# The HoTT-Book

## Chapters 1.8 through 1.10

HoTT-Seminar, SS2017

# Today's Topics

1. The Type of Booleans
2. The Natural Numbers
3. Pattern Matching and Recursion

# The Type of Booleans

# The Type of Booleans

## Introduction

- The type of booleans $\mathbf{2} : \mathcal{U}$ is intended to have exactly two elements $0_{\mathbf{2}}, 1_{\mathbf{2}} : \mathbf{2}$.
- It can be constructed out of coproduct and unit types as $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}$:

$$0_{\mathbf{2}} :\equiv \mathrm{inl}(\star),$$
$$1_{\mathbf{2}} :\equiv \mathrm{inr}(\star).$$

- We will formulate explicit rules nonetheless since it is used frequently.

# The Type of Booleans

## Recursor

- To derive a function $f : \mathbf{2} \to C$ we need $c_0, c_1 : C$ and add the defining equations

$$f(0_{\mathbf{2}}) :\equiv c_0,$$
$$f(1_{\mathbf{2}}) :\equiv c_1.$$

- The recursor for $\mathbf{2}$ generating such functions is

$$\mathrm{rec}_{\mathbf{2}} : \prod_{C:\mathcal{U}} C \to C \to \mathbf{2} \to C$$

with the defining equations

$$\mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0,$$
$$\mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1.$$

- This is if-then-else!

- Given $C : \mathbf{2} \to \mathcal{U}$, to derive a dependent function $f : \prod_{(x:\mathbf{2})} C(x)$ we need $c_0 : C(0_\mathbf{2})$ and $c_1 : C(1_\mathbf{2})$, in which case we can give the defining equations

$$f(0_\mathbf{2}) :\equiv c_0,$$
$$f(1_\mathbf{2}) :\equiv c_1.$$

- The induction principle for $\mathbf{2}$ generating such (dependant) functions is

$$\mathrm{ind}_\mathbf{2} : \prod_{(C:\mathbf{2}\to\mathcal{U})} C(0_\mathbf{2}) \to C(1_\mathbf{2}) \to \prod_{(x:\mathbf{2})} C(x)$$

with the defining equations

$$\mathrm{ind}_\mathbf{2}(C, c_0, c_1, 0_\mathbf{2}) :\equiv c_0,$$
$$\mathrm{ind}_\mathbf{2}(C, c_0, c_1, 1_\mathbf{2}) :\equiv c_1.$$

# The Type of Booleans

## Example

- Let's prove that $\mathbf{2}$ really has only two inhabitants.
- That is, we have to prove that every $x : \mathbf{2}$ equals either $0_\mathbf{2}$ or $1_\mathbf{2}$ (propositionally).
- Written formally, this proposition becomes the type

$$\prod_{x:\mathbf{2}} (x = 0_\mathbf{2}) + (x = 1_\mathbf{2})$$

- Again, we use equality types although we have not yet introduced them. But as before, we need only the fact that everything is equal to itself by $\mathrm{refl}_x : x = x$.
- Now to prove our proposition, we have to construct an element of that type, i.e. a (dependant) function assigning to each $x : \mathbf{2}$ either an equality $x = 0_\mathbf{2}$ or an equality $x = 1_\mathbf{2}$.

# The Type of Booleans

## Example (cont.)

- We use the induction principle for $\mathbf{2}$

$$\mathrm{ind}_{\mathbf{2}} : \prod_{(C:\mathbf{2}\to\mathcal{U})} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \textstyle\prod_{(x:\mathbf{2})} C(x)$$

  with $C(x) :\equiv (x = 0_{\mathbf{2}}) + (x = 1_{\mathbf{2}})$.
- The two inputs are $\mathrm{inl}(\mathrm{refl}_{0_{\mathbf{2}}}) : C(0_{\mathbf{2}})$ and $\mathrm{inr}(\mathrm{refl}_{1_{\mathbf{2}}}) : C(1_{\mathbf{2}})$.
- So we've got:

$$\mathrm{ind}_{\mathbf{2}}\big(\lambda x.\,(x = 0_{\mathbf{2}}) + (x = 1_{\mathbf{2}}),\ \mathrm{inl}(\mathrm{refl}_{0_{\mathbf{2}}}),\ \mathrm{inr}(\mathrm{refl}_{1_{\mathbf{2}}})\big) : \prod_{x:\mathbf{2}} (x = 0_{\mathbf{2}}) + (x = 1_{\mathbf{2}}).$$

