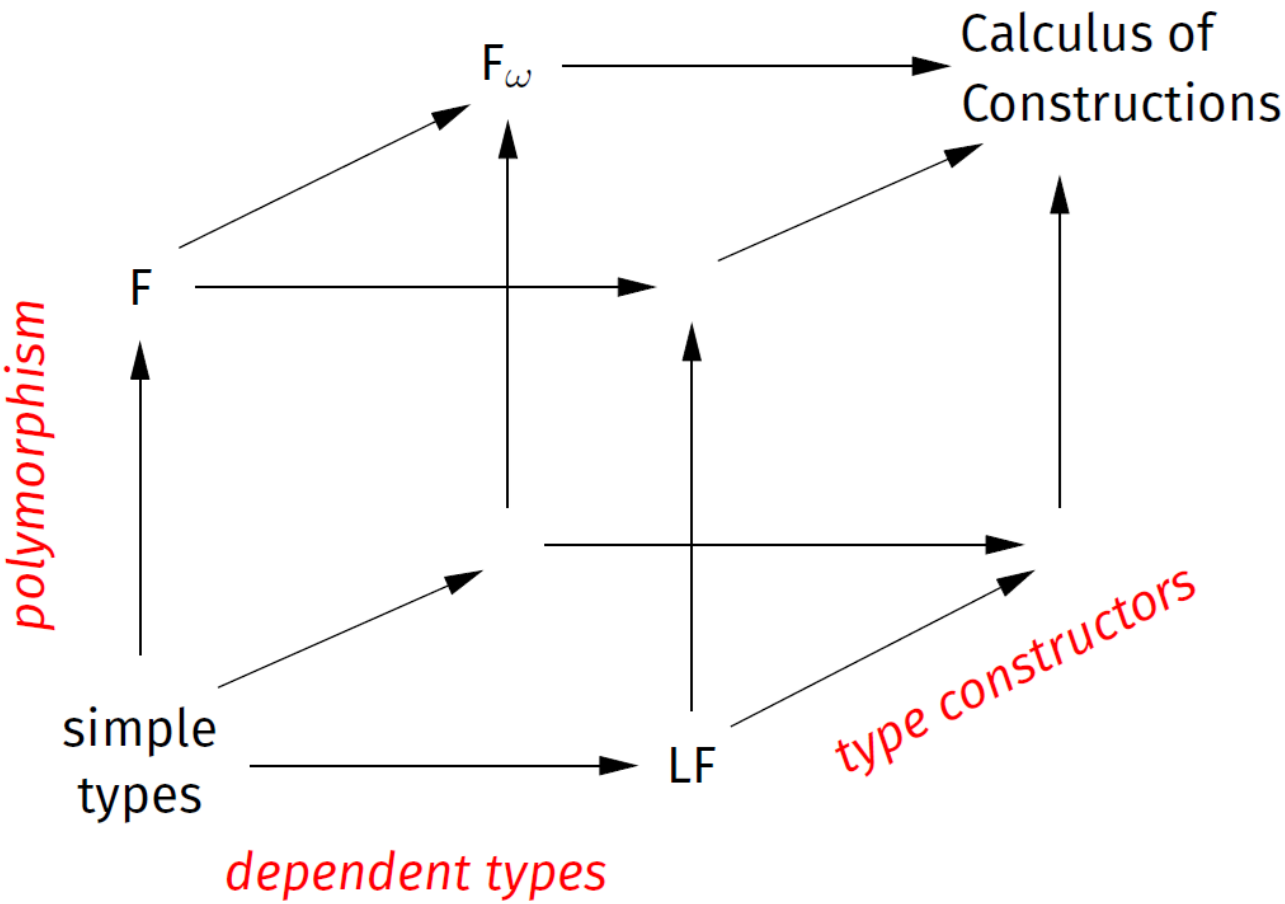
System F_ω: Higher-Order Polymorphism

Reference: Chapters 29 and 30 in Pierce's *TAPL*

Lambda Cube



This class: System F_ω.

Recall System F

$$
\begin{array}{rrcl}
\text{(Terms)} & M & ::= & x \mid \lambda x : \tau.\, M \mid M\, M \mid \Lambda\alpha.\, M \mid M\,\langle\tau\rangle \\
\text{(Types)} & \tau & ::= & \alpha \mid \mathtt{T} \mid \tau \to \tau \mid \forall\alpha.\, \tau \\
\text{(Values)} & v & ::= & \lambda x : \tau.\, M \mid \Lambda\alpha.\, M \\
\\
\text{(Contexts)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\text{(TypeVarContexts)} & \Delta & ::= & \bullet \mid \Delta, \alpha
\end{array}
$$

Reduction

$$\frac{}{(\lambda x : \tau.\, M_1)\, M_2 \longrightarrow M_1[M_2/x]}\ \text{(E-AppAbs)}$$

$$\frac{}{(\Lambda\alpha.\, M_1)\,\langle\tau_2\rangle \longrightarrow M_1[\tau_2/\alpha]}\ \text{(E-TAppTAbs)}$$

