

Dependent Type Theory

Charles Averill

The University of Texas at Dallas

October 22nd, 2026



Where System $F\omega$ Fails

- System $F\omega$ is pretty good, forms the basis of OCaml, Haskell, Rust, C++, Swift type systems
- Point of type systems is “well-typed programs can't ‘go wrong’ ”
- But...they still can!

```
let hd (l : 'a list) : 'a =
  match l with
  | [] -> failwith "can't take hd of []"
  | h :: _ -> h
```

Where System $F\omega$ Fails

- System $F\omega$ is pretty good, forms the basis of OCaml, Haskell, Rust, C++, Swift type systems
- Point of type systems is “well-typed programs can’t ‘go wrong’ ”
- But...they still can!

```
let hd (l : 'a list) : 'a =
  match l with
  | [] -> failwith "can't take hd of []"
  | h :: _ -> h
```

Where System $F\omega$ Fails

- System $F\omega$ is pretty good, forms the basis of OCaml, Haskell, Rust, C++, Swift type systems
- Point of type systems is “well-typed programs can’t ‘go wrong’ ”
- But... they still can!

```
let hd (l : 'a list) : 'a =
  match l with
  | [] -> failwith "can't take hd of []"
  | h :: _ -> h
```

Where System F_ω Fails

What if we could prevent such errors at compile time by extending our type system?

```
let hd (l : 'a (n + 1) list) : 'a =  
  match l with  
  | h :: _ -> h
```

The type of `hd`'s input argument *depends* on the value `n + 1` to ensure that the list is non-empty.

Lambda Calculus Refresher

Syntax

$$e := v \mid \lambda v. e \mid e_1 e_2$$

Semantics

$$\overline{(\lambda v. e_1) e_2 \Downarrow e_1[e_2/v]}^{(\rightarrow_\beta)}$$

$$\frac{P \quad P \Longrightarrow Q}{Q}$$

$$(\lambda x. x)(\lambda y. y) \rightarrow_\beta \lambda y. y$$

Lambda Calculus Refresher

Syntax

$$e := v \mid \lambda v. e \mid e_1 e_2$$

Semantics

$$\overline{(\lambda v. e_1) e_2 \Downarrow e_1[e_2/v]}^{(\rightarrow_\beta)}$$

$$\frac{P \quad P \Longrightarrow Q}{Q}$$

$$\begin{aligned} & (\lambda x. x)(\lambda y. y) \rightarrow_\beta \lambda y. y \\ & (\lambda x. \lambda y. x)(\lambda a. a)(\lambda b. b) \rightarrow_\beta (\lambda y. (\lambda a. a))(\lambda b. b) \rightarrow_\beta \lambda a. a \end{aligned}$$

Lambda Calculus Refresher

Syntax

$$e := v \mid \lambda v. e \mid e_1 e_2$$

Semantics

$$\overline{(\lambda v. e_1) e_2 \Downarrow e_1[e_2/v]}^{(\rightarrow_\beta)}$$

$$\frac{P \quad P \implies Q}{Q}$$

$$(\lambda x. x)(\lambda y. y) \rightarrow_\beta \lambda y. y$$

$$(\lambda x. \lambda y. x)(\lambda a. a)(\lambda b. b) \rightarrow_\beta (\lambda y. (\lambda a. a))(\lambda b. b) \rightarrow_\beta \lambda a. a$$

$$(\lambda f. (\lambda x. f \ x \ x)(\lambda x. f \ x \ x))(\lambda f. (\lambda x. f \ x \ x)(\lambda x. f \ x \ x)) \rightarrow_\beta ?$$

Lambda Calculus Refresher

Syntax

$$e := v \mid \lambda v. e \mid e_1 e_2$$

Semantics

$$\overline{(\lambda v. e_1) e_2 \Downarrow e_1[e_2/v]}^{(\rightarrow_\beta)}$$

$$\frac{P \quad P \implies Q}{Q}$$

$$(\lambda x. x)(\lambda y. y) \rightarrow_\beta \lambda y. y$$

$$(\lambda x. \lambda y. x)(\lambda a. a)(\lambda b. b) \rightarrow_\beta (\lambda y. (\lambda a. a))(\lambda b. b) \rightarrow_\beta \lambda a. a$$

$$(\lambda f. (\lambda x. f \ x \ x)(\lambda x. f \ x \ x))(\lambda f. (\lambda x. f \ x \ x)(\lambda x. f \ x \ x)) \rightarrow_\beta ?$$

$$\text{succ } [1, 2, 3] \rightarrow_\beta ?$$

System $F\omega$ Refresher

Syntax

$$e ::= () \mid v \mid \lambda v : T. e \mid e_1 e_2$$

$$T ::= \text{unit} \mid \text{void} \mid T_1 \rightarrow T_2$$

Typing Semantics

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma. e) : (\sigma \rightarrow \tau)} \quad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \quad (\lambda_{\rightarrow})$$

