Initial Algebras for the Uninitiated

Nicholas Cowle
@nickcowle

March 2019

1.4.4 Example In the category Ω-**Alg** of algebras with signature Ω, the initial object is the *initial algebra* (or *term algebra*) whose carrier consists of all finite trees where each node is labeled with an operator from Ω and where each node labeled with ω has exactly $ar(\omega)$ subtrees. (It is easy to see that this defines an Ω-algebra. The initiality of this algebra is a standard result of universal algebra [41].) The unique homomorphism from the term algebra to another Ω-algebra is a *semantic interpretation function*.

Thanks for attending my talk

Nicholas Cowle
@nickcowle
March 2019

*not a true story

**please don’t ever do this

SuperArrow.fsi

```
type SuperArrow<'a, 'b>


module SuperArrow =

    val make : string -> ('a -> 'b) -> SuperArrow<'a, 'b>

    val compose :
        SuperArrow<'a, 'b> -> SuperArrow<'b, 'c> ->
        SuperArrow<'a, 'c>
```

SuperArrow.fs

```
type SuperArrow<'a, 'b> = SA of string * ('a -> 'b)

module SuperArrow =

    let make name f = SA (name, f)

    let compose
        (SA (name1, f))
        (SA (name2, g)) =
        let name = sprintf "%s -> %s" name1 name2
        let h = f >> g
        SA (name, h)
```

SuperArrow2.fsi

...but what about?

```
module SuperArrow =

    val countParts : SuperArrow<'a, 'b> -> int
```

SuperArrow2.fs

```fsharp
type SuperArrow2<'a, 'b> =
    SA2 of string * ('a -> 'b) * int

module SuperArrow2 =

    let make name f = SA2 (name, f, 1)

    let compose
        (SA2 (name1, f, n)) (SA2 (name2, g, m)) =
        let name = sprintf "%s -> %s" name1 name2
        let h = f >> g
        let count = n + m
        SA2 (name, h, count)
```

SuperArrow2.fs

```fsharp
type SuperArrow2<'a, 'b> =
    SA2 of string * ('a -> 'b) * int

module SuperArrow2 =

    let make name f = SA2 (name, f, 1)

    let compose
        (SA2 (name1, f, n)) (SA2 (name2, g, m)) =
        let name = sprintf "%s -> %s" name1 name2
        let h = f >> g
        let count = n + m
        SA2 (name, h, count)
```

SuperArrow3.fsi

...but what about?

```
module SuperArrow =

    val print : sep:string -> SuperArrow<'a, 'b> -> string
```

```
let replaceSeparator input sep =
    Regex.Replace(input, "->", sep)
```

SuperArrow3.fs

```fsharp
type SuperArrow3<'a, 'b> =
    SA3 of (string -> string) * ('a -> 'b) * int

module SuperArrow3 =

    let make name f = SA3 ((fun _ -> name), f, 1)

    let compose
        (SA3 (name1, f, n)) (SA3 (name2, g, m)) =
        let name sep = sprintf "%s %s %s"
                            (name1 sep) sep (name2 sep)
        let h = f >> g
        let count = n + m
        SA3 (name, h, count)
```

SuperArrow3.fs

```
type SuperArrow3<'a, 'b> =
    SA3 of (string -> string) * ('a -> 'b) * int

module SuperArrow3 =

    let make name f = SA3 ((fun _ -> name), f, 1)

    let compose
        (SA3 (name1, f, n)) (SA3 (name2, g, m)) =
        let name sep = sprintf "%s %s %s"
                           (name1 sep) sep (name2 sep)
        let h = f >> g
        let count = n + m
        SA3 (name, h, count)
```

SuperArrow4.fsi

...but what about?

```
module SuperArrow =

    val getTypes : SuperArrow<'a, 'b> -> Type list
```

What’s the problem?

What’s the problem?

