

# <u>Outline</u>

A couple left-over topics from last lecture:

- Sums in "real" programming languages
- A fuller explanation of why it's called fix

Two totally different topics:

- Curry-Howard Isomorphism
  - Types are propositions
  - Programs are proofs
- Evaluation contexts, explicit stacks, and first-class continuations

### Recall sums

 $e ::= \ldots | A(e) | B(e) |$  match e with Ax. e | Bx. e $v ::= \ldots | \mathsf{A}(v) | \mathsf{B}(v)$  $\tau ::= \ldots \mid \tau_1 + \tau_2$ match A(v) with  $Ax. e_1 \mid By. e_2 \rightarrow e_1[v/x]$ match B(v) with Ax.  $e_1 \mid By. e_2 \rightarrow e_2[v/y]$  $\frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$  $e \rightarrow e'$ match e with Ax.  $e_1 \mid By. e_2 \rightarrow match e'$  with Ax.  $e_1 \mid By. e_2$  $\Gamma \vdash e : \tau_1$  $\Gamma \vdash e : \tau_2$  $\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2 \qquad \qquad \Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2$  $\Gamma dash e: au_1 + au_2 \qquad \Gamma, x{:} au_1 dash e_1: au \qquad \Gamma, y{:} au_2 dash e_2: au$  $\Gamma \vdash$  match e with Ax.  $e_1 \mid$  By.  $e_2 : \tau$ 

#### What are sums for?

- Pairs, structs, records, aggregates are fundamental data-builders
- Sums are just as fundamental: "this or that not both"
- You have seen how Caml does sums (datatypes)
- Worth showing how C and Java do the same thing
  - A primitive in one language is an idiom in another

### Sums in C

```
type t = A of t1 | B of t2 | C of t3
match e with A x \rightarrow ...
One way in C:
 struct t {
   enum \{A, B, C\}
                               tag;
   union \{t1 a; t2 b; t3 c;\} data;
 };
... switch(e->tag){ case A: t1 x=e->data.a; ...

    No static checking that tag is obeyed

 • As fat as the fattest variant (avoidable with casts)
    – Mutation costs us again!
 • Shameless plug: Cyclone has ML-style datatypes
```

#### Sums in Java

```
type t = A of t1 | B of t2 | C of t3 match e with A x \rightarrow ...
```

One way in Java (t4 is the match-expression's type):

```
abstract class t {abstract t4 m();}
class A extends t { t1 x; t4 m(){...}}
class B extends t { t2 x; t4 m(){...}}
class C extends t { t3 x; t4 m(){...}}
... e.m() ...
```

- A new method for each match expression
- Supports extensibility via new variants (subclasses) instead of extensibility via new operations (match expressions)

# Recall Fix

Note: Like let rec but using a  $\lambda$  to bind the name

 $e := \dots | \text{fix } e$ 

e  ightarrow e'	
$fix \ e  ightarrow fix \ e'$	fix $\lambda x.~e  ightarrow e[$ fix $\lambda x.~e/x]$
	$rac{\Gammadash e: au o au}{\Gammadash  ext{ fix }e: au}$

Factorial example:

fix  $\lambda f$ .  $\lambda n$ . if n < 1 then 1 else n \* (f(n-1))

- Operationally, substitution unrolls the recursion one level
- For type system,  $\lambda f. \lambda n.$  if n < 1 then 1 else n \* (f(n-1))has type (int  $\rightarrow$  int)  $\rightarrow$  (int  $\rightarrow$  int).

# Why called fix?

*My* slide in the last lecture could have explained fix-points much better...

In math, the fix-point of a function g is an x such that g(x) = x.

- This makes sense only if g has type au o au for some au.
- A particular g could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type  $int \rightarrow int$ :
  - $\lambda x. x + 1$  has no fix-points
  - $\lambda x. x * 0$  has one fix-point
  - $\lambda x$ . absolute\_value(x) has an infinite number of fix-points
  - $\lambda x$ . if x < 10 && x > 0 then x else 0 has 10 fix-points

## Higher types

At higher types like (int  $\rightarrow$  int)  $\rightarrow$  (int  $\rightarrow$  int), the notion of fix-point is exactly the same (but harder to think about)

• For what inputs f of type  $\operatorname{int} \to \operatorname{int}$  is g(f) = f.

Examples:

- $\lambda f. \ \lambda x. \ (f \ x) + 1$  has no fix-points
- $\lambda f. \lambda x. (f x) * 0$  (or just  $\lambda f. \lambda x. 0$ ) has 1 fix-point

- The function that always returns 0

- In math, there is exactly one such function (cf. equivalence)
- λf. λx. absolute\_value(f x) has an infinite number of fix-points: Any function that never returns a negative result

### Back to factorial

Now, what are the fix-points of  $\lambda f. \lambda x.$  if x < 1 then 1 else x \* (f(x - 1))?

It turns out there is exactly one (in math): the factorial function!

And fix  $\lambda f$ .  $\lambda x$ . if x < 1 then 1 else x \* (f(x - 1)) behaves just like the factorial function, i.e., it behaves just like the fix-point of  $\lambda f$ .  $\lambda x$ . if x < 1 then 1 else x \* (f(x - 1)).

(This isn't really important, but I like explaining terminology and showing that programming is deeply connected to mathematics.)

