Goals of paper

* Show that Uniplate and Compos data types are isomorphic
* Develop a third library (Multiplate) that combines the two

How do we get there

* The **Lens** structure
* The **Store** Comonad
* Two different representations of the **Lens** structure
* Core data type of Uniplate is a generalization of first representation
* Core data type of Compos is a generalization of second representation
* Proof that both data types are isomorphic
* Creation of Multiplate library

The **Lens** structure

* During programming, you often have tree-like data structures you wish to manipulate
* For example, replacing a subtree (or subexpression)
* Often used for abstract syntax trees, which are often built from mutually recursive data types

[NOTE: Know what abstract syntax tree is]

* Lens structure = functional reference or **accessor**
* Data Lens a b = Address { get :: a -> b, set :: a -> b -> a }
* Used for manipulating subexpressions:
  + get, set and modify a substructure of a given structure

Two different representations of the **Lens** structure

1. As a coalgebra of the store comonad
2. As a monoidal natural transformation

**Lens structure as a coalgebra of the store comonad**

[NOTE: Comonad info from <http://www.cas.mcmaster.ca/~carette/CAS706/F2006/presentations/comonads.pdf>]

**Comonad?**

* Monads “put things *into* a box”
* Comonads “pull things *out of* a box”
* (>>=) chains action 1 to action 2
* (extend) chains action 2 to action 1
* Monads take into account past computations
* Comonads take into account future computations

Example: List

[NOTE: list example taken from <http://blog.sigfpe.com/2007/02/comonads-and-reading-from-future.html>]

[NOTE: why w ? well it’s m upside down, so “dual”]

> class Functor w => Comonad w where

> extract :: w a -> a (Dual to return)

> duplicate :: w a -> w(w a)

> duplicate = extend id (Dual to join)

> extend :: (w a -> g) -> w a -> w g (Dual to bind)

> extend f x = fmap f (duplicate x)

> instance Comonad [] where

> extract (x:xs) = x

> extend f [] = []

> extend f (x:xs) = f (x:xs) : extend f xs

> cfix :: Comonad d => d (d a -> a) -> a

> cfix d = extract d (extend cfix d)

Comonadic fix: given a comonad, applies the comonad recursively to the comonadic fix (which yields a list of “future computations”). It then gives the output of the comonad based on those future computations.

> instance Show (x -> a)

> instance Eq (x -> a)

> instance (Num a,Eq a) => Num (x -> a) where

> fromInteger = const . fromInteger

> f + g = \x -> f x + g x

> f \* g = \x -> f x \* g x

> negate = (negate .)

> abs = (abs .)

> signum = (signum .)

> ouroboros = [2\*head.tail,1+head.tail,17]

> test = cfix ouroboros

> step1 = cfix [2\*head.tail,1+head.tail,17]

> step2 = extract [2\*head.tail,1+head.tail,17] (extend cfix [2\*head.tail,1+head.tail,17])

> step3 = (2\*head.tail) (extend cfix [2\*head.tail,1+head.tail,17])

> step4 = (2\*head.tail) (cfix [2\*head.tail,1+head.tail,17] : cfix [1+head.tail,17] : cfix [17] : cfix [])

> step5 = (2) \* (cfix [1+head.tail,17])

> step6 = (2) \* ( extract [1+head.tail,17] (extend cfix [1+head.tail,17]) )

> step7 = (2) \* ( (1+head.tail) (extend cfix [1+head.tail,17]) )

> step8 = (2) \* ( (1+head.tail) (cfix [1+head.tail,17] : cfix [17] : cfix []) )

> step9 = (2) \* ( (1+) (cfix [17]) )

> step10 = 2 \* ( 1 + (extract [17] (extend cfix [17])) )

