MLW

Henk Barendregt and Freek Wiedijk
assisted by Andrew Polonsky

Radboud University Nijmegen

March 26, 2012

inductive types

inductive types

= types consisting of closed terms built from 'constructors'

$$\begin{aligned} 0 &: \mathsf{nat} \\ \mathsf{suc} &: \mathsf{nat} \to \mathsf{nat} \end{aligned}$$

$$\mathsf{nat} = \{0,\ \mathsf{suc}\ 0,\ \mathsf{suc}\,(\mathsf{suc}\ 0),\ \mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\ 0)),\ \ldots\}$$

$$\begin{aligned} \mathsf{leaf} &: \mathsf{tree} \\ \mathsf{node} &: \mathsf{tree} \to \mathsf{tree} \to \mathsf{tree} \end{aligned}$$

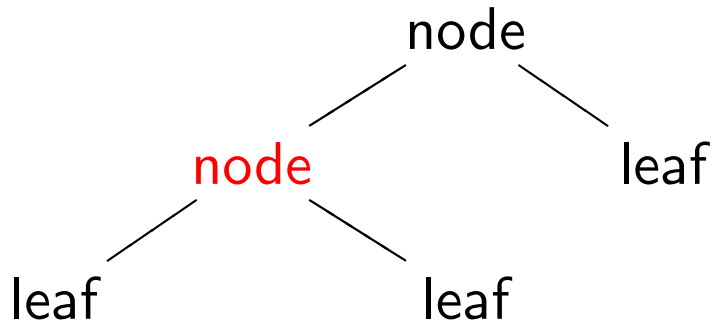
$$\mathsf{tree} = \{\mathsf{leaf},\ \mathsf{node}\ \mathsf{leaf}\ \mathsf{leaf},\ \mathsf{node}\,(\mathsf{node}\ \mathsf{leaf}\ \mathsf{leaf})\ \mathsf{leaf},\ \ldots\}$$

terms as data structures

the term:

node (node leaf leaf) leaf

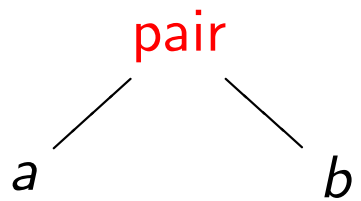
corresponds to the tree data structure:



constructors can have arguments

$$\text{pair} \quad : \quad A \to B \to \text{prod}\ A\ B$$

$$\text{prod}\ A\ B = \{\text{pair}\ a\ b \mid a \in A,\ b \in B\}$$



. . . and the disjoint union

$$\begin{aligned} \mathsf{inl} &: A \to \mathsf{sum}\ A\ B \\ \mathsf{inr} &: B \to \mathsf{sum}\ A\ B \end{aligned}$$

$$\mathsf{sum}\ A\ B = \{\mathsf{inl}\ a \mid a \in A\} \cup \{\mathsf{inr}\ b \mid b \in B\}$$

inductive types can be finite

$$\begin{array}{rcl} \mathsf{true} & : & \mathsf{bool} \\ \mathsf{false} & : & \mathsf{bool} \end{array}$$

$$\mathsf{bool} = \{\mathsf{true}, \mathsf{false}\}$$

$$\bullet \quad : \quad \mathsf{unit}$$

$$\mathsf{unit} = \{\bullet\}$$

$$\mathsf{empty} = \{\}$$

two kinds of true and false

$$\begin{aligned} \mathsf{true} &: \mathsf{bool} \\ \mathsf{false} &: \mathsf{bool} \end{aligned}$$

$$\mathsf{bool} = \{\mathsf{true}, \mathsf{false}\}$$

$$\bullet : \mathsf{True}$$

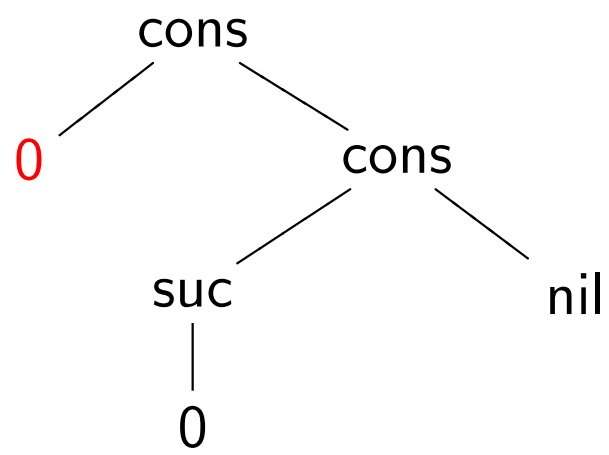
$$\mathsf{True} = \{\bullet\}$$

$$\mathsf{False} = \{\}$$

lists

$$\begin{aligned} \mathsf{nil} &: \mathsf{list} \\ \mathsf{cons} &: \mathsf{nat} \to \mathsf{list} \to \mathsf{list} \end{aligned}$$