Recall System F

Type well-formedness: $\Delta \vdash \tau$

Typing judgment: $\Delta; \Gamma \vdash M : \tau$

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau}\ (\text{T-Var}) \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta; \Gamma, x : \tau_1 \vdash M : \tau_2}{\Delta; \Gamma \vdash (\lambda x : \tau_1.\ M) : \tau_1 \to \tau_2}\ (\text{T-Abs})$$

$$\frac{\Delta; \Gamma \vdash M_1 : \tau \to \tau' \qquad \Delta; \Gamma \vdash M_2 : \tau}{\Delta; \Gamma \vdash M_1\ M_2 : \tau'}\ (\text{T-App})$$

$$\frac{\Delta, \alpha; \Gamma \vdash M : \tau}{\Delta; \Gamma \vdash (\Lambda\alpha.\ M) : \forall\alpha.\tau}\ (\text{T-TAbs}) \qquad \frac{\Delta; \Gamma \vdash M_1 : \forall\alpha.\tau \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash M_1\ \langle\tau_2\rangle : \tau[\tau_2/\alpha]}\ (\text{T-TApp})$$

Recall System F

We can encode boolean values:

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, x \\ \text{False} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, y \end{aligned}$$

Their type: $\text{Bool} \stackrel{\text{def}}{=} \forall\alpha.\, \alpha \to \alpha \to \alpha.$

Recall System F

We can encode boolean values:

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, x \\ \text{False} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, y \end{aligned}$$

Their type: $\text{Bool} \stackrel{\text{def}}{=} \forall\alpha.\, \alpha \to \alpha \to \alpha.$

We can also encode a pair (x, y), where x has type $\mathtt{T}_1$ and y has type $\mathtt{T}_2$:

$$(x, y) \stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda f : \mathtt{T}_1 \to \mathtt{T}_2 \to \alpha.\, f\, x\, y$$

Its type:

$$\text{Pair}\ \mathtt{T}_1\ \mathtt{T}_2 \stackrel{\text{def}}{=} \forall\alpha.\, (\mathtt{T}_1 \to \mathtt{T}_2 \to \alpha) \to \alpha$$

Type Operators (a.k.a. Type Constructors)

Type operators: type-level abstraction.

"*Types depend on types.*"

Example

$$\text{Pair} \stackrel{\text{def}}{=} \lambda\alpha_1.\, \lambda\alpha_2.\, \forall\alpha.\, (\alpha_1 \to \alpha_2 \to \alpha) \to \alpha$$

Kinds

The abstraction and application mechanism at the *type* level gives us the ability to write meaningless type expressions. For example, (Bool Nat) makes no more sense than (True 6) at the term level.

To prevent this sort of nonsense, we introduce a system of kinds that classify type expressions.

$$\text{(Kinds)} \quad \kappa \quad ::= \quad * \mid \kappa \to \kappa$$

Kinds are "the types of types". Examples:

type	Bool	Bool → Bool	Pair
kind	$*$	$*$	$* \to * \to *$

Example: Pair

$$\mathsf{Pair}\ \mathtt{T}_1\ \mathtt{T}_2 \stackrel{\text{def}}{=} \forall\alpha.\ (\mathtt{T}_1 \to \mathtt{T}_2 \to \alpha) \to \alpha$$

$$(x, y) \stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda f : \mathtt{T}_1 \to \mathtt{T}_2 \to \alpha.\ f\ x\ y$$

$$\mathsf{Pair} \stackrel{\text{def}}{=} \lambda\alpha_1 :: *.\ \lambda\alpha_2 :: *.\ \forall\alpha.\ (\alpha_1 \to \alpha_2 \to \alpha) \to \alpha$$