# Curry-Howard Isomorphism

What we did:

- Define a programming language
- Define a type system to rule out programs we don't want

What logicians do:

• Define a logic (a way to state propositions)

– Example: Propositional logic  $p ::= b \mid p \wedge p \mid p \lor p \mid p \rightarrow p$ 

• Define a proof system (a way to prove propositions)

But it turns out we did that too!

Slogans:

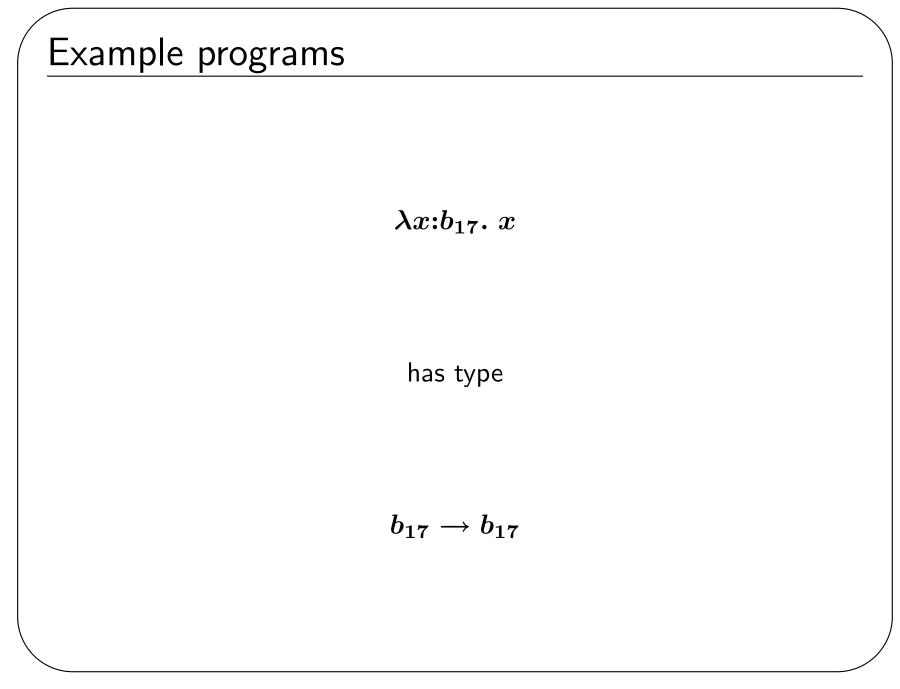
- "Propositions are Types"
- "Proofs are Programs"

# A slight variant

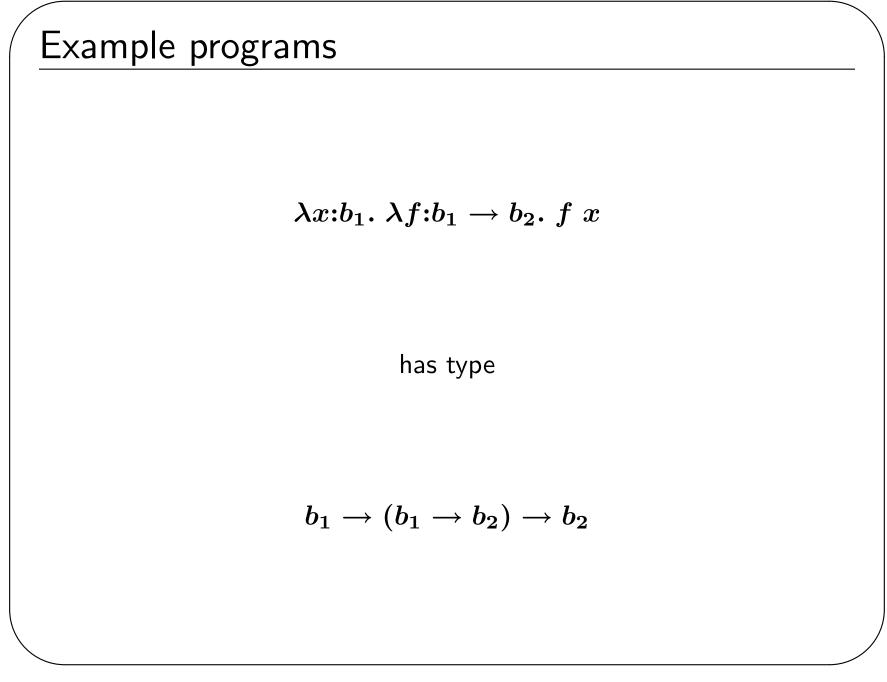
Let's take the explicitly typed ST $\lambda$ C with base types  $b_1, b_2, \ldots$ , *no constants*, pairs, and sums

