Walking the way of duality (to programming corner)

Noam Zeilberger

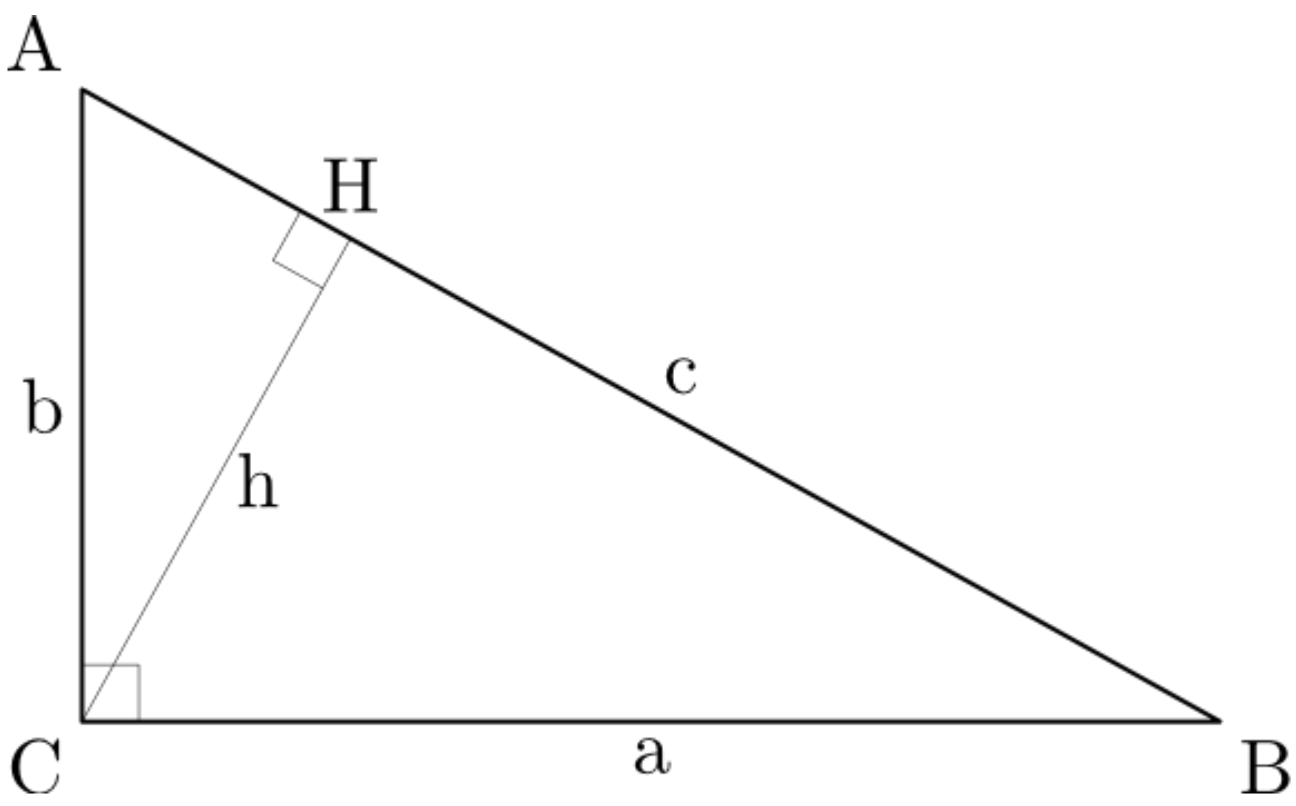
October 10th, 2008

A remarkable analogy

Proving is like programming

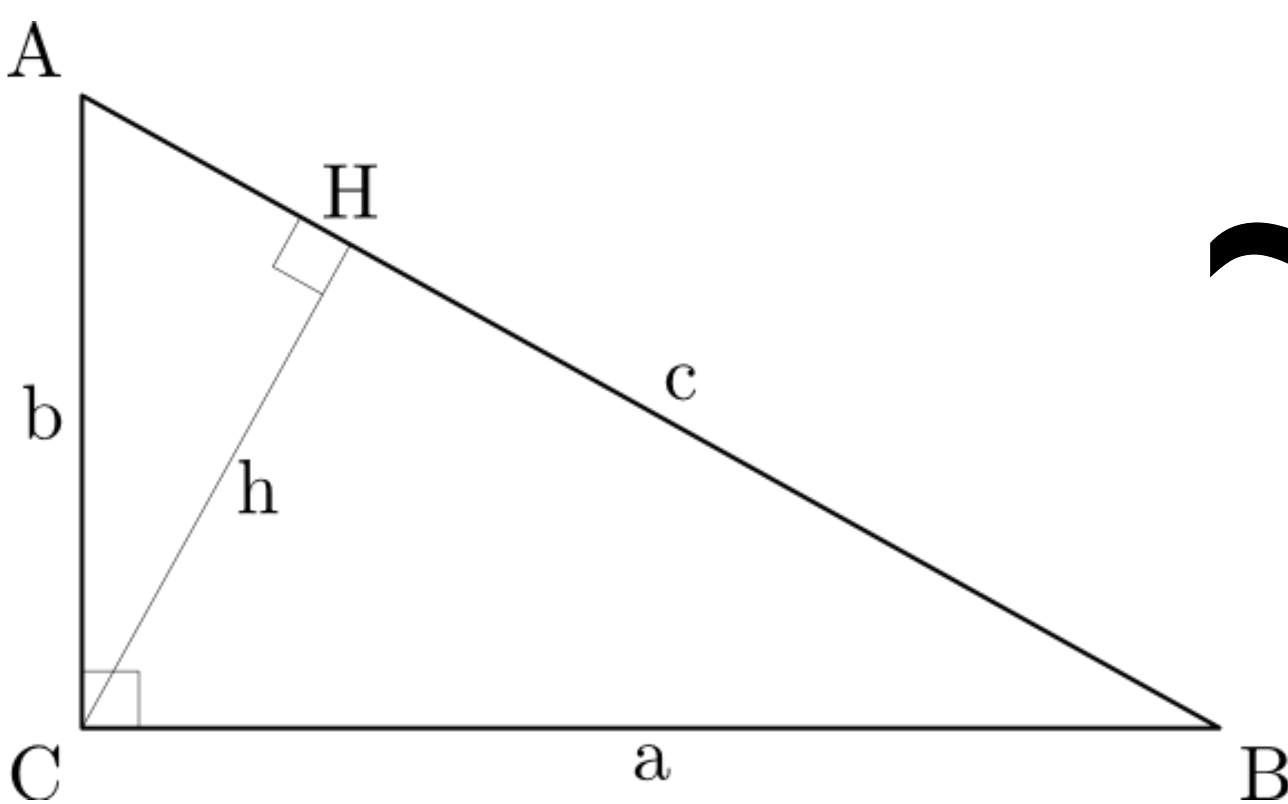
A remarkable analogy

Proving is like programming



A remarkable analogy

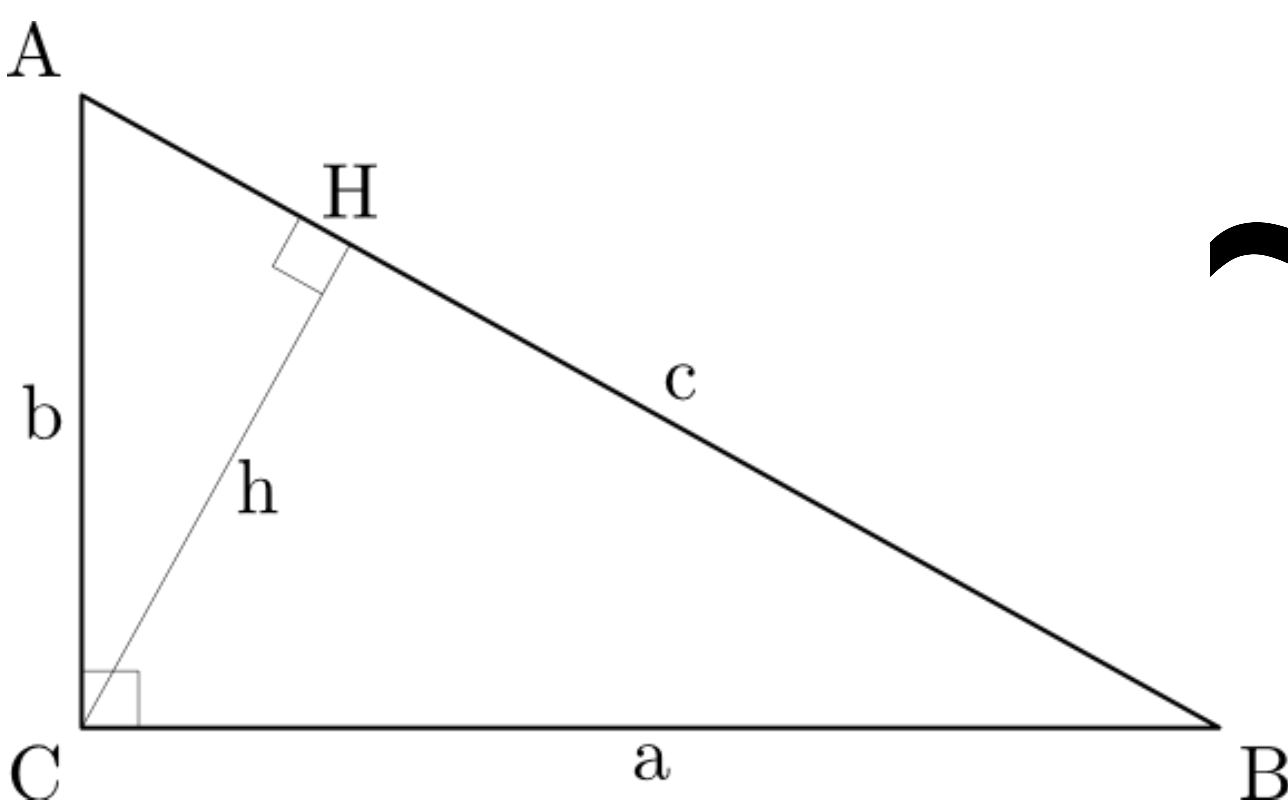
Proving is like programming



~

A remarkable analogy

Proving is like programming



~

```
'ExplodeGorilla:
'  Causes gorilla explosion when a c
'Parameters:
'  X#, Y# - shot location
FUNCTION ExplodeGorilla (x#, y#)
  YAdj = Scl(12)
  XAdj = Scl(5)
  SclX# = ScrWidth / 320
  SclY# = ScrHeight / 200
  IF x# < ScrWidth / 2 THEN PlayerHi
  PLAY "MBO0L16EFGEFDC"

  FOR i = 1 TO 8 * SclX#
    CIRCLE (GorillaX(PlayerHit) + 3.
SclY# + YAdj), i, ExplosionColor, ,
    LINE (GorillaX(PlayerHit) + 7 *
(GorillaX(PlayerHit), GorillaY(Playe
  NEXT i
  FOR i = 1 TO 16 * SclX#
```

A good analogy is like a diagonal frog

—Kai Krause

The analogy is suggestive...

...to programmers/PL designers it suggests:

