# F-Algebras and Cedille

ERIC BOND

PURPL 2019

## Motivation

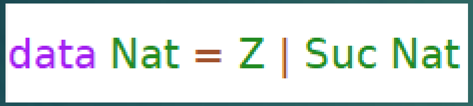
- What?
  - Representing inductive datatypes as functions in a generic and modular way.
- How?
  - Separate the 'shape' of a type from its inductive definition
    - Shape of a datatype represented as a functor
    - Use a single generic fixedpoint to generate inductive types from functors.
  - Define one step of a recursive function as an algebra
    - Use a single generic fold to generate recursive functions from algebras
- Why?
  - Modular extensible inductive datatypes as functions!
    - See Datatypes a la Carte
  - Modular extensible proofs!
    - See Meta-Theory a la Carte

## Outline

- F-Algebras
  - Definitions
  - Inductive datatype as least fixpoint of a functor
  - Functions on inductive datatypes as algebras
- Mendler style F-Algebras in Cedille
  - Lambda encoded inductive types
  - Functions with explicit control of recursive call

## An Old Friend

Nat



```
data Nat = Z | Suc Nat
```

# An Old Friend

## Nat

```haskell
data Nat = Z | Suc Nat
```

## Nat Functor

```haskell
data NatF a = Z | Suc a
```

# An Old Friend

Nat

```
data Nat = Z | Suc Nat
```

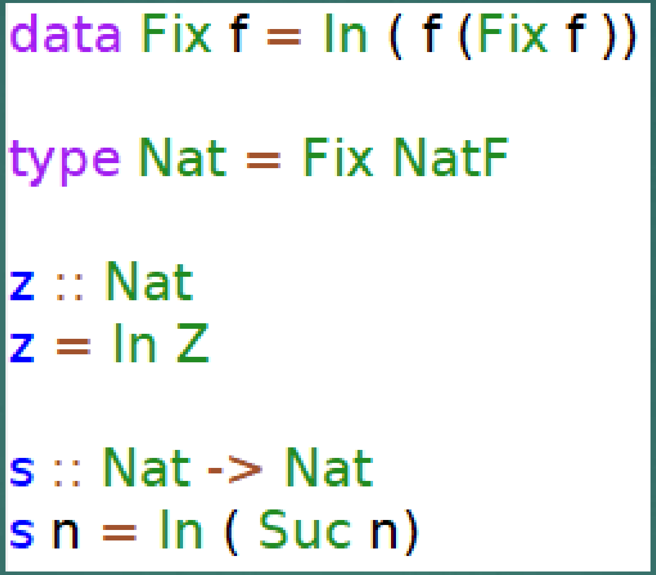
```
(Suc (Suc (Suc Z))) : Nat
```

Nat Functor

```
data NatF a = Z | Suc a
```

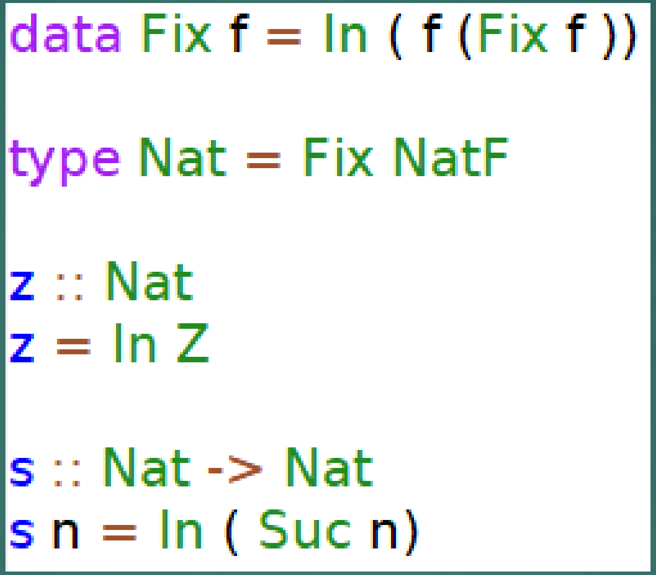
```
(Suc (Suc (Suc Z))) : NatF (NatF (NatF (NatF a)))
```

# Least Fixed Point of a Functor

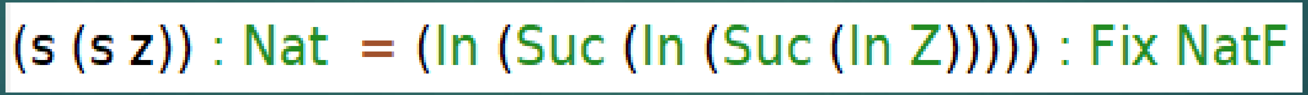


```
data Fix f = In ( f (Fix f ))

type Nat = Fix NatF

z :: Nat
z = In Z

s :: Nat -> Nat
s n = In ( Suc n)
```

# Least Fixed Point of a Functor



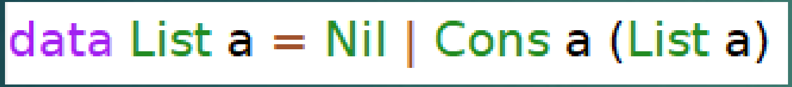
```haskell
data Fix f = In ( f (Fix f ))

type Nat = Fix NatF

z :: Nat
z = In Z

s :: Nat -> Nat
s n = In ( Suc n)
```



```
(s (s z)) : Nat  = (In (Suc (In (Suc (In Z))))) : Fix NatF
```

## Another Old Friend

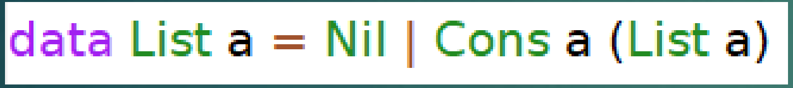
List



```
data List a = Nil | Cons a (List a)
```

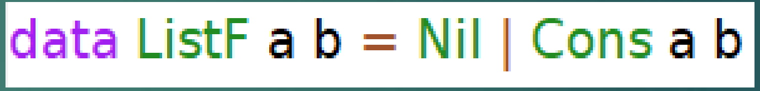
# Another Old Friend

List



```
data List a = Nil | Cons a (List a)
```

ListF



```
data ListF a b = Nil | Cons a b
```

## Another Old Friend

List

```
data List a = Nil | Cons a (List a)
```

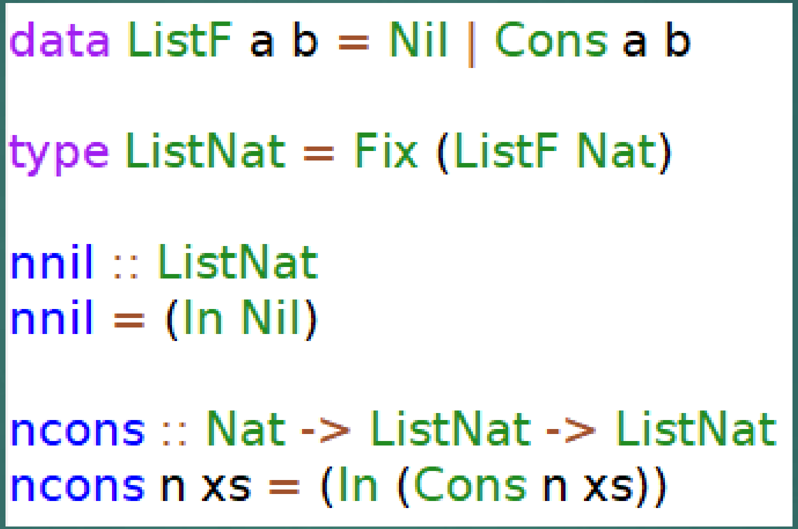
```
Cons (s z) (Cons z Nil) : List Nat
```