$\langle 0, 1 \rangle =$



inductive predicates

= dependent inductive types

$$\begin{array}{lcl} \mathsf{le\_refl} & : & \Pi n : \mathsf{nat}.\, \mathsf{le}\ n\ n \\ \mathsf{le\_suc} & : & \Pi n : \mathsf{nat}.\, \Pi m : \mathsf{nat}.\, \mathsf{le}\ n\ m \longrightarrow \mathsf{le}\ n\ (\mathsf{suc}\ m) \end{array}$$

proof of $0 \leq 1$:

$$\mathsf{le\_suc}\ 0\ 0\ (\mathsf{le\_refl}\ 0) \quad : \quad \mathsf{le}\ 0\ (\mathsf{suc}\ 0)$$

Leibniz equality

$$\begin{aligned} \mathsf{eq} &: \Pi a : *.\, a \to a \to * \\ \mathsf{eq\_refl} &: \Pi a : *.\, \Pi x : a.\, \mathsf{eq}\ a\ x\ x \end{aligned}$$

$$\mathsf{eq}\ A\ M\ N = \begin{cases} \{\mathsf{eq\_refl}\ A\ M\ M\} & \text{if } M = N \\ \{\} & \text{if } M \neq N \end{cases}$$

$$\mathsf{eq}\ A\ M\ N \text{ inhabited} \iff M = N$$

defining inductive types

- *impredicative definitions*

  does not work well

  $0 \neq 1$ *not* provable
  induction principle *not* provable

- *hardwired into the foundations*

  $0 \neq 1$ *is* provable
  induction principle *is* provable

  inductive types are primitive like λ and Π

terms in type theory

syntax:

$$M \quad ::= \quad x \mid MM \mid \lambda x : M.\, M \mid \Pi x : M.\, M \mid s \mid c$$

sorts

constants

$$s \quad ::= \quad * \mid \Box \mid \ldots$$

$$c \quad ::= \quad \ldots \mid \mathsf{nat} \mid 0 \mid \mathsf{suc} \mid \mathsf{rec} \mid \ldots$$

constants

- *defined* constants
  abbreviating terms:

$$c := M$$

  δ-reduction $=$ replacing constant by its definition

$$c \to_\delta M$$

- constants related to *inductive types*
  - the type itself — nat
  - constructors — 0, suc
  - 'recursion principle' — rec
    $=$ eliminator
    $=$ destructor

  ι-reduction $=$ destructor applied to a constructor

$$\text{rec} \ldots 0 \ldots \to_\iota \ldots$$

two approaches

1. finite number of inductive types

   'hardwired' into the system

   Gödel's T
   MLW

2. possibility to define *arbitrary* inductive types

   constructors determine recursion principle and reduction

   pCIC $\rightsquigarrow$ Coq
   next time

inductive type: four kinds of rules

1. type formation
2. constructor types
3. eliminator type

   two possibilities:

   - non-dependent eliminator
   - dependent eliminator

4. reduction rules

   one for every constructor:

   eliminator acting on constructor

   ι-reduction: $\longrightarrow_\iota$

first three: typing rules
fourth: computation rules

what to do with a term?

