CSE 505: Programming Languages

Lecture 15 — The Curry-Howard Isomorphism

Zach Tatlock
Winter 2015

We are Language Designers!

What have we done?

- Define a programming language
  - we were fairly formal
  - still pretty close to OCaml if you squint real hard

We are Language Designers!

What have we done?

- Define a programming language
  - we were fairly formal
  - still pretty close to OCaml if you squint real hard
- Define a type system
  - **outlaw** *bad programs* that "get stuck"
  - sound: no typable programs get stuck
  - incomplete: knocked out some OK programs too, ohwell

Elsewhere in the Universe (or the other side of campus)

What do logicians do?

- Define formal logics
  - tools to precisely state propositions

Elsewhere in the Universe (or the other side of campus)

What do logicians do?

- Define formal logics
  - tools to precisely state propositions
- Define proof systems
  - tools to figure out which propositions are true

Elsewhere in the Universe (or the other side of campus)

What do logicians do?

- Define formal logics
  - tools to precisely state propositions
- Define proof systems
  - tools to figure out which propositions are true

Turns out, we did that too!

Punchline

We are accidental logicians!

Punchline

We are accidental logicians!

The Curry-Howard Isomorphism

- Proofs : Propositions :: Programs : Types
- proofs are to propositions as programs are to types

Woah. Back up a second. Logic?!

Woah. Back up a second. Logic?!

Let's trim down our (explicitly typed) simply-typed λ-calculus to:

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.\ e \mid e\ e \\
 & \mid & (e, e) \mid e.1 \mid e.2 \\
 & \mid & \mathsf{A}(e) \mid \mathsf{B}(e) \mid \textbf{match } e \textbf{ with } \mathsf{A}x.\ e \mid \mathsf{B}x.\ e
\end{array}
$$

$$
\tau \ ::= \ b \mid \tau \to \tau \mid \tau * \tau \mid \tau + \tau
$$

- Lambdas, Pairs, and Sums
- Any number of base types $b_1, b_2, \ldots$
- No constants (can add one or more if you want)
- No `fix`

Woah. Back up a second. Logic?!

Let's trim down our (explicitly typed) simply-typed $\boldsymbol{\lambda}$-calculus to:

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.\ e \mid e\ e \\ & \mid & (e, e) \mid e.1 \mid e.2 \\ & \mid & \mathsf{A}(e) \mid \mathsf{B}(e) \mid \textbf{match } e \textbf{ with } \mathsf{A}x.\ e \mid \mathsf{B}x.\ e \end{array}$$

$$\tau \quad ::= \quad b \mid \tau \to \tau \mid \tau * \tau \mid \tau + \tau$$

- Lambdas, Pairs, and Sums
- Any number of base types $b_1, b_2, \ldots$
- No constants (can add one or more if you want)
- No `fix`

What good is this?!

Well, even sans constants, plenty of terms type-check with $\Gamma = \cdot$

$$\boldsymbol{\lambda x{:}b.\ x}$$

has type

$$\boldsymbol{\lambda x{:}b.\ x}$$

has type

$$\boldsymbol{b \to b}$$

$$\boldsymbol{\lambda x{:}b_1.\ \lambda f{:}b_1 \to b_2.\ f\ x}$$

has type

$$\boldsymbol{\lambda x{:}b_1.\ \lambda f{:}b_1 \to b_2.\ f\ x}$$

has type

$$\boldsymbol{b_1 \to (b_1 \to b_2) \to b_2}$$

$$\boldsymbol{\lambda x{:}b_1 \to b_2 \to b_3.\ \lambda y{:}b_2.\ \lambda z{:}b_1.\ x\ z\ y}$$

has type

$$\boldsymbol{\lambda x{:}b_1 \to b_2 \to b_3.\ \lambda y{:}b_2.\ \lambda z{:}b_1.\ x\ z\ y}$$

has type

$$\boldsymbol{(b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3}$$

$$\lambda x{:}b_1.\ (\mathsf{A}(x), \mathsf{A}(x))$$

has type

$$\lambda x{:}b_1.\ (\mathsf{A}(x), \mathsf{A}(x))$$

has type

$$b_1 \rightarrow ((b_1 + b_7) * (b_1 + b_4))$$

$$\boldsymbol{\lambda f{:}b_1 \rightarrow b_3.\ \lambda g{:}b_2 \rightarrow b_3.\ \lambda z{:}b_1 + b_2.}$$
$$\textbf{(match } \boldsymbol{z} \textbf{ with A}\boldsymbol{x.\ f\ x} \mid \textbf{B}\boldsymbol{x.\ g\ x}\textbf{)}$$

