Dependent types in Idris

Łukasz Hanuszczak

April 02, 2015

Why?

Static typing

If it compiles, it runs.

– some Haskell guy

Static typing

If it compiles, it runs.

– some Haskell guy

It's a lie.

The night is dark and full of terrors

```
import System.IO.Unsafe

unsafePerformIO :: IO a -> a
```

The night is dark and full of terrors

```
import System.IO.Unsafe

unsafePerformIO :: IO a -> a
```

Not that bad.

The night is dark and full of terrors

```
import System.IO.Unsafe

unsafePerformIO :: IO a -> a
```

Not that bad.

```
head :: [a] -> a
tail :: [a] -> [a]
(!!) :: Int -> [a] -> a
```

The night is dark and full of terrors

```
import System.IO.Unsafe

unsafePerformIO :: IO a -> a
```

Not that bad.

```
head :: [a] -> a
tail :: [a] -> [a]
(!!) :: Int -> [a] -> a
```

Dreadful.

The night is dark and full of terrors

```
import System.IO.Unsafe

unsafePerformIO :: IO a -> a
```

Not that bad.

```
head :: [a] -> Maybe a
tail :: [a] -> Maybe [a]
(!!) :: Int -> [a] -> Maybe a
```

Useless.

Previously. . .

Tagged lists

```
data Empty
data NonEmpty

data List t a where
    Nil :: List Empty a
    Cons :: a -> List t a -> List NonEmpty a
```

Tagged lists

```
data Empty
data NonEmpty

data List t a where
    Nil :: List Empty a
    Cons :: a -> List t a -> List NonEmpty a

head :: List NonEmpty a -> a
head (Cons x _) = x
```

Tagged lists

```
data Empty
data NonEmpty

data List t a where
    Nil :: List Empty a
    Cons :: a -> List t a -> List NonEmpty a

head :: List NonEmpty a -> a
head (Cons x _) = x

tail :: List NonEmpty a -> ???
tail (Cons _ t) = t
```

Vectors (first attempt)

```
{-# LANGUAGE GADTs #-}
```

Vectors (first attempt)

```
{-# LANGUAGE GADTs #-}

data Zero
data Succ n where
    Succ :: Succ n
```

Vectors (first attempt)

```
{-# LANGUAGE GADTs #-}

data Zero
data Succ n where
    Succ :: Succ n

data Vect n a where
    Nil :: Vect Zero a
    Cons :: a -> Vect n a -> Vect (Succ n) a
```

Vectors (first attempt)

```
{-# LANGUAGE GADTs #-}

data Zero
data Succ n where
    Succ :: Succ n

data Vect n a where
    Nil :: Vect Zero a
    Cons :: a -> Vect n a -> Vect (Succ n) a

head :: Vector (Succ n) a -> a
head (Cons x xs) = x

tail :: Vector (Succ n) a -> Vector n a
tail (Cons x xs) = xs
```

Vectors (first attempt)

```
{-# LANGUAGE GADTs #-}

data Zero
data Succ n where
    Succ :: Succ n

data Vect n a where
    Nil :: Vect Zero a
    Cons :: a -> Vect n a -> Vect (Succ n) a

head :: Vector (Succ n) a -> a
head (Cons x xs) = x


tail :: Vector (Succ n) a -> Vector n a
tail (Cons x xs) = xs

(++) :: Vector n a -> Vector m a -> ???
(++) Nil ys = ys
(++) (Cons x xs) ys = Cons x (xs ++ ys)
```

Vectors (first+ attempt)

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}
```

Vectors (first+ attempt)

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}

class Plus n m nm
instance Plus Zero m m
instance (Plus n m nm) => Plus (Succ n) m (Succ nm)
```

Vectors (first+ attempt)

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}

class Plus n m nm
instance Plus Zero m m
instance (Plus n m nm) => Plus (Succ n) m (Succ nm)

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

Vectors (first+ attempt)

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-}
{-# LANGUAGE UndecidableInstances #-}

class Plus n m nm
instance Plus Zero m m
instance (Plus n m nm) => Plus (Succ n) m (Succ nm)

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

. . . but it doesn't work.