Γ : *typing context*, function that maps ids to types

$\Gamma \vdash e : \tau$: “in context Γ , e has type τ ”

$$\lambda x : \text{int}. \lambda y : \text{int}. x + y$$

$$(\lambda s : \text{str}. \text{"Hello, " + } s + \text{"!"}) \text{"World"}$$

System F_ω Refresher

Syntax

$$e := () \mid v \mid \lambda v : T. e \mid e_1 e_2 \mid \Lambda \alpha. e$$

$$T := \mathbf{unit} \mid \mathbf{void} \mid T_1 \rightarrow T_2 \mid \alpha \mid \forall \alpha. T$$

Typing Semantics

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma. e) : (\sigma \rightarrow \tau)} \quad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \quad (\lambda_{\rightarrow})$$

$$\frac{\Gamma, \alpha \vdash e : \sigma}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \sigma} \quad \frac{\Gamma \vdash e : \forall \alpha. \sigma}{\Gamma \vdash e [T] : \sigma [T/\alpha]} \quad (\lambda_2)$$

$$\mathbf{CONS} : \forall \alpha. \alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list} := \Lambda \alpha. \lambda hd : \alpha. \lambda tl : \alpha \text{ list}. (hd, tl)$$

System F_ω Refresher

Syntax

$$e ::= () \mid v \mid \lambda v : T. e \mid e_1 e_2 \mid \Lambda \alpha :: K. e \mid e [T]$$

$$T ::= \text{unit} \mid \text{void} \mid T_1 \rightarrow T_2 \mid \alpha \mid \forall \alpha :: K. T \mid \lambda \alpha :: K. T \mid T_1 T_2$$

$$K ::= * \mid K_1 \rightarrow K_2$$

Typing Semantics (assuming type equivalence via β -reduction)

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma. e) : (\sigma \rightarrow \tau)} \quad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \quad (\lambda_{\rightarrow})$$

$$\frac{\Gamma, \alpha \vdash e : \sigma}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \sigma} \quad \frac{\Gamma \vdash e : \forall \alpha. \sigma \quad \Gamma \vdash T :: \kappa}{\Gamma \vdash e [T] : \sigma [T/\alpha]} \quad (\lambda_2) \quad \frac{\alpha :: \kappa \in \Gamma}{\Gamma \vdash \alpha :: \kappa} \quad \frac{t \in \lambda_2}{\Gamma \vdash t :: *}$$

$$\frac{\Gamma, x :: \kappa_1 \vdash t :: \kappa_2}{\Gamma \vdash (\lambda x :: \kappa_1. t) :: (\kappa_1 \rightarrow \kappa_2)} \quad \frac{\Gamma \vdash t_1 :: \kappa_1 \rightarrow \kappa_2 \quad \Gamma \vdash t_2 :: \kappa_1}{\Gamma \vdash t_1 t_2 :: \kappa_2} \quad (\lambda_{\omega})$$

System F_ω Refresher

PAIR:: $* \rightarrow * \rightarrow * :=$

$\lambda A :: *. \lambda B :: *. \forall F. (A \rightarrow B \rightarrow F) \rightarrow F$

tuple: $\forall A. \forall B. A \rightarrow B \rightarrow \text{PAIR } A B :=$

$\lambda A. \lambda B. \lambda fst : A. \lambda snd : B. \lambda F. \lambda f : (A \rightarrow B \rightarrow F). f \text{ fst } snd$

SUM:: $* \rightarrow * \rightarrow * :=$

$\lambda A :: *. \lambda B :: *. \forall F. (A \rightarrow F) \rightarrow (B \rightarrow F) \rightarrow F$

left: $\forall A. \forall B. A \rightarrow \text{SUM } A B :=$

$\lambda A. \lambda B. \lambda a : A. \lambda F. \lambda f_1 : (A \rightarrow F). \lambda f_2 : (B \rightarrow F). f_1 \ a$

right: $\forall A. \forall B. B \rightarrow \text{SUM } A B :=$

$\lambda A. \lambda B. \lambda b : B. \lambda F. \lambda f_1 : (A \rightarrow F). \lambda f_2 : (B \rightarrow F). f_2 \ b$

