBoth Int String` is

- either just an `Int` (tagged with `OnlyLeft`)
- or just a `String` (tagged with `OnlyRight`)
- or both an `Int` and a `String` (tagged with `Both`).

Intermediate summary

- Datatypes such as `Maybe`, lists, `NonEmptyList`, `Either` or `OneOrBoth` allow us to be very precise in what is expected.
- Unexpected configurations are not representable.
- These types are not built-in, and therefore new concepts can be added with ease.
- The type language is **compositional**.

Precision?

```
data User =
  MkUser
    { userEmail          :: String
    , isVerified         :: Bool
    , verifiedByPassport :: Bool   -- otherwise driver's license
    , idDocumentNumber   :: String
    }
```

Well-Typed

Precision?

```haskell
data User =
  MkUser
    { userEmail          :: String
    , isVerified         :: Bool
    , verifiedByPassport :: Bool   -- otherwise driver's license
    , idDocumentNumber   :: String
    }
```

- Is any string an email address?
- What status can a user really be in?
- Should name and document number have the same type?
- What to put in document number for unverified users?
- Different formats for passport and driver's license numbers.

Introducing an explicit status type

```
data UserStatus =
    Unverified
  | VerifiedByPassport
  | VerifiedByDriversLicense
```

Introducing an explicit status type

```haskell
data UserStatus =
    Unverified
  | VerifiedByPassport
  | VerifiedByDriversLicense

data User =
  MkUser
    { userEmail        :: String
    , userStatus       :: UserStatus
    , idDocumentNumber :: String
    }
```

Well-Typed

Introducing an explicit status type

```haskell
data UserStatus =
    Unverified
  | VerifiedByPassport       String
  | VerifiedByDriversLicense String
```

```haskell
data User =
  MkUser
    { userEmail  :: String
    , userStatus :: UserStatus
    }
```

Introducing an explicit status type

```haskell
data UserStatus =
    Unverified
  | VerifiedByPassport       PassportNumber
  | VerifiedByDriversLicense DriversLicenseNumber
```

```haskell
data User =
  MkUser
    { userEmail  :: String
    , userStatus :: UserStatus
    }
```

```haskell
data PassportNumber       = MkPassportNumber       String
data DriversLicenseNumber = MkDriversLicenseNumber String
```

Introducing an explicit status type

```
data UserStatus =
    Unverified
  | VerifiedByPassport       PassportNumber
  | VerifiedByDriversLicense DriversLicenseNumber
```

```
data User =
  MkUser
    { userEmail  :: EmailAddress
    , userStatus :: UserStatus
    }
```

```
data PassportNumber       = MkPassportNumber       String
data DriversLicenseNumber = MkDriversLicenseNumber String
data EmailAddress         = MkEmailAddress         String
```

Working with a user status

```
data UserStatus =
    Unverified
  | VerifiedByPassport       PassportNumber
  | VerifiedByDriversLicense DriversLicenseNumber
```

Working with a user status

```
data UserStatus =
    Unverified
  | VerifiedByPassport       PassportNumber
  | VerifiedByDriversLicense DriversLicenseNumber
```

```
someFunction ... =
  ...
  case userStatus of
    Unverified                          -> ...
    VerifiedByPassport passportNumber -> ...
    VerifiedByDriversLicense dlNumber -> ...
```

Well-Typed

Working with a user status

```
data UserStatus =
    Unverified
  | VerifiedByPassport       PassportNumber
  | VerifiedByDriversLicense DriversLicenseNumber
```

```
someFunction ... =
  ...
  case userStatus of
    Unverified                           -> ...
    VerifiedByPassport passportNumber -> ...
    VerifiedByDriversLicense dlNumber -> ...
```

We cannot even access a `PassportNumber` unless we are in the right case!

Distinguishing types with the same representation

```
data PassportNumber        = MkPassportNumber        String
data DriversLicenseNumber  = MkDriversLicenseNumber  String
data EmailAddress          = MkEmailAddress          String
```

Distinguishing types with the same representation

```haskell
data PassportNumber        = MkPassportNumber        String
data DriversLicenseNumber = MkDriversLicenseNumber String
data EmailAddress         = MkEmailAddress         String

data URL                  = MkURL                  String
data SQLQuery             = MkSQLQuery             String
data HTML                 = MkHTML                 String
```

Distinguishing types with the same representation

```
data PassportNumber         = MkPassportNumber         String
data DriversLicenseNumber   = MkDriversLicenseNumber   String
data EmailAddress           = MkEmailAddress           String

data URL                    = MkURL                    String
data SQLQuery               = MkSQLQuery               String
data HTML                   = MkHTML                   String

data UserId                 = MkUserId                 Int
data Age                    = MkAge                    Int
data Quantity               = MkQuantity               Int
data Score                  = MkScore                  Int

data Distance               = MkDistance               Double
data Temperature            = MkTemperature            Double
...
```

Validation

```
validateEmailAddress :: String -> Maybe EmailAddress
validateEmailAddress string =
  if
    matchesEmailRedex string
  then
    Just (MkEmailAddress string)
  else
    Nothing
```

Well-Typed

Validation

```
validateEmailAddress :: String -> Maybe EmailAddress
validateEmailAddress string =
  if
    matchesEmailRedex string
  then
    Just (MkEmailAddress string)
  else
    Nothing
```

We are in control of the interface:

- Make `MkEmailAddress` private.
- Now `validateEmailAddress` is the **only** way to produce a value of type `EmailAddress`.

Effectful two-step verification

```
data PassportNumber = MkPassportNumber String
validatePassportNumber :: String -> Maybe PassportNumber
```

Effectful two-step verification

```haskell
data PassportNumber = MkPassportNumber String

validatePassportNumber :: String -> Maybe PassportNumber
```

```haskell
data VerifiedPassportNumber =
  MkVerifiedPassportNumber
    { passportNumber             :: PassportNumber
    , verificationTransactionId :: TransactionId
    }

passportVerificationService ::
  PassportNumber -> IO (Maybe VerifiedPassportNumber)
```

Witnesses for successful (or unsuccessful) tests

```
function x =
  if
    someTest x
  then
    doThis x
  else
    doThat x
```

Witnesses for successful (or unsuccessful) tests

```
function x =
  if
    someTest x
  then
    doThis x
  else
    doThat x
```

```
someTest :: Item -> Bool
doThis   :: Item -> Result
doThat   :: Item -> Result
```

Witnesses for successful (or unsuccessful) tests

```haskell
function x =
  case
    someTest x
  of
    Just y -> doThis y
    Nothing -> doThat x
```

```haskell
someTest :: Item -> Maybe ValidatedItem
doThis   :: ValidatedItem -> Result
doThat   :: Item -> Result
```

What if the model evolves?

We can always change the types without fear:

- the more precise our types are, the better the compiler errors we will get,
- we can make local changes to the code to fix all the type errors,
- after fixing the errors, there is a good chance the program still passes all tests.

What if the model evolves?

We can always change the types without fear:

- the more precise our types are, the better the compiler errors we will get,
- we can make local changes to the code to fix all the type errors,
- after fixing the errors, there is a good chance the program still passes all tests.

- Refactoring is easy.
- Static types are **good** for rapid prototyping.

Conclusions

- Types are easy to define.
- Types give us a way to exchange programming language terms for **domain-specific terms**.
- We control the interface for new types. They do not support any operations we do not explicitly enable.
- We can represent data models but also business logic by using types.
- When writing programs, types then guide the coding.
- Refactoring is easy.