Simply typed functions:
  type of result depends on
  type of argument, but not its value

VS

Dependently typed functions:
  type of result depends on
  type of argument and on its value
Functions on types

In PLC, $\Lambda \alpha (M)$ is an anonymous notation for the function $F$ mapping each type $\tau$ to the value of $M[\tau/\alpha]$ (of some particular type).
Functions on types

In PLC, $\Lambda \alpha (M)$ is an anonymous notation for the function $F$ mapping each type $\tau$ to the value of $M[\tau/\alpha]$ (of some particular type).

So $\forall \alpha (\tau)$ is a type of "dependently-typed" functions.
Dependent Functions

Given a set $A$ and a family of sets $B_a$ indexed by the elements $a$ of $A$, we get a set

$$\prod_{a \in A} B_a \triangleq \{ F \in A \to \bigcup_{a \in A} B_a \mid \forall (a, b) \in F \ (b \in B_a) \}$$

of dependent functions. Each $F \in \prod_{a \in A} B_a$ is a single-valued and total relation that associates to each $a \in A$ an element in $B_a$ (usually written $Fa$).
Dependent Functions

Given a set \( A \) and a family of sets \( B_a \) indexed by the elements \( a \) of \( A \), we get a set

\[
\prod_{a \in A} B_a \triangleq \{ F \in A \to \bigcup_{a \in A} B_a \mid \forall (a, b) \in F (b \in B_a) \}
\]

of dependent functions. Each \( F \in \prod_{a \in A} B_a \) is a single-valued and total relation that associates to each \( a \in A \) an element in \( B_a \) (usually written \( Fa \)).

For example if \( A = \mathbb{N} \) and for each \( n \in \mathbb{N}, B_n = \{0,1\}^n \to \{0,1\} \), then \( \prod_{n \in \mathbb{N}} B_n \) consists of functions mapping each number \( n \) to an \( n \)-ary Boolean operation.
fun taut x f = if x = 0 then f else
    (taut(x - 1)(f true))
  andalso (taut(x - 1)(f false))
A tautology checker

fun taut x f = if x = 0 then f else
            (taut(x - 1)(f true))
            andalso (taut(x - 1)(f false))

Defining types $n \text{AryBoolOp}$ for each natural number $n \in \mathbb{N}$

\[
\begin{align*}
0 \text{AryBoolOp} & \triangleq \text{bool} \\
(n + 1) \text{AryBoolOp} & \triangleq \text{bool} \rightarrow (n \text{AryBoolOp})
\end{align*}
\]

Eg. 3 \text{AryBoolOp} = \text{bool} \rightarrow (\text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}))

3 arguments
A tautology checker

\[
\text{fun } \text{taut } x \ f = \ \text{if } x = 0 \ \text{then } f \ \text{else} \\
(taut(x - 1)(f \ \text{true})) \\
\text{andalso } (taut(x - 1)(f \ \text{false}))
\]

Defining types \(n \text{AryBoolOp}\) for each natural number \(n \in \mathbb{N}\)

\[
\begin{align*}
0 \text{AryBoolOp} & \triangleq \text{bool} \\
(n + 1) \text{AryBoolOp} & \triangleq \text{bool} \rightarrow (n \text{AryBoolOp})
\end{align*}
\]

then \(\text{taut } n\) has type \((n \text{AryBoolOp}) \rightarrow \text{bool}\), i.e. the result type of the function \(\text{taut}\) depends upon the value of its argument.
The tautology checker in Agda

data Bool : Set where
  true : Bool
  false : Bool

_and_ : Bool -> Bool -> Bool
true and true  = true
true and false = false
false and _   = false

data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

_AryBool0p : Nat -> Set
zero   AryBool0p = Bool
(succ x) AryBool0p = Bool -> x AryBool0p

taut : (x : Nat) -> x AryBool0p -> Bool
  taut zero        f = f
  taut (succ x) f = taut x (f true) and taut x (f false)
The tautology checker in Agda

data Bool : Set where
  true : Bool
  false : Bool

_and_ : Bool -> Bool -> Bool
true  and true  = true
true  and false = false
false and _    = false

data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

_AryBoolOp : Nat -> Set
zero    AryBoolOp = Bool
(succ x) AryBoolOp = Bool -> x AryBoolOp

taut : (x : Nat) -> x AryBoolOp -> Bool
  taut zero     f = f
  taut (succ x) f = taut x (f true) and taut x (f false)
Dependent function types $\Pi x : \tau (\tau')$

$\tau'$ may ‘depend’ on $x$, i.e. have free occurrences of $x$.

