The type ℕ of natural numbers

We repeat the definition given earlier:

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

{-# BUILTIN NATURAL ℕ #-}
```

Elimination principle

The elimination principle for all type formers follow the same pattern: they tell us how to define dependent functions *out* of the given type. In the case of natural numbers, the eliminator gives primitive recursion. Given a base case `a : A 0` and a step function `f : (k : ℕ) → A k → A (suc k)`, we get a function `h : (n : ℕ) → A n` defined by primitive recursion as follows:

```
ℕ-elim : {A : ℕ → Type}
       → A 0
       → ((k : ℕ) → A k → A (suc k))
       → (n : ℕ) → A n
ℕ-elim {A} a f = h
 where
  h : (n : ℕ) → A n
  h 0       = a
  h (suc n) = f n (h n)
```

In usual accounts of primitive recursion outside type theory, one encounters the following particular case of primitive recursion, which is the non-dependent version of the above.

```

ℕ-nondep-elim : {A : Type}
              → A
              → (ℕ → A → A)
              → ℕ → A
ℕ-nondep-elim {A} = ℕ-elim {λ _ → A}
```

Notice that this is like `fold` for lists. There is a further restricted version, which is usually called iteration:

```
ℕ-iteration : {A : Type}
            → A
            → (A → A)
            → ℕ → A
ℕ-iteration a f = ℕ-nondep-elim a (λ k x → f x)
```

Intuitively, `ℕ-iteration a f n = f (f (f (⋯ f a)))` where we apply `n` times the function `f` to the element `a`, which sometimes is written `fⁿ(a)` in the literature.

The induction principle for ℕ

In logical terms, one can see immediately what the type of `ℕ-elim` is: it is simply the principle of induction on natural numbers, which say that in order to show that a property `A` holds for all natural numbers, it is enough to show that `A 0` holds (this is called the base case), and that if `A k` holds then so does `A (suc k)` (this is called the induction step). In Agda, in practice, we don't explicitly use this induction principle, but instead write definition recursively, just as we defined the above function `h` recursively.

Addition and multiplication

As an **exercise**, you may try to define the following functions using some version of the above eliminators instead of using pattern matching and recursion:

```
_+_ : ℕ → ℕ → ℕ
zero  + y = y
suc x + y = suc (x + y)

_*_ : ℕ → ℕ → ℕ
zero  * y = 0
suc x * y = x * y + y

infixr 20 _+_
infixr 30 _*_
```

Reasoning with negation

Recall that we defined the negation `¬ A` of a type `A` to be the function type `A → 𝟘`, and that we also wrote `is-empty A` as a synonym of `¬ A`.

Emptiness of the empty type

We have the following two proofs of "not false" or "the empty type is empty":

```
not-false : ¬ 𝟘
not-false = 𝟘-elim

not-false' : ¬ 𝟘
not-false' = id
```

A lot of things about negation don't depend on the fact that the target type of the function type is `𝟘`. We will begin by proving some things about negation by generalizing `𝟘` to any type `R` of "results".

Implication from disjunction and negation

If `¬ A` or `B`, then `A implies B`:

```
implication-from-disjunction-and-negation : {A B : Type} → ¬ A ∔ B → (A → B)
implication-from-disjunction-and-negation (inl f) a = 𝟘-elim (f a)
implication-from-disjunction-and-negation (inr b) a = b
```

Contrapositives

If `A` implies `B`, then `B → R` implies `A → R`:

```
arrow-contravariance : {A B R : Type}
                     → (A → B)
                     → (B → R) → (A → R)
arrow-contravariance f g = g ∘ f
```

A particular case of interest is the following. The contrapositive of an implication `A → B` is the implication `¬ B → ¬ A`:

```
contrapositive : {A B : Type} → (A → B) → (¬ B → ¬ A)
contrapositive {A} {B} = arrow-contravariance {A} {B} {𝟘}
```

This can also be read as "if we have a function A → B and B is empty, then also A must be empty".

Double and triple negations

We now introduce notation for double and triple negation, to reduce the number of needed brackets:

```
¬¬_ ¬¬¬_ : Type → Type
¬¬  A = ¬(¬ A)
¬¬¬ A = ¬(¬¬ A)
```

We have that `A` implies `¬¬ A`. This is called double negation introduction. More generally, we have the following:

```
dni : (A R : Type) → A → ((A → R) → R)
dni A R a u = u a

double-negation-intro : (A : Type) → A → ¬¬ A
double-negation-intro A = dni A 𝟘
```

We don't always have `¬¬ A → A` in proofs-as-programs. This has to do with *computability theory*. But sometimes we do. For example, if we know that `A ∔ ¬ A` then `¬¬A → A` follows:

```
 ¬¬-elim : {A : Type} → A ∔ ¬ A → ¬¬ A → A
 ¬¬-elim (inl x) f = x
 ¬¬-elim (inr g) f = 𝟘-elim (f g)
```

For more details, see the lecture notes on decidability, where we discuss `¬¬-elim` again. But three negations always imply one, and conversely:

```
three-negations-imply-one : (A : Type) → ¬¬¬ A → ¬ A
three-negations-imply-one A = contrapositive (double-negation-intro A)

one-negation-implies-three : (A : Type) → ¬ A → ¬¬¬ A
one-negation-implies-three A = double-negation-intro (¬ A)
```

Negation of the identity type

It is useful to introduce a notation for the negation of the identity type:

```
_≢_ : {X : Type} → X → X → Type
x ≢ y = ¬ (x ≡ y)

≢-sym : {X : Type} {x y : X} → x ≢ y → y ≢ x
≢-sym = contrapositive sym

false-is-not-true : false ≢ true
false-is-not-true ()

true-is-not-false : true ≢ false
true-is-not-false ()
```

The following is more interesting:

```
not-false-is-true : (x : Bool) → x ≢ false → x ≡ true
not-false-is-true true  f = refl true
not-false-is-true false f = 𝟘-elim (f (refl false))

not-true-is-false : (x : Bool) → x ≢ true → x ≡ false
not-true-is-false true  f = 𝟘-elim (f (refl true))
not-true-is-false false f = refl false
```

Disjointness of binary sums

We now show something that is intuitively the case:

```
inl-is-not-inr : {X Y : Type} {x : X} {y : Y} → inl x ≢ inr y
inl-is-not-inr ()
```

Agda just knows it.

Disjunctions and negation

If `A or B` holds and `B` is false, then `A` must hold:

```
right-fails-gives-left-holds : {A B : Type} → A ∔ B → ¬ B → A
right-fails-gives-left-holds (inl a) f = a
right-fails-gives-left-holds (inr b) f = 𝟘-elim (f b)

left-fails-gives-right-holds : {A B : Type} → A ∔ B → ¬ A → B
left-fails-gives-right-holds (inl a) f = 𝟘-elim (f a)
left-fails-gives-right-holds (inr b) f = b
```

Negation of the existential quantifier:

If there is no `x : X` with `A x`, then for all `x : X` not `A x`:

```
not-exists-implies-forall-not : {X : Type} {A : X → Type}
                              → ¬ (Σ x : X , A x)
                              → (x : X) → ¬ A x
not-exists-implies-forall-not f x a = f (x , a)
```

The converse also holds:

```
forall-not-implies-not-exists : {X : Type} {A : X → Type}
                              → ((x : X) → ¬ A x)
                              → ¬ (Σ x : X , A x)
forall-not-implies-not-exists g (x , a) = g x a
```

Notice how these are particular cases of `curry` and `uncurry`.

Implication truth table

Here is a proof of the implication truth-table:

```
open import empty-type
open import unit-type

implication-truth-table : ((𝟘 → 𝟘) ⇔ 𝟙)
                        × ((𝟘 → 𝟙) ⇔ 𝟙)
                        × ((𝟙 → 𝟘) ⇔ 𝟘)
                        × ((𝟙 → 𝟙) ⇔ 𝟙)
implication-truth-table = ((λ _ → ⋆)   , (λ _ → id)) ,
                          ((λ _ → ⋆)   , (λ _ _ → ⋆)) ,
                          ((λ f → f ⋆) , (λ ⋆ _ → ⋆)) ,
                          ((λ _ → ⋆)   , (λ _ _ → ⋆))
```

TODO. Find a better home for the above truth table.

Function extensionality

Recall that we defined pointwise equality `f ∼ g` of functions in the identity type handout. The principle of function extensionality says that pointwise equal functions are equal and is given by the following type `FunExt`:

```
open import identity-type

FunExt = {A : Type} {B : A → Type} {f g : (x : A) → B x} → f ∼ g → f ≡ g
```

Unfortunately, this principle is not provable or disprovable in Agda or MLTT (we say that it is independent). But it is provable in Cubical Agda, which is based on Cubical Type Theory and is outside the scope of these lecture notes. Sometimes we will use function extensionality as an explicit assumption.

[[And some simple examples of uses of function extensionality here.]]

Propositions as types versus propositions as booleans

When programming in Haskell, and indeed in C or Java or Python, etc., we use *booleans* rather than *types* to encode logical propositions.

We now discuss *why* we use *types* to encode logical propositions, and *when* we can use *booleans* instead. It is not always. It is here that the prerequisite *Theories of Computation* shows up.

Discussion and motivation

In Haskell, we have a function `(==) : Eq a => a -> a -> Bool`. The type constraint `Eq a` is a prerequisite for this function because not all types have decidable equality. What does this mean? It means that, in general, there is no algorithm to decide whether the elements of a type are equal or not.

Examples. We *can check* equality of booleans, integers, strings and much more.

Counter-example. We *can't check* equality of functions of type `ℕ → ℕ`, for instance. Intuitively, to check that two functions `f` and `g` of this type are equal, we need to check infinitely many cases, namely `f x = g x` for all `x : ℕ`. But, we are afraid, intuition is not enough. This has to be proved. Luckily in our case, Alan Turing provided the basis to prove that. He showed that the Halting Problem can't be solved by an algorithm in any programming language. It follows from this that we can't check whether two such functions `f` and `g` are equal or not using an algorithm.

The above examples and counter-examples show that sometimes we can decide equality with an algorithm, and sometimes we can't. However, for example, the identity type `_≡_` applies to *all* types, whether they have decidable equality or not, and this is why it is useful. We can think about equality, not only in our heads but also in Agda, without worrying whether it can be *checked* to be true or not by a computer. The identity type is not about *checking* equality. It is about asserting that two things are equal, and then proving this ourselves. In fact, equality is very often not checkable by the computer. It is instead about *stating* and *proving* or *disproving* equalities, where the proving and disproving is done by people (the lecturers and the students in this case), by hard, intelligent work.

Decidable propositions

Motivated by the above discussion, we define when a logical proposition is decidable under the understanding of propositions as types:

```
is-decidable : Type → Type
is-decidable A = A ∔ ¬ A
```

This means that we can produce an element of `A` or show that no such element can be found.

Although it is not possible in general to write a program of type `¬¬ A → A`, this is possible when `A` is decidable:

```
¬¬-elim : {A : Type} → is-decidable A → ¬¬ A → A
¬¬-elim (inl x) f = x
¬¬-elim (inr g) f = 𝟘-elim (f g)
```

Decidable propositions as booleans

The following shows that a type `A` is decidable if and only if there is `b : Bool` such that `A` holds if and only if the boolean `b` is `true`.

For the purposes of this handout, understanding the following proof is not important at a first reading. What is important is to understand *what* the type of the following function is saying, which is what we explained above.

```
decidability-with-booleans : (A : Type) → is-decidable A ⇔ Σ b : Bool , (A ⇔ b ≡ true)
decidability-with-booleans A = f , g
 where
  f : is-decidable A → Σ b : Bool , (A ⇔ b ≡ true)
  f (inl x) = true , (α , β)
   where
    α : A → true ≡ true
    α _ = refl true

    β : true ≡ true → A
    β _ = x
  f (inr ν) = false , (α , β)
   where
    α : A → false ≡ true
    α x = 𝟘-elim (ν x)

    β : false ≡ true → A
    β ()

  g : (Σ b : Bool , (A ⇔ b ≡ true)) → is-decidable A
  g (true ,  α , β) = inl (β (refl true))
  g (false , α , β) = inr (λ x → false-is-not-true (α x))
```

Decidable predicates as boolean-valued functions

Consider the logical statement "x is even". This is decidable, because there is an easy algorithm that tells whether a natural number `x` is even or not. In programming languages we write this algorithm as a procedure that returns a boolean. But an equally valid definition is to say that `x` is even if there is a natural number `y` such that `x = 2 * y`. This definition doesn't automatically give an algorithm to check whether or not `x` is odd.

