$T(b)$ is fully determined by two types, namely by the types $T(\mathrm{no})$ and $T(\mathrm{yes})$. The elements of $\sum_{b:\mathrm{Bool}} T(b)$ are dependent pairs (no, x) with x in $T(\mathrm{no})$ and (yes, y) with y in $T(\mathrm{yes})$. The resulting type can be viewed as the *disjoint union* of $T(\mathrm{no})$ and $T(\mathrm{yes})$: from an element of $T(\mathrm{no})$ or an element of $T(\mathrm{yes})$ we can produce an element of $\sum_{b:\mathrm{Bool}} T(b)$.

Such types can be described more clearly in the following way. The *binary sum* of two types X and Y, denoted $X \amalg Y$, is an inductive type with two constructors: $\mathrm{inl} : X \to X \amalg Y$ and $\mathrm{inr} : Y \to X \amalg Y$.[24] Proving a property of any element of $X \amalg Y$ means proving that this property holds of any inl_x with $x : X$ and any inr_y with $y : Y$. In general, constructing a function f of type $\prod_{z:X\amalg Y} T(z)$, where $T(z)$ is a type depending on z, is done by defining $f(\mathrm{inl}_x)$ for all x in X and $f(\mathrm{inr}_y)$ for all y in Y.

Identification of two elements a and b in $X \amalg Y$ is only possible if they are constructed with the same constructor. Thus $\mathrm{inl}_x = \mathrm{inr}_y$ is always empty, and identifications $\mathrm{inl}_x = \mathrm{inl}_{x'}$ are equivalent to identifications $x = x'$ in X, and identifications $\mathrm{inr}_y = \mathrm{inr}_{y'}$ are equivalent to identifications $y = y'$ in Y.

2.6.5 *Unary sums*

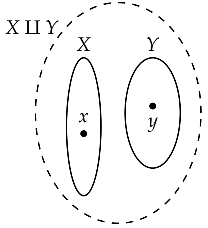
sec:unary-sum-types

Sometimes it is useful to be able to make a copy of a type X: A new type that behaves just like X, though it is not definitionally equal to X. The *unary sum* or *wrapped copy* of X is an inductive type $\mathrm{Copy}(X)$ with a single constructor $\mathrm{in} : X \to \mathrm{Copy}(X)$.[25] Constructing a function $f : \prod_{z:\mathrm{Copy}(X)} T(z)$, where $T(z)$ is a type depending on $z : \mathrm{Copy}(X)$, is done by defining $f(\mathrm{in}_x)$ for all $x : X$. Taking $T(z)$ to be the constant family at X, we get a function $\mathrm{out} : \mathrm{Copy}(X) \to X$, called the *destructor*, with $\mathrm{out}(\mathrm{in}_x) :\equiv x$ for $x : X$, and the induction principle implies that $\mathrm{in}_{\mathrm{out}(z)} = z$ for all $z : \mathrm{Copy}(X)$, so $\mathrm{Copy}(X)$ and X are equivalent, as expected. In fact, we will assume that the latter equation even holds definitionally. It follows that identifications $\mathrm{in}_x = \mathrm{in}_{x'}$ in $\mathrm{Copy}(X)$ are equivalent to identifications $x = x'$ in X, and identifications $\mathrm{out}\, z = \mathrm{out}\, z'$ in X are equivalent to identifications $z = z'$ in $\mathrm{Copy}(X)$.

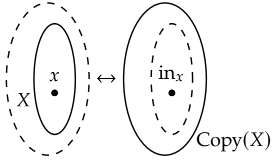
Here's an example to illustrate why it can useful to make such a wrapped type: We introduced the natural numbers $\mathbb{N}$ in Section 2.2. Suppose we want a type consisting of negations of naturals numbers, $\{\dots, -2, -1, 0\}$, perhaps as an intermediate step towards building the set of integers $\{\dots, -2, -1, 0, 1, 2, \dots\}$.[26] Of course, the type $\mathbb{N}$ itself would do, but then we would need to pay extra attention to whether $n : \mathbb{N}$ is supposed to represent n as an integer or its negation. So instead we take the wrapped copy $\mathbb{N}^- :\equiv \mathrm{Copy}(\mathbb{N})$ and write $- :\equiv \mathrm{in} : \mathbb{N} \to \mathbb{N}^-$ for the constructor. There is then no harm in also writing $- :\equiv \mathrm{out} : \mathbb{N}^- \to \mathbb{N}$ for the destructor. This means that $\mathbb{N}^-$ is a type equivalent to $\mathbb{N}$, whose elements are exactly $-n$ for $n : \mathbb{N}$, indeed, $-(-n) \equiv n$ for n an element of either $\mathbb{N}$ or $\mathbb{N}^-$, and identifications $-n = -n'$ are equivalent

One harm would be that
-n is an integer, so we
expect -(-n) to be one, too.

[24] Be aware that in a picture, the same point may refer either to x in X or to inl_x in the sum $X \amalg Y$:



[25] A point $x : X$ corresponds to the point $\mathrm{in}_x : \mathrm{Copy}(X)$:



Note that $\mathrm{Copy}(X)$ can also be described as $\sum_{z:\mathrm{True}} Y(z)$, where $Y(\mathrm{triv}) :\equiv X$.

Why not write X instead of Y(z)??

No, it's not a description, it's
an alternative.

[26] (FIX) Maybe we'll actually do just that!