(Free occurrences of $x$ in $\tau'$ are bound in $\Pi x : \tau (\tau')$.)
Dependent function types $\Pi x : \tau \ (\tau')$

$$
\frac{\Gamma, x : \tau \vdash M : \tau'}{
\Gamma \vdash \lambda x : \tau \ (M) : \Pi x : \tau \ (\tau')}
\quad \text{if } x \notin \text{dom}(\Gamma)
$$
Dependent function types \( \Pi x : \tau (\tau') \)

\[
\Gamma, x : \tau \vdash M : \tau' \\
\Gamma \vdash \lambda x : \tau (M) : \Pi x : \tau (\tau') \\
\text{if } x \notin \text{dom}(\Gamma)
\]

\[
\Gamma \vdash M : \Pi x : \tau (\tau') \\
\Gamma \vdash M' : \tau \\
\Gamma \vdash MM' : \tau'[M'/x]
\]
Dependent type systems usually feature a rule of the form

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau'}$$

if $\tau \approx \tau'$

where $\tau \approx \tau'$ is some relation of *equality between types* (e.g. inductively defined in some way).

For example one would expect $(1 + 1)\text{AryBoolOp} \approx 2\text{AryBoolOp}$. 
Conversion typing rule

Dependent type systems usually feature a rule of the form

\[
\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau'} \text{ if } \tau \approx \tau'
\]

where \( \tau \approx \tau' \) is some relation of equality between types (e.g. inductively defined in some way).

For example one would expect \((1 + 1) \text{AryBoolOp} \approx 2 \text{AryBoolOp}\).

For decidability of type-checking, one needs \( \approx \) to be a decidable relation between type expressions.
Pure Type Systems (PTS) – syntax

In a PTS type expressions and term expressions are lumped together into a single syntactic category of *pseudo-terms*:

\[
t ::= \begin{array}{l}
  x \quad \text{variable} \\
  s \quad \text{sort} \\
  \Pi x : t (t) \quad \text{dependent function type} \\
  \lambda x : t (t) \quad \text{function abstraction} \\
  tt \quad \text{function application}
\end{array}
\]

where \( x \) ranges over a countably infinite set \( \text{Var} \) of variables and \( s \) ranges over a disjoint set \( \text{Sort} \) of *sort symbols* – constants that denote various universes (= types whose elements denote types of various sorts) \([\text{kind} \text{ is a commonly used synonym for sort}]. \) \( \lambda x : t (t') \) and \( \Pi x : t (t') \) both bind free occurrences of \( x \) in the pseudo-term \( t' \).

E.g., if \( S \) is a sort for types

\[
\lambda x : S (\lambda y : x (y)) \quad \text{is like PLC term} \quad \Lambda \alpha (\Lambda y : \alpha (y))
\]
Pure Type Systems (PTS) – syntax

In a PTS type expressions and term expressions are lumped together into a single syntactic category of *pseudo-terms*:

\[
\begin{align*}
t & ::= x & \text{variable} \\
    & \mid s & \text{sort} \\
    & \mid \Pi x : t (t) & \text{dependent function type} \\
    & \mid \lambda x : t (t) & \text{function abstraction} \\
    & \mid t t & \text{function application}
\end{align*}
\]

where \( x \) ranges over a countably infinite set \( \text{Var} \) of variables and \( s \) ranges over a disjoint set \( \text{Sort} \) of *sort symbols* – constants that denote various universes (= types whose elements denote types of various sorts) [kind is a commonly used synonym for sort]. \( \lambda x : t (t') \) and \( \Pi x : t (t') \) both bind free occurrences of \( x \) in the pseudo-term \( t' \).

\[t[t'/x]\] denotes result of capture-avoiding substitution of \( t' \) for all free occurrences of \( x \) in \( t \).
Pure Type Systems (PTS) – syntax

In a PTS type expressions and term expressions are lumped together into a single syntactic category of \textit{pseudo-terms}:

\[
t ::= x \quad \text{variable} \\
    | s \quad \text{sort} \\
    | \Pi x : t \ (t) \quad \text{dependent function type} \\
    | \lambda x : t \ (t) \quad \text{function abstraction} \\
    | t \ t \quad \text{function application}
\]

where \(x\) ranges over a countably infinite set \texttt{Var} of variables and \(s\) ranges over a disjoint set \texttt{Sort} of \textit{sort symbols} – constants that denote various universes (= types whose elements denote types of various sorts) [\textit{kind} is a commonly used synonym for \textit{sort}]. \(\lambda x : t \ (t')\) and \(\Pi x : t \ (t')\) both bind free occurrences of \(x\) in the pseudo-term \(t'\).

