Towards Being Positively Negative About Dependent Types

TYPES 2025

Jan de Muijnck-Hughes

jfdm@discuss.systems https://tyde.systems

University of Strathclyde Science

MSP && StrathCyber

We need to talk about runtime failure!

```
testAndPrint : List Nat -> IO ()
testAndPrint ns = case all isZero ns of
                    Yes prf => putStrLn "All Zero"
                    No  why => putStrLn "Some Non Zero"
```

We need to talk about runtime failure!

```
testAndPrint : List Nat -> IO ()
testAndPrint ns = case all isZero ns of
                    Yes prf => putStrLn "All Zero"
                    No  why => putStrLn "Some Non Zero"
```

```
all : (f : (x : a) -> Dec (p x)) -> (xs : List a) -> Dec (All p xs)
isZero : (n : Nat) -> Dec (IsZero n)
```

We need to talk about runtime failure!

```
testAndPrint : List Nat -> IO ()
testAndPrint ns = case all isZero ns of
                    Yes prf => putStrLn "All Zero"
                    No  why => putStrLn "Some Non Zero"
```

```
> all isZero [0,0,0,0]
Yes [IZ,IZ,IZ,IZ]

> all isZero [0,0,1,0]
No (\lamc => let p :: _ = lamc in absurd p)
```

```
all : (f : (x : a) -> Dec (p x)) -> (xs : List a) -> Dec (All p xs)
isZero : (n : Nat) -> Dec (IsZero n)
```

We need to talk about runtime failure!

```
testAndPrint : List Nat -> IO ()
testAndPrint ns = case all isZero ns of
                    Yes prf => putStrLn "All Zero"
                    No  why => putStrLn "Some Non Zero"
```

```
> all isZero [0,0,0,0]
Yes [IZ,IZ,IZ,IZ]

> all isZero [0,0,1,0]
No (\lamc => let p :: _ = lamc in absurd p)
```

```
data Dec : (a : Type) -> Type where
  Yes : (prf    : a)         -> Dec a
  No  : (contra : a -> Void) -> Dec a
```

```
all : (f : (x : a) -> Dec (p x)) -> (xs : List a) -> Dec (All p xs)
isZero : (n : Nat) -> Dec (IsZero n)
```

'Two' Positive Actions can lead to Nothing

```
record Decidable where
  constructor D
  Positive : Type
  Negative : Type
  0 Cancelled : Positive -> Negative -> Void
```

'Two' Positive Actions can lead to Nothing

```
record Decidable where
  constructor D
  Positive : Type
  Negative : Type
  0 Cancelled : Positive -> Negative -> Void
```

```
Dec : Decidable -> Type
Dec d = Either (Negative d)
               (Positive d)
```

'Two' Positive Actions can lead to Nothing

```
record Decidable where
  constructor D
  Positive : Type
  Negative : Type
  0 Cancelled : Positive -> Negative -> Void
```

```
Dec : Decidable -> Type
Dec d = Either (Negative d)
               (Positive d)
```

Robert Atkey. *Data Types with Negation*. Ninth Workshop on Mathematically Structured Functional Programming. Extended Abstract (Talk Only). 2nd Apr. 2022. URL: https://youtu.be/mZZjOKWCF4A

Example: Shape of Natural Numbers

```
ISZERO : (n : Nat) -> Decidable
ISZERO n = D (IsZero n) (NonZero n) prf
```

Example: Shape of Natural Numbers

```
ISZERO : (n : Nat) -> Decidable
ISZERO n = D (IsZero n) (NonZero n) prf
```

```
data IsZero : (n : Nat) -> Type where
  IZ : IsZero Z

data NonZero : (n : Nat) -> Type where
  NZ : NonZero (S n)
```

```
prf : IsZero n -> NonZero n -> Void
prf IZ NZ impossible
```

Example: Shape of Natural Numbers

```
ISZERO : (n : Nat) -> Decidable
ISZERO n = D (IsZero n) (NonZero n) prf
```

```
data IsZero : (n : Nat) -> Type where
  IZ : IsZero Z

data NonZero : (n : Nat) -> Type where
  NZ : NonZero (S n)
```

```
isZero : (n : Nat) -> Dec (ISZERO n)
isZero    Z   = Right IZ
isZero (S k) = Left  NZ
```

```
prf : IsZero n -> NonZero n -> Void
prf IZ NZ impossible
```