```
  is-even : ℕ → Type
  is-even x = Σ y : ℕ , x ≡ 2 * y
```

This says what to be even *means*. But it doesn't say how we *check* with a computer program whether a number is even or not, which would be given by a function `check-even : ℕ → Bool`.

```
  check-even : ℕ → Bool
  check-even 0             = true
  check-even 1             = false
  check-even (suc (suc x)) = check-even x
```

For this function to be correct, it has to be the case that

```
  is-even x ⇔ check-even x ≡ true
```

Exercise. We you have learned enough Agda, try this.

This is possible because

```
  (x : X) → is-decidable (is-even x) .
```

The following generalizes the above discussion and implements it in Agda.

First we define what it means for a predicate, such as `A = is-even`, to be decidable:

```
is-decidable-predicate : {X : Type} → (X → Type) → Type
is-decidable-predicate {X} A = (x : X) → is-decidable (A x)
```

In our example, this means that we can tell whether a number is even or not.

Next we show that a predicate `A` is decidable if and only if there is a boolean valued function `α` such that `A x` holds if and only if `α x ≡ true`. In the above example, `A` plays the role of `is-even` and `alpha` plays the role of `check-even`.

Again, what is important at a first reading of this handout is not to understand the proof but what the type of the function is saying. But we will discuss the proof in lectures.

```
predicate-decidability-with-booleans : {X : Type} (A : X → Type)
                                     → is-decidable-predicate A
                                     ⇔ Σ α : (X → Bool) , ((x : X) → A x ⇔ α x ≡ true)
predicate-decidability-with-booleans {X} A = f , g
 where
  f : is-decidable-predicate A → Σ α : (X → Bool) , ((x : X) → A x ⇔ α x ≡ true)
  f d = α , β
   where
    α : X → Bool
    α x = fst (lr-implication I (d x))
     where
      I : is-decidable (A x) ⇔ Σ b : Bool , (A x ⇔ b ≡ true)
      I = decidability-with-booleans (A x)
    β : (x : X) → A x ⇔ α x ≡ true
    β x = ϕ , γ
     where
      I : is-decidable (A x) → Σ b : Bool , (A x ⇔ b ≡ true)
      I = lr-implication (decidability-with-booleans (A x))

      II : Σ b : Bool , (A x ⇔ b ≡ true)
      II = I (d x)

      ϕ : A x → α x ≡ true
      ϕ = lr-implication (snd II)

      γ : α x ≡ true → A x
      γ = rl-implication (snd II)

  g : (Σ α : (X → Bool) , ((x : X) → A x ⇔ α x ≡ true)) → is-decidable-predicate A
  g (α , ϕ) = d
   where
    d : is-decidable-predicate A
    d x = III
     where
      I : (Σ b : Bool , (A x ⇔ b ≡ true)) → is-decidable (A x)
      I = rl-implication (decidability-with-booleans (A x))

      II : Σ b : Bool , (A x ⇔ b ≡ true)
      II = (α x , ϕ x)

      III : is-decidable (A x)
      III = I II
```

Although boolean-valued predicates are fine, we prefer to use type-valued predicates for the sake of uniformity, because we can always define type valued predicates, but only on special circumstances can we define boolean-valued predicates. It is better to define all predicates in the same way, and then write Agda code for predicates that happen to be decidable.

Preservation of decidability

If `A` and `B` are logically equivalent, then `A` is decidable if and only if `B` is decidable. We first prove one direction.

```
map-decidable : {A B : Type} → (A → B) → (B → A) → is-decidable A → is-decidable B
map-decidable f g (inl x) = inl (f x)
map-decidable f g (inr h) = inr (λ y → h (g y))

map-decidable-corollary : {A B : Type} → (A ⇔ B) → (is-decidable A ⇔ is-decidable B)
map-decidable-corollary (f , g) = map-decidable f g , map-decidable g f
```

Variation:

```
map-decidable' : {A B : Type} → (A → ¬ B) → (¬ A → B) → is-decidable A → is-decidable B
map-decidable' f g (inl x) = inr (f x)
map-decidable' f g (inr h) = inl (g h)
```

```
pointed-types-are-decidable : {A : Type} → A → is-decidable A
pointed-types-are-decidable = inl

empty-types-are-decidable : {A : Type} → ¬ A → is-decidable A
empty-types-are-decidable = inr

𝟙-is-decidable : is-decidable 𝟙
𝟙-is-decidable = pointed-types-are-decidable ⋆

𝟘-is-decidable : is-decidable 𝟘
𝟘-is-decidable = empty-types-are-decidable 𝟘-is-empty

∔-preserves-decidability : {A B : Type}
                         → is-decidable A
                         → is-decidable B
                         → is-decidable (A ∔ B)
∔-preserves-decidability (inl x) _       = inl (inl x)
```

```
∔-preserves-decidability (inr _) (inl y) = inl (inr y)
∔-preserves-decidability (inr h) (inr k) = inr (∔-nondep-elim h k)

×-preserves-decidability : {A B : Type}
                         → is-decidable A
                         → is-decidable B
                         → is-decidable (A × B)
×-preserves-decidability (inl x) (inl y) = inl (x , y)
×-preserves-decidability (inl _) (inr k) = inr (λ (x , y) → k y)
×-preserves-decidability (inr h) _       = inr (λ (x , y) → h x)

→-preserves-decidability : {A B : Type}
                         → is-decidable A
                         → is-decidable B
                         → is-decidable (A → B)
→-preserves-decidability _       (inl y) = inl (λ _ → y)
→-preserves-decidability (inl x) (inr k) = inr (λ f → k (f x))
→-preserves-decidability (inr h) (inr k) = inl (λ x → 𝟘-elim (h x))

¬-preserves-decidability : {A : Type}
                         → is-decidable A
                         → is-decidable (¬ A)
¬-preserves-decidability d = →-preserves-decidability d 𝟘-is-decidable
```

Exhaustively searchable types

We will explain in a future lecture why we need to use `Type₁` rather than `Type` in the following definition. For the moment we just mention that because the definition mentions `Type`, there would be a circularity if the type of the definition where again `Type`. Such circular definitions are not allowed because if they were then we would be able to prove that `0=1`. We have that `Type : Type₁` (the type of `Type` is `Type₁`) but we can't have `Type : Type`.

