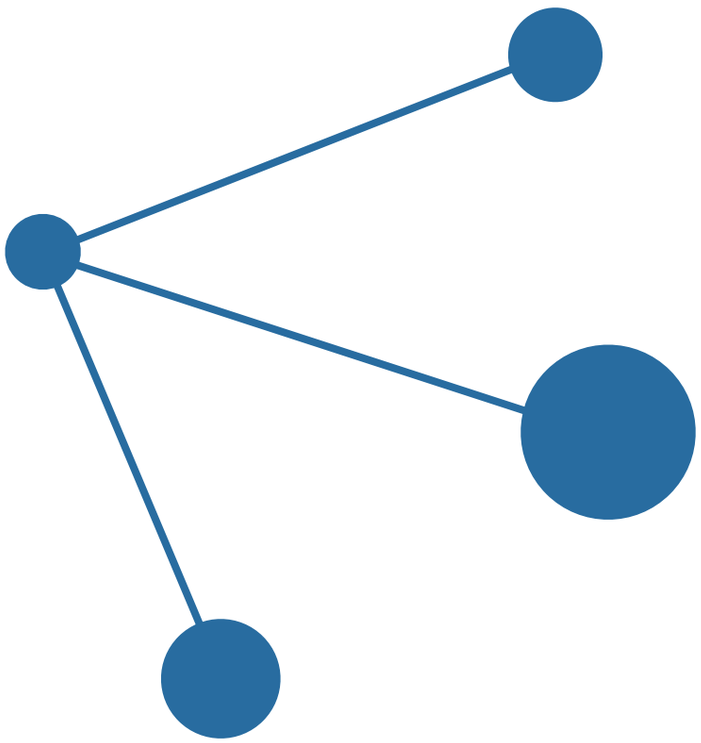


Why I Like Functional Programming

Adelbert Chang
Box, Inc.

Let's go back..

http://gauss.cs.ucsb.edu/home/images/UCSB-from-air.jpg

Timeline

2010
Entered UCSB as CS major.

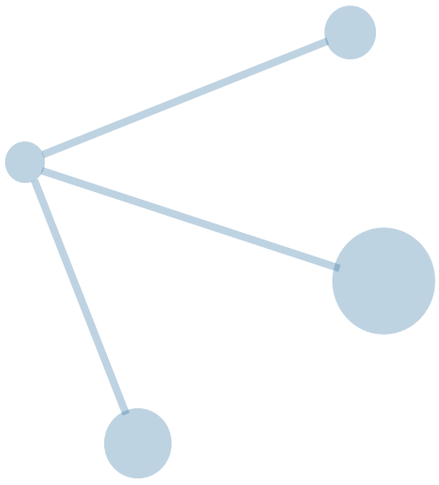
2011
Took first two college physics courses. Considered changing to physics.

2012
Took upper-division math courses. Considered minoring in mathematics. Started learning Scala.

2013
Started learning Haskell. Took graduate static analysis course.

2014
Graduated UCSB as a CS major. Working at Box in a functional codebase.

2015
Presenting at an awesome FP conference.



Why CS

- Computers are pretty cool
- Applied problems to solve
- Fairly interactive
- Rooted in logic
- Can be done pretty much anywhere

2011

- Languages learned/used: Python, C, C++
- Took first two college physics courses
    - Considered switching major to physics

```
swapped = True
while swapped:
    swapped = False
    for i in range(len(xs) - 1):
        if xs[i] > xs[i + 1]:
            xs[i], xs[i + 1] = xs[i + 1], xs[i]
            swapped = True
```

Physics and Math

- Problem solving activity
- Can be done pretty much anywhere
- Simplicity and consistency across problems
- Example: Deriving law of motion from first principles
  - $x = x_0 + v_0t + \frac{1}{2}at^2$

Physics and Math

Deriving $x = x_0 + v_0t + \frac{1}{2}at^2$

$v_{avg} = \frac{1}{2}(v_f + v_0)$

$\frac{1}{2}(v_0 + a\Delta t + v_0)$

$\frac{1}{2}(2v_0 + a\,\Delta t)$

$v_{avg} = v_0 + \frac{1}{2}a\Delta t$

$a = \Delta v / \Delta t$

$a = (v_f - v_0) / \Delta t$

$v_f - v_0 = a\,\Delta t$

$v_f = v_0 + a\,\Delta t$

$v_{avg} = \Delta x/\Delta t$

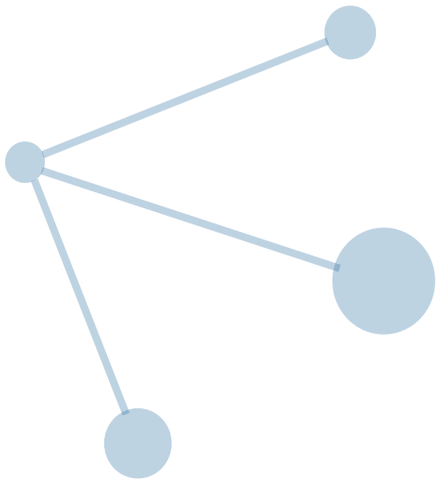
$\Delta x = v_{avg} * \Delta t$

$(x - x_0) = v_{avg} * \Delta t$

$x = x_0 + (v_{avg} * \Delta t)$

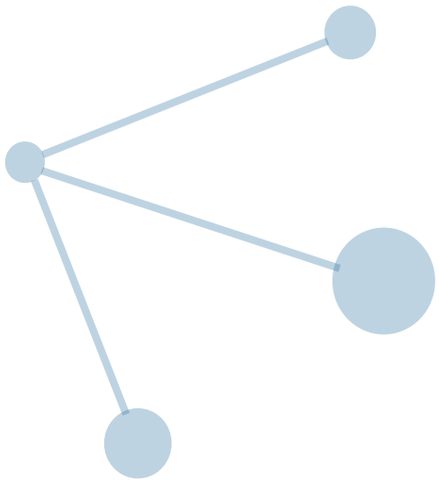
$x = x_0 + ((v_0 + \frac{1}{2}a\Delta t) * \Delta t)$

