Dependent Types

Stephen A. Edwards

Columbia University

Spring 2023

Andres Löh, Conor McBride and Wouter Swierstra
A Tutorial Implementation of a Dependently Typed Lambda Calculus
Fundamenta Informaticae, 102(2):117–207, April 2010
`https://www.andres-loeh.de/LambdaPi/`

Stephanie Weirich
Implementing Dependent Types in `pi-forall`
`https://github.com/sweirich/pi-forall`
See doc/oplss.pdf. August 15, 2022

Lambda (Barendregt) Cube

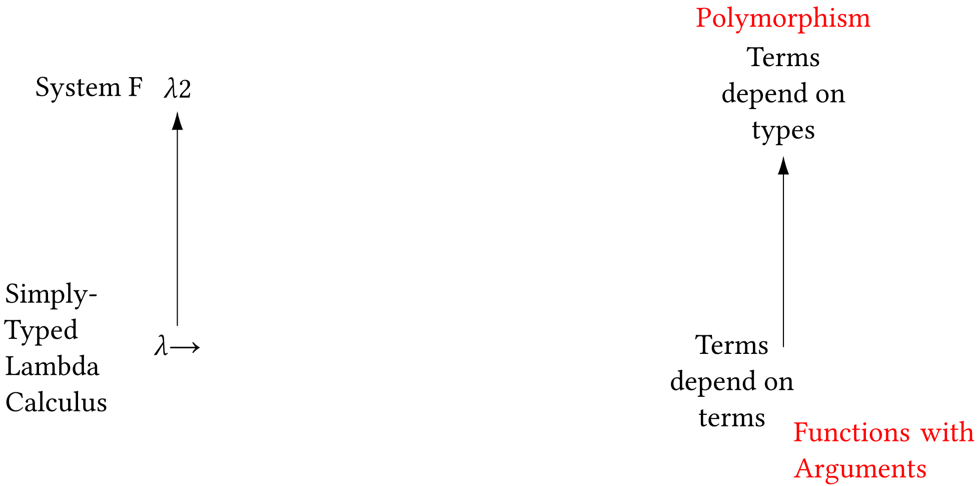
Simply-Typed Lambda Calculus

$\lambda{\rightarrow}$

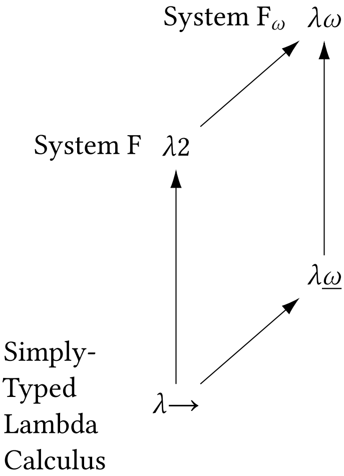
Terms depend on terms

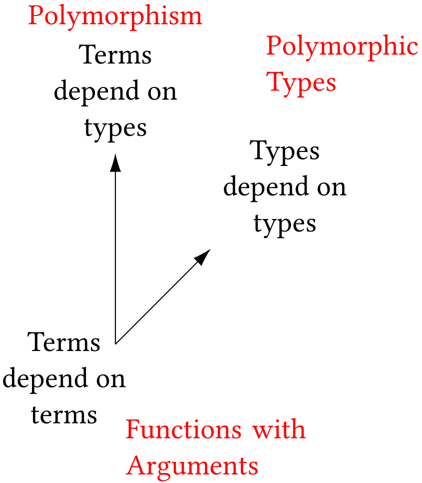
Functions with Arguments

Lambda (Barendregt) Cube

