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

Dependently typed Functions on types

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

if $\Lambda\alpha\,(M) : \forall\alpha\,(\tau')$,
then for each argument τ,
the type of $M[\tau/\alpha]$ is $\tau'[\tau/\alpha]$,
– it <u>depends</u> on the argument τ

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 $F\,a$).

Dependent Functions

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

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

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 $F\,a$).

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.

A tautology checker

$$\begin{aligned}
\texttt{fun}\ \mathit{taut}\ x\ f\ = \ & \texttt{if}\ x = 0\ \texttt{then}\ f\ \texttt{else} \\
& (\mathit{taut}(x-1)(f\ \texttt{true})) \\
& \texttt{andalso}\ (\mathit{taut}(x-1)(f\ \texttt{false}))
\end{aligned}$$

A tautology checker

$$\begin{aligned}
&\texttt{fun}\ \boldsymbol{taut}\ x\ f\ =\ \texttt{if}\ x = \mathbf{0}\ \texttt{then}\ f\ \texttt{else} \\
&\qquad\qquad (\boldsymbol{taut}(x-\mathbf{1})(f\ \texttt{true})) \\
&\qquad\qquad \texttt{andalso}\ (\boldsymbol{taut}(x-\mathbf{1})(f\ \texttt{false}))
\end{aligned}$$

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

$$\begin{cases}
\mathbf{0}\,\mathit{AryBoolOp} & \triangleq \mathit{bool} \\
(n+\mathbf{1})\,\mathit{AryBoolOp} & \triangleq \mathit{bool} \to (n\,\mathit{AryBoolOp})
\end{cases}$$

Eg. $3\ \mathit{AryBoolOp} = \underbrace{bool \to (bool \to (bool}_{3\ \text{arguments}} \to bool))$

A tautology checker

$$\begin{aligned}\texttt{fun}\ \mathit{taut}\ x\ f\ =\ & \texttt{if}\ x = 0\ \texttt{then}\ f\ \texttt{else}\\ & (\mathit{taut}(x-1)(f\ \texttt{true}))\\ & \texttt{andalso}\ (\mathit{taut}(x-1)(f\ \texttt{false}))\end{aligned}$$

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

$$\begin{cases} 0\,\mathit{AryBoolOp} & \triangleq \mathit{bool} \\ (n+1)\,\mathit{AryBoolOp} & \triangleq \mathit{bool} \to (n\,\mathit{AryBoolOp}) \end{cases}$$

then $\mathit{taut}\,n$ has type $(n\,\mathit{AryBoolOp}) \to \mathit{bool}$, 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

_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)
```

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)
```

a simply typed function

a dependently typed function

Dependent function types $\Pi x : \tau \, (\tau')$

τ' may 'depend' on x, i.e. have free occurrences of x.

(Free occurrences of x in τ' are bound in $\Pi x : \tau \, (\tau')$.)

Dependent function types $\Pi x : \tau \, (\tau')$

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

Dependent function types $\Pi x : \tau\,(\tau')$

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

$$\frac{\Gamma \vdash M : \Pi x : \tau\,(\tau') \qquad \Gamma \vdash M' : \tau}{\Gamma \vdash M\,M' : \tau'[M'/x]}$$

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 $(1 + 1)\, \texttt{AryBoolOp} \approx 2\, \texttt{AryBoolOp}$.

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 $(1+1)\ \texttt{AryBoolOp} \approx 2\ \texttt{AryBoolOp}$.

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*:

$$
\begin{array}{rcll}
t & ::= & x & \text{variable} \\
  & \mid & s & \text{sort} \\
  & \mid & \Pi x : t\,(t) & \text{dependent function type} \\
  & \mid & \lambda x : t\,(t) & \text{function abstraction} \\
  & \mid & t\,t & \text{function application}
\end{array}
$$

where x ranges over a countably infinite set **Var** of variables and s ranges over a disjoint set **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'.

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*:

$$
\begin{array}{rcll}
t & ::= & x & \text{variable} \\
& | & s & \text{sort} \\
& | & \Pi x : t\,(t) & \text{dependent function type} \\
& | & \lambda x : t\,(t) & \text{function abstraction} \\
& | & t\,t & \text{function application}
\end{array}
$$

where x ranges over a countably infinite set $\mathbf{Var}$ of variables and s ranges over a disjoint set $\mathbf{Sort}$ 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'.

$\boxed{t[t'/x]}$ 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 *pseudo-terms*:

$$
\begin{array}{lcll}
t & ::= & x & \text{variable} \\
 & | & s & \text{sort} \\
 & | & \Pi x : t\,(t) & \text{dependent function type} \\
 & | & \lambda x : t\,(t) & \text{function abstraction} \\
 & | & t\,t & \text{function application}
\end{array}
$$

where x ranges over a countably infinite set $\mathbf{Var}$ of variables and s ranges over a disjoint set $\mathbf{Sort}$ 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'.

simply-typed functions are a special case of dependently-typed functions

$\boxed{t \to t'} \triangleq \Pi x : t\,(t')$ where $x \notin fv(t')$.