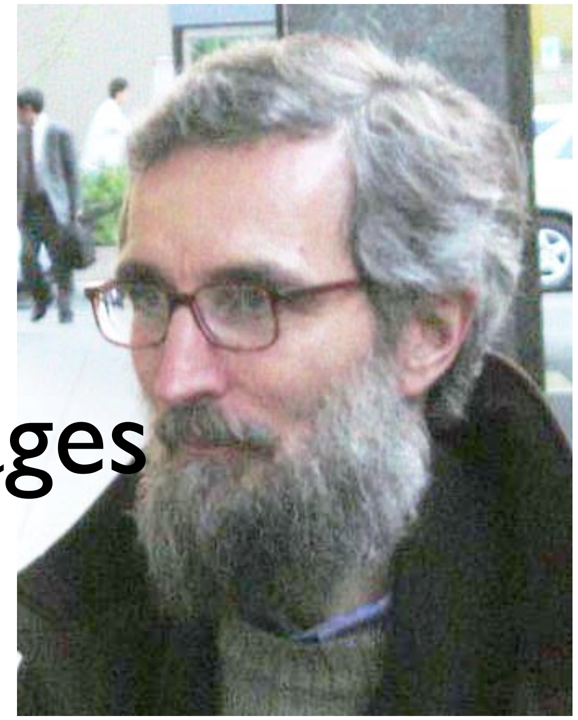
- new programming techniques
- new ways of understanding old languages
- ...ideas for organizing new languages

...to mathematicians/proof theorists it suggests:

- ways of mechanizing mathematics

The analogy is suggestive...

...to programmers/PL designers it suggests:



- new programming techniques
- new ways of understanding old languages
- ...ideas for organizing new languages

...to mathematicians/proof theorists it suggests:

- ways of mechanizing mathematics

...and inspiring...

...to programmers:

...and inspiring...

...to programmers:

- *I'm not just hacking, I'm proving theorems!*

...and inspiring...

...to programmers:

- *I’m not just hacking, I’m proving theorems!*

...to mathematicians:

...and inspiring...

...to programmers:

- *I'm not just hacking, I'm proving theorems!*

...to mathematicians:

- *I'm not just philosophizing, I'm writing programs!*

...and more than
an analogy!...

an isomorphism(s)

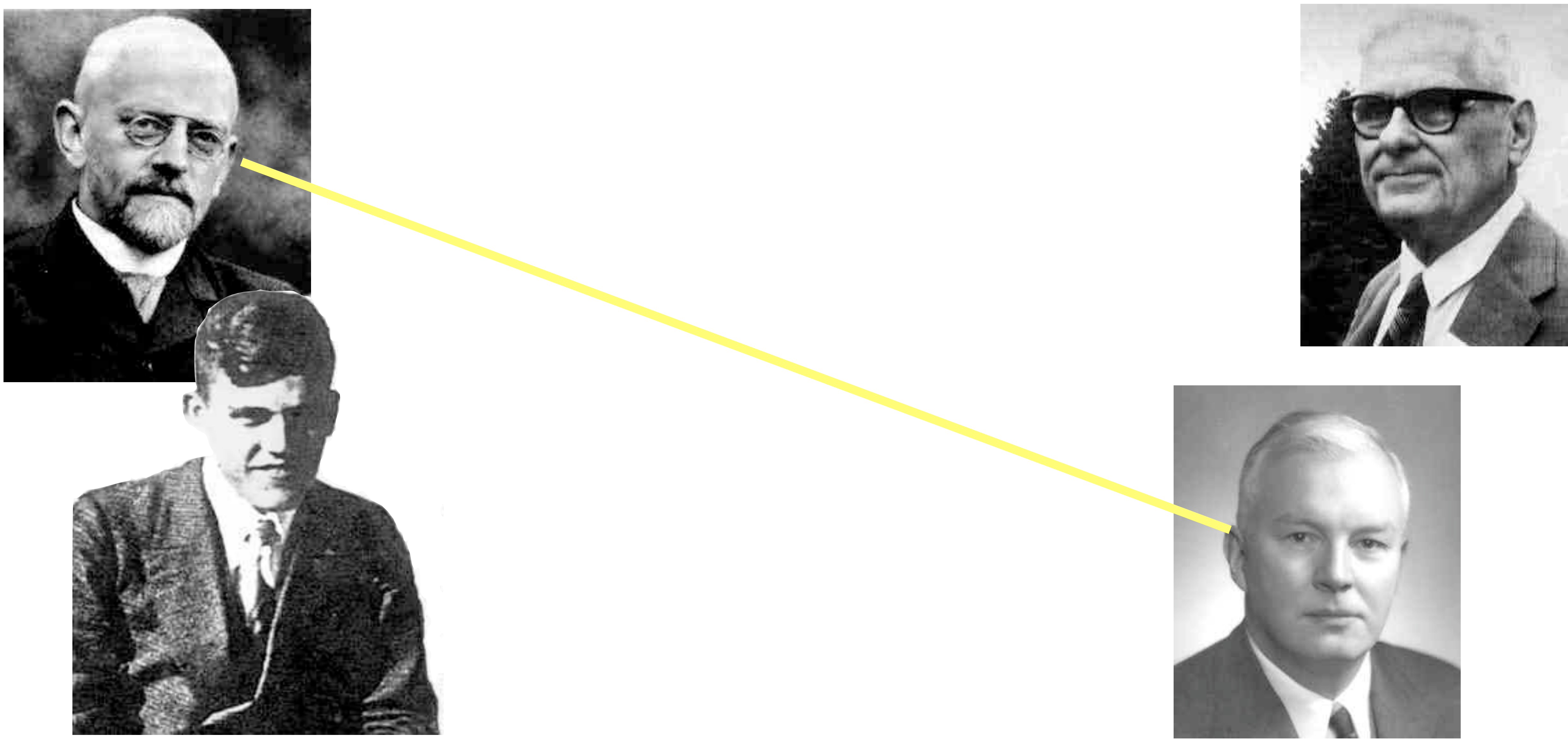
an isomorphism(s)

an isomorphism(s)