- type check

  $\rightsquigarrow$ proof check

- compute

$$\longrightarrow_{\beta\delta\iota}$$

β-reduction	$=$	substitute in a lambda term
δ-reduction	$=$	unfold definition of a constant
ι-reduction	$=$	compute in an inductive type
η-reduction	$=$	dual of β-reduction
$\bar{\eta}$-expansion	$=$	converse of η-reduction

Gödel's system T

Gödel's system T

$= \text{`}\lambda\!\!\rightarrow + \text{nat'}$

syntax:

one base type

$$\begin{array}{rcl} & | & \\ A & ::= & \mathsf{nat} \mid A \to A \\ M & ::= & x \mid MM \mid \lambda x : A.\, M \mid 0 \mid \mathsf{suc} \mid \mathsf{rec}\ A \end{array}$$

typing rules

$$\frac{}{\Gamma \vdash x : A}\ x : A \in \Gamma$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}} \qquad \frac{}{\Gamma \vdash \mathsf{suc} : \mathsf{nat} \to \mathsf{nat}}$$

$$\frac{}{\Gamma \vdash \mathsf{rec}\ A : A \to (\mathsf{nat} \to A \to A) \to (\mathsf{nat} \to A)}$$

computation rules

$$(\lambda x : A.\, M)\, N \quad \to_\beta \quad M[x := N]$$

$$\begin{aligned} \text{rec}\ A\ M\ F\ 0 \quad &\to_\iota \quad M \\ \text{rec}\ A\ M\ F\ (\text{suc}\ N) \quad &\to_\iota \quad F\ N\ (\text{rec}\ A\ M\ F\ N) \end{aligned}$$

exercise: define plus and ...

recursive definition of plus:

$$\begin{aligned} \text{plus}\ x\ 0 &= x \\ \text{plus}\ x\ (\text{suc}\ y) &= \text{suc}\ (\text{plus}\ x\ y) \end{aligned}$$

$$\text{plus} := \lambda x : \text{nat}.\ \text{rec nat}\ x\ (\lambda y : \mathit{nat}.\ \lambda r : \mathit{nat}.\ \text{suc}\ r)$$

. . . and calculate $1 + 1$

$$\mathsf{plus} \quad := \quad \lambda x : \mathsf{nat}.\, \mathsf{rec}\ \mathsf{nat}\ x\ (\lambda y : \mathit{nat}.\, \lambda r : \mathit{nat}.\, \mathsf{suc}\ r)$$

$$\begin{aligned} \mathsf{rec}\ A\ M\ F\ 0 \quad &\to_\iota \quad M \\ \mathsf{rec}\ A\ M\ F\ (\mathsf{suc}\ N) \quad &\to_\iota \quad F\ N\ (\mathsf{rec}\ A\ M\ F\ N) \end{aligned}$$

$$\overbrace{\mathsf{plus}\ (\mathsf{suc}\ 0)\ (\mathsf{suc}\ 0)}^{1+1} \to_\delta$$
$$(\lambda x : \mathsf{nat}.\, \mathsf{rec}\ \mathsf{nat}\ x\ (\lambda y : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{suc}\ r))\ (\mathsf{suc}\ 0)\ (\mathsf{suc}\ 0) \to_\beta$$
$$\mathsf{rec}\ \mathsf{nat}\ (\mathsf{suc}\ 0)\ (\lambda y : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{suc}\ r)\ (\mathsf{suc}\ 0) \to_\iota$$
$$(\lambda y : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{suc}\ r)\ 0\ (\mathsf{rec}\ \mathsf{nat}\ (\mathsf{suc}\ 0)\ (\lambda y : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{suc}\ r)\ 0) \to_\beta$$
$$(\lambda r : \mathsf{nat}.\, \mathsf{suc}\ r)\ (\mathsf{rec}\ \mathsf{nat}\ (\mathsf{suc}\ 0)\ (\lambda y : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{suc}\ r)\ 0) \to_\beta$$
$$\mathsf{suc}\ \underbrace{(\mathsf{rec}\ \mathsf{nat}\ (\mathsf{suc}\ 0)\ (\lambda y : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{suc}\ r)\ 0)}_{1+0} \to_\iota \underbrace{\mathsf{suc}\ (\mathsf{suc}\ 0)}_{2}$$

rules for nat together

1. type formation:

$$\mathsf{nat} \quad : \quad *$$

2. constructor types:

$$\begin{array}{rcl} 0 & : & \mathsf{nat} \\ \mathsf{suc} & : & \mathsf{nat} \longrightarrow \mathsf{nat} \end{array}$$

3. eliminator type:

$$\begin{array}{rcl} \mathsf{rec} & : & \Pi a : *.\, a \longrightarrow (\mathsf{nat} \longrightarrow a \longrightarrow a) \longrightarrow (\mathsf{nat} \longrightarrow a) \\ \mathsf{rec} & : & \Pi a : (\mathsf{nat} \longrightarrow *).\, a\, 0 \longrightarrow \\ & & \quad (\Pi n : \mathsf{nat}.\, a\, n \longrightarrow a\, (\mathsf{suc}\, n)) \longrightarrow \Pi n : \mathit{nat}.\, a\, n \end{array}$$