- New question → changes to the data structure
- We have to rewrite code that depends on its shape
- This makes us sad 🙁

SuperArrow.fsi

```
module SuperArrow =

    val make
    val compose

    val countParts
    val print
    val getTypes
```

SuperArrow.fsi

```
module SuperArrow =

    val make
    val compose

    val countParts
    val print
    val getTypes
```

Creates SuperArrows

Operates on SuperArrows

Algebras

e.g. the Integers:

$$\{\mathbb{Z}, [+, *, 0, 1]\}$$

Algebras

e.g. the Integers:

$$\{\mathbb{Z}, [+, *, 0, 1]\}$$

The ‘carrier’, i.e. datatype

The ‘signature’, i.e. public interface

Algebra Example - $\{\mathbb{Z}, [+, *, 0, 1]\}$

```
module Integer =

    val plus  : Integer -> Integer -> Integer
    val times : Integer -> Integer -> Integer
    val zero  : Integer
    val one   : Integer
```

Integer.fs

Algebra Example - $\{\mathbb{Z}, [+, *, 0, 1]\}$

```
type Integer1 =
| Plus  of Integer1 * Integer1
| Times of Integer1 * Integer1
| Zero
| One
```

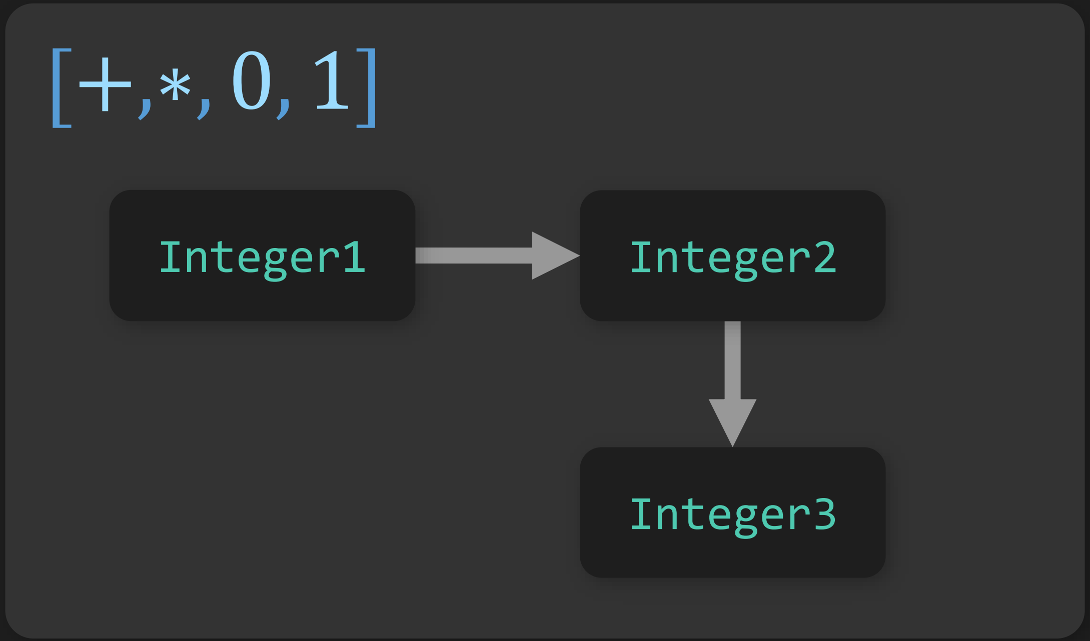
```
type Integer2 = int

type Integer3 = Unit
```

Carriers of an Algebra form a Category

```
type Integer1 =
| Plus  of Integer1 * Integer1
| Times of Integer1 * Integer1
| Zero
| One

type Integer2 = int
type Integer3 = Unit
```

