Краткий экскурс в системы типов или как избежать дезинтеграции

Денис Редозубов, @rufuse

June 22, 2015

Martian Climate Orbiter Distaster

- 1998 - спутник NASA Martian Climate Orbiter сгорел в атмосфере марса из-за программной ошибки.

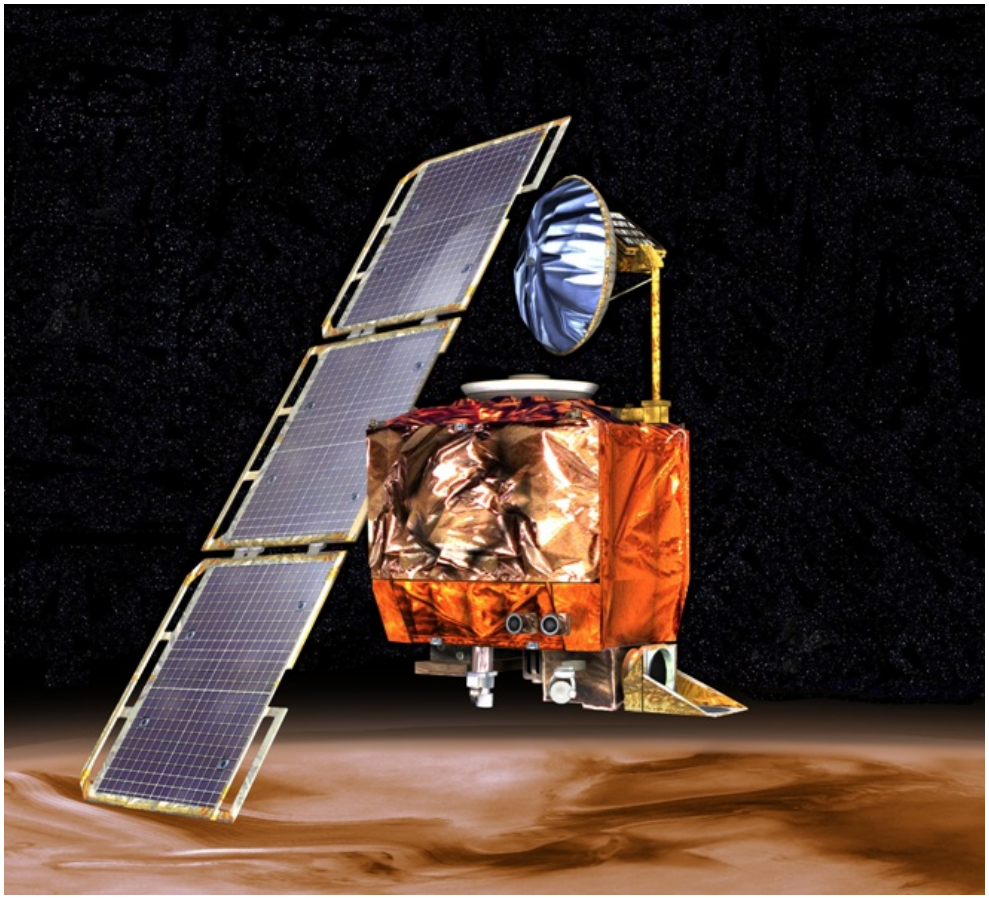


Figure: Martian Climate Orbiter

Martian Climate Orbiter Distaster (2)

Ошибка заключалась в том, что выполнялась арифметика с единицами из разных систем измерения.

> *due to ground-based computer software which produced output in non-SI units of pound-seconds (lbf×s) instead of the metric units of newton-seconds (N×s) specified in the contract between NASA and Lockheed — Wikipedia*

Этого можно было бы избежать, если бы разработчики утилизировали систему типов

Часто приоритеты таковы что

Correctness is dog

Figure: performance is tail

(C) @runarorama

Что такое тип?

В первом приближении - множество применимых вычислимых выражений.

