Week 8. Inductive types

In-class problems

1. Let be given the inductive type of natural numbers $\mathsf{nat} : *$ with constructors $\mathsf{0} : \mathsf{nat}$ and $\mathsf{suc} : \mathsf{nat} \to \mathsf{nat}$. Using its recursor, define the function $\mathsf{sub} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ of *truncated* subtraction (i.e., if the difference would be negative, then the function has value zero.)

   The type of the (non-dependent) recursor is:

   $$\mathsf{rec} \;:\; \forall A : *.\, A \to (\mathsf{nat} \to A \to A) \to (\mathsf{nat} \to A)$$

   We want to use this to get the recursive equations:

   $$\begin{aligned} \mathsf{pred}\ 0 &= 0 \\ \mathsf{pred}\ (n+1) &= n \\ \\ \mathsf{sub}\ m\, 0 &= m \\ \mathsf{sub}\ m\, (n+1) &= \mathsf{pred}\, (\mathsf{sub}\ m\, n) \end{aligned}$$

   This is accomplished by taking:

   $$\begin{aligned} \mathsf{pred} &:= \mathsf{rec}\ \mathsf{nat}\ 0\ (\lambda n : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, n) \\ \mathsf{sub} &:= \lambda m : \mathsf{nat}.\, \mathsf{rec}\ \mathsf{nat}\ m\, (\lambda n : \mathsf{nat}.\, \lambda r : \mathsf{nat}.\, \mathsf{pred}\ r) \end{aligned}$$

2. One version of the *Ackermann function* is recursively defined by the equations:

   $$\begin{aligned} A(0, y) &= y + 1 \\ A(x+1, 0) &= A(x, 1) \\ A(x+1, y+1) &= A(x, A(x+1, y)) \end{aligned}$$

   Define this function as a term $A : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ of Gödel's system T. Also, to get an impression of this function, give explicit formulas for $A(1, y)$, $A(2, y)$ and $A(3, y)$ and calculate the value of $A(4, 2)$.

The formulas and value are:

$$\begin{aligned} A(1,y) &= y+2 \\ A(2,y) &= 2y+3 \\ A(3,y) &= 2^{y+3}-3 \\ A(4,y) &= 2^{2^{2^{2}}}-3 = 2^{65536}-3 = 2.00352\cdots 56733\times 10^{19728} \end{aligned}$$

This function can be defined using recursion on the type $\mathsf{nat}\to\mathsf{nat}$:

$$\begin{aligned} A \;:=\; & \mathsf{rec}\,(\mathsf{nat}\to\mathsf{nat}) \\ & \quad \mathsf{suc} \\ & \quad (\lambda x:\mathsf{nat}.\,\lambda r:\mathsf{nat}\to\mathsf{nat}. \\ & \qquad\qquad \mathsf{rec}\ \mathsf{nat}\ (r\,(\mathsf{suc}\,0))\,(\lambda y:\mathsf{nat}.\,\lambda s:\mathsf{nat}.\,rs)) \end{aligned}$$

3. Give the typing (formation, constructors, recursor) and reduction rules for the type of lists of natural numbers. Give both types for a dependent and a non-dependent recursor.

$$\begin{aligned} \mathsf{natlist} &: * \\ \mathsf{nil} &: \mathsf{natlist} \\ \mathsf{cons} &: \mathsf{nat}\to\mathsf{natlist}\to\mathsf{natlist} \\ \mathsf{natlist\_rec} &: \forall A:*. \\ &\qquad A\to \\ &\qquad (\mathsf{nat}\to\mathsf{natlist}\to A\to A)\to \\ &\qquad (\mathsf{natlist}\to A) \\ \mathsf{natlist\_rec} &: \forall A:(\mathsf{natlist}\to *). \\ &\qquad A\,\mathsf{nil}\to \\ &\qquad (\forall n:\mathsf{nat}.\,\forall l:\mathsf{natlist}.\,A\,l\to A\,(\mathsf{cons}\,n\,l))\to \\ &\qquad (\forall l:\mathsf{natlist}.\,A\,l) \end{aligned}$$

$$\begin{aligned} \mathsf{natlist\_rec}\ A\,M\,F\,\mathsf{nil} &\to_\iota M \\ \mathsf{natlist\_rec}\ A\,M\,F\,(\mathsf{cons}\,N\,L) &\to_\iota F\,N\,L\,(\mathsf{natlist\_rec}\ A\,F\,L) \end{aligned}$$