ListF

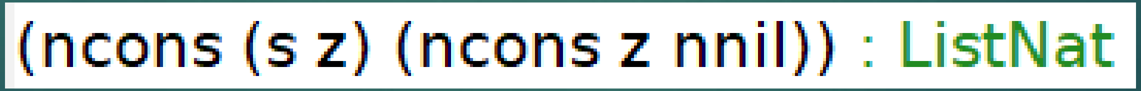
```
data ListF a b = Nil | Cons a b
```

```
(Cons (s z) (Cons z Nil)) :: ListF Nat (ListF Nat (ListF a b))
```

## List as Least Fixed Point of a Functor

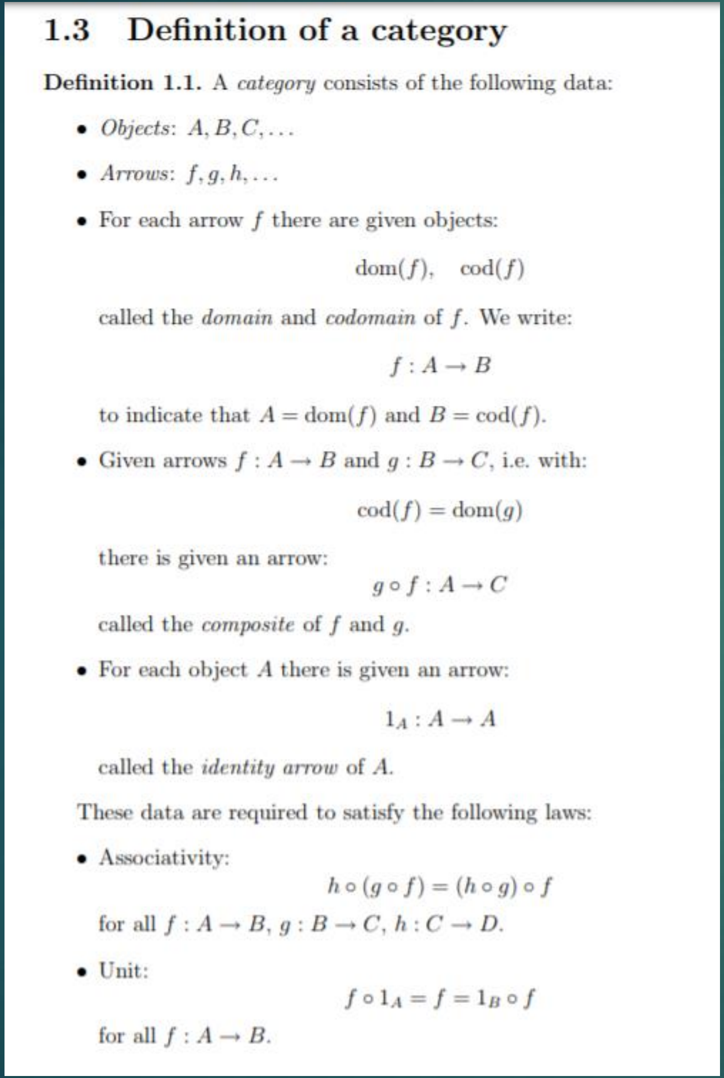


```haskell
data ListF a b = Nil | Cons a b

type ListNat = Fix (ListF Nat)

nnil :: ListNat
nnil = (In Nil)

ncons :: Nat -> ListNat -> ListNat
ncons n xs = (In (Cons n xs))
```



```
(ncons (s z) (ncons z nnil)) : ListNat
```

# Definitions



### 1.3 Definition of a category

**Definition 1.1.** A *category* consists of the following data:

- *Objects*: $A, B, C, \ldots$
- *Arrows*: $f, g, h, \ldots$
- For each arrow $f$ there are given objects:

  $$\mathrm{dom}(f), \quad \mathrm{cod}(f)$$

  called the *domain* and *codomain* of $f$. We write:

  $$f : A \to B$$

  to indicate that $A = \mathrm{dom}(f)$ and $B = \mathrm{cod}(f)$.
- Given arrows $f : A \to B$ and $g : B \to C$, i.e. with:

  $$\mathrm{cod}(f) = \mathrm{dom}(g)$$

  there is given an arrow:

  $$g \circ f : A \to C$$

  called the *composite* of $f$ and $g$.
- For each object $A$ there is given an arrow:

  $$1_A : A \to A$$

  called the *identity arrow* of $A$.

These data are required to satisfy the following laws:

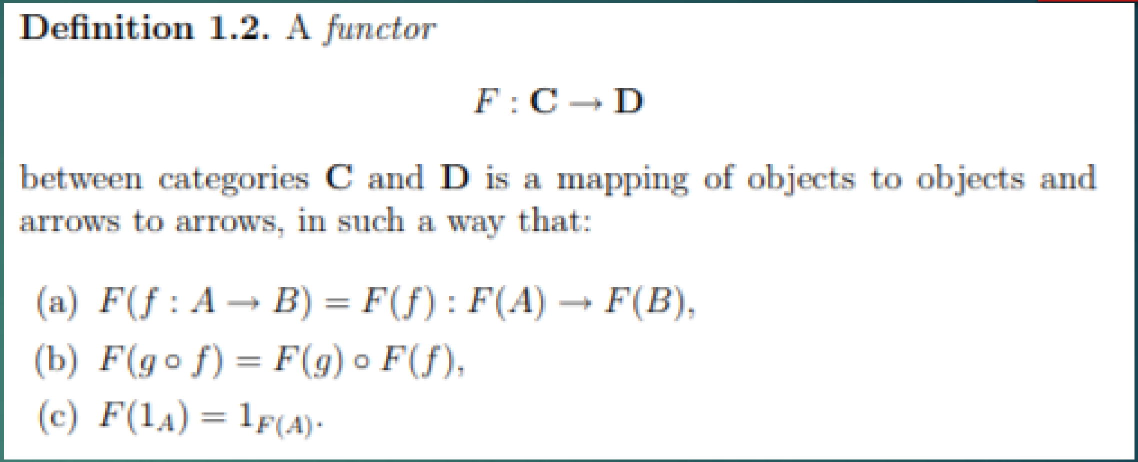
- Associativity:

  $$h \circ (g \circ f) = (h \circ g) \circ f$$

  for all $f : A \to B$, $g : B \to C$, $h : C \to D$.
- Unit:

  $$f \circ 1_A = f = 1_B \circ f$$

  for all $f : A \to B$.