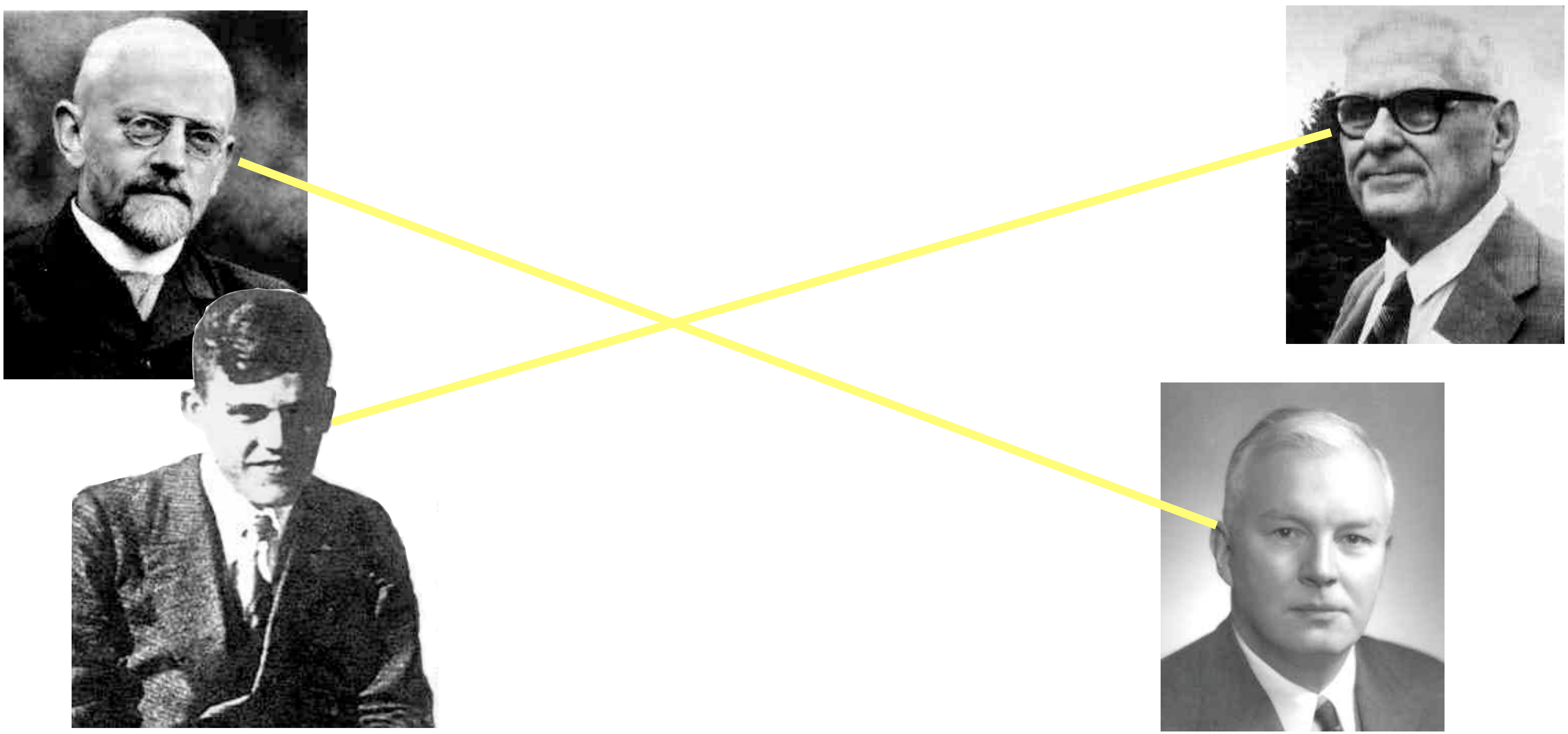
an isomorphism(s)

an isomorphism(s)

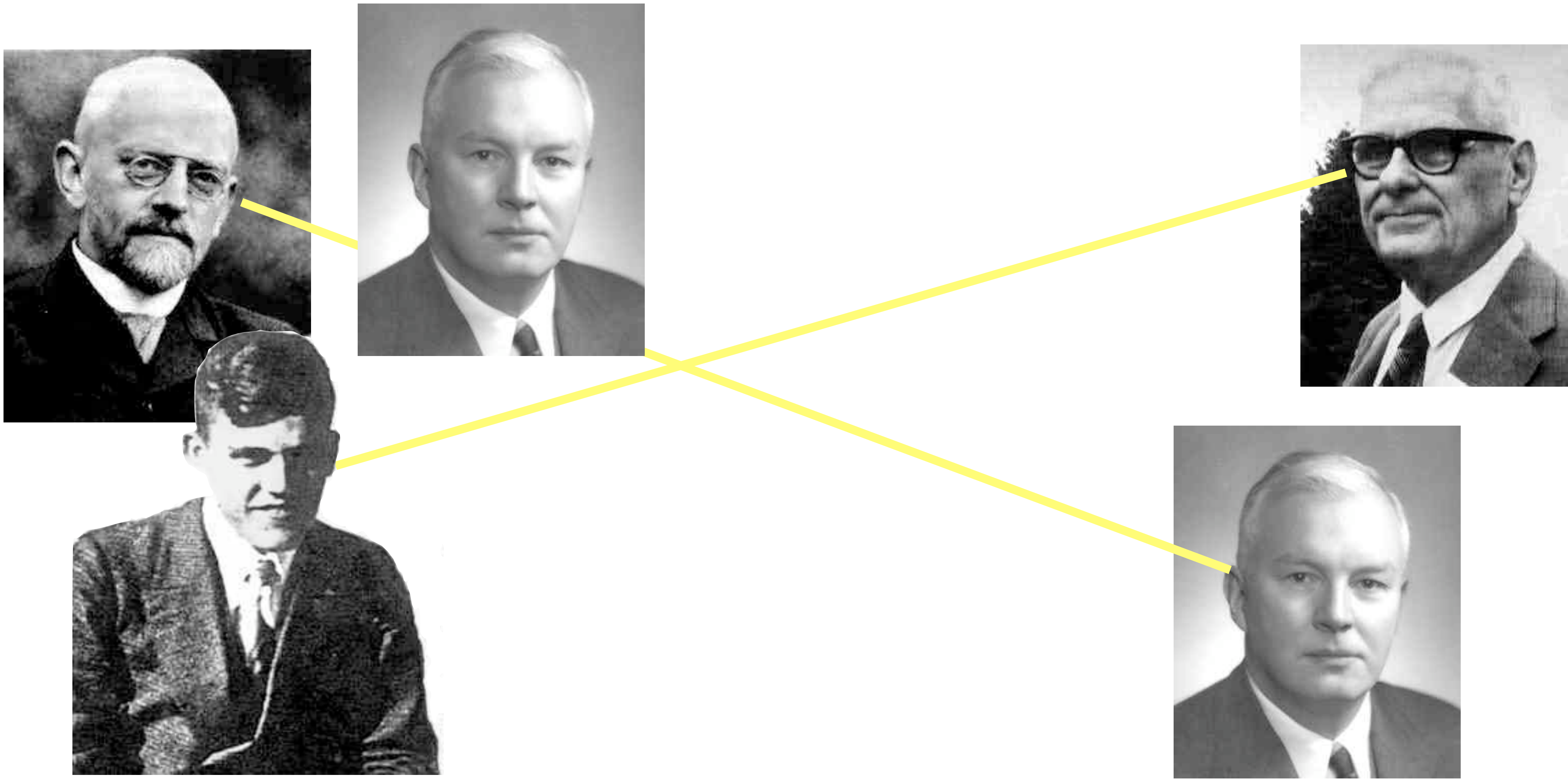
an isomorphism(s)