has type

$$\lambda f{:}b_1 \rightarrow b_3.\ \lambda g{:}b_2 \rightarrow b_3.\ \lambda z{:}b_1 + b_2.$$
$$(\textbf{match}\ z\ \textbf{with}\ \mathsf{A}x.\ f\ x \mid \mathsf{B}x.\ g\ x)$$

has type

$$(b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3$$

$$\lambda x{:}b_1 * b_2.\ \lambda y{:}b_3.\ ((y, x.1), x.2)$$

has type

$$\lambda x{:}b_1 * b_2.\ \lambda y{:}b_3.\ ((y, x.1), x.2)$$

has type

$$(b_1 * b_2) \to b_3 \to ((b_3 * b_1) * b_2)$$

Empty and Nonempty Types

Just saw a few "nonempty" types

- τ *nonempy* if closed term e has type τ
- τ *empty* otherwise

Empty and Nonempty Types

Just saw a few "nonempty" types

- τ *nonempy* if closed term e has type τ
- τ *empty* otherwise

Are there any empty types?

Empty and Nonempty Types

Just saw a few "nonempty" types

- τ *nonempy* if closed term e has type τ
- τ *empty* otherwise

Are there any empty types?

Sure! $\quad b_1 \quad b_1 \to b_2 \quad b_1 \to (b_2 \to b_1) \to b_2$

Empty and Nonempty Types

Just saw a few "nonempty" types

- τ *nonempy* if closed term e has type τ
- τ *empty* otherwise

Are there any empty types?

Sure! $\quad b_1 \quad b_1 \to b_2 \quad b_1 \to (b_2 \to b_1) \to b_2$

What does this one mean?

$$b_1 + (b_1 \to b_2)$$

Empty and Nonempty Types

Just saw a few "nonempty" types

- τ *nonempy* if closed term e has type τ
- τ *empty* otherwise

Are there any empty types?

Sure! $\quad b_1 \quad b_1 \to b_2 \quad b_1 \to (b_2 \to b_1) \to b_2$

What does this one mean?

$$b_1 + (b_1 \to b_2)$$

I wonder if there's any way to distinguish empty vs. nonempty...

Empty and Nonempty Types

Just saw a few "nonempty" types

- τ *nonempy* if closed term e has type τ
- τ *empty* otherwise

Are there any empty types?

Sure! $\quad b_1 \quad b_1 \to b_2 \quad b_1 \to (b_2 \to b_1) \to b_2$

What does this one mean?

$$b_1 + (b_1 \to b_2)$$

I wonder if there's any way to distinguish empty vs. nonempty...

Ohwell, now for a *totally irrelevant* tangent!

Totally irrelevant tangent.

Propositional Logic

Propositional Logic

Suppose we have some set b of basic propositions $b_1, b_2, \ldots$

- e.g. "ML is better than Haskell"

Propositional Logic

Suppose we have some set b of basic propositions $b_1, b_2, \ldots$

- e.g. "ML is better than Haskell"

Then, using standard operators $\supset, \wedge, \vee$, we can define formulas:

$$p \quad ::= \quad b \mid p \supset p \mid p \wedge p \mid p \vee p$$

- e.g. "ML is better than Haskell" $\wedge$ "Haskell is not pure"

Propositional Logic

Suppose we have some set b of basic propositions $b_1, b_2, \ldots$

- e.g. "ML is better than Haskell"

Then, using standard operators $\supset, \wedge, \vee$, we can define formulas:

$$p \quad ::= \quad b \mid p \supset p \mid p \wedge p \mid p \vee p$$

- e.g. "ML is better than Haskell" $\wedge$ "Haskell is not pure"

Some formulas are *tautologies*: by virtue of their structure, they are always true regardless of the truth of their constituent propositions.

- e.g. $p_1 \supset p_1$

Propositional Logic

Suppose we have some set b of basic propositions $b_1, b_2, \ldots$

- e.g. "ML is better than Haskell"

Then, using standard operators $\supset, \wedge, \vee$, we can define formulas:

$$p \quad ::= \quad b \mid p \supset p \mid p \wedge p \mid p \vee p$$

- e.g. "ML is better than Haskell" $\wedge$ "Haskell is not pure"

Some formulas are *tautologies*: by virtue of their structure, they are always true regardless of the truth of their constituent propositions.