4. Give the typing (formation, constructors, recursor) and reduction rules for the type of polymorphic vectors. Give both types for a dependent and a non-dependent recursor.

$$
\begin{aligned}
\mathsf{polyvec} &: * \to \mathsf{nat} \to * \\
\mathsf{polynil} &: \forall A : *.\, \mathsf{polyvec}\, A\, 0 \\
\mathsf{polycons} &: \forall A : *.\, \forall n : \mathsf{nat}.\, A \to \mathsf{polyvec}\, A\, n \to \mathsf{polyvec}\, A\, (\mathsf{suc}\, n) \\
\mathsf{polyvec\_rec} &: \forall A : *.\, \forall P : (\mathsf{nat} \to *). \\
&\quad P\ 0 \to \\
&\quad (\forall n : \mathsf{nat}.\, \forall x : A.\, (\mathsf{polyvec}\ A\, n) \to \\
&\qquad\qquad P\, n \to P\, (\mathsf{suc}\, n)) \to \\
&\quad (\forall n : \mathsf{nat}.\, (\mathsf{polyvec}\ A\, n) \to P\, n) \\
\mathsf{polyvec\_rec} &: \forall A : *.\, \forall P : (\forall n : \mathsf{nat}.\, \mathsf{polyvec}\, A\, n \to *). \\
&\quad P\ 0\ (\mathsf{polynil}\ A) \to \\
&\quad (\forall n : \mathsf{nat}.\, \forall x : A.\, \forall l : \mathsf{polyvec}\ A\, n. \\
&\qquad\qquad P\, n\, l \to P\, (\mathsf{suc}\, n)\, (\mathsf{polycons}\, A\, n\, x\, l)) \to \\
&\quad (\forall n : \mathsf{nat}.\, \forall l : \mathsf{polyvec}\ A\, n.\, P\, n\, l)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{polyvec\_rec}\ A\, P\, M\, F\, N'\, (\mathsf{polynil}\ A') &\to_\iota M \\
\mathsf{polyvec\_rec}\ A\, P\, M\, F\, N'\, (\mathsf{polycons}\, A'\, N\, X\, L) &\to_\iota F\, N\, X\, L\, (\mathsf{polyvec\_rec}\ A\, P\, M\, F\, N\, L)
\end{aligned}
$$

5. Show how to define an inductive predicate $\mathsf{even} : \mathsf{nat} \to *$ that says whether its argument is even. Give the typing (formation, constructors, recursor) and reduction rules. Give both types for a dependent and a non-dependent recursor. (Hint: it is generally easier to first determine the type of the dependent recursor, as described in the lecture.)

$$
\begin{aligned}
\mathsf{even} &: \mathsf{nat} \to * \\
\mathsf{even\_0} &: \mathsf{even}\ 0 \\
\mathsf{even\_step} &: \forall n : \mathsf{nat}.\, \mathsf{even}\ n \to \mathsf{even}\, (\mathsf{suc}\, (\mathsf{suc}\, n))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{even\_rec} \;:\; & \forall P : (\mathsf{nat} \to *). \\
& P\,\mathsf{0} \to \\
& (\forall n : \mathsf{nat}.\, \mathsf{even}\, n \to \\
& \qquad P\, n \to P\,(\mathsf{suc}\,(\mathsf{suc}\, n))) \to \\
& (\forall n : \mathsf{nat}.\, \mathsf{even}\, n \to P\, n) \\
\mathsf{even\_rec} \;:\; & \forall P : (\forall n : \mathsf{nat}.\, \mathsf{even}\ n \to *). \\
& P\,\mathsf{0}\ \mathsf{even\_0} \to \\
& (\forall n : \mathsf{nat}.\, \forall X : \mathsf{even}\, n. \\
& \qquad P\, n\, X \to P\,(\mathsf{suc}\,(\mathsf{suc}\, n))\,(\mathsf{even\_step}\ n\, X)) \to \\
& (\forall n : \mathsf{nat}.\, \forall X : \mathsf{even}\, n.\, P\, n\, X)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{even\_rec}\ P\, M\, F\, N'\, \mathsf{even\_0} &\to_\iota M \\
\mathsf{even\_rec}\ P\, M\, F\, N'\, (\mathsf{even\_step}\ N\, X) &\to_\iota F\, N\, X\, (\mathsf{even\_rec}\ P M F N X)
\end{aligned}
$$