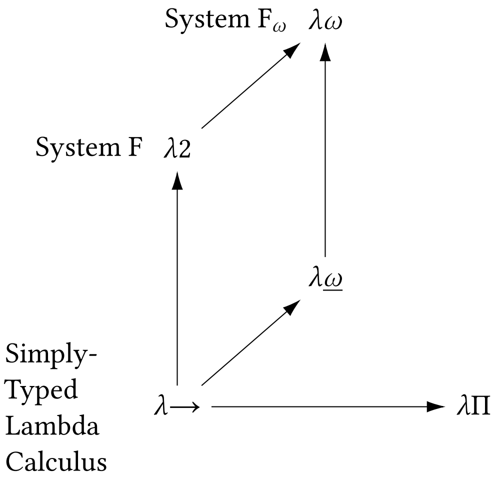


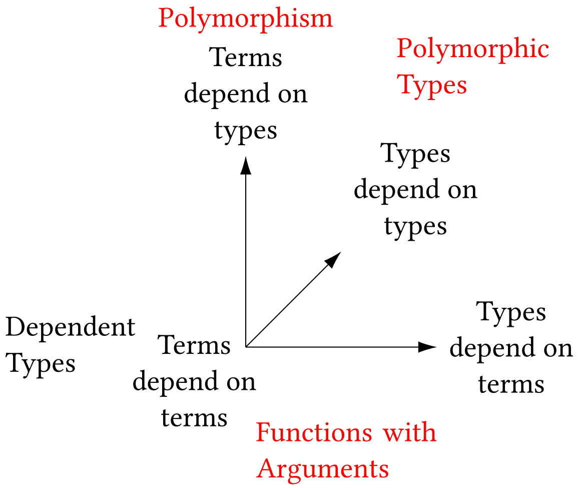
Lambda (Barendregt) Cube



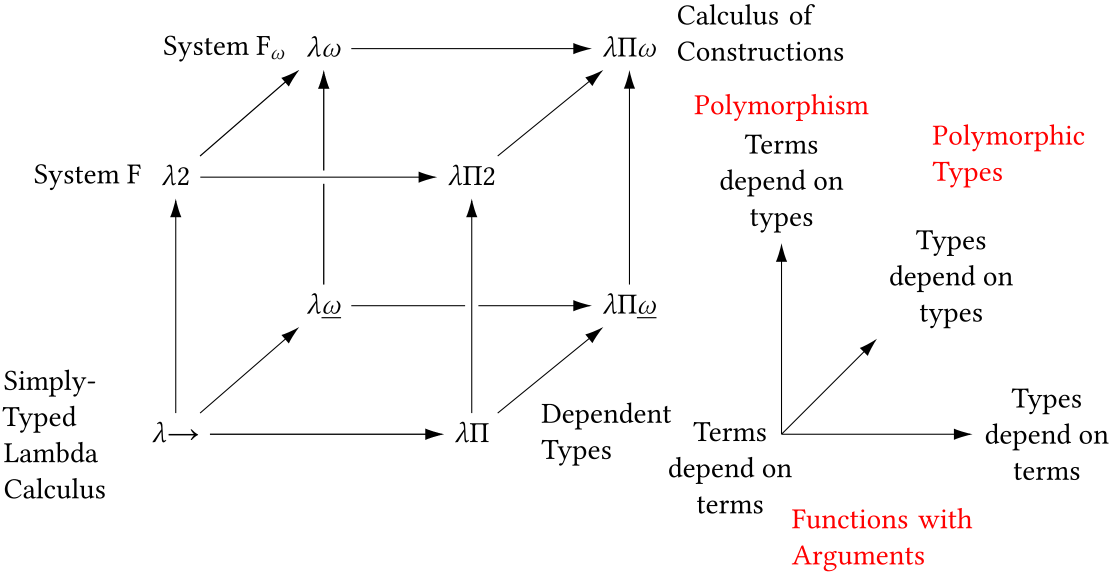


Lambda (Barendregt) Cube





Lambda (Barendregt) Cube



System	Term	Type
$\lambda\rightarrow$	$\lambda x : \texttt{bool} \,.\, x$	$\texttt{bool} \rightarrow \texttt{bool}$

Base types and functions from type to type

Binder arguments annotated with a type

No polymorphism: $\lambda x \,.\, x$ must have a specific type

System	Term	Type
$\lambda\!\rightarrow$	$\lambda x : \texttt{bool} \,.\, x$	$\texttt{bool} \rightarrow \texttt{bool}$
System F	$\Lambda\alpha \,.\, \lambda x : \alpha \,.\, \alpha$	$\forall\alpha \,.\, \alpha \longrightarrow \alpha$