```
is-exhaustively-searchable : Type → Type₁
is-exhaustively-searchable X = (A : X → Type)
                             → is-decidable-predicate A
                             → is-decidable (Σ x : X , A x)
```

Exercise. Show, in Agda, that the types `𝟘`, `𝟙`, `Bool` and `Fin n`, for any `n : ℕ`, are exhaustively searchable. The idea is that we check whether or not `A x` holds for each `x : A`, and if we find at least one, we conclude that `Σ x : X , A x`, and otherwise we conclude that `¬ (Σ x : X , A x)`. This is possible because these types are finite.

Exercise. Think why there can't be any program of type `is-exhaustively-searchable ℕ`, by reduction to the Halting Problem. No Agda code is asked in this question. In fact, the question is asking you to think why such Agda code can't exist. This is very different from proving, in Agda, that `¬ is-exhaustively-searchable ℕ`. Interestingly, this is also not provable in Agda, but explaining why this is the case is beyond the scope of this module. In any case, this is an example of a statement `A` such that neither `A` nor `¬ A` are provable in Agda. Such statements are called *independent*. It must be remarked that the reason why there isn't an Agda program of type `is-exhaustively-searchable ℕ` is *not* merely that `ℕ` is infinite, because there are, perhaps surprisingly, infinite types `A` such that a program of type `is-exhastively-searchable A` can be coded in Agda. One really does an argument such as reduction to the Halting Problem to show that there is no program that can exaustively search the set `ℕ` of natural numbers.

```
Π-exhaustibility : (X : Type)
                 → is-exhaustively-searchable X
                 → (A : X → Type)
                 → is-decidable-predicate A
                 → is-decidable (Π x : X , A x)
Π-exhaustibility X s A d = VI
 where
  I : is-decidable-predicate (λ x → ¬ (A x))
  I x = ¬-preserves-decidability (d x)

  II : is-decidable (Σ x : X , ¬ (A x))
  II = s (λ x → ¬ (A x)) I

  III : (Σ x : X , ¬ (A x)) → ¬ (Π x : X , A x)
  III (x , f) g = f (g x)

  IV : ¬ (Σ x : X , ¬ (A x)) → (Π x : X , A x)
  IV h x = ii
   where
    i : ¬¬ A x
    i f = h (x , f)
    ii : A x
    ii = ¬¬-elim (d x) i

  V : is-decidable (Σ x : X , ¬ (A x)) → is-decidable (Π x : X , A x)
  V = map-decidable' III IV

  VI : is-decidable (Π x : X , A x)
  VI = V II
```

Exercises. If two types `A` and `B` are exhaustively searchable types, then so are the types `A × B` and `A + B`. Moreover, if `X` is an exhaustively searchable type and `A : X → Type` is a family of types, and the type `A x` is exhaustively searchable for each `x : X`, then the type `Σ x : X , A x` is exhaustively searchable.

Decidable equality

A particular case of interest regarding the above discussion is the notion of a type having decidable equality, which can be written in Agda as follows.

```
has-decidable-equality : Type → Type
has-decidable-equality X = (x y : X) → is-decidable (x ≡ y)
```

Exercise. Show, in Agda, that a type `X` has decidable equality if and only if there is a function `X → X → Bool` that checks whether two elements of `X` are equal or not.

Some examples:

```
Bool-has-decidable-equality : has-decidable-equality Bool
Bool-has-decidable-equality true  true  = inl (refl true)
Bool-has-decidable-equality true  false = inr true-is-not-false
Bool-has-decidable-equality false true  = inr false-is-not-true
Bool-has-decidable-equality false false = inl (refl false)

open import natural-numbers-functions

ℕ-has-decidable-equality : has-decidable-equality ℕ
ℕ-has-decidable-equality 0       0       = inl (refl zero)
ℕ-has-decidable-equality 0       (suc y) = inr zero-is-not-suc
ℕ-has-decidable-equality (suc x) 0       = inr suc-is-not-zero
ℕ-has-decidable-equality (suc x) (suc y) = III
 where
  IH : is-decidable (x ≡ y)
  IH = ℕ-has-decidable-equality x y

  I : x ≡ y → suc x ≡ suc y
  I = ap suc

  II : suc x ≡ suc y → x ≡ y
  II = suc-is-injective

  III : is-decidable (suc x ≡ suc y)
  III = map-decidable I II IH
```

Propositions as types and basic Martin-Löf type theory

We now complete the proposition-as-types interpretation of logic.

Logic	English	Type theory	Agda	Handouts	Alternative terminology
⊥	false	$\mathbb{N}_0$	𝟘	empty type	
⊤	true (*)	$\mathbb{N}_1$	𝟙	unit type	
A ∧ B	A and B	A × B	A × B	binary product	cartesian product
A ∨ B	A or B	A + B	A ∔ B	binary sum	coproduct, binary disjoint union
A → B	A implies B	A → B	A → B	function type	non-dependent function type
¬ A	not A	A → $\mathbb{N}_0$	A → 𝟘	negation	
∀ x : A, B x	for all x:A, B x	Π x : A , B x	(x : A) → B x	product	dependent function type
∃ x : A, B x	there is x:A such that B x	Σ x : A , B x	Σ x ꞉ A , B x	sum	disjoint union, dependent pair type
x = y	x equals y	Id x y	x ≡ y	identity type	equality type, propositional equality

Remarks

- (*) Not only the unit type 𝟙, but also any type with at least one element can be regarded as "true" in the propositions-as-types interpretation.
- In Agda we can use any notation we like, of course, and people do use slightly different notatations for e.g. `𝟘` , `𝟏` , `+` and `Σ` .
- We will refer to the above type theory together with the type `ℕ` of natural numbers as *Basic Martin-Löf Type Theory*.
- As we will see, this type theory is very expressive and allows us to construct rather sophisticated types, including e.g. lists, vectors and trees.
- All types in MLTT (Martin-Löf type theory) come with *introduction* and *elimination* principles.

The empty type 𝟘

It is convenient to have an empty type `𝟘`, with no elements at all. For example, this can be used in order to define negation, among other things.

