A Taste of Linear

~~Logic~~

~~Haskell~~

Optics

Edward Kmett
BX2021

Linear Logic for Constructive Mathematics

Mathematics > Logic

[Submitted on 19 May 2018]

Linear logic for constructive mathematics

Michael Shulman

We show that numerous distinctive concepts of constructive mathematics arise automatically from an interpretation of "linear higher-order logic" into intuitionistic higher-order logic via a Chu construction. This includes apartness relations, complemented subsets, anti-subgroups and anti-ideals, strict and non-strict order pairs, cut-valued metrics, and apartness spaces. We also explain the constructive bifurcation of classical concepts using the choice between multiplicative and additive linear connectives. Linear logic thus systematically "constructivizes" classical definitions and deals automatically with the resulting bookkeeping, and could potentially be used directly as a basis for constructive mathematics in place of intuitionistic logic.

Comments: 39 pages
Subjects: **Logic (math.LO)**
Cite as: arXiv:1805.07518 **[math.LO]**
(or arXiv:1805.07518v1 **[math.LO]** for this version)

Submission history

From: Michael Shulman [view email]
[v1] Sat, 19 May 2018 05:41:18 UTC (78 KB)

LINEAR LOGIC FOR CONSTRUCTIVE MATHEMATICS

MICHAEL SHULMAN

Abstract. We show that numerous distinctive concepts of constructive mathematics arise automatically from an interpretation of "linear higher-order logic" into intuitionistic higher-order logic via a Chu construction. This includes apartness relations, complemented subsets, anti-subgroups and anti-ideals, strict and non-strict order pairs, cut-valued metrics, and apartness spaces. We also explain the constructive bifurcation of classical concepts using the choice between multiplicative and additive linear connectives. Linear logic thus systematically "constructivizes" classical definitions and deals automatically with the resulting bookkeeping, and could potentially be used directly as a basis for constructive mathematics in place of intuitionistic logic.

1. Introduction

One of the explicit motivations of Girard's linear logic [Gir87] was to recover an involutory "classical" negation while retaining "constructive content":

> . . . the linear negation . . . is a constructive and involutive negation; by the way, linear logic works in a classical framework, while being more constructive than intuitionistic logic. [Gir87, p3]

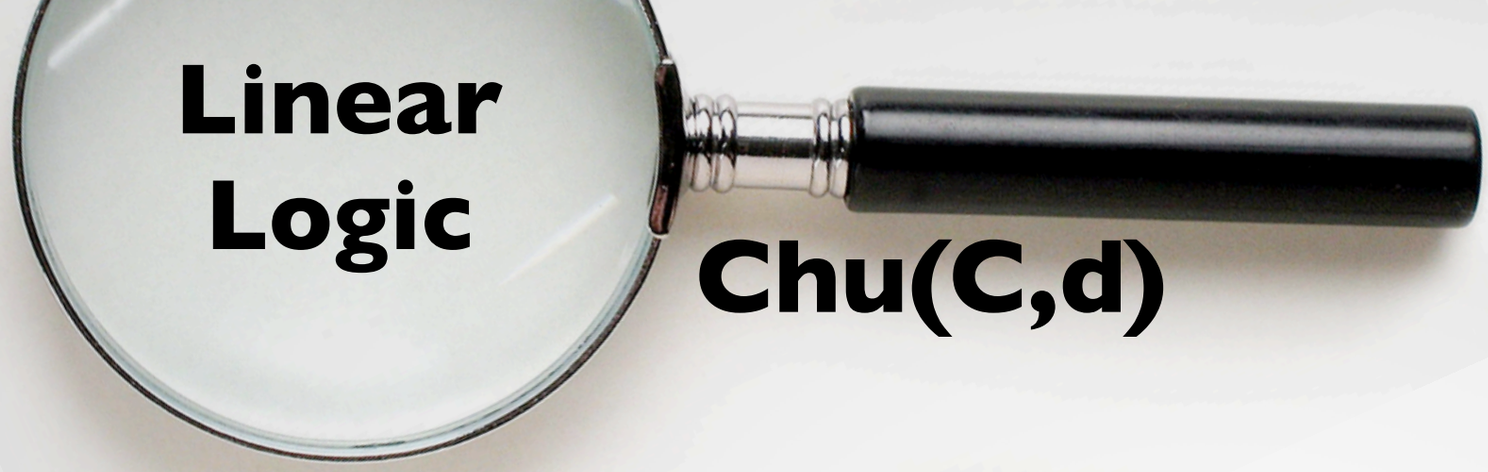
One might therefore expect that over the past three decades some practicing constructive mathematicians would have adopted linear logic instead of intuitionistic logic;[1] but this does not seem to be the case. One might conjecture many reasons for this. However, I will instead argue that constructive mathematicians (going back all the way to Brouwer) *have* been using linear logic without realizing it!

Of course I don't mean this literally; what I mean is that there are aspects of constructive mathematical practice that are better explained by linear logic than by intuitionistic logic. Specifically, the non-involutory nature of intuitionistic negation often leads constructive mathematicians to study both a classical concept and its formal de Morgan dual, such as equality and apartness, subgroups and antisubgroups, topological spaces and apartness spaces, and so on. We will show that such

Date: May 22, 2018.
This material is based on research sponsored by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053 and and FA9550-16-1-0292. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government, or Carnegie Mellon University.

[1] I will use "intuitionistic" to refer to the formal logic codified by Heyting, and "constructive" for the programme of doing mathematics with "constructive content". This is unfaithful to the original philosophical meaning of "intuitionistic", but for better or for worse the phrase "intuitionistic logic" has come to refer to Heyting's logic, and I have been unable to think of a satisfactory alternative. I will use "classical" to refer to classical mathematics and classical nonlinear logic, and "linear" to refer to the "classical" form of linear logic with an involutive negation.


There is a certain
logic to your logic.

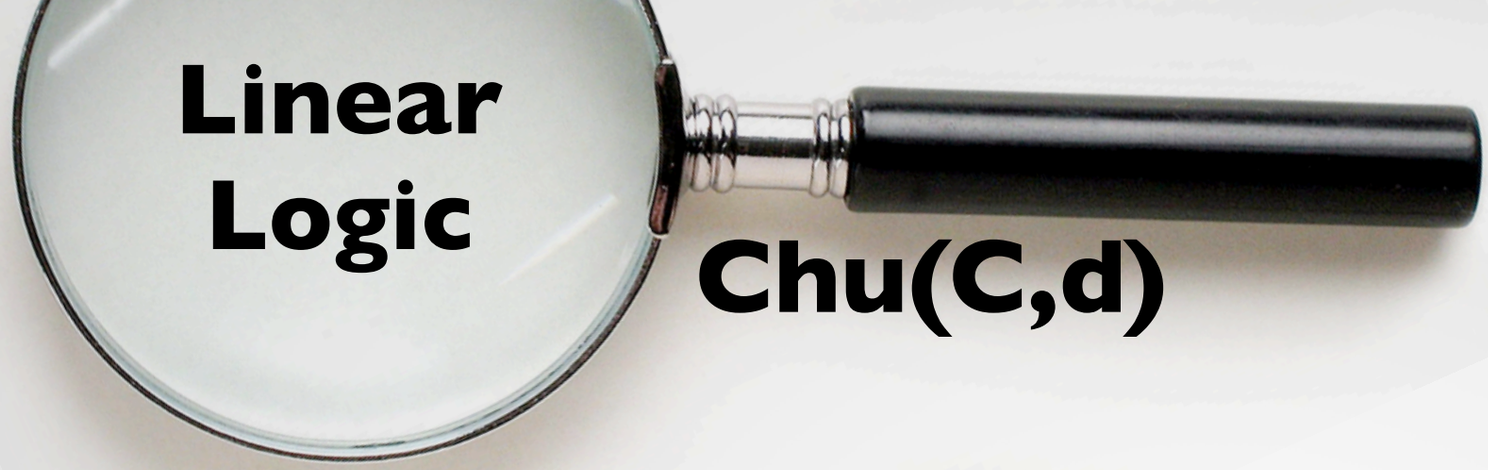


Objects in **Chu(C,d)** are:

(a, b : C, f: a * b -> d)

Morphisms from (a,b,r) to (x,y,s) are pairs of morphisms:
(f : a -> x, g : y -> b) such that

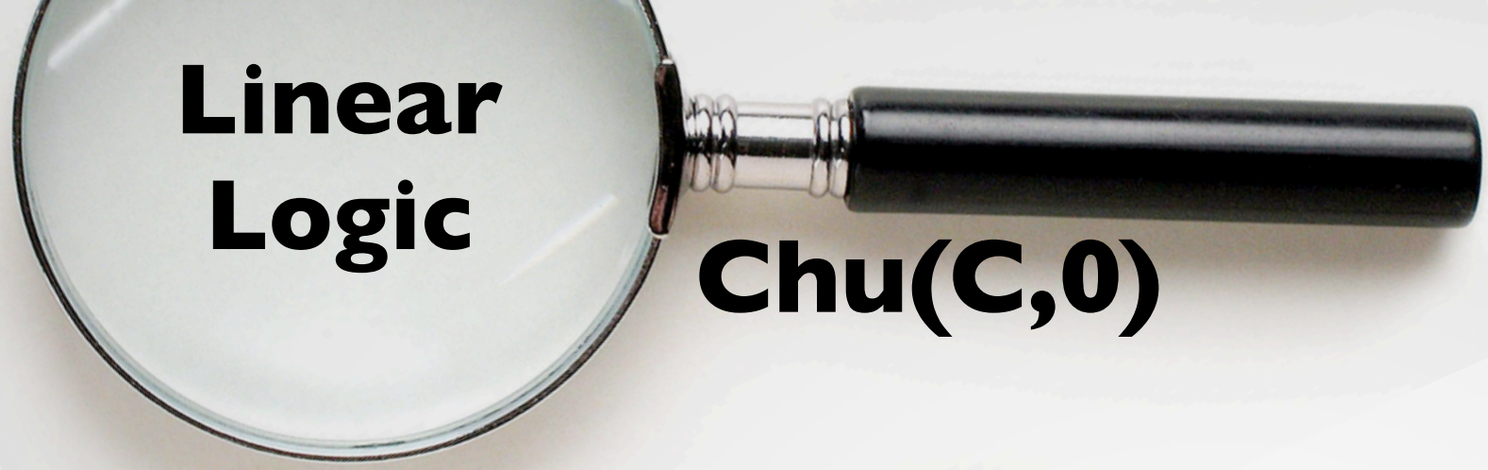
$$\begin{array}{ccc} a \otimes y & \xrightarrow{1_a \otimes g} & a \otimes b \\ f \otimes 1_y \downarrow & & \downarrow r \\ x \otimes y & \xrightarrow{s} & d \end{array}$$



Objects in **Chu(C,d)** are:

(a, b : C, f: a * b -> d)

Chu(C,d) is self-dual (swap a and b, flip the args to f)



Objects in **Chu(C,0)** are:

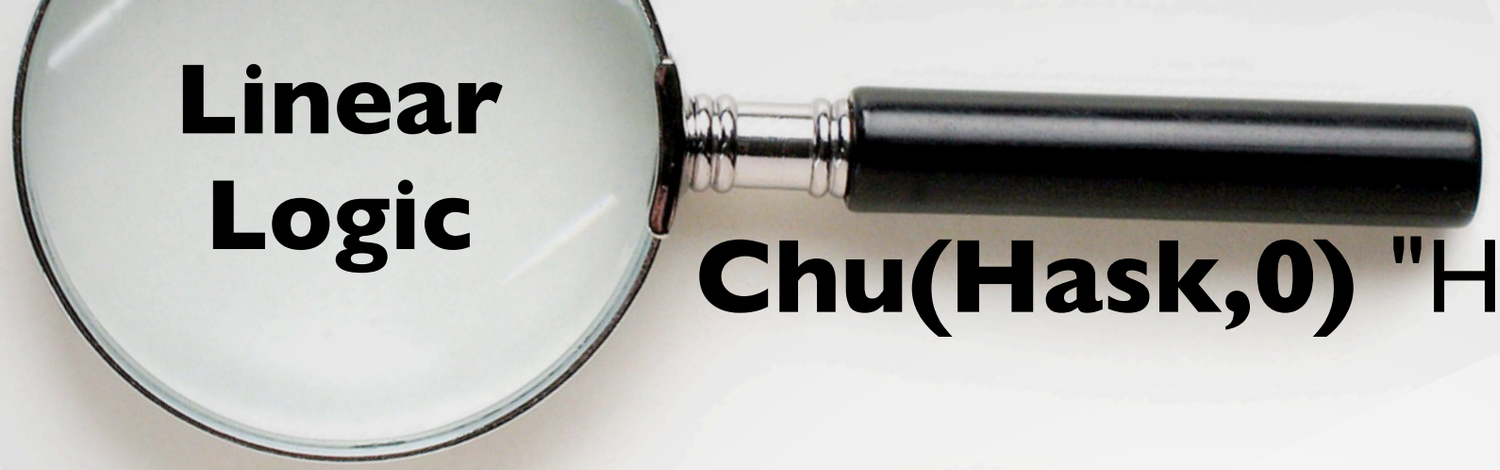
(a, b : C, f: a * b -> 0)

Morphisms from (a,b,r) to (x,y,s) are pairs of morphisms:
(f : a -> x, g : y -> b) such that

$$\begin{array}{ccc} a \otimes y & \xrightarrow{1_a \otimes g} & a \otimes b \\ f \otimes 1_y \downarrow & & \downarrow r \\ x \otimes y & \xrightarrow{s} & d \end{array}$$

Linear Logic for Constructive Mathematics

Connective	Proof	Refutation
A & B	(A^+, B^+)	$A^- + B^-$
A + B	$A^+ + B^+$	(A^-, B^-)
A * B	(A^+, B^+)	$(B^+ \to A^-, A^+ \to B^-)$
A ⅋ B	$(B^- \to A^+, A^- \to B^+)$	(A^-, B^-)



Chu(Hask,0) "Haskell Style"

```haskell
class
  ( Prop (Not a)
  , Not (Not a) ~ a
  ) => Prop a where
  type Not a = c | c -> a
  (!=) :: a -> Not a -> r
```

```haskell
data a & b = With a b

type (+) = Either

instance (Prop a, Prop b) => Prop (a & b) where
  type Not (a & b) = Not a + Not b
  With a _b != Left  na = a != not_a
  With _a b != Right nb = a != not_b

instance (Prop a, Prop b) => Prop (a + b) where
  type Not (a + b) = Not a & Not b
  Left  a != With na _nb  = a != not_a
  Right b != With _na nb = a != not_b
```

```haskell
type (*) = (,)

data a ⅋ b = Par (Not b -> a) (Not a -> b)

instance (Prop a, Prop b) => Prop (a * b) where
  type Not (a * b) = Not a ⅋ Not b
  (a, b) != Par a2nb _b2na = b != a2nb a

instance (Prop a, Prop b) => Prop (a ⅋ b) where
  type Not (a ⅋ b) = Not a * Not b
  Par na2b _nb2a != (na, nb) = na2b na != nb
```

Linear Logic for Constructive Mathematics

Limitations of Shulman's construction:

It only builds an "Affine Logic". You can't prove that there are actually four distinct units, 1, 0, T, and ⊥ with this encoding.