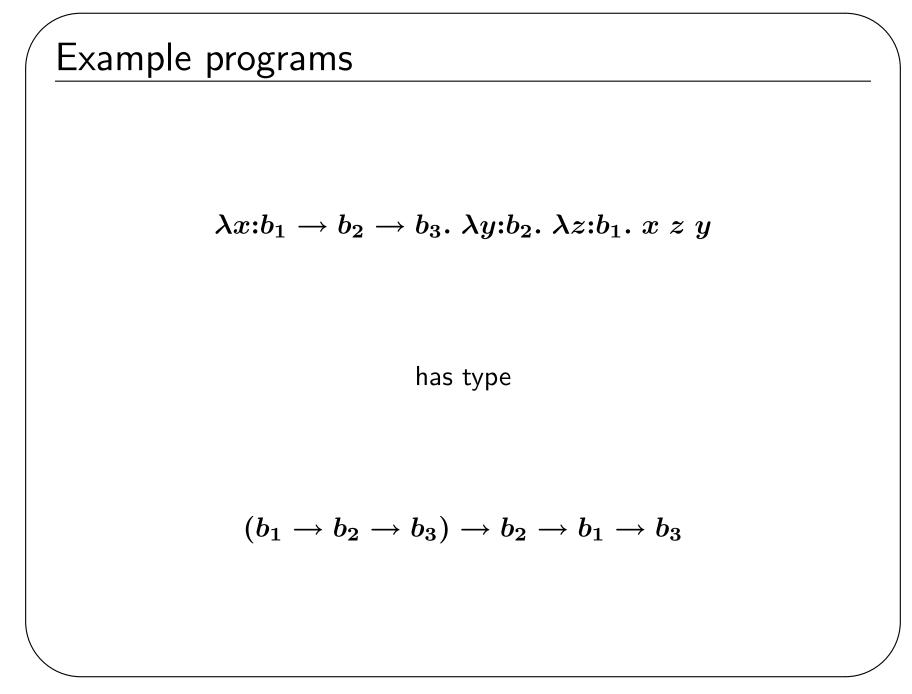
$$e ::= x | \lambda x. e | e e$$
  
| (e, e) | e.1 | e.2  
| A(e) | B(e) | match e with Ax. e | Bx. e  
$$\tau ::= b | \tau \rightarrow \tau | \tau * \tau | \tau + \tau$$

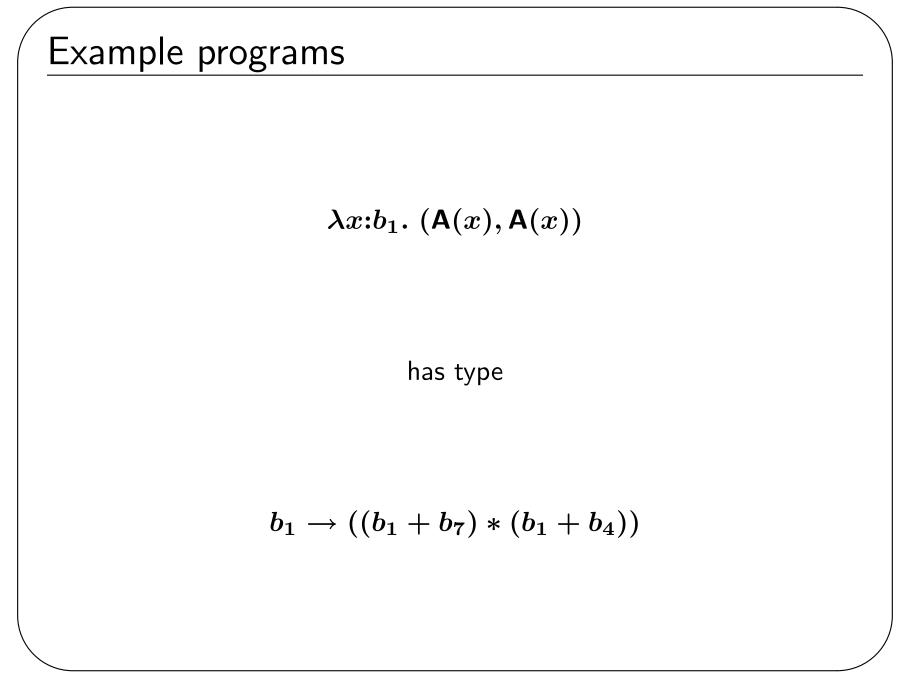
Even without constants, plenty of terms type-check with  $\Gamma=\cdot\ ...$ 