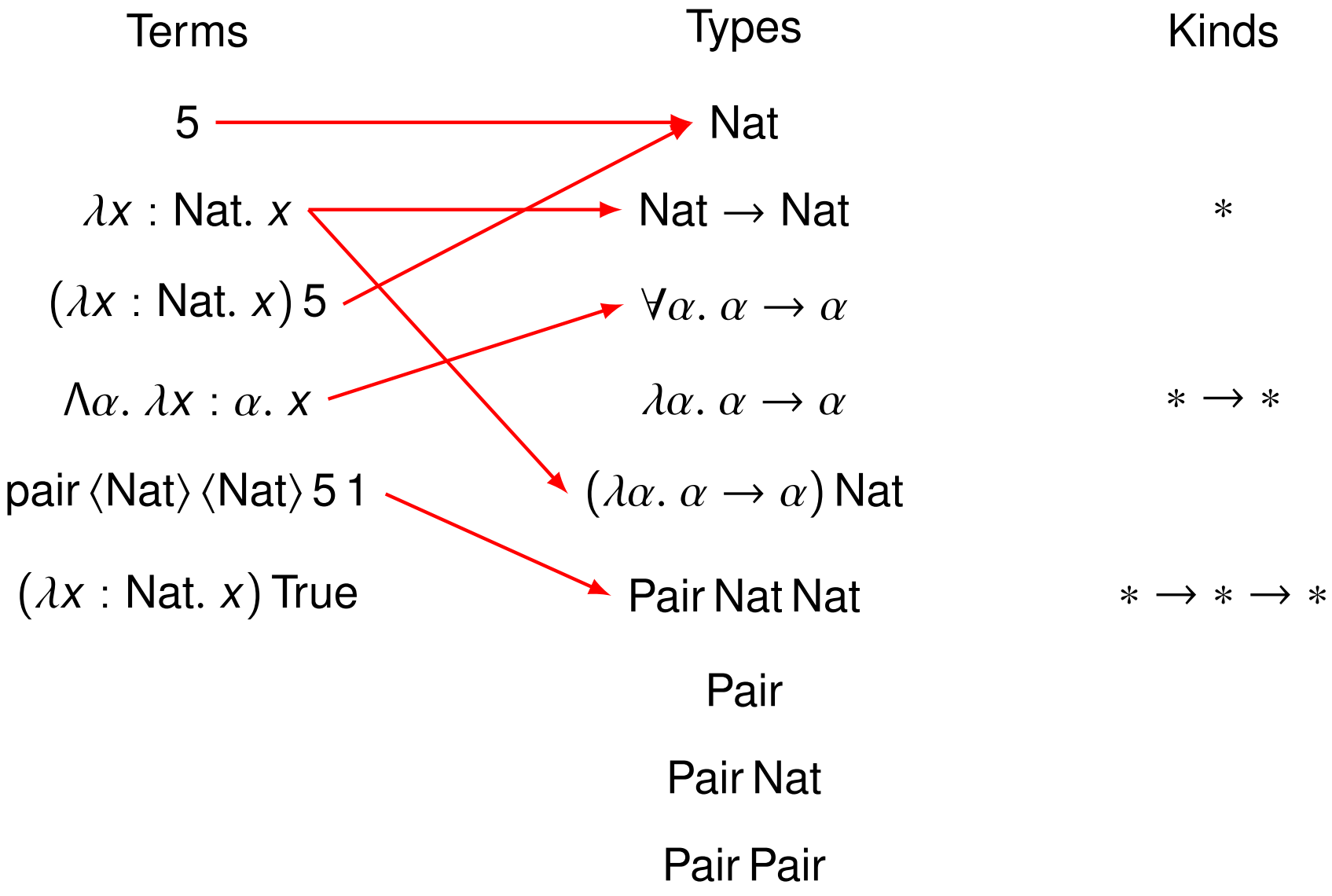
$$\mathsf{pair} \stackrel{\text{def}}{=} \Lambda\alpha_1.\ \Lambda\alpha_2.\ \lambda x : \alpha_1.\ \lambda y : \alpha_2.\ \Lambda\alpha.\ \lambda f : \alpha_1 \to \alpha_2 \to \alpha.\ f\ x\ y$$

$$(x, y) \stackrel{\text{def}}{=} \mathsf{pair}\ \langle\mathtt{T}_1\rangle\ \langle\mathtt{T}_2\rangle\ x\ y$$