**Definition 1.2.** A *functor*

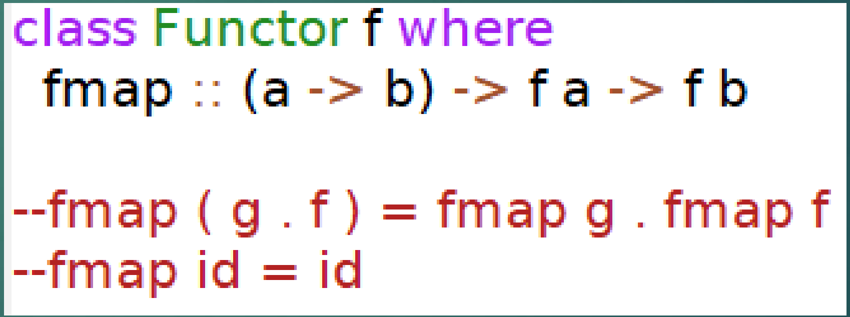
$$F : \mathbf{C} \to \mathbf{D}$$

between categories $\mathbf{C}$ and $\mathbf{D}$ is a mapping of objects to objects and arrows to arrows, in such a way that:

(a) $F(f : A \to B) = F(f) : F(A) \to F(B)$,

(b) $F(g \circ f) = F(g) \circ F(f)$,

(c) $F(1_A) = 1_{F(A)}$.



```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

--fmap ( g . f ) = fmap g . fmap f
--fmap id = id
```

## Represent Datatypes as Functors?

- Components
  - Product - x
  - Coproduct - +
  - Unit - 1
  - Constant – A

$$\begin{tikzcd}
 & Y \arrow[dl, "f_1"'] \arrow[d, dashed, "f"] \arrow[dr, "f_2"] & \\
X_1 & X_1 \times X_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & X_2
\end{tikzcd}$$

$$\begin{tikzcd}
 & Y & \\
X_1 \arrow[ur, "f_1"] \arrow[r, "i_1"'] & X_1 \coprod X_2 \arrow[u, dashed, "f"] & X_2 \arrow[ul, "f_2"'] \arrow[l, "i_2"]
\end{tikzcd}$$

# Represent Datatypes as Functors?

- Components
  - Product - x
  - Coproduct - +
  - Unit - 1
  - Constant – A
- Nat

  $$F(X) = 1 + X$$
- List

  $$F(X) = 1 + (A \times X)$$

$$\begin{tikzcd}
 & Y \arrow[dl, "f_1"'] \arrow[d, dashed, "f"] \arrow[dr, "f_2"] & \\
X_1 & X_1 \times X_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & X_2
\end{tikzcd}$$

$$\begin{tikzcd}
 & Y & \\
X_1 \arrow[ur, "f_1"] \arrow[r, "i_1"'] & X_1 \coprod X_2 \arrow[u, dashed, "f"] & X_2 \arrow[ul, "f_2"'] \arrow[l, "i_2"]
\end{tikzcd}$$

# Friends

- Nat

$$F(X) = 1 + X$$

```
data NatF a = Z | Suc a
```

- List

$$F(X) = 1 + (A \times X)$$

```
data ListF a b = Nil | Cons a b
```

- ?

$$F(X) = 1 + (X \times A \times X)$$

## Friends

- Nat

$$F(X) = 1 + X$$

```
data NatF a = Z | Suc a
```

- List

$$F(X) = 1 + (A \times X)$$

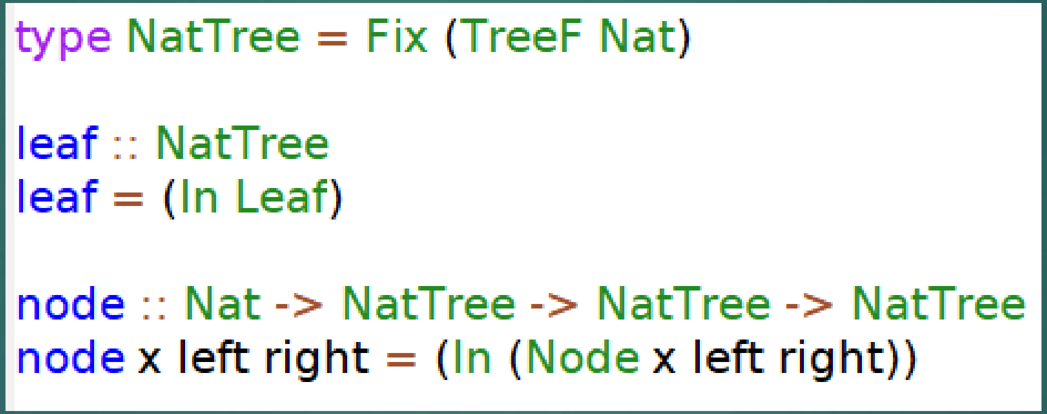
```
data ListF a b = Nil | Cons a b
```

- Binary Tree

$$F(X) = 1 + (X \times A \times X)$$

```
data TreeF a b = Leaf | Node a b b
```

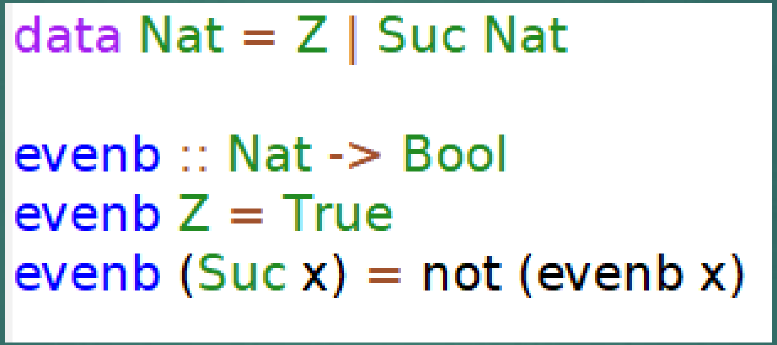
# Binary Tree as Least Fixed Point of a Functor



