# Programs

Programs

$$M, A \;::=\; v_k \mid M\ M \mid \lambda M \mid \Pi\ A\ A \mid M\ D \mid c\ \vec{M} \mid B \mid L$$

Definitions, Branches and Labelled Sums

$$D \;::=\; [\vec{M} : \vec{A}] \qquad B \;::=\; c_1\ M_1, \ldots, c_k\ M_k \qquad L \;::=\; c_1\ \vec{A}_1, \ldots, c_k\ \vec{A}_k$$

# Environments and Values

Environments and Values

$$\rho, \sigma \ ::= \ () \mid \rho, u \mid D\rho \qquad u, V \ ::= \ M\rho \mid u\ u \mid X_l \mid \Pi\ V\ V$$

Access rules

$$v_0(\sigma, u) = u \quad v_{k+1}(\sigma, u) = v_k\sigma$$

and if $\rho = [\vec{M} : \vec{A}]\sigma$ then

$$v_i\rho = v_i(\sigma, \vec{M}\rho)$$

# Evaluation rules

$$(M_1\ M_2)\rho = M_1\rho\ (M_2\rho) \qquad (M\ D)\rho = M(D\rho)$$

$$(\Pi\ A\ F)\rho = \Pi\ (A\rho)\ (F\rho) \qquad (c\ \vec{M})\rho = c\ (\vec{M}\rho)$$

$$(\lambda\ M)\rho\ u = M(\rho, u) \qquad (c_1\ N_1, \ldots, c_k\ N_k)\rho\ (c_i\ \vec{u}) = N_i(\rho, \vec{u})$$

# Programs, version with names

Programs

$$M, A \;::=\; x \mid M\ M \mid \lambda x.M \mid \Pi\ x : A, A \mid M\ D \mid c\ \vec{M} \mid B \mid L$$

$$T \;::=\; () \mid T' \qquad T' \;::=\; A \mid (x : A, T')$$

Definitions, Branches and Labelled Sums

$$D \;::=\; [\vec{x} : T = \vec{M}] \qquad B \;::=\; c_1\ \vec{x_1} \to M_1, \ldots, c_k\ \vec{x_k} \to M_k$$

$$L \;::=\; c_1\ T_1, \ldots, c_k\ T_k$$

# Conversion test

Each branch $B$ has a name $f_B$ and each labelled sum $L$ a name $d_L$ associated to it. We test $A_1 = A_2$ by comparing $R_k\ A_1$ and $R_k\ A_2$

$$R_k\ X_l = v_{k-l-1}$$

$$R_k\ ((\lambda M)\rho) = \lambda R_{k+1}(M(\rho, X_k)) \qquad R_k\ (u_1\ u_2) = R_k\ u_1\ (R_k\ u_2)$$

$$R_k\ (\Pi\ V\ \ F) = \Pi\ (R_k\ V)\ (R_k\ F) \qquad R_k\ (c\ \vec{u}) = c\ (R_k\ \vec{u})$$

$$R_k\ (B\rho) = f_B(R_k\ \rho) \qquad R_k\ (L\rho) = d_L(R_k\ \rho)$$

$$R_k\ () = () \qquad R_k\ (\rho, u) = (R_k\ \rho, R_k\ u) \qquad R_k\ (D\rho) = R_k\ \rho$$

# Conversion test

Here is the grammar for the normal forms produced by the readback function $R_k$

$$t ::= \lambda\ t \mid d_L(t, \ldots, t) \mid \Pi\ t\ t \mid f_B(t, \ldots, t) \mid c(t, \ldots, t) \mid n$$
$$n ::= v_l \mid n\ t \mid f_B(t, \ldots, t)\ n$$

# Type-checking

The judgements are of the form $\rho, \Gamma \vdash_k A$ and $\rho, \Gamma \vdash_k M : V$ where $\Gamma$ is a list of type values and $k$ the number of free variables. For instance

$$\frac{}{\rho, \Gamma \vdash_k v_n : \Gamma ! n}$$

$$\frac{\rho, \Gamma \vdash_k N : \Pi\ V\ F \qquad \rho, \Gamma \vdash_k M : V}{\rho, \Gamma \vdash_k N\ M : F\ (M \rho)}$$

$$\frac{(\rho, X_k), (\Gamma, V) \vdash_{k+1} N : F\ X_k}{\rho, \Gamma \vdash_k \lambda N : \Pi\ V\ F}$$

# Examples

```
Nat : Set = 0 | S Nat

Bin : Set = 1 | S0 Bin | S1 Bin

natrec : (P : Nat -> Set) ->
         P 0 -> ((i : Nat) -> P i -> P (S i)) ->
         (n : Nat) -> P n =
 \ P -> \ p0 -> \ pS ->
 [ 0 -> p0
  |S x -> pS x (natrec P p0 pS x) ]
```