Что такое система типов?

Система типов - гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений. — Бенжамин Пирс, TaPL

Что такое система типов? (2)

Неформально - система мер, направленная на исключение возможности для программы перейти в противоречивое или абсурдное состояние.

Статическая типизация

- Языки программирования, в которых тип каждого выражения может быть определен статически(обычно на одном из этапов компиляции), называются статически типизированными.(Scala, Haskell, Ocaml)

Статическая типизация

- Языки программирования, в которых тип каждого выражения может быть определен статически(обычно на одном из этапов компиляции), называются статически типизированными.(Scala, Haskell, Ocaml)

- Языки, в которых проверка типов на согласованность происходит во время выполнения называются динамически типизируемыми. Более верным термином можно считать "динамически проверяемые", т.к. нет никаких гарантий что типы в программе будут согласованы.(Lisp, Clojure)

Сильная/слабая типизация

- Сильная типизация(strongly typed) - свойство языков, в которых типы всех выражений согласованы(Haskell, Common Lisp)

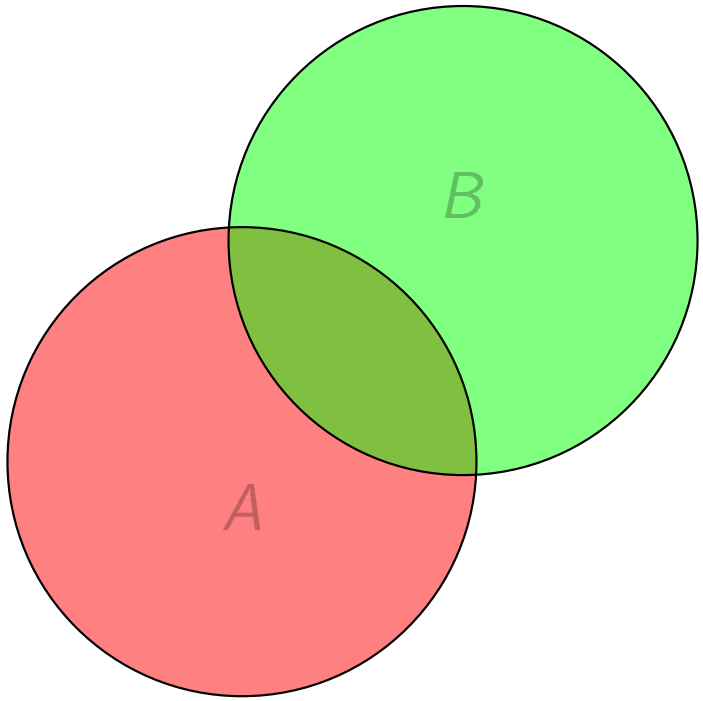
Сильная/слабая типизация

- Сильная типизация(strongly typed) - свойство языков, в которых типы всех выражений согласованы(Haskell, Common Lisp)
- Слабая типизация(weakly typed) - милое название для отсутствия типизации. Отсутствие мер против написания абсурдных программ.

Явность типизации

- Явно типизируемыми называются языки в которых программист должен явно указывать тип каждого выражения(Java, C)
- Неявно типизируемыми называются языки, в которых тип выражения может быть выведений из выражений, его использующих. Неявная типизация требует реализации механизма выведения типов и может быть как полной, так и неполной.(Haskell, Scala)

Многообразие программ



- A - корректно-типизированные(well-typed) программы
- B - работающие программы

Сохранение инвариантов

- Инвариант - свойство выражений оставаться непротиворечивыми в результате изменений.
- Для каждого выражения его тип является инвариантом.
- Цель типизации - сделать неверное состояние невозможным для представлени в программе.

Соответствие Карри-Говарда (1934 - 1969)

- Наблюдение о связи логических доказательств и алгебры вычислений. Это означает, что доказательство в терминах достаточно сильной системы типов является доказательством корректности программы.
- Под "достаточно сильной системой типов" я имею в виду систему типов, способную выразить необходимые инварианты.