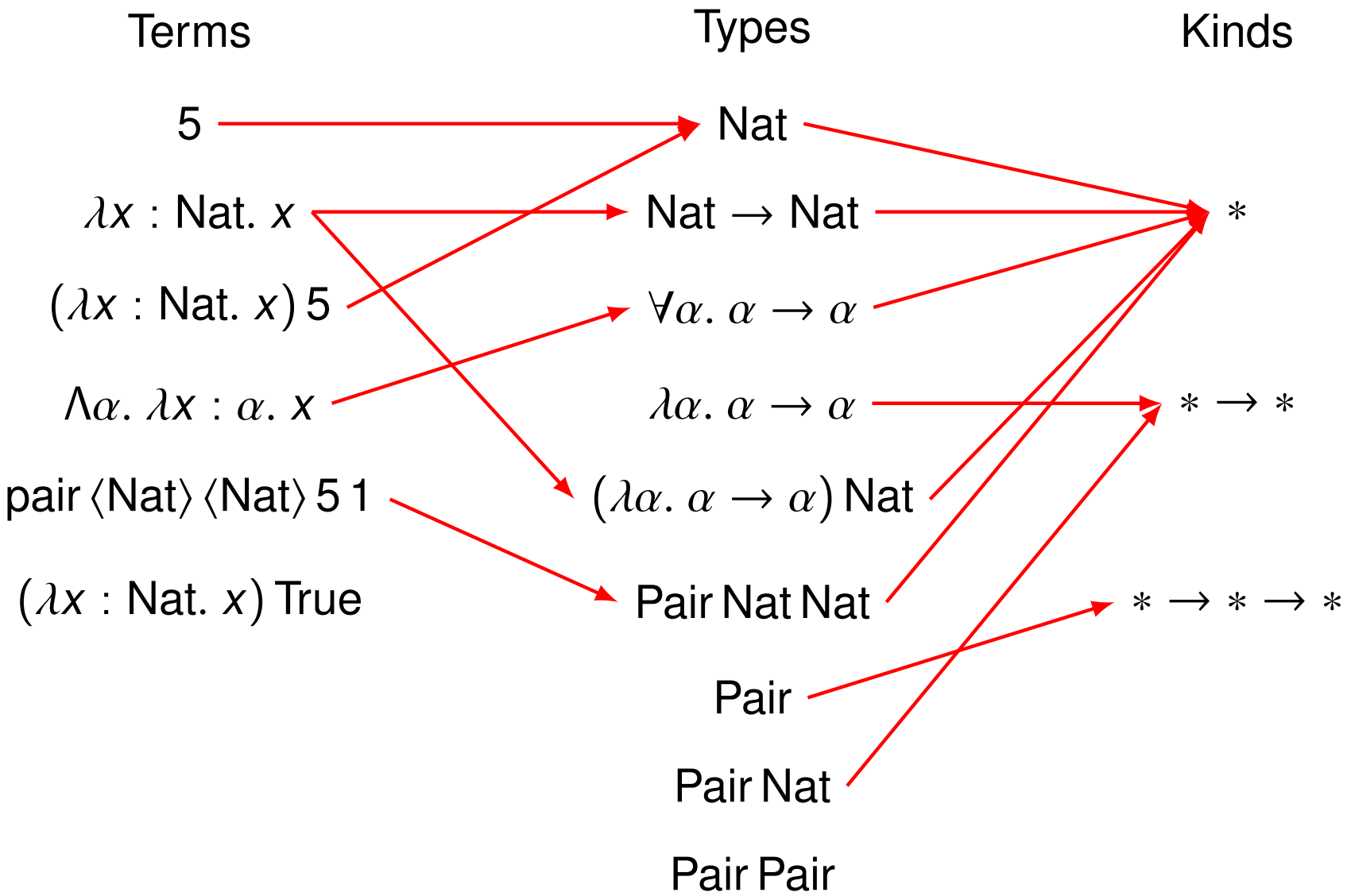
Terms, Types and Kinds

Terms	Types	Kinds
5	Nat	
$\lambda x : \mathsf{Nat}.\ x$	$\mathsf{Nat} \to \mathsf{Nat}$	$*$
$(\lambda x : \mathsf{Nat}.\ x)\ 5$	$\forall \alpha.\ \alpha \to \alpha$	
$\Lambda \alpha.\ \lambda x : \alpha.\ x$	$\lambda \alpha.\ \alpha \to \alpha$	$* \to *$
$\mathsf{pair}\ \langle \mathsf{Nat} \rangle\ \langle \mathsf{Nat} \rangle\ 5\ 1$	$(\lambda \alpha.\ \alpha \to \alpha)\ \mathsf{Nat}$	
$(\lambda x : \mathsf{Nat}.\ x)\ \mathsf{True}$	$\mathsf{Pair}\ \mathsf{Nat}\ \mathsf{Nat}$	$* \to * \to *$
	Pair	
	$\mathsf{Pair}\ \mathsf{Nat}$	
	$\mathsf{Pair}\ \mathsf{Pair}$	

Terms, Types and Kinds



Terms, Types and Kinds



Discussions

Why stop at three levels of expressions? Couldn't we go on to add a fourth level to classify "kinds", and so continue without end?

Such systems have been investigated by the *pure type systems* community.

However, for programming languages, three levels have proved sufficient. In essentially all statically typed programming languages, it is relatively rare for language designers to offer programmers even the full power of the type operators. Some languages (e.g., Java) offer only a few built-in type operators like Array, with no facilities for defining new ones.

$\lambda\underline{\omega}$: Simply-Typed λ-Calculus with Type Operators

$$
\begin{array}{rrcl}
\text{(Terms)} & M & ::= & x \mid \lambda x : \tau.\, M \mid M\, M \\
\text{(Types)} & \tau & ::= & \alpha \mid \mathsf{T} \mid \tau \to \tau \mid \lambda \alpha :: \kappa.\, \tau \mid \tau\, \tau \\
\text{(Kinds)} & \kappa & ::= & * \mid \kappa \to \kappa \\
\text{(Values)} & v & ::= & \lambda x : \tau.\, M
\end{array}
$$

- Type variable α
- Operator abstraction $\lambda \alpha :: \kappa.\, \tau$
- Operator application $\tau\, \tau$
- Kind of proper types $*$
- Kind of operators $\kappa \to \kappa$

Reduction rules are the same as STLC.

$\lambda\underline{\omega}$ – Statics

$$
\begin{array}{rrcl}
\text{(Terms)} & M & ::= & x \mid \lambda x : \tau.\, M \mid M\, M \\
\text{(Types)} & \tau & ::= & \alpha \mid \mathtt{T} \mid \tau \to \tau \mid \lambda \alpha :: \kappa.\, \tau \mid \tau\, \tau \\
\text{(Kinds)} & \kappa & ::= & * \mid \kappa \to \kappa \\
\text{(Values)} & v & ::= & \lambda x : \tau.\, M \\
 & & & \\
\text{(Contexts)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\text{(TypeVarContexts)} & \Delta & ::= & \bullet \mid \Delta, \alpha :: \kappa
\end{array}
$$

Kinding judgment: $\Delta \vdash \tau :: \kappa$

Type equivalence: $\tau \equiv \tau'$

Typing judgment: $\Delta; \Gamma \vdash M : \tau$

$\lambda\underline{\omega}$ – Statics – Kinding

$$\frac{}{\Delta, \alpha :: \kappa \vdash \alpha :: \kappa} \qquad \frac{}{\Delta \vdash \mathsf{T} :: *} \qquad \frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \tau_1 \to \tau_2 :: *}$$

$$\frac{\Delta, \alpha :: \kappa \vdash \tau :: \kappa'}{\Delta \vdash \lambda \alpha :: \kappa.\, \tau :: \kappa \to \kappa'} \qquad \frac{\Delta \vdash \tau_1 :: \kappa \to \kappa' \qquad \Delta \vdash \tau_2 :: \kappa}{\Delta \vdash \tau_1\, \tau_2 :: \kappa'}$$