\[
t \to t' \triangleq \Pi x : t \ (t') \quad \text{where } x \notin \text{fv}(t').
\]
Pure Type Systems – beta-conversion

- **beta-reduction** of pseudo-terms: \( t \rightarrow t' \) means \( t' \) can be obtained from \( t \) (up to alpha-conversion, of course) by replacing a subexpression which is a *redex* by its corresponding *reduct*. There is only one form of redex-reduct pair:

\[
(\lambda x : t(t_1)) t_2 \rightarrow t_1[t_2/x]
\]

- As usual, \( \rightarrow^* \) denotes the reflexive-transitive closure of \( \rightarrow \).
Pure Type Systems – beta-conversion

- **beta-reduction** of pseudo-terms: $t \rightarrow t'$ means $t'$ can be obtained from $t$ (up to alpha-conversion, of course) by replacing a subexpression which is a *redex* by its corresponding *reduct*. There is only one form of redex-reduct pair:

  $$(\lambda x : t (t_1)) t_2 \rightarrow t_1[t_2/x]$$

- As usual, $\rightarrow^*$ denotes the reflexive-transitive closure of $\rightarrow$.
- **beta-conversion** of pseudo-terms: $=\beta$ is the reflexive-symmetric-transitive closure of $\rightarrow$ (i.e. the smallest equivalence relation containing $\rightarrow$).
Pure Type Systems – typing judgements

take the form

\[ \Gamma \vdash t : t' \]

where \( t, t' \) are pseudo-terms and \( \Gamma \) is a context, a form of typing environment given by the grammar

\[ \Gamma ::= \diamond \mid \Gamma, x : t \]

(Thus contexts are finite ordered lists of (variable,pseudo-term)-pairs, with the empty list denoted \( \diamond \), the head of the list on the right and list-cons denoted by \( _,_ \). Unlike previous type systems in this course, the order in which typing declarations \( x : t \) occur in a context is important.)

\[ \text{e.g. } \alpha : \text{Type}, F : \alpha \rightarrow \text{Type}, x : \alpha \vdash F x : \text{Type} \]

(\( \alpha, F, x \) variables; Type a sort symbol)
Pure Type Systems – specifications

The typing rules for a particular PTS are parameterised by a specification $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where:

- $\mathcal{S} \subseteq \text{Sort}$
  Elements $s \in \mathcal{S}$ denote the different universes of types in this PTS.

- $\mathcal{A} \subseteq \text{Sort} \times \text{Sort}$
  Elements $(s_1, s_2) \in \mathcal{A}$ are called axioms. They determine the typing relation between universes in this PTS.

- $\mathcal{R} \subseteq \text{Sort} \times \text{Sort} \times \text{Sort}$
  Elements $(s_1, s_2, s_3) \in \mathcal{R}$ are called rules. They determine which kinds of dependent function can be formed and in which universes they live.
The typing rules for a particular PTS are parameterised by a **specification** \( S = (S, A, R) \) where:

- \( S \subseteq \text{Sort} \)
  - Elements \( s \in S \) denote the different universes of types in this PTS.

- \( A \subseteq \text{Sort} \times \text{Sort} \)
  - Elements \( (s_1, s_2) \in A \) are called *axioms*. They determine the typing relation between universes in this PTS.

- \( R \subseteq \text{Sort} \times \text{Sort} \times \text{Sort} \)
  - Elements \( (s_1, s_2, s_3) \in R \) are called rules. They determine which kinds of dependent function can be formed and in which universes they live.

The PTS with specification \( S \) will be denoted \( \lambda S \).
Pure Type Systems – typing judgements

take the form

\[ \Gamma \vdash t : t' \]

where \( t, t' \) are pseudo-terms and \( \Gamma \) is a \textit{context}, a form of typing environment given by the grammar

\[ \Gamma ::= \diamond \mid \Gamma, x : t \]

(Thus contexts are finite ordered lists of (variable,pseudo-term)-pairs, with the empty list denoted \( \diamond \), the head of the list on the right and list-cons denoted by \( _\leftarrow \), \( _\rightarrow \). Unlike previous type systems in this course, \textit{the order in which typing declarations} \( x : t \) \textit{occur in a context is important}.)