- e.g. $p_1 \supset p_1$

Not too hard to build a *proof system* to establish tautologyhood.

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1}$$

Proof System

$$\boldsymbol{\Gamma} \quad ::= \quad \cdot \mid \boldsymbol{\Gamma}, \boldsymbol{p}$$

$$\boxed{\boldsymbol{\Gamma} \vdash \boldsymbol{p}}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{p_1} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{p_2}}{\boldsymbol{\Gamma} \vdash \boldsymbol{p_1} \wedge \boldsymbol{p_2}} \qquad \frac{\boldsymbol{\Gamma} \vdash \boldsymbol{p_1} \wedge \boldsymbol{p_2}}{\boldsymbol{\Gamma} \vdash \boldsymbol{p_1}} \qquad \frac{\boldsymbol{\Gamma} \vdash \boldsymbol{p_1} \wedge \boldsymbol{p_2}}{\boldsymbol{\Gamma} \vdash \boldsymbol{p_2}}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_2}$$

$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 \vee p_2}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_2}$$

$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 \vee p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 \vee p_2}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_2}$$

$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 \vee p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 \vee p_2}$$

$$\frac{\Gamma \vdash p_1 \vee p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_2}$$

$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 \vee p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 \vee p_2}$$

$$\frac{\Gamma \vdash p_1 \vee p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}$$

$$\frac{p \in \Gamma}{\Gamma \vdash p}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \land p_2} \qquad \frac{\Gamma \vdash p_1 \land p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 \land p_2}{\Gamma \vdash p_2}$$

$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 \lor p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 \lor p_2}$$

$$\frac{\Gamma \vdash p_1 \lor p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}$$

$$\frac{p \in \Gamma}{\Gamma \vdash p} \qquad \frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \supset p_2}$$

Proof System

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash p_1 \wedge p_2} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_1} \qquad \frac{\Gamma \vdash p_1 \wedge p_2}{\Gamma \vdash p_2}$$

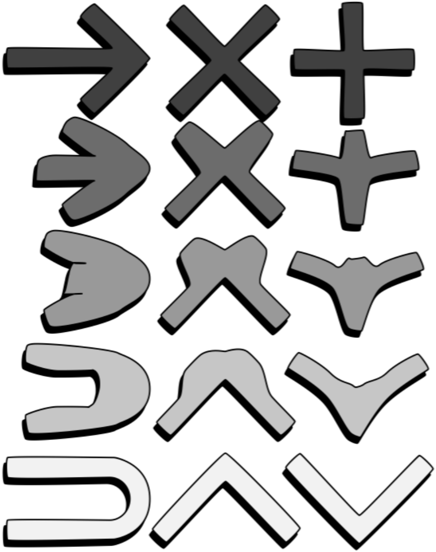
$$\frac{\Gamma \vdash p_1}{\Gamma \vdash p_1 \vee p_2} \qquad \frac{\Gamma \vdash p_2}{\Gamma \vdash p_1 \vee p_2}$$

$$\frac{\Gamma \vdash p_1 \vee p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3}{\Gamma \vdash p_3}$$

$$\frac{p \in \Gamma}{\Gamma \vdash p} \qquad \frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \supset p_2} \qquad \frac{\Gamma \vdash p_1 \supset p_2 \qquad \Gamma \vdash p_1}{\Gamma \vdash p_2}$$

Wait a second...

Wait a second...



Wait a second... ZOMG!

That's *exactly* our type system! Just erase terms, change each τ to a p, and translate $\rightarrow$ to $\supset$, $*$ to $\wedge$, $+$ to $\vee$.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textsf{match } e \textsf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

What does it all mean? The Curry-Howard Isomorphism.

- Given a well-typed closed term, take the typing derivation, erase the terms, and have a propositional-logic proof
- Given a propositional-logic proof, there exists a closed term with that type
- A term that type-checks is a *proof* — it tells you exactly how to derive the logic formula corresponding to its type

What does it all mean? The Curry-Howard Isomorphism.