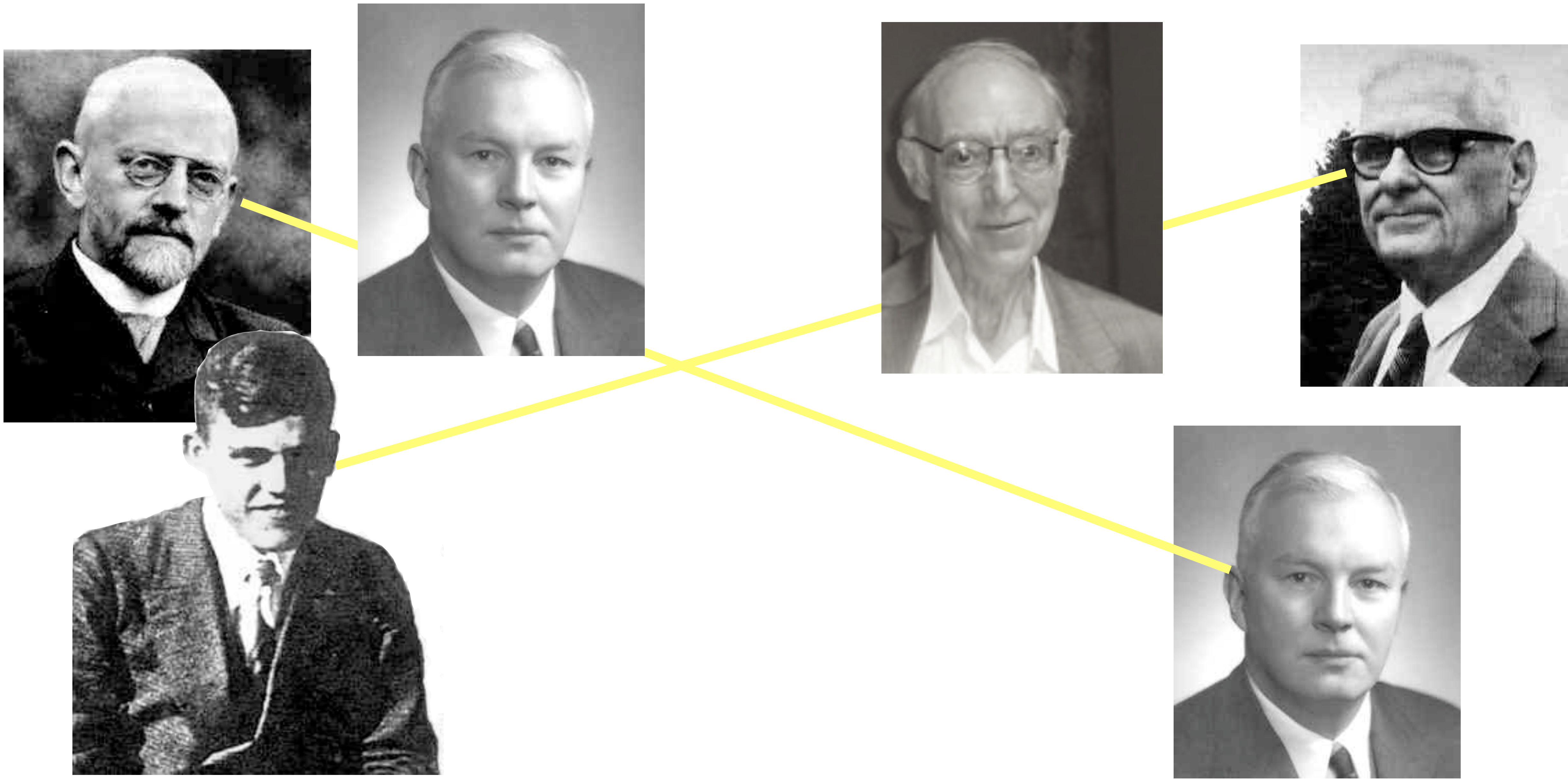
an isomorphism(s)



an isomorphism(s)



an isomorphism(s)

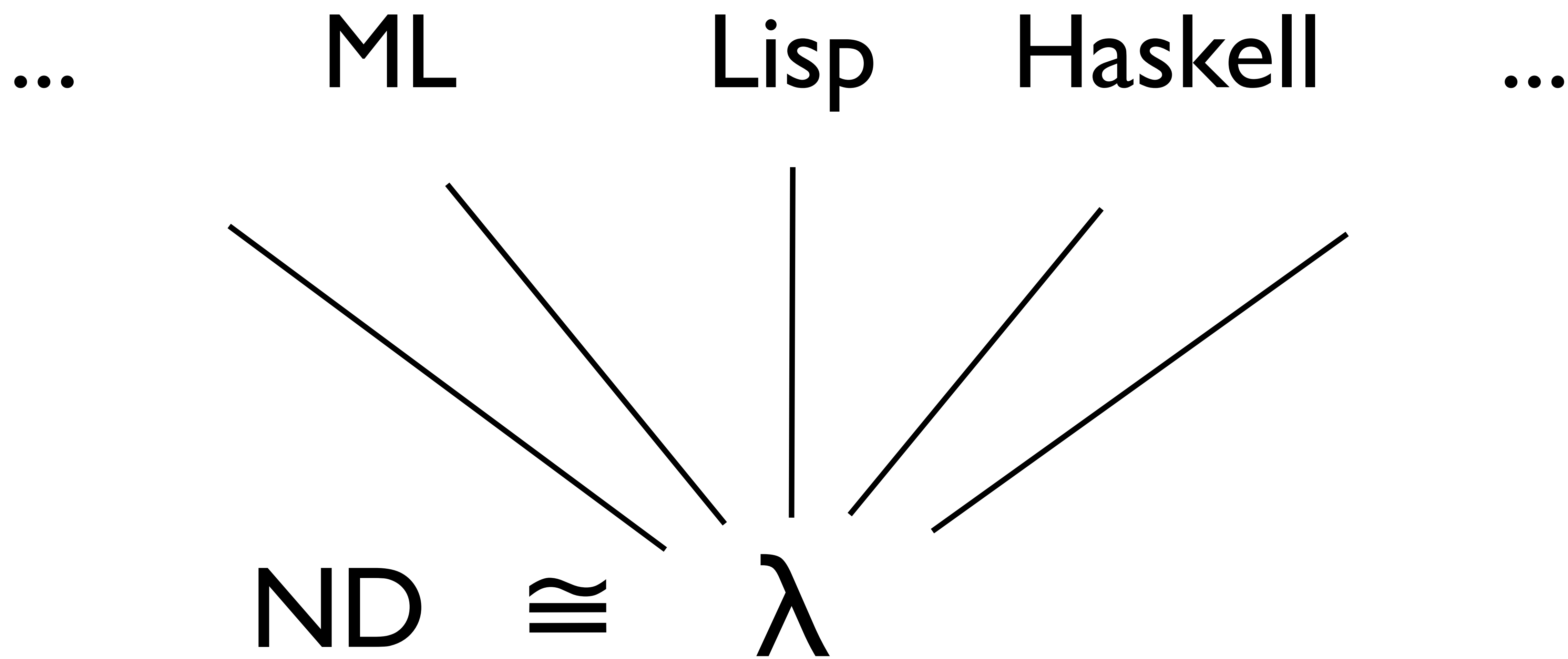


an isomorphism(s)

The foundation of functional programming

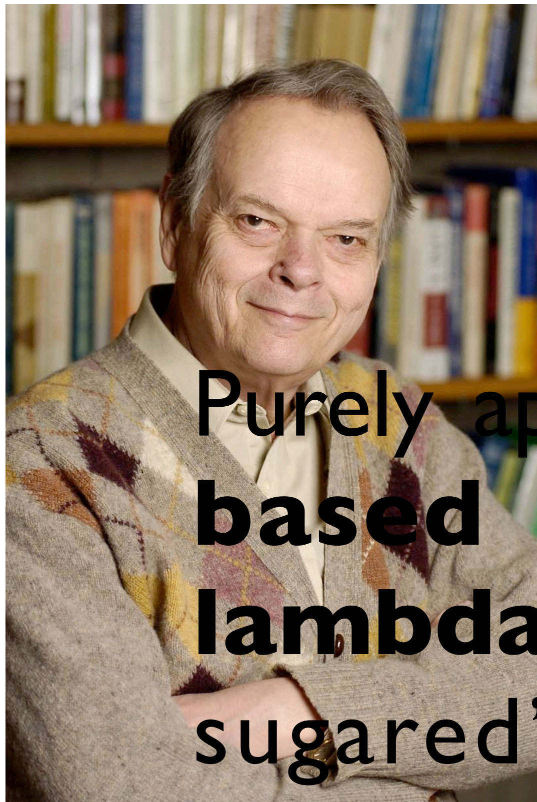
$$\mathrm{ND} \cong \lambda$$

The foundation of functional programming



A shaky foundation!

Purely applicative languages are **often said to be based on a logical system called the lambda calculus,** or even to be "syntactically sugared" versions of the lambda calculus.... **However,** as we will see, although an unsugared applicative language is syntactically equivalent to the lambda calculus, **there is a subtle semantic difference.** Essentially, the "real" lambda calculus implies a different "order of application"...than most applicative programming languages. —John Reynolds



Purely applicative languages are **often said to be based on a logical system called the lambda calculus,** or even to be "syntactically sugared" versions of the lambda calculus.... **However,** as we will see, although an unsugared applicative language is syntactically equivalent to the lambda calculus, **there is a subtle semantic difference.** Essentially, the "real" lambda calculus implies a different "order of application"...than most applicative programming languages. —John Reynolds

My work

Proposes an alternative foundation

- ...for *real-world* functional PLs
- with programs-as-proofs isomorphism

...based on **duality**

- duality between proofs and refutations
- duality between values and continuations

My message

There are deep mathematical symmetries...