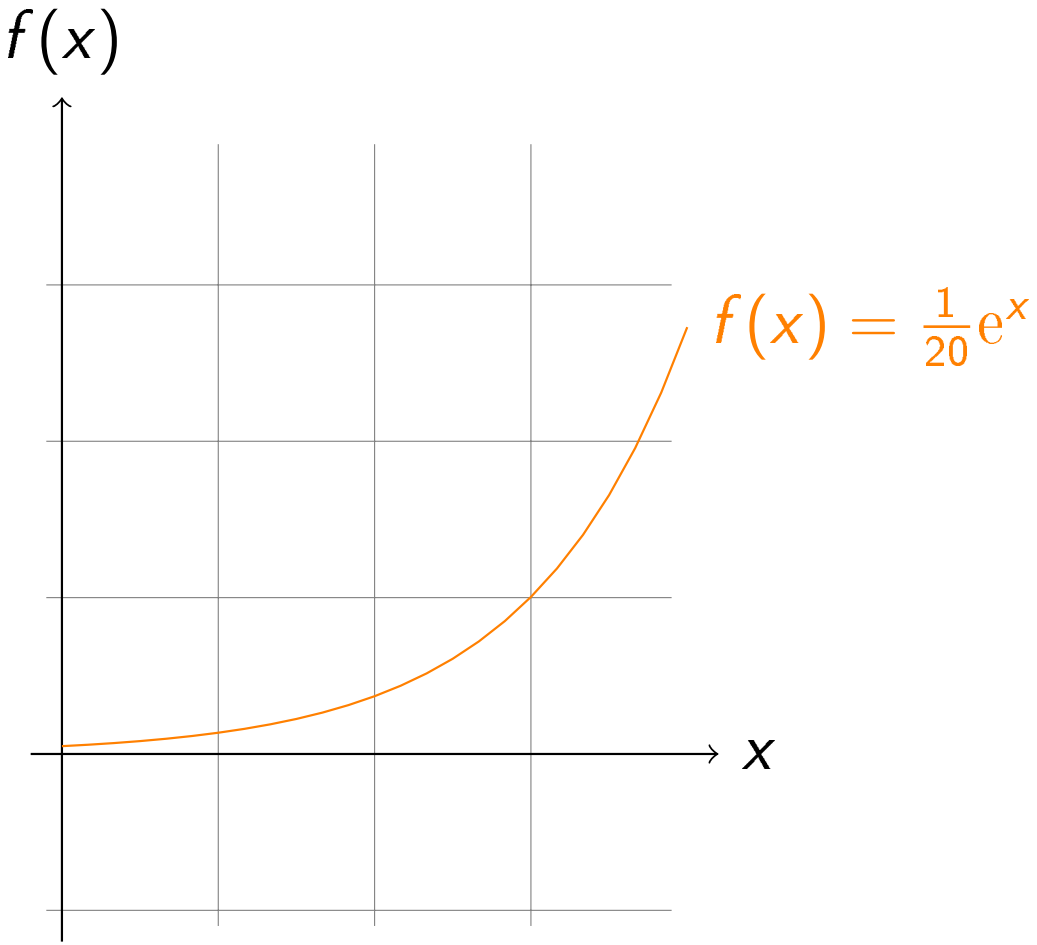
Перенос части проблем из runtime в compile time

- Не требует запуска программы и кучи интеграционных мероприятий для проверки кода
- Быстрее запуска test-suite на проектах, которые больше чем тривиальные

Рефакторинг и поддержка кода

- unsound программа просто не будет компилиться, после несогласованных или неполных изменений.
- Позволяет сразу увидеть все места, где изменения в программе ведут к поломке.
- Позволяет меньше опираться на тесты, которые устаревают, ломаются и требуют постоянной поддержки. Кроме того, тесты не дают никаких формальных гарантий, т.к. за ними нет научных теорий.
- Не требует знать и любить все костыли в огромных системах - все тайное становится явным.

