# 1 Emitting

C++ code, C++ lambda recursion.
Inline all lambdas vs bind to value (untyped $\lambda$-calculus).

# 2 IR as $\lambda$-calculus

OCaml ZINC

## 2.1 ADT

*MogensenScott encoding* Dana Scott.
*Church encoding*
*Boehm-Berarducci encoding*
Function as data (or rather parameters).

$$D = \{c_i\}_{i=1}^{N}, c_i \text{ has arity } A_i \tag{1}$$

$$\tag{2}$$

$x_k$ is the constructor value for field $k \in \{1 \ldots A_i\}$.
$c_i$ is constructor $i \in N$, effectively.

$$c_n : x_1 \to x_2 \cdots \to x_{A_i} \to D$$

Such that all constructors can be modelled as.

$$\lambda x_1 \ldots x_{A_i}.\lambda c_1 \ldots c_N.c_i x_1 \ldots x_{A_i}$$

The "data" is extracted by: Given a set of functions $\{f_1 \ldots f_N\}$ which each can handle union case $1 \ldots N$.

```
data Maybe a =
        | Nothing
        | Just a
```

Listing 1: Maybe in Haskell

Since we use untyped lambda calculus, we can model every parameter as *any* type. Furthermore, the code in Figure 1 can be expressed in scott encoding.

```haskell
newtype MaybeAlgebra =
    MaybeAlgebra{ unMaybe :: forall a b. ((a -> b) -> b -> b) }

just :: a -> MaybeAlgebra
just a = \onjust onnothing -> onjust a

nothing :: MaybeAlgebra
nothing = \onjust onnothing -> onnothing

...
(just 5) (\x -> x) (12)
```

Listing 2: Maybe in Haskell as catamorphism