- within languages like ML and Haskell
- *between* languages like ML and Haskell
- revealed by examining *patterns*

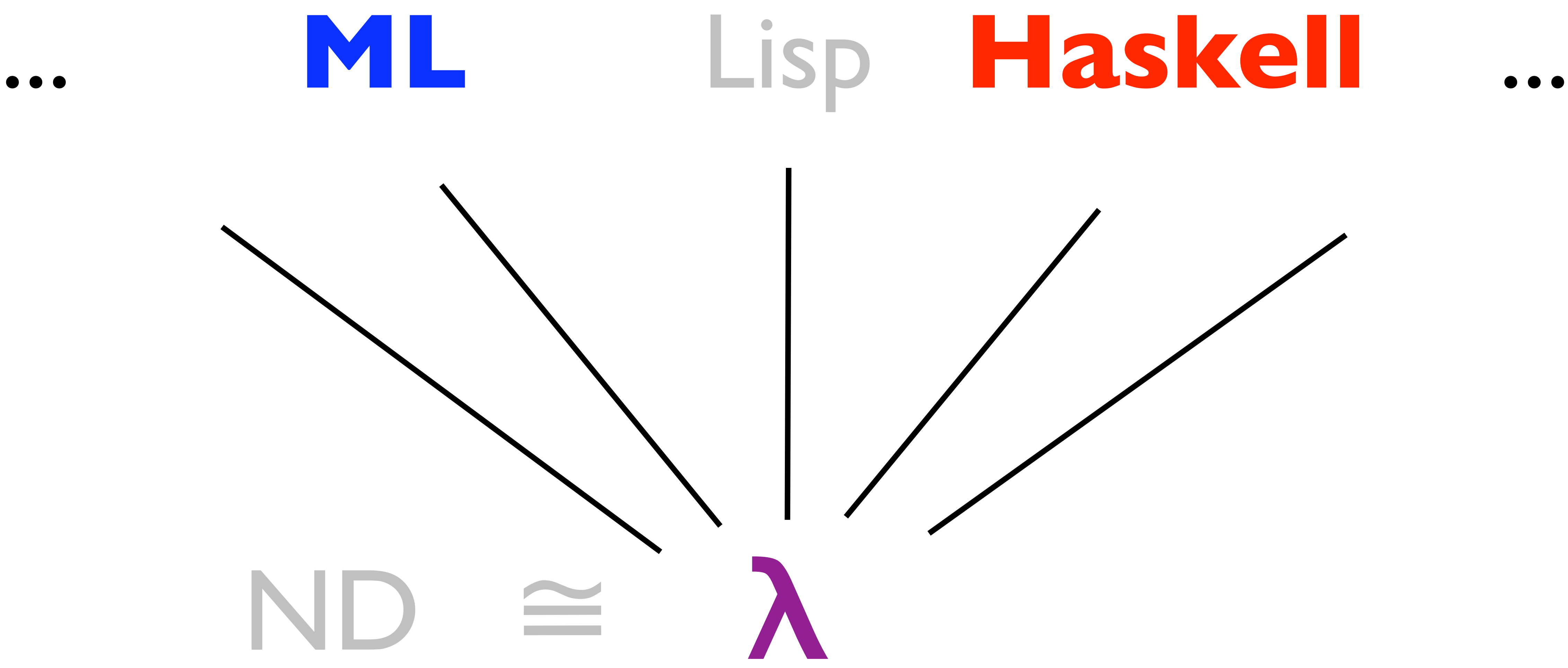
Duality is like a ~~diagonal frog~~ square

Talk outline

- The proofs-as-programs analogy
- Explain why λ is an inadequate foundation
- Explain duality of proofs and refutations
- *Extract* a new foundational PL
- Profit

The trouble with λ

The foundation of functional programming



arith.sml

```
datatype nat = Z | S of nat
fun plus Z n = n
  | plus (S m) n = S (plus m n)
fun times Z n = Z
  | times (S m) n = plus n (times m n)
```

arith.hs

```
data Nat = Z | S (Nat)
plus Z n = n
plus (S m) n = S (plus m n)
times Z n = Z
times (S m) n = plus n (times m n)
```

```
Standard ML of New Jersey v110.67
- use "arith.sml";
[opening arith.sml]
- val two = S (S Z);
- val three = S two;
- times two three;
```

```
Standard ML of New Jersey v110.67
- use "arith.sml";
[opening arith.sml]
- val two = S (S Z);
- val three = S two;
- times two three;
val it = S (S (S (S (S (S Z))))) : nat
```

```
GHCi, version 6.8.3
Prelude> :load arith
Ok, modules loaded: Main.
*Main> let two = S (S Z)
*Main> let three = S two
*Main> times two three
S (S (S (S (S (S Z)))))
```

- fun infinity() = S(infinity())

```
- fun infinity() = S(infinity())
- times Z (infinity());
```

```
- fun infinity() = S(infinity())
- times Z (infinity());
^CInterrupt
```

```
*Main> let infinity = S(infinity)
```

```
*Main> let infinity = S(infinity)
*Main> times Z infinity
Z
```

Evaluation order

ML

call-by-value

Haskell

call-by-name

Evaluation order

ML

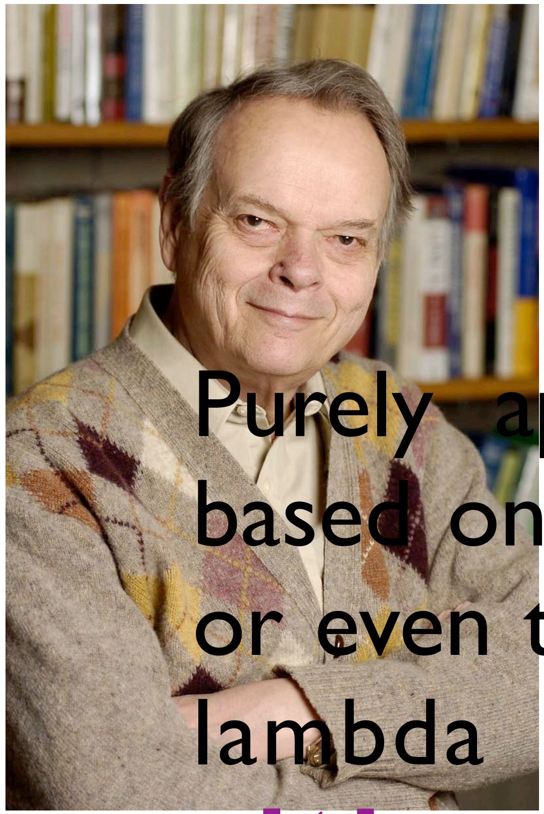
call-by-value

Haskell

call-by-name

λ

undecided



Purely applicative languages are often said to be based on a logical system called the lambda calculus, or even to be "syntactically sugared" versions of the lambda calculus.... However, as we will see, **although an unsugared applicative language is syntactically equivalent to the lambda calculus, there is a subtle semantic difference.** Essentially, the "real" lambda calculus implies a different "order of application"...than most applicative programming languages. —John Reynolds

Syntax and semantics are not independent!

A **type system** syntactically guarantees semantic properties (“well-typed programs don’t go wrong”)

...as a type system gets more precise, it *must* take evaluation order into account.

“ML with callcc is unsound”

Safety violation in SML/NJ, discovered in ‘91

Bad interaction polymorphism ↔ effects

Stopgap measure: a *value restriction*

- ML needs one
- Haskell does not

Why didn't you warn us, λ?

λ tells us nothing about typing with effects

But we *need* guidance in developing...

- union and intersection types
- dependent types
- module systems
- ...the languages of the future

And now for
something different
(but actually the same)

Talk outline

- The proofs-as-programs analogy
- Explain why λ is an inadequate foundation
- Explain duality of proofs and refutations
- *Extract* a new foundational PL
- Profit

a proof-biased logic

Once we have understood how to discover individual patterns which are alive, we may then make a language for ourselves, for any building task we face.