Pure Type Systems – beta-conversion

- *beta-reduction* of pseudo-terms: $\boxed{t \longrightarrow t'}$ 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:

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

- As usual, $\longrightarrow^*$ denotes the reflexive-transitive closure of $\longrightarrow$.

Pure Type Systems – beta-conversion

- *beta-reduction* of pseudo-terms: $\boxed{t \longrightarrow t'}$ 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:

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

- As usual, $\longrightarrow^*$ denotes the reflexive-transitive closure of $\longrightarrow$.
- *beta-conversion* of pseudo-terms: $=_\beta$ is the reflexive-symmetric-transitive closure of $\longrightarrow$ (i.e. the smallest equivalence relation containing $\longrightarrow$).

Pure Type Systems – typing judgements

take the form

$$\boxed{\Gamma \vdash t : t'}$$

where t, t' are pseudo-terms and Γ 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. $\alpha : Type, F : \alpha \to Type, x : \alpha \vdash F\,x : Type$

(α, F, x variables; Type a sort symbol)

Pure Type Systems – specifications

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

- $\mathcal{S} \subseteq \mathbf{Sort}$

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

- $\mathcal{A} \subseteq \mathbf{Sort} \times \mathbf{Sort}$

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

- $\mathcal{R} \subseteq \mathbf{Sort} \times \mathbf{Sort} \times \mathbf{Sort}$

  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.

Pure Type Systems – specifications

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

- $\mathcal{S} \subseteq \mathbf{Sort}$

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

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

  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 PTS with specification $\mathbf{S}$ will be denoted $\boxed{\lambda\mathbf{S}}$.

Pure Type Systems – typing judgements

take the form

$$\boxed{\Gamma \vdash t : t'}$$

where t, t' are pseudo-terms and Γ 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*.)

A typing judgement is *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

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

Pure Type Systems – typing rules

$$(\textbf{axiom})\ \frac{}{\diamond \vdash s_1 : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{A}$$

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

$$(\textbf{weaken})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A} \quad \text{if } x \notin dom(\Gamma)$$

Pure Type Systems – typing rules

$$\textbf{(axiom)}\ \frac{}{\diamond \vdash s_1 : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{A}$$

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

$$\textbf{(weaken)}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A} \quad \text{if } x \notin dom(\Gamma)$$

$$\textbf{(conv)}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad \text{if } A =_\beta B$$

Pure Type Systems – typing rules

$$(\textbf{axiom})\ \frac{}{\diamond \vdash s_1 : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{A}$$

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

$$(\textbf{weaken})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A} \quad \text{if } x \notin dom(\Gamma)$$

$$(\textbf{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad \text{if } A =_\beta B$$

$$(\textbf{prod})\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A\ (B) : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

Pure Type Systems – typing rules

$$(\textbf{axiom})\ \frac{}{\diamond \vdash s_1 : s_2} \quad \text{if } (s_1, s_2) \in \mathcal{A}$$

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

$$(\textbf{weaken})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A} \quad \text{if } x \notin dom(\Gamma)$$

$$(\textbf{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad \text{if } A =_\beta B$$

$$(\textbf{prod})\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A\,(B) : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\textbf{abs})\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A\,(B) : s}{\Gamma \vdash \lambda x : A\,(M) : \Pi x : A\,(B)}$$

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

(A, B, M, N range over pseudoterms, s, s_1, s_2, s_3 over sort symbols)

Example PTS typing derivations

$$(\textbf{prod})\,\dfrac{(\textbf{axiom})\,\dfrac{}{\diamond \vdash * : \Box} \qquad (\textbf{weaken})\,\dfrac{(\textbf{axiom})\,\dfrac{}{\diamond \vdash * : \Box} \qquad (\textbf{axiom})\,\dfrac{}{\diamond \vdash * : \Box}}{\diamond, x : * \vdash * : \Box}}{\diamond \vdash * \to * : \Box}$$

$$(\textbf{abs})\,\dfrac{(\textbf{start})\,\dfrac{(\textbf{axiom})\,\dfrac{}{\diamond \vdash * : \Box}}{\diamond, x : * \vdash x : *} \qquad \dfrac{\vdots}{\diamond \vdash * \to * : \Box}}{\diamond \vdash \lambda x : * \,(x) : * \to *}$$

Here we assume that the PTS specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ has $* \in \mathcal{S}$, $\Box \in \mathcal{S}$, $(*, \Box) \in \mathcal{A}$ and $(\Box, \Box, \Box) \in \mathcal{R}$.
(Recall that $* \to * \triangleq \Pi x : * \,(*)$.)