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	
<code>void</code>	
<code>unit → unit</code>	
<code>unit → void</code>	
<code>void → unit</code>	
<code>void → void</code>	
$\forall \alpha :: *. \alpha \rightarrow \alpha$	
$\forall \alpha :: *. \alpha$	
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	
<code>void → unit</code>	
<code>void → void</code>	
$\forall \alpha :: *. \alpha \rightarrow \alpha$	
$\forall \alpha :: *. \alpha$	
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	
<code>void → void</code>	
$\forall \alpha :: *. \alpha \rightarrow \alpha$	
$\forall \alpha :: *. \alpha$	
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	
$\forall \alpha :: *. \alpha \rightarrow \alpha$	
$\forall \alpha :: *. \alpha$	
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	<code>λx : void. x</code>
$\forall \alpha :: *. \alpha \rightarrow \alpha$	
$\forall \alpha :: *. \alpha$	
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	<code>λx : void. x</code>
<code>∀α :: *. α → α</code>	<code>Λα :: *. λx : α. x</code>
<code>∀α :: *. α</code>	
<code>∀α :: *. ∀β :: *. α → β</code>	
<code>(λβ :: *. β → β) unit</code>	
<code>∀F :: * → *. ∀α :: *. F α → F α</code>	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	<code>λx : void. x</code>
$\forall \alpha :: *. \alpha \rightarrow \alpha$	$\Lambda \alpha :: *. \lambda x : \alpha. x$
$\forall \alpha :: *. \alpha$	<code>×</code>
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	<code>λx : void. x</code>
<code>∀α :: *. α → α</code>	<code>Λα :: *. λx : α. x</code>
<code>∀α :: *. α</code>	<code>×</code>
<code>∀α :: *. ∀β :: *. α → β</code>	<code>×</code>
<code>(λβ :: *. β → β) unit</code>	<code>×</code>
<code>∀F :: * → *. ∀α :: *. F α → F α</code>	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	<code>λx : void. x</code>
$\forall \alpha :: *. \alpha \rightarrow \alpha$	$\Lambda \alpha :: *. \lambda x : \alpha. x$
$\forall \alpha :: *. \alpha$	<code>×</code>
$\forall \alpha :: *. \forall \beta :: *. \alpha \rightarrow \beta$	<code>×</code>
$(\lambda \beta :: *. \beta \rightarrow \beta) \text{ unit}$	<code>λx : unit. x</code>
$\forall F :: * \rightarrow *. \forall \alpha :: *. F \alpha \rightarrow F \alpha$	

Type Inhabitation

A type is *inhabited* iff there exists a term with that type.

Type	Inhabited?
<code>unit</code>	<code>()</code>
<code>void</code>	<code>×</code>
<code>unit → unit</code>	<code>λx : unit. x</code>
<code>unit → void</code>	<code>×</code>
<code>void → unit</code>	<code>λx : void. ()</code>
<code>void → void</code>	<code>λx : void. x</code>
<code>∀α :: *. α → α</code>	<code>Λα :: *. λx : α. x</code>
<code>∀α :: *. α</code>	<code>×</code>
<code>∀α :: *. ∀β :: *. α → β</code>	<code>×</code>
<code>(λβ :: *. β → β) unit</code>	<code>λx : unit. x</code>
<code>∀F :: * → *. ∀α :: *. F α → F α</code>	<code>ΛF :: * → *. Λα :: *. λx : F α. x</code>

Type Inhabitation

We can fill out this relation with some added standard types, like tuples:

Type	Inhabited
<code>unit</code>	True
<code>void</code>	False

Type Inhabitation

We can fill out this relation with some added standard types, like tuples:

Type	Inhabited
<code>unit</code>	True
<code>void</code>	False
<code>void × void</code>	False
<code>void × unit</code>	False
<code>unit × void</code>	False
<code>unit × unit</code>	True

Type Inhabitation

We can fill out this relation with some added standard types, like tuples:

Type	Inhabited
<code>unit</code>	True
<code>void</code>	False
<code>void × void</code>	False
<code>void × unit</code>	False
<code>unit × void</code>	False
<code>unit × unit</code>	True
<code>void → void</code>	True
<code>void → unit</code>	True
<code>unit → void</code>	False
<code>unit → unit</code>	True

Type Inhabitation

We can fill out this relation with some added standard types, like tuples:

Type	Inhabited
<code>unit</code>	True
<code>void</code>	False
<code>void × void</code>	False
<code>void × unit</code>	False
<code>unit × void</code>	False
<code>unit × unit</code>	True
<code>void → void</code>	True
<code>void → unit</code>	True
<code>unit → void</code>	False
<code>unit → unit</code>	True
$\forall \alpha :: *. \alpha$	False
$\forall \alpha :: *. \alpha \rightarrow \alpha$	True

Type Inhabitation

We can fill out this relation with some added standard types, like tuples:

Logical Expression	Truth Value
T	True
F	False
$F \wedge F$	False
$F \wedge T$	False
$T \wedge F$	False
$T \wedge T$	True
$F \rightarrow F$	True
$F \rightarrow T$	True
$T \rightarrow F$	False
$T \rightarrow T$	True
$\forall P. P$	False
$\forall P. P \rightarrow P$	True

Break! Any Questions?

