

Introduction to Dependent Types

Eagan Technology Unconference

Joseph Ching

September 11, 2015

Table of Contents

1 Preface

Table of Contents

1 Preface

2 Review of Basics

Table of Contents

- 1 Preface
- 2 Review of Basics
- 3 What is Dependent Type

Table of Contents

- 1 Preface
- 2 Review of Basics
- 3 What is Dependent Type
- 4 Steps toward Dependent Types

Table of Contents

- 1 Preface
- 2 Review of Basics
- 3 What is Dependent Type
- 4 Steps toward Dependent Types
- 5 Closing

Section Outline

1 Preface

Quick Question

How many are familiar with this topic?

A Joke

This is not a $\mathsf{m-}$ tutorial,

A Joke

This is not a \mathbb{M} -tutorial, and nothing here will involve burritos.

About This Talk

Example languages with dependent types:

About This Talk

Example languages with dependent types:

- Idris

About This Talk

Example languages with dependent types:

- Idris
- Epigram

About This Talk

Example languages with dependent types:

- Idris
- Epigram
- Agda

About This Talk

Example languages with dependent types:

- Idris
- Epigram
- Agda
- Coq

About This Talk

Example languages with dependent types:

- Idris
- Epigram
- Agda
- Coq

But we will be using Haskell though.

Honestly, it's because they're way over his head...

Disclaimer

There will be many code examples with *very* loose translations to imperative/OOP as we go along. Though please keep in mind that these are merely made up syntactical translations, the actual concepts may differ vastly.

Section Outline

- 2 Review of Basics
 - Values and Types
 - Defining Data Types
 - Functions

Values and Types

Values has Types, or Values are classified by Types.

..., -1, 0, 1, 2, 3, ... :: Int

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
```

```
True, False :: Bool
```

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
```

```
True, False :: Bool
```

```
'a', 'b', 'c' :: Char
```

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
```

```
True, False :: Bool
```

```
'a', 'b', 'c' :: Char
```

```
"abc" :: String ~ [Char]
```

Values are also called Terms

About Data Types

How are data types defined?

About Data Types

How are data types defined?

- Some are built in magic: `Int`, `Char`, function arrow

About Data Types

How are data types defined?

- Some are built in magic: `Int`, `Char`, function arrow
- Some are built in sugar: list, tuples
 - We can define equivalent non-sugar version ourselves

About Data Types

How are data types defined?

- Some are built in magic: `Int`, `Char`, function arrow
- Some are built in sugar: list, tuples
 - We can define equivalent non-sugar version ourselves
- Rest can be user defined: `Bool`, `String`, `Maybe`

About Data Types

What are the data types like?

About Data Types

What are the data types like?

- Multiple **Value** constructors

About Data Types

What are the data types like?

- Multiple **Value** constructors
- Parametrize over another **Type**

About Data Types

What are the data types like?

- Multiple **Value** constructors
- Parametrize over another **Type**
- Recursive definition

About Data Types

What are the data types like?

- Multiple **Value** constructors
- Parametrize over another **Type**
- Recursive definition
- Synonyms of other **Types**

About Data Types

What are the data types like?

- Multiple **Value** constructors
- Parametrize over another **Type**
- Recursive definition
- Synonyms of other **Types**
- Combination of the above

Defining Data Types

Define new data type with `data`.

Defining Data Types

Define new data type with `data`.

- Left hand side (LHS) - `Type` constructor
- Right hand side (RHS) - `Value` constructor

Defining Data Types

Define new data type with `data`.

- Left hand side (LHS) - `Type` constructor
- Right hand side (RHS) - `Value` constructor

`Type` and `Value` constructors are capticalized.

Our First Example!

Define a person:

```
-- | params for firstname, lastname, age respectively
data Person = Person String String Int

barbara :: Person
barbara = Person "Barbara" "Smith" 30
```

Our First Example!

Define a person:

```
-- | params for firstname, lastname, age respectively  
data Person = Person String String Int
```

```
barbara :: Person
```

```
barbara = Person "Barbara" "Smith" 30
```

The **Type** of the **Person Value** constructor:

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

Our First Example!

Define a person:

```
-- | params for firstname, lastname, age respectively
data Person = Person String String Int

barbara :: Person
barbara = Person "Barbara" "Smith" 30
```

The **Type** of the **Person Value** constructor:

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

A loose translation:

```
enum Person {
  Person(String firstname, String lastname, Int age)
}

Person barbara = new Person("Barbara", "Smith", 30)
```

Multiple Value Constructors

Data can have multiple **Value** constructors:

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

```
data Weekdays = Sunday | Monday | Tuesday | Wednesday  
               | Thursday | Friday | Saturday
```

Multiple Value Constructors

Data can have multiple **Value** constructors:

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

```
data Weekdays = Sunday | Monday | Tuesday | Wednesday  
               | Thursday | Friday | Saturday
```

A loose translation:

```
enum Bool { False, True }
```

```
enum Weekdays {  
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,  
    Saturday  
}
```


Multiple Value Constructor

You can do type aliasing with `type`:

```
type Side = Double  
type Radius = Double
```

Multiple Value Constructor

You can do type aliasing with `type`:

```
type Side = Double  
type Radius = Double
```

For example:

```
data Shape = Triangle Side Side Side  
           | Rectangle Side Side  
           | Circle Radius
```

Multiple Value Constructor

You can do type aliasing with `type`:

```
type Side = Double
type Radius = Double
```

For example:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

A loose translation:

```
enum Shape {
  Triangle(Double side1, Double side2, Double side3),
  Rectangle(Double length, Double width),
  Circle(Double radius)
}
```

Multiple Value Constructor

Recall $\text{Side} \sim \text{Radius} \sim \text{Double}$:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Multiple Value Constructor

Recall `Side ~ Radius ~ Double`:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Types of the 3 `Value` constructors:

```
Triangle  :: Side -> Side -> Side -> Shape
Rectangle :: Side -> Side -> Shape
Circle    :: Radius -> Shape
```

Multiple Value Constructor

Recall `Side` \sim `Radius` \sim `Double`:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Types of the 3 `Value` constructors:

```
Triangle  :: Side -> Side -> Side -> Shape
Rectangle :: Side -> Side -> Shape
Circle    :: Radius -> Shape
```

Example `Shapes`:

```
myTri, myRect, myCir :: Shape
myTri  = Triangle 2.1 3.2 5
myRect = Rectangle 4 4
myCir  = Circle 7.2
```

Parametrization

Types can parametrize over another type:

```
data Identity a = Identity a
```

```
intIdwrtSum :: Identity Int
```

```
intIdwrtSum = Identity 0
```

Parametrization

Types can parametrize over another type:

```
data Identity a = Identity a
```

```
intIdwrtSum :: Identity Int
```

```
intIdwrtSum = Identity 0
```

With:

```
Identity :: a -> Identity a
```


Parametrization

Types can parametrize over another type:

```
data Identity a = Identity a
```

```
intIdwrtSum :: Identity Int
```

```
intIdwrtSum = Identity 0
```

With:

```
Identity :: a -> Identity a
```

A loose translation:

```
enum Identity<A> {  
  Identity(A a)  
}
```

Tuple

Parametrize over 2 types - 2-tuple!

```
data Tuple a b = Tuple a b
```

Tuple

Parametrize over 2 types - 2-tuple!

```
data Tuple a b = Tuple a b
```

With:

```
Tuple :: a -> b -> Tuple a b
```

Tuple

Parametrize over 2 types - 2-tuple!

```
data Tuple a b = Tuple a b
```

With:

```
Tuple :: a -> b -> Tuple a b
```

A loose translation:

```
enum Tuple<A, B> {  
  Tuple(A a, B b)  
}
```

Tuple

Actual built-in sugar:

```
data Tuple a b = Tuple a b
=> data (,) a b = (,) a b
=> data (a, b) = (a, b)
```

Tuple

Actual built-in sugar:

```
data Tuple a b = Tuple a b
=> data (,) a b = (,) a b
=> data (a, b) = (a, b)
```

An example:

```
type Employed = Bool

barbara, chet, luffy :: (Person, Employed)
barbara = (Person "Barbara" "Smith" 30, True)
chet    = (Person "Chet" "Awesome-Laser" 2, False)
luffy   = (Person "Luffy D." "Monkey" 19, False)
```

Maybe

Like `Bool`, but parametrizes a `Type a` over the `True` part:

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

Maybe

Like `Bool`, but parametrizes a `Type a` over the `True` part:

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

With:

```
Nothing :: Maybe a  
Just    :: a -> Maybe a
```


Maybe

Like `Bool`, but parametrizes a `Type a` over the `True` part:

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

With:

```
Nothing :: Maybe a  
Just    :: a -> Maybe a
```

A loose translation:

```
enum Maybe<A> {  
  Nothing,  
  Just(A a)  
}
```

Maybe

From previous slide:

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

Maybe

From previous slide:

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

Say more with **Occupation**:

```
type Occupation = Maybe String
```

```
barbara2, chet2, luffy2 :: (Person, Occupation)  
barbara2 = (Person "Barbara" "Smith" 30, Just "dancer")  
chet2    = (Person "Chet" "Awesome-Laser" 2, Nothing)  
luffy2   = (Person "Luffy D." "Monkey" 19, Just "pirate"  
            )
```

Either

Like `Bool`, but parametrizes over both `True` and `False`:

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

Either

Like `Bool`, but parametrizes over both `True` and `False`:

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

With:

```
Left  :: a -> Either a b  
Right :: b -> Either a b
```

Either

Like `Bool`, but parametrizes over both `True` and `False`:

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

With:

```
Left  :: a -> Either a b  
Right :: b -> Either a b
```

A loose translation:

```
enum Either<A, B> {  
  Left(A a),  
  Right(B b)  
}
```

Either

From previous slide:

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

Either

From previous slide:

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

Refine with **Earning**:

```
type Earning = Either String Int
```

```
barbara3, chet3, luffy3 :: (Person, Earning)
barbara3 = (Person "Barbara" "Smith" 30,
            Right 100000)
chet3    = (Person "Chet" "Awesome-Laser" 2,
            Left "Is a baby")
luffy3   = (Person "Luffy D." "Monkey" 19,
            Right 2000000)
```


Types with Recursion

Natural number:

```
data Nat = Z | S Nat
```

With:

```
Z :: Nat  
S :: Nat -> Nat
```

Types with Recursion

Natural number:

```
data Nat = Z | S Nat
```

With:

```
Z :: Nat  
S :: Nat -> Nat
```

A loose translation:

```
enum Nat {  
  Z,  
  S(Nat n)  
}
```

Types with Recursion

Natural number:

```
data Nat = Z | S Nat
```

```
Z :: Nat
```

```
S :: Nat -> Nat
```

```
0 ~ Z
```

```
1 ~ S Z
```

```
2 ~ S (S Z)
```

```
3 ~ S (S (S Z))
```

Types with Recursion

List - recursive **Type** that parametrizes over another **Type**:

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

With:

```
Nil    :: List a  
Cons   :: a -> List a -> List a
```

Types with Recursion

List - recursive **Type** that parametrizes over another **Type**:

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

With:

```
Nil    :: List a  
Cons   :: a -> List a -> List a
```

A loose translation:

```
enum List<A> {  
  Nil,  
  Cons(A a, List<A> as)  
}
```

Types with Recursion

Actual built-in sugar is something like:

```
data List a = Nil | Cons a (List a)
=> data [] a = [] | (:) a ([] a)
=> data [a] = [] | (:) a [a]
```

Types with Recursion

Actual built-in sugar is something like:

```
data List a = Nil | Cons a (List a)
=> data [] a = [] | (:) a ([] a)
=> data [a] = [] | (:) a [a]
```

Sugar that `List`:

```
ints :: List Int
ints = Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))