Vectors (second attempt)

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

Vectors (second attempt)

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

data Nat = Zero | Succ Nat
```

Vectors (second attempt)

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

data Nat = Zero | Succ Nat

type family Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus Zero m = m
type instance Plus (Succ n) m = Succ (Plus n m)
```

Vectors (second attempt)

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

data Nat = Zero | Succ Nat

type family Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus Zero m = m
type instance Plus (Succ n) m = Succ (Plus n m)

data Vect n a where
    Nil :: Vect Zero a
    Cons :: a -> Vect n a -> Vect (Succ n) a
```

Vectors (second attempt)

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

data Nat = Zero | Succ Nat

type family Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus Zero m = m
type instance Plus (Succ n) m = Succ (Plus n m)

data Vect n a where
    Nil :: Vect Zero a
    Cons :: a -> Vect n a -> Vect (Succ n) a

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

Idris

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation
- foreign function interface (C, JavaScript)

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation
- foreign function interface (C, JavaScript)
- type classes

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation
- foreign function interface (C, JavaScript)
- type classes
- function overloading

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation
- foreign function interface (C, JavaScript)
- type classes
- function overloading
- a lot of sugar

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation
- foreign function interface (C, JavaScript)
- type classes
- function overloading
- a lot of sugar
- modules, namespaces

Idris

It is a general purpose pure functional programming language with dependent types, featuring:

- Haskell-like syntax
- totality checking
- eager evaluation
- foreign function interface (C, JavaScript)
- type classes
- function overloading
- a lot of sugar
- modules, namespaces
- ... a lot more

Dependent types

What does it mean to be functional?

Dependent types

What does it mean to be functional?

To have functions as first class values.

Dependent types

What does it mean to be functional?

To have functions as first class values.

What does it mean to be dependently typed?

Dependent types

What does it mean to be functional?

To have functions as first class values.

What does it mean to be dependently typed?

To have types as first class values.

Types as first class values

Haskell

```
type Foo = [Int]
```

Types as first class values

Haskell

```
type Foo = [Int]
```

Idris

```
Foo : Type
Foo = List Int
```

Types as first class values

Haskell

```
type Foo = [Int]
```

Idris

```
Foo : Type
Foo = List Int

foo : Bool -> Type
foo True  = Int
foo False = String
```

Peano numbers

```
data Nat : Type where
     Z : Nat
     S : Nat -> Nat
```

Peano numbers

```
data Nat : Type where
     Z : Nat
     S : Nat -> Nat

(+) : Nat -> Nat -> Nat
(+) Z     m = m
(+) (S n) m = S (n + m)
```

Vectors (finally, the right way!)

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

Vectors (finally, the right way!)

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

head : Vect (S n) a -> a
head (x :: _) = x

tail : Vect (S n) a -> Vect n a
tail (_ :: xs) = xs
```

Vectors (finally, the right way!)

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

head : Vect (S n) a -> a
head (x :: _) = x

tail : Vect (S n) a -> Vect n a
tail (_ :: xs) = xs
```

(heavy breathing)

Vectors (finally, the right way!)

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

head : Vect (S n) a -> a
head (x :: _) = x

tail : Vect (S n) a -> Vect n a
tail (_ :: xs) = xs
```

(heavy breathing)

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

A new roadblock...

```
index : Nat -> Vect n a -> a
index Z     (x :: _)  = x
index (S n) (_ :: xs) = index n xs
index (S n) Nil       = ???
```

. . . and a solution

```
data Fin : Nat -> Type where
    FZ : Fin (S n)
    FS : Fin n -> Fin (S n)
```

. . . and a solution

```
data Fin : Nat -> Type where
     FZ : Fin (S n)
     FS : Fin n -> Fin (S n)

index : Fin n -> Vect n a -> a
index FZ     (x :: _)  = x
index (FS n) (_ :: xs) = index n xs
```

Pairs

Pairs

Boring ones...

```
data Pair a b = MkPair a b

foo : (String, Int)
foo = ("Foo", 42)
```

