Introduction to Dependent Types

Eagan Technology Unconference

Joseph Ching

September 22, 2015

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Section Outline

Quick Question

How many are familiar with this topic?

A Joke

This is not a `m-` tutorial.

A Joke

This is not a `m-` tutorial. Nor is it a `lens` tutorial

A Joke

This is not a `m-` tutorial. Nor is it a `lens` tutorial (aka the new new `m-` tutorial. . .

A Joke

This is not a `m-` tutorial. Nor is it a `lens` tutorial (aka the new new `m-` tutorial......because `arrows` *were* the new `m-` tutorials).

About This Talk

`Agda`, `Idris`, `Coq` and co* have full support for dependent types.

About This Talk

`Agda`, `Idris`, `Coq` and co* have full support for dependent types.

Because of that, it's harder to see the build up, so we won't be directly using them in this talk.

About This Talk

`Agda`, `Idris`, `Coq` and co* have full support for dependent types.

Because of that, it's harder to see the build up, so we won't be directly using them in this talk.

Honestly though, it's because they're way over my head :(

(*) There was another mini joke here...

About This Talk

But we will be using `Haskell` though :)

About This Talk

But we will be using `Haskell` though :)

It's not truely dependent, but we can do more and more with each `language extension` that comes along.

About This Talk

But we will be using `Haskell` though :)

It's not truely dependent, but we can do more and more with each `language extension` that comes along.

For the examples, there will also be *very* loose translation to imperative/OOP. Though please keep in mind that these are merely syntax translations, the actual concepts can 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 Types

How are data types defined?

About Types

How are data types defined?

- Some are built in magic: Int, Char, functions

About Types

How are data types defined?

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

About Types

How are data types defined?

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

About Types

What are the data types like?

About Types

What are the data types like?

- Multiple Value constructors

About Types

What are the data types like?

- Multiple Value constructors
- Paremetrize over another type

About Types

What are the data types like?

- Multiple Value constructors
- Paremetrize over another type
- Recursive definition

About Types

What are the data types like?

- Multiple Value constructors
- Paremetrize over another type
- Recursive definition
- Synonyms of other types

About Types

What are the data types like?

- Multiple Value constructors
- Paremetrize over another type
- Recursive definition
- Synonyms of other types
- A 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
```

Our First Example!

Define a person:

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

A loose translation:

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

Our First Example!

Define a person:

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

A loose translation:

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

In this example, the Type and Value constructor have the same name. The Type of the Person constructor:

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

bobby :: Person
bobby = Person "Bobby" "Smith" 23
```

Our First Example!

Define a person:

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

A loose translation:

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

In this example, the Type and Value constructor have the same name. The Type of the Person constructor:

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

bobby :: Person
bobby = Person "Bobby" "Smith" 23

-- a loose translation:
Person bobby = new Person("Bobby", "Smith", 23)
```

Multiple Value Constructors

Data can have multiple Value constructors:

```
data Bool = False | True

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

Does this remind you of anything?

Multiple Value Constructors

Data can have multiple Value constructors:

```
data Bool = False | True

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

Does this remind you of anything?

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 Side $\sim$ Radius $\sim$ 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 Value constructors:

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

Multiple Value Constructor

Recall Side ∼ Radius ∼ Double:

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

Types of the 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
```

Parametrization

Types can parametrize over another type:

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

A loose translation:

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

Parametrization

Types can parametrize over another type:

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

A loose translation:

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

The Type of the Identity constructor:

```
Identity :: a -> Identity a

intIdwrtSum :: Indentity Int
intIdwrtSum = Identity 0
```

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
```

A loose translation:

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

Tuple

Parametrize over 2 types - 2-tuple!

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

A loose translation:

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

With:

```
Tuple :: a -> b -> Tuple a 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" "Sakura" 30, True)
chet    = (Person "Chet" "Awesome-Laser" 2, False)
luffy   = (Person "Luff D." "Monkey" 19, False)
```

Maybe

Like Bool, but parametrize an *a* over the True part:

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

Maybe

Like Bool, but parametrize an *a* over the True part:

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

A loose translation:

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

Maybe

Like Bool, but parametrize an *a* over the True part:

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

A loose translation:

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

The Types of the two Value constructors:

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

Maybe

From previous slide:

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

Maybe

From previous slide:

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

Say more with the occupation:

```
type Occupation = Maybe String

barbara, chet, luffy :: (Person, Occupation)
barbara = (Person "Barbara" "Sakura" 30, Just "dancer")
chet    = (Person "Chet" "Awesome-Laser" 2, Nothing)
luffy   = (Person "Luff D." "Monkey" 19, Just "pirate")
```

Either

Like Bool, but parametrize over both True and False:

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

Either

Like Bool, but parametrize over both True and False:

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

A loose translation:

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

Either

Like Bool, but parametrize over both True and False:

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

A loose translation:

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

The two Value constructors have Types:

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

```
type Earning = Either String Int

barbara, chet, luffy :: (Person, Earning)
barbara = (Person "Barbara" "Sakura" 30,
           Right 100000)
chet    = (Person "Chet" "Awesome-Laser" 2,
           Left "Is a baby")
luffy   = (Person "Luff D." "Monkey" 19,
           Right 2000000)
```

Types with Recursion

Natural number:

```
data Nat = Z | S Nat

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

Types with Recursion

Natural number:

```
data Nat = Z | S Nat

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 while parametrize over another type:

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

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

Types with Recursion

List - recursive type while parametrize over another type:

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

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]
```

De-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 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 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:

```
toInt :: Nat -> Int
toInt Z     = 0
toInt (S n) = 1 + toInt n
```

Functions with Recursion

Use recursion for recursive types:

```
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);
}
```

Type of Functions

Q: Actual type of functions?