The way to read the dependent recursor is as an induction principle saying: 'if the predicate P on the type is conserved under all type constructors, then it holds on the whole type'. The non-dependent recursor then is obtained by removing the argument of the predicate that is the element of the type (in this case the second argument: but note that the first argument remains; and note that *all* quantifiers in the step case remain as well!)

A way to read the non-dependent recursor is that the set of even numbers is the smallest set that is closed under the constructors. However, with that interpretation it is not so easy to see what is the similarity in structure to the recursors for the natural numbers and lists.

6. Give the typing (formation, constructors, recursor) and reduction rules for Leibniz equality. Give both types for a dependent and a non-dependent recursor.

$$
\begin{aligned}
\mathsf{eq} \;:\; & \forall A : *.\, A \to A \to * \\
\mathsf{eq\_refl} \;:\; & \forall A : *.\, \forall x : A.\, \mathsf{eq}\ A\, x\, x
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{eq\_rec} \;:\; & \forall A : *.\, \forall P : (A \to A \to *). \\
& \quad (\forall x : A.\, P\, x\, x) \to \\
& \quad (\forall x : A.\, \forall y : A.\, \mathsf{eq}\; A\, x\, y \to P\, x\, y) \\
\mathsf{eq\_rec} \;:\; & \forall A : *.\, \forall P : (\forall x : A.\, \forall y : A.\, \mathsf{eq}\; A\, x\, y \to *). \\
& \quad (\forall x : A.\, P\, x\, x\, (\mathsf{eq\_refl}\; A\, x)) \to \\
& \quad (\forall x : A.\, \forall y : A.\, \forall X : \mathsf{eq}\; A\, x\, y.\, P\, x\, y\, X)
\end{aligned}
$$

$$
\mathsf{eq\_rec}\; A\, P\, F\, N_1\, N_2\, (\mathsf{eq\_refl}\, A'\, N) \quad \to_\iota \quad F\, N
$$

There is a subtlety here that should be explained. For each of the arguments of the inductive type, we can choose whether we keep them fixed or whether we allow them to change when applying the constructors. This matters for the shape of the recursion principle and for the reduction rules. What really happens when we decide to keep one or more arguments fixed is that we define a *family* of inductive types, one for each choice of those fixed arguments.

In fact we already did this in the rules for poly_vec: there we kept the type A fixed. It is an interesting exercise to figure out what the recursion principle for poly_vec becomes if one does not do that. Basically one will get extra '$\forall A : *$' quantifiers in various places, instead of just on the outside. To ask Coq for this recursion principle, execute:

```
Inductive polyvec : Type -> nat -> Type :=
| polynil : forall A, polyvec A O
| polycons : forall A n, A -> polyvec A n -> polyvec A (S n).

Check polyvec_rect.
```

In the rules for eq above, we kept the first argument A fixed and allowed the second and third arguments x and y to vary. This corresponds to the Coq definition:

```
Inductive eq (A : Type) : A -> A -> Type :=
| eq_refl : forall x : A, eq A x x.
```

However, in the actual definition of eq that is used in Coq *also* the second argument is kept fixed! This corresponds with the definition:

```
Inductive eq (A : Type) (x : A) : A -> Type :=
| eq_refl : eq A x x.
```