$\lambda\underline{\omega}$ – Statics – Type Equivalence

$$\frac{}{(\lambda\alpha :: \kappa.\, \tau_1)\, \tau_2 \equiv \tau_1[\tau_2/\alpha]}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}$$

$$\frac{\tau \equiv \tau'}{\lambda\alpha :: \kappa.\, \tau \equiv \lambda\alpha :: \kappa.\, \tau'} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\, \tau_2 \equiv \tau_1'\, \tau_2'}$$

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau \equiv \tau'}{\tau' \equiv \tau} \qquad \frac{\tau \equiv \tau' \qquad \tau \equiv \tau''}{\tau \equiv \tau''}$$

$\lambda\underline{\omega}$ – Statics – Typing

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau} \text{(T-Var)}$$

$$\frac{\Delta \vdash \tau_1 :: * \qquad \Delta; \Gamma, x : \tau_1 \vdash M : \tau_2}{\Delta; \Gamma \vdash (\lambda x : \tau_1.\, M) : \tau_1 \to \tau_2} \text{(T-Abs)}$$

$$\frac{\Delta; \Gamma \vdash M_1 : \tau \to \tau' \qquad \Delta; \Gamma \vdash M_2 : \tau}{\Delta; \Gamma \vdash M_1\, M_2 : \tau'} \text{(T-App)}$$

$$\frac{\Delta; \Gamma \vdash M : \tau \qquad \tau \equiv \tau' \qquad \Delta \vdash \tau' :: *}{\Delta; \Gamma \vdash M : \tau'} \text{(T-Eq)}$$

F_ω: Combining System F and $\lambda\underline{\omega}$

$$
\begin{array}{rrcl}
\text{(Terms)} & M & ::= & x \mid \lambda x : \tau.\, M \mid M\, M \mid \Lambda\alpha :: \kappa.\, M \mid M\langle\tau\rangle \\
\text{(Types)} & \tau & ::= & \alpha \mid \mathsf{T} \mid \tau \to \tau \mid \forall\alpha :: \kappa.\, \tau \mid \lambda\alpha :: \kappa.\, \tau \mid \tau\, \tau \\
\text{(Kinds)} & \kappa & ::= & * \mid \kappa \to \kappa \\
\text{(Values)} & v & ::= & \lambda x : \tau.\, M \mid \Lambda\alpha :: \kappa.\, M \\
 & & & \\
\text{(Ctxts)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\text{(TVCtxts)} & \Delta & ::= & \bullet \mid \Delta, \alpha :: \kappa
\end{array}
$$

F_ω – Reduction

$$\frac{}{(\lambda x : \tau.\ M_1)\, M_2 \longrightarrow M_1[M_2/x]}\ \text{(E-AppAbs)}$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \text{(E-App1)} \qquad \frac{M_2 \longrightarrow M_2'}{M_1\ M_2 \longrightarrow M_1'\ M_2'}\ \text{(E-App2)}$$

$$\frac{M \longrightarrow M'}{\lambda x : \tau.\ M \longrightarrow \lambda x : \tau.\ M'}\ \text{(E-Abs)}$$

$$\frac{}{(\Lambda \alpha :: \kappa.\ M_1)\, \langle \tau_2 \rangle \longrightarrow M_1[\tau_2/\alpha]}\ \text{(E-TAppTAbs)}$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\ \langle \tau_2 \rangle \longrightarrow M_1'\ \langle \tau_2 \rangle}\ \text{(E-TApp)} \qquad \frac{M \longrightarrow M'}{\Lambda \alpha :: \kappa.\ M \longrightarrow \Lambda \alpha :: \kappa.\ M'}\ \text{(E-TAbs)}$$

F_ω – Statics – Kinding

$$\frac{}{\Delta, \alpha :: \kappa \vdash \alpha :: \kappa} \qquad \frac{}{\Delta \vdash \mathsf{T} :: *} \qquad \frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \tau_1 \to \tau_2 :: *}$$

$$\frac{\Delta, \alpha :: \kappa \vdash \tau :: \kappa'}{\Delta \vdash \lambda \alpha :: \kappa.\, \tau :: \kappa \to \kappa'} \qquad \frac{\Delta \vdash \tau_1 :: \kappa \to \kappa' \qquad \Delta \vdash \tau_2 :: \kappa}{\Delta \vdash \tau_1\, \tau_2 :: \kappa'}$$

$$\frac{\Delta, \alpha :: \kappa \vdash \tau :: *}{\Delta \vdash \forall \alpha :: \kappa.\, \tau :: *}$$

F_ω – Statics – Type Equivalence

$$\frac{}{(\lambda\alpha :: \kappa.\ \tau_1)\,\tau_2 \equiv \tau_1[\tau_2/\alpha]}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\tau \equiv \tau'}{\forall\alpha :: \kappa.\ \tau \equiv \forall\alpha :: \kappa.\ \tau'}$$

$$\frac{\tau \equiv \tau'}{\lambda\alpha :: \kappa.\ \tau \equiv \lambda\alpha :: \kappa.\ \tau'} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\,\tau_2 \equiv \tau_1'\,\tau_2'}$$

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau \equiv \tau'}{\tau' \equiv \tau} \qquad \frac{\tau \equiv \tau' \qquad \tau \equiv \tau''}{\tau \equiv \tau''}$$