Polymorphic functions: (terms may depend on types)

Type variables, $\forall$ types

A term Λ binds a type argument to a type variable α

Type variables are untyped

System	Term	Type
$\lambda\!\rightarrow$	$\lambda x : \mathtt{bool} \, . \, x$	$\mathtt{bool} \longrightarrow \mathtt{bool}$
System F	$\Lambda\alpha \, . \, \lambda x : \alpha \, . \, \alpha$	$\forall\alpha \, . \, \alpha \longrightarrow \alpha$
System F_ω	$\Lambda\alpha : \star \, . \, \lambda h : \alpha \, . \, \lambda t : \mathtt{List}\,\alpha \, . \, \ldots$	$\forall\alpha : \star \, . \, \alpha \longrightarrow \mathtt{List}\,\alpha \longrightarrow \mathtt{List}\,\alpha$

Polymorphic types: (types may depend on types)

The type of a type is a kind. $\star$ is a simple type, $\star \longrightarrow \star$ is a type constructor like $\mathtt{List}$.

This is the "cons" function for the polymorphic list type $\mathtt{List}\,\alpha$

It takes a simple type α, an object of type α, and a list of α's, and produces a list of α's

System	Term	Type
$\lambda\!\rightarrow$	$\lambda x : \texttt{bool} \,.\, x$	$\texttt{bool} \longrightarrow \texttt{bool}$
System F	$\Lambda\alpha \,.\, \lambda x : \alpha \,.\, \alpha$	$\forall\alpha \,.\, \alpha \longrightarrow \alpha$
System F_ω	$\Lambda\alpha : * \,.\, \lambda h : \alpha \,.\, \lambda t : \texttt{List}\,\alpha \,.\, \ldots$	$\forall\alpha : * \,.\, \alpha \longrightarrow \texttt{List}\,\alpha \longrightarrow \texttt{List}\,\alpha$
$\lambda\Pi$	Identity on n-element vectors	$(\alpha : *) \longrightarrow (n : \texttt{Nat}) \longrightarrow (v : \texttt{Vec}\,\alpha\,n) \longrightarrow \texttt{Vec}\,\alpha\,n$

Types may depend on ordinary terms such as natural numbers

You can define a polymorphic **Vec** type parameterized by a type α and a natural number length n

You can define an identity function on such vectors

Aside: Types as Terms: Tuples as Type Lists in Haskell (System F_ω)

```
$ ghci
> :set -XTypeOperators                                   -- Infix type constructor syntax
> data Nil          = Nil          deriving Show  -- Empty list
> data tail :. head = tail :. head deriving Show  -- List of types (wrong associativity)
> x = Nil :. (5::Int) :. 'a'      -- :. as data constructor
> :t x
x :: (Nil :. Int) :. Char         -- :. as type constructor: Char, Int "pairs"
```

Aside: Types as Terms: Tuples as Type Lists in Haskell (System F_ω)

```
$ ghci
> :set -XTypeOperators                                       -- Infix type constructor syntax
> data Nil          = Nil           deriving Show  -- Empty list
> data tail :. head = tail :. head deriving Show  -- List of types (wrong associativity)
> x = Nil :. (5::Int) :. 'a'       -- :. as data constructor
> :t x
x :: (Nil :. Int) :. Char          -- :. as type constructor: Char, Int "pairs"
> myhd (t :. h) = h                -- Head of list
> :t myhd
myhd :: (tail :. head) -> head     -- myhd on a "Cons" type returns the type of its head
> mytl (t :. h) = t                -- Tail of list
> :t mytl
mytl :: (tail :. head) -> tail     -- mytl on a "Cons" type returns the type of its tail
```