With this definition we get the following rules for eq (note that the formation and constructor types stay the same!):

$$\begin{aligned}
\mathsf{eq} \quad &: \quad \forall A : *.\, A \to A \to * \\
\mathsf{eq_refl} \quad &: \quad \forall A : *.\, \forall x : A.\, \mathsf{eq}\ A\, x\, x \\
\mathsf{eq_rec} \quad &: \quad \forall A : *.\, \forall x : A.\, \forall P : (A \to *). \\
&\qquad P\, x \to \\
&\qquad (\forall y : A.\, \mathsf{eq}\ A\, x\, y \to P\, y) \\
\mathsf{eq_rec} \quad &: \quad \forall A : *.\, \forall x : A.\, \forall P : (\forall y : A.\, \mathsf{eq}\ A\, x\, y \to *). \\
&\qquad P\, x\ (\mathsf{eq_refl}\ A\, x) \to \\
&\qquad (\forall y : A.\, \forall X : \mathsf{eq}\ A\, x\, y.\, P\, y\, X)
\end{aligned}$$

$$\mathsf{eq\_rec}\ A\, X\, P\, F\, Y\, (\mathsf{eq\_refl}\, A'\, X') \quad \to_\iota \quad F$$

The non-dependent recursor states that if some property holds for some x ('$P\,x$') and we know that $x = y$ ('$\mathsf{eq}\,A\,x\,y$'), then the property also holds for y ('$P\,y$'). The philosopher Gottfried Wilhelm Leibniz defined two objects to be equal precisely when they have exactly the same properties, which explains the use of the name Leibniz equality.

7. Give the type of the dependent recursor for the product type $A \times B$, and from that derive the type of a non-dependent recursor. Show how the functions π_1 and π_2 from MLW can be defined from this second recursor. Also, show how this recursor can be defined in terms of π_1 and π_2.

We had

$$\mathsf{pair} \quad : \quad \forall A : *.\, \forall B : *.\, A \to B \to \mathsf{prod} A B$$

and therefore the dependent recursion principle becomes (where we consider the arguments A and B fixed):

$$\begin{aligned}
\mathsf{prod_rec} \quad &: \quad \forall A : *.\, \forall B : *.\, \forall P : (\mathsf{prod}\ A\, B \to *). \\
&\qquad (\forall x : A.\, \forall y : B.\, P\, (\mathsf{pair}\ A\, B\, x\, y)) \to \\
&\qquad (\forall p : \mathsf{prod}\ A\, B.\, P\, p)
\end{aligned}$$

I.e., if P 'is conserved under' the constructors (in this case: holds for all objects constructed by the constructor pair), then it holds for all objects in the type.

The non-dependent recursion principle that corresponds to this is:

$$\begin{aligned}\mathsf{prod\_rec} \;:\; & \forall A:*.\,\forall B:*.\,\forall P:*.\\ & (A \to B \to P) \to\\ & (\mathsf{prod}\; A\, B \to P)\end{aligned}$$

The way that $\mathsf{prod\_rec}$ and the pair π_1 and π_2 can be defined in terms of each other is:

$$\begin{aligned}\pi_1 &:= \lambda A:*.\,\lambda B:*.\,\lambda p:\mathsf{prod}\; A\, B.\\ &\qquad \mathsf{prod\_rec}\; A\, B\, A\; (\lambda x:A.\,\lambda y:B.\,x)\, p\\ \pi_2 &:= \lambda A:*.\,\lambda B:*.\,\lambda p:\mathsf{prod}\; A\, B.\\ &\qquad \mathsf{prod\_rec}\; A\, B\, B\; (\lambda x:A.\,\lambda y:B.\,y)\, p\\ \mathsf{prod\_rec} &:= \lambda A:*.\,\lambda B:*.\,\lambda P:*.\,\lambda f:A\to B\to P.\\ &\qquad \lambda p:\mathsf{prod}\; A\, B.\, f\,(\pi_1\, p)\,(\pi_2\, p)\end{aligned}$$

8. Define the lists over a given type A as a W-type.

We first define a function

$$\mathsf{case} \;:\; \forall A:*.\,\forall B:*.\,A+B\to 2$$

by (remember that we defined $A+B := \Sigma x:2.\,R_2^*\; A\, B\, x$):

$$\mathsf{case} := \lambda A:*.\,\lambda B:*.\,\lambda z:(\Sigma x:2.\,R_2^*\; A\, B\, x).\,\pi_1\, 2\,(\lambda x:2.\,R_2^*\; A\, B\, x)\, z$$