It readily permits (nay encourages) constructions like weakening from A * B to A & B, because they happen to share proofs in this particular encoding, but & requires a stronger refutation.

What is Linear Haskell?

Computer Science > Programming Languages

[Submitted on 26 Oct 2017 (v1), last revised 8 Nov 2017 (this version, v2)]

Linear Haskell: practical linearity in a higher-order polymorphic language

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, Arnaud Spiwack

Linear type systems have a long and storied history, but not a clear path forward to integrate with existing languages such as OCaml or Haskell. In this paper, we study a linear type system designed with two crucial properties in mind: backwards-compatibility and code reuse across linear and non-linear users of a library. Only then can the benefits of linear types permeate conventional functional programming. Rather than bifurcate types into linear and non-linear counterparts, we instead attach linearity to function arrows. Linear functions can receive inputs from linearly-bound values, but can also operate over unrestricted, regular values.

To demonstrate the efficacy of our linear type system – both how easy it can be integrated in an existing language implementation and how streamlined it makes it to write programs with linear types – we implemented our type system in GHC, the leading Haskell compiler, and demonstrate two kinds of applications of linear types: mutable data with pure interfaces; and enforcing protocols in I/O-performing functions.

Subjects: **Programming Languages (cs.PL)**
DOI: 10.1145/3158093
Cite as: arXiv:1710.09756 **[cs.PL]**
(or arXiv:1710.09756v2 **[cs.PL]** for this version)

Submission history

From: Arnaud Spiwack [view email]
[v1] Thu, 26 Oct 2017 15:28:32 UTC (115 KB)
[v2] Wed, 8 Nov 2017 17:07:41 UTC (129 KB)

A Taste of Linear Haskell

Linear Haskell adds a "linear arrow" to the existing Haskell type system, and then tries to retrofit the existing rules of Haskell's type inference to work well with this concept.

An arrow

```
f :: a %1 -> b
```

says that in order to *consume* the result of type 'b' once, you will *consume* 'a' exactly once.

A Taste of Linear Haskell

Can we use Linear Haskell for Linear Logic?

```
(,) :: a %1 -> b %1 -> (a, b)
```

We've got (*)! (multiplicative conjunction)

```
Left :: a %1 -> a + b
Right :: b %1 -> a + b
```

We've got (+)! (additive disjunction)

A Taste of Linear Haskell

Can we use Linear Haskell for Linear Logic?

```
data Y a b c where
  L :: Y a b a
  R :: Y a b b

newtype a & b = With { runWith :: forall c. Y a b c -> c }

with :: (forall c. Y a b c -> c) %1 -> a & b
with = With

withL :: a & b %1 -> a
withL p = runWith p L

withR :: a & b %1 -> b
withR p = runWith p R
```

```
with \case
  L -> ...
  R -> ...
```

```
Γ |- a
Γ |- b
---------- with
Γ |- a & b
```

```
Γ |- a & b
---------- withL
Γ |- a
```

```
Γ |- a & b
---------- withR
Γ |- b
```

We've got (&)! (additive conjunction)

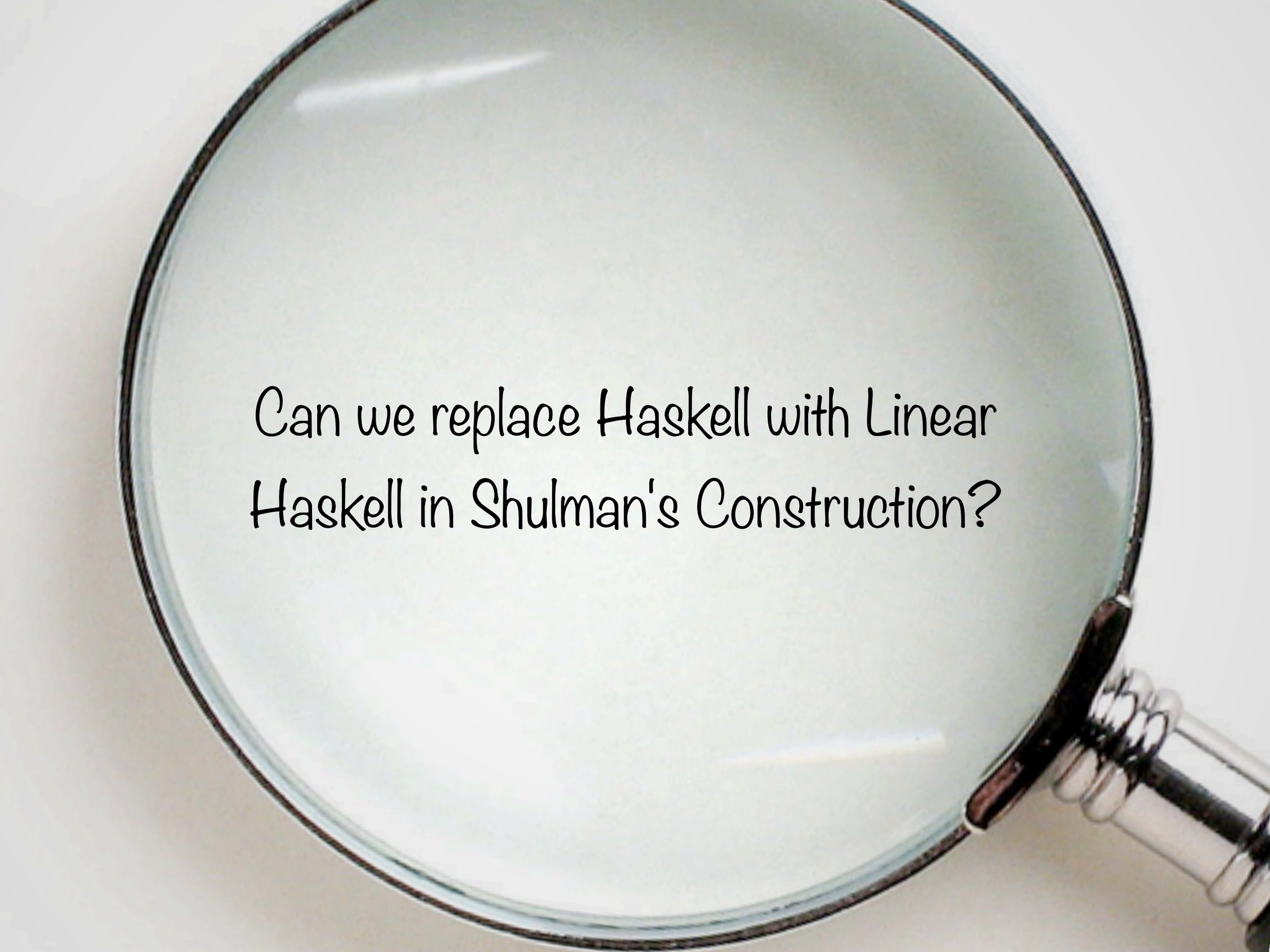
A Taste of Linear Haskell

Can we use Linear Haskell for Linear Logic?

Kind of.

We only have the "proof" fragment, not the refutations.

We don't get "par" (⅋), which is traditionally limited to the domain of "classical" linear logic.


Can we replace Haskell with Linear
Haskell in Shulman's Construction?