-- built-in sugar
ints :: [] Int
ints = 1 : 2 : 3 : 4 : []

-- 2x the sugar!
ints :: [Int]
ints = [1, 2, 3, 4]
```

Functions

Maps **Values** of a **Type** to **Values** of another **Type**:

```
even :: Int -> Bool
even 0 = True
even n = if rem n 2 == 0
         then True
         else False
```


Functions

Maps **Values** of a **Type** to **Values** of another **Type**:

```
even :: Int -> Bool
even 0 = True
even n = if rem n 2 == 0
         then True
         else False
```

Not as loose translation:

```
Bool even(Int n) {
  switch n:
    case n == 0:
      return True;
  default:
    if rem(n, 2) == 0
      return True;
    else
      return False;
}
```

Functions with Recursion

Use recursion for recursive **Types**:

```
data Nat = Z | S Nat
```

```
toInt :: Nat -> Int
```

```
toInt Z = 0
```

```
toInt (S n) = 1 + toInt n
```

Functions with Recursion

Use recursion for recursive **Types**:

```
data Nat = Z | S Nat
```

```
toInt :: Nat -> Int
```

```
toInt Z = 0
```

```
toInt (S n) = 1 + toInt n
```

Not as loose translation:

```
Int toInt(Nat n) {  
  switch n:  
  case Z:  
    return 0;  
  case (S m): -- n ~ (S m)  
    return 1 + toInt(m);  
}
```

Functions with Recursion

Use recursion for recursive **Types**:

```
data Nat = Z | S Nat
```

```
toInt :: Nat -> Int
```

```
toInt Z = 0
```

```
toInt (S n) = 1 + toInt n
```

Evaluation is a series of substitutions:

```
three = S (S (S Z)) :: Nat
```

```
toInt three :: Int
```

```
= toInt (S (S (S Z)))
```

```
= 1 + toInt (S (S Z))
```

```
= 1 + 1 + toInt (S Z)
```

```
= 1 + 1 + 1 + toInt Z
```

```
= 1 + 1 + 1 + 0
```

```
= 1 + 1 + 1
```

```
= 1 + 2
```

```
= 3
```

Functions with Parametric Polymorphism

Functions can be parametric:

```
id :: a -> a  
id x = x
```

Functions with Parametric Polymorphism

Functions can be parametric:

```
id :: a -> a  
id x = x
```

Not as loose translation:

```
A id<A>(A a) {  
  return a;  
}
```

Functions with Parametric Polymorphism

Functions can be parametric:

```
data [a] = [] | (:) a [a]
```

```
append :: [a] -> [a] -> [a]
```

```
append [] ys = ys
```

```
append (x:xs) ys = x : append xs ys
```

Functions with Parametric Polymorphism

Functions can be parametric:

```
data [a] = [] | (:) a [a]
```

```
append :: [a] -> [a] -> [a]
```

```
append [] ys = ys
```

```
append (x:xs) ys = x : append xs ys
```

A translation:

```
List<A> append(List<A> l1, List<A> l2) {  
  switch l1:  
    case Nil:  
      return l2;  
    case Cons(x, xs):  
      List<A> rest = append(xs, l2);  
      return Cons(x, rest);  
}
```


Functions with Parametric Polymorphism

Functions can be parametric:

```
data [a] = [] | (:) a [a]

append :: [a] -> [a] -> [a]
append []      ys = ys
append (x:xs)  ys = x : append xs ys
```

Evaluation is a series of substitutions:

```
xs = [4, 8] = 4 : 8 : [] :: [Int]
ys = [15, 16, 23, 42] = 15 : 16 : 23 : 42 : [] :: [Int]

append xs ys :: [Int]
= append [4, 8] [15, 16, 23, 42]
= 4 : append [8] [15, 16, 23, 42]
= 4 : 8 : append [] [15, 16, 23, 42]
= 4 : 8 : [15, 16, 23, 42]
= 4 : [8, 15, 16, 23, 42]
= [4, 8, 15, 16, 23, 42]
```

Higher-order Functions

Functions that take functions as params:

```
-- actual name is ($)
apply :: (a -> b) -> a -> b
apply f x = f x

-- actual name is (.)
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

Higher-order Functions

Functions that take functions as params:

```
-- actual name is ($)
apply :: (a -> b) -> a -> b
apply f x = f x

-- actual name is (.)
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

Yay translations:

```
B apply(Func<A, B> f, A a) {
  return f(a);
}

Func<A, C> compose(Func<B, C> f, Func<A, B> g) {
  return x => f(g(x));
}
```

More Functions Examples

map:

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs)      = f x : map f xs
```

More Functions Examples

map:

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs) = f x : map f xs
```

A translation:

```
List<B> map(Func<A, B> f, List<A> la) {
  switch la:
  case Nil:
    return Nil;
  case Cons(a, as):
    B b = f(a)
    List<B> rest = map(f, as);
    return Cons(b, rest);
}
```

More Functions Examples

map:

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

Evaluation is a series of substitutions:

```
xs = [4, 8, 15, 16, 23, 42] :: [Int]
even :: Int -> Bool
```

```
map even xs :: [Bool]
= map even [4, 8, 15, 16, 23, 42]
= even 4 : map even [8, 15, 16, 23, 42]
= True : even 8 : map even [15, 16, 23, 42]
= True : True : even 15 : map even [16, 23, 42]
= True : True : False : even 16 : map even [23, 42]
= True : True : False : True : even 23 : map even [42]
= True : True : False : True : False : even 24 : map even
  []
= True : True : False : True : False : True : []
= [True, True, False, True, False, True]
```

More Functions Examples

zip:

```
zip :: [a] -> [b] -> [(a,b)]  
zip []      ys      = []  
zip xs      []      = []  
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

More Functions Examples

zip:

```
zip :: [a] -> [b] -> [(a,b)]
zip []      ys      = []
zip xs      []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

A translation:

```
List<Tuple<A, B>> zip(List<A> l1, List<B> l2) {
  switch l1:
    case Nil:
      return Nil;
    case Cons(a, as):
      switch l2:
        case Nil:
          return Nil;
        case Cons(b, bs):
          Tuple<A, B> front = Tuple(a, b);
          List<Tuple<A, B>> rest = zip(as, bs);
          return Cons(front, rest);
}
```


More Functions Examples

zip:

```
zip :: [a] -> [b] -> [(a,b)]
zip []      ys      = []
zip xs      []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Evaluation is a series of substitutions:

```
xs = ['a', 'b', 'c'] :: [Char]
ys = [1, 2, 3, 4] :: [Int]

zip xs ys :: [(Char, Int)]
= zip ['a', 'b', 'c'] [1, 2, 3, 4]
= ('a', 1) : zip ['b', 'c'] [2, 3, 4]
= ('a', 1) : ('b', 2) : zip ['c'] [3, 4]
= ('a', 1) : ('b', 2) : ('c', 3) : zip [] [4]
= ('a', 1) : ('b', 2) : ('c', 3) : []
= [('a', 1), ('b', 2), ('c', 3)]
```

Section Outline

- 3 What is Dependent Type
 - λ -Calculus
 - Extensions on λ -calculus

λ -Calculus

So far, we have seen:

λ -Calculus

So far, we have seen:

- function application

λ -Calculus

So far, we have seen:

- function application
- function abstraction (aka higher-order functions)

λ -Calculus

So far, we have seen:

- function application
- function abstraction (aka higher-order functions)
- variable binding

λ -Calculus

So far, we have seen:

- function application
- function abstraction (aka higher-order functions)
- variable binding
- substitution

λ -Calculus

So far, we have seen:

- function application
- function abstraction (aka higher-order functions)
- variable binding
- substitution

\Rightarrow basis for simply typed λ -calculus.

λ -Calculus

Q: Sure, but can we have more?

λ -Calculus

Q: Sure, but can we have more?

A: Yes, extend λ -calculus so we can have more forms of abstractions.

λ -Calculus

Q: Sure, but can we have more?

A: Yes, extend λ -calculus so we can have more forms of abstractions.

Q: But how?

λ -Calculus

Q: Sure, but can we have more?

A: Yes, extend λ -calculus so we can have more forms of abstractions.

Q: But how?

A: What if I tell you...

λ -Calculus

Q: Sure, but can we have more?

A: Yes, extend λ -calculus so we can have more forms of abstractions.

Q: But how?

A: What if I tell you...

...you should already be familiar with 2 axes of extension :)

Subtype Polymorphism

Given data types T and P , if there is a relation between T and P by some notion of substitutability with T in place of P , then we say T is a subtype of the supertype P , denoted by $T <: P$.

Subtype Polymorphism

Given data types T and P , if there is a relation between T and P by some notion of substitutability with T in place of P , then we say T is a subtype of the supertype P , denoted by $T <: P$.

There is an extension on λ -calculus with subtype polymorphism and is denoted by $\lambda_{<:}$.

Subtype Polymorphism

Given data types T and P , if there is a relation between T and P by some notion of substitutability with T in place of P , then we say T is a subtype of the supertype P , denoted by $T <: P$.

There is an extension on λ -calculus with subtype polymorphism and is denoted by $\lambda_{<:}$.

=> Object Oriented Programming

Subtype Polymorphism

Given data types T and P , if there is a relation between T and P by some notion of substitutability with T in place of P , then we say T is a subtype of the supertype P , denoted by $T <: P$.

There is an extension on λ -calculus with subtype polymorphism and is denoted by $\lambda_{<:}$.

=> Object Oriented Programming

Though this is not an axis that we are interested in.

Parametric Polymorphism

Introduce a mechanism of universal quantification over **Types**: **Types** can abstract over **Types**, allows for generic data types and generic functions.

Parametric Polymorphism

Introduce a mechanism of universal quantification over **Types**: **Types** can abstract over **Types**, allows for generic data types and generic functions.

=> Generic Programming

Parametric Polymorphism

Introduce a mechanism of universal quantification over
Types: **Types** can abstract over **Types**, allows for generic data
types and generic functions.

=> Generic Programming

Recall:

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)

(.) :: (b -> c) -> (a -> b) -> (a -> c)
map :: (a -> b) -> [a] -> [b]
```

Parametric Polymorphism

Introduce a mechanism of universal quantification over **Types**: **Types** can abstract over **Types**, allows for generic data types and generic functions.

=> Generic Programming

Recall:

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)

(.) :: (b -> c) -> (a -> b) -> (a -> c)
map :: (a -> b) -> [a] -> [b]
```

The name for this extension is formally second order λ -calculus, aka System F, denoted by $\lambda 2$,

Value and Type Interdependency

Re-thinking functions:

```
even :: Int -> Bool
even 0 = True
even n = if rem n 2 == 0
         then True
         else False
```

even maps **Ints** to **True** and **False**.

Value and Type Interdependency

Re-thinking functions:

```
even :: Int -> Bool
even 0 = True
even n = if rem n 2 == 0
         then True
         else False
```

even maps **Ints** to **True** and **False**.

=> **Values** on RHS depends on the **Values** on LHS

Value and Type Interdependency

Re-thinking functions:

```
even :: Int -> Bool
even 0 = True
even n = if rem n 2 == 0
         then True
         else False
```

even maps **Ints** to **True** and **False**.

=> **Values** on RHS depends on the **Values** on LHS

=> **Values** depending on **Values**

Value and Type Interdependency

Re-thinking parametrized data types:

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

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

Value and Type Interdependency

Re-thinking parametrized data types:

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

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

Maybe and List take a Type and return Value constructors

Value and Type Interdependency

Re-thinking parametrized data types:

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

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

Maybe and List take a Type and return Value constructors

=> Values on RHS depends on the Type on LHS

Value and Type Interdependency

Re-thinking parametrized data types:

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

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

Maybe and List take a Type and return Value constructors

=> Values on RHS depends on the Type on LHS

=> Values depending on Types

Value and Type Interdependency

Re-thinking parametrized data types:

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

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

`Maybe` and `List` take a `Type` and return `Value` constructors

=> `Values` on RHS depends on the `Type` on LHS

=> `Values` depending on `Types`

=> Parametric polymorphism of $\lambda 2$ again

Value and Type Interdependency

Then what about the other cases of dependencies?