Aside: Types as Terms: Tuples as Type Lists in Haskell (System F_ω)

```
$ ghci
> :set -XTypeOperators                                -- Infix type constructor syntax
> data Nil          = Nil          deriving Show  -- Empty list
> data tail :. head = tail :. head deriving Show  -- List of types (wrong associativity)
> x = Nil :. (5::Int) :. 'a'       -- :. as data constructor
> :t x
x :: (Nil :. Int) :. Char          -- :. as type constructor: Char, Int "pairs"
> myhd (t :. h) = h                -- Head of list
> :t myhd
myhd :: (tail :. head) -> head     -- myhd on a "Cons" type returns the type of its head
> mytl (t :. h) = t                -- Tail of list
> :t mytl
mytl :: (tail :. head) -> tail     -- mytl on a "Cons" type returns the type of its tail
> myhd x
'a'
> mytl x
Nil :. 5
> :t mytl x
mytl x :: Nil :. Int               -- Type of Int singletons
> myhd (mytl x)
5                                  -- WORKS, BUT CAN'T USE LIST MACHINERY
```

$\lambda\!\longrightarrow$ Types and Expressions

$\tau ::= \alpha$	base type
$\quad\;\; \tau \longrightarrow \tau$	function type
$e ::= e : \tau$	annotated term
$\quad\;\; x$	variable
$\quad\;\; e\; e$	application
$\quad\;\; \lambda x\, .\, e$	lambda abstraction

$\lambda\!\longrightarrow$ Values

$v ::= n$	neutral term
$\quad\;\; \lambda x\, .\, v$	lambda abstraction
$n ::= x$	variable
$\quad\;\; n\; v$	application

Another Simply Typed Lambda Calculus

Base types α are, e.g., **`Bool`**, **`Nat`**

Type annotations on expressions instead of variables in λ terms. Not a significant change

Values (normal forms of evaluation): e.g., x, $\lambda x\, .\, x$, $x\; y\, (\lambda z\, .\, z)$. No redexes allowed

Löh et al. write $e :: \tau$ for $e : \tau$ and $\lambda x \longrightarrow e$ for $\lambda x\, .\, e$

$\lambda\!\longrightarrow$ Types and Expressions

$\tau ::= \alpha$	base type
$\quad\quad \tau \longrightarrow \tau$	function type
$e ::= e : \tau$	annotated term
$\quad\quad x$	variable
$\quad\quad e\, e$	application
$\quad\quad \lambda x \,.\, e$	lambda abstraction

$\lambda\!\longrightarrow$ Values

$v ::= n$	neutral term
$\quad\quad \lambda x \,.\, v$	lambda abstraction
$n ::= x$	variable
$\quad\quad n\, v$	application

$\lambda\!\longrightarrow$ Big-Step Evaluation Rules

$$\frac{e \Downarrow v}{e : \tau \Downarrow v} \qquad \frac{}{x \Downarrow x} \qquad \frac{e \Downarrow \lambda x \,.\, v \quad v[x := e'] \Downarrow v'}{e\, e' \Downarrow v'} \qquad \frac{e \Downarrow n \quad e' \Downarrow v'}{e\, e' \Downarrow n v'} \qquad \frac{e \Downarrow v}{\lambda x \,.\, e \Downarrow \lambda x \,.\, v}$$

Strongly normalizing; normal form always exists; reduce to a value in a single step

"Ignore type annotations" "Stop at variables" "Apply a λ value by substituting"

"Applying a neutral term, just reduce argument" "Reduce the body of a λ"

$$\Big((\lambda x \,.\, x) : (\alpha \longrightarrow \alpha)\Big)\, y \Downarrow y \quad \Big((\lambda x \,.\, \lambda y \,.\, x) : (\beta \longrightarrow \beta) \longrightarrow \alpha \longrightarrow \beta \longrightarrow \beta\Big)\, (\lambda z \,.\, z)\, y \Downarrow (\lambda z \,.\, z)$$

$\lambda\!\!\longrightarrow$ Types and Expressions

$\tau ::= \alpha$		base type
	$\tau \longrightarrow \tau$	function type
$e ::= e : \tau$		annotated term
	x	variable
	$e\, e$	application
	$\lambda x\,.\,e$	lambda abstraction

$\lambda\!\!\longrightarrow$ Values

$v ::= n$		neutral term
	$\lambda x\,.\,v$	lambda abstraction
$n ::= x$		variable
	$n\, v$	application

Contexts

$\Gamma ::= \epsilon$		empty context
	$\Gamma, \alpha : *$	adding a type identifier
	$\Gamma, x : \tau$	adding a term identifier