$x_0 + v_0t + \frac{1}{2}at^2$



2012

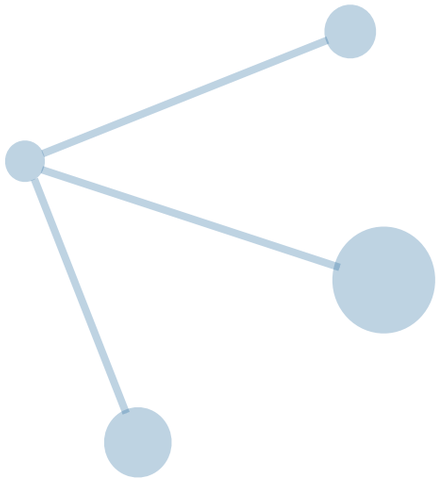
```
void insert(Node* &tree, int value) {
    if (!tree) return;

    Node *trav = tree, *parent;
    while (trav) {
        parent = trav;
        if (value < trav->data) trav = trav->left;
        else if (value > trav->data) trav = trav->right;
        else break;
    }
    if (value < parent->data)
        parent->left = new Node(value);
    else
        parent->right = new Node(value);
}
```



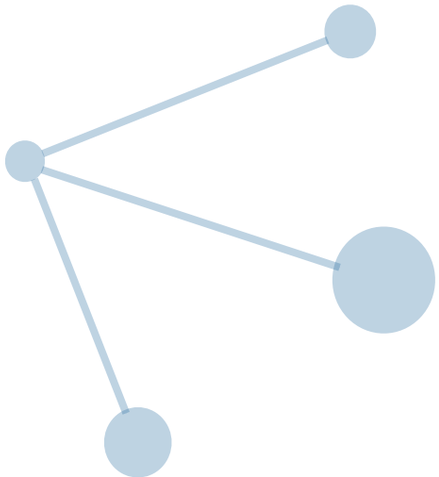
2012

```
sealed abstract class Tree
case class Branch(data: Int, left: Tree, right: Tree)
  extends Tree
case class Leaf() extends Tree
```

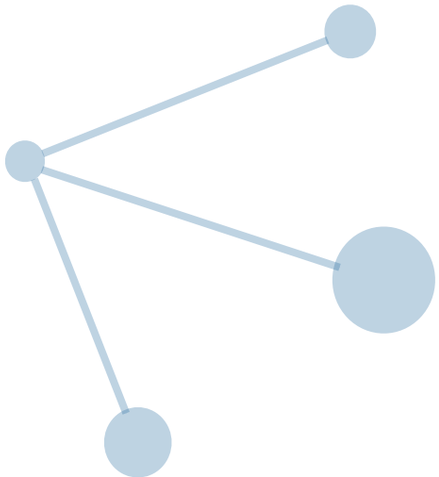
```
def insert(tree: Tree, value: Int): Tree =
  tree match {
    case Leaf() => Branch(value, Leaf(), Leaf())
    case b@Branch(d, l, r) =>
      if (value < d) Branch(d, insert(l, value), r)
      else if (value > d) Branch(d, l, insert(r, value)
      else b
  }
```



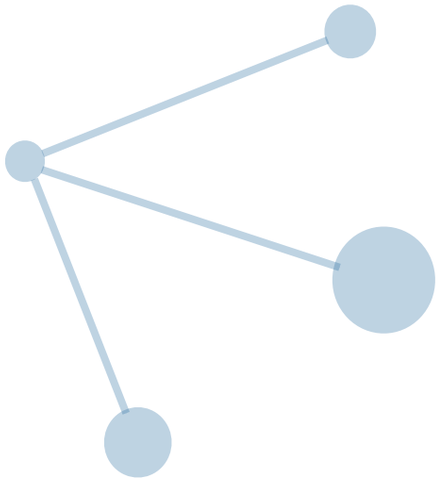
Hmm.. functional programming is pretty cool.



But is it as cool as physics and math?

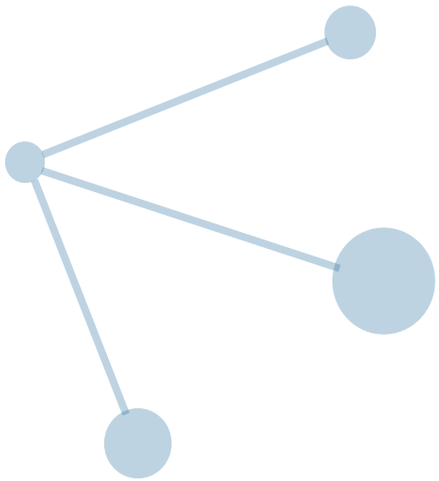


Let's see..