□

# The Type of Booleans

## Encoding Product and Coproduct with Booleans

- Last time, we heard that $\Sigma$-types can be regarded as analogous to indexed disjoint unions, while coproducts are binary disjoint unions.
- It is natural to expect that a binary disjoint union $A + B$ could be constructed as an indexed one over the two-element type $\mathbf{2}$.
- In fact, this can be done quite easily by defining

$$A + B :\equiv \sum_{x:\mathbf{2}} \mathrm{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x).$$

with

$$\mathrm{inl}(a) :\equiv (0_{\mathbf{2}}, a),$$
$$\mathrm{inr}(b) :\equiv (1_{\mathbf{2}}, b).$$

# The Type of Booleans

## Encoding Product and Coproduct with Booleans (cont.)

- We can apply the same idea to products and $\Pi$-types:
- We could have defined

$$A \times B :\equiv \prod_{x:\mathbf{2}} \mathrm{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x).$$

- Pairs could then be constructed using induction for $\mathbf{2}$:

$$(a, b) :\equiv \mathrm{ind}_{\mathbf{2}}(\mathrm{rec}_{\mathbf{2}}(\mathcal{U}, A, B), a, b)$$

while the projections are straightforward applications

$$\mathrm{pr}_1(p) :\equiv p(0_{\mathbf{2}}),$$
$$\mathrm{pr}_2(p) :\equiv p(1_{\mathbf{2}}).$$

# The Type of Booleans

## Formal Presentation

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{2} : \mathcal{U}_i}\ \mathbf{2}\text{-FORM} \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash 0_{\mathbf{2}} : \mathbf{2}}\ \mathbf{2}\text{-INTRO}_1 \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash 1_{\mathbf{2}} : \mathbf{2}}\ \mathbf{2}\text{-INTRO}_2$$

$$\frac{\begin{array}{c}\Gamma, z{:}\mathbf{2} \vdash C : \mathcal{U}_i \\ \Gamma \vdash c : C[0_{\mathbf{2}}/z] \qquad \Gamma \vdash d : C[1_{\mathbf{2}}/z] \qquad \Gamma \vdash e : \mathbf{2}\end{array}}{\Gamma \vdash \mathrm{ind}_{\mathbf{2}}(z.C, c, d, e) : C[e/z]}\ \mathbf{2}\text{-ELIM}$$

$$\frac{\Gamma, z{:}\mathbf{2} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[0_{\mathbf{2}}/z] \qquad \Gamma \vdash d : C[1_{\mathbf{2}}/z]}{\Gamma \vdash \mathrm{ind}_{\mathbf{2}}(z.C, c, d, 0_{\mathbf{2}}) \equiv c : C[0_{\mathbf{2}}/z]}\ \mathbf{2}\text{-COMP}_1$$

$$\frac{\Gamma, z{:}\mathbf{2} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[0_{\mathbf{2}}/z] \qquad \Gamma \vdash d : C[1_{\mathbf{2}}/z]}{\Gamma \vdash \mathrm{ind}_{\mathbf{2}}(z.C, c, d, 1_{\mathbf{2}}) \equiv d : C[1_{\mathbf{2}}/z]}\ \mathbf{2}\text{-COMP}_2$$

# The Natural Numbers

# The Natural Numbers

## Introduction

- Let's construct infinite types.
- The simplest infinite type is the type $\mathbb{N} : \mathcal{U}$ of natural numbers.
- As usual, the elements of $\mathbb{N}$ are constructed using $0 : \mathbb{N}$ and the successor operation $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$.
- When denoting natural numbers, we adopt the usual decimal notation

$$1 :\equiv \mathrm{succ}(0),$$
$$2 :\equiv \mathrm{succ}(1),$$
$$3 :\equiv \mathrm{succ}(2),$$
$$\ldots$$

- We can define functions $f : \mathbb{N} \to C$ by recursion.
- We can perform proofs $p : \prod_{(n:\mathbb{N})} C(n)$ by induction.
- Now the words “recursion” and “induction” are in a more familiar context.
- However, we will see that they are not a new concept, but correspond to the recursor and the induction principle for the type of natural numbers, respectively.