- Given a well-typed closed term, take the typing derivation, erase the terms, and have a propositional-logic proof
- Given a propositional-logic proof, there exists a closed term with that type
- A term that type-checks is a *proof* — it tells you exactly how to derive the logic formula corresponding to its type
- Constructive (hold that thought) propositional logic and simply-typed lambda-calculus with pairs and sums are *the same thing*.
  - Computation and logic are *deeply* connected
  - $\boldsymbol{\lambda}$ is no more or less made up than implication
- Revisit our examples under the logical interpretation...

$$\boldsymbol{\lambda x{:}b.\ x}$$

is a proof that

$$\boldsymbol{b \rightarrow b}$$

$$\boldsymbol{\lambda x{:}b_1.\ \lambda f{:}b_1 \to b_2.\ f\ x}$$

is a proof that

$$\boldsymbol{b_1 \to (b_1 \to b_2) \to b_2}$$

$$\lambda x{:}b_1 \to b_2 \to b_3.\ \lambda y{:}b_2.\ \lambda z{:}b_1.\ x\ z\ y$$

is a proof that

$$(b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3$$

$$\lambda x{:}b_1.\ (\mathsf{A}(x), \mathsf{A}(x))$$

is a proof that

$$b_1 \rightarrow ((b_1 + b_7) * (b_1 + b_4))$$

$$\lambda f{:}b_1 \to b_3.\ \lambda g{:}b_2 \to b_3.\ \lambda z{:}b_1 + b_2.$$
$$(\textbf{match } z \textbf{ with } \mathsf{A}x.\ f\ x \mid \mathsf{B}x.\ g\ x)$$

is a proof that

$$(b_1 \to b_3) \to (b_2 \to b_3) \to (b_1 + b_2) \to b_3$$

$$\lambda x{:}b_1 * b_2.\ \lambda y{:}b_3.\ ((y, x.1), x.2)$$

is a proof that

$$(b_1 * b_2) \rightarrow b_3 \rightarrow ((b_3 * b_1) * b_2)$$

So what?

Because:

- This is just fascinating (glad I'm not a dog)
- Don't think of logic and computing as distinct fields
- Thinking "the other way" can help you know what's possible/impossible
- Can form the basis for theorem provers
- Type systems should not be *ad hoc* piles of rules!

So what?

Because:

- This is just fascinating (glad I'm not a dog)
- Don't think of logic and computing as distinct fields
- Thinking "the other way" can help you know what's possible/impossible
- Can form the basis for theorem provers
- Type systems should not be *ad hoc* piles of rules!

So, every typed $\boldsymbol{\lambda}$-calculus is a proof system for some logic...

Is STLC with pairs and sums a *complete* proof system for propositional logic? Almost...

Classical vs. Constructive

Classical propositional logic has the "law of the excluded middle":

$$\frac{}{\Gamma \vdash p_1 + (p_1 \to p_2)}$$

(Think "$p + \neg p$" – also equivalent to double-negation $\neg\neg p \to p$)

Classical vs. Constructive

Classical propositional logic has the "law of the excluded middle":

$$\frac{}{\Gamma \vdash p_1 + (p_1 \to p_2)}$$

(Think "$p + \neg p$" – also equivalent to double-negation $\neg\neg p \to p$)

STLC does not support this law; for example, no closed expression has type $b_1 + (b_1 \to b_2)$

Classical vs. Constructive

Classical propositional logic has the "law of the excluded middle":

$$\frac{}{\Gamma \vdash p_1 + (p_1 \to p_2)}$$

(Think "$p + \neg p$" – also equivalent to double-negation $\neg\neg p \to p$)

STLC does not support this law; for example, no closed expression has type $b_1 + (b_1 \to b_2)$

Logics without this rule are called *constructive*. They're useful because proofs "know how the world is" and "are executable" and "produce examples"

Classical vs. Constructive

Classical propositional logic has the “law of the excluded middle”:

$$\frac{}{\Gamma \vdash p_1 + (p_1 \to p_2)}$$

(Think “$p + \neg p$” – also equivalent to double-negation $\neg\neg p \to p$)

STLC does not support this law; for example, no closed expression has type $b_1 + (b_1 \to b_2)$

Logics without this rule are called *constructive*. They’re useful because proofs “know how the world is” and “are executable” and “produce examples”

Can still “branch on possibilities” by making the excluded middle an explicit assumption:

$$((p_1 + (p_1 \to p_2)) * (p_1 \to p_3) * ((p_1 \to p_2) \to p_3)) \to p_3$$

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers a and b such that a^b is rational.

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers a and b such that a^b is rational.

Classical Proof:

Let $x = \sqrt{2}$. *Either* x^x *is rational or it is irrational.*
If x^x *is rational, let* $a = b = \sqrt{2}$, *done.*
If x^x *is irrational, let* $a = x^x$ *and* $b = x$. *Since*
$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^{2} = 2$, *done.*

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers a and b such that a^b is rational.