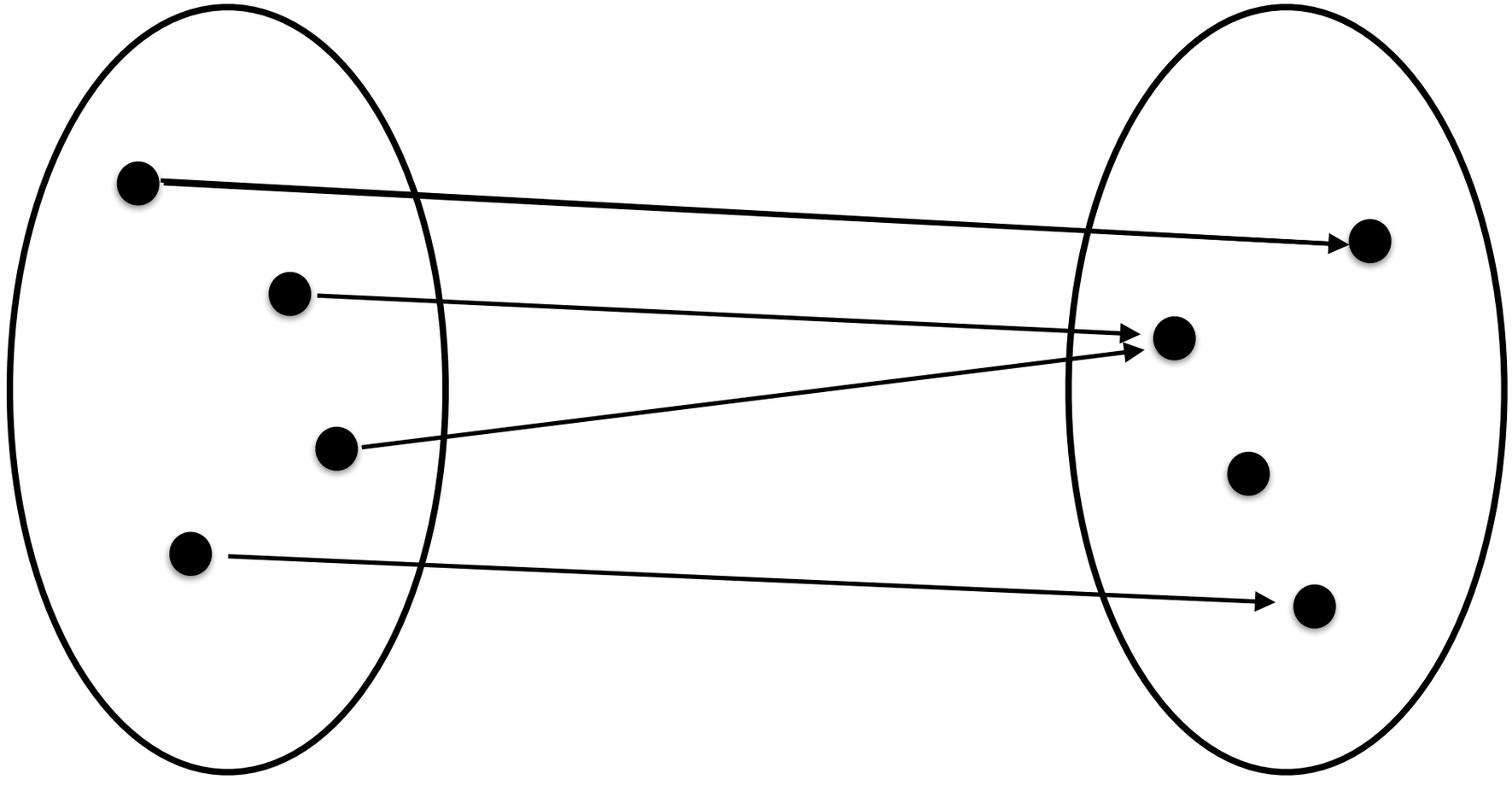
Functional Programming

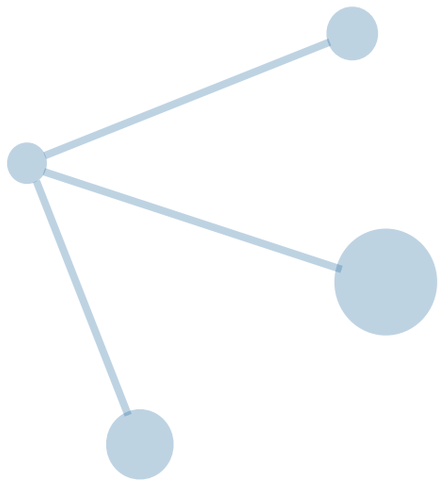
programming with functions



Functional Programming

programming with **pure** functions

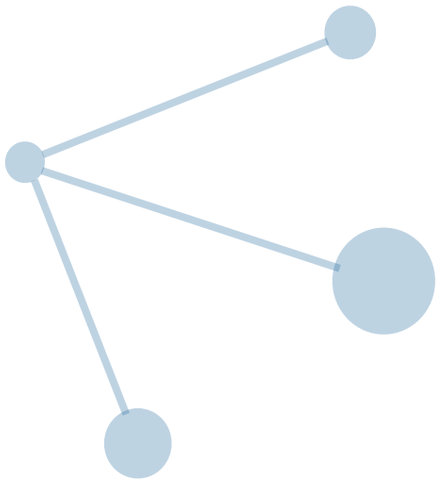




Functions

```
def parseIntPartial(s: String): Int =
  s.toInt // can throw NumberFormatException
```

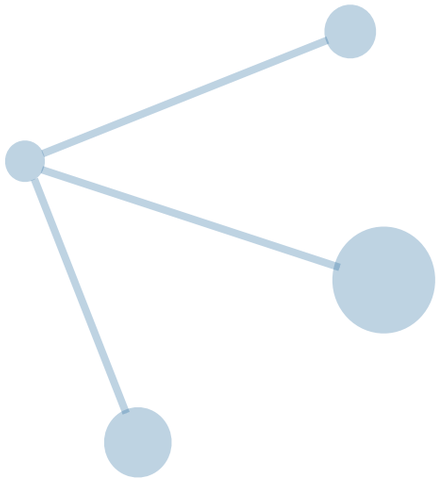
```
def parseIntTotal(s: String): Option[Int] =
  try {
    Some(s.toInt)
  } catch {
    case nfe: NumberFormatException => None
  }
```



Functions

```
class Rng(var seed: Long) {
  def nextInt(): Int = {
    val int = getInt(seed)
    mutate(seed)
    int
  }
}
```

```
case class Rng(seed: Long) {
  def nextInt: (Rng, Int) = {
    val int = getInt(seed)
    val newSeed = f(seed)
    (Rng(newSeed), int)
  }
}
```



Referential Transparency

An expression **e** is referentially transparent if
for all programs **p**
every occurrence of **e** in **p**
can be replaced with the result of evaluating **e**
without changing the result of evaluating **p**.*

* Taken from Higher Order blog post "What Purity Is and Isn't" May 29, 2015 http://blog.higher-order.com/blog/2012/09/13/what-purity-is-and-isnt/

Referential Transparency

$x^4 - 12x^2 + 36 = 0$

$y = x^2$

$y^2 - 12y + 36 = 0$

$y = 6$

$x^2 = 6$

$x = \pm \sqrt{6}$

$ax^2 + bx + c = 0$

$(-b \pm \sqrt{b^2 - 4ac}) / 2a$

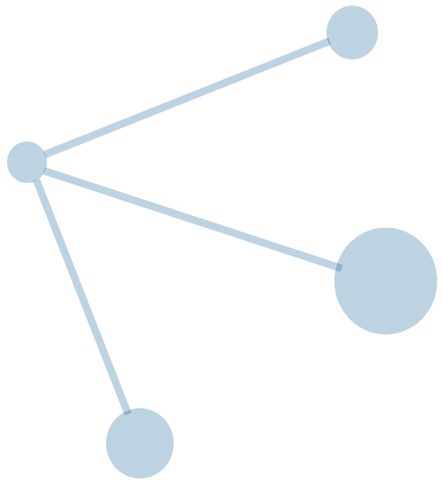
$a = 1$

$b = -12$

$c = 36$

$(-(-12) \pm \sqrt{(-12)^2 - 4(1)(36)}) / 2(1)$

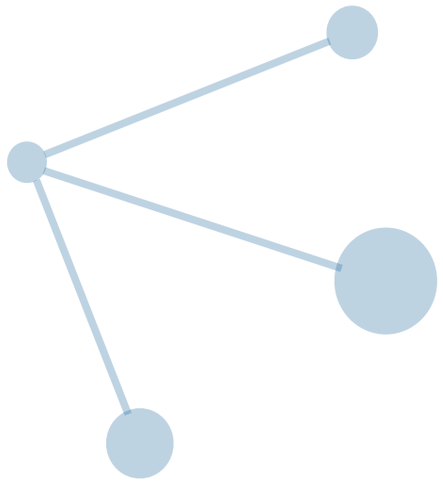
$12 / 2$



Referential Transparency

```
def userInfo(id: UserId): Future[UserData]

val userId = . . .

val fetchData = userInfo(userId)

fetchData.retry {
  case StatusCode(429) => fetchData
}
```