"$*$" means "base type"; vacuous for now

Valid Contexts

$$\frac{}{\mathrm{valid}(\epsilon)} \qquad \frac{\mathrm{valid}(\Gamma)}{\mathrm{valid}(\Gamma, \alpha : *)} \qquad \frac{\mathrm{valid}(\Gamma) \qquad \Gamma \vdash \tau : *}{\mathrm{valid}(\Gamma, x : \tau)}$$

"The empty context is valid"
"A base type may be in context"
"An identifier in context has a base type"

$\lambda\!\to$ Types and Expressions

$\tau ::= \alpha$		base type
	$\tau \to \tau$	function type
$e ::= e : \tau$		annotated term
	x	variable
	$e\, e$	application
	$\lambda x \,.\, e$	lambda abstraction

$\lambda\!\to$ Values

$v ::= n$		neutral term
	$\lambda x \,.\, v$	lambda abstraction
$n ::= x$		variable
	$n\, v$	application

$\lambda\!\to$ Type Rules in Bidirectional Style

$$\frac{\Gamma(\alpha) = *}{\Gamma \vdash \alpha : *}\ \text{var1} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \tau' : *}{\Gamma \vdash \tau \to \tau' : *}\ \text{fun} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_{\uparrow} \tau}\ \text{var2} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash e :_{\downarrow} \tau}{\Gamma \vdash (e : \tau) :_{\uparrow} \tau}\ \text{ann}$$

$$\frac{\Gamma \vdash e :_{\uparrow} \tau \to \tau' \quad \Gamma \vdash e' :_{\downarrow} \tau}{\Gamma \vdash e\, e' :_{\uparrow} \tau'}\ \text{app} \qquad \frac{\Gamma \vdash e :_{\uparrow} \tau}{\Gamma \vdash e :_{\downarrow} \tau}\ \text{chk} \qquad \frac{\Gamma, x : \tau \vdash e :_{\downarrow} \tau'}{\Gamma \vdash (\lambda x \,.\, e) :_{\downarrow} \tau \to \tau'}\ \text{lam}$$

Type checked: “$:_{\downarrow} \tau$” “Is this τ right?” Type inferred: “$:_{\uparrow} \tau$” “Found type is τ”

“Context: base type” “Function: base type” “Variable’s type” “Confirm type annotation”

“Infer result of application” “Inferred type checks out” “Check type of lambda”

$\lambda_{\rightarrow}$ Types and Expressions

$\tau ::= \alpha$		base type
	$\tau \rightarrow \tau$	function type
$e ::= e : \tau$		annotated term
	x	variable
	$e\ e$	application
	$\lambda x\ .\ e$	lambda abstraction

$\lambda_{\rightarrow}$ Values

$v ::= n$		neutral term
	$\lambda x\ .\ v$	lambda abstraction
$n ::= x$		variable
	$n\ v$	application

$\lambda_{\rightarrow}$ Type Rules in Bidirectional Style

$$\frac{\Gamma(\alpha) = *}{\Gamma \vdash \alpha : *}\text{var1} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \tau' : *}{\Gamma \vdash \tau \rightarrow \tau' : *}\text{fun} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_{\uparrow} \tau}\text{var2} \qquad \frac{\Gamma \vdash \tau : * \quad \Gamma \vdash e :_{\downarrow} \tau}{\Gamma \vdash (e : \tau) :_{\uparrow} \tau}\text{ann}$$

$$\frac{\Gamma \vdash e :_{\uparrow} \tau \rightarrow \tau' \quad \Gamma \vdash e' :_{\downarrow} \tau}{\Gamma \vdash e\,e' :_{\uparrow} \tau'}\text{app} \qquad \frac{\Gamma \vdash e :_{\uparrow} \tau}{\Gamma \vdash e :_{\downarrow} \tau}\text{chk} \qquad \frac{\Gamma, x : \tau \vdash e :_{\downarrow} \tau'}{\Gamma \vdash (\lambda x\ .\ e) :_{\downarrow} \tau \rightarrow \tau'}\text{lam}$$

$$\epsilon, \alpha : *, y : \alpha \vdash \Big(\big((\lambda x\ .\ x) : (\alpha \rightarrow \alpha)\big)\ y\Big) : \alpha$$

$$\epsilon, \alpha : *, y : \alpha, \beta : * \vdash \Big(\big((\lambda x\ .\ \lambda y\ .\ x) : (\beta \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \rightarrow \beta\big)\ (\lambda z\ .\ z)\ y\Big) : \beta \rightarrow \beta$$

