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 direct hit occurs
' 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 PlayerHit = 1 ELSE PlayerHit = 2
PLAY "MBO00L16EFGFEDC"
FOR i = 1 TO 8 * SclX#
CIRCLE (GorillaX(PlayerHit) + 3.5 * SclX# + XAdj, GorillaY(PlayerHit) + 7 * SclY# + YAdj), i, ExplosionColor, , , -1.57
LINE (GorillaX(PlayerHit) + 7 * SclX#, GorillaY(PlayerHit) + 9 * SclY# - i), ExplosionColor
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

\[ \text{ND} \equiv \lambda \]
The foundation of functional programming

\[ \text{ND} \equiv \lambda \]

... ML Lisp Haskell ...
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 $\lambda$ is an inadequate foundation
- Explain duality of proofs and refutations
- *Extract* a new foundational PL
- Profit
The trouble with $\lambda$
The foundation of functional programming

... ML Lisp Haskell ...

ND ≅ λ
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());
^CInterruption
*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

\textit{call-by-value}

\lambda

Haskell

\textit{call-by-name}

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 $A \land B$, prove both $A$ and $B$”
proof patterns

Describe how to prove a proposition

• “to prove $A \land B$, prove both $A$ and $B$”

• “to prove $A \lor B$, prove either $A$ or $B$”
proof patterns

Describe how to prove a proposition

• “to prove $A \land B$, prove both $A$ and $B$”

• “to prove $A \lor B$, prove either $A$ or $B$”

• “to prove $\neg A$, refute $A$”
proof patterns

Describe how to prove a proposition

- “to prove $A \land B$, prove both $A$ and $B$”
- “to prove $A \lor B$, 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 \(A \land B\), prove both \(A\) and \(B\)”
• “to prove \(A \lor B\), 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 \land (\neg B \lor \neg C) \text{ true}
the pattern is holey

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

\[
\neg A \text{ true} \quad \neg B \lor \neg C \text{ true} \\
\hline
\neg A \land (\neg B \lor \neg C) \text{ true}
\]
the pattern is holey

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

\[
\begin{align*}
\text{A false} & \\
\vdash \neg A \text{ true} & \quad \vdash \neg B \lor \neg C \text{ true} \\
\hline
\vdash \neg A \land (\neg B \lor \neg C) \text{ true}
\end{align*}
\]
the pattern is holey

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

\[
\begin{align*}
\text{A false} & \\
\hline
\neg A & \text{true} \\
\hline
\neg B & \text{true} \\
\hline
\neg B \lor \neg C & \text{true} \\
\hline
\neg A \land (\neg B \lor \neg C) & \text{true}
\end{align*}
\]
the pattern is holey

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

\[
\begin{align*}
A \text{ false} & \quad \text{B false} \\
\neg A & \quad \neg B \text{ true} \\
\neg A \land (\neg B \lor \neg C) & \text{ true}
\end{align*}
\]
There is a proof pattern for $A$, leaving holes for refutations of $A_1 \ldots A_n$
There is a proof pattern for $A$, leaving holes for refutations of $A_1, \ldots, A_n$
pattern axioms

- if $\Delta_1 \models A$ true and $\Delta_2 \models B$ true
  then $\Delta_1 \Delta_2 \models A \land B$ true

- if $\Delta \models A$ true then $\Delta \models A \lor B$ true

- if $\Delta \models B$ true then $\Delta \models A \lor B$ true

- $A$ false $\models \lnot A$ true

- $\cdot \models True$ true
proofs and refutations, informally...

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

To refute $A$, consider every proof pattern for $A$ and show that its holes can't be filled.
formal system

• $\Gamma \vdash A \text{ true }$ iff $\Delta \Vdash A \text{ true }$ and $\Gamma \vdash \Delta$
formal system

- $\Gamma \vdash A \text{ true }$ iff $\Delta \vdash A \text{ true }$ and $\Gamma \vdash \Delta$
- $\Gamma \vdash \Delta$ iff $A \text{ false } \in \Delta$ implies $\Gamma \vdash A \text{ false}$
formal system

• $\Gamma \vdash A \text{ true} \iff \Delta \models A \text{ true} \text{ and } \Gamma \vdash \Delta$

• $\Gamma \vdash \Delta \iff A \text{ false} \in \Delta \text{ implies } \Gamma \vdash A \text{ false}$