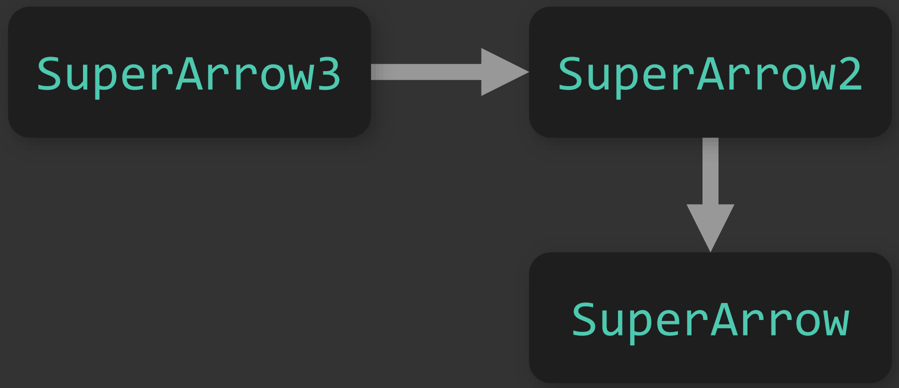


SuperArrow is an algebra

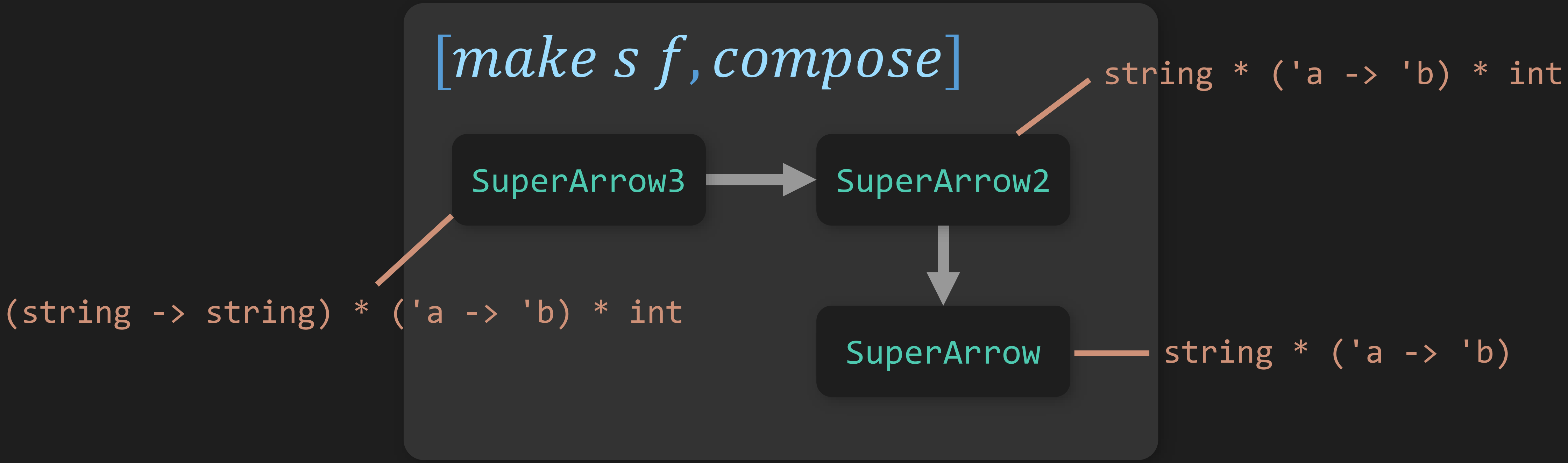
$$\{SuperArrow, [make\ s\ f, compose]\}$$