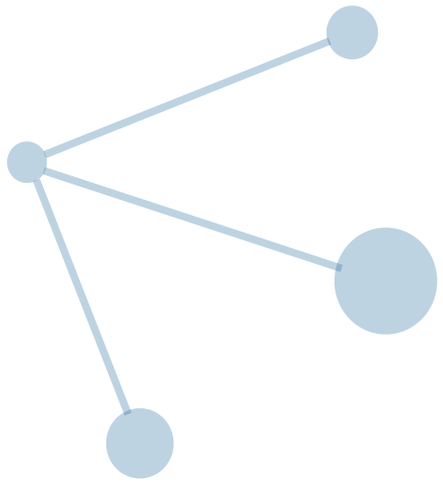
* Taken from Higher Order blog post "What Purity Is and Isn't" May 29, 2015 http://blog.higher-order.com/blog/2012/09/13/what-purity-is-and-isnt/



Referential Transparency

```
def userInfo(id: UserId): Future[UserData]

val userId = . . .

val fetchData = userInfo(userId)

fetchData.retry {
  case StatusCode(429) => userInfo(userId)
}
```

* Taken from Higher Order blog post "What Purity Is and Isn't" May 29, 2015 http://blog.higher-order.com/blog/2012/09/13/what-purity-is-and-isnt/



Referential Transparency

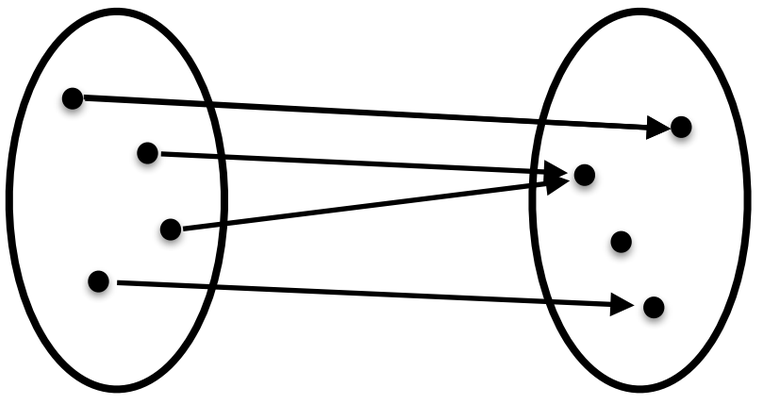
```
def userInfo(id: UserId): Task[UserData]

val userId = . . .

val fetchData = userInfo(userId)

fetchData.retry {
  case StatusCode(429) => fetchData
}
```

```
trait Task[A] {
  def unsafeRun(): A
}
```

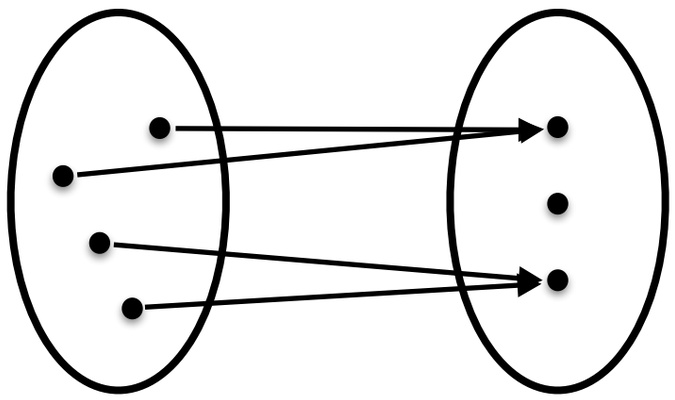
* Taken from Higher Order blog post "What Purity Is and Isn't" May 29, 2015 http://blog.higher-order.com/blog/2012/09/13/what-purity-is-and-isnt/