• $\Gamma \vdash A \text{ false} \iff \Delta \models A \text{ true} \text{ implies } \Gamma, \Delta \vdash \#$
formal system

• \( \Gamma \vdash A \ true \) iff \( \Delta \models A \ true \) and \( \Gamma \vdash \Delta \)

• \( \Gamma \vdash \Delta \) iff \( A \ false \in \Delta \) implies \( \Gamma \vdash A \ false \)

• \( \Gamma \vdash A \ false \) iff \( \Delta \models A \ true \) implies \( \Gamma, \Delta \vdash \# \)

• \( \Gamma \vdash \# \) iff \( A \ false \in \Gamma \) and \( \Gamma \vdash A \ true \)
a square of dualities

- \( \Gamma \vdash A \text{ true} \iff \Delta \vDash A \text{ true} \) and \( \Gamma \vdash \Delta \)
- \( \Gamma \vdash \Delta \iff A \text{ false} \in \Delta \) implies \( \Gamma \vdash A \text{ false} \)
- \( \Gamma \vdash A \text{ false} \iff \Delta \vDash A \text{ true} \) implies \( \Gamma, \Delta \vdash \# \)
- \( \Gamma \vdash \# \iff A \text{ false} \in \Gamma \) and \( \Gamma \vdash A \text{ true} \)
• \( \Gamma \vdash A \text{ true} \) iff \( \Delta \models A \text{ true} \) and \( \Gamma \vdash \Delta \)

• \( \Gamma \vdash \Delta \) iff \( A \text{ false} \in \Delta \) implies \( \Gamma \vdash A \text{ false} \)

• \( \Gamma \vdash A \text{ false} \) iff \( \Delta \models A \text{ true} \) implies \( \Gamma, \Delta \vdash \# \)

• \( \Gamma \vdash \# \) iff \( A \text{ false} \in \Gamma \) and \( \Gamma \vdash A \text{ true} \)
a square of dualities

• $\Gamma \vdash A \text{ true }$ iff $\Delta \not\models A \text{ true }$ and $\Gamma \not\vdash \Delta$

• $\Gamma \vdash \Delta$ iff $A \text{ false } \in \Delta$ implies $\Gamma \not\vdash A \text{ false }$

• $\Gamma \vdash A \text{ false }$ iff $\Delta \not\models A \text{ true }$ implies $\Gamma, \Delta \vdash \#$

• $\Gamma \vdash \#$ iff $A \text{ false } \in \Gamma$ and $\Gamma \not\vdash A \text{ true }$
a square of dualities

• $\Gamma \vdash A_{true}$ iff $\Delta \not\vdash A_{true}$ and $\Gamma \vdash \Delta$

• $\Gamma \vdash \Delta$ iff $A_{false} \in \Delta$ implies $\Gamma \vdash A_{false}$

• $\Gamma \vdash A_{false}$ iff $\Delta \not\vdash A_{true}$ implies $\Gamma, \Delta \vdash \#$

• $\Gamma \vdash \#$ iff $A_{false} \in \Gamma$ and $\Gamma \vdash A_{true}$
a square of dualities

• $\Gamma \vdash A \text{ true} \iff \Delta \models A \text{ true} \quad \text{and} \quad \Gamma \vdash \Delta$

• $\Gamma \vdash \Delta \iff A \text{ false} \in \Delta \implies \Gamma \vdash A \text{ false}$

• $\Gamma \vdash A \text{ false} \iff \Delta \models A \text{ true} \implies \Gamma, \Delta \vdash \#$

• $\Gamma \vdash \# \iff A \text{ false} \in \Gamma \quad \text{and} \quad \Gamma \vdash A \text{ true}$
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 $A \land B$, refute either $A$ or $B$”
refutation patterns

Describe how to refute a proposition

- “to refute $A \land B$, refute either A or B”
- “to refute $A \lor B$, refute both A and B”
refutation patterns

Describe how to refute a proposition

• “to refute $A \land B$, refute either $A$ or $B$”
• “to refute $A \lor B$, refute both $A$ and $B$”
• “to refute $\neg A$, prove $A$”
refutation patterns

Describe how to refute a proposition

• “to refute $A \land B$, refute either $A$ or $B$”
• “to refute $A \lor B$, 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 $A \land B$, refute either $A$ or $B$”
• “to refute $A \lor B$, refute both $A$ and $B$”
• “to refute $\neg A$, prove $A$”
• “to refute True, tough luck.”
• “to refute False, just did.”
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 $\Delta_1 \Delta_2 \vdash A \lor B \ false$