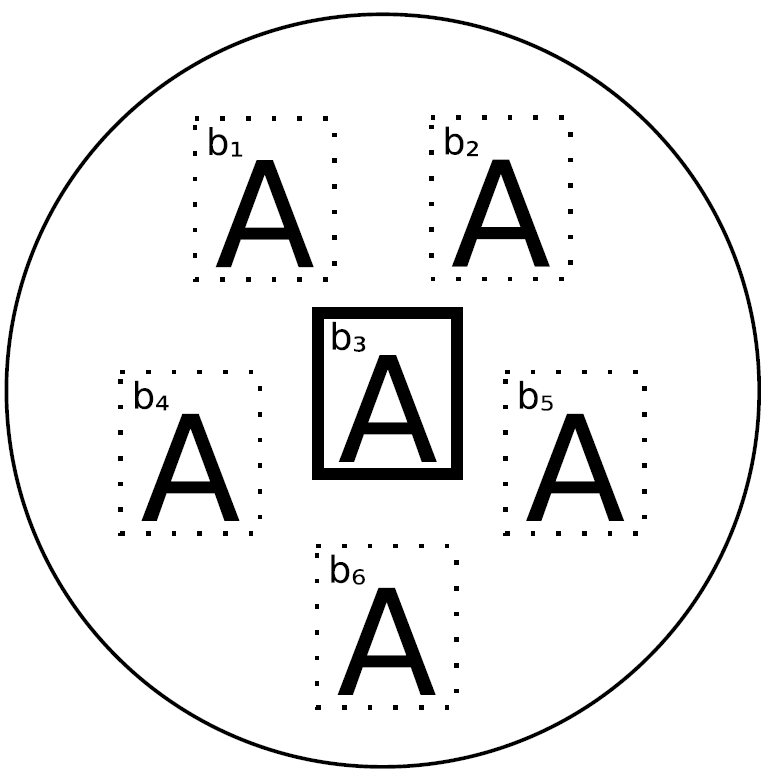
> step11 = 2 \* ( 1 + ( 17 ( cfix [17] : cfix [] ) ) )

> step12 = 2 \* ( 1 + ( 17 ) )

🡺 So we can do some cool stuff with comonads!

**The Store Comonad**

* Dual to the State monad
* Also called costate, context, state-in-context
* Data Store b a = Store { peek :: b -> a, pos :: b }
  + A value of type Store b a = collection of values of type a
  + Each element of the collection is indexed by a value of b (peek)
  + One special “selected” location (pos)



Example of Store b a

* The a values can vary, but for simplicity data type A is used everywhere
* Each A value belongs to a location labeled with bi
* b3 is currently the “selected” location: if extract is called upon this comonad, it will be the value returned.

It is a comonad for every b with the following comonadic operations:

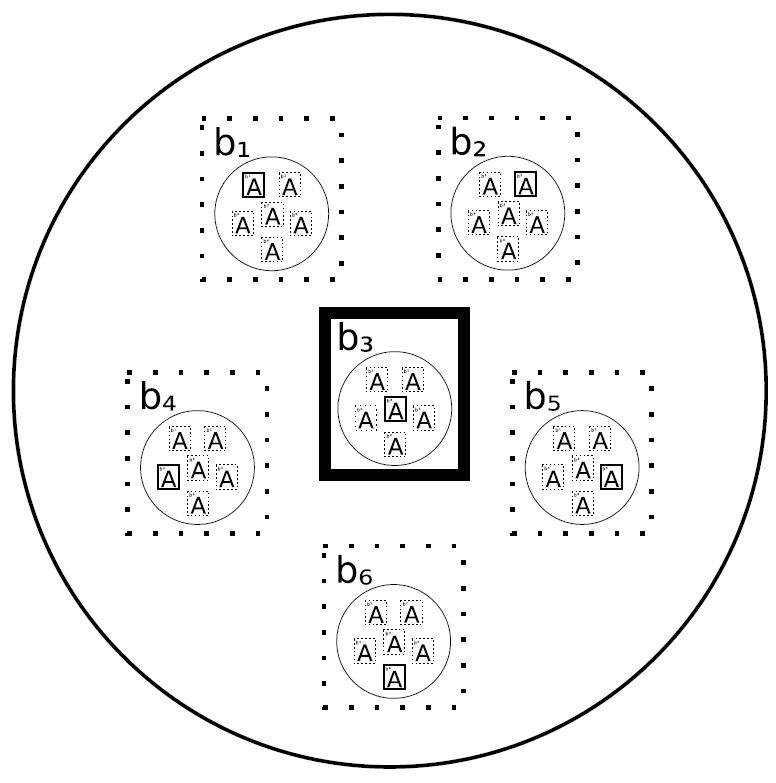
instance Functor (Store b)where

fmap f (Store v b) = Store (f ◦ v) b

instance Comonad (Store b)where

extract (Store v b) = vb

duplicate (Store v b) = Store (Store v) b



* Duplicate produces interesting results:
  + Producs a collection of all possible selections for the input
  + Each cell bi is replaced with a copy of the original collection, but with cell bi selected within that collection

[NOTE: Note that the original collection ends up in the originally selected cell.]