Pairs

Boring ones...

```
data Pair a b = MkPair a b

foo : (String, Int)
foo = ("Foo", 42)
```

... and dependent ones

```
data Sigma : (A : Type) -> (P : A -> Type) -> Type where
     MkSigma : {A : Type} -> {P : A -> Type}
            -> (a : A) -> P a -> Sigma A P

bar : Sigma Nat (\n => Vect n String)
bar = MkSigma 2 ["a", "b"]
```

Pairs

Boring ones...

```
data Pair a b = MkPair a b

foo : (String, Int)
foo = ("Foo", 42)
```

... and dependent ones

```
data Sigma : (A : Type) -> (P : A -> Type) -> Type where
     MkSigma : {A : Type} -> {P : A -> Type}
            -> (a : A) -> P a -> Sigma A P

bar : (n : Nat ** Vect n String)
bar = (2 ** ["a", "b"])
```

Pairs

Boring ones...

```
data Pair a b = MkPair a b

foo : (String, Int)
foo = ("Foo", 42)
```

... and dependent ones

```
data Sigma : (A : Type) -> (P : A -> Type) -> Type where
     MkSigma : {A : Type} -> {P : A -> Type}
            -> (a : A) -> P a -> Sigma A P

bar : (n ** Vect n String)
bar = (_ ** ["a", "b"])
```

Filtering

```
filter : (a -> Bool) -> Vect n a -> ???
filter _ [] = []
filter p (x :: xs) =
    let xs' = filter p xs
    in if p x then x :: xs' else xs'
```

Filtering

```
filter : (a -> Bool) -> Vect n a -> (m ** Vect m a)
filter _ [] = (0 ** [])
filter p (x :: xs) =
    let (m ** xs') = filter p xs
    in if p x then (S m ** x :: xs') else (m ** xs')
```

Filtering

```
filter : (a -> Bool) -> Vect n a -> (m ** Vect m a)
filter _ [] = (_ ** [])
filter p (x :: xs) =
    let (_ ** xs') = filter p xs
    in if p x then (_ ** x :: xs') else (_ ** xs')
```

Predicates

Program is a proof of its type.

Predicates

Program is a proof of its type.

```
data Elem : a -> Vect n a -> Type where
    Here : Elem x (x :: xs)
    There : Elem x xs -> Elem x (y :: xs)
```

Predicates

Program is a proof of its type.

```
data Elem : a -> Vect n a -> Type where
    Here : Elem x (x :: xs)
    There : Elem x xs -> Elem x (y :: xs)

numbers : Vect 6 Int
numbers = [4, 8, 15, 16, 23, 42]

test : Elem 15 numbers
test = There (There Here)
```

Predicates

```
mapEl : {xs : Vect n a} -> {f : a -> b}
     -> Elem x xs -> Elem (f x) (map f xs)
mapEl Here       = Here
mapEl (There e) = There (mapEl e)
```

Predicates

```
mapEl : {xs : Vect n a} -> {f : a -> b}
     -> Elem x xs -> Elem (f x) (map f xs)
mapEl Here       = Here
mapEl (There e) = There (mapEl e)

numbers' : Vect 6 Int
numbers' = map (* 2) numbers

test' : Elem 30 numbers'
test' = mapEl test { f = (* 2) }
```

Predicates

```
mapEl : {xs : Vect n a} -> {f : a -> b}
     -> Elem x xs -> Elem (f x) (map f xs)
mapEl Here       = Here
mapEl (There e) = There (mapEl e)

numbers' : Vect 6 Int
numbers' = map (* 2) numbers

test' : Elem 30 numbers'
test' = mapEl test { f = (* 2) }

replaceEl : (xs : Vect k t) -> Elem x xs -> (y : t)
         -> (ys : Vect k t ** Elem y ys)
replaceEl (_ :: xs) Here       y = (y :: xs ** Here)
replaceEl (x :: xs) (There ex) y =
    let (ys ** ey) = replaceEl xs ex y
    in (x :: ys ** There ey)
```