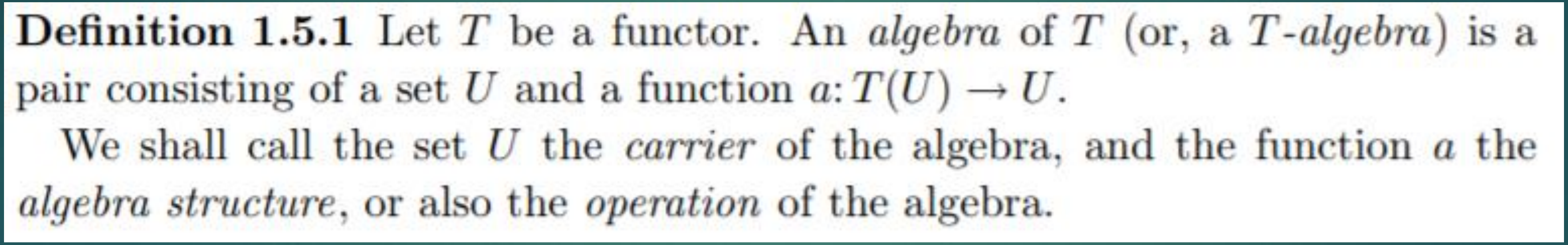
```
type NatTree = Fix (TreeF Nat)

leaf :: NatTree
leaf = (In Leaf)

node :: Nat -> NatTree -> NatTree -> NatTree
node x left right = (In (Node x left right))
```

## F-Algebras

## Functions as Algebras

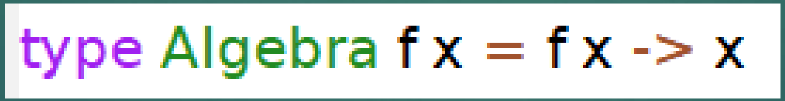

```haskell
data Nat = Z | Suc Nat

evenb :: Nat -> Bool
evenb Z = True
evenb (Suc x) = not (evenb x)
```

## F-Algebra



**Definition 1.5.1** Let $T$ be a functor. An *algebra* of $T$ (or, a $T$-*algebra*) is a pair consisting of a set $U$ and a function $a: T(U) \to U$.

We shall call the set $U$ the *carrier* of the algebra, and the function $a$ the *algebra structure*, or also the *operation* of the algebra.



```
type Algebra f x = f x -> x
```

## Functions as Algebras

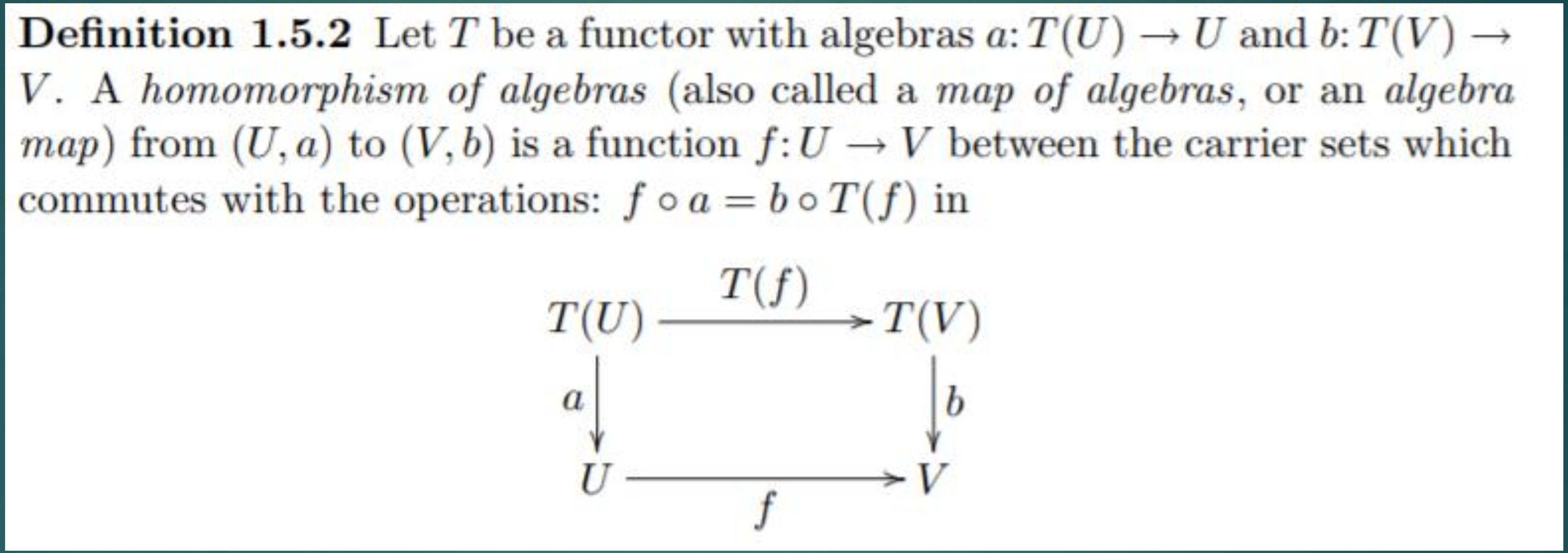
```haskell
data Nat = Z | Suc Nat

evenb :: Nat -> Bool
evenb Z = True
evenb (Suc x) = not (evenb x)
```

```haskell
type Algebra f x = f x -> x
data NatF a = Z | S a

evenbAlg :: Algebra NatF Bool
evenbAlg Z = True
evenbAlg (Suc b) = not b
```

$$\begin{tikzcd} NatF(Bool) \arrow[d, "evenbAlg"] \\ Bool \end{tikzcd}$$

```haskell
evenb = fold evenbAlg
```