Value and Type Interdependency

Then what about the other cases of dependencies?

- **Values** depending on **Values**: λ -calculus

Value and Type Interdependency

Then what about the other cases of dependencies?

- **Values** depending on **Values**: λ -calculus
- **Values** depending on **Types**: $\lambda 2$, System F

Value and Type Interdependency

Then what about the other cases of dependencies?

- **Values** depending on **Values**: λ -calculus
- **Values** depending on **Types**: $\lambda 2$, System F
- **Types** depending on **Types**:

Value and Type Interdependency

Then what about the other cases of dependencies?

- **Values** depending on **Values**: λ -calculus
- **Values** depending on **Types**: $\lambda 2$, System F
- **Types** depending on **Types**: λ_{ω}
=> Type-level programming via type operators

Value and Type Interdependency

Then what about the other cases of dependencies?

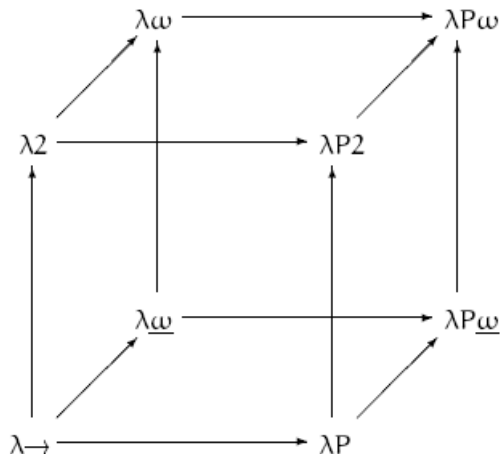
- **Values** depending on **Values**: λ -calculus
- **Values** depending on **Types**: $\lambda 2$, System F
- **Types** depending on **Types**: λ_{ω}
=> Type-level programming via type operators
- **Types** depending on **Values**:

Value and Type Interdependency

Then what about the other cases of dependencies?

- **Values** depending on **Values**: λ -calculus
- **Values** depending on **Types**: $\lambda 2$, System F
- **Types** depending on **Types**: λ_{ω}
=> Type-level programming via type operators
- **Types** depending on **Values**: $\lambda\Pi$
=> Dependent types

Lambda Cube



Dependent Types

Why do we want them?

- more expressive type system
- encode stronger invariants
- proving correctness of code

System F_c

Currently, Haskell as of GHC 7.10.2

- no true type operators

System F_c

Currently, Haskell as of GHC 7.10.2

- no true type operators
- type-level programming through:
 - type families

System F_c

Currently, Haskell as of GHC 7.10.2

- no true type operators
- type-level programming through:
 - type families
 - equalities on **Types**

System F_c

Currently, Haskell as of GHC 7.10.2

- no true type operators
- type-level programming through:
 - type families
 - equalities on **Types**

This axis of extension on $\lambda 2$ is termed System F_c .

System F_c

Currently, Haskell as of GHC 7.10.2

- not truly dependent either:

System F_c

Currently, Haskell as of GHC 7.10.2

- not truly dependent either:
 - strong distinction between **Values** and **Types**

System F_c

Currently, Haskell as of GHC 7.10.2

- not truly dependent either:
 - strong distinction between **Values** and **Types**
- emulate dependent types with:
 - handful of language extensions

System F_c

Currently, Haskell as of GHC 7.10.2

- not truly dependent either:
 - strong distinction between **Values** and **Types**
- emulate dependent types with:
 - handful of language extensions
 - **Kind** system

Teaser

Example please:

```
infixr 5 :>
data Vect (n :: Nat) a where
  VNil :: Vect 0 a
  (:>) :: a -> Vect n a -> Vect (n + 1) a

vs :: Vect 6 Int
vs = 4 :> 8 :> 15 :> 16 :> 23 :> 42 :> VNil
```

Teaser

Example please:

```
infixr 5 :>
data Vect (n :: Nat) a where
  VNil :: Vect 0 a
  (:>) :: a -> Vect n a -> Vect (n + 1) a

vs :: Vect 6 Int
vs = 4 :> 8 :> 15 :> 16 :> 23 :> 42 :> VNil
```

Translation* please:

```
enum Vect<Nat n, A> {
  Vect<0, A> VNil,
  Vect<n + 1, A> VCons(A a, Vect<n, A> va)
}

Vect<6, Int> vs = VCons(4, VCons(8, VCons(15, VCons(16,
  VCons(23, VCons(42, VNil))))));
```

(*) *supreme looseness and totally made-up syntax!!!*

Section Outline

- 4 Steps toward Dependent Types
 - Kinds
 - Language Extensions
 - Dependent Type Programming with Vectors
 - Heterogeneous Collections
 - Pi and Sigma Types

Kinds

Q: **Types** classify **Values**, but what classifies **Types**?

Kinds

Q: **Types** classify **Values**, but what classifies **Types**?

A: **Kinds**

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
```

```
Bool ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
```

```
Bool :: *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] ::
```


Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) ::
```


Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
Either String Earning ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
Either String Earning :: *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
Either String Earning :: *
Either String ::
```


Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
Either String Earning :: *
Either String :: * -> *
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
Either String Earning :: *
Either String :: * -> *
Either ::
```

Introducing ★

```
-- built-in magic: infinitely many value constructors
data Int = ... | -1 | 0 | 1 | 2 | ...
data Bool = False | True
data [a] = Nil | (:) a [a]
data Maybe a = Nothing | Just a
data (a, b) = (a, b)
data Either a b = Left a | Right b

(Person, Bool) :: *
(,) Person :: * -> *
(,) :: * -> * -> *
Either String Earning :: *
Either String :: * -> *
Either :: * -> * -> *
```

Introducing Constraint

Haskell has typeclasses that very loosely resemble interfaces in OOP. A basic **Typeclass** consists of a collection of function signatures for a **Type** to implement. Afterward, this **Typeclass** instance can be used to provide contexts for functions.