• $A \ true \vdash \neg A \ false$

• $\vdash False \ false$
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

- $\Gamma \vdash A \text{ true} \iff \Delta \models A \text{ true} \quad \text{and} \quad \Gamma \vdash \Delta$
- $\Gamma \vdash \Delta \iff A \text{ false} \in \Delta \implies \Gamma \vdash A \text{ false}$
- $\Gamma \vdash A \text{ false} \iff \Delta \models A \text{ true} \implies \Gamma, \Delta \vdash \#$
- $\Gamma \vdash \# \iff A \text{ false} \in \Gamma \quad \text{and} \quad \Gamma \vdash A \text{ true}$
strike that, reverse it...

• $\Gamma \vdash A \text{ false} \iff \Delta \vdash A \text{ false} \text{ and } \Gamma \vdash \Delta$

• $\Gamma \vdash \Delta \iff A \text{ true} \in \Delta \text{ implies } \Gamma \vdash A \text{ true}$

• $\Gamma \vdash A \text{ true} \iff \Delta \vdash A \text{ false} \text{ implies } \Gamma, \Delta \vdash \#$

• $\Gamma \vdash \# \iff A \text{ true} \in \Gamma \text{ and } \Gamma \vdash A \text{ false}$
Square of Opposition
Square of Opposition
So what does this have to do with programming?
Talk outline

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

<table>
<thead>
<tr>
<th>Logical judgment</th>
<th>Syntactic category</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \vdash A \text{ true} )</td>
<td>value pattern</td>
<td>VPat</td>
</tr>
<tr>
<td>( \Delta \vdash A \text{ false} )</td>
<td>continuation pattern</td>
<td>KPPat</td>
</tr>
<tr>
<td>( A \text{ false} \in \Delta )</td>
<td>continuation variable</td>
<td>KVar</td>
</tr>
<tr>
<td>( A \text{ true} \in \Delta )</td>
<td>value variable</td>
<td>VVar</td>
</tr>
<tr>
<td>( \Gamma \vdash A \text{ true} )</td>
<td>value</td>
<td>Val</td>
</tr>
<tr>
<td>( \Gamma \vdash A \text{ false} )</td>
<td>continuation</td>
<td>Kon</td>
</tr>
<tr>
<td>( \Gamma \vdash \Delta )</td>
<td>substitution</td>
<td>Sub</td>
</tr>
<tr>
<td>( \Gamma \vdash # )</td>
<td>computation</td>
<td>Cmp</td>
</tr>
</tbody>
</table>
Language #1
from proof patterns...

- if $\Delta_1 \vdash A$ true and $\Delta_2 \vdash B$ true then $\Delta_1 \Delta_2 \vdash A \land B$ true
- if $\Delta \vdash A$ true then $\Delta \vdash A \lor B$ true
- if $\Delta \vdash B$ true then $\Delta \vdash A \lor B$ true
- $A$ false $\vdash \neg A$ true
- $\cdot \vdash True$ true
...to value patterns

• if $p_1 \in \text{VPat}$ and $p_2 \in \text{VPat}$
  then $(p_1, p_2) \in \text{VPat}$

• if $p \in \text{VPat}$ then $\text{inl } p \in \text{VPat}$

• if $p \in \text{VPat}$ then $\text{inr } p \in \text{VPat}$

• if $k \in \text{KVar}$ then $k \in \text{VPat}$

• $() \in \text{VPat}$
e.g.,

\[ \neg A \land (\neg B \lor \neg C) \text{ true} \]

\[ \neg A \text{ true} \]

\[ \neg B \text{ true} \]

\[ \neg B \lor \neg C \text{ true} \]

\[ \neg A \land (\neg B \lor \neg C) \text{ true} \]

\[ \Rightarrow (k_1, \text{inl} \ k_2) \]
from logic...

