Meta Theory a la Agda

Dec 2021

DSLs in the wild

- https://typelevel.org/cats/datatypes/freemonad.html
- https://www.slideshare.net/hermannhueck/composing-an-app-with-free-monads-using-cats
- https://www.baeldung.com/scala/tagless-final-pattern

```scala
trait ShoppingCarts[F[_]] {
  def create(id: String): F[Unit]
  def find(id: String): F[Option[ShoppingCart]]
  def add(sc: ShoppingCart, product: Product): F[ShoppingCart]
}
```

Example Expression DSL

```
data Exp : Set where
    Val : ℕ → Exp
    Add : Exp → Exp → Exp
    Mult : Exp → Exp → Exp

eval : Exp → ℕ
eval (Val x) = x
eval (Add x y) = eval x + eval y
eval (Mult x y) = eval x * eval y
```

```
exp : Exp
exp = (Mult (Add (Val 3) (Val 2)) (Val 3))

example : eval exp ≡ 15
example = refl
```

Datatypes a la Carte

- Not modular or extensible
- What if we wanted just Vals and Add?
- What if we wanted just Vals and Mult?
- What if we wanted to add operations?

```
data Exp : Set where
    Val : ℕ → Exp
    Add : Exp → Exp → Exp
    Mult : Exp → Exp → Exp

eval : Exp → ℕ
eval (Val x) = x
eval (Add x y) = eval x + eval y
eval (Mult x y) = eval x * eval y
```

Datatypes a la Carte

- Goal

```
ex₁ : Fix (ValF ⊹ AddF)
ex₁ = add (add (val 3) (val 5)) (val 5)
```

```
ex₂ : Fix (ValF ⊹ (AddF ⊹ MultF))
ex₂ = mult (add (val 5) (val 3)) (val 9)
```

Datatypes a la Carte

- Datatypes as a least fixpoint of a functor
- Parameterize by a type T
- Replace instances of Exp with T

```
data Exp : Set where
    Val : ℕ → Exp
    Add : Exp → Exp → Exp
    Mult : Exp → Exp → Exp

exp : Exp
exp = (Mult (Add (Val 3) (Val 2)) (Val 3))
```

```
data ExpF (T : Set) : Set where
    Val' : ℕ → ExpF T
    Add' : T → T → ExpF T
    Mult' : T → T → ExpF T

exp' : {T : Set} → ExpF (ExpF (ExpF T))
exp' = (Mult' (Add' (Val' 3) (Val' 2)) (Val' 3))
```

Datatypes a la Carte

- Datatypes as a least fixpoint of a functor

```
data Nat : Set where
    Z : Nat
    S : Nat → Nat

three : Nat
three = S (S (S Z))
```

```
data NatF (T : Set) : Set where
    Z : NatF T
    S : T → NatF T

three' : {T : Set} → NatF (NatF (NatF (NatF T)))
three' = S (S (S Z))
```

Datatypes a la Carte

```
data Fix (F : Set → Set) : Set where
    In : F (Fix F) → Fix F
```

```
data NatF (T : Set) : Set where
    Z : NatF T
    S : T → NatF T
```

```
Nat : Set
Nat = Fix NatF

z : Nat
z = In Z

s : Nat → Nat
s n = In (S n)

three : Nat
three = s (s (s z))
```

Datatypes a la Carte

- NatF is a Functor

```
record Functor(F : Set → Set): Set where
    field
        fmap : {X Y : Set} → (f : X → Y) → F X → F Y
```

```
instance
    NatF-Functor : Functor NatF
    NatF-Functor = record {
                        fmap = λ{f Z → Z
                                 ; f (S x) → S (f x) }}
```

Datatypes a la Carte

- What about functions?

```
evenb : Nat → Bool
evenb Z = true
evenb (S x) = not (evenb x)
```

F-Algebras

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 $a\colon T(U) \to U$.

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.

```
Algebra : (Set → Set) → Set → Set
Algebra F A = F A → A
```

```
evenbalg : Algebra NatF Bool
evenbalg Z = true
evenbalg (S b) = not b
```

$$\begin{array}{c} NatF(Bool) \\ \big\downarrow{\scriptstyle evenbAlg} \\ Bool \end{array}$$

F-Algebras

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

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

F-Algebras

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

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

```
cata : {F : Set → Set}{A : Set}{{_ : Functor F}}
    → Algebra F A → Fix F → A
cata alg = alg ∘ (fmap (cata alg) ∘ out)
```

```
evenbalg : Algebra NatF Bool
evenbalg Z = true
evenbalg (S b) = not b

even : Nat → Bool
even = cata evenbalg
```