Рефакторинг и поддержка кода (2)



Хорошо типизированная программа является самодокументируемой

- Пример плохо типизированной программы

```
f :: Double
  -> Double
  -> String
  -> Double
```

Хорошо типизированная программа является самодокументируемой

- Пример плохо типизированной программы

```
f :: Double
  -> Double
  -> String
  -> Double
```

- Пример хорошо типизированной программы

```
f :: Commission
  -> Commission
  -> CombinationStrategy
  -> Commission
```

Строгость системы типов напрямую связана с корректностью программ

- Все хотят спокойно спать по ночам
- Бизнес, а иногда и человеческие жизни, зависят от правильности кода

Систему типов не имеет смысла рассматривать без языка программирования, они неразрывно связаны.

- Поэтому будут упомянуты не только свойства систем типов, но и фичи языков программирования

Типы-суммы

- aka Sum-types, Union-types, disjoined union
- Представляют собой непересекающиеся объединения "меток"
- примеры(слева - конструктор типа, справа - конструкторы данных):

```
data Direction = Forward | Backward

data Movement = Step Direction
              | Jump Direction
              | Dash Direction
```

Pattern-Matching

- например

```
move :: Movement
     -> GameState
     -> Character
     -> GameState
move (Step direction) s c = step direction s c
move (Jump direction) s c = jump direction s c
move (Dash direction) s c = dash direction s c
```

Pattern-Matching (2)

- Никто не любит писать много, поэтому записывают так:

```
move :: Movement
     -> GameState
     -> Character
     -> GameState
move (Step d) = step d
move (Jump d) = jump d
move (Dash d) = dash d
```

Типы-пересечения

- aka Product-types
- Пары
- Кортежи
- Рекорды
- Объекты

```
data Color = RGB { red   :: Value
                 , green :: Value
                 , blue  :: Value }
```

Алгебраические типы данных

- aka Algebraic Data Types(ADT)
- Объединяют в себе свойства типов-сумм и типов-пересечений