Lambda calculus interpreter

Goals

- simple types (integers, booleans and functions)
- variables
- *if-then-else* construct
- compile-time rejection of ill-typed expressions

Types

```
data Ty : Type where
     TyInt  : Ty
     TyBool : Ty
     TyFun  : Ty -> Ty -> Ty
```

Types

```
data Ty : Type where
     TyInt  : Ty
     TyBool : Ty
     TyFun  : Ty -> Ty -> Ty

data HasType : Fin n -> Vect n Ty -> Ty -> Type where
     Stop : HasType FZ (t :: _) t
     Pop  : HasType i G t -> HasType (FS i) (_ :: G) t
```

Types

```
data Ty : Type where
    TyInt  : Ty
    TyBool : Ty
    TyFun  : Ty -> Ty -> Ty

data HasType : Fin n -> Vect n Ty -> Ty -> Type where
    Stop : HasType FZ (t :: _) t
    Pop  : HasType i G t -> HasType (FS i) (_ :: G) t

interpTy : Ty -> Type
interpTy TyInt       = Int
interpTy TyBool      = Bool
interpTy (TyFun a r) = interpTy a -> interpTy r
```

Expressions

```
data Expr : Vect n Ty -> Ty -> Type where
     Var : HasType i G t -> Expr G t
     Val : Int -> Expr G TyInt
     Lam : Expr (a :: G) r -> Expr G (TyFun a r)
     App : Expr G (TyFun a r) -> Expr G a -> Expr G r
     Op  : (interpTy a -> interpTy b -> interpTy c)
        -> Expr G a -> Expr G b -> Expr G c
     If  : Expr G TyBool
        -> Lazy (Expr G t) -> Lazy (Expr G t)
        -> Expr G t
```

Expression examples

```
plus : Expr G (TyFun TyInt (TyFun TyInt TyInt))
plus = Lam (Lam (Op (+) (Var Stop) (Var (Pop Stop))))
```

Expression examples

```
plus : Expr G (TyFun TyInt (TyFun TyInt TyInt))
plus = Lam (Lam (Op (+) (Var Stop) (Var (Pop Stop))))

fact : Expr G (TyFun TyInt TyInt)
fact = (Lam (If (Op (==) (Var Stop) (Val 0))
                (Val 1)
                (Op (*) (App fact (Op (-) (Var Stop)
                                          (Val 1)))
                        (Var Stop))))
```

Environment

```
data Env : Vect n Ty -> Type where
     Nil  : Env Nil
     (::) : interpTy t -> Env G -> Env (t :: G)
```

Environment

```
data Env : Vect n Ty -> Type where
     Nil  : Env Nil
     (::) : interpTy t -> Env G -> Env (t :: G)

lookup : HasType i G t -> Env G -> interpTy t
lookup Stop    (t :: _)  = t
lookup (Pop i) (_ :: ts) = lookup i ts
```

Interpreter

```
interp : Env G -> Expr G t -> interpTy t
interp env (Var x)     = lookup x env
interp env (Val c)     = c
interp env (Lam e)     = \a => interp (a :: env) e
interp env (App f e)   = (interp env f) (interp env e)
interp env (Op f l r) = f (interp env l) (interp env r)
interp env (If c t f) = if interp env c
                        then interp env t
                        else interp env f
```

Interpreter

```
interp : Env G -> Expr G t -> interpTy t
interp env (Var x)     = lookup x env
interp env (Val c)     = c
interp env (Lam e)     = \a => interp (a :: env) e
interp env (App f e)   = (interp env f) (interp env e)
interp env (Op f l r) = f (interp env l) (interp env r)
interp env (If c t f) = if interp env c
                        then interp env t
                        else interp env f

interp' : Expr [] t -> interpTy t
interp' = interp []
```

Theorem proving

Equality

```
data (=) : a -> b -> Type where
  Refl : x = x
```

Equality

```
data (=) : a -> b -> Type where
  Refl : x = x

fiveIsFive : 5 = 5
fiveIsFive = Refl
```