F_ω – Statics – Typing

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau} \text{ (T-Var)}$$

$$\frac{\Delta \vdash \tau_1 :: * \qquad \Delta; \Gamma, x : \tau_1 \vdash M : \tau_2}{\Delta; \Gamma \vdash (\lambda x : \tau_1.\, M) : \tau_1 \to \tau_2} \text{ (T-Abs)}$$

$$\frac{\Delta; \Gamma \vdash M_1 : \tau \to \tau' \qquad \Delta; \Gamma \vdash M_2 : \tau}{\Delta; \Gamma \vdash M_1\, M_2 : \tau'} \text{ (T-App)}$$

$$\frac{\Delta, \alpha :: \kappa; \Gamma \vdash M : \tau}{\Delta; \Gamma \vdash (\Lambda \alpha :: \kappa.\, M) : \forall \alpha :: \kappa . \tau} \text{ (T-TAbs)}$$

$$\frac{\Delta; \Gamma \vdash M_1 : \forall \alpha :: \kappa . \tau \qquad \Delta \vdash \tau_2 :: \kappa}{\Delta; \Gamma \vdash M_1 \langle \tau_2 \rangle : \tau[\tau_2/\alpha]} \text{ (T-TApp)}$$

$$\frac{\Delta; \Gamma \vdash M : \tau \qquad \tau \equiv \tau' \qquad \Delta \vdash \tau' :: *}{\Delta; \Gamma \vdash M : \tau'} \text{ (T-Eq)}$$

F_ω – Properties

Theorem (Preservation)

For all M, M' and τ, if $\bullet; \bullet \vdash M : \tau$ and $M \longrightarrow M'$, then $\bullet; \bullet \vdash M' : \tau$.

Theorem (Progress)

For all M and τ, if $\bullet; \bullet \vdash M : \tau$, then either $M \in$ Values or $\exists M'.\ M \longrightarrow M'$.

Strong normalization: Every reduction path starting from a well-typed F_ω term is guaranteed to terminate.

Going Further: Dependent Types

lambda abstraction	$\lambda x : \tau.\ M$	terms depend on terms
parametric polymorphism	$\Lambda\alpha :: \kappa.\ M$	terms depend on types
type constructors	$\lambda\alpha :: \kappa.\ \tau$	types depend on types

Going Further: Dependent Types

lambda abstraction	$\lambda x : \tau.\ M$	terms depend on terms
parametric polymorphism	$\Lambda\alpha :: \kappa.\ M$	terms depend on types
type constructors	$\lambda\alpha :: \kappa.\ \tau$	types depend on types
dependent function types	$\Pi x : \tau_1.\ \tau_2$	types depend on terms

(In the degenerate case, when τ_2 does not mention x, we write $\Pi x : \tau_1.\ \tau_2$ as $\tau_1 \to \tau_2$.)

Dependent Function Types – Examples

FloatList n represents the type of lists with n float elements.

$$\begin{array}{lcl} \text{nil} & : & \text{FloatList } 0 \\ \text{cons} & : & \Pi n : \text{Nat. Float} \to \text{FloatList } n \to \text{FloatList (succ } n) \\ \text{hd} & : & \Pi n : \text{Nat. FloatList (succ } n) \to \text{Float} \\ \text{tl} & : & \Pi n : \text{Nat. FloatList (succ } n) \to \text{FloatList } n \end{array}$$