# Homomorphism of Algebras

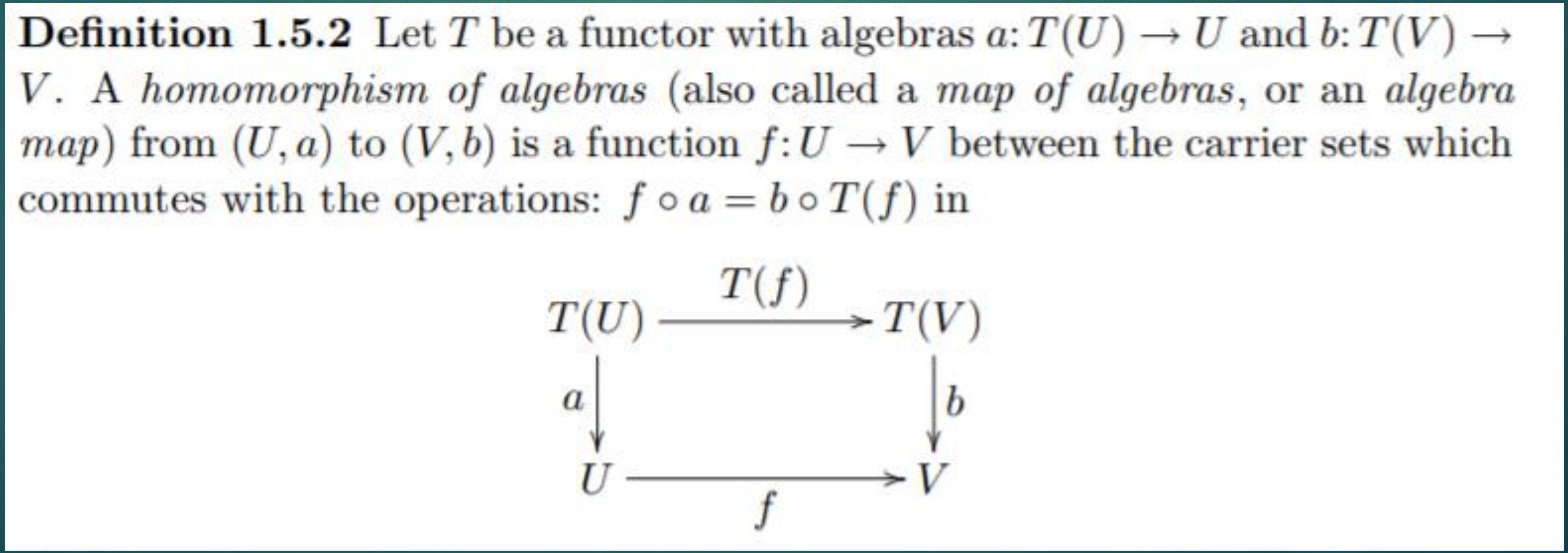


**Definition 1.5.2** Let $T$ be a functor with algebras $a\colon T(U) \to U$ and $b\colon T(V) \to V$. A *homomorphism of algebras* (also called a *map of algebras*, or an *algebra map*) from $(U, a)$ to $(V, b)$ is a function $f\colon U \to V$ between the carrier sets which commutes with the operations: $f \circ a = b \circ T(f)$ in

$$\begin{tikzcd}
T(U) \arrow[r, "T(f)"] \arrow[d, "a"'] & T(V) \arrow[d, "b"] \\
U \arrow[r, "f"'] & V
\end{tikzcd}$$

# Category of Algebras of a Functor

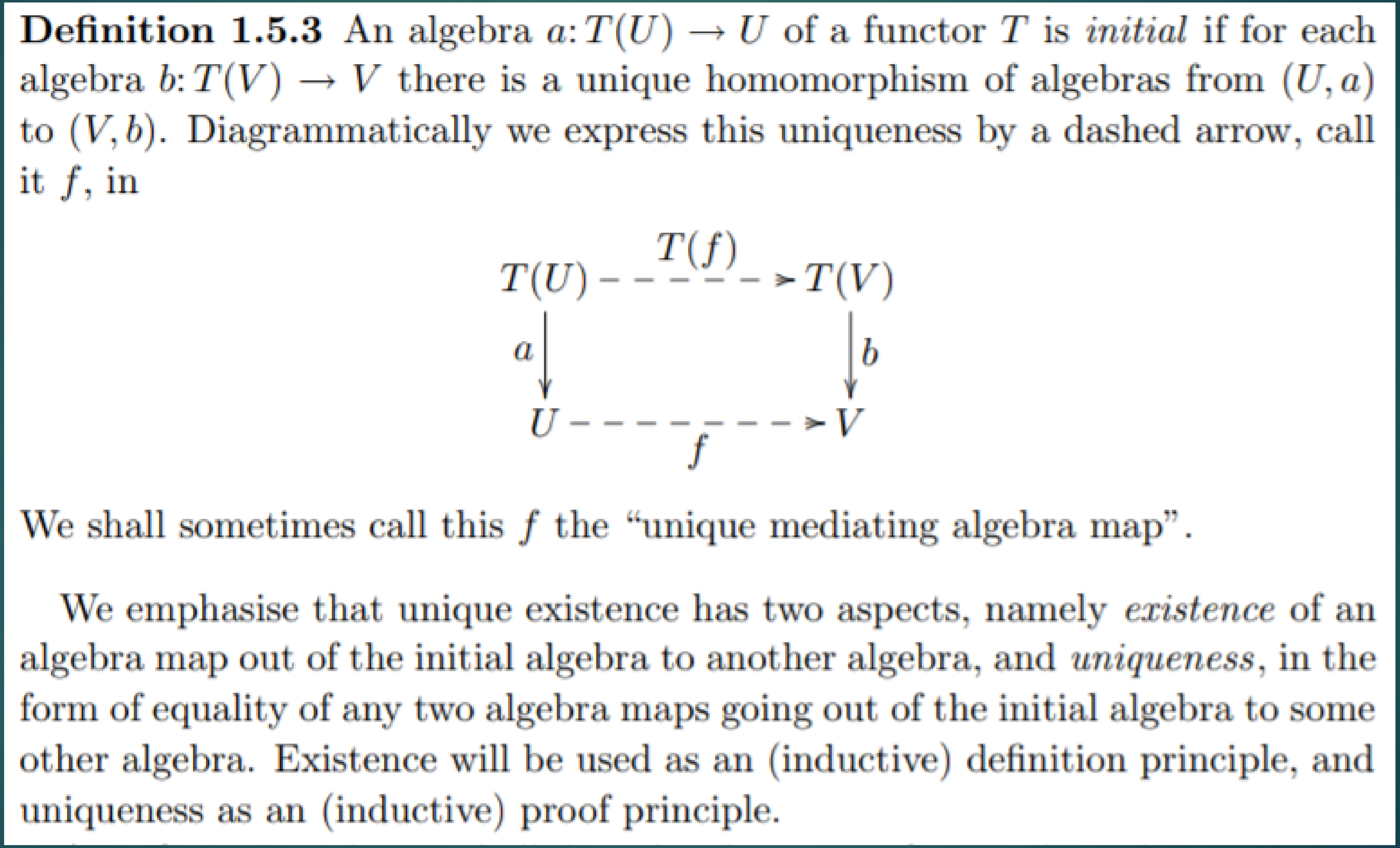
- For a given functor F,
  - Objects: F-Algebras
    - Pair ( X, a : F(X) -> X)
  - Morphisms: Algebra homomorphisms



**Definition 1.5.2** Let $T$ be a functor with algebras $a\colon T(U) \to U$ and $b\colon T(V) \to V$. A *homomorphism of algebras* (also called a *map of algebras*, or an *algebra map*) from $(U, a)$ to $(V, b)$ is a function $f\colon U \to V$ between the carrier sets which commutes with the operations: $f \circ a = b \circ T(f)$ in

$$\begin{tikzcd}
T(U) \arrow[r, "T(f)"] \arrow[d, "a"'] & T(V) \arrow[d, "b"] \\
U \arrow[r, "f"'] & V
\end{tikzcd}$$

# Initial Algebra



**Definition 1.5.3** An algebra $a\colon T(U) \to U$ of a functor $T$ is *initial* if for each algebra $b\colon T(V) \to V$ there is a unique homomorphism of algebras from $(U, a)$ to $(V, b)$. Diagrammatically we express this uniqueness by a dashed arrow, call it $f$, in

$$\begin{tikzcd}
T(U) \arrow[r, dashed, "T(f)"] \arrow[d, "a"'] & T(V) \arrow[d, "b"] \\
U \arrow[r, dashed, "f"'] & V
\end{tikzcd}$$

We shall sometimes call this $f$ the "unique mediating algebra map".