Лямбда-куб

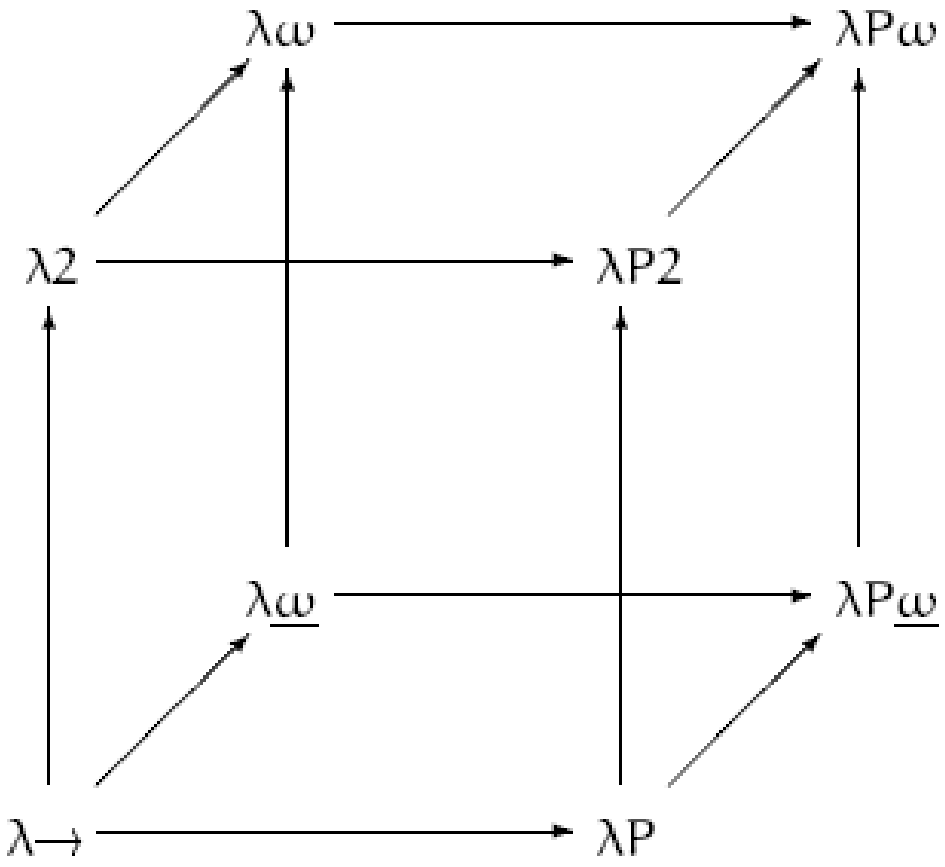


Figure: Lambda Cube

λ - простое типизированное лямбда-исчисление

Параметрический полиморфизм

- Открывает возможность для создания абстрацкий
- Самые банальные примеры - всем известные контейнеры

```
data List a = Nil | Cons a (List a)
```

ad-hoc полиморфизм

Интерфейсы/тайпклассы/трейты

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

instance Eq a => Eq (List a) where
  Nil          == Nil          = True
  Nil          == _            = False
  _            == Nil          = False
  (Cons h1 t1) == (Cons h2 t2) = h1 == h2 && t1 == t2
```

Type functions/operators

Возможность описывать преобразования типов

```
class Add a b where
  type SumTy a b
  add :: a -> b -> SumTy a b

instance Add Float Integer where
  type SumTy Float Integer = Float
  add a b = a + fromInteger b

instance Num a => Add a a where
  type SumTy a a = a
  add = (+)
```

Зависимые типы

Типы, зависимые от термов

```
data Nat     = Z        | S Nat

data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  (::) : a -> Vect k a -> Vect (S k) a

(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

Как сделать, чтобы система типов была помощью и опорой?

- Делаем неверные состояния невозможными для представления в программе
- Выносим на уровень типов то, что имеет смысл для нормальной работы программы
- Если что-то можно статически гарантировать - почему бы это не сделать?

И снова Martian Orbiter Disaster

- Как такие ошибки происходят?

```
data Distance = Distance Decimal

-- Oops! Так можно сложить метры и ярды!
add :: Distance -> Distance -> Distance
add (Distance x) (Distance y) = Distance (x + y)
```

И снова Martian Orbiter Disaster (2)

```
newtype Metric = Metric { fromMetric :: Decimal }
newtype Imperial = Imperial { fromImperial :: Decimal }
data Distance a = Distance a

class Unit a where
  fromValue :: a -> Decimal
  toValue   :: Decimal -> a
instance Unit Metric where ...
instance Unit Imperial where ...

add :: Unit a => Distance a -> Distance a -> Distance a
add (Distance x) (Distance y) = Distance wrapped
  where wrapped = toValue added
        added   = fromValue x + fromValue y
```

Требования добавляются

- Необходимо представить разные единицы измерения в разных системах измерения
- Необходимо уметь складывать единицы измерения в пределах каждой из систем, в результате сложения размерностью становится большая единица
- Можно складывать дистанцию только в одинаковой системе измерения

И снова Martian Orbiter Disaster (3)

```
data Metric = Meter Decimal | Kilometer Decimal

addMetric :: Metric -> Metric -> Metric
addMetric = ...

data US = Inch Decimal | Yard Decimal

addUS :: US -> US -> US
addUS = ...

data Distance = MetricDistance Metric | USDistance US
-- ... см. 4.2
```

И снова Martian Orbiter Disaster (3.2)

```
safeAddDistance :: Distance
                -> Distance
                -> Maybe Distance
safeAddDistance (MetricDistance x) (MetricDistance y) =
   Just (MetricDistance (addMetric x y))
safeAddDistance (USDistance x)     (USDistance y)     =
   Just (USDistance (addUS x y))
safeAddDistance _                  _                  =
   Nothing
```