A typing judgement is \textit{derivable} if it is in the set inductively generated by the rules on the next slide, which are parameterised by a given specification \( S = (\mathcal{S}, \mathcal{A}, \mathcal{R}) \).
Pure Type Systems – typing rules

(axiom) \[ \Diamond \vdash s_1 : s_2 \] if \((s_1, s_2) \in \mathcal{A}\)
Pure Type Systems – typing rules

(axiom) \( \vdash s_1 : s_2 \) if \((s_1, s_2) \in \mathcal{A}\)

(start) \( \Gamma \vdash A : s \)
\( \Gamma, x : A \vdash x : A \) if \( x \notin \text{dom}(\Gamma) \)

(weaken) \( \Gamma \vdash M : A \quad \Gamma \vdash B : s \)
\( \Gamma, x : B \vdash M : A \) if \( x \notin \text{dom}(\Gamma) \)
Pure Type Systems – typing rules

(axiom) \[ \diamond \vdash s_1 : s_2 \quad \text{if} \quad (s_1, s_2) \in \mathcal{A} \]

(start) \[ \Gamma \vdash A : s \quad \text{if} \quad x \notin \text{dom}(\Gamma) \]

(weaken) \[ \Gamma \vdash M : A \quad \Gamma \vdash B : s \quad \text{if} \quad x \notin \text{dom}(\Gamma) \]

(conv) \[ \Gamma \vdash M : A \quad \Gamma \vdash B : s \quad \text{if} \quad A =_{\beta} B \]
Pure Type Systems – typing rules

(axiom) \[ \Diamond \vdash s_1 : s_2 \] if \((s_1, s_2) \in \mathcal{A}\)

(start) \[ \Gamma \vdash A : s \quad \Gamma, x : A \vdash x : A \] if \(x \notin \text{dom}(\Gamma)\)

(weaken) \[ \Gamma \vdash M : A \quad \Gamma \vdash B : s \quad \Gamma, x : B \vdash M : A \] if \(x \notin \text{dom}(\Gamma)\)

(conv) \[ \Gamma \vdash M : A \quad \Gamma \vdash B : s \quad \Gamma \vdash M : B \] if \(A =_{\beta} B\)

(prod) \[ \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \Gamma \vdash \Pi x : A (B) : s_3 \] if \((s_1, s_2, s_3) \in \mathcal{R}\)
Pure Type Systems – typing rules

(axiom) \( \diamond \vdash s_1 : s_2 \) if \((s_1, s_2) \in \mathcal{A}\)

(start) \( \Gamma \vdash A : s \) if \( x \notin \text{dom}(\Gamma) \)

(weaken) \( \Gamma \vdash M : A \quad \Gamma \vdash B : s \) if \( x \notin \text{dom}(\Gamma) \)

(conv) \( \Gamma \vdash M : A \quad \Gamma \vdash B : s \) if \( A =_\beta B \)

(prod) \( \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \) if \((s_1, s_2, s_3) \in \mathcal{R}\)

(abs) \( \Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A (B) : s \) \( \Gamma \vdash \lambda x : A (M) : \Pi x : A (B) \)

(app) \( \Gamma \vdash M : \Pi x : A (B) \quad \Gamma \vdash N : A \) \( \Gamma \vdash MN : B[N/x] \)

\((A, B, M, N\text{ range over pseudoterms, } s, s_1, s_2, s_3\text{ over sort symbols})\)
Example PTS typing derivations

\[(\text{axiom}) \quad \vdash * : \Box \quad (\text{axiom}) \quad \vdash * : \Box \quad (\text{axiom}) \quad \vdash * : \Box \]

\[\vdash * \to * : \Box\]

\[\vdash * \to * : \Box\]

\[(\text{axiom}) \quad \vdash * : \Box \quad (\text{axiom}) \quad \vdash * : \Box \]

\[\vdash * \to * : \Box\]

\[\vdash \lambda x : * (x) : * \to *\]

Here we assume that the PTS specification \(S = (S, A, R)\) has \(* \in S\), \(\Box \in S\), \((*, \Box) \in A\) and \((\Box, \Box, \Box) \in R\).

(Recall that \(* \to * \equiv \Pi x : * (\ast)\).)