**Lenses represented with the Store Comonad**

* Type Lens a b = a 🡪 Store b a
* Let’s take an example.

data Address = Address { phone\_ :: PhoneNumber, website\_ :: URI }

patrick = Address { phone\_ = 333-4444, website\_ = http://www.patrick.com }

phoneLens :: Lens Address PhoneNumber

phoneLens = \addr 🡪 Store peek = (\newnumber 🡪 addr { phone\_ = newnumber } ), pos = (phone\_ addr)

(\newnumber 🡪 addr { phone\_ = newnumber } = (PhoneNumber 🡪 Address )

This is an update function that, given a new phone number, returns a new address with the updated phone number.

(phone\_ addr) = PhoneNumber

This is the pos component of the Store. It simply returns the phone number inside the given address.

* Given an Address,
* Produce a Store PhoneNumber Address,
  + (PhoneNumber 🡪 Address )
  + PhoneNumber

**Lens Isomorphism**

* A lens is isomorphic to a pair of getter and setter functions:
* Lens a b ≈ (a -> b) x (a -> b -> a)

(let’s make this a column)

From lens:

get :: Lens a b -> a b

get lens a = pos (lens a)

set :: Lens a b -> a -> b -> a

set lens a = peek (lens a)

To lens:

lens :: (a -> b) -> (a -> b -> a) -> Lens a b

lens getf setf =\ a -> Store (setf a) (getf a)

* With these getters and setters, we can manipulate the fields of our record:
* get phoneLens patrick = 333-4444
* set phoneLens Patrick 555-666 = Address { phone\_ = 555-666, website\_ = http://www.patrick.com }

**Lenses are the coalgebras for the store comonad**

1. The get and set functions should satisfy certain laws:
   * get *l* (set *l s b*) = *b*
   * set *l s* (get *l s*) = *s*
   * set *l* (set *l s b1*) *b2* = set *l s b2*
2. We can express these laws using the comonadic operations for the store comonad:
   * extract ◦ *l* = id
   * fmap *l* ◦ *l* = dupliate ◦ *l*
3. For all *l* of type Lens a b, (1) holds iff (2) holds.
4. A coalgebra for a functor F is a function f :: A -> F A for some type A.
5. If W is a comonad, f :: A -> W A is a coalgebra for the comonad W if the above laws are satisfied.
6. Hence, lenses are exactly the coalgebras for the Store comonad.

This will come in handy later on.

**The Cartesian Store Comonad and Biplates**

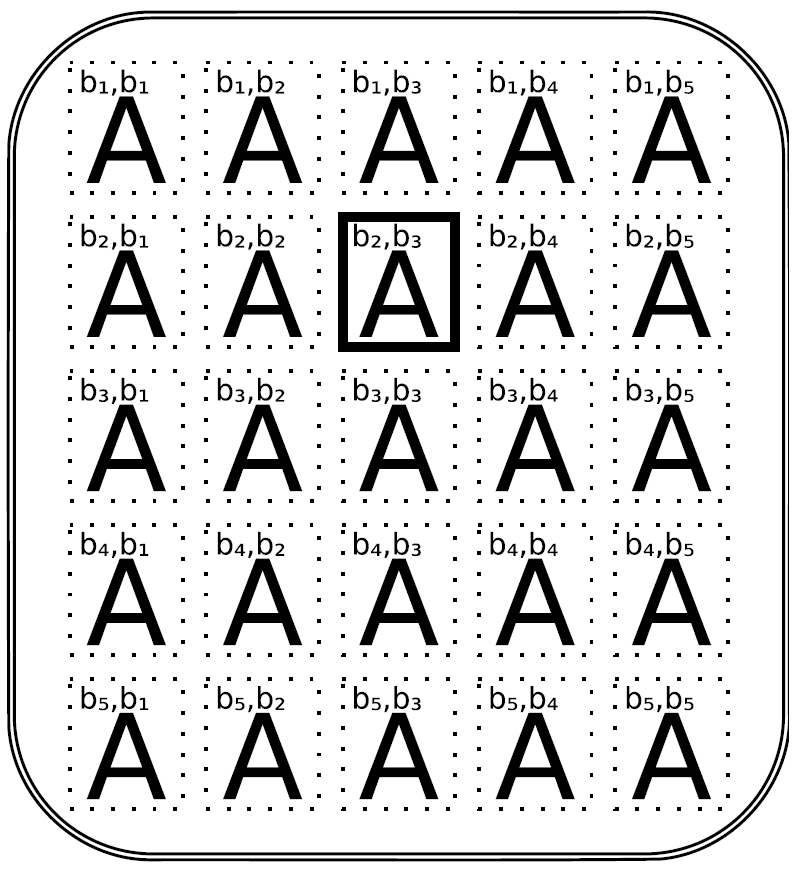
Now we know that lenses are the coalgebras for the store comonad, let’s see if we can use this information to get an isomorphism for Uniplate’s Biplate type.