Functions

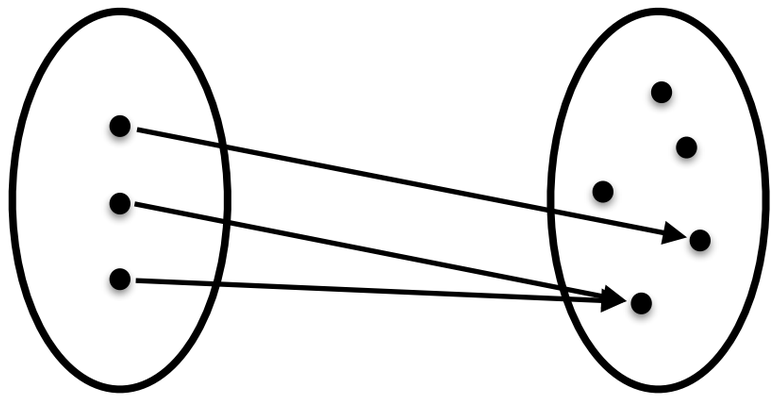
foo: A => B



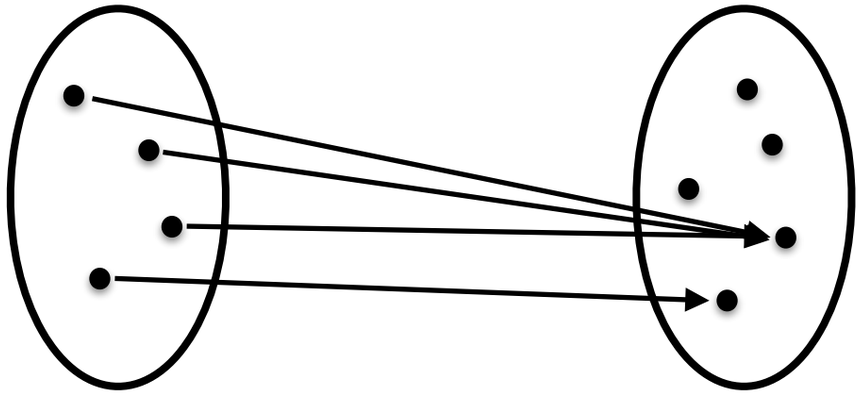
bar: B => C

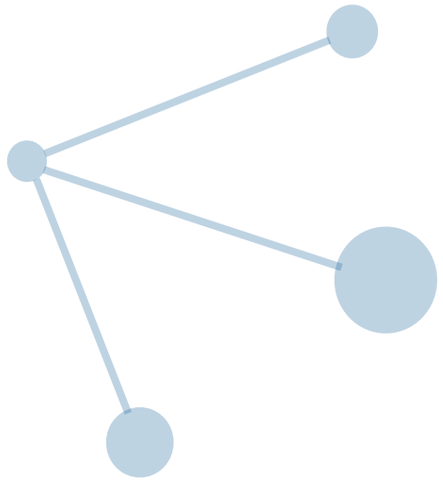


baz: C => D



baz.compose(bar).compose(foo): A => D

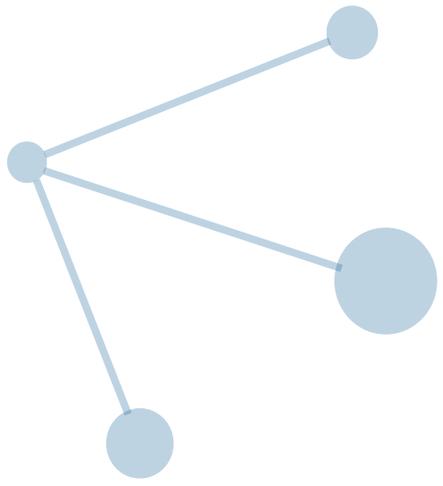




Tracking Effects

- Interesting programs will have effects
  - Optional values, error handling, asynchronous computation, input/output
  - Usual means aren't quite nice

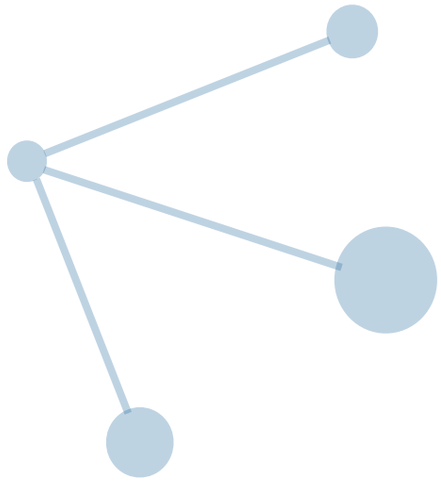
```
def lookup(map: Map[Foo, Bar], foo: Foo): Bar =
  if (map.contains(foo)) map(foo) else null

val bar = lookup(someMap, someFoo)
if (bar != null) {
  . . .
} else {
  . . .
}
```



Tracking Effects

- Interesting programs will have effects
  - Optional values, error handling, asynchronous computation, input/output
  - Usual means aren't quite nice

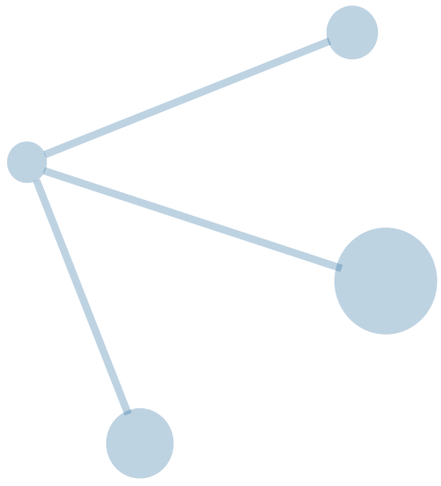
```
def lookup(map: Map[Foo, Bar], foo: Foo): Bar =
  map(foo)

try {
  val bar = lookup(map, someFoo)
} catch {
  case e: NoSuchElementException => . . .
}
```



Tracking Effects

- Reify effects as values

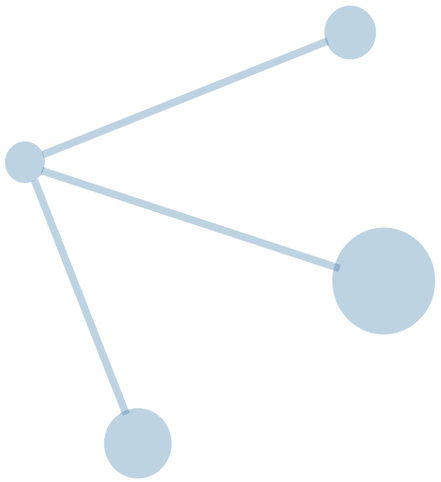
```scala
// A value that might exist is either..
sealed abstract class Option[A]

// ..there
final case class Some[A](a: A) extends Option[A]

// .. or not there
final case class None[A]() extends Option[A]
```



Tracking Effects

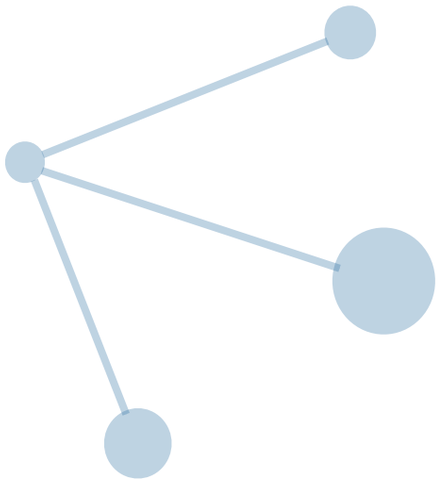
- Reify effects as values

```
def lookup(map: Map[Foo, Bar], foo: Foo): Option[Bar] =
  if (map.contains(foo)) Some(map(foo))
  else None
```



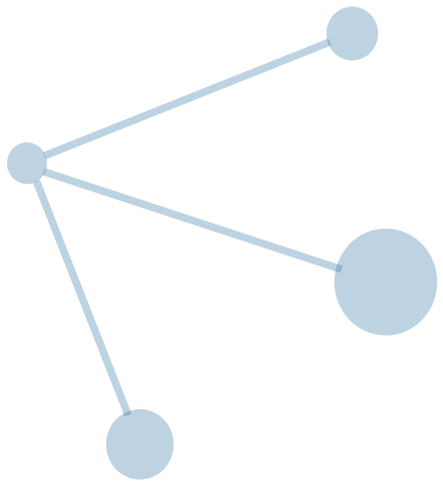
Missing values

```
sealed abstract class Option[A]

final case class Some[A](a: A) extends Option[A]

final case class None[A]() extends Option[A]
```