Linear ~~Logic~~ for Constructive Mathematics Haskell

Connective	Proof	Refutation
A & B	A^+ & B^+	A^- + B^-
A + B	A^+ + B^+	A^- & B^-
A * B	(A^+, B^+)	(B^+ %1-> A^-) & (A^+ %1-> B^-)
A ⅋ B	(B^- %1-> A^+) & (A^- %1-> B^+)	(A^-, B^-)

Linear Logic

Full Intuitionistic Linear Logic

Technical Report

UCAM-CL-TR-213
ISSN 1476-2986

Number 213

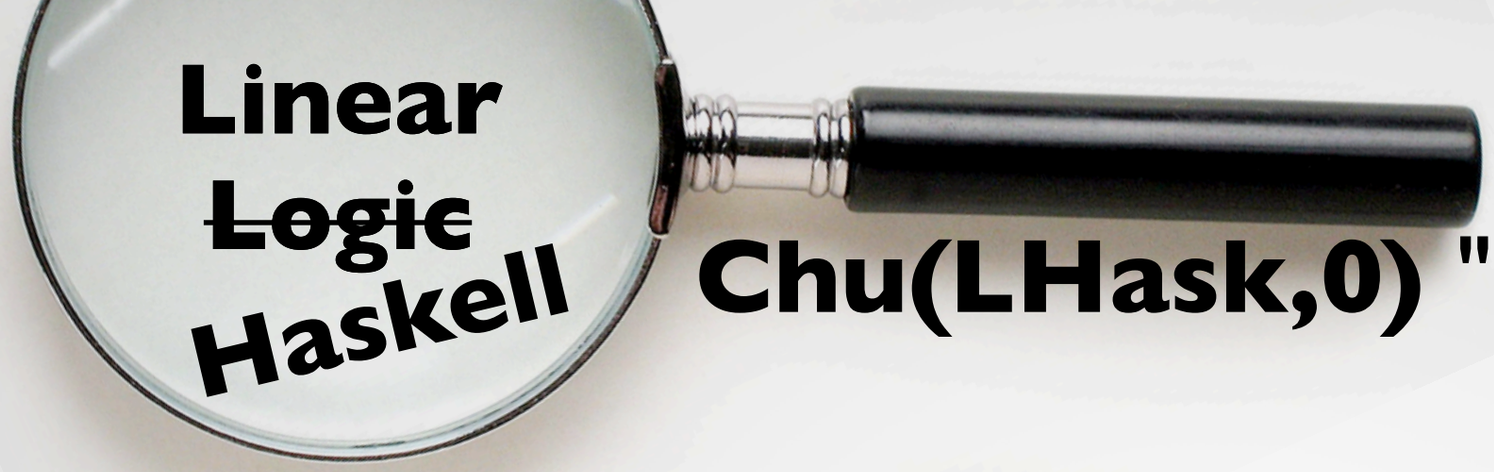
UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The Dialectica categories

Valeria Correa Vaz de Paiva

January 1991

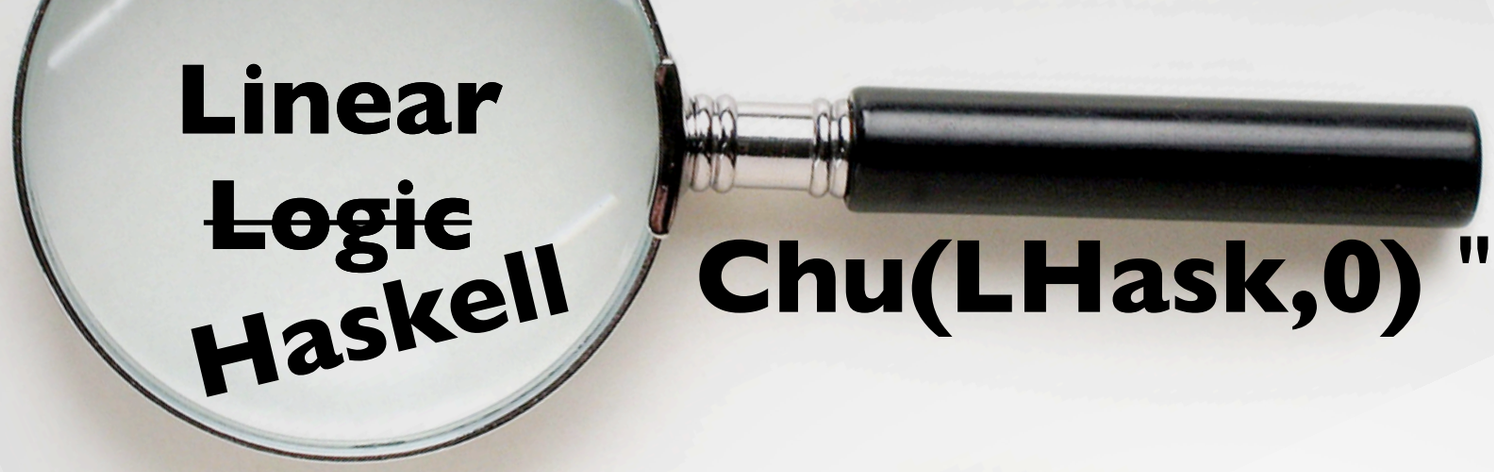
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
http://www.cl.cam.ac.uk/



Chu(LHask,0) "Haskell Style"

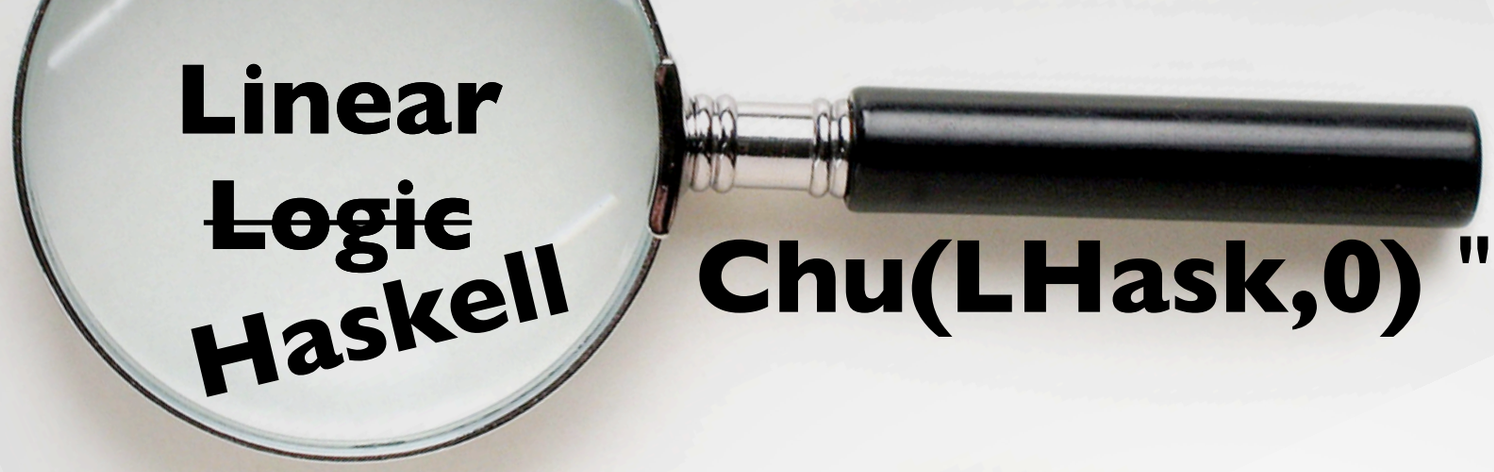
```
class
  ( Prop (Not a)
  , Not (Not a) ~ a
  ) => Prop a where
  type Not a = c | c -> a
  (!=) :: a %1 -> Not a %1 -> r
```

```
type (+) = Either

instance (Prop a, Prop b) => Prop (a & b) where
  type Not (a & b) = Not a + Not b
  awb != Left  na = withL awb != na
  awb != Right nb = withR awb != nb

instance (Prop a, Prop b) => Prop (a + b) where
  type Not (a + b) = Not a & Not b
  Left  a != nawnb = a != withL nawnb
  Right b != nawnb = a != withR nawnb
```