- To construct a function $f : \mathbb{N} \to C$ by recursion, we need a starting point $c_0 : C$ and a "next step" function $c_s : \mathbb{N} \to C \to C$ and add the defining equations

$$f(0) :\equiv c_0,$$
$$f(\mathrm{succ}(n)) :\equiv c_s(n, f(n)).$$

- We say that $f$ is defined by primitive recursion.
- The recursor for $\mathbb{N}$ generating such (recursive) functions is

$$\mathrm{rec}_{\mathbb{N}} : \prod_{(C:\mathcal{U})} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

with the defining equations

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0,$$
$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) :\equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n)).$$

# The Natural Numbers

## Example

- Let's define a function double : $\mathbb{N} \to \mathbb{N}$ which doubles its argument:
- We put $c_0 :\equiv 0$ and $c_s(n, y) :\equiv \mathrm{succ}(\mathrm{succ}(y))$ obtaining:

$$\mathrm{double}(0) :\equiv 0$$
$$\mathrm{double}(\mathrm{succ}(n)) :\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{double}(n))).$$

- Using $\mathrm{rec}_{\mathbb{N}}$ we can present double as follows:

$$\mathrm{double} :\equiv \mathrm{rec}_{\mathbb{N}}\left(\mathbb{N},\ 0,\ \lambda n.\, \lambda y.\, \mathrm{succ}(\mathrm{succ}(y))\right)$$

- This indeed has the correct computational behavior, for example:

$$\begin{aligned}
\mathrm{double}(2) &\equiv \mathrm{double}(\mathrm{succ}(\mathrm{succ}(0))) \\
&\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{double}(\mathrm{succ}(0)))) \\
&\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{double}(0))))) \\
&\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0)))) \\
&\equiv 4.
\end{aligned}$$

- We can define multi-variable functions by primitive recursion as well, by currying and allowing $C$ to be a function type.
- For example, we define addition $\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ with $C :\equiv \mathbb{N} \to \mathbb{N}$ and the following "starting point" and "next step" data:

$$c_0 : \mathbb{N} \to \mathbb{N}$$
$$c_0(n) :\equiv n$$
$$c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$
$$c_s(m, g)(n) :\equiv \mathrm{succ}(g(n)).$$

- We thus obtain $\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ satisfying

$$\mathrm{add}(0, n) \equiv n$$
$$\mathrm{add}(\mathrm{succ}(m), n) \equiv \mathrm{succ}(\mathrm{add}(m, n)).$$

- Or: $\mathrm{add} :\equiv \mathrm{rec}_{\mathbb{N}}\left(\mathbb{N} \to \mathbb{N}, \lambda n.\, n, \lambda m.\, \lambda g.\, \lambda n.\, \mathrm{succ}(g(n))\right)$.

- Given $C : \mathbb{N} \to \mathcal{U}$, an element $c_0 : C(0)$, and a (dependant) function $c_s : \prod_{(n:\mathbb{N})} C(n) \to C(\mathrm{succ}(n))$, we can construct $f : \prod_{(n:\mathbb{N})} C(n)$ with the defining equations:

$$f(0) :\equiv c_0,$$
$$f(\mathrm{succ}(n)) :\equiv c_s(n, f(n)).$$

- The induction principle for $\mathbb{N}$ generating such functions is

$$\mathrm{ind}_{\mathbb{N}} : \prod_{(C:\mathbb{N}\to\mathcal{U})} C(0) \to \left( \prod_{(n:\mathbb{N})} C(n) \to C(\mathrm{succ}(n)) \right) \to \prod_{(n:\mathbb{N})} C(n)$$

with the defining equations

$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0,$$
$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) :\equiv c_s(n, \mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, n)).$$