```
data 𝟘 : Type where
```

And that's the complete definition. The list of constructors that define the type `𝟘` is empty.

Perhaps counter-intuitively, there is one function `𝟘 → 𝟘`, namely the identity function. So although the type `𝟘` is empty, the type `𝟘 → 𝟘` is non-empty. In fact, the non-dependent elimination principle generalizes that.

Proposition as type interpretation

The empty type is used to interpret "false". Because there is no way to prove the statement `false`, we use the empty type to represent `false` as a type.

In logic, in order to prove that a proposition is false, we assume it is true and use this assumption to reach a contradiction, such as `0 = 1`. With proofs as programs, in order to show that a statement represented by a type `A` is false, we assume a hypothetical element `x : A`, and from this we try to build a (necessarily impossible) element of the type `𝟘`, which is the desired contradiction. Because of this, in logic the negation of `A` is defined as `A implies false` or `A implies a contradiction`. Hence in type theory we define the negation of a type `A` to be the function type `A → 𝟘`:

```
¬_ : Type → Type
¬ A = A → 𝟘

infix 1000 ¬_
```

Elimination principle

```
𝟘-elim : {A : 𝟘 → Type} (x : 𝟘) → A x
𝟘-elim ()
```

The absurd pattern `()` expresses the fact that there is no pattern available because the type is empty. So in the same way that we define the type by giving an empty list of constructors, we define the function `𝟘-elim` by giving an empty list of equations. But we indicate this explicitly with the absurd pattern.

In terms of logic, this says that in order to show that a property `A` of elements of the empty type holds for all `x : 𝟘`, we have to do nothing, because there is no element to check, and by doing nothing we exhaust all possibilities. This is called vacuous truth.

It is important to notice that this is not a mere technicality. We'll see practical examples in due course.

The non-dependent version of the eliminator says that there is a function from the empty type to any type:

```
𝟘-nondep-elim : {A : Type} → 𝟘 → A
𝟘-nondep-elim {A} = 𝟘-elim {λ _ → A}
```

Definition of emptiness

On the other hand, there is a function `f : A → 𝟘` if and only if `A` has no elements, that is, if `A` is also empty. This is because if `x : A`, there is no element `y : 𝟘` we can choose in order to define `f x` to be `y`. In fact, we make this observation into our definition of emptiness:

```
is-empty : Type → Type
is-empty A = A → 𝟘
```

So notice that this is the same definition as that of negation.

Here is another example of a type that is empty. In the introduction we defined the identity type former `_≡_`, which we will revisit, and we have that, for example, the type `3 ≡ 4` is empty, whereas the type `3 ≡ 3` has an element `refl 3`. Here are some examples coded in Agda:

```
𝟘-is-empty : is-empty 𝟘
```

```
𝟘-is-empty = 𝟘-nondep-elim

open import unit-type

𝟙-is-nonempty : ¬ is-empty 𝟙
𝟙-is-nonempty f = f ⋆
```

The last function works as follows. First we unfold the definition of `¬ is-empty 𝟙` to get `is-empty 𝟙 → 𝟘`. Unfolding again, we get the type `(𝟙 → 𝟘) → 𝟘`. So, given a hypothetical function `f : 𝟙 → 𝟘`, which of course cannot exist (and this what we are trying to conclude), we need to produce an element of `𝟘`. We do this by simply applying the mythical `f` to `⋆ : 𝟙`. We can actually incorporate this discussion in the Agda code, if we want:

```
𝟙-is-nonempty' : ¬ is-empty 𝟙
𝟙-is-nonempty' = γ
 where
  γ : (𝟙 → 𝟘) → 𝟘
  γ f = f ⋆
```

Agda accepts this second version because it automatically unfolds definitions, just as we have done above, to check whether what we have written makes sense. In this case, Agda knows that `¬ is-empty 𝟙` is exactly the same thing as `(𝟙 → 𝟘) → 𝟘` *by definition* of `¬` and `is-empty`. More examples are given in the file negation.

The unit type 𝟙

We now redefine the unit type as a record:

```
record 𝟙 : Type where
 constructor
  ⋆

open 𝟙 public
```

In logical terms, we can interpret `𝟙` as "true", because we can always exhibit an element of it, namely `⋆`. Its elimination principle is as follows:

```
𝟙-elim : {A : 𝟙 → Type}
       → A ⋆
       → (x : 𝟙) → A x
𝟙-elim a ⋆ = a
```

In logical terms, this says that it order to prove that a property `A` of elements of the unit type `𝟙` holds for all elements of the type `𝟙`, it is enough to prove that it holds for the element `⋆`. The non-dependent version says that if A holds, then "true implies A".

```
𝟙-nondep-elim : {A : Type}
              → A
              → 𝟙 → A
𝟙-nondep-elim {A} = 𝟙-elim {λ _ → A}
```

add comment to binary-types
mhe authored 7 minutes ago

34e4c7f1

binary-type.lagda.md 1.41 KB

The binary type 𝟚

This type can be defined to be `𝟙 ∔ 𝟙` using binary sums, but we give a direct definition which will allow us to discuss some relationships between the various type formers of Basic MLTT.