Example: Shape of Natural Numbers

```
ISZERO : (n : Nat) -> Decidable
ISZERO n = D (IsZero n) (NonZero n) prf
```

```
data IsZero : (n : Nat) -> Type where
  IZ : IsZero Z

data NonZero : (n : Nat) -> Type where
  NZ : NonZero (S n)
```

```
prf : IsZero n -> NonZero n -> Void
prf IZ NZ impossible
```

```
isZero : (n : Nat) -> Dec (ISZERO n)
isZero    Z   = Right IZ
isZero (S k) = Left  NZ
```

Reuse Definitions

```
nonZero : (n : Nat)
            -> Dec (Mirror (ISZERO n))
nonZero = (mirror . isZero)
```

Motivating Example Revisited

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Left  why => printLn why
      Right prf => putStrLn "All Zero"
```

Motivating Example Revisited

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Left  why => printLn why
      Right prf => putStrLn "All Zero"
```

Before:

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Yes prf => putStrLn "All Zero"
      No  why => putStrLn "Some Non Zero"
```

Motivating Example Revisited

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Left  why => printLn why
      Right prf => putStrLn "All Zero"
```

```
> all isZero [0,0,0,0]
Right [IZ,IZ,IZ,IZ]

> all isZero [0,0,1,0]
Left (There IZ (There IZ (Here NZ)))
```

Before:

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Yes prf => putStrLn "All Zero"
      No  why => putStrLn "Some Non Zero"
```

Motivating Example Revisited

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Left  why => printLn why
      Right prf => putStrLn "All Zero"
```

```
> all isZero [0,0,0,0]
Right [IZ,IZ,IZ,IZ]

> all isZero [0,0,1,0]
Left (There IZ (There IZ (Here NZ)))
```

Before:

```
testAndPrint : List Nat -> IO ()
testAndPrint ns
  = case all isZero ns of
      Yes prf => putStrLn "All Zero"
      No  why => putStrLn "Some Non Zero"
```

```
> all isZero [0,0,0,0]
Yes [IZ,IZ,IZ,IZ]

> all isZero [0,0,1,0]
No (\lamc => let p :: _ = lamc in
               absurd p)
```

On Decidable Equality

```
interface DecEQ type where

  EQUAL  : (x,y : type) -> Decidable

  toRefl : {x,y : type} -> Positive (EQUAL x y) -> x === y

  toVoid : {x,y : type} -> Negative (EQUAL x y) -> x === y -> Void

  decEq  : (x,y : type) -> Dec (EQUAL x y)

  refl   : (x   : type) -> Positive (EQUAL x x)
```

On Decidable Equality

```
interface DecEQ type where

  EQUAL  : (x,y : type) -> Decidable

  toRefl : {x,y : type} -> Positive (EQUAL x y) -> x === y

  toVoid : {x,y : type} -> Negative (EQUAL x y) -> x === y -> Void

  decEq  : (x,y : type) -> Dec (EQUAL x y)

  refl   : (x   : type) -> Positive (EQUAL x x)
```

A Better decEq?

```
decEq' : DecEQ type => (x,y : type) -> Either (Negative (EQUAL x y)) (x === y)
decEq' x y = either Left (Right . toRefl) (Positive.decEq x y)
```

Slowly Embracing (Positive) Negativity: In Progress

Library of (Positive) Decisions

`https://github.com/jfdm/positively-negative/`

→ **Thought Required** →

Base Decisions

Primitives

- Nats, Strings...

Datatypes

- Pairs, Lists, Trees...

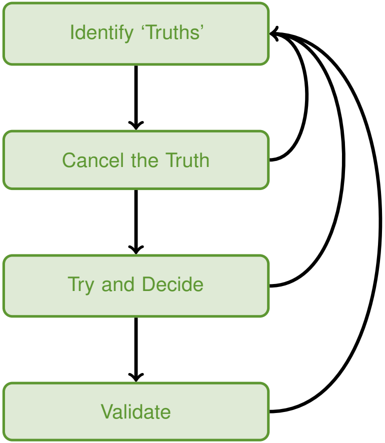
Use in ~~Programs~~ Elaborators

- Well-Scoped STLC
- Intrinsically-Typed STLC
- Efficient De Bruijn indicies
- ...

BIG Decisions (Early)

- Duality of Binary Sessions
  (No Choice)
- Typing STLC
- BiDi Type Inference
- ...

A Recipe for Decidability



1. Identify 'Truths':
   - What's Positive? (**Easy**)
   - What's Negative? (**Hard**)
2. Cancel the Truth
   - Build proof of void;
3. Try and Decide
   - Do they lead to decidability?
4. Validate
   - Check that it works as intended!