Introducing Constraint

Haskell has typeclasses that very loosely resemble interfaces in OOP. A basic **Typeclass** consists of a collection of function signatures for a **Type** to implement. Afterward, this **Typeclass** instance can be used to provide contexts for functions.

```
Show -- types that can be serialized to String
```

Introducing Constraint

Haskell has typeclasses that very loosely resemble interfaces in OOP. A basic **Typeclass** consists of a collection of function signatures for a **Type** to implement. Afterward, this **Typeclass** instance can be used to provide contexts for functions.

```
Show -- types that can be serialized to String  
Eq  -- types that can be compared for equality
```

Introducing Constraint

Haskell has typeclasses that very loosely resemble interfaces in OOP. A basic **Typeclass** consists of a collection of function signatures for a **Type** to implement. Afterward, this **Typeclass** instance can be used to provide contexts for functions.

```
Show -- types that can be serialized to String
Eq   -- types that can be compared for equality
Ord  -- types that can be ordered
```

Introducing Constraint

Haskell has typeclasses that very loosely resemble interfaces in OOP. A basic **Typeclass** consists of a collection of function signatures for a **Type** to implement. Afterward, this **Typeclass** instance can be used to provide contexts for functions.

```
Show  -- types that can be serialized to String
Eq    -- types that can be compared for equality
Ord   -- types that can be ordered
Num   -- types that are like numbers: +, -, *, ...
        -- and many others
```


Introducing Constraint

An example:

```
data Ordering = LT | EQ | GT

show      :: Show a => a -> String    -- toString()
(==)      :: Eq a  => a -> a -> Bool
compare   :: Ord a => a -> a -> Ordering
(+)       :: Num a => a -> a -> a
sequenceA :: (Applicative f, Traversable t) => t (f a)
          -> f (t a)
```

Introducing Constraint

An example:

```
data Ordering = LT | EQ | GT

show      :: Show a => a -> String    -- toString()
(==)      :: Eq a  => a -> a -> Bool
compare   :: Ord a => a -> a -> Ordering
(+)       :: Num a => a -> a -> a
sequenceA :: (Applicative f, Traversable t) => t (f a)
          -> f (t a)
```

A loose translation with `:` for implements:

```
enum Ordering { LT, EQ, GT }

String show<A>(A a) where A : Show
Bool  equal<A>(A a, A a) where A : Eq
Ordering compare<A>(A a, A a) where A : Ord
A plus<A>(A a, A a) where A : Num
F<T<_>> sequenceA<F,T>(T<F<_>> tfa) where F :
    Applicative, T : Traversable
```

Introducing Constraint

These **Typeclass** contexts have **Kind Constraint**.

```
Show :: * -> Constraint
```

```
Eq    :: * -> Constraint
```

```
Ord   :: * -> Constraint
```

```
Num   :: * -> Constraint
```

Introducing Constraint

These **Typeclass** contexts have **Kind Constraint**.

```
Show :: * -> Constraint
Eq    :: * -> Constraint
Ord   :: * -> Constraint
Num   :: * -> Constraint
```

```
{-# LANGUAGE ConstraintKinds #-}
```

```
type ShowContext a b = (Show a, Show b)
```

```
sameSerialization :: ShowContext a b => a -> b -> Bool
sameSerialization a b = show a == show b
```

Introducing Constraint

These **Typeclass** contexts have **Kind Constraint**.

```
Show  :: * -> Constraint
Eq    :: * -> Constraint
Ord   :: * -> Constraint
Num   :: * -> Constraint
```

```
{-# LANGUAGE ConstraintKinds #-}
```

```
type ShowContext a b = (Show a, Show b)
```

```
sameSerialization :: ShowContext a b => a -> b -> Bool
sameSerialization a b = show a == show b
```

```
ShowContext :: * -> * -> Constraint
```

Other Kinds

There are other **Kinds** aside from ***** and **Constraint**

```
import GHC.Prim

(*)          -- kind of fully realized type
(#)         -- kind of unboxed stuff used internally
Constraint  -- kind of constraints and type equality
OpenKind    -- superkind of (*) and (#)
AnyK        -- polymorphic kind for flexible arity
```

Other Kinds

There are other **Kinds** aside from ***** and **Constraint**

```
import GHC.Prim

(*)           -- kind of fully realized type
(#)          -- kind of unboxed stuff used internally
Constraint  -- kind of constraints and type equality
OpenKind   -- superkind of (*) and (#)
AnyK       -- polymorphic kind for flexible arity

-- the only sort, sorts classify kinds
(*), (#), Constraint, OpenKind, AnyK :: BOX
BOX :: BOX
```

All these **Kinds** are built-in and inferred as of GHC 7.10.2.

All of this will be changed with the next GHC 8.0 release.

Language Extensions

Compiler extensions that enable a variety of new functionalities:

Language Extensions

Compiler extensions that enable a variety of new functionalities:

- Syntax extension
- Type-level programming
- Generic deriving
- FFI
- Type disambiguation
- Typeclass extension

Language Extensions

Compiler extensions that enable a variety of new functionalities:

- Syntax extension
- Type-level programming
- Generic deriving
- FFI
- Type disambiguation
- Typeclass extension

Each extension has a name, and is enabled with the LANGUAGE pragma.