# The Natural Numbers

## Example

- Let's prove that our function add is associative.
- As usual, we write $\mathsf{add}(m, n)$ as $m + n$.
- Written formally, our goal is

$$\mathsf{assoc} : \prod_{i,j,k:\mathbb{N}} i + (j + k) = (i + j) + k$$

  so we have to construct such an element assoc.
- We use the induction principle for $\mathbb{N}$:

$$\mathsf{ind}_{\mathbb{N}} : \prod_{(C:\mathbb{N}\to\mathcal{U})} C(0) \to \left( \prod_{(n:\mathbb{N})} C(n) \to C(\mathsf{succ}(n)) \right) \to \prod_{(n:\mathbb{N})} C(n)$$

  with $C(i) :\equiv \prod_{(j,k:\mathbb{N})} i + (j + k) = (i + j) + k$.

- So we need

$$\mathsf{assoc}_0 : \prod_{j,k:\mathbb{N}} 0 + (j + k) = (0 + j) + k$$

and

$$\mathsf{assoc}_s : \prod_{i:\mathbb{N}} \left( \prod_{j,k:\mathbb{N}} i + (j + k) = (i + j) + k \right)$$

$$\to \left( \prod_{j,k:\mathbb{N}} \mathsf{succ}(i) + (j + k) = (\mathsf{succ}(i) + j) + k \right)$$

- Now recall the two defining equations for $+$:

$$0 + n \equiv n$$

$$\mathsf{succ}(m) + n \equiv \mathsf{succ}(m + n).$$

# The Natural Numbers

## Example (cont.)

- For $\mathsf{assoc}_0$ we use $0 + n \equiv n$ to get

$$0 + (j + k) \equiv j + k \equiv (0 + j) + k.$$

So we can just set

$$\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}.$$

- For $\mathsf{assoc}_s$ we use $\mathsf{succ}(m) + n \equiv \mathsf{succ}(m + n)$ to get

$$\mathsf{succ}(i) + (j + k) \equiv \mathsf{succ}(i + (j + k)) \qquad \text{and}$$
$$(\mathsf{succ}(i) + j) + k \equiv \mathsf{succ}((i + j) + k).$$

So, we can rewrite the output type of $\mathsf{assoc}_s$ as

$$\mathsf{succ}(i + (j + k)) = \mathsf{succ}((i + j) + k),$$

while its input type is

$$i + (j + k) = (i + j) + k.$$

- So it suffices to invoke the fact that if two natural numbers are equal, then so are their successors:

$$\mathrm{ap}_{\mathrm{succ}} : (m =_{\mathbb{N}} n) \to (\mathrm{succ}(m) =_{\mathbb{N}} \mathrm{succ}(n)),$$

- We will be able to prove this obvious fact when we have introduced identity types.
- So now we can define

$$\mathrm{assoc}_s(i, h, j, k) :\equiv \mathrm{ap}_{\mathrm{succ}}(h(j, k)),$$

and together with $\mathrm{assoc}_0$ we can define

$$\mathrm{assoc} :\equiv \mathrm{ind}_{\mathbb{N}}(C, \mathrm{assoc}_0, \mathrm{assoc}_s) : \prod_{i,j,k:\mathbb{N}} i + (j + k) = (i + j) + k$$

□

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i}\ \mathbb{N}\text{-FORM}$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash 0 : \mathbb{N}}\ \mathbb{N}\text{-INTRO}_1 \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{succ}(n) : \mathbb{N}}\ \mathbb{N}\text{-INTRO}_2$$

$$\frac{\Gamma \vdash c_0 : C[0/x] \qquad \begin{array}{c}\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \\ \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\text{succ}(x)/x]\end{array} \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, n) : C[n/x]}\ \mathbb{N}\text{-ELIM}$$

$$\frac{\Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathrm{succ}(x)/x]}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, 0) \equiv c_0 : C[0/x]}\ \mathbb{N}\text{-COMP}_1$$

$$\frac{\Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathrm{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, \mathrm{succ}(n)) \equiv c_s[n, \mathrm{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, n)/x, y] : C[\mathrm{succ}(n)/x]}\ \mathbb{N}\text{-COMP}_2$$

- Other inductively defined types follow the same general scheme.