4. reduction rules:

$$\begin{array}{rcl} \mathsf{rec}\, A\, M\, F\, 0 & \longrightarrow_\iota & M \\ \mathsf{rec}\, A\, M\, F\, (\mathsf{suc}\, N) & \longrightarrow_\iota & F\, N\, (\mathsf{rec}\, A\, M\, F\, N) \end{array}$$

how to determine the dependent eliminator type

$$\begin{aligned} \mathsf{rec} \quad : \quad & \Pi a : (\mathsf{nat} \to *). \\ & a\,0 \to \\ & (\Pi n : \mathsf{nat}.\, a\,n \to a\,(\mathsf{suc}\,n)) \to \\ & \Pi n : \mathsf{nat}.\, a\,n \end{aligned}$$

dependent recursion over the datatype

Curry-Howard: 'recursion principle' = 'induction principle'

- for all predicates a on the inductive type ...
- ... if this predicate is conserved by all constructors ...
- ... then it holds for all objects in the inductive type

how to determine the non-dependent eliminator type

$$\Pi a : (\mathsf{nat} \to *).\ a\,0 \to (\Pi n : \mathsf{nat}.\ a\,n \to a\,(\mathsf{suc}\,n)) \to \Pi n : \mathsf{nat}.\ a\,n$$

just erase the dependency on the inductive type!

$$\Pi a : *.\ a \to (\mathsf{nat} \to a \to a) \to (\mathsf{nat} \to a)$$

MLW

Martin-Löf type theory

- Bengt Nordström, Kent Petersson, Jan M. Smith,
  Programming in Martin-Löf Type Theory,
  1990

- Peter Aczel,
  On relating type theories and set theories,
  1998

  MLW defined on pp. 3–9

  here: just overview
  $\rightsquigarrow$ many rules!

  MLW still is *much* simpler than pCIC ... ☺

type constructions in MLW

$$\frac{\Pi x : A.\, B}{\left.\begin{array}{l} \Sigma x : A.\, B \\ W x : A.\, B \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \end{array}\right\} \text{ inductive types}}$$

not primitive in MLW, because definable

$$
\begin{aligned}
A \to B &:= \Pi x : A.\, B \\
A \times B &:= \Sigma x : A.\, B \\
A + B &:= \Sigma x : \mathbf{2}.\, R_2^*\, A\, B\, x \\
\mathsf{nat} &:= W x : \mathbf{2}.\, R_2^*\, \mathbf{0}\, \mathbf{1}\, x
\end{aligned}
$$

R_2^* is recursion over $\mathbf{2}$:

$$R_2^*\, A\, B\, x = \text{`if } x \text{ then } A \text{ else } B\text{'}$$

Curry-Howard

$\mathbf{0}$	empty	$\bot$
$\mathbf{1}$	unit	$\top$
$\mathbf{2}$	bool	
$A \to B$		$A \to B$
$A \times B$	prod A B	$A \wedge B$
$A + B$	sum A B	$A \vee B$
$\Pi x : A.\, B$		$\forall x : A.\, B$
$\Sigma x : A.\, B$	sig $(\lambda x : A.\, B)$	$\exists x : A.\, B$
$W x : A.\, B$	$W\ (\lambda x : A.\, B)$	

2

1. type formation:

$$\mathbf{2} \quad : \quad *$$

2. constructor types:

$$1 \quad : \quad \mathbf{2}$$
$$2 \quad : \quad \mathbf{2}$$

3. eliminator type:

$$R_2 \quad : \quad \Pi a : (\mathbf{2} \to *).\, a\,0 \to a\,1 \to \Pi x : \mathbf{2}.\, a\,x$$
$$R_2^* \quad : \quad * \to * \to (\mathbf{2} \to *)$$

4. reduction rules:

$$R_2\, A\, N_1\, N_2\, 1 \quad \to_\iota \quad N_1$$
$$R_2\, A\, N_1\, N_2\, 2 \quad \to_\iota \quad N_2$$
$$R_2^*\, A_1\, A_2\, 1 \quad \to_\iota \quad A_1$$
$$R_2^*\, A_1\, A_2\, 2 \quad \to_\iota \quad A_2$$