• $\Gamma \vdash A \text{ true}$ \textbf{iff} $\Delta \models A \text{ true}$ \text{ and } $\Gamma \vdash \Delta$

• $\Gamma \vdash \Delta$ \textbf{iff} $A \text{ false} \in \Delta$ implies $\Gamma \vdash A \text{ false}$

• $\Gamma \vdash A \text{ false}$ \textbf{iff} $\Delta \models A \text{ true}$ implies $\Gamma, \Delta \vdash \#$

• $\Gamma \vdash \#$ \textbf{iff} $A \text{ false} \in \Gamma$ \text{ and } $\Gamma \vdash A \text{ true}$
...to language

- $\text{Val} = \text{VPat} \times \text{Sub}$
- $\text{Sub} = \text{KVar} \rightarrow \text{Kon}$
- $\text{Kon} = \text{VPat} \rightarrow \text{Cmp}$
- $\text{Cmp} = \text{KVar} \times \text{Val}$
...to language

- Val \( = \) VPat \( \times \) Sub
- Sub \( = \) KVar \( \rightarrow \) Kon
- Kon \( = \) VPat \( \rightarrow \) Cmp
- Cmp \( = \) KVar \( \times \) Val
Is that *really* a language?
Yes, trust me.

Like $\lambda$, it is minimalistic, but unlike $\lambda$ it...

- has inherent support for products, sums, and pattern-matching
- inherently enforces \textit{call-by-value}

(NB: can think of image of CBV CPS transform)
Yes, trust me.

Like $\lambda$, it is minimalistic, but unlike $\lambda$ 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 $\Delta \models A \text{ false}$ then $\Delta \models A \land B \text{ false}$

• if $\Delta \models B \text{ false}$ then $\Delta \models A \land B \text{ false}$

• if $\Delta_1 \models A \text{ false}$ and $\Delta_2 \models B \text{ false}$ then $\Delta_1 \Delta_2 \models A \lor B \text{ false}$

• $A \text{ true} \models \neg A \text{ false}$

• $\cdot \models \text{ False false}$
...to continuation patterns

- if \( d \in \text{KPat} \) then \( \text{fst} \ d \in \text{KPat} \)
- if \( d \in \text{KPat} \) then \( \text{snd} \ d \in \text{KPat} \)
- if \( d_1 \in \text{KPat} \) and \( d_2 \in \text{KPat} \) then \( [d_1, d_2] \in \text{KPat} \)
- if \( x \in \text{VVar} \) then \( x \in \text{KPat} \)
- \( [] \in \text{KPat} \)
okay so continuation patterns are a little weird...
from logic...

• $\Gamma \vdash A \ false$ iff $\Delta \not\vdash A \ false$ and $\Gamma \vdash \Delta$

• $\Gamma \vdash \Delta$ iff $A \ true \in \Delta$ implies $\Gamma \vdash A \ true$

• $\Gamma \vdash A \ true$ iff $\Delta \not\vdash A \ false$ implies $\Gamma, \Delta \vdash \#$

• $\Gamma \vdash \#$ iff $A \ true \in \Gamma$ and $\Gamma \vdash A \ false$
...to language

- $\text{Kon} = \text{KPat} \times \text{Sub}$
- $\text{Sub} = \text{VVar} \rightarrow \text{Val}$
- $\text{Val} = \text{KPat} \rightarrow \text{Cmp}$
- $\text{Cmp} = \text{VVar} \times \text{Kon}$
...to language

- $\text{Kon} = \text{KPat} \times \text{Sub}$
- $\text{Sub} = \text{VVar} \rightarrow \text{Val}$
- $\text{Val} = \text{KPat} \rightarrow \text{Cmp}$
- $\text{Cmp} = \text{VVar} \times \text{Kon}$
Recap
the CBV square

<table>
<thead>
<tr>
<th>Val</th>
<th>Kon</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Val} = \text{VPat} \times \text{Sub}$</td>
<td>$\text{Kon} = \text{VPat} \to \text{Cmp}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub</th>
<th>Cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Sub} = \text{KVar} \to \text{Kon}$</td>
<td>$\text{Cmp} = \text{KVar} \times \text{Val}$</td>
</tr>
</tbody>
</table>
the CBN square

<table>
<thead>
<tr>
<th>Val</th>
<th>=</th>
<th>KPat → Cmp</th>
<th>Kon</th>
<th>=</th>
<th>KPat × Sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub</td>
<td>=</td>
<td>VVar → Val</td>
<td>Cmp</td>
<td>=</td>
<td>VVar × Kon</td>
</tr>
</tbody>
</table>
CBV-CBN duality

\[ \text{Val}^+ = \text{VPat} \times \ldots \]

\[ \text{Kon}^+ = \text{VPat} \rightarrow \ldots \]

\[ \text{Val}^- = \text{KPat} \rightarrow \ldots \]

\[ \text{Kon}^- = \text{KPat} \times \ldots \]
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