We will label the nodes of the tree by elements of $1+A$, and the case function will tell us whether the node represents a nil or a cons. Once we have that, we can use a construction that is very similar to the one of the natural numbers:

$$\mathsf{polylist} := \lambda A:*.\,W x:(1+A).\,R_2^*\; 0\; 1\;(\mathsf{case}\; 1\; A\, x)$$

9. Give the typing (formation, constructors, recursor) and reduction rules of the W-trees. Give both types for a dependent and a non-dependent recursor.

We can think hard about this ourselves, but we can also ask Coq to show us the dependent recursor:

```
Inductive W (A : Type) (B : A -> Type) : Type :=
| sup : forall x : A, (B x -> W A B) -> W A B.

Check W_rect.
```

(Note that we keep the types A and B fixed here.) Coq will then tell us;

```
W_rect
     : forall (A : Type) (B : A -> Type) (P : W A B -> Type),
       (forall (x : A) (w : B x -> W A B),
        (forall b : B x, P (w b)) -> P (sup A B x w)) ->
       forall w : W A B, P w
```

This means that the typings are:

$$
\begin{aligned}
\mathsf{W} \;&:\; \forall A : *.\, (A \to *) \to * \\
\mathsf{sup} \;&:\; \forall A : *.\, \forall B : (A \to *).\, \forall x : A.\, (B\,x \to \mathsf{W}\,A\,B) \to \mathsf{W}\,A\,B \\
\mathsf{W\_rec} \;&:\; \forall A : *.\, \forall B : (A \to *).\, \forall P : *. \\
&\qquad (\forall x : A.\, (B\,x \to \mathsf{W}\,A\,B) \to \\
&\qquad\qquad (B\,x \to P) \to P) \to \\
&\qquad (\mathsf{W}\,A\,B \to P) \\
\mathsf{W\_rec} \;&:\; \forall A : *.\, \forall B : (A \to *).\, \forall P : (\mathsf{W}\,A\,B \to *). \\
&\qquad (\forall x : A.\, \forall w : (B\,x \to \mathsf{W}\,A\,B). \\
&\qquad\qquad (\forall b : B\,x.\, P\,(w\,b)) \to P\,(\mathsf{sup}\,A\,B\,x\,w)) \to \\
&\qquad (\forall w : \mathsf{W}\,A\,B.\, P\,w)
\end{aligned}
$$

The way to understand the constructor is: given a node label x and a function that gives you a W-tree for each edge label in Bx, you get a new W-tree by hanging all those W-trees under that node.

The way to understand the dependent recursor is: given a property P that is closed under this tree-building sup operation (i.e., whenever each of the trees that you give for an edge label satisfies the property – '$\forall b : B\,x.\,P\,(w\,b)$' – then the bigger tree you build from them satisfies it too – '$P\,(\mathsf{sup}\;A\,B\,x\,w)$'), then the property holds for all the W-trees.

The reduction that goes with these recursors is:

$$\mathsf{W\_rec}\;A\,B\,P\,F\,(\mathsf{sup}\;A'\,B'\,X\,W) \quad \rightarrow_\iota \quad F\,X\,W\,(\lambda b : (B\,X).\,\mathsf{W\_rec}\;A\,B\,P\,F\,(W\,b))$$

To make Coq agree that this is indeed the reduction rule that it uses, you can type:

```
Lemma W_iota : forall A B P F X W,
  W_rect A B P F (sup A B X W) =
  F X W (fun b : B X => W_rect A B P F (W b)).
Proof.
intros.
reflexivity.
Qed.
```

The `reflexivity` tactic only works when both sides of the equality are convertible.

You might wonder why in the reduction rule we have A versus A', while in the Coq lemma they are the same variable. The reason is that for the left hand side of the reduction rule to be well-typed A and A' have to be convertible: however, they don't have to be *identical* terms.