И снова Martian Orbiter Disaster (4)

```
{-# LANGUAGE DataKinds, GADTs, KindSignatures #-}

-- Unit code skipped

data Measurement = MetricUnit | USUnit

data Distance :: Measurement -> * where
  MetricDistance :: Metric -> Distance 'MetricUnit
  USDistance     :: US     -> Distance 'USUnit

addDistance :: Distance a -> Distance a -> Distance a
addDistance (MetricDistance x) (MetricDistance y) = ...
addDistance (USDistance x)     (USDistance y)     = ...
```

Average

```
import Data.Foldable (sum)

average :: Fractional a => [a] -> a
average xs = sum xs / fromIntegral (length xs)
```

Обобщенный Average

average можно обобщить!

```
import Data.Foldable (sum, toList)

average :: (Fractional a, Foldable t) => t a -> a
average xs = sum xs / len
  where len = fromIntegral . length . toList $ xs
```

Конкретизированный Average

Предыдущие реализации average вернут 0 от пустого листа. Что если это неприемлимо? Что если пустой лист уже является ошибкой?

```
import Data.Foldable (sum)
import qualified Data.List.NonEmpty as NE

-- data NonEmpty a = a :| [a]

average :: Fractional a => NE.NonEmpty a -> a
average xs = sum xs / NE.length xs
```

Функции на типах (1)

```
data family Array a
data instance Array Int        =
  IntArr UnboxedIntArr
data instance Array Bool       =
  BoolArr UnboxedBitVector
data instance Array (a,b)      =
  PairArr (Array a) (Array b)
data instance Array (Array a) =
  ArrArr (Array Int) (Array a)
```

Функции на типах (2)

Servant

```
type HackageAPI =
         "users" :> Get '[JSON] [UserSummary]
    :<|> "user" :> Capture "username" Username
                :> Get '[JSON] UserDetailed
    :<|> "packages" :> Get '[JSON] [Package]
```

Функции на типах + TH (acid-state)

```
type Key = String

type Value = String

data KeyValue = KeyValue !(Map.Map Key Value) deriving
```

Функции на типах + TH (acid-state) 2

```
insertKey :: Key -> Value -> Update KeyValue ()
insertKey key value = do
  KeyValue m <- get
  put (KeyValue (Map.insert key value m))

lookupKey :: Key -> Query KeyValue (Maybe Value)
lookupKey key = do
  KeyValue m <- ask
  return (Map.lookup key m)

$(makeAcidic ''KeyValue ['insertKey])
```

Функции на типах + TH (acid-state) 2

```
main :: IO ()
main = do
  ...
  update acid (InsertKey key val)
  ...
```

Есть ли минусы?

- Вы все равно сможете писать отстойный код, даже с хорошей системой типов. Градус остойности, вероятно, будет ниже, но тем не менее.

Есть ли минусы?

- Вы все равно сможете писать отстойный код, даже с хорошей системой типов. Градус остойности, вероятно, будет ниже, но тем не менее.
- Не стоит думать что наиболее мощная система типов подойдет для вашей задачи. Coq или Agda запросто могут быть оверкиллом.

Какая система типов нужна вам?

Главный фактор - цена доказательства.
Что стоит больше для вас/компании: отвергнутая компилятором корректная программа или принятая неверная?

Куда копать дальше

- "Types and Programming Languages" и "Advanced Topics in Types and Programming Languages" - Benjamin C. Pierce
- "Homotopy Type Theory" - Univalent Foundations Program

Контакты

- twitter: @rufuse
- email: denis.redozubov@gmail.com
- `http://denisredozubov.com`
- Ссылка на слайды `https://www.dropbox.com/s/37l2pa2yek9qssm/TS-ruhaskell.ru.pdf?dl=0`