$\mathbf{2}$ is the Booleans

$$
\begin{array}{lcl}
\mathbf{2} & \rightsquigarrow & \text{bool} \\
1 & \rightsquigarrow & \text{true} \\
2 & \rightsquigarrow & \text{false} \\
R_2\ A\ N_1\ N_2\ M & \rightsquigarrow & \text{if } M \text{ then } N_1 \text{ else } N_2 \\
R_2^*\ A_1\ A_2\ M & \rightsquigarrow & \text{if } M \text{ then } A_1 \text{ else } A_2
\end{array}
$$

the three finite types

- $\mathbf{0}$
  $=$ empty $= \bot$

  no constructors

- $\mathbf{1}$
  $=$ unit $= \top$

  $\bullet \ : \ \mathbf{1}$

- $\mathbf{2}$
  $=$ bool

  $1 \ : \ \mathbf{2}$

  $2 \ : \ \mathbf{2}$

$A \times B$

1. type formation:

$$\text{prod} \quad : \quad * \longrightarrow * \longrightarrow *$$

2. constructor types:

$$\text{pair} \quad : \quad \Pi a : *. \, \Pi b : *. \, a \longrightarrow b \longrightarrow \text{prod}\ a\ b$$

3. eliminator type:

$$\pi_1 \quad : \quad \Pi a : *. \, \Pi b : *. \, \text{prod}\ a\ b \longrightarrow a$$
$$\pi_2 \quad : \quad \Pi a : *. \, \Pi b : *. \, \text{prod}\ a\ b \longrightarrow b$$

4. reduction rules:

$$\pi_1\ A\ B\ (\text{pair}\ A\ B\ M_1\ M_2) \quad \longrightarrow_\iota \quad M_1$$
$$\pi_2\ A\ B\ (\text{pair}\ A\ B\ M_1\ M_2) \quad \longrightarrow_\iota \quad M_2$$

$\Sigma x : A.\, B$

1. type formation:

$$\text{sig} \quad : \quad \Pi a : *.\, (a \to *) \to *$$

2. constructor types:

$$\text{pair} \quad : \quad \Pi a : *.\, \Pi b : (a \to *).\, \Pi x : a.\, b\, x \to \text{sig}\ a\ b$$

3. eliminator type:

$$\pi_1 \quad : \quad \Pi a : *.\, \Pi b : (a \to *).\, \text{sig}\ a\ b \to a$$
$$\pi_2 \quad : \quad \Pi a : *.\, \Pi b : (a \to *).\, \Pi x : \text{sig}\ a\ b.\, b\, (\pi_1\ a\ b\ x)$$

4. reduction rules:

$$\pi_1\ A\ B\ (\text{pair}\ A\ B\ M_1\ M_2) \quad \to_\iota \quad M_1$$
$$\pi_2\ A\ B\ (\text{pair}\ A\ B\ M_1\ M_2) \quad \to_\iota \quad M_2$$

W-trees

unlabeled binary trees

1. type formation:

$$\mathsf{tree} \;:\; *$$

2. constructor types:

$$\begin{aligned} \mathsf{leaf} &: \mathsf{tree} \\ \mathsf{node} &: \mathsf{tree} \to \mathsf{tree} \to \mathsf{tree} \end{aligned}$$

3. eliminator type:

$$\begin{aligned} \mathsf{binrec} \;:\; & \Pi a : (\mathsf{tree} \to *).\ a\ \mathsf{leaf} \to \\ & (\Pi t_1 : \mathsf{tree}.\, \Pi t_2 : \mathsf{tree}.\ a\ t_1 \to a\ t_2 \to a\ (\mathsf{node}\ t_1\ t_2)) \to \\ & \Pi t : \mathit{tree}.\ a\ t \end{aligned}$$

4. reduction rules:

$$\begin{aligned} \mathsf{binrec}\ A\ M\ F\ \mathsf{leaf} &\to_\iota M \\ \mathsf{binrec}\ A\ M\ F\ (\mathsf{node}\ N_1\ N_2) &\to_\iota F\ N_1\ N_2\ (\mathsf{binrec}\ A\ M\ F\ N_1)\ (\mathsf{binrec}\ A\ M\ F\ N_2) \end{aligned}$$

