OPLSS 22 Introduction to Type Theory (3)

Cosmo Viola, Jiawei Chen, Nabil Hassein, Daniel Zackon

June 2022

1 The Law of the Excluded Middle and Reductio ad Absurdum

Yesterday, we saw that one of deMorgan's laws couldn't be proven in intuitionistic logic. Intuitionistic logic is able to prove fewer statements than classical logic because it lacks the law of the excluded middle (or in Latin, tertium non datur):

$$TND : prop \to prop$$
$$TND\ P = P \lor \neg P$$

If we assume the law of the excluded middle, we can prove the last of deMorgan's laws.

$$deMorgan' : TND\ P \to \neg(P \land Q) \iff \neg P \lor \neg Q$$
$$proj_1\ (deMorgan'\ (inj_1 p))\ npq = inj_2\ (\lambda\ q \to npq\ (p, q))$$
$$proj_1\ (deMorgan'\ (inj_2\ np))\ npq = inj_2\ np$$
$$proj_2\ (deMorgan'\_)\ (inj_1 np)\ pq = np\ (proj_1\ pq)$$
$$proj_2\ (deMorgan'\ \_)\ (inj_2\ nq)\ pq = nq\ (proj_2\ pq)$$

In classical logic, we often prove a statement by disproving its negation. This argument is called reductio ad absurdum. In type theory, this is

$$RAA : prop \to prop$$
$$RAA\ P = \neg\neg P \to P$$

It turns out that reductio ad absurdum is equivalent to the law of the excluded middle:

$$tnd \to raa : TND\ P \to RAA\ P$$
$$tnd \to raa : (inj_1\ p)\ nnp = p$$
$$tnd \to raa(inj_2\ np)\ nnp = case\bot\ (nnp\ np)$$

$$nntnd : \neg\neg(P \lor \neg P)$$
$$nntnd\ h = h\ (inj_2\ (\lambda p\ \ \to h\ (inj_1\ p)))$$

$$raa \to tnd : ((P : prop) \to RAA\ P) \to TND\ Q$$
$$raa \to tnd : raa = raa\ nntnd$$

In classical logic, we can define disjunction as follows:

$$\_ \lor^c \_ : prop \to prop \to prop$$
$$P \lor^c Q = \neg(\neg P \land \neg Q)$$

From this definition, we can prove the law of the excluded middle:

$$\begin{aligned}
&\textit{tnd-neg} : P \vee^c \neg P \\
&\textit{tnd-neg}\ h = proj_2\ h\ (proj_1\ h) \\
\\
&inj_1^c : P \to P \vee^c Q \\
&inj_1^c\ p\ h = proj_1\ h\ p \\
\\
&case^c : RAA\ P \to (P \to R) \to (Q \to R) \to (P \vee^c Q) \to R \\
&case^c\ raa\ f\ g\ x = raa\ (\lambda nr \to x\ ((\lambda p \to nr\ (f\ p)), \lambda q \to nr\ (g\ q)))
\end{aligned}$$

Thus, we see that $\neg, \wedge, \vee^c, True, False$ preserve reductio ad absurdum and hence are classical. We also see that $\vee^c$ can be defined in terms of other connectives, and likewise with $\exists$ in predicate logic.

2 Inductive Types

Now, we will consider inductive and coinductive types. One example in Agda is the natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

We can consider some examples of elements of $\mathbb{N}$:

$$\begin{aligned}
&one : \mathbb{N} \\
&one = suc\ zero \\
\\
&two : \mathbb{N} \\
&two = suc\ one
\end{aligned}$$

Agda has the $\mathbb{N}$ type built in, and translates Arabic numerals written to be elements of that inductive type.

2.1 Structural Recursion

Now, let's define some functions on $\mathbb{N}$.

$$\begin{aligned}
&double : \mathbb{N} \to \mathbb{N} \\
&double\ zero = 0 \\
&double\ (suc\ n) = suc\ (suc\ (double\ n)) \\
\\
&half : \mathbb{N} \to \mathbb{N} \\
&half\ zero = 0 \\
&half\ (suc\ zero) = 0 \\
&half\ (suc\ (suc\ n)) = suc\ (half\ n)
\end{aligned}$$