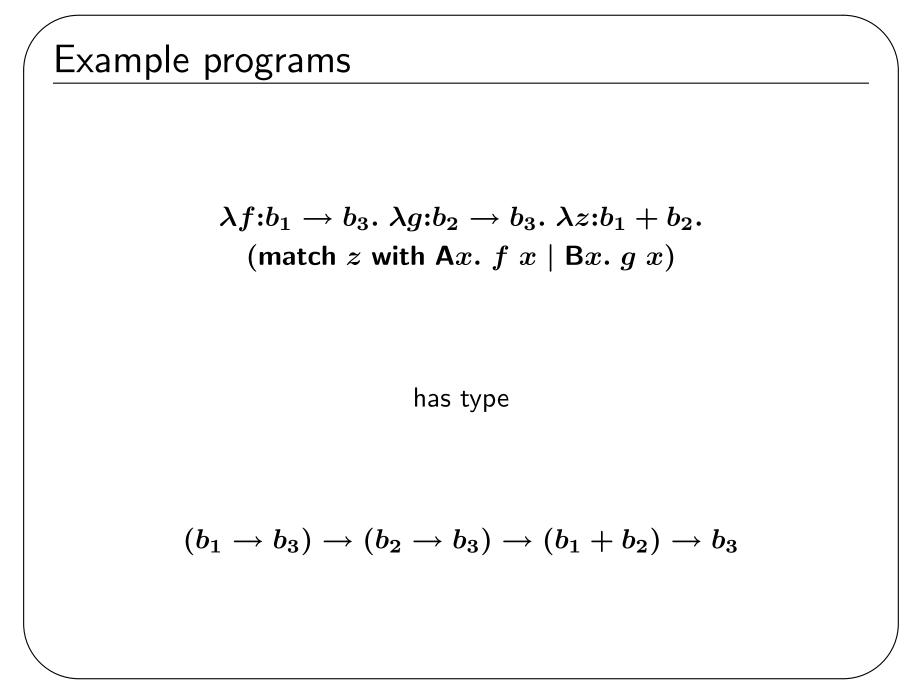


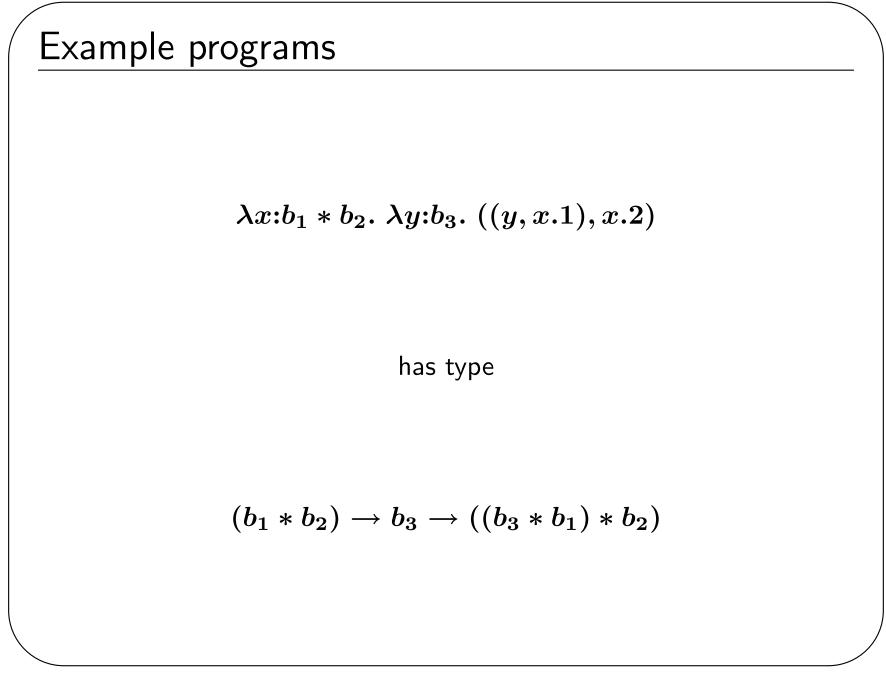
Dan Grossman





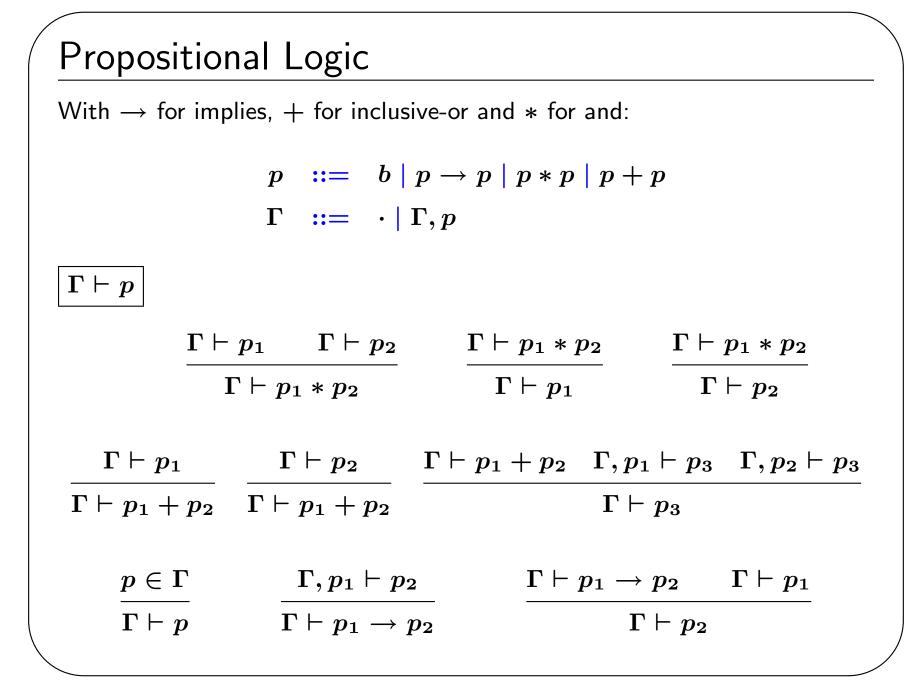






## Empty and Nonempty Types

So we have seen several "nonempty" types (closed terms of that type):  $b_{17} \rightarrow b_{17}$  $b_1 
ightarrow (b_1 
ightarrow b_2) 
ightarrow b_2$  $(b_1 
ightarrow b_2 
ightarrow b_3) 
ightarrow b_2 
ightarrow b_1 
ightarrow b_3$  $b_1 \rightarrow ((b_1 + b_7) * (b_1 + b_4))$  $(b_1 
ightarrow b_3) 
ightarrow (b_2 
ightarrow b_3) 
ightarrow (b_1 + b_2) 
ightarrow b_3$  $(b_1 * b_2) \rightarrow b_3 \rightarrow ((b_3 * b_1) * b_2)$ But there are also lots of "empty" types (no closed term of that type):  $b_1 \qquad b_1 
ightarrow b_2 \qquad b_1 + (b_1 
ightarrow b_2) \qquad b_1 
ightarrow (b_2 
ightarrow b_1) 
ightarrow b_2$ And "I" have a "secret" way of knowing whether a type will be empty; let me show you propositional logic...