—Christopher Alexander

proof patterns

Describe how to prove a proposition

proof patterns

Describe how to prove a proposition

- *"to prove $A \wedge B$, prove both A and B"*

proof patterns

Describe how to prove a proposition

- *"to prove $A \wedge B$, prove both A and B"*
- *"to prove $A \vee B$, prove either A or B"*

proof patterns

Describe how to prove a proposition

- *“to prove $A \wedge B$, prove both A and B”*
- *“to prove $A \vee B$, prove either A or B”*
- *“to prove $\neg A$, refute A”*

proof patterns

Describe how to prove a proposition

- *"to prove $A \wedge B$, prove both A and B"*
- *"to prove $A \vee B$, prove either A or B"*
- *"to prove $\neg A$, refute A"*
- *"to prove True, done!"*

proof patterns

Describe how to prove a proposition

- *"to prove $A \wedge B$, prove both A and B"*
- *"to prove $A \vee B$, prove either A or B"*
- *"to prove $\neg A$, refute A"*
- *"to prove True, done!"*
- *"to prove False, no way."*

the pattern is holey

A proof pattern gives us the outline of a proof, but leaves holes for refutations

the pattern is holey

A proof pattern gives us the outline of a proof, but leaves holes for refutations

$$\neg A \wedge (\neg B \vee \neg C) \; \textit{true}$$

the pattern is holey

A proof pattern gives us the outline of a proof, but leaves holes for refutations

$$\frac{\neg A \ \mathit{true} \qquad \neg B \vee \neg C \ \mathit{true}}{\neg A \wedge (\neg B \vee \neg C) \ \mathit{true}}$$

the pattern is holey

A proof pattern gives us the outline of a proof, but leaves holes for refutations

$$\dfrac{\dfrac{A\ \textit{false}}{\neg A\ \textit{true}} \qquad \neg B \vee \neg C\ \textit{true}}{\neg A \wedge (\neg B \vee \neg C)\ \textit{true}}$$

the pattern is holey

A proof pattern gives us the outline of a proof, but leaves holes for refutations

$$\dfrac{\dfrac{A\ \mathit{false}}{\neg A\ \mathit{true}} \qquad \dfrac{\neg B\ \mathit{true}}{\neg B \vee \neg C\ \mathit{true}}}{\neg A \wedge (\neg B \vee \neg C)\ \mathit{true}}$$

the pattern is holey

A proof pattern gives us the outline of a proof, but leaves holes for refutations

$$\dfrac{\dfrac{\color{blue}{A\ \mathit{false}}}{\neg A\ \mathit{true}} \qquad \dfrac{\dfrac{\color{blue}{B\ \mathit{false}}}{\neg B\ \mathit{true}}}{\neg B \vee \neg C\ \mathit{true}}}{\neg A \wedge (\neg B \vee \neg C)\ \mathit{true}}$$

notation

$$A_1 \textit{ false}, \ldots, A_n \textit{ false} \Vdash A \textit{ true}$$

There is a proof pattern for A, leaving holes for refutations of $A_1 \ldots A_n$

notation

$$\underbrace{A_1 \textit{ false}, \ldots, A_n \textit{ false}}_{\Delta} \Vdash A \textit{ true}$$

There is a proof pattern for A, leaving holes for refutations of $A_1 \ldots A_n$

pattern axioms

- if $\Delta_1 \Vdash A\ true$ and $\Delta_2 \Vdash B\ true$ then $\Delta_1\Delta_2 \Vdash A \wedge B\ true$
- if $\Delta \Vdash A\ true$ then $\Delta \Vdash A \vee B\ true$
- if $\Delta \Vdash B\ true$ then $\Delta \Vdash A \vee B\ true$
- $A\ false \Vdash \neg A\ true$
- $\cdot \Vdash True\ true$

proofs and refutations, informally...

To prove *A*, find a proof pattern for *A* and fill in its holes.

To refutate *A*, consider every proof pattern for *A* and show that its holes can't be filled.

formal system

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$

formal system

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$

formal system

- $\Gamma \vdash A$ *true* **iff** $\Delta \Vdash A$ *true* and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** A *false* $\in \Delta$ implies $\Gamma \vdash A$ *false*
- $\Gamma \vdash A$ *false* **iff** $\Delta \Vdash A$ *true* implies $\Gamma, \Delta \vdash \#$

formal system

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$
- $\Gamma \vdash A\ \mathit{false}$ **iff** $\Delta \Vdash A\ \mathit{true}$ implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** $A\ \mathit{false} \in \Gamma$ and $\Gamma \vdash A\ \mathit{true}$

a square of dualities

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$
- $\Gamma \vdash A\ \mathit{false}$ **iff** $\Delta \Vdash A\ \mathit{true}$ implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** $A\ \mathit{false} \in \Gamma$ and $\Gamma \vdash A\ \mathit{true}$

a square of dualities

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$
- $\Gamma \vdash A\ \mathit{false}$ **iff** $\Delta \Vdash A\ \mathit{true}$ implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** $A\ \mathit{false} \in \Gamma$ and $\Gamma \vdash A\ \mathit{true}$

a square of dualities

- $\Gamma \vdash A$ *true* **iff** $\Delta \Vdash A$ *true* and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** A *false* $\in \Delta$ implies $\Gamma \vdash A$ *false*
- $\Gamma \vdash A$ *false* **iff** $\Delta \Vdash A$ *true* implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** A *false* $\in \Gamma$ and $\Gamma \vdash A$ *true*

a square of dualities

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$
- $\Gamma \vdash A\ \mathit{false}$ **iff** $\Delta \Vdash A\ \mathit{true}$ implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** $A\ \mathit{false} \in \Gamma$ and $\Gamma \vdash A\ \mathit{true}$

a square of dualities

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$
- $\Gamma \vdash A\ \mathit{false}$ **iff** $\Delta \Vdash A\ \mathit{true}$ implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** $A\ \mathit{false} \in \Gamma$ and $\Gamma \vdash A\ \mathit{true}$

Duality, dualized

So much time and so little to do. Wait a minute. Strike that. Reverse it.

—Willy Wonka

a refutation-biased logic

refutation patterns

Describe how to refute a proposition

refutation patterns

Describe how to refute a proposition

- *"to refute $A \wedge B$, refute either A or B"*

refutation patterns

Describe how to refute a proposition

- *"to refute $A \wedge B$, refute either A or B"*
- *"to refute $A \vee B$, refute both A and B"*

refutation patterns

Describe how to refute a proposition

- *“to refute $A \wedge B$, refute either A or B”*
- *“to refute $A \vee B$, refute both A and B”*
- *“to refute $\neg A$, prove A”*

refutation patterns

Describe how to refute a proposition

- *"to refute $A \wedge B$, refute either A or B"*
- *"to refute $A \vee B$, refute both A and B"*
- *"to refute $\neg A$, prove A"*
- *"to refute True, tough luck."*

refutation patterns

Describe how to refute a proposition

- *"to refute $A \wedge B$, refute either A or B"*
- *"to refute $A \vee B$, refute both A and B"*
- *"to refute $\neg A$, prove A"*
- *"to refute True, tough luck."*
- *"to refute False, just did."*

notation

$$\underbrace{A_1\ true, \ldots, A_n\ true}_{\Delta} \Vdash A\ false$$

There is a refutation pattern for A, leaving holes for proofs of $A_1 \ldots A_n$

pattern axioms

- if $\Delta \Vdash A$ *false* then $\Delta \Vdash A \wedge B$ *false*
- if $\Delta \Vdash B$ *false* then $\Delta \Vdash A \wedge B$ *false*
- if $\Delta_1 \Vdash A$ *false* and $\Delta_2 \Vdash B$ *false* then $\Delta_1 \Delta_2 \Vdash A \vee B$ *false*
- A *true* $\Vdash \neg A$ *false*
- $\cdot \Vdash$ *False false*

proofs and refutations, informally...

To refute *A*, find a refutation pattern for *A* and fill in its holes.

To prove *A*, consider every refutation pattern for *A* and show that its holes can't be filled.