Chu(LHask,0) "Haskell Style"

```
type (*) = (,)


data a ⅋ b = Par
  { runPar :: forall c.
    Y (Not b %1 -> a)
      (Not a %1 -> b)
      c
    -> c
  }

par :: (forall c. Y (Not b %1 -> a) (Not a %1 -> b) c -> c) %1 -> a ⅋ b
parL :: a ⅋ b %1 -> Not b %1 -> a
parR :: a ⅋ b %1 -> Not a %1 -> b

instance (Prop a, Prop b) => Prop (a * b) where
  type Not (a * b) = Not a ⅋ Not b
  (a, b) != napnb = b != parR napnb a


instance (Prop a, Prop b) => Prop (a ⅋ b) where
  type Not (a ⅋ b) = Not a * Not b
  apb != (na, nb) = parR apb na != nb
```



Chu(LHask,0) "Haskell Style"

```
type (*) = (,)


data a ⅋ b = Par
  { runPar :: forall c.
    Y (Not b %1 -> a)
      (Not a %1 -> b)
      c
    -> c
  }

par :: (forall c. Y (Not b %1 -> a) (Not a %1 -> b) c -> c) %1 -> a ⅋ b
parL :: a ⅋ b %1 -> Not b %1 -> a
parR :: a ⅋ b %1 -> Not a %1 -> b

instance (Prop a, Prop b) => Prop (a * b) where
  type Not (a * b) = Not a ⅋ Not b
  (a, b) != napnb = b != parR napnb a


instance (Prop a, Prop b) => Prop (a ⅋ b) where
  type Not (a ⅋ b) = Not a * Not b
  apb != (na, nb) = parR apb na != nb
```

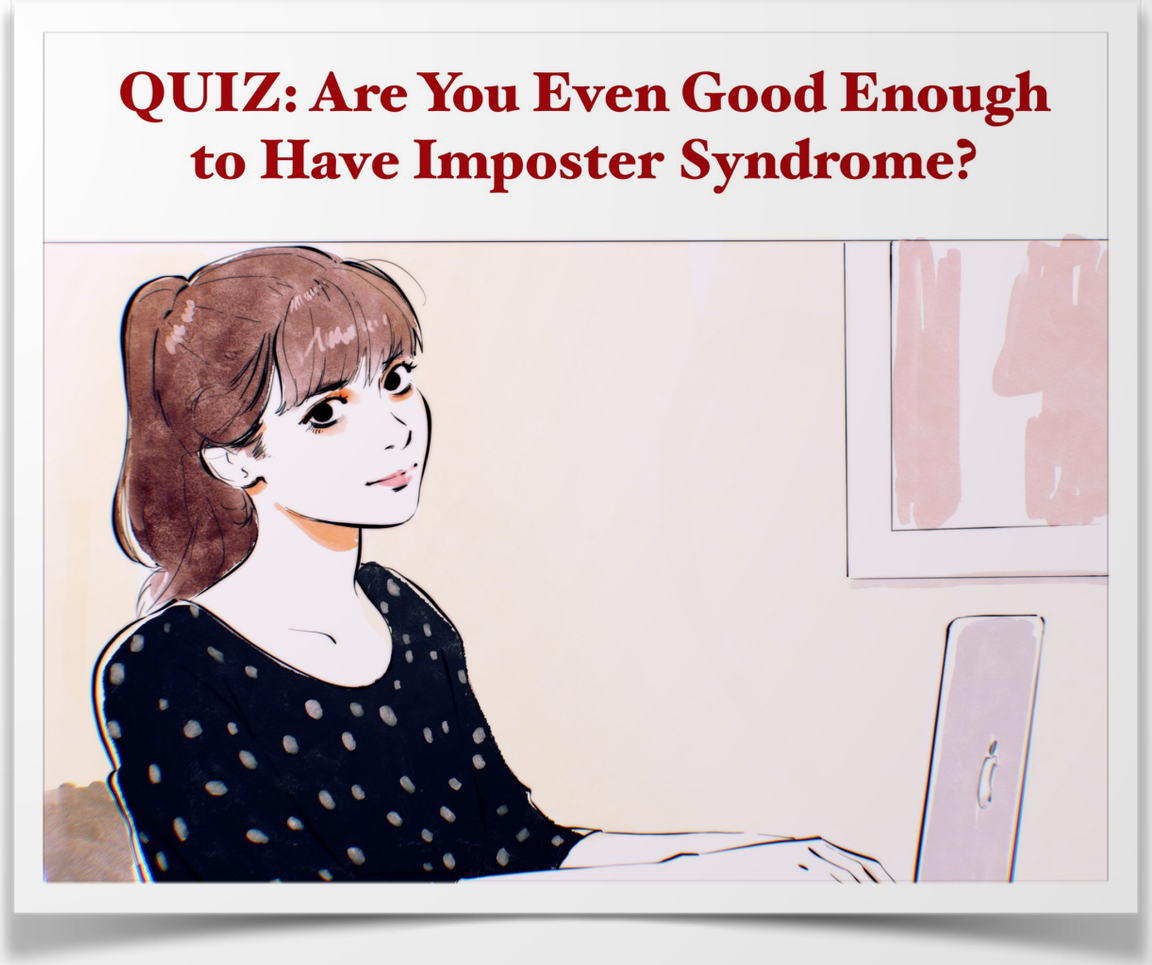
```
par \case
  L -> \nb -> ...
  R -> \na -> ...
```

Full Intuitionistic Linear Logic

With the modified Shulman construction I sketched above, we get (⅋), sort of. We're still only focusing in on one thing on the right hand side of the |-.

This *seems* to map onto a logic due to De Paiva and Hyland called "Full Intuitionistic Linear Logic" (**FILL**).

I am not a real logician, feel free to call me out on this later. I'll even give additional ammunition you can use.



Exploring the Implications

```
type a ⊸ b = Not a ⅋ b
```

?

```
type (⊸) a = (⅋) (Not a)
```

?

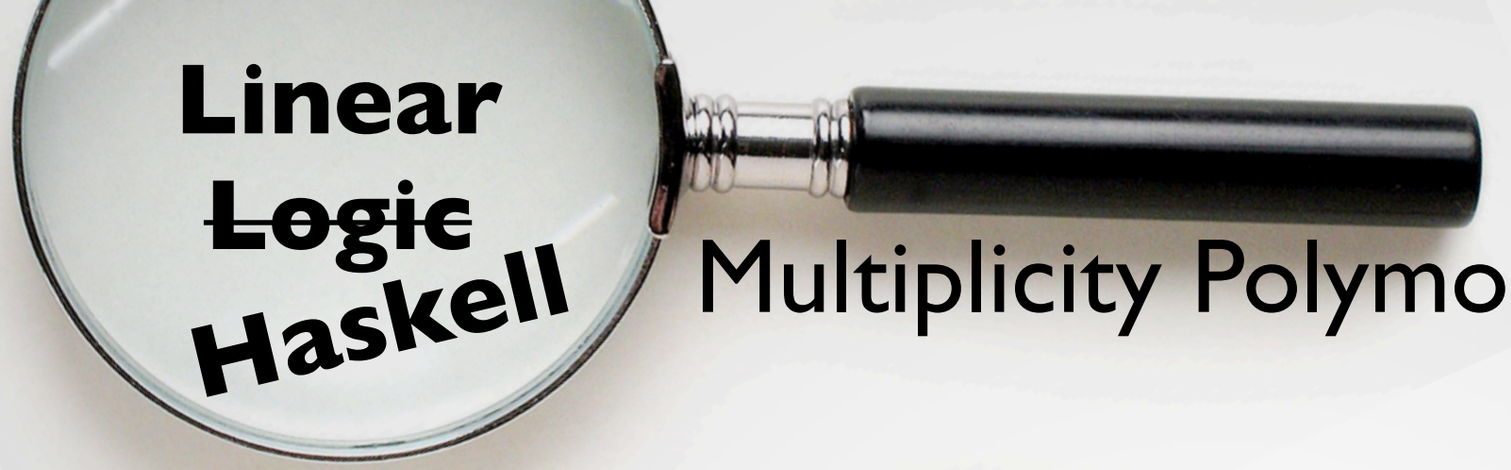
```
newtype a ⊸ b = Lol (forall c. Y (Not b %1 -> Not a) (a %1 -> b) c -> c)
```

Of course, by making a new type, we need to name its type of refutations

We also need to do this for (a %1 -> b).