We emphasise that unique existence has two aspects, namely *existence* of an algebra map out of the initial algebra to another algebra, and *uniqueness*, in the form of equality of any two algebra maps going out of the initial algebra to some other algebra. Existence will be used as an (inductive) definition principle, and uniqueness as an (inductive) proof principle.

## Initial Algebra of the NatF Functor

$$\begin{tikzcd} NatF(Fix(NatF)) \arrow[d, "In"'] \\ Fix(NatF) \end{tikzcd}$$

```haskell
data Fix f = In (f (Fix f))
```

# Initial Algebra of the NatF Functor

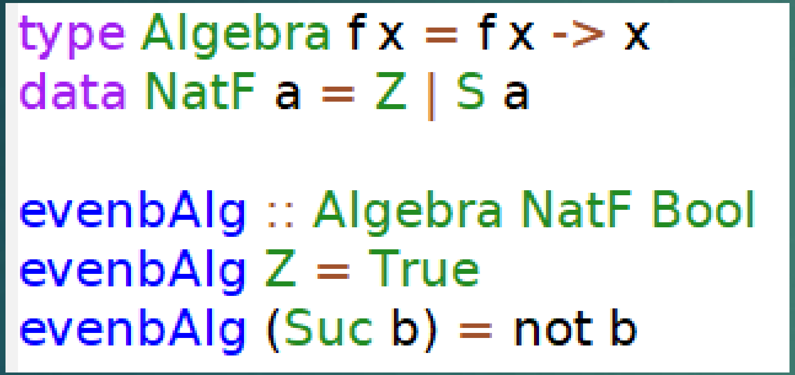
$$\begin{tikzcd}
NatF(Fix(NatF)) \arrow[d, "In"'] \\
Fix(NatF)
\end{tikzcd}$$

```haskell
data Fix f = In (f (Fix f))
```

$$\begin{tikzcd}
NatF(Fix(NatF)) \\
Fix(NatF) \arrow[u, "Out"]
\end{tikzcd}$$

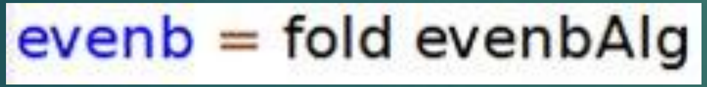
```haskell
out :: Fix f -> f (Fix f)
out (In x) = x
```

# Back to the problem...



```
type Algebra f x = f x -> x
data NatF a = Z | S a

evenbAlg :: Algebra NatF Bool
evenbAlg Z = True
evenbAlg (Suc b) = not b
```

$$\begin{tikzcd}
NatF(Fix(NatF)) \arrow[r, "fmap(g)"] \arrow[d, "In"'] & NatF(Bool) \arrow[d, "evenbAlg"] \\
Fix(NatF) \arrow[r, "g"] & Bool
\end{tikzcd}$$



```
evenb = fold evenbAlg
```

## Back to the problem...

```haskell
type Algebra f x = f x -> x
data NatF a = Z | S a

evenbAlg :: Algebra NatF Bool
evenbAlg Z = True
evenbAlg (Suc b) = not b
```

$$\begin{tikzcd}
NatF(Fix(NatF)) \arrow[r, "fmap(g)"] & NatF(Bool) \arrow[d, "evenbAlg"'] \\
Fix(NatF) \arrow[u, "Out"] \arrow[r, "g"] & Bool
\end{tikzcd}$$

```haskell
evenb = fold evenbAlg
```

$$g = evenbAlg \circ (fmap\ g) \circ Out$$

## Back to the problem...

```
type Algebra f x = f x -> x
data NatF a = Z | S a

evenbAlg :: Algebra NatF Bool
evenbAlg Z = True
evenbAlg (Suc b) = not b
```

$$\begin{tikzcd}
NatF(Fix(NatF)) \arrow[r, "fmap(g)"] & NatF(Bool) \arrow[d, "evenbAlg"] \\
Fix(NatF) \arrow[u, "Out"] \arrow[r, "g"] & Bool
\end{tikzcd}$$

```
evenb = fold evenbAlg
```

$$g = evenbAlg \circ (fmap\ g) \circ Out$$

```
fold ::Functor f =>  Algebra f x -> (Fix f) -> x
fold alg = alg . (fmap (fold alg)) . out
```

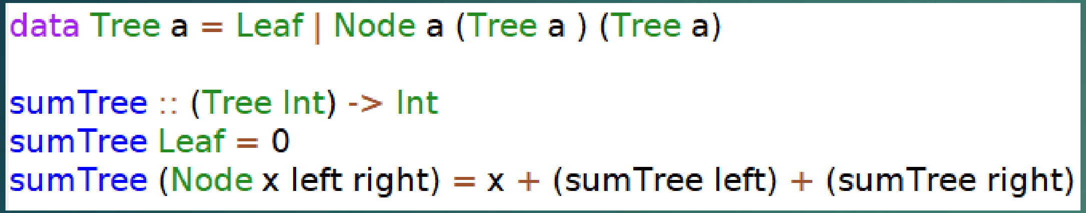
# Functions as Algebras

```haskell
data Nat = Z | Suc Nat

evenb :: Nat -> Bool
evenb Z = True
evenb (Suc x) = not (evenb x)
```

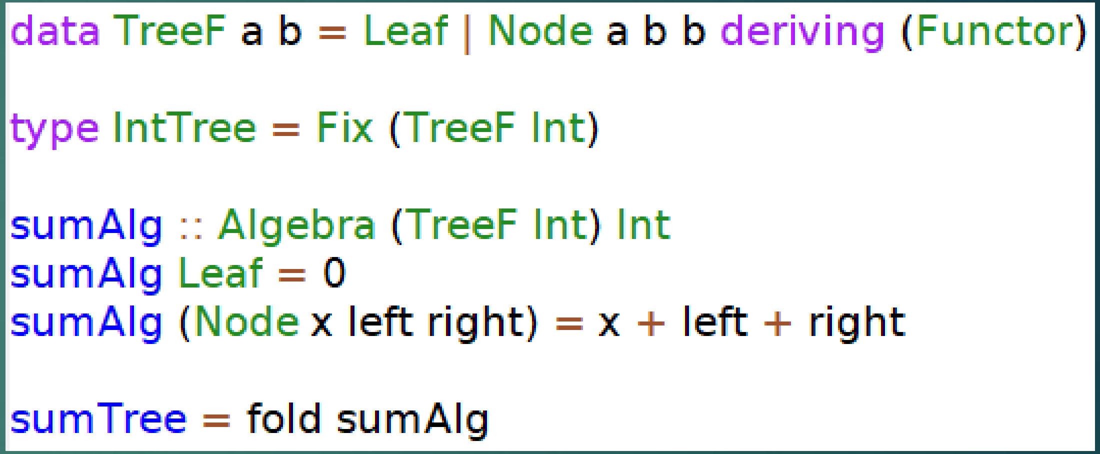
```haskell
type Algebra f x = f x -> x

evenbAlg :: Algebra NatF Bool
evenbAlg Z = True
evenbAlg (Suc b) = not b
```

```haskell
fold ::Functor f =>  Algebra f x -> (Fix f) -> x
fold alg = alg . (fmap (fold alg)) . out

evenb = fold evenbAlg
```