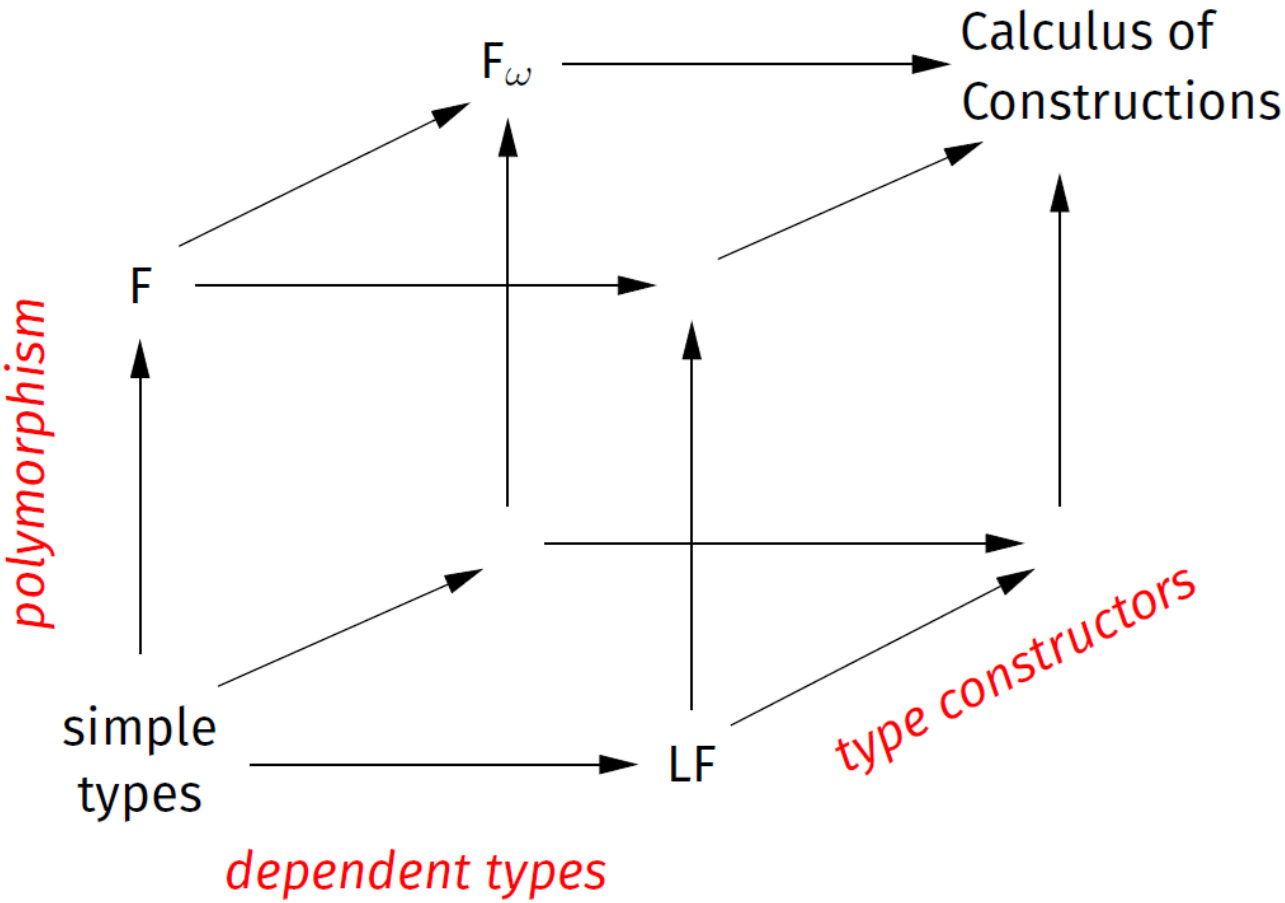
Dependent Function Types – Examples

$$\begin{aligned}\text{consthree} \stackrel{\text{def}}{=} \lambda n : \text{Nat}.\ \lambda f : \text{Float}.\ \lambda l : \text{FloatList}\ n.\\ \text{cons}\ (\text{succ}\ (\text{succ}\ n))\ f\\ \text{cons}\ (\text{succ}\ n)\ f\\ (\text{cons}\ n\ f\ l)\end{aligned}$$

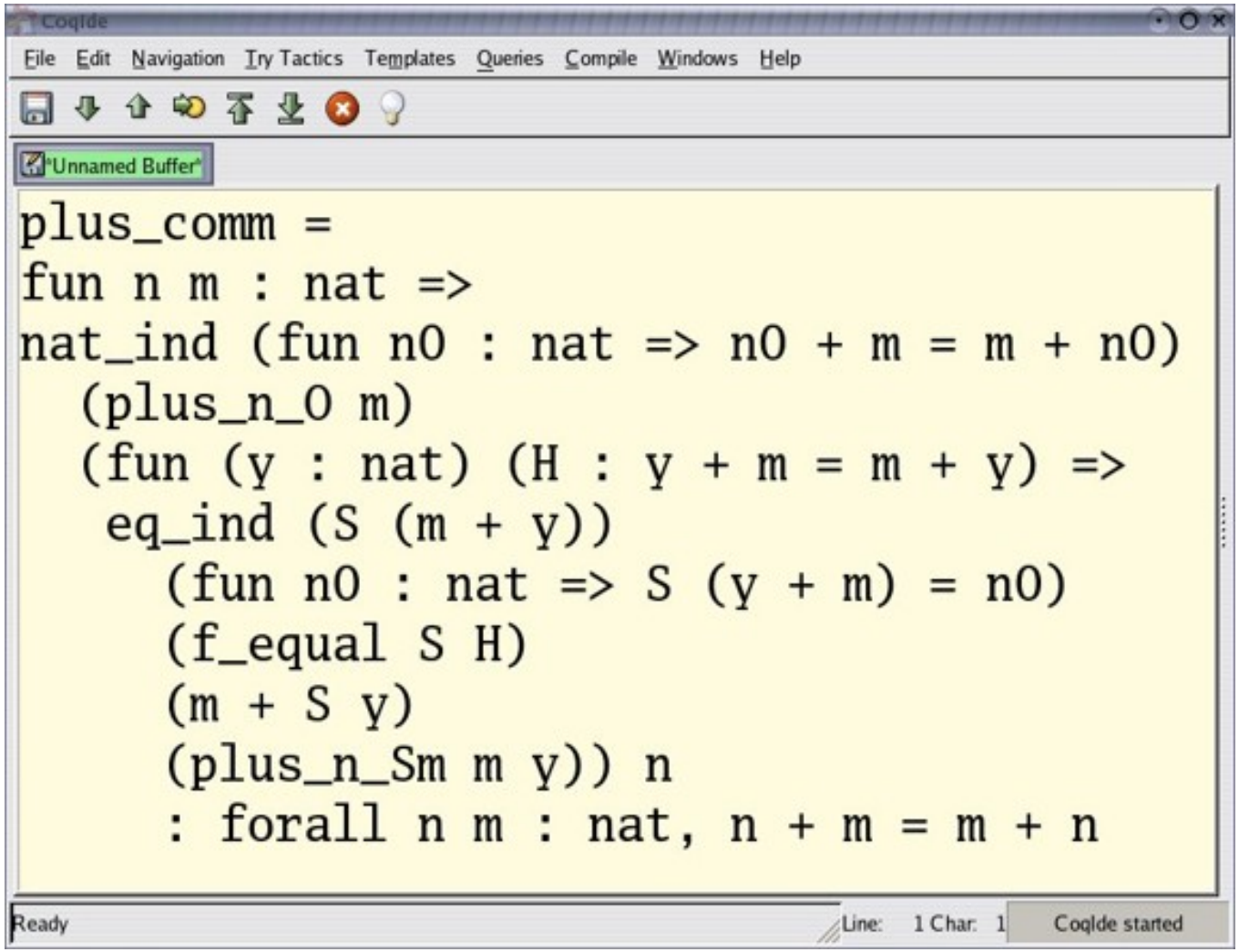
Its type:

$$\Pi n : \text{Nat}.\ \text{Float} \to \text{FloatList}\ n \to \text{FloatList}\ (\text{succ}\ (\text{succ}\ (\text{succ}\ n)))$$

Lambda Cube


F_ω
Calculus of
Constructions
F
polymorphism
simple
types
LF
type constructors
dependent types

Coq's Basis: Calculus of Inductive Constructions



Lambda Cube – Syntax

Unify terms and types:

$$A \quad ::= \quad x \mid s \mid A\,A \mid \lambda x : A.\, A \mid \Pi x : A.\, A$$

Here the sort s is either $*$, denoting the type of types, or $\Box$, which is one level up, denoting the type of kinds (such as $* \to *$ and $\Pi x : *.\, *$).

$$s \quad ::= \quad * \mid \Box$$

Lambda Cube – Syntax

Unify terms and types:

$$A \quad ::= \quad x \mid s \mid A\,A \mid \lambda x : A.\, A \mid \Pi x : A.\, A$$

Here the sort s is either $*$, denoting the type of types, or $\Box$, which is one level up, denoting the type of kinds (such as $* \to *$ and $\Pi x : *.\, *$).

$$s \quad ::= \quad * \mid \Box$$

Typing context:

$$\Gamma \quad ::= \quad \bullet \mid \Gamma, x : A$$

Lambda Cube – Typing Rules

$$\frac{}{\bullet \vdash * : \Box}\ (\textsc{Sort}) \qquad \frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash x : A}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s \qquad x \notin \Gamma}{\Gamma, x : C \vdash A : B}\ (\textsc{Weak})$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.\, B \ : \ s_2}\ (\textsc{Form})$$

$$\frac{\Gamma \vdash M \ : \ \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B[A/x]}\ (\textsc{App})$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B \ : \ s}{\Gamma \vdash \lambda x : A.\, M \ : \ \Pi x : A.\, B}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}\ (\textsc{Conv})$$

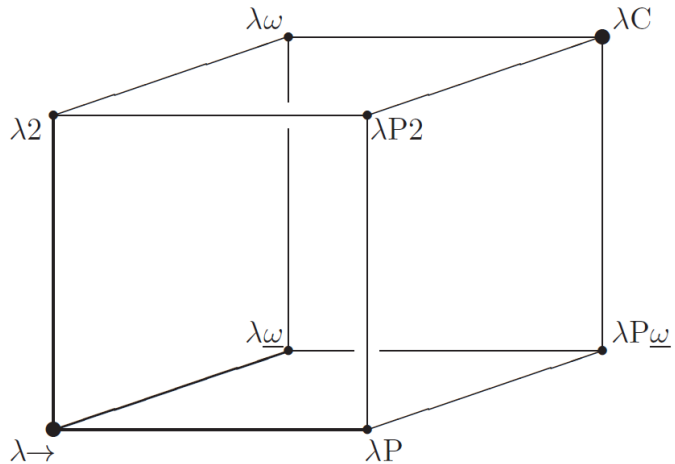
Lambda Cube – Expressing Different Dependencies

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.\, B \;:\; s_2}\ (\textsc{Form})$$

$x : A : s_1$	$b : B : s_2$	(s_1, s_2)	$\lambda x : A.b$	from
$*$	$*$	$(*, *)$	term depending on term	STLC
$\Box$	$*$	$(\Box, *)$	term depending on type	System F
$\Box$	$\Box$	$(\Box, \Box)$	type depending on type	$\lambda\underline{\omega}$
$*$	$\Box$	$(*, \Box)$	type depending on term	LF

Lambda Cube

system:	combinations (s_1, s_2) allowed:			
$\lambda\!\to$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
$\lambda\underline{\omega}$	$(*,*)$		$(\Box,\Box)$	
$\lambda\mathrm{P}$	$(*,*)$			$(*,\Box)$
$\lambda\omega$	$(*,*)$	$(\Box,*)$	$(\Box,\Box)$	
$\lambda\mathrm{P}2$	$(*,*)$	$(\Box,*)$		$(*,\Box)$
$\lambda\mathrm{P}\underline{\omega}$	$(*,*)$		$(\Box,\Box)$	$(*,\Box)$
$\lambda\mathrm{P}\omega = \lambda\mathrm{C}$	$(*,*)$	$(\Box,*)$	$(\Box,\Box)$	$(*,\Box)$



Taken from the book *Type Theory and Formal Proof: An Introduction* by Rob Nederpelt and Herman Geuvers.