```
{-# OPTIONS -type-in-type #-}
module Type where
open import Agda.Primitive public
  renaming (Set to Type; Setω to Typeω; SSet to SType
           ;Level to Lvl;lzero to ℓ0; lsuc to ℓs)
  using    (_⊔_)

open import Data.Product using (∃; _×_; _,_) renaming (proj₁ to π₁; proj₂ to π₂)
open import Relation.Binary.PropositionalEquality as Eq using (_≡_; refl)
open Eq.≡-Reasoning using (_≡⟨⟩_; _∎)

data ⊥ : Type where
record ⊤ : Type where constructor ⋆
data 𝔹 : Type where 𝕗 : 𝔹; 𝕥 : 𝔹
_iff_ : Type → Type → Type
p iff q = (p → q) × (q → p)
```

In GHC Haskell, every normal type has the kind `Type` (previously `*`), including `Type :: Type`. Previously this behavior was turned on with `-XTypeInType`, but is now the default.

Agda's type hierarchy is more complex.

Instead of `Type : Type` we instead have `Type`$_0$ `:` `Type`$_1$ and `Type`$_1$ `:` `Type`$_2$ and so on.

Any type that quantifies over another type must be larger than it. For example:

```
record Dynamic : Type₁ where
  field
    Ty : Type₀
    term : Ty

Dynamic∀ : Type₀
Dynamic∀ = ∀ {r : Type₀} → (((Ty : Type₀) → Ty) → r) → r
```

Contrast this with the identity type, which is at the same level as `Ty` because it takes it as a generic parameter, rather than existentially quantifying over it.

```
record Identity (Ty : Type₀) : Type₀ where
  field term : Ty

Identity∀ : Type₀ → Type₀
Identity∀ Ty = Ty
```

It would be annoying to figure out which type level to use on our own, so types are polymorphic over *universe level*. `Type` alone is sugar for `Type`$_0$, which is itself sugar for `Type ℓ0`, and `Type`$_1$ is sugar for `Type (ℓs ℓ0)` etc. The full polymorphic type is `Type : (ℓ : Lvl) → Type (ℓs ℓ)`, for a special `Lvl` type:

```
- postulate Lvl :  Type
- {-# BUILTIN LEVEL Lvl #-}
- {-# BUILTIN LEVELZERO ℓ0 #-} - :  Lvl
- {-# BUILTIN LEVELSUC ℓs #-} - :  Lvl → Lvl
- {-# BUILTIN LEVELMAX _⊔_ #-} - :  Lvl → Lvl → Lvl
- infixl 6 _⊔_
```

`Lvl` is a magic type. It isn't possible to pattern match on levels, and they're erased at run-time.

When multiple polymorphic levels appear in a type, the resulting level must bound all of them. This is achieved with the max operator. e.g. the dependent pair:

```
record Σ {ℓa ℓb} (A : Type ℓa) (B : A → Type ℓb) : Type (ℓa ⊔ ℓb) where
  field
    fst : A
    snd : B fst
```

So universe levels create a nice hierarchy that statically guarantee we only have a finite depth of meta types, but things get weird when we try to quantify over level polymorphic types. What level should we give this type?

```
very-meta = (ℓ : Lvl) → Type ℓ
```

Russel's Paradox

To understand this better let's see what goes wrong in the classic example that inspired universe levels: Russel's Paradox

Russel's paradox relies on *comprehension*, which lets us construct a class by quantifying over sets satisfying a predicate. e.g.

$$\{x | x \in \mathbb{N}, \text{even}? x\}$$

We can encode this as:

```
module russel where
  data TYPE : Typeω where
    [_∋_] : ∀ {ℓ} (X : Type ℓ) → (pred : X → TYPE) → TYPE
```

Notice a comprehension isn't a normal `Type`, it's a proper class: (`TYPE`)! It won't type-check as Type without `--type-in-type`, because the resulting type wants its level to be `ℓs ℓ` for *every* ℓ - i.e. it wants an infinite level. Agda doesn't have an an infinite level ω, but it does have a `Typeω` which could be thought of as `Type ω` if one did exist. `Typeω` exists for precisely this reason - `TYPE` will type-check as `Typeω` even without `--type-in-type`. In general, `((ℓ : Lvl) → Type ℓ) : Typeω` and the only constructor that can be applied to expressions of kind `Typeω` is →.