CSE505 Fall 2009, Lecture 10

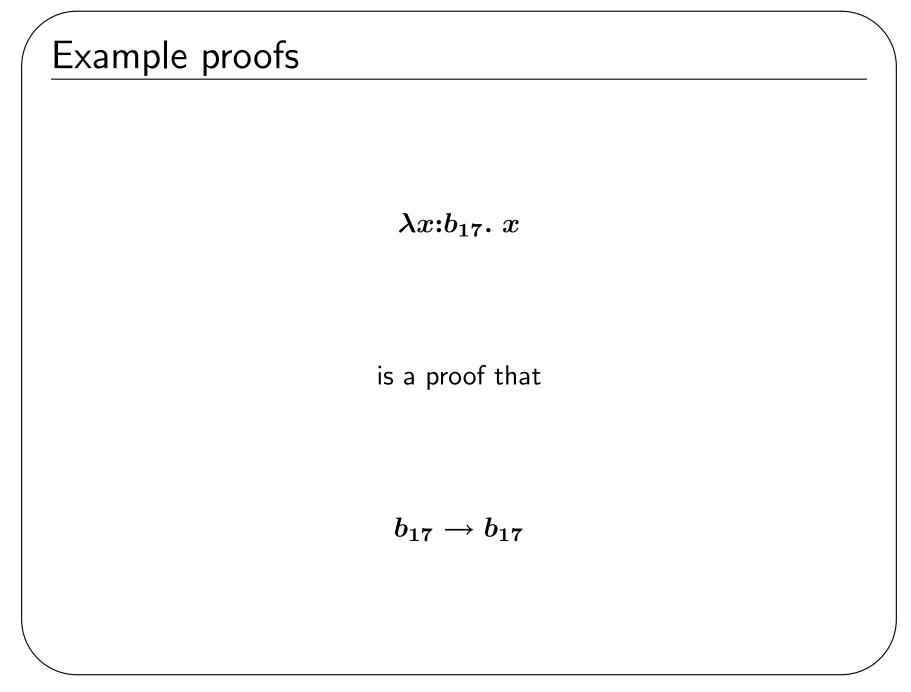
#### Guess what!!!!

That's *exactly* our type system, erasing terms and changing every au to a p $\Gamma \vdash e: au$ 

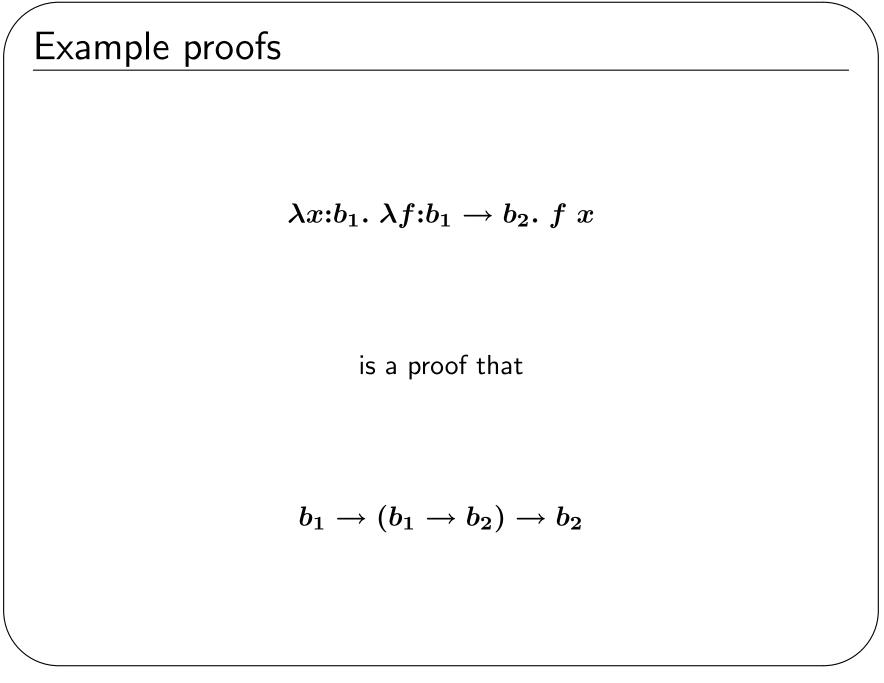
 $\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e : \tau_1 * \tau_2 \qquad \Gamma \vdash e : \tau_1 * \tau_2$  $\Gamma \vdash (e_1, e_2) : au_1 * au_2 \qquad \Gamma \vdash e.1 : au_1 \qquad \Gamma \vdash e.2 : au_2$  $\Gamma \vdash e : \tau_1$  $\Gamma \vdash e : \tau_2$  $\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2 \qquad \qquad \Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2$  $\Gamma dashermain e: au_1 + au_2 \quad \Gamma, x{:} au_1 dashermain e_1: au \quad \Gamma, y{:} au_2 dashermain e_2: au$  $\Gamma \vdash$  match e with Ax.  $e_1 \mid$  By.  $e_2 : \tau$  $\Gamma(x) = au \qquad \Gamma, x: au_1 dash e: au_2 \qquad \Gamma dash e_1: au_2 o au_1 \quad \Gamma dash e_2: au_2$  $\Gamma \vdash x: au \quad \Gamma \vdash \lambda x. \ e: au_1 
ightarrow au_2 \qquad \qquad \Gamma \vdash e_1 \ e_2: au_1$ 