Mirroring: Why Validation!

```
data Holds : (p : (x : ty) -> Decidable)
           -> (x : ty)
             -> Type
  where
    Yes : Positive (p x) -> Holds p x
```

```
data HoldsNot : (p : (x : ty) -> Decidable)
              -> (x : ty)
                -> Type
  where
    No : Negative (p x) -> HoldsNot p x
```

Mirroring: Why Validation!

Negative becomes Positive but is still 'Negative'

```
HOLDSNOT p x = Mirror (HOLDS p x)
             = Mirror (D (Holds    p x) (HoldsNot p x)  prf)
             =        (D (HoldsNot p x) (Holds    p x)) prf')
```

```
data Holds : (p : (x : ty) -> Decidable)
          -> (x : ty)
            -> Type
  where
    Yes : Positive (p x) -> Holds p x
```

```
data HoldsNot : (p : (x : ty) -> Decidable)
             -> (x : ty)
               -> Type
  where
    No : Negative (p x) -> HoldsNot p x
```

Mirroring: Why Validation!

Negative becomes Positive but is still ‘Negative’

```
HOLDSNOT p x = Mirror (HOLDS p x)
             = Mirror (D (Holds    p x) (HoldsNot p x)  prf)
             =        (D (HoldsNot p x) (Holds    p x)) prf')
```

Ideally, call `Mirror` on `p`, but want to swap polarity on entire predicate.

```
data Holds : (p : (x : ty) -> Decidable)
          -> (x : ty)
            -> Type
  where
    Yes : Positive (p x) -> Holds p x
```

```
data HoldsNot : (p : (x : ty) -> Decidable)
             -> (x : ty)
               -> Type
  where
    No : Negative (p x) -> HoldsNot p x
```

Mirroring: Why Validation!

Generic Definition

```
data Holdable : (p   : (x : type) -> Decidable)
             -> (get : Decidable -> Type)
             -> (x   : type)
                     -> Type
  where
    H : proj (p x) -> Holdable p proj x
```

Mirroring: Why Validation!

Generic Definition

```
data Holdable : (p   : (x : type) -> Decidable)
             -> (get : Decidable -> Type)
             -> (x   : type)
                     -> Type
  where
    H : proj (p x) -> Holdable p proj x
```

Swap by Hand?

```
HOLDS p x = D (Holdable p Positive x)
              (Holdable p Negative x)
              prf'

HOLDSNOT p x = D (Holdable p Negative x)
                 (Holdable p Positive x)
                  prf
```

Mirroring: Why Validation!

How best to define and realise Mirroring?

```
HOLDSNOT p x = D (Holdable (Mirror p) Positive x) (Holdable (Mirror p) Negative x) prf
```

Generic Definition

```
data Holdable : (p   : (x : type) -> Decidable)
             -> (get : Decidable -> Type)
             -> (x   : type)
                     -> Type
  where
    H : proj (p x) -> Holdable p proj x
```

Swap by Hand?

```
HOLDS p x = D (Holdable p Positive x)
              (Holdable p Negative x)
              prf'

HOLDSNOT p x = D (Holdable p Negative x)
                 (Holdable p Positive x)
                 prf
```

Backporting: Include Informative Error Messages

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

↓

```
data Dec a = No (Not a) | Yes a
```

Backporting: Include Informative Error Messages

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

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

```
data Dec a = No (Not a) | Yes a
```

Backporting: Include Informative Error Messages

```
data Maybe a = Nothing | Just a
data Either e a = Left e | Right a
```

```
data Dec a = No (Not a) | Yes a
data Dec e a = No e (Not a) | Yes a
```

Backporting: Include Informative Error Messages

```
data Maybe a = Nothing | Just a        →  data Either e a = Left e | Right a
        ↓                                          ↓
data Dec a = No (Not a) | Yes a        →  data Dec e a = No e (Not a) | Yes a
```

- Less principled
- Fantastic Error Messages

```
all : (f  : (x : a) -> Dec e (p x))
   -> (xs : List a)
         -> Dec (AllNot e p xs)
                (All      p xs)
```

```
Dec : Type -> Type
Dec = Dec ()

No : Not a -> Dec a
No = No ()
```

Concluding Remarks

- **Dependently Typed Programmes Fail**
  - Users need Negativity
  - Proofs & Programs as well
- **Being Positively Negative helps**
  - Tricky to do well
  - easy to be happy; harder to be negative
- **Work in progress**
  - Limits of approach…
  - Design patterns…
  - More principled mirroring?

https://www.re-origin.com/articles/turning-negatives-into-positives