## Another Example



```haskell
data Tree a = Leaf | Node a (Tree a ) (Tree a)

sumTree :: (Tree Int) -> Int
sumTree Leaf = 0
sumTree (Node x left right) = x + (sumTree left) + (sumTree right)
```



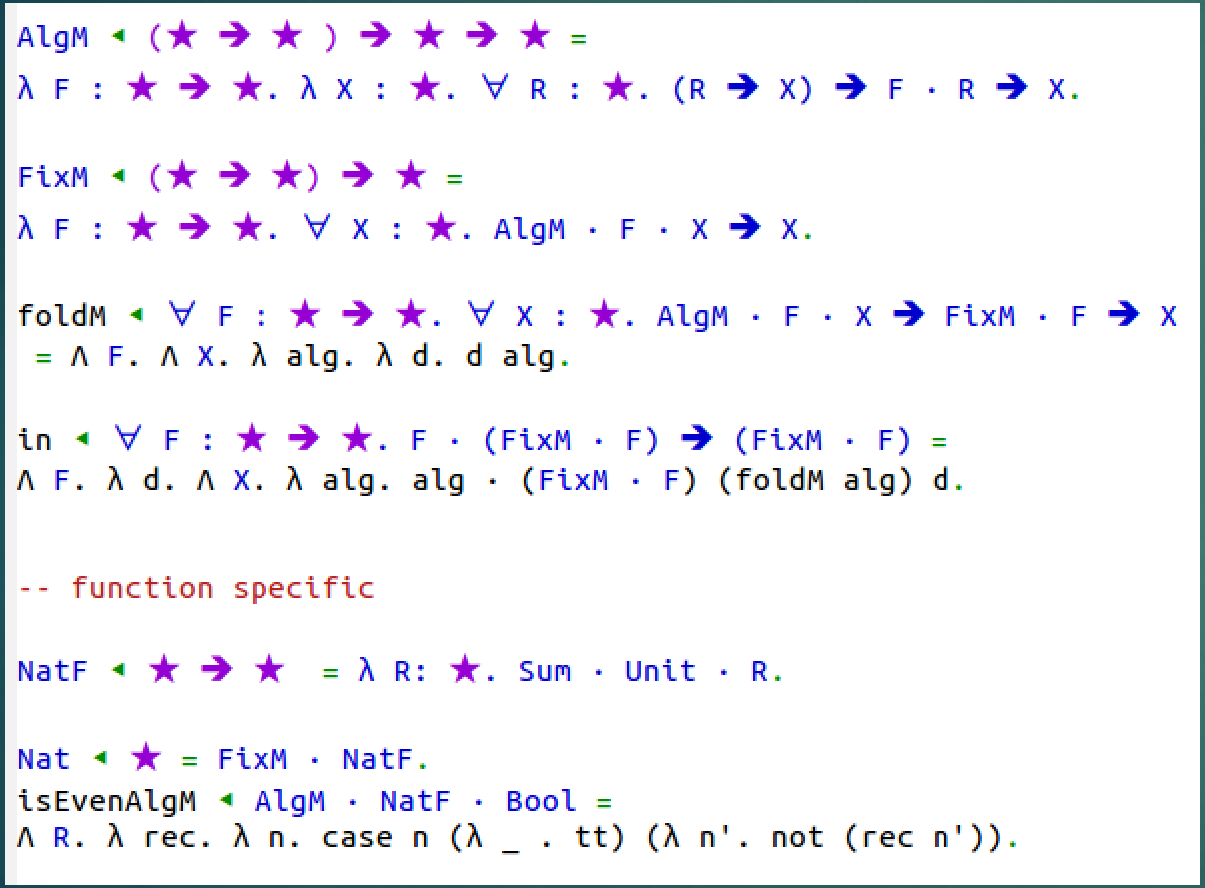
```haskell
data TreeF a b = Leaf | Node a b b deriving (Functor)

type IntTree = Fix (TreeF Int)

sumAlg :: Algebra (TreeF Int) Int
sumAlg Leaf = 0
sumAlg (Node x left right) = x + left + right

sumTree = fold sumAlg
```

## Mendler Algebras in Cedille

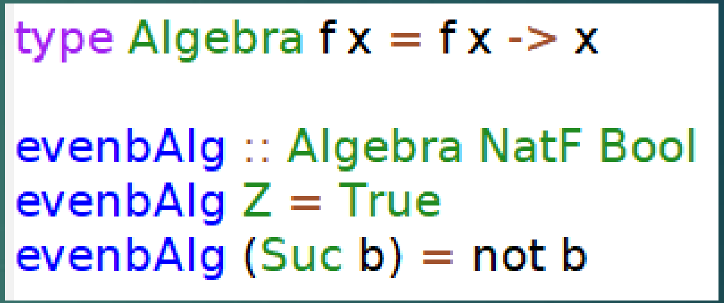
## What is Cedille?

- Minimal core type theory with derivable induction principles
  - All data are lambda encoded
  - Possible to generically derive induction for Mendler-style lambda-encodings
- Pure type theory extending the extrinsic Calculus of Constructions
  - Terms are just pure untyped lambda terms
  - Adds:
    - Implicit Product Type
    - Dependent Intersection Type
    - Heterogeneous Equality Type