```
data 𝟚 : Type where
 𝟎 𝟏 : 𝟚
```

This type is not only isomorphic to `𝟙 ∔ 𝟙` but also to the type `Bool` of booleans. Its elimination principle is as follows:

```
𝟚-elim : {A : 𝟚 → Type}
       → A 𝟎
       → A 𝟏
       → (x : 𝟚) → A x
𝟚-elim x₀ x₁ 𝟎 = x₀
𝟚-elim x₀ x₁ 𝟏 = x₁
```

In logical terms, this says that it order to show that a property `A` of elements of the binary type `𝟚` holds for all elements of the type `𝟚`, it is enough to show that it holds for `𝟎` and for `𝟏`. The non-dependent version of the eliminator is the following:

```
𝟚-nondep-elim : {A : Type}
              → A
              → A
              → 𝟚 → A
𝟚-nondep-elim {A} = 𝟚-elim {λ _ → A}
```

Notice that the non-dependent version is like if-then-else, if we think of one of the elements of `A` as `true` and the other as `false.

The dependent version generalizes the *type* of the non-dependent version, with the same definition of the function.

Products

We discuss function types of the form `A → B` (called non-dependent function types) and of the form `(x : A) → B x` (called dependent function types). The latter are also called *products* in type theory and this terminology comes from mathematics.

The identity function

To *introduce* functions, we use `λ` -abstractions like in Haskell. For example, the identity function can be defined as follows:

```
  id : {A : Type} → A → A
  id = λ x → x
```

But of course this function can be equivalently written as follows, which we take as our official definition:

```
id : {A : Type} → A → A
id x = x
```

Logical implication

A logical implication `A → B` is represented by the function type `A → B`, and it is a happy coincidence that both use the same notation.

In terms of logic, the identity function implements the tautology "`A` implies `A`" when we understand the type `A` as a logical proposition. In general, most things we define in Agda have a dual interpretation type/proposition or program/proof, like this example.

The *elimination* principle for function types is given by function application, which is built-in in Agda, but we can explicitly (re)define it. We do with the arguments swapped here:

```
modus-ponens : {A B : Type} → A → (A → B) → B
modus-ponens a f = f a
```

Modus ponens is the rule logic that says that if the proposition `A` holds and `A` implies `B`, then `B` also holds. The above definition gives a computational realization of this.

Dependent functions

As already discussed, a dependent function type `(x : A) → B x`, with `A : Type` and `B : A → Type`, is that of functions with input `x` in the type `A` and output in the type `B x`. It is called "dependent" because the output *type* `B x` depends on the input *element* `x : A`.

Universal quantification

The logical interpretation of `(x : A) → B x` is "for all x : A, B x". This is because in order to show that this universal quantification holds, we have to show that `B x` holds for each given `x:A`. The input is the given `x : A`, and the output is a justification of `B x`. We will see some concrete examples shortly.

Dependent function composition

Composition can be defined for "non-dependent functions", as in usual mathematics and in Haskell, and, more generally, with dependent functions. With non-dependent functions, it has the following type and definition:

```
  _∘_ : {A B C : Type} → (B → C) → (A → B) → (A → C)
  g ∘ f = λ x → g (f x)
```

In terms of computation, this means "first do f and then do g". For this reason the function composition `g ∘ f` is often read "`g` after `f`". In terms of logic, this implements "If B implies C and also A implies B, then A implies C".

With dependent types, it has the following more general type but the same definition, which is the one we adopt:

```
_∘_ : {A B : Type} {C : B → Type}
    → ((y : B) → C y)
    → (f : A → B)
    → (x : A) → C (f x)
g ∘ f = λ x → g (f x)
```

Its computational interpretation is the same, "first do f and then do g", but its logical understanding changes: "If it is the case that for all y : B we have that C y holds, and we have a function f : A → B, then it is also the case that for all x : A, we have that C (f x) holds".

Π notation

We have mentioned in the propositions as types table that the official notation in MLTT for the dependent function type uses `Π`, the Greek letter Pi, for *product*. We can, if we want, introduce the same notation in Agda, using a `syntax` declaration:

```
Pi : (A : Type) (B : A → Type) → Type
Pi A B = (x : A) → B x

syntax Pi A (λ x → b) = Π x ꞉ A , b
```

Important. We are using the alternative symbol `꞉` (typed "`\:4`" in the emacs mode for Agda), which is different from the reserved symbol "`:`". We will use the same alternative symbol when we define syntax for the sum type `Σ`.

Notice that, for some reason, Agda has this kind of definition backwards. The end result of this is that now `(x : A) → B x` can be written as `Π x ꞉ A , B x` in Agda as in type theory. (If you happen to know a bit of maths, you may be familiar with the cartesian product of a family of sets, and this is the reason the letter `Π` is used.)

The sum type former Σ

Very often in computation we work with the type of pairs `(x , y)` with `x : A` and `y : B` where `A` and `B` are types. We will write `A × B` for the type of such pairs, and call it the *cartesian product*. We will define this type as a particular case of a more general type, whose elements are again of the form `(x , y)` but with the difference that `x : A` and `y : B x` where `A : Type` and `B : A → Type`. The difference amounts to the fact that the type of the second component `y` depends on the first element `x`. The default notation for this type will be `Σ {A} B`, or simply `Σ B` when `A` can be inferred from the context, but we will also introduce the more common sum notation `Σ x : A , B x`. This type is also called the disjoint union of the type family `B : A → Type`.

Examples

A simple example is the type `Σ xs : List X , Vector X (length xs)` with `X : Type`. An element of this type is a pair `(xs , ys)` where `xs` is a list and `ys` is a vector of the same length as `xs`.

Another example, which iterates the `Σ` type construction, is `Σ x : ℕ , Σ y : ℕ , Σ z : ℕ , x ≡ y * z`. An element of this type is now a quadruple `(x , (y , (z , p)))` where `x y z : ℕ` and `p : x ≡ y * z`. That is, the fourth component ensures that `x y z : ℕ` are allowed in the tuple if, and only if, `x ≡ y * z`. We will see more interesting examples shortly.

Definition

The `Σ` type can be defined in Agda using a `data` declaration as follows:

```
  data Σ {A : Type } (B : A → Type) : Type  where
   _,_ : (x : A) (y : B x) → Σ {A} B

  fst : {A : Type} {B : A → Type} → Σ B → A
  fst (x , y) = x

  snd : {A : Type} {B : A → Type} → (z : Σ B) → B (fst z)
  snd (x , y) = y
```

Notice that the type of `snd` is dependent and uses `fst` to express the dependency.

However, for a number of reasons to be explained later, we prefer to define it using a record definition:

```
record Σ {A : Type } (B : A → Type) : Type  where
 constructor
  _,_
 field
  fst : A
  snd : B fst
```

Here we automatically get the projections with the same types and definitions as above and hence we don't need to provide them. In order for the projections `fst` and `snd` to be visible outside the scope of the record, we `open` the record. Moreover, we open it `public` so that when other files import this one, these two projections will be visible in the other files. The "constructor" allows to form an element of this type. Because "," is not a reserved symbol in Agda, we can use it as a binary operator to write `x , y`. However, following mathematical tradition, we will write brackets around that, to get `(x , y)`, even if this is not necessary. We also declare a fixity and precedence for this operator.