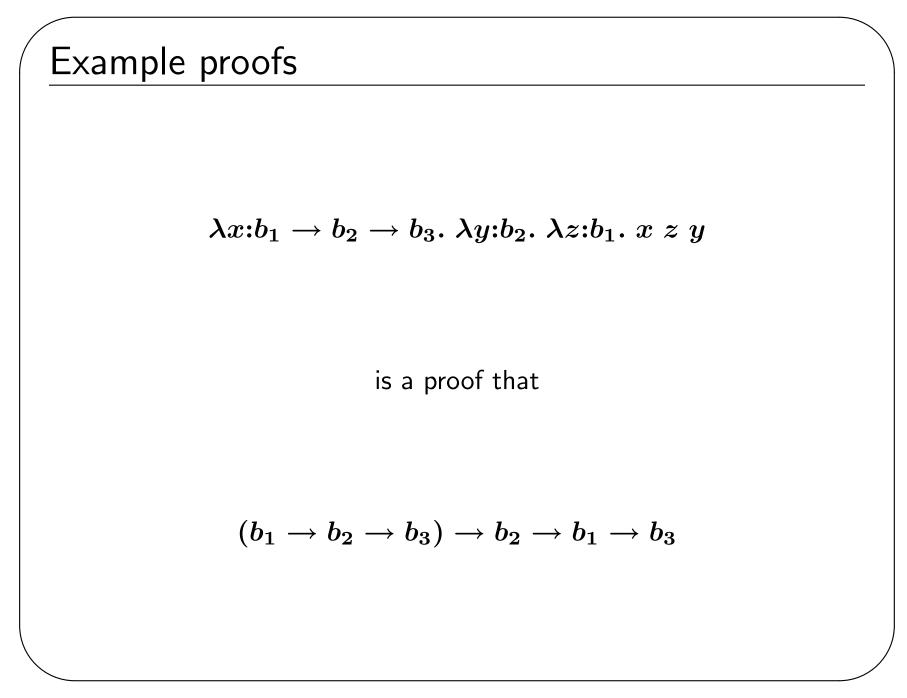
## Curry-Howard Isomorphism

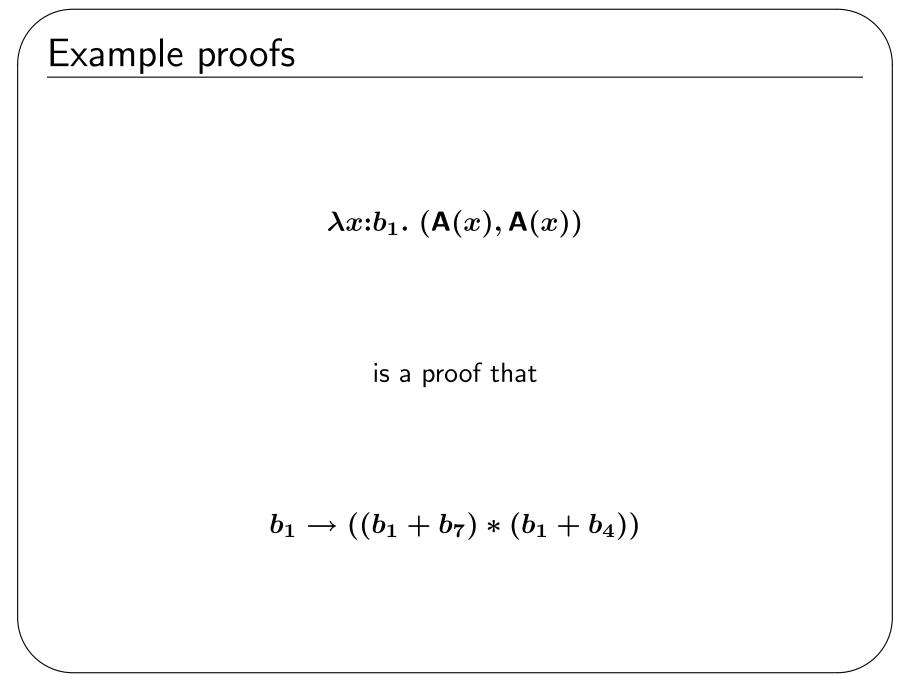
- Given a closed term that type-checks, we can take the typing derivation, erase the terms, and have a propositional-logic proof.
- Given a propositional-logic proof, there exists a closed term with that type.
- A term that type-checks is a *proof* it tells you exactly how to derive the logic formula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

