```
discard = const id
```

is a function that discards one argument and returns the identity, `id`. This way of defining multiple argument functions is called the **currying** of a function in honor to the american logician Haskell Curry in [CF58]. It is a particular instance of a deeper fact: exponentials are defined by the following adjunction

$$\hom(A \times B, C) \cong \hom(A, \hom(B, C)).$$

9.2 A TECHNIQUE ON INDUCTIVE DATA ENCODING

We will implicitly use a technique on the majority of our data encodings that allows us to write an encoding for any algebraically inductive generated data. This technique is used without comment on [Sel13] and represents the basis of what is called the **Church encoding** of data in λ-calculus.

We start considering the usual inductive representation of a data type with constructors, as we do when representing a syntax with a BNF, for example,

$$\texttt{Nat ::= Zero} \mid \texttt{Succ Nat}.$$

Or, in general

$$\texttt{D ::=} \ C_1 \mid C_2 \mid C_3 \mid \dots$$

It is not possible to directly encode constructors on λ-calculus. Even if we were able, they would have, in theory, no computational content; the data structure would not be reduced under any λ-term, and we would need at least the ability to pattern match on the constructors to define functions on them. Our λ-calculus would need to be extended with additional syntax for every new data structure.

This technique, instead, defines a data term as a function on multiple arguments representing the missing constructors. In our example, the number 2, which would be written as `Succ(Succ(Zero))`, would be encoded as

$$2 = \lambda s.\ \lambda z.\ s(s(z)).$$

In general, any instance of the data structure `D` would be encoded as a λ-expression depending on all its constuctors

$$\lambda c_1.\ \lambda c_2.\ \lambda c_3.\ \dots\ \lambda c_n.(term).$$

This acts as the definition of an initial algebra over the constructors and lets us to compute over instances of that algebra by instantiating it on particular cases. Particular examples are described on the following sections.

9.3 BOOLEANS

Booleans can be defined as the data generated by a pair of constuctors

$$\texttt{Bool ::= True} \mid \texttt{False}.$$

Consequently, the Church encoding of booleans takes these constructors as arguments and defines

```
true  = \t.\f.t
false = \t.\f.f
```

Note that `true` and `const` are exactly the same term up to α-conversion. The same thing happens with `false` and `alwaysid`. The absence of types prevents us to make any effort to discriminate between these two uses of the same λ-term. Another side-effect of this definition is that our `true` and `false` terms can be interpreted as binary functions choosing between two arguments, that is,

- `true`$(a, b) = a$,
- `false`$(a, b) = b$.

We can test this interpretation on the interpreter to get

```
true id const
false id const
--- [1]: id
--- [2]: const
```

This inspires the definition of an `ifelse` combinator as the identity

```
ifelse = \b.b
(ifelse true) id const
(ifelse false) id const
--- [1]: id
--- [2]: const
```

The usual logic gates can be defined profiting from this interpretation of booleans

```
and = \p.\q.p q p
or = \p.\q.p p q
not = \b.b false true
xor = \a.\b.a (not b) b
implies = \p.\q.or (not p) q

xor true true
```