$\lambda\Pi$ (Dependently Typed) Syntax

$e, \rho ::=$	$e : \boxed{\rho}$	annotated term
	$\boxed{*}$	the type of types
	$\boxed{(x : \rho) \longrightarrow \rho}$	dependent function
	x	variable
	$e\ e$	application
	$\lambda x\ .\ e$	lambda abstraction

In $\lambda\Pi$, *everything* is an expression, including type "expressions" ρ and kinds

$(x : \rho) \longrightarrow \rho'$ is the type of a function from ρ to ρ'

Weirich writes $(x : \rho) \longrightarrow \rho'$; Löh et al. write $\forall x :: \rho\ .\ \rho'$; $\Pi x : \rho\ .\ \rho'$ is traditional

This is where $\lambda\Pi$, the dependently typed lambda calculus, gets its Π

$\Pi x : \rho\ .\ \rho'$ parallels $\lambda x\ .\ e$ x is made available to the body ρ'

$\lambda\Pi$ (Dependently Typed) Syntax

$e, \rho ::=$	$e : \rho$	annotated term
	$*$	the type of types
	$(x : \rho) \longrightarrow \rho$	dependent function
	x	variable
	$e\ e$	application
	$\lambda x\ .\ e$	lambda abstraction

An example: the type of the identity function on n-element vectors of α's

$$(\alpha : *) \longrightarrow (n : \mathrm{Nat}) \longrightarrow (v : \mathrm{Vec}\ \alpha\ n) \longrightarrow \mathrm{Vec}\ \alpha\ n$$

The type variable v isn't used

$\lambda\Pi$ (Dependently Typed) Syntax

$e, \rho ::=$	$e : \rho$	annotated term
	$*$	the type of types
	$(x : \rho) \longrightarrow \rho$	dependent function
	x	variable
	$e\ e$	application
	$\lambda x\ .\ e$	lambda abstraction

$\lambda\Pi$ (Dependently Typed) Values

$v, \tau ::=$	n	neutral term
	$*$	the type of types
	$(x : \tau) \longrightarrow \tau$	dependent function
	$\lambda x\ .\ v$	lambda abstraction
$n ::=$	x	variable
	$n\ v$	application

"Types" τ are now just particular values

The kind $*$ is just a particular value

$\lambda\Pi$ (Dependently Typed) Syntax

$$
\begin{array}{rll}
e, \rho ::= & e : \rho & \text{annotated term} \\
 & * & \text{the type of types} \\
 & (x : \rho) \longrightarrow \rho & \text{dependent function} \\
 & x & \text{variable} \\
 & e\; e & \text{application} \\
 & \lambda x \,.\, e & \text{lambda abstraction}
\end{array}
$$

$\lambda\Pi$ (Dependently Typed) Values

$$
\begin{array}{rll}
v, \tau ::= & n & \text{neutral term} \\
 & * & \text{the type of types} \\
 & (x : \tau) \longrightarrow \tau & \text{dependent function} \\
 & \lambda x \,.\, v & \text{lambda abstraction} \\
 & & \\
n ::= & x & \text{variable} \\
 & n\; v & \text{application}
\end{array}
$$

Contexts

$$
\begin{array}{rll}
\Gamma ::= & \epsilon & \text{empty context} \\
 & \Gamma, x : \tau & \text{adding a variable}
\end{array}
$$

No distinction between values and types makes this simple

Valid Contexts

$$
\frac{}{\mathrm{valid}(\epsilon)} \qquad \frac{\mathrm{valid}(\Gamma) \qquad \Gamma \vdash \tau :_{\downarrow} *}{\mathrm{valid}(\Gamma, x : \tau)}
$$