They defined their BiPlate type as

* **type** Biplate a b = a -> ([b] x ([b] -> a))
* Similar to lenses: accessors to multiple substructures at the same time
  + Simultaneous retrieval and updating
  + BUT Biplates require that the first [b] is of the same length as the second [b]
* What the creators actually wanted to write was:
* **type** Biplate α β = a -> ∃n :: N. βn×(βn→α)
* Recall the Store b a data type: Store { peek :: b -> a, pos :: b }
* Both are comonads

**Cartesian Store Comonad**

* Could use GADTs to express ∃n :: N. βn×(βn→α) in Haskell
* Van Laarhoven’s data type uses nested data types, no GADTs required:
* data CartesianStore b a = Unit a | Battery (CartesianStore b (b -> a)) b
* Recall again the Store b a data type: Store { peek :: b -> a, pos :: b }
* Looks pretty similar, but a Cartesian Store has multiple dimensions
* Hence, Cartesian Store items are indexed by a coordinate system of some dimension

Example

* Dimension = 2
* Selected location = {b2, b3}
* Comonadic operations for the Cartesian Store:

instance Functor (CartesianStore β) where

fmap f (Unit a) = Unit (f a)

fmap f (Battery v b) = Battery (fmap (f ◦ ) v) b

instance Comonad (CartesianStore β) where

extract (Unit a) = a

extract (Battery v b) = extract v b

duplicate (Unit a) = Unit (Unit a)

duplicate (Battery v b) = Battery (extend Battery v) b – Adds another dimension

instance Applicative (CartesianStore β)where

pure6Unit

f h∗i (Unit a) 6fmap ($a)f

f h∗i (Battery vb) Battery ((◦) h$i f h∗i v) b

* Cartesian store is a generalization of a store
* Every store can be transformed into a Cartesian Store with *singleStore*:

singleStore ;Store βα→CartesianStore βα

singleStore (Store vb)6Battery (Unit v) b

* Stripping a dimension is also possible

stripDimension;CartesianStore βα→Maybe (Store βα×CartesianStore βα)

stripDimension (Unit a) 6Nothing

stripDimension (Battery vb) Just (Store (extract v) b, fmap ($b) v)

[NOTE: Also check unfold stripDimension, understand what it does]

**Biplates as a Cartesian Store**

* So, we know that we can define a comonad of which the function component only accepts inputs that are the same length as the data component of the value
* So we can define a Biplate as such:

**type** Biplate α β = α →CartesianStore β α

* This means Biplates must be coalgebras of the Cartesian store comonad
* So we need to satisfy the two coalgebraic laws (we defined these in the Store Comonad slides):

extract ◦l = id

fmap l ◦ l = duplicate ◦l

* Actually, a Biplate is a generalization of Lens:
* A Lens is a reference to a substructure inside a large structure
* A Biplate is a reference to a list of 0 (or more) substructures inside a large structure
* Thus, a Lens is a Biplate where the amount of substructures referenced is exactly one

[NOTE: Overview on all the isomorphisms and such???]

**Overview**

* Lenses can be defined as Stores
* Cartesian Stores are generalizations of Stores
* Biplates can be defined as Cartesian Stores
* Biplates are generalizations of Lenses

So we can see there are a lot of relations between all these structures. Let’s tie the knot.

