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, $\boxed{\Lambda\alpha\,(M)}$ 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, $\Lambda\alpha\,(M)$ 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 depends 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

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

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

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

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

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

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

A tautology checker

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

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

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

Eg. 3 AryBoolOp = bool → (bool → (bool → bool))

3 arguments

A tautology checker

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

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

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

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

(written in Agda as $(x : \tau) \to \tau'$)

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

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*:

t	$::=$	x	variable
	$\mid$	s	sort
	$\mid$	$\Pi x : t\,(t)$	dependent function type
	$\mid$	$\lambda x : t\,(t)$	function abstraction
	$\mid$	$t\,t$	function application

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'.

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*:

t	$::=$	x	variable
	$\mid$	s	sort
	$\mid$	$\Pi x : t\,(t)$	dependent function type
	$\mid$	$\lambda x : t\,(t)$	function abstraction
	$\mid$	$t\,t$	function application

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'.

Binders: $\Pi x : t\,(-)$

$\lambda x : t\,(-)$

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	variable
	$\mid$	s	sort
	$\mid$	$\Pi x : t\,(t)$	dependent function type
	$\mid$	$\lambda x : t\,(t)$	function abstraction
	$\mid$	$t\,t$	function application

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*:

t	$::=$	x	variable
	$\mid$	s	sort
	$\mid$	$\Pi x : t\,(t)$	dependent function type
	$\mid$	$\lambda x : t\,(t)$	function abstraction
	$\mid$	$t\,t$	function application

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

$t \to t' \triangleq \Pi x : t\,(t')$ 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*.)

e.g. $\diamond, x : s, f : x \to s \vdash f\,x : s$

(s a sort, x & f variables)