The "type" of a variable in context must check as the type of a type

λΠ (Dependently Typed) Syntax

$e, \rho ::= e : \rho$		annotated term
	$*$	the type of types
	$(x : \rho) \longrightarrow \rho$	dependent function
	x	variable
	$e\ e$	application
	$\lambda x\ .\ e$	lambda abstraction

λΠ (Dependently Typed) Values

$v, \tau ::= n$		neutral term
	$*$	the type of types
	$(x : \tau) \longrightarrow \tau$	dependent function
	$\lambda x\ .\ v$	lambda abstraction
$n ::= x$		variable
	$n\ v$	application

λΠ Big-Step Evaluation Rules

$$\frac{e \Downarrow v}{e : \tau \Downarrow v} \qquad \frac{}{* \Downarrow *} \qquad \frac{\rho \Downarrow \tau \quad \rho' \Downarrow \tau'}{(x : \rho) \longrightarrow \rho' \Downarrow (x : \tau) \longrightarrow \tau'} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e \Downarrow \lambda x\ .\ v \quad v[x := e'] \Downarrow v'}{e\ e' \Downarrow v'} \qquad \frac{e \Downarrow n \quad e' \Downarrow v'}{e\ e' \Downarrow n v'} \qquad \frac{e \Downarrow v}{\lambda x\ .\ e \Downarrow \lambda x\ .\ v}$$

Type expressions (ρ) in function type terms are evaluated to types (τ); the types remain

$e, \rho \; ::= e : \rho \mid * \mid (x : \rho) \longrightarrow \rho \mid x \mid e\, e \mid \lambda x \,.\, e$ $\quad v, \tau ::= n \mid * \mid (x : \tau) \longrightarrow \tau \mid \lambda x \,.\, v$ $\quad n ::= x \mid n\, v$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

$\lambda\Pi$ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\,\text{ann}$$

Checking that ρ is a $*$ now uses "regular" type rules

ρ is now a type expression, so it is reduced to a type τ before checking the type of body e

Type checking now requires *executing* type expressions

$$e, \rho \ ::= e : \rho \mid * \mid (x : \rho) \longrightarrow \rho \mid x \mid e\, e \mid \lambda x \,.\, e \qquad v, \tau ::= n \mid * \mid (x : \tau) \longrightarrow \tau \mid \lambda x \,.\, v \qquad n ::= x \mid n\, v$$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

$\lambda\Pi$ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\,\text{ann} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\,\text{var}$$

The single *var* rule now handles values, types, and kinds

$e, \rho ::= e : \rho \mid * \mid (x : \rho) \longrightarrow \rho \mid x \mid e\,e \mid \lambda x \,.\, e \qquad v, \tau ::= n \mid * \mid (x : \tau) \longrightarrow \tau \mid \lambda x \,.\, v \qquad n ::= x \mid n\,v$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

λΠ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\text{ann} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\text{var} \qquad \frac{}{\Gamma \vdash * :_\uparrow *}\text{star}$$

The kind $*$ is of type $*$

This simple choice leaves the type system unsound (allows a kind of Russell's paradox)

Choosing $* : *_1, *_1 : *_2, *_2 : *_3$, etc. solves the soundness problem

$e, \rho ::= e : \rho \mid * \mid (x : \rho) \longrightarrow \rho \mid x \mid e\, e \mid \lambda x \,.\, e \qquad v, \tau ::= n \mid * \mid (x : \tau) \longrightarrow \tau \mid \lambda x \,.\, v \qquad n ::= x \mid n\, v$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

$\lambda\Pi$ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\,\text{ann} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\,\text{var} \qquad \frac{}{\Gamma \vdash * :_\uparrow *}\,\text{star}$$

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma, x : \tau \vdash \rho' :_\downarrow *}{\Gamma \vdash (x : \rho) \longrightarrow \rho' :_\uparrow *}\,\text{pi}$$

This replaces the *fun* rule in $\lambda\!\!\longrightarrow$, which concluded $\tau \longrightarrow \tau' : *$