Errors

```
sealed abstract class Either[+E, +A]

final case class Success[+A](a: A) extends Either[Nothing, A]

final case class Failure[+E](e: E) extends Either[E, Nothing]
```



Ex. Errors

```
sealed abstract class Error
final case object UserDoesNotExist extends Error
final case object InvalidToken     extends Error

def userInfo(uid: UserId,
             tk: Token): Either[Error, UserInfo] = . . .
```

Manipulating Effects

- The type signature of our functions reflect the effects involved

```
def tokenFor(uid: UserId): IO[EncryptedToken]

def decrypt(token: EncryptedToken): Token

/** Client side */
val encrypted: IO[EncryptedToken] =
  tokenFor(. . .)

// Want EncryptedToken, have IO[EncryptedToken]
val decrypted = decrypt(???)
```

Manipulating Effects

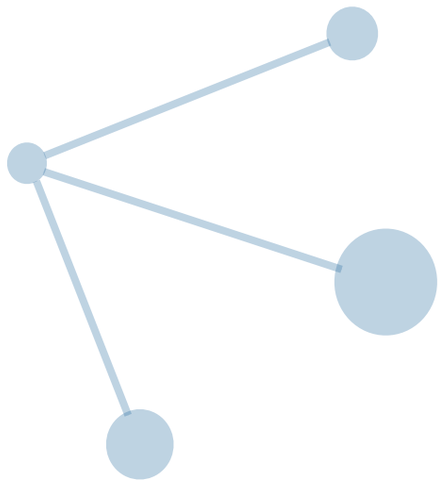
```scala
/** Apply a pure function to an effectful value */
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

```scala
new Functor[IO] {
  def map[A, B](fa: IO[A])(f: A => B): IO[B] =
    new IO[B] {
      def unsafePerformIO(): B =
        f(fa.unsafePerformIO())
    }
}
```

Manipulating Effects

```
def tokenFor(uid: UserId): IO[EncryptedToken]

def decrypt(token: EncryptedToken): Token

/** Client side */
val encrypted: IO[EncryptedToken] =
  tokenFor(. . .)

// Want EncryptedToken, have IO[EncryptedToken]
val decrypted: IO[Token] =
  encrypted.map(et => decrypt(et))
```

- Similar mechanisms for manipulating multiple effects

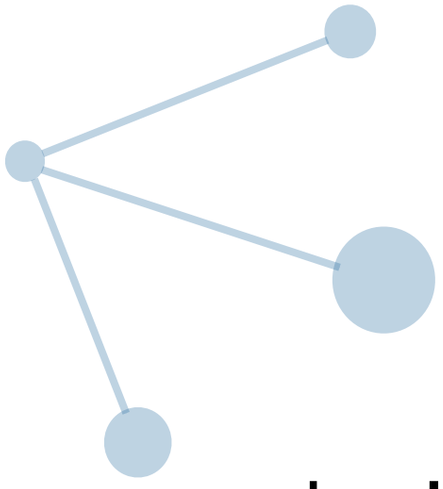


Lens

- Often want getters/setters when working with data
- If getters/setters are per-object, cannot compose
- As with effects, reify as data

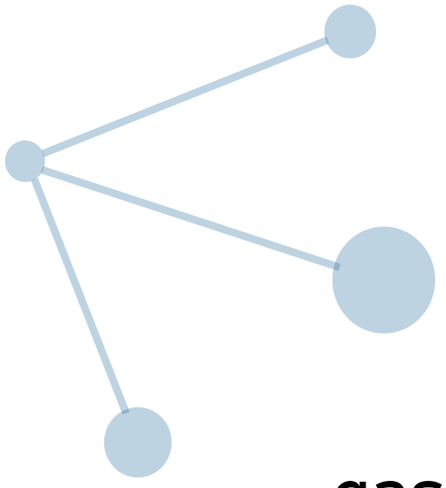
- Getter: get an **A** field in an object **S**
- Setter: change an **A** field in an object **S**

```
trait Lens[S, A] {
  def get(s: S): A

  def set(s: S, a: A): S
}
```



Lens

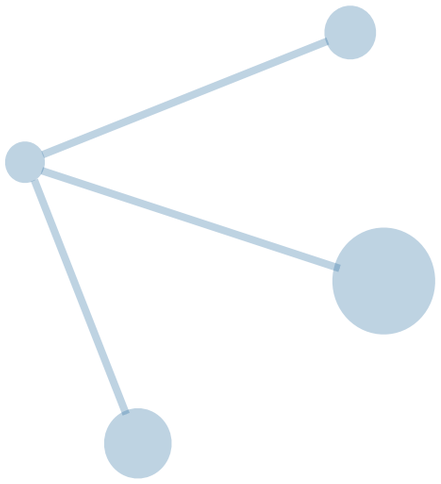
```
trait Lens[S, A] { outer =>
  def get(s: S): A

  def set(s: S, a: A): S

  def modify(s: S, f: A => A): S =
    set(s, f(get(s)))

  def compose[B](other: Lens[A, B]): Lens[S, B] =
    new Lens[S, B] {
      def get(s: S): B = other.get(get(s))

      def set(s: S, b: B): S =
        set(s, other.set(outer.get(s), b))
    }
}
```



Lens

```
case class Employee(position: Position)

object Employee {
  val position: Lens[Employee, Position] = . . .
}
```

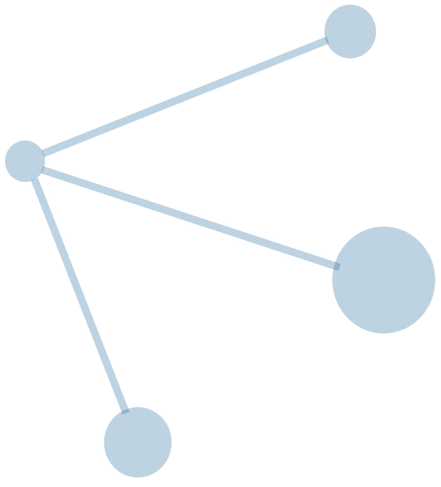
```
case class Team(manager: Employee, . . .)