# Mender-Algebra, NatF, and Evenb

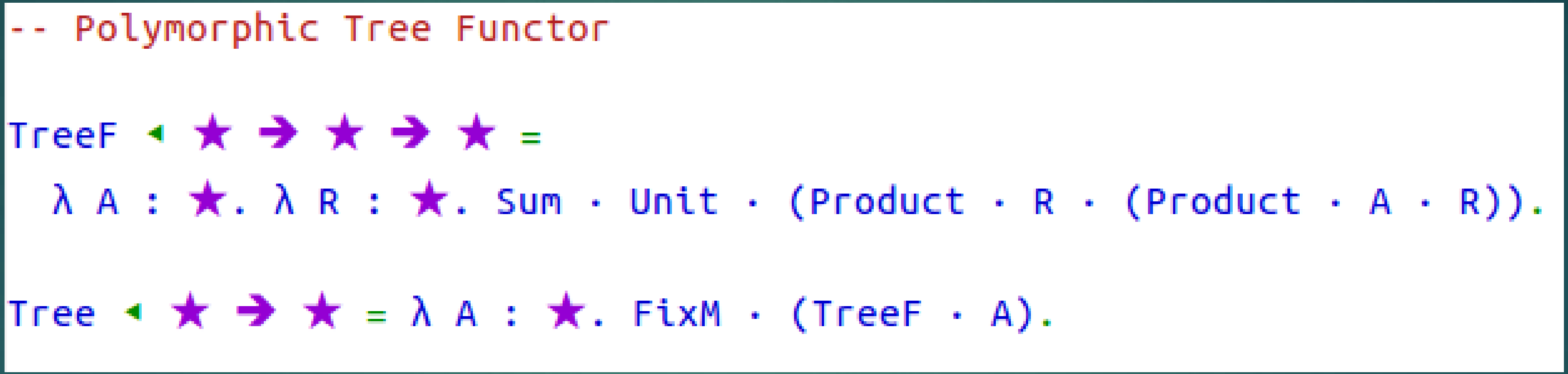


```
AlgM ◂ (★ ➔ ★ ) ➔ ★ ➔ ★ =
λ F : ★ ➔ ★. λ X : ★. ∀ R : ★. (R ➔ X) ➔ F · R ➔ X.

FixM ◂ (★ ➔ ★) ➔ ★ =
λ F : ★ ➔ ★. ∀ X : ★. AlgM · F · X ➔ X.

foldM ◂ ∀ F : ★ ➔ ★. ∀ X : ★. AlgM · F · X ➔ FixM · F ➔ X
 = Λ F. Λ X. λ alg. λ d. d alg.

in ◂ ∀ F : ★ ➔ ★. F · (FixM · F) ➔ (FixM · F) =
Λ F. λ d. Λ X. λ alg. alg · (FixM · F) (foldM alg) d.


-- function specific

NatF ◂ ★ ➔ ★  = λ R: ★. Sum · Unit · R.

Nat ◂ ★ = FixM · NatF.
isEvenAlgM ◂ AlgM · NatF · Bool =
Λ R. λ rec. λ n. case n (λ _ . tt) (λ n'. not (rec n')).
```

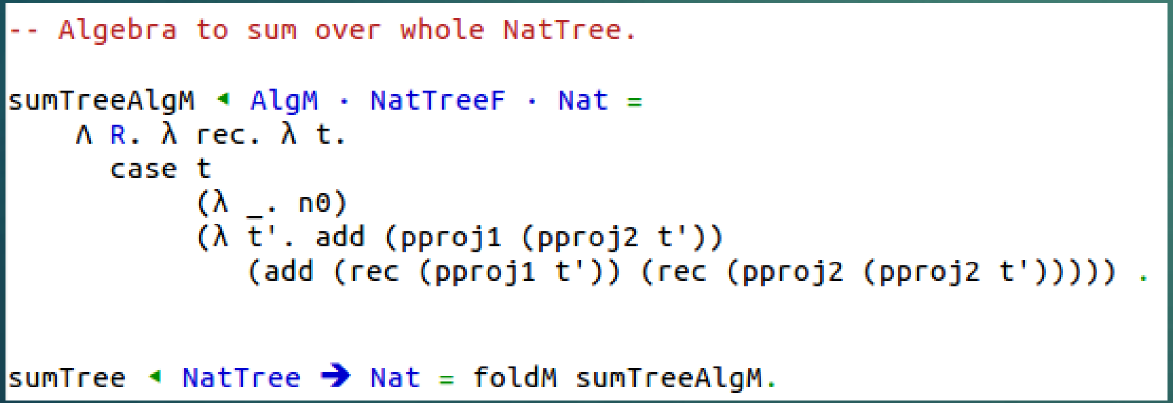


```
type Algebra f x = f x -> x

evenbAlg :: Algebra NatF Bool
evenbAlg Z = True
evenbAlg (Suc b) = not b
```

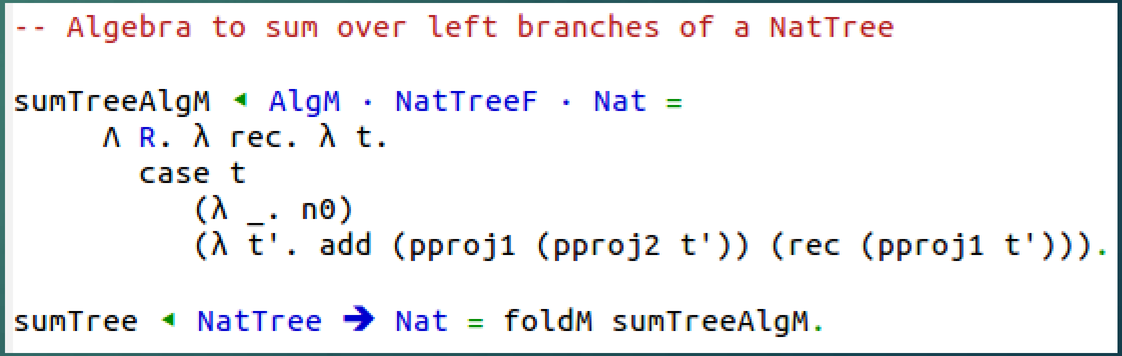
# Explicit Recursive Calls using Mendler-Algebra



```
-- Polymorphic Tree Functor

TreeF ◂ ★ ➔ ★ ➔ ★ =
  λ A : ★. λ R : ★. Sum · Unit · (Product · R · (Product · A · R)).

Tree ◂ ★ ➔ ★ = λ A : ★. FixM · (TreeF · A).
```



```
-- Algebra to sum over whole NatTree.

sumTreeAlgM ◂ AlgM · NatTreeF · Nat =
    Λ R. λ rec. λ t.
      case t
           (λ _. n0)
           (λ t'. add (pproj1 (pproj2 t'))
              (add (rec (pproj1 t')) (rec (pproj2 (pproj2 t'))))) .


sumTree ◂ NatTree ➔ Nat = foldM sumTreeAlgM.
```



```
-- Algebra to sum over left branches of a NatTree

sumTreeAlgM ◂ AlgM · NatTreeF · Nat =
     Λ R. λ rec. λ t.
       case t
          (λ _. n0)
          (λ t'. add (pproj1 (pproj2 t')) (rec (pproj1 t'))).

sumTree ◂ NatTree ➔ Nat = foldM sumTreeAlgM.
```

## Wrapping up

- F-Algebras
  - Inductive types as a least fixed point of a functor
  - Recursive functions as a fold of an algebra
- Mendler-style F-Algebras
  - Explicit recursive calls
- Cedille

# References

- https://homepages.cwi.nl/~janr/papers/files-of-papers/2011_Jacobs_Rutten_new.pdf
- http://www.cs.utexas.edu/~wcook/Drafts/2012/MTC.pdf
- http://www.cs.ru.nl/~W.Swierstra/Publications/DataTypesALaCarte.pdf
- http://homepage.cs.uiowa.edu/~astump/papers/from-realizability-to-induction-aaron-stump.pdf
- http://homepage.cs.uiowa.edu/~astump/papers/cpp-2018.pdf
- https://kodu.ut.ee/~varmo/papers/thesis.pdf
- https://www.schoolofhaskell.com/user/bartosz/understanding-algebras