GADTs

Define data and explicit give type signatures to the **Value** constructors.

```
data Bool = False | True
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
```

GADTs

Define data and explicit give type signatures to the **Value** constructors.

```
data Bool = False | True
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
```

Becomes:

```
{-# LANGUAGE GADTs #-}
data Bool where
  False :: Bool
  True  :: Bool

data Maybe a where
  Nothing :: Maybe a
  Just    :: a -> Maybe a

data List a where
  Nil    :: List a
  Cons   :: a -> List a -> List a
```

GADTs

Define data and explicit give type signatures to the **Value** constructors.

```
data Bool = False | True
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
```

Loose translations:

```
enum Bool {
  Bool False,
  Bool True
}
```

```
enum Maybe<A> {
  Maybe<A> Nothing,
  Maybe<A> Just(A a)
}
```

```
enum List<A> {
  List<A> Nil,
  List<A> Cons(A a, List<A> as)
}
```

KindSignatures

Specify the **Kind** of the **Type** variables:

```
{-# LANGUAGE GADTs #-}  
{-# LANGUAGE KindSignatures #-}  
data Bool :: * where  
  False :: Bool  
  True  :: Bool  
  
data Maybe :: * -> * where  
  Nothing :: Maybe a  
  Just    :: a -> Maybe a  
  
data List :: * -> * where  
  Nil    :: List a  
  Cons   :: a -> List a -> List a
```

DataKinds

Kinds are built-in; no user defined **Kinds**.

DataKinds

Kinds are built-in; no user defined Kinds.

Want Values at the Type level though!

DataKinds

Kinds are built-in; no user defined Kinds.

Want Values at the Type level though!

=> Data kind promotion :)

DataKinds

Example:

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

With DataKinds, we get something like:

```
{- # LANGUAGE DataKinds #-}
```

Kind			Bool	
Type	Bool		'True	'False
Value	True	False		

DataKinds

Example:

```
data Nat = Z | S Nat
```

With DataKinds, we get something like:

```
{- # LANGUAGE DataKinds #-}
```

Kind		Nat
Type	Nat	'Z 'S Nat
Value	Z S Nat	

Example

Example with GADTs:

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

data Bool = False | True

data TextInput (a :: Bool) where
  RawText    :: String -> TextInput 'False
  SafeText   :: String -> TextInput 'True

sanitize :: TextInput a -> TextInput 'True
sanitize (RawText str) = SafeText (htmlEncode str)
sanitize x             = x
```

Notice that the *a* here is phantom.

Example

Translation^{*}:

```
enum Bool {
  Bool False,
  Bool True
}

enum TextInput<Bool b> {
  TextInput<'False> RawText(String str),
  TextInput<'True> SafeText(String str)
}

TextInput<'True> sanitize(TextInput<B> input) {
  switch input:
  case RawText(input):
    return SafeText(htmlEncode(input));
  default:
    return input;
}
```

(*) *supreme looseness and totally made-up syntax!!!*

Type Families

Type families - type level functions, computed and checked at compile time.

Type Families

Type families - type level functions, computed and checked at compile time.

Comes in 2 flavors:

- type synonym families
- data families

Type Families

Type families - type level functions, computed and checked at compile time.

Comes in 2 flavors:

- type synonym families
- data families

and have a few options:

- associated vs. unassociated
- open vs. closed¹
- injectivity²

Type Families

At **Value** level:

```
data Nat = Z | S Nat
```

```
add :: Nat -> Nat -> Nat
```

```
add Z      m = m
```

```
add (S n) m = S (add n m)
```

```
    add (S (S Z)) (S Z)
```

```
=> S (add (S Z) (S Z))
```

```
=> S (S (add Z (S Z)))
```

```
=> S (S (S Z))
```

Type Families

At **Type** level:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
```

```
data Nat = Z | S Nat
```

```
type family Add (n :: Nat) (m :: Nat) :: Nat where
  Add 'Z      m = m
  Add ('S n) m = 'S (Add n m)
```

```
    Add ('S ('S 'Z)) ('S 'Z)
=> 'S (Add ('S 'Z) ('S 'Z))
=> 'S ('S (Add 'Z ('S 'Z)))
=> 'S ('S ('S 'Z))
```

Type Operators

Allows usage of symbols in place of **Type** constructors and **Type** families.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
```

```
data Nat = Z | S Nat
```

```
type family (:+) n m where
  'Z      :+ m = m
  ('S n)  :+ m = 'S (n :+ m)
```

```
      ('S ('S 'Z)) :+ ('S 'Z)
=> 'S (('S 'Z) :+ ('S 'Z))
=> 'S ('S ('Z :+ ('S 'Z)))
=> 'S ('S ('S 'Z))
```

Extended Haskell

Assume LANGUAGE extensions are turned on from now on.

Extended Haskell

Assume LANGUAGE extensions are turned on from now on.

Bad news, no more translations :(

Vectors

Like `List`, but also indexed by `Nat` to indicate length.

Vectors

Like `List`, but also indexed by `Nat` to indicate length.

`List`:

```
data List a where
  Nil  :: List a
  Cons :: a -> List a -> List a
```

Vectors

Like `List`, but also indexed by `Nat` to indicate length.

`List`:

```
data List a where
  Nil  :: List a
  Cons :: a -> List a -> List a
```

`Vector`:

```
-- 'Z ~ 0
-- 'S n ~ n :+ 1
data Vect (n :: Nat) a where
  VNil  :: Vect Z a
  (:>) :: a -> Vect n a -> Vect (S n) a

type Six = S (S (S (S (S (S Z))))))
vs :: Vect Six Int
vs = 4 :> 8 :> 15 :> 16 :> 23 :> 42 :> VNil
```


Vectors

Like `List`, but also indexed by `Nat` to indicate length.

`List`:

```
data List a where
  Nil    :: List a
  Cons   :: a -> List a -> List a
```

Module `GHC.TypeLits` provide type-level literals:

```
-- 'Z ~ 0
-- 'S n ~ n :+ 1
data Vect (n :: Nat) a where
  VNil    :: Vect 0 a
  (:>)    :: a -> Vect n a -> Vect (n :+ 1) a

vs :: Vect 6 Int
vs = 4 :> 8 :> 15 :> 16 :> 23 :> 42 :> VNil
```

Head

head returns the first element of the [List](#):