**Tying the knot**

* Another representation of a Store A B is:

∀κ : Functor. (B→κB)→κA

[NOTE: If we have time, this would be great for class interaction!]

* Why this representation? 🡪 The parametricity will greatly restrict what values of this data type can be.
* How can we produce a value of κA?
* The only tool we can use is fmapk
* We have to utilize our parameter f :: B→κB
* So, our type needs to internally have a value b :: B, and apply it to f, yielding a κB.
* But how can we transform our κB to a κA?
* Our type will also have to hold a value v :: B → A, so we can use fmapk v to get a κA
* In conclusion, we need a B and a B → A.
* These are exactly the two components of Store B A!

Data Store b a = Store { peek :: b -> a, pos :: b }

isoStore1;StoreBA→∀κ;Functor. (B→κB)→κA

isoStore1 (Store vb)6\_κ→λf→fmap\_ v (fb)

isoStore2;(∀κ;Functor. (B→κB)→κA)→StoreBA

isoStore2 y6y(StoreB) idLensB

* This isomorphism implies the following set of isomorphisms:

StoreBA ≈ ∀κ;Functor. (B→κB)→κA

Lens AB ≈ A→∀κ;Functor. (B→κB)→κA

Lens AB ≈ ∀κ;Functor. (B→κB)→A→κA (van Laarhoven representation for a lens)

* So, we can transport lens laws through the isomorphism to get laws for van Laarhoven lenses
* Reasoning this way is awkward
* Is there a more natural way? Yes!
* **type** Coalgebra ακ6α→κα
* 🡺Arguments are flipped: Coalgebra α :: (⋆→ ⋆)→ ⋆
* **type** κ1⇒κ26∀α.κ1 α→κ2 α
* 🡺Natural transformation from functor k1 to functor k2

coalgMap;(κ1⇒κ2)→Coalgeba βκ1→Coalgebra βκ2

coalgMap ηc6η◦c

We can rewrite the the type of a van Laarhoven lens:

From l :: ∀κ;Functor. (B→κB)→A→κA

To l :: ∀κ: Functor. Coalgebra Bκ→Coalgebra Aκ

* Hmmm, it has the type of a natural transformation from Coalgebra B to Coalgebra A
* We get a free theorem: coalgMap η◦l=l◦ coalgMap η [NOTE: Why is it free?]
* Both the category of functors and the category of types are monoidal categories
* Types: 1 is the identity function and X (Cartesian product) is the binary operation
* Functors: Id is the identity function and ◦is the binary operation
* Coalgebras preserve this monoidal structure with the following operations

idCoalg ;1→Coalgebra α Id

idCoalg ()6id

composeCoalg;(Functor κ1, Functor κ2)⇒(Coalgebra ακ1×Coalgebra ακ2)→

Coalgebra α (κ1 ◦ κ2)

composeCoalg (c1, c2)6fmap\_1 c2 ◦c1

* Given that CoAlgebra A and Coalgebra B are both monoidal functors, we note that l is actually a monoidal natural transformation
* The laws for a monoidal natural transformation lens are the following

l (idCoalg ()) = idCoalg () (3)

l (composeCoalg (c1, c2)) = composeCoalg (lc1, lc2)

* This is a crucial part here. Let’s recall the laws of (1) and (2):
  + get *l* (set *l s b*) = *b*
  + set *l s* (get *l s*) = *s*
  + set *l* (set *l s b1*) *b2* = set *l s b2*
  + extract ◦ *l* = id
  + fmap *l* ◦ *l* = dupliate ◦ *l*
* It turns out that (3) is satisfied iff (2) holds, and (1) holds iff (2) holds.
* Thus, (1), (2) and (3) are all equivalent
* Three different characterizations of two different representations of a lens structure

**Van Laarhoven Biplates**

* Similarly, CartesianStore A B can be represented as
* ∀κ;Applicative. (B→κB)→κA
* Again, parametricity restricts what these values can do
* But now, we can also use pure and <\*>
* So, if our value contains an a :: A, we can just return purek a
* On the other hand, if our value contains a b :: B and a function v :: B 🡪 A, we can return purek v <\*>k f b (where f is the B → κB argument)
* Because fmap is defined as fmap\_ vx=pure\_ v h∗i\_ x, it is essentially identical to the lens case
* Another venue:
  + Two values b1, b2 :: B
  + Function v :: B -> B -> A
  + We would return pure\_ v h∗i\_ fb1 h∗i\_ fb2
  + We could keep adding B’s
  + In essence, we have the following type