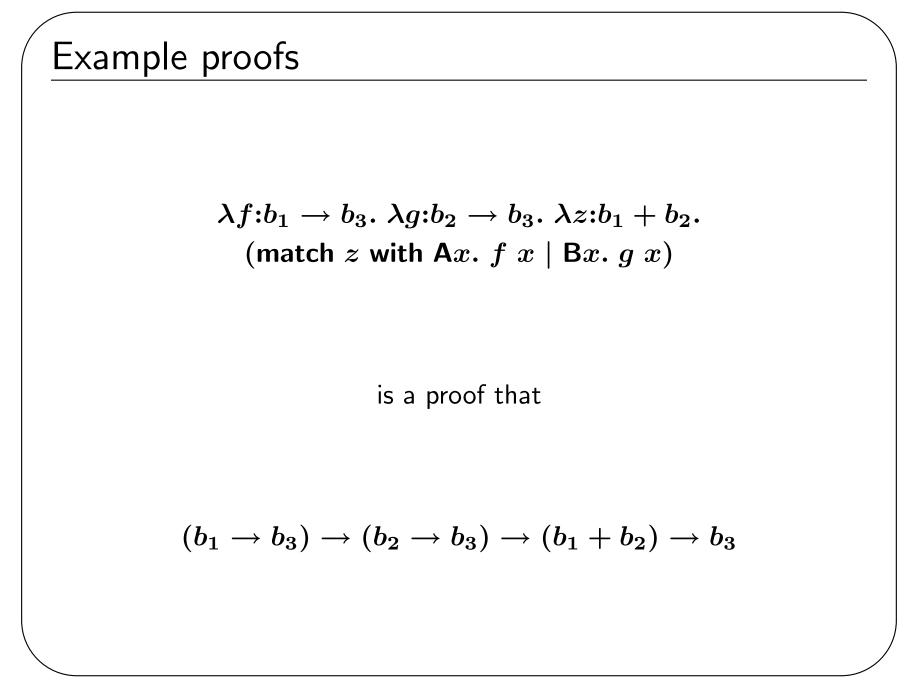


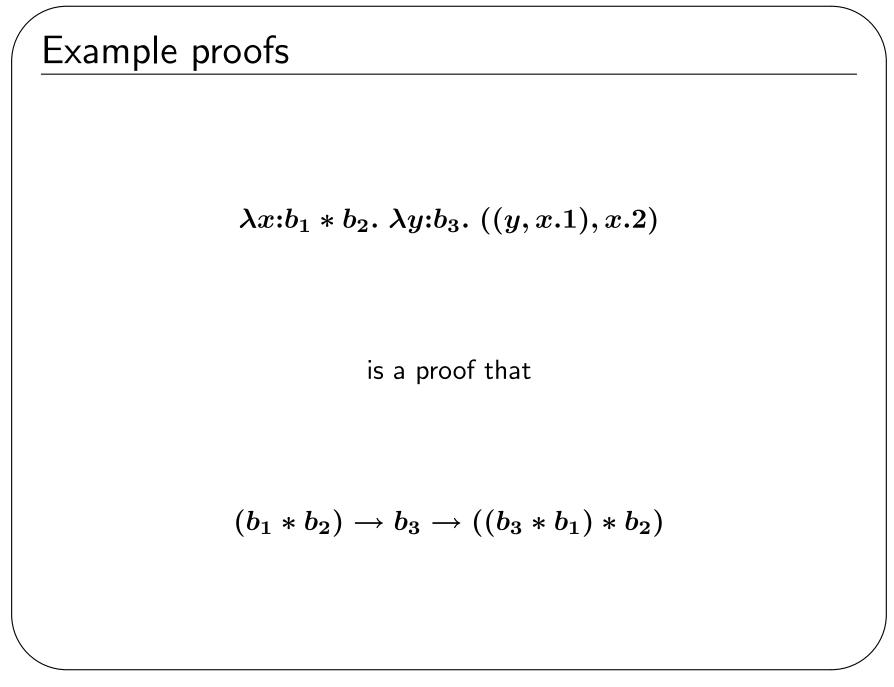
Dan Grossman











# Why care?

Because:

- This is just fascinating (glad I'm not a dog).
- For decades these were separate fields.
- Thinking "the other way" can help you know what's possible/impossible
- Can form the basis for automated theorem provers
- Type systems should not be *ad hoc* piles of rules!

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

Is ST $\lambda$ C 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":

 $\Gamma \vdash p_1 + (p_1 \rightarrow p_2)$ 

(Think "p or not p" – also equivalent to double-negation.)

 $ST\lambda C$  has *no* proof for this; there is no closed expression with this type.

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

You can still "branch on possibilities":

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

## Example classical proof

Theorem: I can always wake up at 9AM and get to campus by 10AM.

Proof: If it is a weekday, I can take a bus that leaves at 9:30AM. If it is not a weekday, traffic is light and I can drive. Since it is a weekday or not a weekday, I can get to campus by 10AM.

Problem: If you wake up and don't know if it's a weekday, this proof does not let you construct a plan to get to campus by 10AM.

In constructive logic, that never happens. You can always extract a program from a proof that "does" what you proved "could be".

You could not prove the theorem above, but you could prove, "If I know whether it is a weekday or not, then I can get to campus by 10AM."