Classical Proof:

Let $x = \sqrt{2}$. *Either* x^x *is rational or it is irrational.*
If x^x *is rational, let* $a = b = \sqrt{2}$, *done.*
If x^x *is irrational, let* $a = x^x$ *and* $b = x$. *Since*
$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^{2} = 2$, *done.*

Well, I guess we know there are *some* a and b satisfying the theorem... but which ones?

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers a and b such that a^b is rational.

Classical Proof:

> *Let* $x = \sqrt{2}$. *Either* x^x *is rational or it is irrational.*
> *If* x^x *is rational, let* $a = b = \sqrt{2}$, *done.*
> *If* x^x *is irrational, let* $a = x^x$ *and* $b = x$. *Since*
> $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^{2} = 2$, *done.*

Well, I guess we know there are *some* a and b satisfying the theorem... but which ones? LAME.

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers a and b such that a^b is rational.

Classical Proof:

Let $x = \sqrt{2}$*. Either* x^x *is rational or it is irrational.*
If x^x *is rational, let* $a = b = \sqrt{2}$*, done.*
If x^x *is irrational, let* $a = x^x$ *and* $b = x$*. Since*
$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^2 = 2$*, done.*

Well, I guess we know there are *some* a and b satisfying the theorem... but which ones? LAME.

Constructive Proof:

Let $a = \sqrt{2}$*,* $b = \log_2 9$*.*
Since $\sqrt{2}^{\log_2 9} = 9^{\log_2 \sqrt{2}} = 9^{\log_2(2^{0.5})} = 9^{0.5} = 3$*, done.*

Classical vs. Constructive, an Example

Theorem: There exist irrational numbers a and b such that a^b is rational.

Classical Proof:

> *Let* $x = \sqrt{2}$. *Either* x^x *is rational or it is irrational.*
> *If* x^x *is rational, let* $a = b = \sqrt{2}$, *done.*
> *If* x^x *is irrational, let* $a = x^x$ *and* $b = x$. *Since*
> $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^2 = 2$, *done.*

Well, I guess we know there are *some* a and b satisfying the theorem... but which ones? LAME.

Constructive Proof:

> *Let* $a = \sqrt{2}$, $b = \log_2 9$.
> *Since* $\sqrt{2}^{\log_2 9} = 9^{\log_2 \sqrt{2}} = 9^{\log_2 (2^{0.5})} = 9^{0.5} = 3$, *done.*

To prove that something exists, we actually had to produce it. SWEET.

Classical vs. Constructive, a Perspective

Constructive logic allows us to distinguish between things that classical logic lumps together.

Classical vs. Constructive, a Perspective

Constructive logic allows us to distinguish between things that classical logic lumps together.

Consider "P is true." vs. "It would be absurd if P were false."

- P vs. $\neg\neg P$

Classical vs. Constructive, a Perspective

Constructive logic allows us to distinguish between things that classical logic lumps together.

Consider "P is true." vs. "It would be absurd if P were false."

- P vs. $\neg\neg P$

Those are different things, but classical logic can't tell.

Classical vs. Constructive, a Perspective

Constructive logic allows us to distinguish between things that classical logic lumps together.

Consider "P is true." vs. "It would be absurd if P were false."

- P vs. $\neg\neg P$

Those are different things, but classical logic can't tell.

Our friends Gödel and Gentzen gave us this nice result:

P is provable in classical logic iff $\neg\neg P$ is provable in constructive logic.

Fix

A "non-terminating proof" is no proof at all.

Remember the typing rule for **fix**:

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ e : \tau}$$

That let's us prove anything! Example: **fix** $\lambda x{:}b.\ x$ has type b

So the "logic" is *inconsistent* (and therefore worthless)

Related: In ML, a value of type `'a` never terminates normally (raises an exception, infinite loop, etc.)

```
let rec f x = f x
let z = f 0
```

Last word on Curry-Howard

It's not just STLC and constructive propositional logic

Every logic has a correspondng typed $\boldsymbol{\lambda}$ calculus (and no consistent logic has something as "powerful" as **fix**).

- Example: When we add universal types ("generics") in a later lecture, that corresponds to adding universal quantification

Last word on Curry-Howard

It's not just STLC and constructive propositional logic

Every logic has a correspondng typed $\boldsymbol{\lambda}$ calculus (and no consistent logic has something as "powerful" as **fix**).

- Example: When we add universal types ("generics") in a later lecture, that corresponds to adding universal quantification

If you remember one thing: the typing rule for function application is *modus ponens*