Directed Apartness

```
newtype a ⊸ b = Lol
  { runLol ::
    forall c. Y (Not b %1 -> Not a) (a %1 -> b) c -> c
  }

data b <#- a where
  (:-#>) :: a %1 -> Not b %1 -> b <#- a

infixl 0 <#-
infixr 3 :-#>

instance (Prop a, Prop b) => Prop (a ⊸ b) where
  type Not (a ⊸ b) = b <#- a
  f != (a :-#> nb) = runLol f R a != nb

instance (Prop a, Prop b) => Prop (b <#- a) where
  type Not (b <#- a) = a ⊸ b
  (a :-#> nb) != f = runLol f R a != nb
```



Multiplicity Polymorphism

```haskell
-- Refuting Haskell arrows (a %m -> b)
-- m can be 'Many or 'One

data Nofun m b a where
  Nofun :: a %m -> Not b %1 -> Nofun m b a

instance Prop b => Prop (a %m -> b) where
  type Not (a %m -> b) = Nofun m b a
  f != (Nofun a nb) = f a != nb

instance Prop b => Prop (Nofun m b a) where
  type Not (Nofun m b a) = a %m -> b
  Nofun a nb != f = f a != nb
```

Isomorphisms and Apartness

```
newtype a ⊸ b = Iso
  { runIso ::
    forall c. Y (b ⊸ a) (a ⊸ b) c -> c
  }
infixr 0 ⊸
```

```
data b # a
  = ApartL (Not a) b
  | ApartR a (Not b)

instance (Prop a, Prop b) => Prop (a ⊸ b) where
  type Not (a ⊸ b) = b # a
  Iso f != ApartR a nb = f R != (a :-#> nb)
  Iso f != ApartL na b = f L != (b :-#> na)

instance (Prop a, Prop b) => Prop (b # a) where
  type Not (b # a) = a ⊸ b
  ApartR a nb != Iso f = f R != (a :-#> nb)
  ApartL na b != Iso f = f L != (b :-#> na)
```

Internalization vs. Usability

```
class
  ( forall a b. (Prop a, Prop b) => Prop (p a b)
  ) => Iso p where
  iso :: (forall c. Y (b ⊸ a) (a ⊸ b) c -> c) %1 -> p a b
  apart :: Not (p a b) %1 -> b # a

class Iso p => Lol p where
  lol :: (forall c. Y (Not b %1 -> Not a) (a %1 -> b) c -> c) %1 -> p a b
  apartR :: Not (p a b) %1 -> b <#- a
```

```
instance Iso (⊸) where
  type NotIso (⊸) = (#)
  iso = Iso
  apart = \x -> x

instance Iso (⊸) where
  type NotIso (⊸) = (<#-)
  iso f = f R
  apart (a :-#> nb) = ApartR a nb

instance Iso (FUN m) where
  type NotIso (FUN m) = Nofun m
  iso f = \x -> fun' (f R) x
  apart (Nofun a nb) = ApartR a nb
```

```
instance Lol (⊸) where
  lol = Lol
  apartR x = x

instance Lol (FUN m) where
  lol f = \x -> f R x
  apartR (Nofun a nb) = a :-#> nb
```

```
lol \case
  L -> \nb -> ...
  R -> \a -> ...
```

We Don't Have to Choose

```haskell
left :: Lol l => l a (a + b)
left = lol \case
  L -> withL'
  R -> Left

right :: Lol l => l b (a + b)
right = lol \case
  L -> withR'
  R -> Right
```

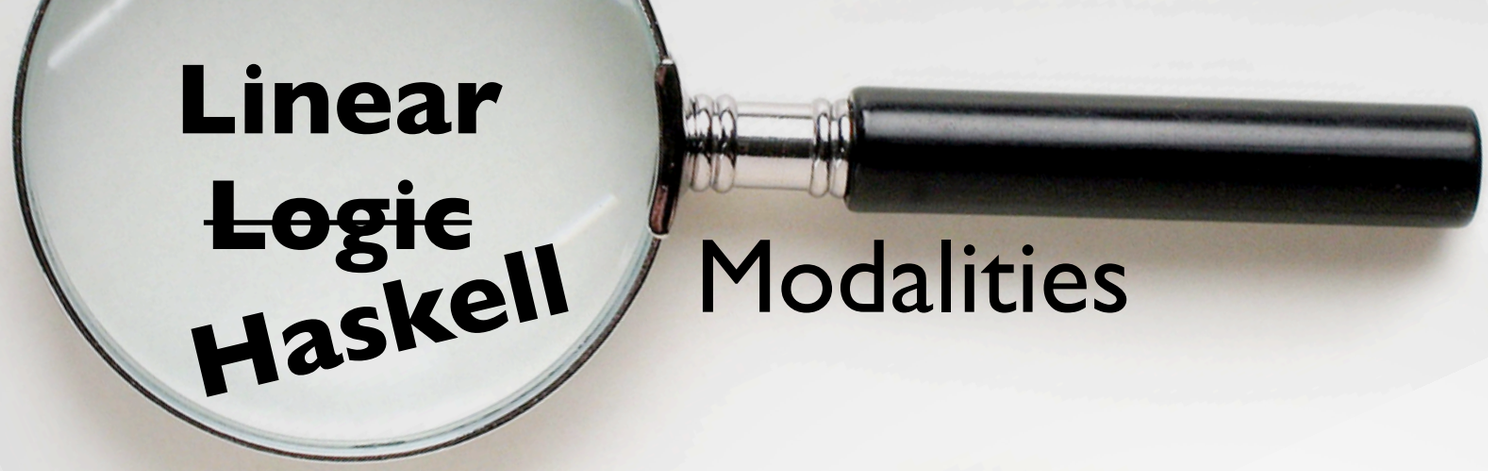
```haskell
withL :: Lol l => l (a & b) a
withL = lol \case
  L -> Left
  R -> withL'

withR :: Lol l => l (a & b) b
withR = lol \case
  L -> Right
  R -> withR'
```

```haskell
parR :: (Lol l, Lol l', Prop a) => l (a ⅋ b) (l' (Not a) b)
parR = lol \case
  L -> \g -> apartR g & \(x :-#> y) -> (x, y)
  R -> \p -> lol \case
    L -> parL' p
    R -> parR' p

parL :: (Lol l, Lol l', Prop b) => l (a ⅋ b) (l' (Not b) a)
parL = lol \case
  L -> \g -> apartR g & \(x :-#> y) -> (y, x)
  R -> \p -> lol \case
    L -> parR' p
    R -> parL' p
```