We can encode a few example natural number `TYPE`s by quantifying over basic `Type`s.

```
∅ ⟨∅⟩ ⟨∅,⟨∅⟩⟩ : TYPE
∅ = [ ⊥ ∋ (λ()) ]          - 0
⟨∅⟩ = [ ⊤ ∋ (λ ⋆ → ∅) ] - 1
⟨∅,⟨∅⟩⟩ = [ 𝔹 ∋ (λ{𝕗 → ⟨∅⟩; 𝕥 → ∅}) ] - 2
```

The trouble comes when we try to examine the elements of a comprehension:

```
_∈_ : TYPE → TYPE → Type - This won't typecheck without -type-in-type
a ∈ [ X ∋ f ] = ∃ λ x → a ≡ f x

_∉_ : TYPE → TYPE → Type
a ∉ b = (a ∈ b) → ⊥
```

We can now encode Russell's paradoxical class directly as the `TYPE` of all `TYPE` that don't contain themselves:

```
Δ : TYPE
Δ = [ (∃ λ t → t ∉ t) ∋ (λ{(t , t∉t) → t}) ]
```

This does indeed capture the right notion we can prove that a class is in Δ if and only iff it doesn't contain itself:

```
x∈Δ→x∉x : ∀ {X} → X ∈ Δ → X ∉ X
x∈Δ→x∉x ((Y , Y∉Y) , refl) = Y∉Y

x∉x→x∈Δ : ∀ {X} → X ∉ X → X ∈ Δ
x∉x→x∈Δ {X} X∉X = (X , X∉X) , refl

x∈Δ↔x∉x : ∀ {X} → (X ∈ Δ) iff (X ∉ X)
x∈Δ↔x∉x = x∈Δ→x∉x , x∉x→x∈Δ
```

Now the infamous contradiction is direct: Δ both does and does not contain itself!

```
Δ∉Δ : Δ ∉ Δ
Δ∉Δ Δ∈Δ = (π₁ x∈Δ↔x∉x) Δ∈Δ Δ∈Δ
```

```
Δ∈Δ : Δ ∈ Δ
Δ∈Δ = (π₂ x∈Δ↔x∉x) Δ∉Δ

paradox : ⊥
paradox = Δ∉Δ Δ∈Δ

open import Data.Nat using (ℕ)
x : ℕ
x = 3
```

So how bad is this? What's actually going on?

If we're using Agda as a proof assistant, it's pretty terrible: we can now prove anything we like! This is why Agda uses such onerous level checking by default.

But Russel's paradox is quite weird, and one might wonder how easy it is to accidentally violate well-foundedness.

we can unroll `paradox` to see what's happening computationally:

```
- _ :  paradox ≡ paradox
- _ = paradox
- ≡⟨⟩ Δ∉Δ Δ∈Δ
- ≡⟨⟩ (x∈Δ→x∉x Δ∈Δ) Δ∈Δ
- ≡⟨⟩ (x∈Δ→x∉x (x∉x→x∈Δ Δ∉Δ)) Δ∈Δ
- ≡⟨⟩ (x∈Δ→x∉x ((Δ , Δ∉Δ) , refl)) Δ∈Δ
- ≡⟨⟩ Δ∉Δ Δ∈Δ ■
```

If you try to compile the above commented code the type-checker will hang, but we can see why: Δ∉Δ Δ∈Δ reduces to itself!

It turns out that *all* unsound uses of `--type-in-type` result in type-checking non-termination, so we'll never construct an ill-typed value at run-time.

For this reason we follow Haskell's example and include `--type-in-type` by default, to maintain clarity by reducing noise.

When checking a serious proof it's worth reintroducing `Levels` for consistency, but for interactive exploratory work or didactic communication it's nearly always a hindrance.