For functions from ρ to ρ', both ρ and ρ' must have kind $*$

However, ρ is reduced to type τ and passed to ρ' through the context (dependency)

$$e, \rho \;::=\; e : \rho \mid * \mid (x : \rho) \longrightarrow \rho \mid x \mid e\, e \mid \lambda x \,.\, e \qquad v, \tau ::= n \mid * \mid (x : \tau) \longrightarrow \tau \mid \lambda x \,.\, v \qquad n ::= x \mid n\, v$$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

λΠ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\,\text{ann} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\,\text{var} \qquad \frac{}{\Gamma \vdash * :_\uparrow *}\,\text{star}$$

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma, x : \tau \vdash \rho' :_\downarrow *}{\Gamma \vdash (x : \rho) \longrightarrow \rho' :_\uparrow *}\,\text{pi} \qquad \frac{\Gamma, x : \tau \vdash e :_\downarrow \tau'}{\Gamma \vdash (\lambda x \,.\, e) :_\downarrow (x : \tau) \longrightarrow \tau'}\,\text{lam}$$

The $\lambda\!\longrightarrow$ version checked $(\lambda x \,.\, e) : \tau \longrightarrow \tau'$; this checks an equivalent term

$e, \rho ::= e : \rho \mid * \mid (x : \rho) \to \rho \mid x \mid e\, e \mid \lambda x \,.\, e \qquad v, \tau ::= n \mid * \mid (x : \tau) \to \tau \mid \lambda x \,.\, v \qquad n ::= x \mid n\, v$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

λΠ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \quad \rho \Downarrow \tau \quad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\text{ann} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\text{var} \qquad \frac{}{\Gamma \vdash * :_\uparrow *}\text{star}$$

$$\frac{\Gamma \vdash \rho :_\downarrow * \quad \rho \Downarrow \tau \quad \Gamma, x : \tau \vdash \rho' :_\downarrow *}{\Gamma \vdash (x : \rho) \to \rho' :_\uparrow *}\text{pi} \qquad \frac{\Gamma, x : \tau \vdash e :_\downarrow \tau'}{\Gamma \vdash (\lambda x \,.\, e) :_\downarrow (x : \tau) \to \tau'}\text{lam}$$

$$\frac{\Gamma \vdash e :_\uparrow (x : \tau) \to \tau' \quad \Gamma \vdash e' :_\downarrow \tau \quad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\, e' :_\uparrow \tau''}\text{app}$$

Instead of $\tau \to \tau'$, we infer $(x : \tau) \to \tau'$

$(x : \tau) \to \tau'$ provides the type variable x to τ' via a substitution

$e, \rho ::= e : \rho \mid * \mid (x : \rho) \longrightarrow \rho \mid x \mid e\, e \mid \lambda x \,.\, e \qquad v, \tau ::= n \mid * \mid (x : \tau) \longrightarrow \tau \mid \lambda x \,.\, v \qquad n ::= x \mid n\, v$

Type checked: "$:_\downarrow \tau$" Type inferred: "$:_\uparrow \tau$"

λΠ Type Rules

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\downarrow \tau}{\Gamma \vdash (e : \rho) :_\uparrow \tau}\,\text{ann} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_\uparrow \tau}\,\text{var} \qquad \frac{}{\Gamma \vdash * :_\uparrow *}\,\text{star}$$

$$\frac{\Gamma \vdash \rho :_\downarrow * \qquad \rho \Downarrow \tau \qquad \Gamma, x : \tau \vdash \rho' :_\downarrow *}{\Gamma \vdash (x : \rho) \longrightarrow \rho' :_\uparrow *}\,\text{pi} \qquad \frac{\Gamma, x : \tau \vdash e :_\downarrow \tau'}{\Gamma \vdash (\lambda x \,.\, e) :_\downarrow (x : \tau) \longrightarrow \tau'}\,\text{lam}$$

$$\frac{\Gamma \vdash e :_\uparrow (x : \tau) \longrightarrow \tau' \qquad \Gamma \vdash e' :_\downarrow \tau \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\, e' :_\uparrow \tau''}\,\text{app} \qquad \frac{\Gamma \vdash e :_\uparrow \tau}{\Gamma \vdash e :_\downarrow \tau}\,\text{chk}$$

This says "given a type τ, we can conclude e has that type if we can infer type τ for e"