<u>Fix</u>

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

Remember the typing rule for fix:

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

That let's us prove anything! For example: fix  $\lambda x:b_3$ . x has type  $b_3$ . 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 ST $\lambda$ C and intuitionistic propositional logic.

*Every* logic has a corresponding typed  $\lambda$  calculus (and no consistent logic has something like fix).

• Example: When we add universal types ("generics") in a few lectures, that corresponds to adding universal quantification.

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

## Toward Evaluation Contexts

(untyped)  $\lambda$ -calculus with extensions has lots of "boring inductive rules":

$$\begin{array}{c|c} \begin{array}{c} e_1 \rightarrow e_1' \\ \hline e_1 \ e_2 \rightarrow e_1' \ e_2 \end{array} & \begin{array}{c} e_2 \rightarrow e_2' \\ \hline v \ e_2 \rightarrow v \ e_2' \end{array} & \begin{array}{c} e \rightarrow e' \\ \hline e.1 \rightarrow e'.1 \end{array} & \begin{array}{c} e \rightarrow e' \\ \hline e.2 \rightarrow e'.2 \end{array} \\ \hline e_1 \rightarrow e_1' \\ \hline (e_1, e_2) \rightarrow (e_1', e_2) \end{array} & \begin{array}{c} e_2 \rightarrow e_2' \\ \hline (v_1, e_2) \rightarrow (v_1, e_2') \end{array} & \begin{array}{c} e \rightarrow e' \\ \hline A(e) \rightarrow A(e') \end{array} & \begin{array}{c} e \rightarrow e' \\ \hline B(e) \rightarrow B(e') \end{array} \\ \hline \hline B(e) \rightarrow B(e') \end{array} \\ \hline \end{array}$$

and some "interesting do-work rules":

 $\overline{(\lambda x.\ e)\ v
ightarrow e[v/x]} \qquad \overline{(v_1,v_2).1
ightarrow v_1} \qquad \overline{(v_1,v_2).2
ightarrow v_2}$ 

match A(v) with  $Ax. e_1 \mid By. e_2 \rightarrow e_1[v/x]$ 

match  $\mathsf{B}(v)$  with  $\mathsf{A}y.\ e_1 \mid \mathsf{B}x.\ e_2 \to e_2[v/x]$ 

### Evaluation Contexts

We can define *evaluation contexts*, which are expressions with one hole where "interesting work" may occur:

 $E ::= [\cdot] \mid E \mid v \mid E \mid (E,e) \mid (v,E) \mid E.1 \mid E.2$  $A(E) \mid B(E) \mid (match \ E \text{ with } Ax. \ e_1 \mid By. \ e_2)$ Define "filling the hole" E[e] in the obvious way (stapling). Semantics now uses two judgments e 
ightarrow e' and  $e \stackrel{\mathrm{p}}{
ightarrow} e'$ , but the former has only 1 rule and the latter has just the "interesting work":  $e \xrightarrow{p} e'$  $\overline{E[e] 
ightarrow E[e']}$  $\overline{(\lambda x.\ e)\ v \xrightarrow{\mathrm{p}} e[v/x]} \qquad \overline{(v_1, v_2).1 \xrightarrow{\mathrm{p}} v_1} \qquad \overline{(v_1, v_2).2 \xrightarrow{\mathrm{p}} v_2}$ match A(v) with  $Ax. e_1 \mid By. e_2 \xrightarrow{p} e_1[v/x]$ match B(v) with Ay.  $e_1 \mid Bx. \ e_2 \xrightarrow{p} e_2[v/x]$ 

## So what?

So far, all we have done is rearrange our semantics to be more concise

• Each boring rule become a form of  ${m E}$ 

Evaluation relies on *decomposition* (unstapling the right subtree): Given e, find an E,  $e_a$ ,  $e'_a$  such that  $e = E[e_a]$  and  $e_a \xrightarrow{p} e'_a$ .

Theorem (Unique Decomposition): If  $\cdot \vdash e : \tau$ , then e is a value or there is exactly one decomposition of e.

- Hence evaluation is deterministic
- In fact it's still CBV left-to-right

But the real power from defining E is that it lets us *reify* continuations (evaluation stacks) ...

## **Continuations**

First-class continuations in one slide:

- $e ::= \ldots | \text{letcc } x. e | \text{ throw } e e | \text{ cont } E$
- $v ::= \dots | \operatorname{cont} E$
- $E ::= \ldots$  | throw E e | throw v E

$$E[\operatorname{\mathsf{letcc}} x. e] \to E[(\lambda x. e)(\operatorname{\mathsf{cont}} E)]$$

 $E[{
m throw}~({
m cont}~E')~v] 
ightarrow E'[v]$ 

*Very* powerful and general: For example, non-preemptive multithreading *in the language*. Exceptions. "Time travel."

### Connection to interpreters

A "real" (efficient, natural) interpreter for lambda-calculus (or ML) would not be like our small-step semantics

• Would re-decompose the whole program for each step!

Instead, maintain the decomposition incrementally

• With a stack to remember "what to work on next"!

Also, don't use substitution; use environments (see your homework)

• At this point, need just one while-loop, pairs, and malloc

And if your stacks are heap-allocated and immutable, you can implement continuation operations (letcc and throw) in O(1) time.

- A nice (and provably correct) sequence of more primitive interpreters
- Can post Caml code for the curious