# Examples

```
mutual
BSTree : Set =
   slf | snd (a : A) (l r : BSTree) (>=T a l) (<=T a r)

>=T : A -> BSTree -> Set =
 \ a -> [ slf -> True
         |snd x l r _ _ -> (a <= x) & (>=T a r) ]

<=T : A -> BSTree -> Set =
 \ a -> [ slf -> True
         |snd x l r _ _ -> (x <= a) & (<=T a l) ]
```

# Denotational Semantics

Formal neighbourhoods

$$W \ ::= \ \nabla \mid W \to W \mid W \cap W \mid c\ \vec{W} \mid [c_1\ \vec{U_1}, \ldots, c_n\ \vec{U_n}]$$

$$U \ ::= \ \Delta \mid W$$

# Denotational Semantics

$$\frac{\Gamma, \vec{U}^{(0)} \vdash \vec{M} : \vec{U}^{(1)} \quad \ldots \quad \Gamma, \vec{U}^{(l-1)} \vdash \vec{M} : \vec{U}^{(l)} \qquad \Gamma, \vec{U}^{(l)} \vdash N : V}{\Gamma \vdash ND : V}$$

where $D$ is $[\vec{M} : \vec{A}]$ and $\vec{U}^{(0)}$ is $\vec{\Delta}$.

# Denotational Semantics

The elements of the domain D are either constructor terms $c\ \vec{u}$ or product $\Pi\ u\ f$ or labelled sums $[c_1\ \vec{a_1}, \ldots, c_n\ \vec{a_n}]$ or functions $f$

**Theorem:** *If the semantics of a term $M$ is $\neq\perp$ then $M$ is SN*

# Models

A *totality* on D is a subset $X \subseteq$ D such that $\bot \notin X$ and $\top \in X$. We write TP(D) the set of all totality on D.

A partial interpretation of type theory is a pair $(X, El)$ with $X$ in TP(D) and $El$ in $X \rightarrow$ TP(D) such that $El(\top)$ is the singleton $\{\top\}$

We extend $X$ and $El$ to vectors: $()$ in $X$ and $() \in El()$ and $(a, \vec{a})$ in $X$ iff $a \in X$ and $\vec{a}\ u$ in $X$ for all $u \in El(a)$. Then $(u, \vec{u})$ in $El(a, \vec{a})$ iff $u \in El(a)$ and $\vec{u}$ in $El(\vec{a}\ u)$

# Models

$(X, El)$ total interpretation: $b$ in $X$ iff

$b = \Pi\ a\ f$ and $a \in X$ and $f\ u \in X$ for all $u \in El(a)$ and $w \in El(b)$ iff $w\ u \in El(f\ u)$ for all $u \in El(a)$

or $b = [c_1\ \vec{a_1}, \ldots, c_n\ \vec{a_n}]$ and $\vec{a_i} \in X$ and $w \in El(b)$ iff $w = c_i\ \vec{u}$ with $\vec{u} \in El(\vec{a_i})$

# Peano induction for binary numbers

```
Bin : Set = 1 | S0 Bin | S1 Bin

bsuc : Bin -> Bin =
 [ 1 -> S0 1
  |S0 x -> S1 x
  |S1 x -> S0 (bsuc x) ]
```

# Peano induction for binary numbers

```
binPeano : (P : Bin -> Set) -> P 1 ->
           ((i : Bin) -> P i -> P (bsuc i)) ->
           (b : Bin) -> P b =        \ P -> \ p1 -> \ ps ->
 [ 1 -> p1
  |S0 x ->
   binPeano (\ b -> P (S0 b)) (ps 1 p1)
            (\ i h -> ps (S1 1) (ps (S0 i) (ps (S0 i) h))) x
  |S1 x ->
   binPeano (\ b -> P (S1 b)) (ps (S0 1) (ps 1 p1))
            (\ i h ->
              ps (S1 1) (ps (S0 (bsuc i)) (ps (S1 i) h))) x ]
```

# Peano induction for binary numbers

"So that's that, except that it's a bit tricky and a bit higher-order and, worst of all, quite expensive in the *size* of the inductions involved. If we're being scrupulous about universe levels, we have to be careful about quantifying over arbitrary $P : Bin \to Set_i$. To be allowed such a thing we need to use our structural induction principal at $Set_{i+1}$."

Induction principle in this version of type theory (with pattern-matching) works on an *arbitrary* type

Similar analysis in Lorenzen: induction principle is justified on an arbitrary formula

# Peano induction for binary numbers

```
Peano : Bin -> Set where
 p1 : Peano 1
 ps : {x : Bin} -> Peano x -> Peano (bsuc x)

peano : (b : Bin) -> Peano b

double : {b : Peano} -> Peano b -> Peano (S0 b)
```

# Other example

Lookup function on vectors

```
vec : (Nat -> Set) -> Nat -> Set
vec B 0 = One
vec B (S x) = (vec B x) * (B x)

get : (B : Nat -> Set) -> (n x : Nat) ->
      x < n -> vec B n -> B x
```