We can give a combinator for recursion on $\mathbb{N}$:

$$f : N \to M$$
$$f\ zero = m - 0$$
$$f(suc\ n) = m - s\ (f\ n)$$

$$It\text{-}N : M \to (M \to M) \to N \to M$$
$$It\text{-}N\ m - 0\ m - s\ zero = m - 0$$
$$It\text{-}N\ m - 0\ m - s\ (suc\ n) = m - S\ (It\text{-}N\ m - 0\ m - S\ n)$$

Let's try some catamorphisms:

$$double\text{-}it : \mathbb{N} \to \mathbb{N}$$
$$double\text{-}it = It\text{-}N\ 0\ (\lambda d - n \to suc\ (suc\ d - n))$$

$half\text{-}it\text{-}aux : \mathbb{N} \to \mathbb{N} * \mathbb{N} -$ idea: half-it-aux n = half n, half (suc n)

$$half\text{-}it\text{-}aux = It\text{-}N(0,0)(\lambda x \to (proj_2\ x), suc\ (proj_1\ x))$$

$$half\text{-}it : \mathbb{N} \to \mathbb{N}$$
$$half\text{-}it\ n = proj_1\ (half\text{-}it\text{-}aux\ n)$$

Addition:

$$\_ + \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$zero + n = n$$
$$suc\ m + n = suc\ (m + n)$$

$$\_\text{+-}it\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\_\text{+-}it\_ : It\text{-}N(\lambda\ n \to n)\lambda\ m+\ \ n \to suc(m+\ \ n)$$

Multiplication:

$$\_ * \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$zero * n = 0$$
$$sucm * n = m * n + n$$

$$fac : \mathbb{N} \to \mathbb{N}$$
$$fac\ zero = 1$$
$$fac\ (suc\ n) = (suc\ n) * fac\ n$$

Exercise: derive fac-it just using the iterator.

There are many inductive types; common examples include lists and trees. Here is another example in Agda:

```
data Ord : Set where
  zero : Ord
  suc : Ord → Ord
  lim : (ℕ → Ord) → Ord
```

These are the ordinals; the lim constructor gives the limit ordinal.

3 Coinductive Types

We can also consider coinductive types. Here is the type of streams in Agda:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

An inductive type tells you how you can construct data. A coinductive type tells you how you can consume data. Next, we define a function on this coinductive type:

$$
\begin{aligned}
&\_ :: \_ : A \to StreamA \to StreamA \\
&head\ (a :: aa) = a \\
&tail\ (a :: aa) = aa \\
\\
&from : \mathbb{N} \to Stream\mathbb{N} \\
&head\ (from\ n) = n \\
&tail\ (from\ n) = from\ (suc\ n) \\
\\
&mapS : (A \to B) \to StreamA \to StreamB \\
&head\ (mapS\ f\ as) = f\ (head\ as) \\
&tail\ (mapS\ f\ as) = mapS\ f\ (tail\ as)
\end{aligned}
$$

To define a function into Streams, we need to define how to consume each element the function maps to.

$$
\begin{aligned}
&f : M \to Stream\ A \\
&head\ (f\ m) = m\text{-}h\ m \\
&tail\ (f\ m) = f\ (m\text{-}t\ m) \\
\\
&CoIt\text{-}Stream : (M \to A) \to (M \to M) \to M \to StreamA \\
&head\ (CoIt\text{-}Streamm\text{-}h\ m\text{-}t\ m) = m\text{-}h\ m \\
&tail\ (CoIt\text{-}Streamm\text{-}h\ m\text{-}t\ m) = CoIt\text{-}Stream\ m\text{-}\ m\text{-}t\ (m\text{-}t\ m)
\end{aligned}
$$

Exercise: translate from and mapS into using only CoIt-Stream

Next, we define the conatural numbers. A conatural number is something which you can compute a predecessor of.

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just : A → Maybe A

record ℕ∞ : Set where
  coinductive
  field
```

```
    pred : Maybe ℕ∞

zero∞ : ℕ∞
pred zero∞ = nothing

suc∞ : ℕ∞ → ℕ∞
pred (suc∞ n) = just n

∞ : ℕ∞
pred ∞ = just ∞
```

Then, we can define addition on conatural numbers.

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
pred (m +∞ n) with pred m
... | nothing = pred n
... | just m' = just (m' +∞ n)

CoIt−ℕ∞ : (M → Maybe M) → M → ℕ∞
pred (CoIt−ℕ∞ f m) with f m
... | nothing = nothing
... | just m = just (CoIt−ℕ∞ f m)
```

As a challenge, the reader should construct multiplication for coinductive natural numbers.