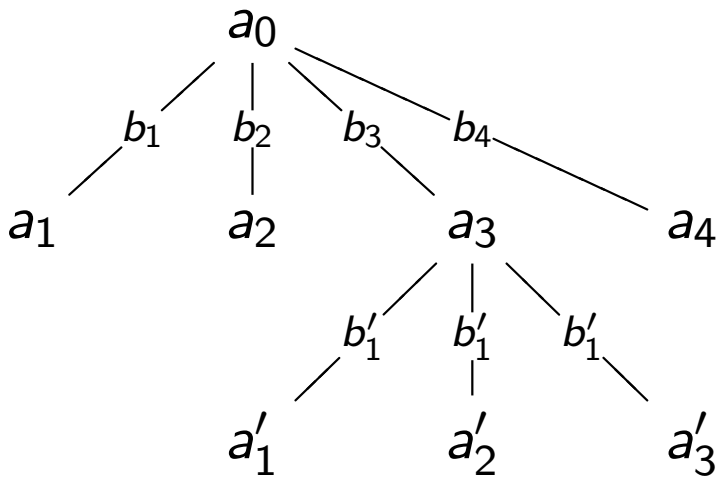
W-trees

$=$ containers

$Wx : A.\, B[x]$ type of labeled trees

A node labels

$B[a]$ edge labels from node labelled a

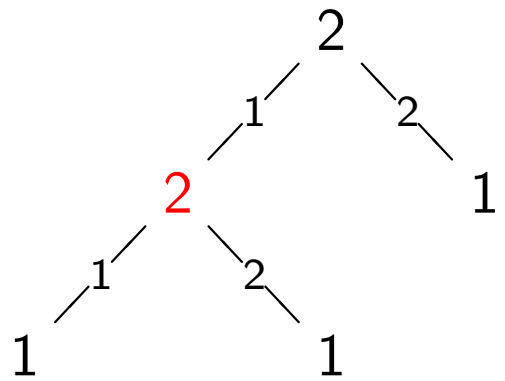


$$
\begin{array}{lcl}
a_0 & : & A \\
b_i & : & B[a_0] \\
a_i & : & A \\
b'_i & : & B[a_3] \\
a'_i & : & A
\end{array}
$$

examples of types defined as W-trees

two kinds of nodes

$$\text{tree} \quad := \quad Wx : \mathbf{2}.\, R_2^*\, \mathbf{0}\, \mathbf{2}\, x$$



$$\text{nat} \quad := \quad Wx : \mathbf{2}.\, R_2^*\, \mathbf{0}\, \mathbf{1}\, x$$

$Wx : A.\, B$

1. type formation:

$$W \quad : \quad \Pi a : *.\, (a \to *) \to *$$

2. constructor types:

$$\text{sup} \quad : \quad \Pi a : *.\, \Pi b : (a \to *).\, \Pi x : a.\, (b\, x \to W\, a\, b) \to W\, a\, b$$

3. eliminator type:

exercise!

4. reduction rules:

exercise!

variations on a theme

differences with the paper by Peter Aczel

- W-trees only work well in an extensional type theory

$$\mathsf{MLW} \rightsquigarrow \mathsf{MLW}^{\mathsf{ext}} \rightsquigarrow \mathsf{MLW}^{\mathsf{ext}}\mathsf{PU}_{<\omega}$$

- Martin-Löf type theory is different:
  - no $*$ and $\Box$ in judgments
  - *four* kind of judgments:

Martin-Löf	*PTS*
$\Gamma \vdash A$ type	$\Gamma \vdash A : *$
$\Gamma \vdash M : A$	$\Gamma \vdash M : A$
$\Gamma \vdash A_1 = A_2$	'$A_1 =_{\beta\iota} A_2$'
$\Gamma \vdash M_1 = M_2 : A$	'$M_1 =_{\beta\iota} M_2$'

function types are not inductive but . . .

1. type formation:

$$\Pi x : A.\, B \quad : \quad s$$

2. constructor types:

$$\lambda x : A.\, M \quad : \quad \Pi x : A.\, B$$

3. eliminator type:

$$MN \quad : \quad B[x := N]$$

4. reduction rules:

$$(\lambda x : A.\, M)\, N \quad \rightarrow_\beta \quad M[x := N]$$

recap

1. inductive types
2. Gödel's system T
3. MLW
4. W-trees
5. variations on a theme