object Team {
  val manager: Lens[Team, Employee] = . . .
}

/** Client side */
val l: Lens[Team, Position] =
  Team.manager.compose(Employee.position)
l.set(someTeam, somePosition)
```



Lens

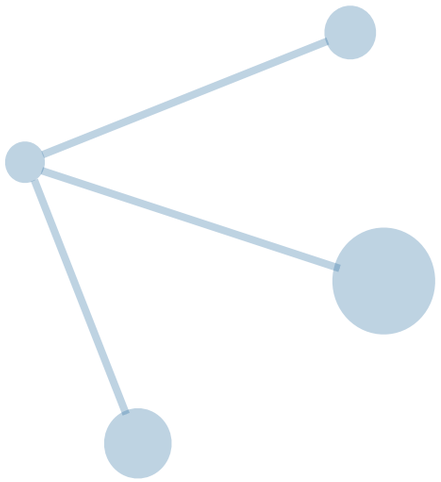
- Effect-ful modifications can be useful
  - Possibly failing: A => Option[A]
  - Many possible values: A => List[A]
  - Async: A => Task[A]

```
trait Lens[S, A] {
  def modifyOption(s: S, f: A => Option[A]): Option[S]

  def modifyList(s: S, f: A => List[A]): List[S]

  def modifyTask(s: S, f: A => Task[A]): Task[S]
}
```



Lens

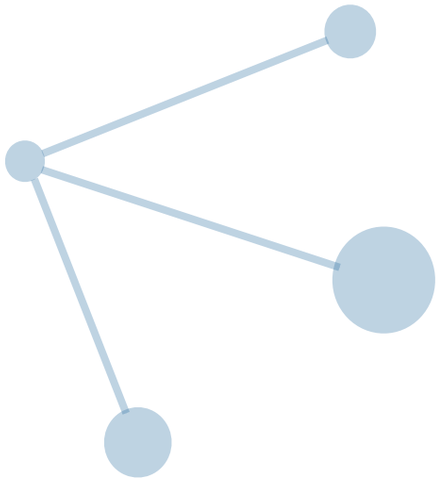
```
trait Lens[S, A] {
  def modifyOption(s: S, f: A => Option[A]): Option[S] =
    f(get(s)).map(a => set(s, a))

  def modifyList(s: S, f: A => List[A]): List[S] =
    f(get(s)).map(a => set(s, a))

  def modifyTask(s: S, f: A => Task[A]): Task[S] =
    f(get(s)).map(a => set(s, a))
}
```



Lens

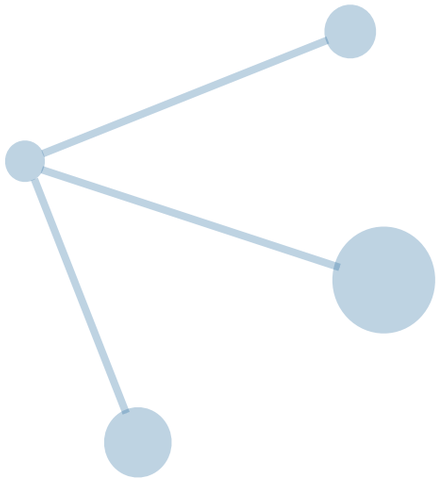
```
trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S, f: A => F[A]): F[S] =
    f(get(s)).map(a => set(s, a))
}
```

- Example functors:
  - Option, Either, Future, List, IO
- What if we want to just modify without any effect?



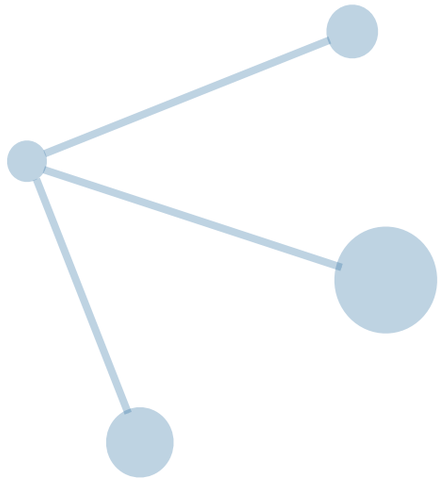
Id

```
type Id[A] = A

new Functor[Id] {
  def map[A, B](fa: Id[A])(f: A => B): Id[B] =
}
```



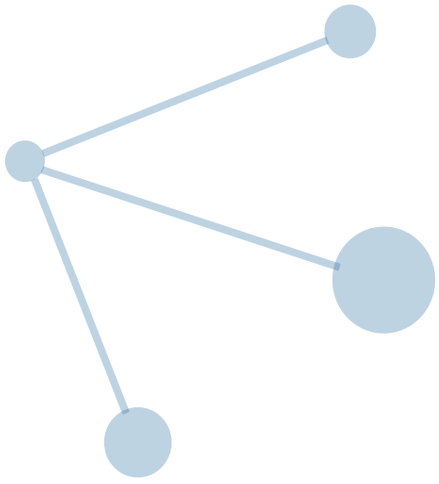
Id

```
type Id[A] = A

new Functor[Id] {
  def map[A, B](fa: A)(f: A => B): B =
}
```



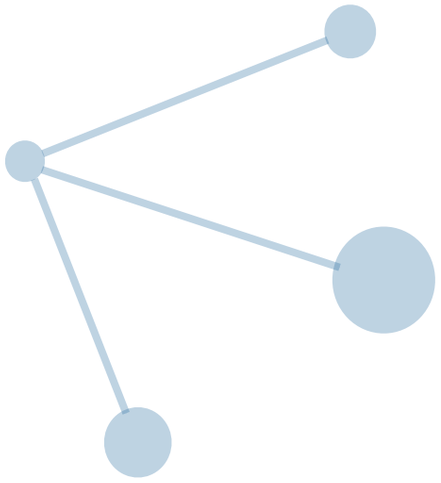
Id

```
type Id[A] = A

new Functor[Id] {
  def map[A, B](fa: A)(f: A => B): B = f(fa)
}
```



Id

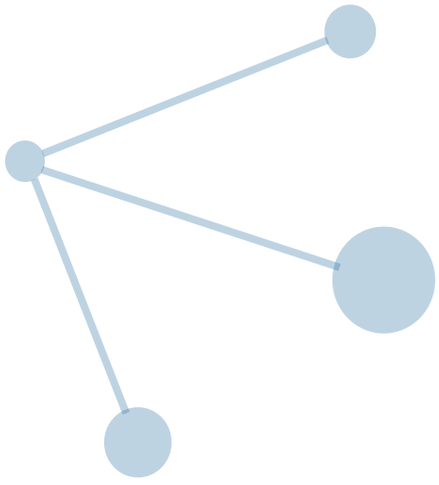
```
trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S, f: A => F[A]): F[S] =
    f(get(s)).map(a => set(s, a))