Carriers of SuperArrrow form a Category

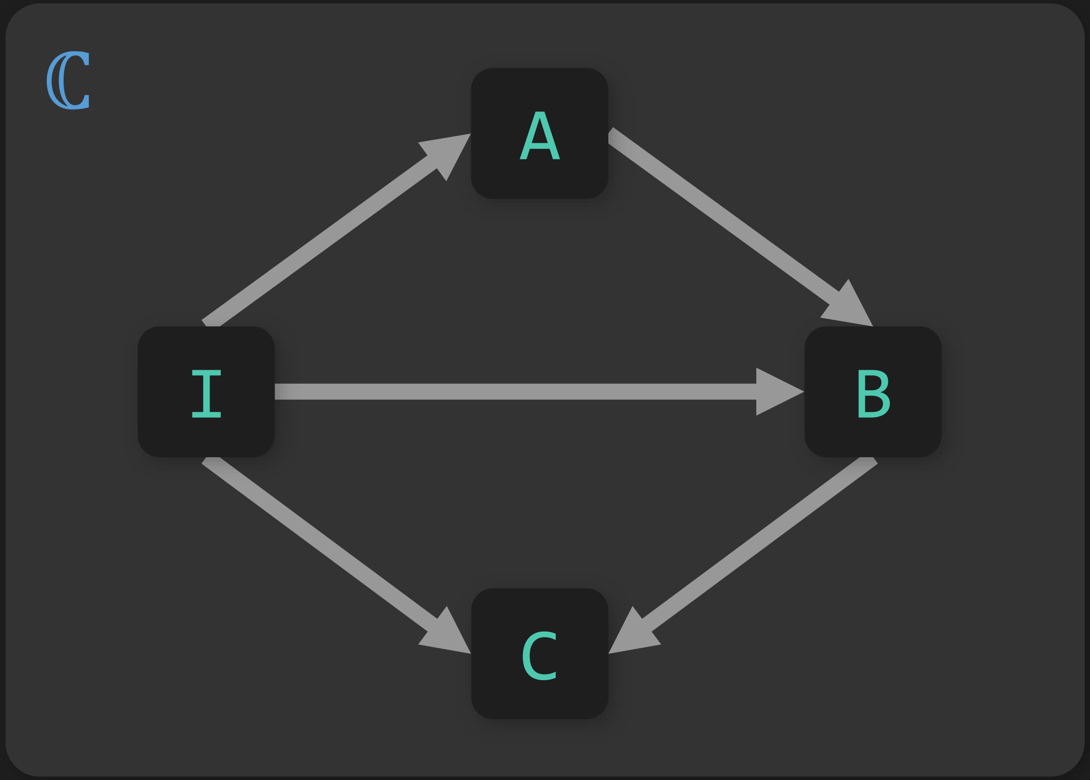
$[make\ s\ f, compose]$



Carriers of SuperArrrow form a Category



Initial Objects



Initial Objects

Given a signature of an algebra, can we find a carrier that's initial?

…Yes we can!

What do I do to make one?

DO NOTHING*

*with your inputs

Algebra Example - $\{\mathbb{Z}, [+, *, 0, 1]\}$

```
module Integer =
    val plus  : Integer -> Integer -> Integer
    val times : Integer -> Integer -> Integer
    val zero  : Integer
    val one   : Integer
```

```
type Integer =
| Plus  of Integer * Integer
| Times of Integer * Integer
| Zero
| One
```

Algebra Example - $\{\mathbb{Z}, [+, *, 0, 1]\}$

```
module Integer =
    let plus  i1 i2 = Plus  (i1, i2)
    let times i1 i2 = Times (i1, i2)
    let zero        = Zero
    let one         = One
```

```
type Integer =
| Plus  of Integer * Integer
| Times of Integer * Integer
| Zero
| One
```

Your Favourite Data Types...

- Tuples
- Lists
- Binary Trees
- (more generally) any algebraic data type

...are all initial algebras!
(wrt. their constructors)

Your Favourite Data Types...

- Tuples - $\{A * B, [(a, b)]\}$
- Lists - $\{List, [Nil, Cons\ a]\}$
- Binary Trees - $\{Tree, [Leaf\ a, Branch]\}$
- (more generally) any algebraic data type

...are all initial algebras!

(wrt. their constructors)

SuperArrow5.fs

```fsharp
type SuperArrow<'a, 'b> =
| Leaf of string * ('a -> 'b)
| Branch of SuperArrow<'a, 'c> * SuperArrow<'c, 'b>


module SuperArrow =

    let make name f = Leaf (name, f)

    let compose arrow1 arrow2 =
        Branch (arrow1, arrow2)
```

SuperArrow5.fs

```
type SuperArrow<'a, 'b> =
| Leaf of string * ('a -> 'b)
| Branch of SuperArrow<'a, 'c> * SuperArrow<'c, 'b>


module SuperArrow =

    let make name f = Leaf (name, f)

    let compose arrow1 arrow2 =
        Branch (arrow1, arrow2)
```

Existentially quantified

Type theory to the rescue...

Universals & Existentials

In .NET…

- First class universals?

Universals & Existentials

```
module List =

    val length<'a> : 'a List -> int
```

Universals & Existentials

```
module List =

    val length<'a> : 'a List -> int
```

First class universal?

Universals & Existentials

```
module List =

    val length<'a> : 'a List -> int
```

First class universal? Not really.

