# Towards Palatable Functional Programming with Dependent Type Theories

Donovan Crichton

August 2023

# Preliminaries

- Slides and Examples available at:
  https://github.com/donovancrichton/Talks
- This talk: BFPG/PalatableProgrammingWithDTT

# About me

Australian National University

- PhD Candidate
- Computing Foundations
- School of Computing

Griffith University

- Visiting Scholar
- Trusted Systems Lab
- IIIS

ASD Australian Signals Directorate

- ASD Co-Lab Scholar

# Recalling some definitions to get us started

# Rose Tree Maps: Haskell vs Idris

The Idris code will not pass the totality checker.

```haskell
module HaskellRose where
{-# LANGUAGE GADTs #-}

data Rose :: * -> *
  where
  MkRose :: a -> [Rose a]
    -> Rose a

instance Functor Rose
  where
  fmap f
   (MkRose node children)
     = MkRose (f node)
       (fmap (fmap f)
         children)
```

```idris
module IdrisRoseBad
%default total -- WHY!??

data Rose : Type -> Type
  where
  MkRose : a -> List (Rose a)
    -> Rose a

implementation Functor Rose
  where
  map f
   (MkRose node children)
     = MkRose (f node)
       (map (map f)
         children)
```

# What is a dependent type?

**Dependent Type(s): $\Pi_{x \in A}.B(x)$ instead of $\lambda_{x \in A}.B$**

A dependent type is a type that *depends* on a specific value of its input.

**$\lambda_{x \in A}.B$ in Idris code, in general.**

```
f : a -> b
f x = ?somedefinition
```

**$\Pi_{x \in A}.B(x)$ in Idris code, in general.**

```
module PiGeneralExample
P : a -> Type
P = ?sometypedefinition

f : (x : a) -> P x
f x = ?somevaluedefinition
```

# What is a dependent type? (concrete examples)

## $\lambda_{x \in A}.B(x)$ in Idris code, concretely.

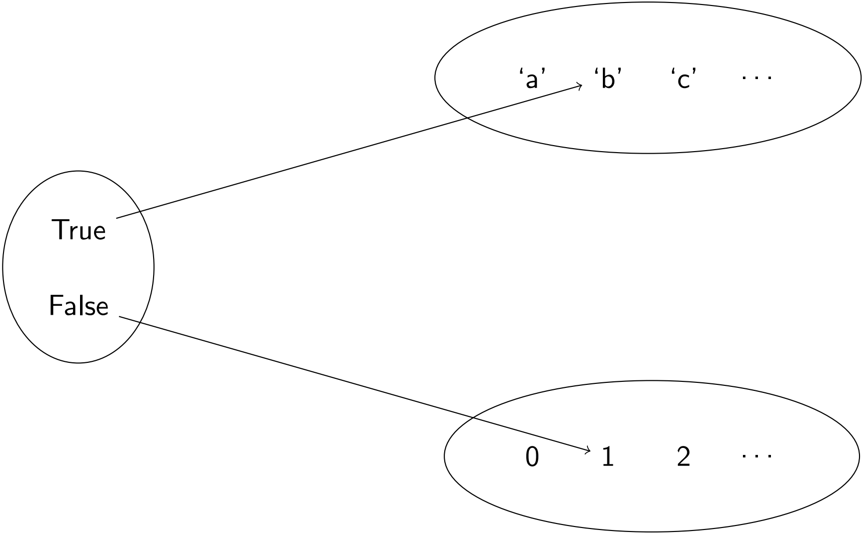
```
f : a -> List
f x = [x]
```

## $\Pi_{x \in A}.B(x)$ in Idris code, concretely.

```
module PiSpecificExample
data Vector : Nat -> Type -> Type where
  Nil : Vector 0 a
  (::) : a -> Vector k a -> Vector (S k) a

trues : (k : Nat) -> Vector k Bool
trues Z = []
trues (S k) = True :: trues k
```

# What is a dependent type? (diagrammatically)



# What is a Dependent Type? (diagrammatic example)

## Did you guess the right Idris type?

```
module PiDiagramExample
CharOrNat : Bool -> Type
CharOrNat True = Char
CharOrNat False = Nat

bOrOne : (b : Bool) -> CharOrNat b
bOrOne True = 'b'
bOrOne False = 1
```

## What about $\Sigma_{x \in A}.B(x)$?

Dependent pairs can be constructed from $\Pi$ types if your language supports recursive algebraic data type definitions.

# Why do we mean by totality? (good)

## Total Functions

A functions is *total* if it terminates and is defined for all possible values for its domain.

## Functions that are total.

```
module TotalExample
contradiction1 : Bool -> Bool
contradiction1 x = False

contradiction2 : Bool -> Bool
contradiction2 True = False
contradiction2 False = False
```

# Why do we mean by totality? (bad)

### Total Functions

A functions is *total* if it terminates and is defined for all possible values for its domain.

### Functions that are *not* total.

```
module TotalBadExample
contradiction : Bool -> Bool
contradiction True = False

contradiction : Bool -> Bool
contradiction x = not $ contradiction x
```

# Why do we care about totality?

## Can we type check this?

$f : \mathbb{N} \rightarrow \mathbb{N}$
$f = \lambda(x \in \mathbb{N}).f\ x$

$v : Vect\ k\ \alpha - > Vect\ (f\ \ k)\ \alpha$
$v\ xs = xs$

## Infinite recursion in types in Idris

```
module ForeverExample
import Data.Vect

f : Nat -> Nat
f x = f x

v : Vect k a -> Vect (f k) a
v xs = xs
```

# Using the totality pragma.

### How Idris reduces Π application in types.

Idris will *only* reduce function application in types if the function has passed the totality checker.

### The totality checker blocks evaluation:

```
module ForeverExampleFail
import Data.Vect
%default total
f : Nat -> Nat
f x = f x

v : Vect k a -> Vect (f k) a
v xs = xs
```

# References