  def modify(s: S, f: A => A): S = modifyF[Id](s, f)
}
```



Setting

```
trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S, f: A => F[A]): F[S] =
    f(get(s)).map(a => set(s, a))

  def modify(s: S, f: A => A): S = modifyF[Id](s, f)

  def set(s: S, a: A): S = modify(s, const(a))
}
```

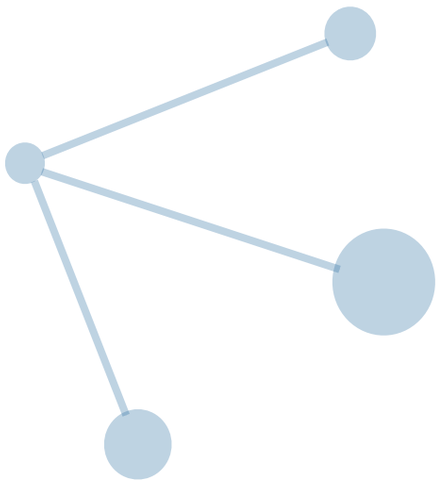
```
def const[A, B](a: A)(b: B): A = a
```



Setting

```
trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S, f: A => F[A]): F[S]

  def get(s: S): A

  def modify(s: S, f: A => A): S = modifyF[Id](s, f)

  def set(s: S, a: A): S = modify(s, const(a))
}
```

```
def const[A, B](a: A)(b: B): A = a
```

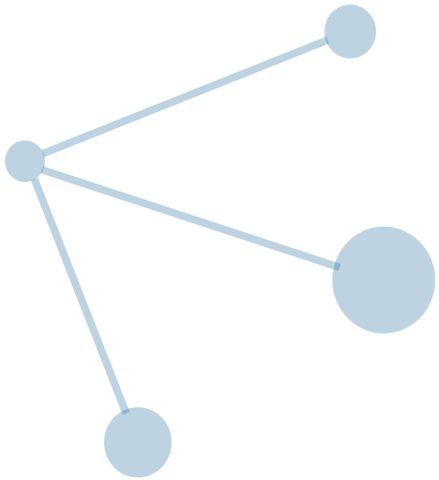


Getting

```
trait Lens[S, A] {
  def modifyF[F[_] : Functor](s: S, f: A => F[A]): F[S]

  def get(s: S): A
}
```

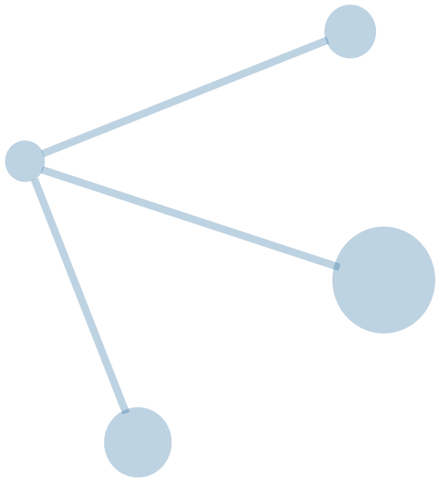
- Can we define `get` in terms of `modify`?
  - `modifyF` gives us some F[S].. but we want an A
  - We need some way of "ignoring" the S parameter and still get an A back
    - Type-level constant function

```
def const[A, B](a: A)(b: B): A = a
```



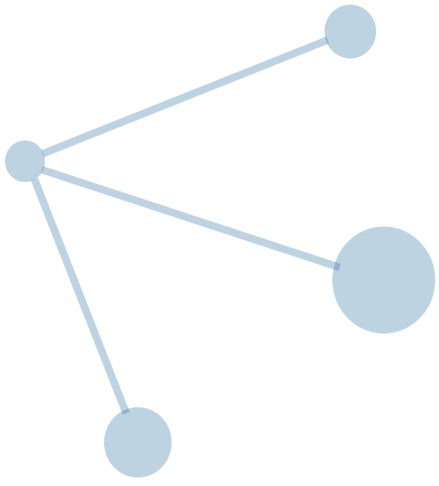
Getting

```
type Const[Z, A] = Z


new Functor[Const[Z, ?]] {
  def map[A, B](fa: Const[Z, A])(f: A => B): Const[Z, B]
}
```



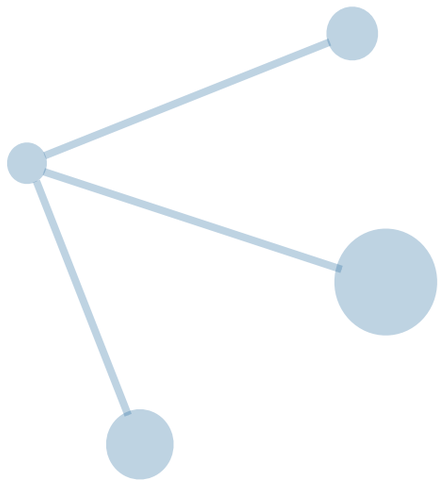
Getting

```
type Const[Z, A] = Z


new Functor[Const[Z, ?]] {
  def map[A, B](fa: Z)(f: A => B): Z =
}
```



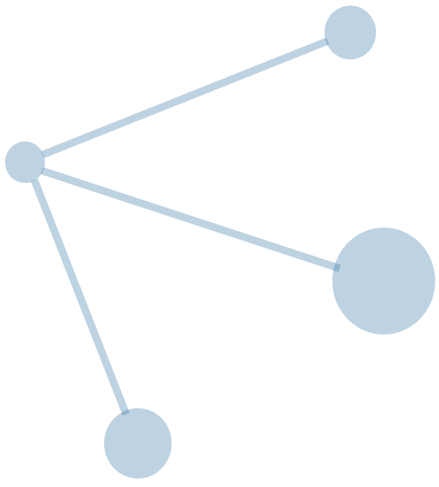
Getting

```
type Const[Z, A] = Z


new Functor[Const[Z, ?]] {
  def map[A, B](fa: Z)(f: A => B): Z = fa
}


trait Lens[S, Z] {
  def modifyF[F[_] : Functor](s: S, f: Z => F[Z]): F[S]

  def get(s: S): Z = {
    val const: Const[Z, S] = modifyF[Const[Z, ?]](s, z => z)
    const
  }
}
```



Interested?

- Functional Programming
  - Cats, Scalaz
  - Argonaut, Atto, Doobie, HTTP4S, Monocle, Remotely, Scalaz-stream, Shapeless, Typelevel.org
- Algebra
  - Algebird, Algebra, Breeze, Spire
- Big Data
  - Scalding + Algebird, Scoobi, Spark



EOF