Universals & Existentials

```
let sumTheLengths
  (xs : int list)
  (ys : string list)
  (getLength : 'a list -> int) =

    getLength xs + getLength ys
```

Universals & Existentials

```
type ListLength =
    abstract member Invoke<'a> : 'a List -> int
```

Universals & Existentials

```
let sumTheLengths
  (xs : int list)
  (ys : string list)
  (getLength : ListLength) =

    getLength.Invoke xs + getLength.Invoke ys
```

Universals & Existentials

In .NET…

- First class universals? **Yes** (sort of)
- First class existentials?

Existential Example

```
type Listy = Listy of 'a List
```

Universals & Existentials

In .NET…

- First class universals? **Yes** (sort of)
- First class existentials? **No**

An Observation

```
T ≅ ∀U (T -> U) -> U
```

"If you give me something that operates on me,
then I can apply it for you"

For Ints

```
Int ≅ ∀U (Int -> U) -> U
```

“If you give me something that operates on me,
then I can apply it for you”

For Ints

```
Int ≅ ∀U (Int -> U) -> U

type IAmAnInt =
    member Apply<'u> : (int -> 'u) -> 'u
```

For Listy

```
type Listy = Listy of 'a List

T ≅ ∀U (T -> U) -> U

∃T List<T> ≅ ∀U ((∃T List<T>) -> U ) -> U
           ≅ ∀U ((∀T List<T>  -> U)) -> U
```

For Listy

```
type Listy = Listy of 'a List

T ≅ ∀U (T -> U) -> U

∃T List<T> ≅ ∀U ((∃T List<T>) -> U ) -> U
           ≅ ∀U ((∀T List<T>  -> U)) -> U
```

Extremely Important Trick

For Listy

```
∃T List<T> ≅ ∀U ((∀T List<T>  -> U)) -> U
```

For Listy

```
∃T List<T> ≅ ∀U ((∀T List<T>  -> U)) -> U
```

```
type ListEvaluator<'u> =
    abstract member Eval<'t> : 't list -> 'u
```

For Listy

$\exists T\ List\langle T\rangle \cong \boxed{\forall U\ ((\forall T\ List\langle T\rangle \to U)) \to U}$

```
type ListEvaluator<'u> =
    abstract member Eval<'t> : 't list -> 'u
```

```
type Listy =
    abstract member Apply<'u> : ListEvaluator<'u> -> 'u
```

SuperArrowInit.fs

```
type SuperArrow<'a, 'b> =
| Leaf of string * ('a -> 'b)
| Branch of SuperArrow<'a, 'c> * SuperArrow<'c, 'b>

Branch<'a, 'b> = ∃'c SuperArrow<'a, 'c> * SuperArrow<'c, 'b>
```

SuperArrowInit.fs

```
type SuperArrow<'a, 'b> =
| Leaf of string * ('a -> 'b)
| Branch of SuperArrow<'a, 'c> * SuperArrow<'c, 'b>

Branch<'a, 'b> = ∃'c SuperArrow<'a, 'c> * SuperArrow<'c, 'b>

Branch<'a, 'b> =
  ∀'ret
    (∀'c SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret)
      -> 'ret
```

Using Extremely
Important Trick

SuperArrowInit.fs

```
Branch<'a, 'b> =
  ∀'ret (∀'c SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret) -> 'ret

type BranchEvaluator<'a, 'b, 'ret> =
    abstract member Eval<'c> :
      SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret

type Branch<'a, 'b> =
    abstract member Apply<'ret> : BranchEvaluator<'a, 'b, 'ret> -> 'ret
```

SuperArrowInit.fs

```
Branch<'a, 'b> =
  ∀'ret (∀'c SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret) -> 'ret

type BranchEvaluator<'a, 'b, 'ret> =
    abstract member Eval<'c> :
      SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret

type Branch<'a, 'b> =
    abstract member Apply<'ret> : BranchEvaluator<'a, 'b, 'ret> -> 'ret
```

SuperArrowInit.fs

```
Branch<'a, 'b> =
  ∀'ret (∀'c SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret) -> 'ret

type BranchEvaluator<'a, 'b, 'ret> =
    abstract member Eval<'c> :
      SuperArrow<'a, 'c> -> SuperArrow<'c, 'b> -> 'ret

type Branch<'a, 'b> =
    abstract member Apply<'ret> : BranchEvaluator<'a, 'b, 'ret> -> 'ret
```