```
open Σ public

infixr 50 _,_
```

Because we make `_,_` right associative, we can write `(x , y , z , p)` rather than `(x , (y , (z , p)))` as we did above.

We also use a syntax declaration, as we did for dependent function types using Π, to get the more traditional type-theoretical notation.

```
Sigma : (A : Type) (B : A → Type) → Type
Sigma A B = Σ {A} B
```

```
syntax Sigma A (λ x → b) = Σ x : A , b

infix -1 Sigma
```

Elimination principle

We now define and discuss the elimination principle.

```
Σ-elim : {A : Type } {B : A → Type} {C : (Σ x : A , B x) → Type}
       → ((x : A) (y : B x) → C (x , y))
       → (z : Σ x : A , B x) → C z
Σ-elim f (x , y) = f x y
```

So the elimination principle for `Σ` is what was called `curry` in Haskell in its non-dependent form. The logical interpretation for this principle is that in order to show that "for all z : Σ x : A , B x) we have that C z holds", it is enough to show that "for all x : A and y : B x we have that C (x , y) holds". This condition is not only sufficient but also necessary:

```
Σ-uncurry : {A : Type } {B : A → Type} {C : (Σ x : A , B x) → Type}
          → ((z : Σ x : A , B x) → C z)
          → (x : A) (y : B x) → C (x , y)
Σ-uncurry g x y = g (x , y)
```

Existential quantification

Regarding logic, the `Σ` type is used to interpret the existential quantifier `∃`. The logical proposition `∃ x : X, A x`, that is, "there exists x : X such that A x", is interpreted as the type `Σ x : X , A x`. The reason is that to show that `∃ x : X, A x` we have to exhibit an example `x : X` and show that `x` satisfies the condition `A x` with some `y : A x`, in a pair `(x , y)`.

For example, the type `Σ x : ℕ , Σ y : ℕ , Σ z : ℕ , x ≡ y * z` can be interpreted as saying that "there are natural numbers x, y, and z such that x = y * z", which is true as witnessed by the element `(6,2,3,refl 6)` of that type. But there are many other witnesses of this type, of course, such as `(10,5,2,refl 10)`.

It is important to notice that it is possible to write types that correspond to false logical statements, and hence are empty. For example, consider `Σ x : ℕ , x ≡ x + 1`. There is no natural number that is its own successor, of course, and so this type is empty. While this type is empty, the type `¬ (Σ x : ℕ , x ≡ x + 1)` has an element, as we will see, which witnesses the fact that "there doesn't exist a natural number `x` such that x = x + 1`.

The cartesian-product type former `_×_`

As discussed before, the cartesian product `A × B` is simply `Σ {A} (λ x → B)` which means that for `(x , y) : A × B` the type of `y` is always `B`, independently of what `x` is. Using our special syntax for `Σ` this can be defined as follows in Agda:

```
open import sums public

_×_ : Type → Type → Type
A × B = Σ x : A , B

infixr 2 _×_
```

So the elements of `A × B` are of the form `(x , y)` with `x : A` and `y : B`.

Logical conjunction ("and")

The logical interpretation of `A × B` is "A and B". This is because in order to show that the proposition "A and B" holds, we have to show that each of A and B holds. To show that A holds we exhibit an element `x` of A, and to show that B holds we exhibit an element `y` of B, and so to show that "A and B" holds we exhibit a pair `(x , y)` with `x : A` and `b : B`

Logical equivalence

We now can define bi-implication, or logical equivalence, as follows:

```
_⇔_ : Type → Type → Type
A ⇔ B = (A → B) × (B → A)

infix -2 _⇔_
```

The symbol `⇔` is often pronounced "if and only if".

We use the first and second projections to extract the left-to-right implication and the right-to-left implication:

```
lr-implication : {A B : Type} → (A ⇔ B) → (A → B)
lr-implication = fst

rl-implication : {A B : Type} → (A ⇔ B) → (B → A)
rl-implication = snd
```

Alternative definition of `_×_`

There is another way to define binary products as a special case of `Π` rather than `Σ`.

The binary-sum type former `_∔_`

This is the same as (or, more precisely, isomorphic to) the `Either` type defined earlier (you can try this as an exercise). The notation in type theory is `_+_`, but we want to reserve this for addition of natural numbers, and hence we use the same symbol with a dot on top:

```
data _∔_ (A B : Type) : Type where
 inl : A → A ∔ B
 inr : B → A ∔ B

infixr 20 _∔_
```

The type `A ∔ B` is called the coproduct of `A` and `B`, or the sum of `A` and `B`, or the disjoint union of `A` and `B`. The elements of `A ∔ B` are of the form `inl x` with `x : A` and `inr y` with `y : B`.

In terms of computation, we use the type `A ∔ B` when we want to put the two types together into a single type. It is also possible to write `A ∔ A`, in which case we will have two copies of the type `A`, so that now every element `x` of `A` has two different incarnations `inl a` and `inr a` in the type `A ∔ A`. For example, the unit type `𝟙` has exactly one element, namely `⋆ : 𝟙`, and hence the type `𝟙 ∔ 𝟙` has precisely two elements, namely `inl ⋆` and `inr ⋆`.

Logical disjunction ("or")

In terms of logic, we use the type `A ∔ B` to express "A or B". This is because in order for "A or B" to hold, at least one of A and B must hold. The type `A → A ∔ B` of the function `inl` is interpreted as saying that if A holds then so does "A or B", and similarly, the type of B → A ∔ B of the function `inr` says that if B holds then so does "A or B". In other words, if `x : A` is a proof of `A`, then `inl x : A + B` is a proof of `A or B`, and if `y : B` is a proof of B, them `inr y : A + B` is a proof of "A or B". Here when we said "proof" we meant "program" after the propositions-as-types and proofs-as-programs paradigm.

Elimination principle

Now suppose we want to define a dependent function `(z : A ∔ B) → C z`. How can we do that? If we have two functions `f : (x : A) → C (inl x)` and `g : (y : B) → C (inr y)`, then, given `z : A ∔ B`, we can inspect whether `z` is of the form `inl x` with `x : A` or of the form `inr y` with `y : B`, and the respectively apply `f` or `g` to get an element of `C z`. This procedure is called the *elimination* principle for the type former `_∔_` and can be written in Agda as follows:

```
∔-elim : {A B : Type} (C : A ∔ B → Type)
       → ((x : A) → C (inl x))
       → ((y : B) → C (inr y))
       → (z : A ∔ B) → C z
∔-elim C f g (inl x) = f x
∔-elim C f g (inr y) = g y
```

So the eliminator amounts to simply definition by cases. In terms of logic, it says that in order to show that "for all z : A ∔ B we have that C z holds" it is enough to show two things: (1) "for all x : A it is the case that C (inl x) holds", and (2) "forall y : B it is the case that C (inr y) holds". This is not only sufficient, but also necessary:

```
open import binary-products

∔-split : {A B : Type} (C : A ∔ B → Type)
        → ((z : A ∔ B) → C z)
        → ((x : A) → C (inl x)) × ((y : B) → C (inr y))
∔-split {A} {B} C h = f , g
 where
  f : (x : A) → C (inl x)
  f x = h (inl x)

  g : (y : B) → C (inr y)
  g y = h (inr y)
```

There is also a version of the eliminator in which `C` doesn't depend on `z : A ∔ B` and is always the same:

```
∔-nondep-elim : {A B C : Type}
              → (A → C)
              → (B → C)
              → (A ∔ B → C)
∔-nondep-elim {A} {B} {C} = ∔-elim (λ z → C)
```

In terms of logic, this means that in order to show that "A or B implies C", it is enough to show that both "A implies C" and "B implies C". This also can be inverted:

```
∔-nondep-split : {A B C : Type}
               → (A ∔ B → C)
               → (A → C) × (B → C)
∔-nondep-split {A} {B} {C} = ∔-split (λ z → C)
```

In terms of logic, this means that if `A or B implies C` then both `A implies C` and `B implies C`.

Alternative definition of `_∔_`

There is another way to define binary sums as a special case of `Σ`.

The identity type former _≡_

The original and main terminology for the following type is *identity type*, but sometimes it is also called the *equality type*. Sometimes this is also called *propositional equality*, but we will avoid this terminology as it sometimes leads to confusion.

```
data _≡_ {A : Type} : A → A → Type where
 refl : (x : A) → x ≡ x

infix 0 _≡_
```

Elimination principle

The elimination principle for this type is defined as follows:

```
≡-elim : {X : Type} (A : (x y : X) → x ≡ y → Type)
       → ((x : X) → A x x (refl x))
       → (x y : X) (p : x ≡ y) → A x y p
≡-elim A f x x (refl x) = f x
```

This says that in order to show that `A x y p` holds for all `x y : X` and `p : x ≡ y`, it is enough to show that `A x x (refl x)` holds for all `x : X`. In the literature, this elimination principle is called `J`. Again, we are not going to use it explicitly, because we can use definitions by pattern matching on `refl`, just as we did for defining it.

We also have the non-dependent version of the eliminator:

```
≡-nondep-elim : {X : Type} (A : X → X → Type)
              → ((x : X) → A x x)
              → (x y : X) → x ≡ y → A x y
≡-nondep-elim A = ≡-elim (λ x y _ → A x y)
```

A property of two variables like `A` above is referred to as a *relation*. The assumption `(x : X) → A x x` says that the relation is reflexive. Then the non-dependent version of the principle says that the reflexive relation given by the identity type `_≡_` can always be mapped to any reflexive relation, or we may say that `_≡_` is the smallest reflexive relation on the type `X`.

Fundamental constructions with the identity type

As an exercise, you may try to rewrite the following definitions to use `≡-nondep-elim` instead of pattern matching on `refl`:

```
trans : {A : Type} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans p (refl y) = p

sym : {A : Type} {x y : A} → x ≡ y → y ≡ x
sym (refl x) = refl x

ap : {A B : Type} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
ap f (refl x) = refl (f x)

transport : {X : Type} (A : X → Type)
          → (x y : X) → x ≡ y → A x → A y
transport A x x (refl x) a = a
```

We have already seen the first three. In the literature, `ap` is often called `cong`. In logical terms, the last one, often called `subst` in the literature, says that if `x` is equal `y` and `A x` holds, then so does `A y`. That is, we can substitute equals for equals in logical statements.

Pointwise equality of functions

It is often convenient to work with *pointwise equality* of functions, defined as follows:

```
_∼_ : {A : Type} {B : A → Type} → ((x : A) → B x) → ((x : A) → B x) → Type
f ∼ g = ∀ x → f x ≡ g x

infix 0 _∼_
```

Unfortunately, it is not provable or disprovable in Agda that pointwise equal functions are equal, that is, that `f ∼ g` implies `f ≡ g` (but it is provable in Cubical Agda, which is outside the scope of these lecture notes). This principle is very useful and is called function extensionality.

Notation for equality reasoning

When writing `trans p q` we lose type information of the identifications `p : x ≡ y` and `q : y ≡ z`, which makes some definitions using `trans` hard to read. We now introduce notation to be able to write e.g.

```
x ≡⟨ p ⟩
```

```
y ≡⟨ q ⟩
```

```
z ≡⟨ r ⟩
```

```
t ∎
```

rather than the more unreadable `trans p (trans q r) : x ≡ t`.

```
_≡⟨_⟩_ : {X : Type} (x : X) {y z : X} → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ p ⟩ q = trans p q

_∎ : {X : Type} (x : X) → x ≡ x
x ∎ = refl x

infixr  0 _≡⟨_⟩_
infix   1 _∎
```

We'll see examples of uses of this in other handouts.