A+(B→A)×B +(B2→A)×B2+ … ≈ ∃n;N.(Bn→A)×Bn ≈ CartesianStoreBA

* We can produce a value of the polymorphic type like so

∀κ :: Applicative. (B→κB)→κA

isoCartesianStore1;CartesianStoreBA→(∀κ;Applicative. (B→κB)→κA)

isoCartesianStore1 (Unit a) 6\_κ→λf→pure\_ a

isoCartesianStore1 (Battery vb) \_κ→λf→(isoCartesianStore1 v)\_ f h∗i\_ fb

* Van Laarhoven had a feeling that CartesianStore A B and ∀κ;Applicative. (B→κB) →κA are isomorphic
* They are

isoCartesianStore2;(∀κ;Applicative. (B→κB)→κA)→CartesianStoreBA

isoCartesianStore2 y6y(CartesianStoreB) idBiplateB

* This isomorphism again implies the following set of isomorphisms

CartesianStoreBA ≈ ∀κ;Applicative. (B→κB)→κA

BiplateBA ≈ A→∀κ;Applicative. (B→κB)→κA

BiplateBA ≈ ∀κ;Applicative. (B→κB)→A→κA

* Again, coalgebra laws for Biplates are equivalent to the laws for a monoidal natural transformation
* It turns out that ∀κ;Applicative. (A→κA) →A→κA is *exactly* the type of compos from the Compos library
* Hence, Uniplate and Compos use (in essence) isomorphic representations

**Multiplate!**

* Let’s look at the van Laarhoven representation of a Biplate again

∀κ;Applicative. (B→κB)→(A→κA)

* Suppose we want to extend the type to support different types of substructures of A (which is the main structure)
* The natural way would be to add more parameters:

∀κ;Applicative. (B→κB)→(C→κC)→(A→κA)

* But if A, B and C are mutually recursive data types, we’ll need three different multireference types:

∀κ;Applicative. (A→κA)→(B→κB)→(C→κC)→(A→κA)

∀κApplicative. (A→κA)→(B→κB)→(C→κC)→(B→κB)

∀κ;Applicative. (A→κA)→(B→κB)→(C→κC)→(C→κC)

* This will get ugly fast!
* Compos supports mutually recursive data types through GADTs
* Uniplate resorts to multiparameter type classes
* But Uniplate supports updating only a single pair of parent-child types at a time
* Ideally, we do not want to require GADTs or multiparameter type classes
* To do this, we will use rank-3 polymorphism
* Instead of creating three multireference types, we can combine them
* This “matrix transformation” will operate on a “vector” of coalgebras
* Let us define a record type parametrized by applicative functions:

data Pκ6



coalgA ;A→κA

coalgB B→κB

coalgC ;C→κC



* Then, we can just write a single type:

∀κ;Applicative. P κ→P κ.

* We can give generic implementations of traversal operations for a given P
* Let’s take an example data type

**Example Language**

data Stm6SDecl Typ Var OSAss Var Expr SBlock [Stm] OSReturn Expr

data Expr6EStm Stm OEAdd Expr Expr EVar Var OEInt Int

data Var6VString

data TypTInt OTFloat

* What now?
* 🡪 For each type, we need a record type with a field for a coalgebra
* This record is then parametrized by an applicative functor
* Let’s call this a **plate**

data Plate κ6Plate



stm ;Stm →κ Stm

expr Expr →κ Expr

var ;Var →κ Var

typ Typ →κ Typ



* Let’s provide the functional multireference (our **multiplate**)

multiplate;Applicative κ⇒Plate κ→Plate κ

multiplate p6Plate



stm 6buildStm

expr buildExpr

var 6buildVar

typ buildTyp



where

buildStm (SDecl tv) 6SDecl h$i typ pt h∗i var pv

buildStm (SAss ve) SAss h$i var pv h∗i expr pe

buildStm (SBlock *ss*) 6SBlock h$i traverse (stm p) *ss*

buildStm (SReturn e) SReturn h$i expr pe

buildExpr (EStms) 6EStm h$i stm ps

buildExpr (EAdd e1 e2) EAdd h$i expr pe1 h∗i expr pe2

buildExpr (EVar v) 6EVar h$i var pv

buildExpr x pure x

buildVar x 6pure x

buildTyp x pure x

* Now, with multiplate, we can for example define a collection of rename functions

rename;Plate Id

rename6Plate



stm 6stm(multiplate rename)

expr expr (multiplate rename)

var 6λ(Vs)→pureId (V (’\_’: s))

typ typ (multiplate rename)