Putting It All Together...

```
type SuperArrowInit<'a, 'b> =
| Leaf of string * ('a -> 'b)
| Branch of Branch<'a, 'b>

and BranchEvaluator<'a, 'b, 'ret> =
    abstract member Eval<'c> :
      SuperArrowInit<'a, 'c> -> SuperArrowInit<'c, 'b> -> 'ret

and Branch<'a, 'b> =
    abstract member Apply<'ret> : BranchEvaluator<'a, 'b, 'ret> -> 'ret
```

In action...

```
let rec countParts<'a, 'b> (arrow : SuperArrowInit<'a, 'b>) : int =
    match arrow with
    | Leaf _ -> 1
    | Branch branch ->
       branch.Apply
           { new BranchEvaluator<'a, 'b, int> with
               member __.Eval<'c>
                   (arrow1 : SuperArrowInit<'a, 'c>)
                   (arrow2 : SuperArrowInit<'c, 'b>) =
                   countParts arrow1 + countParts arrow2
           }
```

In action...

```
let rec countParts<'a, 'b> (arrow : SuperArrowInit<'a, 'b>) : int =
    match arrow with
    | Leaf _ -> 1
    | Branch branch ->
        branch.Apply
            { new BranchEvaluator<'a, 'b, int> with
                member __.Eval<'c>
                    (arrow1 : SuperArrowInit<'a, 'c>)
                    (arrow2 : SuperArrowInit<'c, 'b>) =
                    countParts arrow1 + countParts arrow2
            }
```

In action...

```
let rec print<'a, 'b>
    (sep : string) (arrow : SuperArrowInit<'a, 'b>) : string =
    match arrow with
    | Leaf (name, _) -> name
    | Branch branch ->
        branch.Apply
            { new BranchEvaluator<'a, 'b, string> with
                member __.Eval<'c>
                    (arrow1 : SuperArrowInit<'a, 'c>)
                    (arrow2 : SuperArrowInit<'c, 'b>) =
                    print sep arrow1 + sep + print sep arrow2
            }
```

In action...

```
let rec print<'a, 'b>
    (sep : string) (arrow : SuperArrowInit<'a, 'b>) : string =
    match arrow with
    | Leaf (name, _) -> name
    | Branch branch ->
        branch.Apply
            { new BranchEvaluator<'a, 'b, string> with
                member __.Eval<'c>
                    (arrow1 : SuperArrowInit<'a, 'c>)
                    (arrow2 : SuperArrowInit<'c, 'b>) =
                    print sep arrow1 + sep + print sep arrow2
            }
```