Types as Propositions

It turns out that System $F\omega$ corresponds with Higher-Order Propositional Logic (HOL)! $F\omega$ types are propositions, and its values are proofs of propositions.

```
let modus_ponens : P -> (P -> Q) -> Q =
  fun (p : P) (f : P -> Q) ->
    f p
```

$$\frac{P \quad P \implies Q}{Q}(\text{MP})$$

What do we get with higher-order propositional logic?

- Propositions that talk about other propositions (reasoning about logical connectives):

$$\text{SYMMETRIC}(F) := \forall P Q, F P Q \implies F Q P$$

- Encodings of impredicative relations and data structures:

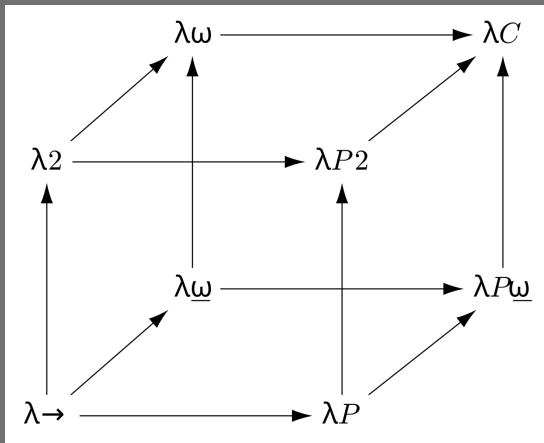
$$\text{EQ}(P, Q) := \forall R, R P \implies R Q$$

Where it Stops

- HOL is pretty nice (see Isabelle/HOL), but has limitations
- Can't even define predicates over values, like `5 is odd`
- HOL propositions can only take *other propositions* as parameters
- Not even induction is native to HOL!

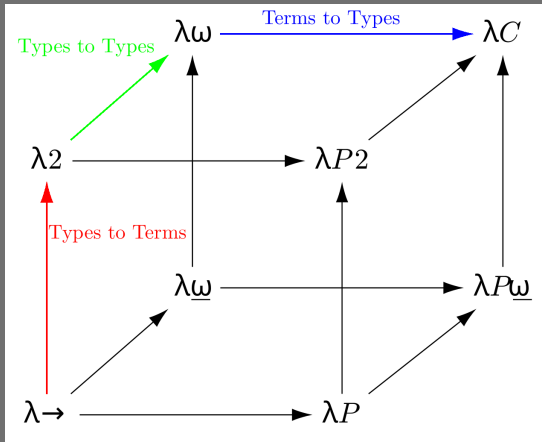
Back to PL

The problem is related to the lambda calculi typing semantics!



Back to PL

The problem is related to the lambda calculi typing semantics!



λC

Syntax

$$e := \lambda_\omega$$

$$T := \lambda_\omega \mid \Pi v : T_1. T_2$$

$$K := \lambda_\omega \mid \Pi v : T. K$$

Typing Semantics (plus λ_ω)

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash T_2 :: *}{\Gamma \vdash \Pi x : T_1. T_2 :: *} \quad \frac{\Gamma \vdash S :: \Pi x : T. K \quad \Gamma \vdash t : T}{\Gamma \vdash S t :: K[t/x]}$$

$$\frac{\Gamma \vdash S :: * \quad \Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S. t) : \Pi x : S. T} \quad \frac{\Gamma \vdash t_1 : \Pi x : S. T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 : T[t_2/x]}$$

$$\frac{\Gamma \vdash T :: * \quad \Gamma, x : T \vdash S :: K}{\Gamma \vdash (\lambda x : T. S) :: \Pi x : T. K} \quad \frac{\Gamma \vdash T :: * \quad \Gamma, x : T \vdash K}{\Gamma \vdash \Pi x : T. K} (\lambda C)$$

Dependent Vectors

```

VEC :: * → Π n : nat. * :=
  λ T :: *. λ n : nat. n = 0 ? unit : PAIR T (VEC T (n - 1))
nil : ∀ T :: *. VEC T 0 := λ T :: *. ()
cons : ∀ T :: *. Π n : nat. T → VEC T n → VEC T (n + 1) :=
  λ T :: *. λ n : nat. λ hd : T. λ tl : VEC T n. tuple [T] [VEC T n] hd tl
hd : ∀ T :: *. Π n : nat. VEC T (n + 1) → T :=
  fst [T] [VEC T n]
tl : ∀ T :: *. Π n : nat. VEC T (n + 1) → VEC T n :=
  snd [T] [VEC T n]

```