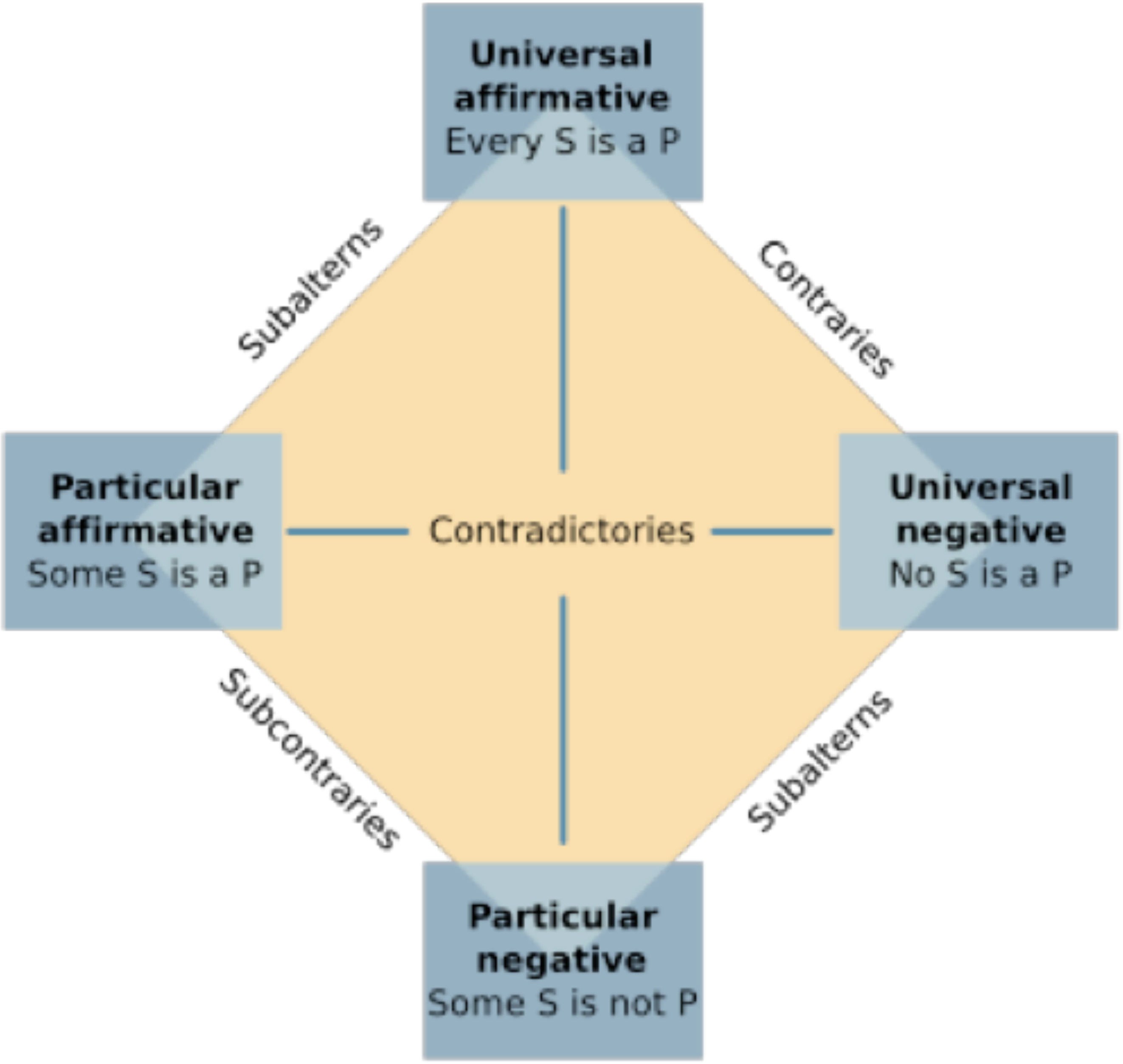
formal system

- $\Gamma \vdash A$ *true* **iff** $\Delta \Vdash A$ *true* and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** A *false* $\in \Delta$ implies $\Gamma \vdash A$ *false*
- $\Gamma \vdash A$ *false* **iff** $\Delta \Vdash A$ *true* implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** A *false* $\in \Gamma$ and $\Gamma \vdash A$ *true*

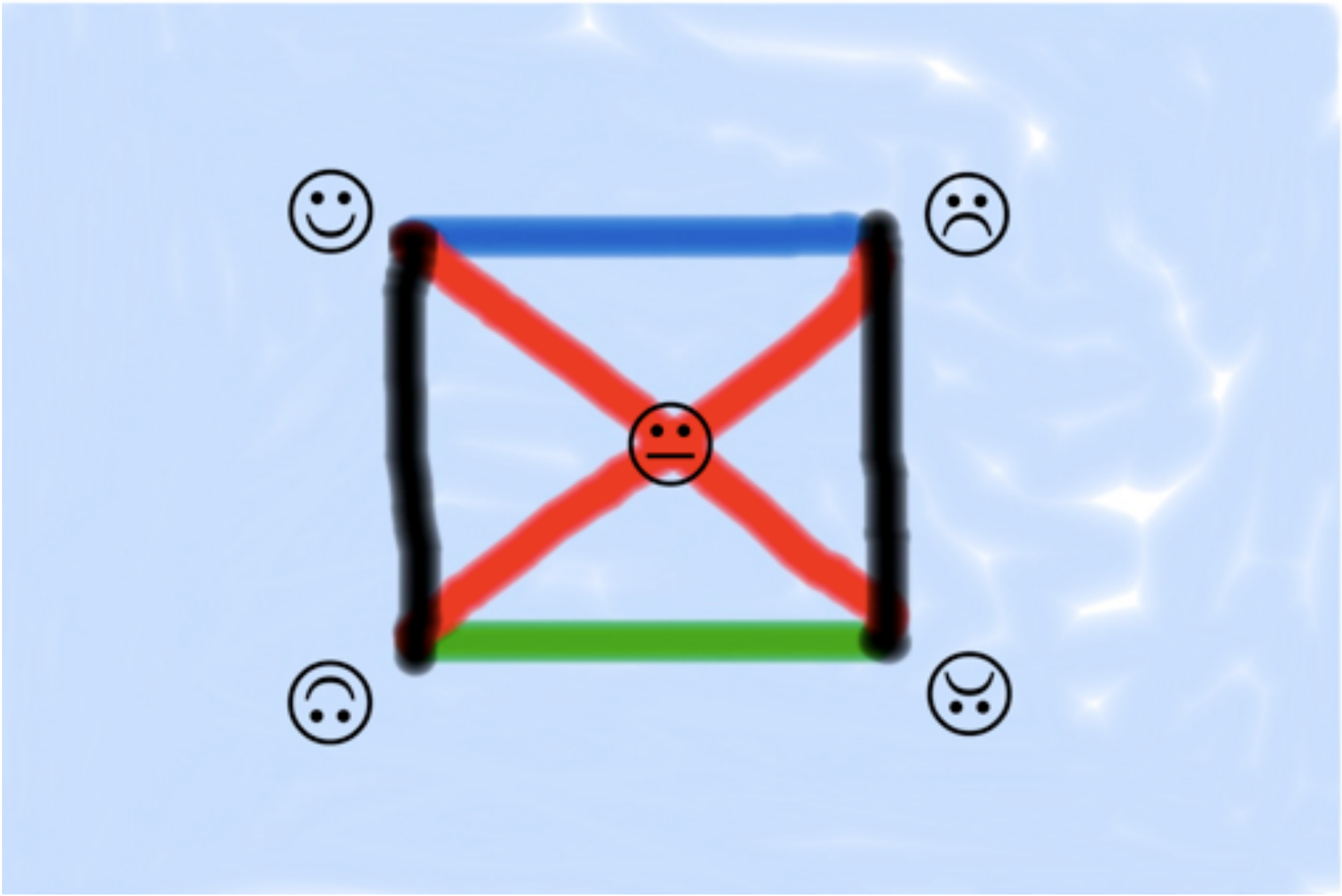
strike that, reverse it...

- $\Gamma \vdash A$ *false* **iff** $\Delta \Vdash A$ *false* and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** A *true* $\in \Delta$ implies $\Gamma \vdash A$ *true*
- $\Gamma \vdash A$ *true* **iff** $\Delta \Vdash A$ *false* implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** A *true* $\in \Gamma$ and $\Gamma \vdash A$ *false*

Square of Opposition



Square of Opposition



So what does this have to do with programming?

Talk outline

- The proofs-as-programs analogy
- Explain why λ is an inadequate foundation
- Explain duality of proofs and refutations
- *Extract* a new foundational PL
- Profit

Punchline

We're already done!

Reading the logical rules *constructively* gives us an *intrinsically typed programming language*

But for simplicity, let's go sans types...

Translation guide

Logical judgment	Syntactic category	Name
$\Delta \Vdash A\ \mathit{true}$	value pattern	VPat
$\Delta \Vdash A\ \mathit{false}$	continuation pattern	KPat
$A\ \mathit{false} \in \Delta$	continuation variable	KVar
$A\ \mathit{true} \in \Delta$	value variable	VVar
$\Gamma \vdash A\ \mathit{true}$	value	Val
$\Gamma \vdash A\ \mathit{false}$	continuation	Kon
$\Gamma \vdash \Delta$	substitution	Sub
$\Gamma \vdash \#$	computation	Cmp

Language #1

from proof patterns...

- if $\Delta_1 \Vdash A\ \mathit{true}$ and $\Delta_2 \Vdash B\ \mathit{true}$ then $\Delta_1\Delta_2 \Vdash A \wedge B\ \mathit{true}$
- if $\Delta \Vdash A\ \mathit{true}$ then $\Delta \Vdash A \vee B\ \mathit{true}$
- if $\Delta \Vdash B\ \mathit{true}$ then $\Delta \Vdash A \vee B\ \mathit{true}$
- $A\ \mathit{false} \Vdash \neg A\ \mathit{true}$
- $\cdot \Vdash \mathit{True}\ \mathit{true}$

...to value patterns

- if $p_1 \in$ VPat and $p_2 \in$ VPat
  then $(p_1, p_2) \in$ VPat
- if $p \in$ VPat then *inl* $p \in$ VPat
- if $p \in$ VPat then *inr* $p \in$ VPat
- if $k \in$ KVar then $k \in$ VPat
- $() \in$ VPat

e.g.,

$$\dfrac{\dfrac{A\ \textit{false}}{\neg A\ \textit{true}} \qquad \dfrac{\dfrac{B\ \textit{false}}{\neg B\ \textit{true}}}{\neg B \vee \neg C\ \textit{true}}}{\neg A \wedge (\neg B \vee \neg C)\ \textit{true}} \quad \mapsto \quad (k_1,\ \mathrm{inl}\ k_2)$$

from logic...