Modularity Boilerplate

```
data _⊹_ (F G : Set → Set) (E : Set) : Set where
    Inl : F E → _⊹_ F G E
    Inr : G E → _⊹_ F G E
open _⊹_

instance
    _ : {F G : Set → Set}{{_ : Functor F}}{{_ : Functor G}}→ Functor (F ⊹ G)
    _ = record { fmap = λ{ f (Inl x) → Inl (fmap f x)
                         ; f (Inr x) → Inr (fmap f x) } }
```

```
record _<:_ (sub sup : Set → Set) {{_ : Functor sub}} {{_ : Functor sup}}: Set where
    field
        inj : {A : Set} → sub A → sup A
open _<:_ {{...}}

instance
    _ : {F : Set → Set}{{_ : Functor F}} → F <: F
    _ = record { inj = id }

instance
    _ : {F G : Set → Set}{{_ : Functor F}}{{_ : Functor G}} → F <: (F ⊹ G)
    _ = record { inj = Inl}

instance
    _ : {F G H : Set → Set}{{_ : Functor F}}{{_ : Functor G}}{{_ : Functor H}}{{_ : F <: G}} → F <: (H ⊹ G)
    _ = record { inj = Inr ∘ inj }


inject : {F G : Set → Set}{{_ : Functor F}}{{_ : Functor G}}{{_ : G <: F}} → G (Fix F) → Fix F
inject = In ∘ inj
```

Modular Data

```
data ValF (E : Set) : Set where
    Val' : ℕ → ValF E

instance
    _ : Functor ValF
    _ = record { fmap = λ{ f (Val' x) → Val' x } }

val : {F : Set → Set}{{_ : Functor F}}{{_ : ValF <: F}} → ℕ → Fix F
val n = inject (Val' n)
```

```
data AddF (E : Set) : Set where
    Add' : E → E → AddF E

instance
    _ : Functor AddF
    _ = record { fmap = λ {f (Add' x y) → Add' (f x) (f y)} }

add : {F : Set → Set}{{_ : Functor F}}{{_ : AddF <: F}}→ Fix F → Fix F → Fix F
add x y = inject (Add' x y)
```

```
data MultF (E : Set) : Set where
    Mult : E → E → MultF E

instance
    _ : Functor MultF
    _ = record { fmap = λ{ f (Mult x y) → Mult (f x) (f y)} }

mult : {F : Set → Set}{{_ : Functor F}}{{_ : MultF <: F}} → Fix F → Fix F → Fix F
mult x y =  inject (Mult x y)
```

```
data ExpF (T : Set) : Set where
    Val' : ℕ → ExpF T
    Add' : T → T → ExpF T
    Mult' : T → T → ExpF T
```

```
ex1 : Fix (ValF ⊹ AddF)
ex1 = add (add (val 3) (val 5)) (val 5)

ex2 : Fix (ValF ⊹ (AddF ⊹ MultF))
ex2 = mult (add (val 5) (val 3)) (val 9)
```

Modular Evaluation

```
record EvalAlg (F : Set → Set): Set where
    field
        evalAlg : Algebra F ℕ
```

```
instance
    _ : EvalAlg ValF
    _ = record { evalAlg = λ{ (Val' x) → x }}

instance
    _ : EvalAlg AddF
    _ = record { evalAlg = λ{ (Add' x y) → x + y } }

instance
    _ : EvalAlg MultF
    _ = record { evalAlg = λ{ (Mult x y) → x * y}}

eval' : {F : Set → Set}{{_ : Functor F}}{{_ : EvalAlg F}} → Fix F → ℕ
eval' = cata evalAlg
```

Different Kinds of Algebras

```
RAlgebra : (Set → Set) → Set → Set
RAlgebra F A = F (Fix F × A) → A

{-# TERMINATING #-}
para : {F : Set → Set}{A : Set}{{_ : Functor F}} → RAlgebra F A → Fix F → A
para ralg = ralg ∘ (fmap < id , para ralg > ∘ out)
```

```
MAlgebra : (Set → Set) → Set → Set
MAlgebra F A = ∀ {R : Set} → (R → A) → F R → A

FixM : (Set → Set) → Set
FixM F = ∀ {A : Set} → MAlgebra F A → A

cataM : {F : Set → Set}{A : Set}{{_ : Functor F}} → MAlgebra F A → FixM F → A
cataM malg fa = fa malg

evalM : MAlgebra ExpF ℕ
evalM ⟦_⟧ (Val x) = x
evalM ⟦_⟧ (Add x y) = ⟦ x ⟧ +  ⟦ y ⟧
```

Proof Algebras

```
ProofAlgebra : {F : Set → Set}(P : Fix F → Set) → Set
ProofAlgebra {F} P = Algebra F (Σ[ e ∈ Fix F ] P e)

Nat-ind :   (P : Nat → Set)
            (Hz : P z)
            (Hs : ∀ (n : Nat) → P n → P (s n))
            → ProofAlgebra P
Nat-ind P hz hs Z = z , hz
Nat-ind P hz hs (S x) = s (proj₁ x) , hs (proj₁ x) (proj₂ x)

WF-proof-alg : {F : Set → Set}{{_ : Functor F}}{P : Fix F → Set}
                → (alg : ProofAlgebra P)
                → Set
WF-proof-alg alg = (proj₁ ∘ alg) ≡ (In ∘ fmap proj₁)
```