Equality

```
data (=) : a -> b -> Type where
  Refl : x = x

fiveIsFive : 5 = 5
fiveIsFive = Refl

twoTwosIsFour : 2 + 2 = 4
twoTwosIsFour = Refl
```

Equality

```
data (=) : a -> b -> Type where
  Refl : x = x

fiveIsFive : 5 = 5
fiveIsFive = Refl

twoTwosIsFour : 2 + 2 = 4
twoTwosIsFour = Refl

eqVectLength : (xs : Vect n a) -> (ys : Vect m a)
            -> (xs = ys) -> n = m
eqVectLength _ _ Refl = Refl
```

Induction

```
cong : {f : t -> u} -> (a = b) -> f a = f b
cong Refl = Refl
```

Induction

```
cong : {f : t -> u} -> (a = b) -> f a = f b
cong Refl = Refl

plusReducesS : (n : Nat) -> (m : Nat)
            -> S (n + m) = n + (S m)
plusReducesS Z m = Refl
plusReducesS (S n) m = cong (plusReducesS n m)
```

Impossible

Usually, when particular pattern match is not possible we just omit the clause. However, sometimes it is useful (as we will see in a moment) to mark it explicitly - Idris provides us with `impossible` keyword for such cases.

Impossible

Usually, when particular pattern match is not possible we just omit the clause. However, sometimes it is useful (as we will see in a moment) to mark it explicitly - Idris provides us with `impossible` keyword for such cases.

```
foo : (n : Nat) -> Vect n a -> Nat
foo Z     (_::_) impossible
foo (S _) []     impossible
foo n     _      = n
```

Negation

```
data Void
```

Negation

```
data Void
```

Having $\bot$ everything is possible:

```
void : Void -> a
```

Negation

```
data Void
```

Having $\bot$ everything is possible:

```
void : Void -> a
```

```
Not : Type -> Type
Not a = a -> Void
```

Negation

```
data Void
```

Having $\bot$ everything is possible:

```
void : Void -> a
```

```
Not : Type -> Type
Not a = a -> Void

data IsZero : Nat -> Type where
  Zero : IsZero Z

succNonZero : (n : Nat) -> IsZero (S n) -> Void
succNonZero _ Zero impossible
```

Negation

```
data Void
```

Having $\bot$ everything is possible:

```
void : Void -> a
```

```
Not : Type -> Type
Not a = a -> Void

data IsZero : Nat -> Type where
  Zero : IsZero Z

succNonZero : (n : Nat) -> Not (IsZero (S n))
succNonZero _ Zero impossible
```

Decidability

```
data Dec : Type -> Type where
    Yes : {A : Type} -> A -> Dec A
    No  : {A : Type} -> Not A -> Dec A
```

Decidability

```
data Dec : Type -> Type where
    Yes : {A : Type} -> A -> Dec A
    No  : {A : Type} -> Not A -> Dec A

decIsZero : (n : Nat) -> Dec (IsZero n)
decIsZero Z     = Yes Zero
decIsZero (S n) = No (succNonZero n)
```

Interactive proving

Idris has many facilities to help with theorem proving:

- good editor suport (Emacs and Vim)
- set of builtin syntax rules
- tactics language

Dependently typed lambda calculus

Syntax

$$e, T ::= x \mid \lambda x.e \mid e_1 e_2 \mid (x : T_1) \to T_2 \mid \mathcal{T}$$

where e stands for expressions, T for types (they are the same thing but are distinguished here for clarity), $* \to *$ for functional type and $\mathcal{T}$ for *type of types*.

Syntax

$$e, T ::= x \mid \lambda x.e \mid e_1 e_2 \mid (x : T_1) \to T_2 \mid \mathcal{T}$$

where e stands for expressions, T for types (they are the same thing but are distinguished here for clarity), $* \to *$ for functional type and $\mathcal{T}$ for *type of types*.

Having something like *type of types* is bad but makes everything simple.