The Third Prime is the Charm

```haskell
assocEither'' :: (a + b) + c %1 -> a + (b + c)
assocEither'' = \case
  Left (Left a) -> Left a
  Left (Right b) -> Right (Left b)
  Right c -> Right (Right c)
```

```haskell
assocWith'' :: (a & b) & c %1 -> a & (b & c)
assocWith'' = \abc -> with \case
  L -> withL' (withL' abc)
  R -> with \case
    L -> withR' (withL' abc)
    R -> withR' abc

assocEither' :: Lol l => l ((a + b) + c) (a + (b + c))
assocEither' = lol \case L -> unassocWith''; R -> assocEither''

assocEither :: Iso i => i ((a + b) + c) (a + (b + c))
assocEither = iso \case L -> unassocEither'; R -> assocEither'
```



Modalities

```haskell
-- ! modality (provided by linear-base)
data Ur a where
 Ur :: a -> Ur a

instance Prop a => Prop (Ur a) where
  type Not (Ur a) = WhyNot (Not a)
  Ur a != f = because f a

instance Prop a => Prop (WhyNot a) where
  type Not (WhyNot a) = Ur (Not a)
  f != Ur a = because f a

-- ? modality
type role WhyNot nominal
newtype WhyNot a = WhyNot (forall r. Not a -> r)

whyNot :: (forall r. Not a -> r) %1 -> WhyNot a
whyNot = WhyNot

because :: WhyNot a %1 -> Not a -> r
because (WhyNot a) = a
{-# inline because #-}
```

Internalized Functors

```
class
  ( forall a. Prop a => Prop (f a)
  ) => Functor f where
  -- fmap' :: (Prop a, Prop b) => Ur (a ⊸ b) ⊸ f a ⊸ f b
  fmap' :: (Prop a, Prop b, Lol l, Lol l') => l (Ur (a ⊸ b)) (l' (f a) (f b))

fmap ::(Functor f, Prop a, Prop b, Lol l) => (a ⊸ b) -> l (f a) (f b)
fmap f = fmap' (Ur f)

instance Prop x => Functor ((⊸) x) where
  -- fmap' = dimap lolPar (inv' lolPar) . fmap'
  fmap' = lol \case
    L -> \nf -> apartR nf & \(x2a :-#> x :-#> nb) ->
      whyNot \a2b -> fun a2b (fun x2a x) != nb
    R -> \(Ur f) -> lol \case
      L -> \x -> x & \(a :-#> nb) -> a :-#> contra f nb
      R -> linear \g -> lol \case
        L -> linear \nb -> contra g (contra f nb)
        R -> \a -> fun f (fun g a)
```

Internalized Adjunctions

```haskell
class (Functor f, Functor g) => Adjunction f g | f -> g, g -> f where
  adj :: (Iso iso, Prop a, Prop b) => iso (f a ⊸ b) (a ⊸ g b)

instance Prop e => Adjunction ((,) e) ((⊸) e) where
  ...

-- The funny choice of direction for <#- comes from this instance.
instance Prop e => Adjunction ((<#-) e) ((⅋) e) where
  adj = iso \case
    L -> lol \case
      L -> \case
        (a :-#> ne) :-#> nb -> a :-#> (ne, nb)
      R -> \a2epb -> lol \case
        L -> \nb -> lol \case
          L -> \ne -> contra a2epb (ne, nb)
          R -> \a -> parL (fun a2epb a) nb
        R -> \(a :-#> ne) -> parR (fun a2epb a) ne
    R -> lol \case
      L -> \(a :-#> (ne, nb)) -> (a :-#> ne) :-#> nb
      R -> \k -> lol \case
        L -> \(ne, nb) -> contra (contra k nb) ne
        R -> \a -> par \case
          L -> linear \nb -> fun (contra k nb) a
          R -> \ne -> fun k (a :-#> ne)
```


UNCERTAINTY
AHEAD

An Active Question I Have

```
-- | Is this actually FILL?

-- Hyland and de Paiva's paper on FILL in section 4 describes this map as problematic
-- but it defines here just fine.

unrestricted
  :: (Prop a, Prop b) => (a ⅋ b ⊸ a) ⅋ b
unrestricted = par \case
  L -> \nb -> lol \case
    L -> \na -> (na, nb)
    R -> \apb -> parL apb nb
  R -> \(apb :-#> na) -> parR apb na
```

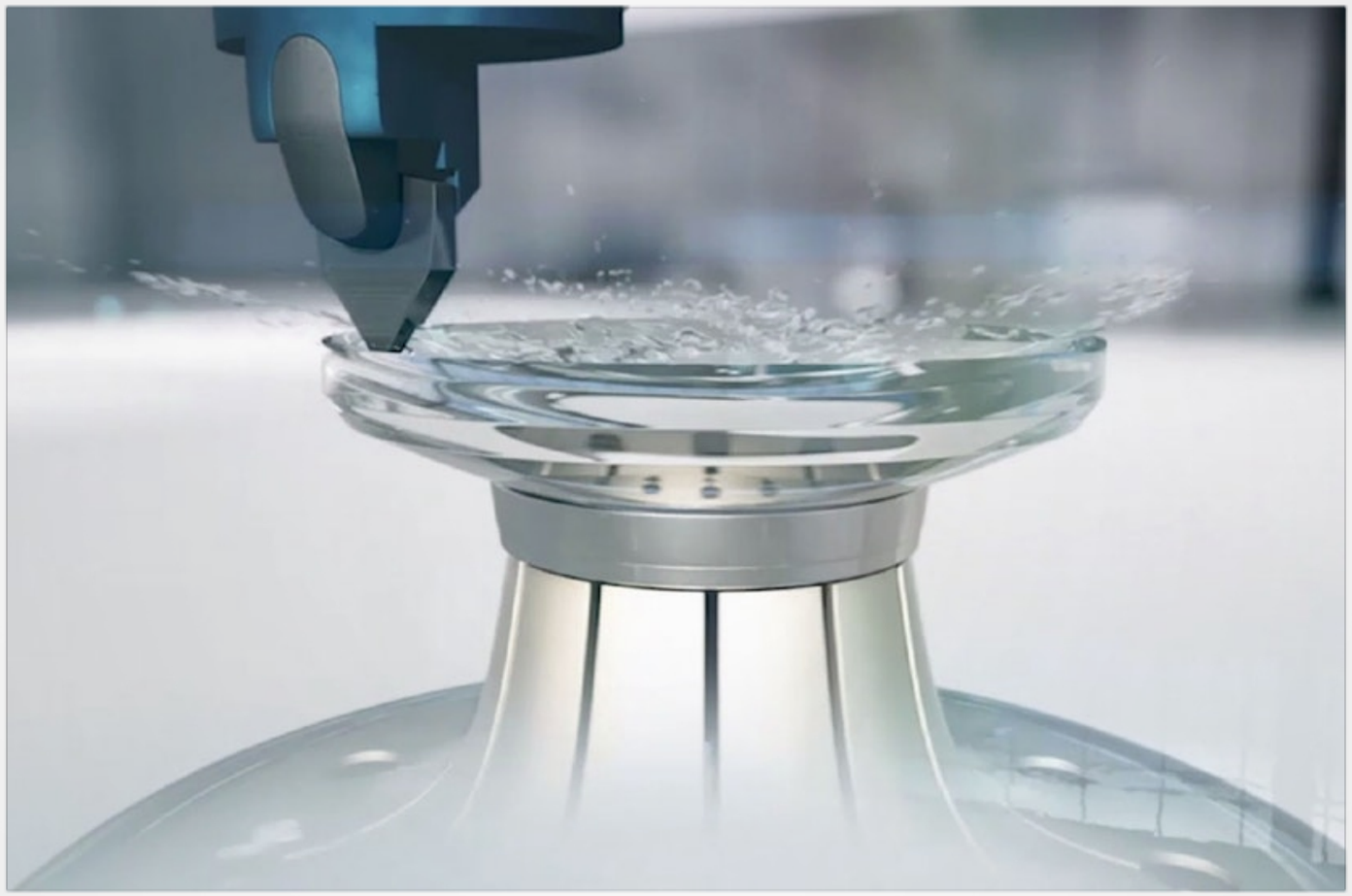
After Futamura introduced his now famous projections, he dove back into partial evaluation with "Generalized Partial Computation" (**GPC**) which can perform "negative" information flow analyses using a bunch of theorem proving.