* These recursive calls will rename all Stm and Expr
* This is the way Compos does it
* Uniplate does it differently:

mapFamily7;Plate Id→Plate Id

mapFamily p6p ‘composePlateId‘multiplate (mapFamily p)

where

composePlateId;Plate Id→Plate Id→Plate Id

p1 ‘composePlateId‘ p26Plate



stm 6stm p1 ◦ stm p2

expr expr p1 ◦ expr p2

var 6var p1 ◦ var p2

typ typ p1 ◦ typ p2



purePlate;Applicative κ⇒Plate κ

purePlate6Plate



stm 6pure

expr pure

var 6pure

typ pure



rename6mapFamily (purePlate {var6renameVar})

where

renameVar (V s)6pure (V(’\_’: s))

kleisliComposePlate;Monadm⇒Platem→Platem→Platem

p1 ‘kleisliComposePlate‘ p26Plate



stm 6stm p1<=<stm p2

expr expr p1<=<expr p2

var 6var p1<=<var p2

typ typ p1<=< typ p2



mapFamilyM;Monadm⇒Platem→Platem

mapFamilyM p6p ‘kleisliComposePlate‘multiplate (mapFamilyM p)

∀α.α→κα

type Projector ρα6∀κ. ρκ→α→κα

mkPlate;(∀α. Projector pα→α→κα)→pκ

mkPlate *build*6Plate



stm 6*build* stm

expr *build* expr

var 6*build* var

typ *build* typ



classMultiplate ρwhere

multiplate ;Applicative κ⇒ρκ→ρκ

mkPlate (∀α. Projector ρα→α→κα)→ρκ

purePlate;(Multiplateρ, Applicative κ)⇒ρκ

purePlate6mkPlate (const pure)

idPlate;Multiplateρ⇒ρ Id

idPlate6purePlate

mapPlate;∀ρκ1 κ2.Multiplate ρ⇒(∀γ.κ1γ→κ2γ)→ρκ1→ρκ2

mapPlate ηp6mkPlate build

where

build;Projector ρα→α→κ2 α

build π6η ◦ πp

composePlate;∀ρκ1 κ2.(Multiplate ρ, Applicative κ1, Applicative κ2)⇒ρκ1→ρκ2→ρ (κ2 ◦ κ1)

p1 ‘composePlate‘ p26mkPlate build

where

build;Projector ρα→α→κ2 (κ1 α)

build π6fmap2 (πp1) ◦ πp2

kleisliComposePlate;(Multiplate ρ,Monadm)⇒ρm→ρm→ρm

p1 ‘kleisliComposePlate‘ p26mapPlate join (p1 ‘composePlate‘ p2)

mapFamilyM;(Multiplate ρ,Monadm)⇒ρm→ρm

mapFamilyM p6p ‘kleisliComposePlate‘multiplate (mapFamilyM p)

instance (Monoid o)⇒Applicative (Const o)where

pure x61o

f h∗i xf ∗o x

appendPlate;∀ρo. (Multiplate ρ,Monoid o)⇒ρ (Const o)→ρ (Const o)→ρ (Const o)

p1 ‘appendPlate‘ p26mkPlate build

where

build;Projector ρα→α→mα

build πa6πp1a h∗ πp2 a

preorderFold ;(Multiplate ρ,Monoid o)⇒ρ (Const o)→ρ (Const o)

preorderFold p6p ‘appendPlate‘multiplate (preorderFold p)

postorderFold ;(Multiplate ρ,Monoid o)⇒ρ (Const o)→ρ (Const o)

postorderFold p6multiplate (postrderFold p) ‘appendPlate‘

multiplate idPlate = idPlate

multiplate (composePlate p1 p2) = composePlate (multiplate p1) (multiplate p2)