# Pattern Matching and Recursion

# Pattern Matching and Recursion

## Recall: Function Definition for Coproducts

- To define a function $f : A + B \to C$, we could either use the recursor:

$$f :\equiv \mathrm{rec}_{A+B}(C, g_0, g_1)$$

or we could give the defining equations:

$$f(\mathrm{inl}(a)) :\equiv g_0(a)$$
$$f(\mathrm{inr}(b)) :\equiv g_1(b).$$

- To go from the former expression of $f$ to the latter, we simply use the computation rules for the recursor.
- Conversely, given any defining equations $f(\mathrm{inl}(a)) :\equiv \Phi_0$ and $f(\mathrm{inr}(b)) :\equiv \Phi_1$, where $\Phi_0$ and $\Phi_1$ are expressions that may involve the variables $a$ and $b$ respectively, we can express these equations equivalently in terms of the recursor by using $\lambda$-abstraction:

$$f :\equiv \mathrm{rec}_{A+B}(C, \lambda a.\, \Phi_0, \lambda b.\, \Phi_1).$$

- In the case of the natural numbers, however, the "defining equations" of a function such as double:

$$\begin{aligned} \mathrm{double}(0) &:\equiv 0 \\ \mathrm{double}(\mathrm{succ}(n)) &:\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{double}(n))) \end{aligned}$$

  involve *the function* double *itself* on the right-hand side.
- Thus, if we want to express these equations in terms of the recursor, we cannot use $\lambda$-abstraction straightaway.
- The solution is to read the expression "double$(n)$" on the right-hand side of the equation as standing in for the result of the recursive call, which in a definition of the form double $:\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$ would be the second argument of $c_s$.

- So, if we have a "definition" of a function $f : \mathbb{N} \to C$ such as

$$f(0) :\equiv \Phi_0$$
$$f(\mathrm{succ}(n)) :\equiv \Phi_s$$

where $\Phi_0$ is an expression of type $C$, and $\Phi_s$ is an expression of type $C$ which may involve the variable $n$ and also the symbol "$f(n)$", we may translate it to a definition

$$f :\equiv \mathrm{rec}_{\mathbb{N}}(C,\ \Phi_0,\ \lambda n.\, \lambda r.\, \Phi'_s)$$

where $\Phi'_s$ is obtained from $\Phi_s$ by replacing all occurrences of "$f(n)$" by the new variable $r$.

- This style of defining functions by recursion (or, more generally, dependent functions by induction) is called definition by pattern matching.
- However, as we have seen, expressing such a definition in terms of the recursor involves the substitution of $r$ for the composite symbol "$f(n)$", which is from a formal point of view quite questionable.
- Thus, whenever we introduce a new kind of inductive definition, we always begin by deriving its induction principle.
- Only then do we introduce an appropriate sort of "pattern matching" which can be justified as a shorthand for the induction principle.

# Pattern Matching and Recursion

## Function Definition by Pattern Matching (cont.)

- Note that (unlike in programming languages) we are restricted in what sort of recursive calls we can make: in order for such a definition to be re-expressible using the recursion principle, the function $f$ being defined can only appear in the body of $f(\mathrm{succ}(n))$ as part of the composite symbol "$f(n)$".
- Otherwise, we could write nonsense functions such as

$$f(0) :\equiv 0$$
$$f(\mathrm{succ}(n)) :\equiv f(\mathrm{succ}(\mathrm{succ}(n))).$$

- If a programmer wrote such a function, it would simply call itself forever on any positive input, going into an infinite loop and never returning a value.
- In mathematics, however, to be worthy of the name, a *function* must always associate a unique output value to every input value, so this would be unacceptable.

# The End

Questions?