Typechecking rules

$$\frac{}{\Gamma \vdash \mathcal{T} : \mathcal{T}} \textbf{ type}$$

Typechecking rules

$$\frac{}{\Gamma \vdash \mathcal{T} : \mathcal{T}} \textbf{ type}$$

$$\frac{}{\Gamma, x : T \vdash x : T} \textbf{ var}$$

Typechecking rules

$$\frac{}{\Gamma \vdash \mathcal{T} : \mathcal{T}} \textbf{ type}$$

$$\frac{}{\Gamma, x : T \vdash x : T} \textbf{ var}$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \quad \Gamma \vdash T_1 : \mathcal{T}}{\Gamma \vdash \lambda x.e : (x : T_1) \to T_2} \textbf{ lambda}$$

Typechecking rules

$$\frac{}{\Gamma \vdash \mathcal{T} : \mathcal{T}}\ \textbf{type}$$

$$\frac{}{\Gamma, x : T \vdash x : T}\ \textbf{var}$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \quad \Gamma \vdash T_1 : \mathcal{T}}{\Gamma \vdash \lambda x.e : (x : T_1) \to T_2}\ \textbf{lambda}$$

$$\frac{\Gamma \vdash T_1 : \mathcal{T} \quad \Gamma, x : T_1 \vdash T_2 : \mathcal{T}}{\Gamma \vdash (x : T_1) \to T_2 : \mathcal{T}}\ \textbf{pi}$$

Typechecking rules

$$\frac{}{\Gamma \vdash \mathcal{T} : \mathcal{T}}\ \textbf{type}$$

$$\frac{}{\Gamma, x : T \vdash x : T}\ \textbf{var}$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \quad \Gamma \vdash T_1 : \mathcal{T}}{\Gamma \vdash \lambda x.e : (x : T_1) \to T_2}\ \textbf{lambda}$$

$$\frac{\Gamma \vdash T_1 : \mathcal{T} \quad \Gamma, x : T_1 \vdash T_2 : \mathcal{T}}{\Gamma \vdash (x : T_1) \to T_2 : \mathcal{T}}\ \textbf{pi}$$

$$\frac{\Gamma \vdash e_1 : (x : T_1) \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2\,[x \mapsto e_2]}\ \textbf{app}$$

Examples

$$id : (t : \mathcal{T}) \to (x : t) \to t$$

$$id = \lambda t.\lambda x.x$$

Examples

$$id : (t : \mathcal{T}) \to (x : t) \to t$$

$$id = \lambda t.\lambda x.x$$

$$idid : (t : \mathcal{T}) \to (x : t) \to t$$

$$idid = id\ ((t : \mathcal{T}) \to (x : t) \to t)\ id$$

Examples

$$bool : \mathcal{T}$$

$$bool = (t : \mathcal{T}) \to (b_t : t) \to (b_f : t) \to t$$

Examples

$$bool : \mathcal{T}$$

$$bool = (t : \mathcal{T}) \to (b_t : t) \to (b_f : t) \to t$$

$$true : bool$$

$$true = \lambda t.\lambda b_t.\lambda b_f.b_t$$

$$false : bool$$

$$false = \lambda t.\lambda b_t.\lambda b_f.b_f$$

Examples

$bool : \mathcal{T}$

$bool = (t : \mathcal{T}) \to (b_t : t) \to (b_f : t) \to t$

$true : bool$

$true = \lambda t.\lambda b_t.\lambda b_f.b_t$

$false : bool$

$false = \lambda t.\lambda b_t.\lambda b_f.b_f$

$if : bool \to (t : T) \to (b_t : t) \to (b_f : t) \to t$

$if = \lambda b.b$

References

- Idris language
  - tutorial: `http://docs.idris-lang.org/en/latest/tutorial/index.html`
  - FAQ: `https://github.com/idris-lang/Idris-dev/wiki/Unofficial-FAQ`
- "*Pi-Forall: How to use and implement a dependently-typed language*", Stephanie Weirich (Compose Conference)
  - video: `https://www.youtube.com/watch?v=6klfKLBnz9k`
  - notes: `https://github.com/sweirich/pi-forall`
- "*Idris: Practical Dependent Types with Practical Examples*", Brian McKenna (Strange Loop 2014)
  - video: `https://www.youtube.com/watch?v=4i7KrG1Afbk`