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

[§5, p. 53 et seq.]
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).

If \( \Lambda \alpha (M) : \forall \alpha (\tau') \), then for each argument \( \tau \), the type of \( M[\tau/\alpha] \) is \( \tau' \), it depends on the argument \( \tau \).

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

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

The set of all $b$ that are in $B_a$ for some $a \in A$. 

---
Dependent Functions

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

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

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 $B_a$. 
Dependent Functions

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

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

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 $F_a$).

For example if $A = \mathbb{N}$ and for each $n \in \mathbb{N}$, $B_n = \{0, 1\}^n \rightarrow \{0, 1\}$, then $\prod_{n \in \mathbb{N}} B_n$ consists of functions mapping each number $n$ to an $n$-ary Boolean operation.
A tautology checker

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

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

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

\[
\begin{cases}
0 \text{ AryBoolOp} & \triangleq \text{bool} \\
(n + 1) \text{ AryBoolOp} & \triangleq \text{bool} \to (n \text{ AryBoolOp})
\end{cases}
\]

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

\text{3 arguments}
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 AryBoolOp` for each natural number `n ∈ \mathbb{N}`

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

then `taut n` has type `(n \text{AryBoolOp}) \rightarrow \text{bool}`, i.e. the result type of the function `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 $\Pi x : \tau (\tau')$

(written in Agda as

$$(x : \tau) \to \tau'$$

$\tau'$ 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 $\Pi x : \tau (\tau')$

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

\[
\frac{
\Gamma \vdash M : \Pi x : \tau (\tau') \quad \Gamma \vdash M' : \tau
}{
\Gamma \vdash MM' : \tau'[M'/x]
} 
\]
Conversion typing rule

Dependent type systems usually feature a rule of the form

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

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

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

Dependent type systems usually feature a rule of the form

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

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:

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

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) \([\text{kind}]\) 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

\[
\lambda x : s (\lambda y : x(y))
\]

is like PLC term \(\Lambda \alpha (\Lambda y : \alpha(y))\)
Pure Type Systems (PTS) – syntax

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

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

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) \([\text{kind}] \) 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 \( t' \).

\[\text{Binders} : \quad \Pi x : t (-) \]
\[\lambda x : t (-) \]
Pure Type Systems (PTS) – syntax

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

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

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}:

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

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

\[
    t \rightarrow t' \triangleq \Pi x : t (t') \quad \text{where} \ x \notin \text{fv}(t').
\]

\textbf{simply-typed functions are a special case of dependently-typed functions}
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: $\equiv_\beta$ 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.)

eg. $$\diamond, x : s, f : x \to s \vdash fx : s$$

($s$ a sort, $x \& f$ variables)