Can the same be done by simply defining the semantics of the evaluator using an implication that allows for propagation of refutations bypassing said theorem proving, or at least packaging it in the interpreter?

Some preliminary tests of NBE have me excited here.

"Normal" Constructive Logic

```
-- | !a ⊸ b = Not (Ur a) ⅋ b
-- Not (!a ⊸ b) = Not b ⊸ ? (Not a)

newtype a ⊃ b = Imp
  { runImp :: forall c.
    (Prop a, Prop b) => Y (Not b %1 -> WhyNot (Not a)) (a -> b) c -> c
  }

infixr 0 ⊃

data Noimp b a where
  Noimp :: a -> Not b %1 -> Noimp b a

instance (Prop a, Prop b) => Prop (a ⊃ b) where
  type Not (a ⊃ b) = Noimp b a
  f != Noimp a b = runImp f R a != b

instance (Prop a, Prop b) => Prop (Noimp b a) where
  type Not (Noimp b a) = a ⊃ b
  Noimp a b != f = runImp f R a != b
```



Internalized Bifunctors

```
class
  ( forall a. Prop a => Functor (t a)
  ) => Bifunctor t where
  bimap'
    :: (Prop a, Prop b, Prop c, Prop d, Lol l, Lol l', Lol l'')
    => l (Ur (a ⊸ b)) (l' (Ur (c ⊸ d)) (l'' (t a c) (t b d)))

bimap
  :: (Bifunctor t, Prop a, Prop b, Prop c, Prop d, Lol l)
  => (a ⊸ b) -> (c ⊸ d) -> l (t a c) (t b d)
bimap f g = dimap' (Ur f) (Ur g)

instance Bifunctor Either
instance Bifunctor (,)
instance Bifunctor (&)
instance Bifunctor (⅋)
...
```

A Symmetric Monoidal Category 4-Ways

```haskell
class Bifunctor t => Semimonoidal t where
  assoc :: (Prop a, Prop b, Prop c, Iso iso) => t (t a b) c `iso` t a (t b c)

class (Prop (I t), Semimonoidal t) => Monoidal t where
  type I t :: Type
  lambda :: (Prop a, Iso iso) => a `iso` t (I t) a
  rho :: (Prop a, Iso iso) => a `iso` t a (I t)

class Symmetric t where
  swap :: (Prop a, Prop b, Iso iso) => t a b `iso` t b a

class (Symmetric t, Monoidal t) => SymmetricMonoidal t

instance Semimonoidal Either where
  assoc = assocEither

...
```

Internalized Profunctors

```
class
  ( forall a. Prop a => Functor (t a)
  ) => Profunctor t where
  dimap'
    :: (Prop a, Prop b, Prop c, Prop d, Lol l, Lol l', Lol l'')
    => l (Ur (a ⊸ b)) (l' (Ur (c ⊸ d)) (l'' (t b c) (t a d)))

dimap
  :: (Profunctor t, Prop a, Prop b, Prop c, Prop d, Lol l)
  => (a ⊸ b) -> (c ⊸ d) -> l (t b c) (t a d)
dimap f g = dimap' (Ur f) (Ur g)

instance Profunctor (⊸)
instance Profunctor (<#-)
instance Profunctor Noimp
instance Profunctor (Nofun m)
instance Profunctor (FUN m)
-- instance Profunctor (⊃)
...
```

We can now build strength with regards to each of the 4 differenct tensors we could be made a symmetric monoidal category with respect to. But because we have the *-autonomy from being a Chu space (**Not** exists and is contravariant) and because we fully "internalize" all our classes above, so we can turn around the arrows, we can get away with just strength.

Tracing is just strength with respect to the dual connective.

There's no need for 'Cochoice', 'Costrong'.


```haskell
class (Profunctor p, Bifunctor t) => Strong p t where
  first :: Lol l => l (p a b) (p (t a c) (t b c))
  second :: Lol l => l (p a b) (p (t c a) (t c b))

-- this needs a way to connect a connective and its dual connective
traceL :: (Strong p (OpBi t), Lol l) => OpPro p (t a c) (t b c) -> OpPro p a b
traceR :: (Strong p (OpBi t), Lol l) => OpPro p (t c a) (t c b) -> OpPro p a b

-- slightly opaque to avoid the tedium of defining the opposite profunctor
-- and opposite bifunctor here, but they are boring.
```

Lots of Juicy Distributive Laws

```
class (Functor f, Bifunctor p) => WeakDist f p where
  weakDist :: (Lol l, Prop b, Prop c) => l (f (p b c)) (p (f b) (f c))

instance Prop a => WeakDist ((,) a) (&)
instance Prop a => WeakDist ((,) a) Either
instance Prop a => WeakDist ((⅋) a) (&)

class WeakDist f p => Dist f p where
  dist :: (Iso i, Prop b, Prop c) => i (f (p b c)) (p (f b) (f c))

instance Prop a => Dist ((,) a) Either
instance Prop a => Dist ((⅋) a) (&)
```

Open: These invite normal-form finding expeditions for different combinations of these different strengths.

Obtaining Closure

I built "generalized" Tambara modules to in the **`profunctors`** library to cover cases like

```
class Profunctor p => Closed p where
  -- | Laws:
  --
  -- @
  -- 'lmap' ('.' f) '.' 'closed' ≡ 'rmap' ('.' f) '.' 'closed'
  -- 'closed' '.' 'closed' ≡ 'dimap' 'uncurry' 'curry' '.' 'closed'
  -- 'dimap' 'const' ('$'()) '.' 'closed' ≡ 'id'
  -- @
  closed :: p a b -> p (x -> a) (x -> b)
```

But recall linear implication (⊸) is directly definable in terms of (⅋), so this is just a normal Tambara module with respect to (⅋), which we can now see should have a previously unexplored connection to **`Costrong`**.

Open: This says something under-explored and interesting about "Grates" (which are now just strong w.r.t. (⅋)) and "Glasses" (strong w.r.t. both multiplicative connectives.).

(time permitting!)

Live Code Explorations

* Multiplicative (Control) Functors
* Additive (Data) Functors
* The breakdown of the relationship between (linear) folds and (linear) traversals due to weakening/ consumption.

Future Work: Typechecker Plugins

GHC is very bad at chasing through implications of QuantifiedConstraints.

As a domain specific optimization, we also should know that if we have a 'wanted' goal of **Not (Not a) ~ b**, we can simplify that into a request for `**a ~ b**`

Also, almost all of these proof terms are trivial. It'd be nice to be able to synthesize them. Maybe dip into some of Valeria's work on linear logic proof search?

Summary

Rebuilt Shulman's "Linear Logic for Constructive Mathematics" on top of Linear Haskell with a different (&) construction.

This let us construct a proper "linear logic".

We made it nicer to use with fancy overloading and typechecker plugins.

Optics using the fully internalized classes we can have there show a few of the usual relationships break down.

Questions?