```
-- from standard library
-- useless unless knowing list is non-empty
head :: [a] -> a
head []      = error "empty list"
head (x:xs) = x
```

Head

head returns the first element of the [List](#):

```
-- from standard library
-- useless unless knowing list is non-empty
head :: [a] -> a
head []      = error "empty list"
head (x:xs) = x
```

Elm now uses [Maybe](#):

```
mhead :: [a] -> Maybe a
mhead []      = Nothing
mhead (x:xs) = Just x
```

Head

head returns the first element of the [List](#):

```
-- from standard library
-- useless unless knowing list is non-empty
head :: [a] -> a
head []      = error "empty list"
head (x:xs) = x
```

Elm now uses [Maybe](#):

```
mhead :: [a] -> Maybe a
mhead []      = Nothing
mhead (x:xs) = Just x
```

With [Vector](#):

```
vhead :: Vect (S n) a -> a
vhead (x:>xs) = x
```

Append

append concatenates 2 Lists:

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

Append

append concatenates 2 Lists:

```
append :: [a] -> [a] -> [a]
append []      ys = ys
append (x:xs)  ys = x : append xs ys
```

With Vector:

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

Map

map maps a function over a List:

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs)      = f x : map f xs
```

Map

map maps a function over a List:

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs)      = f x : map f xs
```

With Vector:

```
vmap :: (a -> b) -> Vect n a -> Vect n b
vmap f VNil       = VNil
vmap f (x:>xs)    = f x :> vmap f xs
```


Zip

zip creates pair-wise tuples:

```
zip :: [a] -> [b] -> [(a,b)]  
zip (x:xs) (y:ys) = (x,y) : zip xs ys  
zip xs          ys          = []
```

Zip

zip creates pair-wise tuples:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip xs        ys        = []
```

With **Vector**:

```
vzip :: Vect n a -> Vect n b -> Vect n (a, b)
vzip (x:>xs) (y:>ys) = (x,y) :> vzip xs ys
vzip VNil    VNil    = VNil
```

zip

zip2 with **Min** type family:

```
type family Min n m where
  Min Z      m      = Z
  Min n      Z      = Z
  Min (S n) (S m) = S (Min n m)

vzip2 :: Vect n a -> Vect m b -> Vect (Min n m) (a, b)
vzip2 (x:>xs) (y:>ys) = (x,y) :> vzip2 xs ys
vzip2 xs      VNil    = VNil
vzip2 VNil    ys      = VNil
```

Heterogeneous List

Heterogeneous List indexed by List of Types:

```
infixr 5 ::>
data HList (t :: [*]) where
  HNil    :: HList '[]
  (::>) :: t -> HList ts -> HList (t ': ts)

defaults :: HList '[Int, Bool, Maybe a]
defaults = 0 ::> False ::> Nothing ::> HNil
```

Heterogeneous Vector

Heterogeneous **Vector** indexed by a **List** of **Types**:

```
infixr 5 :>>
data HVect (n :: Nat) (t :: [*]) where
  HVNil    :: HVect Z '[]
  (:>>)    :: t -> HVect n ts -> HVect (S n) (t ': ts)

vdefaults :: HVect 3 '[Int, Bool, Maybe a]
vdefaults = 0 :>> False :>> Nothing :>> HVNil
```

Replicate and Filter

replicate repeats an element n times:

```
replicate :: Int -> a -> [a]
replicate 0 x = []
replicate n x = x : replicate (n - 1) x
```

Replicate and Filter

replicate repeats an element n times:

```
replicate :: Int -> a -> [a]
replicate 0 x = []
replicate n x = x : replicate (n - 1) x
```

filter selects elements from a list for given predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs) = if f x
                  then x : filter f xs
                  else filter f xs
```

Pi Types

Π -types - **Values** in **Type** signatures. Fake by deriving singleton instances of **Sing** data family to reflect values to the type level.

```
-- fake with singleton types
data instance Sing (n :: Nat) where
  SZ :: Sing Z
  SS :: Sing n -> Sing (S n)

-- so we have
SZ :: Sing Z
SS SZ :: Sing (S Z)
SS (SS SZ) :: Sing (S (S Z))
```


Pi Types

Π -types - **Values** in **Type** signatures. Fake by deriving singleton instances of **Sing** data family to reflect values to the type level.

```
-- fake with singleton types
data instance Sing (n :: Nat) where
  SZ :: Sing Z
  SS :: Sing n -> Sing (S n)

-- so we have
SZ :: Sing Z
SS SZ :: Sing (S Z)
SS (SS SZ) :: Sing (S (S Z))

-- finally:
vreplicate :: Sing (n :: Nat) -> a -> Vect n a
vreplicate SZ      a = VNil
vreplicate (SS n) a = a :> vreplicate n a
```

Sigma Types

Σ -types - tuple where 2nd value depends on 1st:

```
-- using Idris's ** dependent pair syntax
(3 ** 'a' :> 'b' :> 'c' :> VNil)
:: (n :: Nat ** Vect n Char)

vfilter :: (a -> Bool) -> Vect n a
        -> (p :: Nat ** Vect p a)
```

Sigma Types

Σ -types - tuple where 2nd value depends on 1st:

```
-- using Idris's ** dependent pair syntax
(3 ** 'a' :> 'b' :> 'c' :> VNil)
:: (n :: Nat ** Vect n Char)

vfilter :: (a -> Bool) -> Vect n a
        -> (p :: Nat ** Vect p a)
```

Credit to Ertugrul Söylemez:

```
data Sigma :: KProxy a -> (a -> *) -> * where
  (**) :: Sing (x :: a) -> b x
        -> Sigma ('KProxy :: KProxy a) b
```

Section Outline

- 5 Closing
 - Beyond
 - Questions

Beyond Dependent Types

- Total functional languages
 - termination and totality check
 - disallow partial functions
 - distinction between **data** and **codata**

Beyond Dependent Types

- Total functional languages
 - termination and totality check
 - disallow partial functions
 - distinction between **data** and **codata**
- Proof assistant languages
 - Ph.D. first please

Questions?