Type of Functions

Q: Actual type of functions?
A: Built-in magic, it's called the *function arrow*, something like:

```
data (->) a b = implementation

even  :: Int -> Bool
toInt :: Nat -> Int
```

Functions with Parametric Polymorphism

Functions can be parametric:

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

Functions with Parametric Polymorphism

Functions can be parametric:

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

Not as loose translation:

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

Functions with Parametric Polymorphism

Functions can be parametric:

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

Functions with Parametric Polymorphism

Functions can be parametric:

```
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);
}
```

Higher-order Functions

Functions that take functions as params:

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

-- acutal 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

-- acutal 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

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<A> 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);
}
```

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
- 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

=> 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 told 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 told you...
...you already know at least 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 $T <: P$.

The name for this extension of λ-calculus is called *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 $T <: P$.

The name for this extension of λ-calculus is called *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 $T <: P$.

The name for this extension of λ-calculus is called *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 [a] = [] | (:) a [a]

id :: a -> a
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 [a] = [] | (:) a [a]

id :: a -> a
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
```

f maps numbers 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
```

f maps numbers 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
```

f maps numbers 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)
```

List takes 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)
```

List takes 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)
```

List takes 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)
```

List takes 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

Re-thinking parametrized data types:

```
data Maybe a = Nothing | Just a

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

List takes 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

Are we seeing a pattern yet?

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: $\lambda \underline{\omega}$
  => Type-level programming via type operators

Value and Type Interdependency

Then what about the other cases of dependencies?

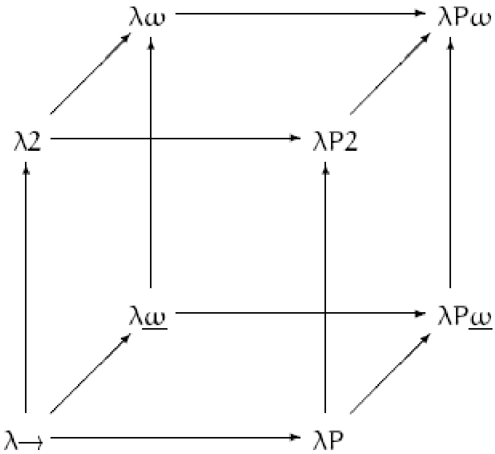
- Values depending on Values: λ-calculus
- Values depending on Types: $\lambda 2$, *System F*
- Types depending on Types: $\lambda \underline{\omega}$
  => Type-level programming via type operators
- Types depending on Values:

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: $\lambda\underline{\omega}$
  => Type-level programming via type operators
- Types depending on Values: $\lambda\Pi$
  => Dependent types

Lambda Cube



Pi and Sigma Types

Π-type - quantification of values on type level

Pi and Sigma Types

Π-type - quantification of values on type level

Σ-type - dependent pair

System F_c

Currently, Haskell as of GHC 7.10.2 doesnot have true type operators. Achieves type-level programming through *type families* and 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 doesnot have true type operators. Achieves type-level programming through *type families* and equalities on Types. This axis of extension on $\lambda 2$ is termed System F_c.

This, plus a Kind system, and a handful of *language extensions*, we are ready fake dependent types in Haskell.

Teaser

Example please:

```
data Vec (n :: Nat) a where
  VNil :: Vec 0 a
  (:>) :: a -> Vec n a -> Vec (n + 1) a

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

Teaser

Example please:

```
data Vec (n :: Nat) a where
  VNil :: Vec 0 a
  (:>) :: a -> Vec n a -> Vec (n + 1) a

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

Translation* please:

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

Vec<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

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

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
Either String Earning ::
```

Introducing ⋆

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

```haskell
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
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 Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
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 Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
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 Either a b = Left a | Right b

Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
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 Either a b = Left a | Right b
```

```
Int :: *
Bool :: *
[Int] :: *
[] :: * -> *
Maybe Person :: *
Maybe :: * -> *
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 context 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 context 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 context 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 context 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 context 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 equals<A>(A a, A a) where A : Eq
Ordering compare<A>(A a, A a) where A : Ordering
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.

```
{-# LANGUAGE ConstraintKinds #-}

type ShowCxt a b = (Show a, Show b)

sameSerialization :: ShowContxt a b => a -> b -> Bool
sameSerialization a b = show a == show b

ShowCxt :: * -> * -> 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.

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 programming
- FFI
- Type disambiguation
- Typeclass extension

Language Extensions

Compiler extensions that enable a variety of new functionalities:

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

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

GADTs

KindSignatures

ConstraintKinds

Type Operators

DataKinds

Type Families

Section Outline

Questions?