- $\Gamma \vdash A\ \mathit{true}$ **iff** $\Delta \Vdash A\ \mathit{true}$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** $A\ \mathit{false} \in \Delta$ implies $\Gamma \vdash A\ \mathit{false}$
- $\Gamma \vdash A\ \mathit{false}$ **iff** $\Delta \Vdash A\ \mathit{true}$ implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** $A\ \mathit{false} \in \Gamma$ and $\Gamma \vdash A\ \mathit{true}$

...to language

- Val = VPat × Sub
- Sub = KVar → Kon
- Kon = VPat → Cmp
- Cmp = KVar × Val

...to language

- Val = VPat × Sub
- Sub = KVar → Kon
- Kon = VPat → Cmp
- Cmp = KVar × Val

Is that *really* a language?

Yes, trust me.

Like λ, it is minimalistic, but unlike λ it...

- has inherent support for products, sums, and pattern-matching
- inherently enforces *call-by-value*

(NB: can think of image of CBV CPS transform)

Yes, trust me.

Like λ, it is minimalistic, but unlike λ it...

- has inherent support for products, sums, and pattern-matching
- inherently enforces *call-by-value*

(NB: can think of image of CBV CPS transform)

...What about CBN?

Language #2

you know the drill...

from refutation patterns...

- if $\Delta \Vdash A$ *false* then $\Delta \Vdash A \wedge B$ *false*
- if $\Delta \Vdash B$ *false* then $\Delta \Vdash A \wedge B$ *false*
- if $\Delta_1 \Vdash A$ *false* and $\Delta_2 \Vdash B$ *false* then $\Delta_1 \Delta_2 \Vdash A \vee B$ *false*
- A *true* $\Vdash \neg A$ *false*
- $\cdot \Vdash$ *False false*

...to continuation patterns

- if $d \in$ KPat then $fst\ d \in$ KPat
- if $d \in$ KPat then $snd\ d \in$ KPat
- if $d_1 \in$ KPat and $d_2 \in$ KPat
  then $[d_1, d_2] \in$ KPat
- if $x \in$ VVar then $x \in$ KPat
- $[] \in$ KPat

okay so continuation patterns are a little weird...

from logic...

- $\Gamma \vdash A$ *false* **iff** $\Delta \Vdash A$ *false* and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ **iff** A *true* $\in \Delta$ implies $\Gamma \vdash A$ *true*
- $\Gamma \vdash A$ *true* **iff** $\Delta \Vdash A$ *false* implies $\Gamma, \Delta \vdash \#$
- $\Gamma \vdash \#$ **iff** A *true* $\in \Gamma$ and $\Gamma \vdash A$ *false*

...to language

- Kon = KPat × Sub
- Sub = VVar → Val
- Val = KPat → Cmp
- Cmp = VVar × Kon

...to language

- Kon = KPat × Sub
- Sub = VVar → Val
- Val = KPat → Cmp
- Cmp = VVar × Kon

Recap

the CBV square

Val = VPat × Sub	Kon = VPat → Cmp
Sub = KVar → Kon	Cmp = KVar × Val

the CBN square

Val = KPat → Cmp	Kon = KPat × Sub
Sub = VVar → Val	Cmp = VVar × Kon

CBV-CBN duality

$\mathsf{Val}^{+} = \mathsf{VPat} \times \ldots$	$\mathsf{Kon}^{+} = \mathsf{VPat} \to \ldots$
$\mathsf{Val}^{-} = \mathsf{KPat} \to \ldots$	$\mathsf{Kon}^{-} = \mathsf{KPat} \times \ldots$

Talk outline

- The proofs-as-programs analogy
- Explain why λ is an inadequate foundation
- Explain duality of proofs and refutations
- *Extract* a new foundational PL
- Profit

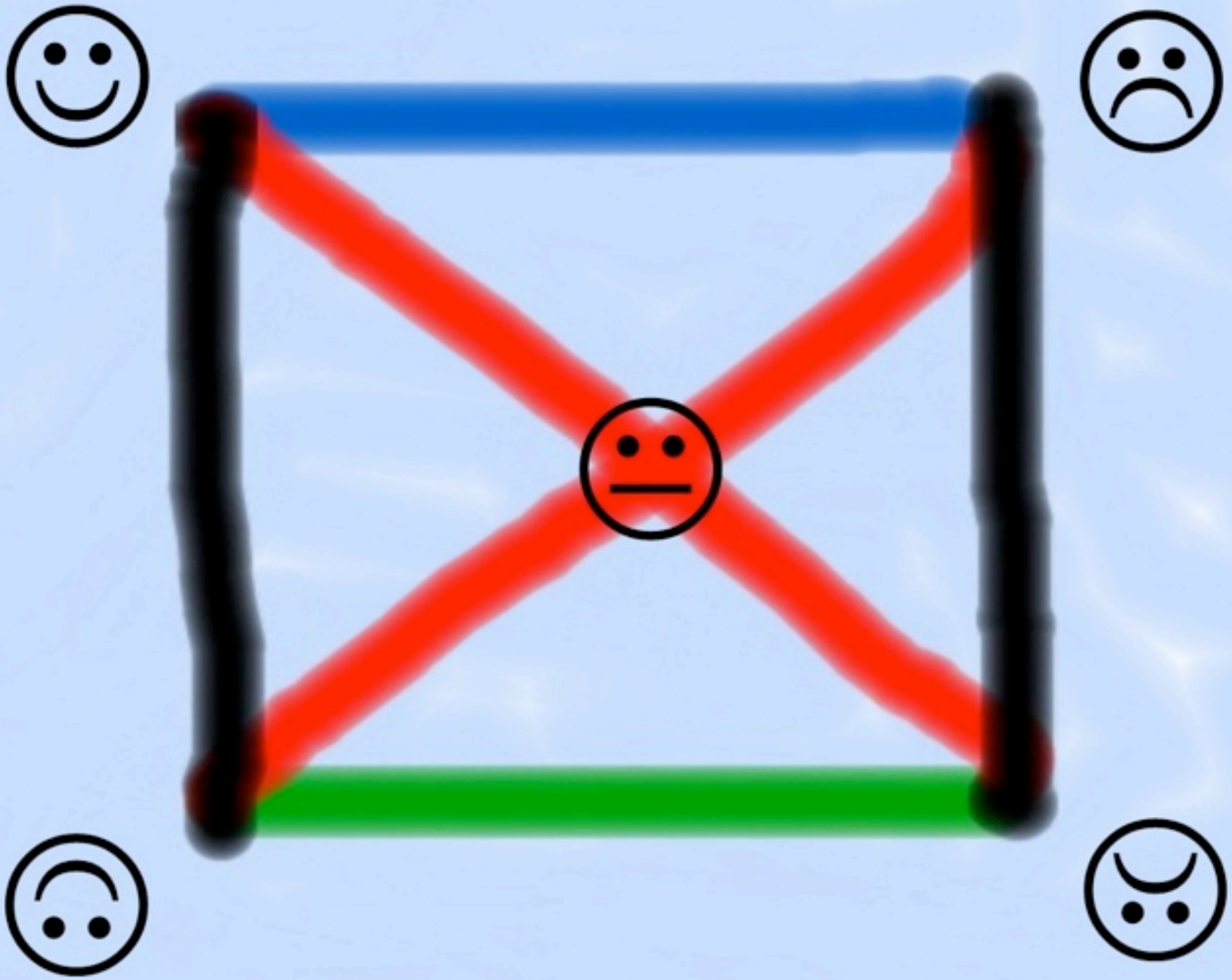
Where are we?

- A Curry-Howard explanation of pattern-matching and evaluation order [POPL'08]
- The ability to *mix* CBV and CBN [APAL]
- A better understanding of the Twelf-Coq (love-hate) relationship [LICS '08, with Dan Licata and Bob Harper]
- A guide to developing refinement type systems [draft paper on website...and hopefully, thesis!]

Where are we going?

- A systematic method for deriving practical programming languages via proof theory?
- Practical uses of duality in programming?
- Topological interpretation?
- Linguistic applications?

Thank you



ML in ML

```
type var = string
datatype pat = Pair of pat*pat  | Unit
                   | Inl of pat  | Inr of pat
                   | KVar of var
datatype vlu = Vlu of pat * sub
       and  kon = Kon of pat -> cmp
       and  sub = Sub of var -> kon
       and  cmp = Cmp of var * vlu
```