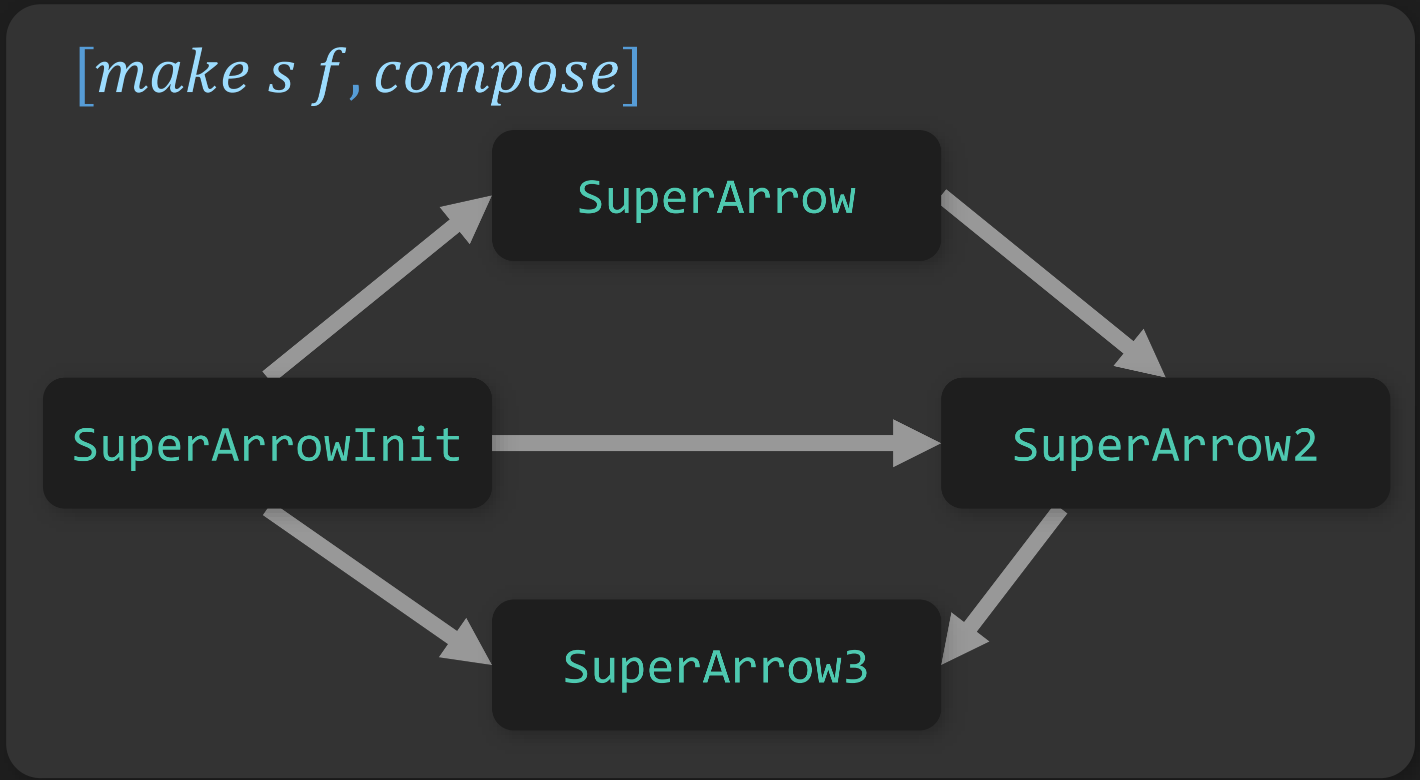
In action...

```
let rec getTypes<'a, 'b> (arrow : SuperArrowInit<'a, 'b>) : Type list =
    match arrow with
    | Leaf _ -> [ typeof<'a> ; typeof<'b> ]
    | Branch branch ->
        branch.Apply
            { new BranchEvaluator<'a, 'b, Type list> with
                member __.Eval<'c>
                    (arrow1 : SuperArrowInit<'a, 'c>)
                    (arrow2 : SuperArrowInit<'c, 'b>) =
                    getTypes arrow1 @ List.tail (getTypes arrow2)
            }
```

In action...

```
let rec getTypes<'a, 'b> (arrow : SuperArrowInit<'a, 'b>) : Type list =
    match arrow with
    | Leaf _ -> [ typeof<'a> ; typeof<'b> ]
    | Branch branch ->
        branch.Apply
            { new BranchEvaluator<'a, 'b, Type list> with
                member __.Eval<'c>
                    (arrow1 : SuperArrowInit<'a, 'c>)
                    (arrow2 : SuperArrowInit<'c, 'b>) =
                    getTypes arrow1 @ List.tail (getTypes arrow2)
            }
```

Success!


[make s f, compose]
SuperArrow
SuperArrowInit
SuperArrow2
SuperArrow3

In Haskell

```haskell
{-# LANGUAGE ExistentialQuantification #-}

data SuperArrow a b =
    Leaf String (a -> b)
  | forall c. Branch (SuperArrow a c) (SuperArrow c b)
```

In Haskell (with GADTs)

```haskell
{-# LANGUAGE GADTs #-}

data SuperArrow a b where
  Leaf :: String -> (a -> b) -> SuperArrow a b
  Branch :: SuperArrow a b -> SuperArrow b c -> SuperArrow a c
```

In Idris

```
data SuperArrow : Type -> Type -> Type where
  Leaf : String -> (a -> b) -> SuperArrow a b
  Branch : SuperArrow a b -> SuperArrow b c -> SuperArrow a c
```

Conclusion

The Good

- Offer a clean separation between description and interpretation
- You find them everywhere in functional programming
- Extremely powerful. Basically awesome.

The Bad

- Need to be careful when writing performant code

The Ugly

- Existentials are extremely verbose in F#... for now?

Thanks for attending my talk

(really this time)

Nicholas Cowle
@nickcowle
March 2019