# FP II (cs4621)

## Lecture 2: Generalising Map

M. R. C. van Dongen

January 19, 2018

# Capturing Patters

- FP lets us capture similar patterns in a simple way.
- E.g. consider the following functions:

**Haskell**

```
inc, sqr :: [Int] -> [Int]
inc []     = []
inc (i:is) = succ i : inc is
sqr []     = []
sqr (i:is) = (^2) i : sqr is
```

- We can easily rewrite them to:

**Haskell**

```
inc is = map succ is
sqr is = map (^2) is
```

- A partial application makes this a bit shorter:

**Haskell**

```
inc = map succ
sqr = map (^2)
```

# Functors

- Function mapping provides function calls to all members of a list.
- The signature of `map` is
  - `(a -> b) -> [] a -> [] b`.
- Wouldn't it be nice if we could generalise this to the following:
  - `(a -> b) -> f a -> f b`,
  - Where `f` is some constructor type?
- For `Functor` instances `f` we can do exactly that:
  - Apply a function to each element of the structure of an `f` instance.

# Functors (Continued)

## Haskell

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

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just a) = Just $ f a

data Tree a = Empty | Node a (Tree a) (Tree a)

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)

instance Functor IO where
    fmap f mx = do { x <- mx; return (f x) }
```

# Functors (Continued)

## Haskell

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

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just a) = Just $ f a

data Tree a = Empty | Node a (Tree a) (Tree a)

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)

instance Functor IO where
    fmap f mx = do { x <- mx; return (f x) }
```

# Functors (Continued)

## Haskell

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

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just a) = Just $ f a

data Tree a = Empty | Node a (Tree a) (Tree a)

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)

instance Functor IO where
    fmap f mx = do { x <- mx; return (f x) }
```

# Functors (Continued)

## Haskell

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

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just a) = Just $ f a

data Tree a = Empty | Node a (Tree a) (Tree a)

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)

instance Functor IO where
    fmap f mx = do { x <- mx; return (f x) }
```

# Functors (Continued)

## Haskell

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

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just a) = Just $ f a

data Tree a = Empty | Node a (Tree a) (Tree a)

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)

instance Functor IO where
    fmap f mx = do { x <- mx; return (f x) }
```

# Functors (Continued)

## Generalising Function Application

**Haskell**

```
inc :: Functor f => f Int -> f Int
inc = fmap succ
```

- `inc Nothing` gives `Nothing`;
- `inc $ Just 3` gives `Just 4`;
- `inc [1,2,1]` gives `[2,3,2]`;
- ...

# Functor Laws

- Functors must satisfy the following two laws:
  - `fmap id      = id`
  - `fmap (g . h) = fmap g . fmap h`
- Your compiler cannot verify the laws are satisfied.

# Functor Laws (Continued)

## Haskell

```
instance Functor [] where
    -- fmap :: (a -> b) -> [] a -> [] b
    fmap g []     = []
    fmap g (x:xs) = g x : fmap g xs
```

## Don't Try This at Home

```
instance Functor [] where
    -- fmap :: (a -> b) -> [] a -> [] b
    fmap g []     = []
    fmap g (x:xs) = fmap g xs ++ [ g x ]
```

# Functor Laws (Continued)

## Why Doesn't this Work?

### Haskell

```
instance Functor [] where
    -- fmap :: (a -> b) -> [] a -> [] b
    fmap g []     = []
    fmap g (x:xs) = g x : fmap g xs
```

### Don't Try This at Home

```
instance Functor [] where
    -- fmap :: (a -> b) -> [] a -> [] b
    fmap g []     = []
    fmap g (x:xs) = fmap g xs ++ [ g x ]
```

# Applicative Functors

- Earlier on we generalised mapping.
  - Given a one-argument function of type `a -> b`;
  - Given a one-argument `Functor` constructor `f`;
  - Given an input with type `f a`;
  - We computed a result of type `f b`.
- `fmap :: Functor f => (a -> b) -> f a -> f b`.

# Applicative Functors

- Earlier on we generalised mapping.
  - Given a one-argument function of type `a -> b`;
  - Given a one-argument `Functor` constructor `f`;
  - Given an input with type `f a`;
  - We computed a result of type `f b`.
- `fmap :: Functor f => (a -> b) -> f a -> f b`.
- Can we generalise mapping to multiple arguments?
  - `fmap0 :: a -> f a`;
  - `fmap1 :: (a -> b) -> f a -> f b`;
  - `fmap2 :: (a -> b -> c) -> f a -> f b -> f c`;
  - ...
- If we could get default implementations this would be nice.

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Applicative Functors (Continued)

## We Define our Hierarchy Indirectly

- Define an `Applicative` class with variable `f` and two functions:
  - `pure :: a -> f a`; and
  - `(<*>) :: f (a -> b) -> f a -> f b`.
  - `<*>` is pronounced "applied over."
- Function `<*>` is left associative.
- As an example, if `f` is `Applicative`, we have
  - `(g :: f (a -> b -> c)) <*> (x :: f a) <*> (y :: f b)`
  - gives `(g :: f (a -> (b -> c))) <*> (x :: f a) <*> (y :: f b)`
  - gives `((g :: f (a -> (b -> c))) <*> (x :: f a)) <*> (y :: f b)`
  - gives `((g <*> x) :: f (b -> c)) <*> (y :: f b)`
  - gives `((g <*> x) <*> y) :: f c`.
- With `pure` and `<*>` we can build our function hierarchy:
  - `fmap0 :: a -> f a`
  - `fmap0 = pure`
  - `fmap1 :: Applicative f => (a -> b) -> f a -> f b`
  - `fmap1 g fa = pure g <*> fa`
  - `fmap2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c`
  - `fmap2 g fa fb = pure g <*> fa <*> fb`
  - ...

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
> pure succ <*> Just 1
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
> pure succ <*> Just 1
2
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
> pure succ <*> Just 1
2
> pure (+)  <*> Just 1 <*> Nothing
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
> pure succ <*> Just 1
2
> pure (+)  <*> Just 1 <*> Nothing
Nothing
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
> pure succ <*> Just 1
2
> pure (+)  <*> Just 1 <*> Nothing
Nothing
> pure (*)  <*> Just 2 <*> Just 21
```

# Example: Maybe

## Haskell

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just
    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    (<*>) Nothing  _  = Nothing
    (<*>) (Just f) mx = fmap f mx
```

## ghci

```
> Just succ <*> Just 1
2
> pure succ <*> Just 1
2
> pure (+)  <*> Just 1 <*> Nothing
Nothing
> pure (*)  <*> Just 2 <*> Just 21
42
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
> pure succ <*> [1,2,4]
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
> pure succ <*> [1,2,4]
[2,3,5]
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
> pure succ <*> [1,2,4]
[2,3,5]
> pure (+)  <*> [1,2] <*> []
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
> pure succ <*> [1,2,4]
[2,3,5]
> pure (+)  <*> [1,2] <*> []
[]
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
> pure succ <*> [1,2,4]
[2,3,5]
> pure (+)  <*> [1,2] <*> []
[]
> pure (*)  <*> [1,2,3] <*> [1,2]
```

# Example: Lists

## Haskell

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure a = [a]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    (<*>) gs xs= [ g x | g <- gs, x <- xs ]
```

## ghci

```
> [succ]    <*> [1,2,4]
[2,3,5]
> pure succ <*> [1,2,4]
[2,3,5]
> pure (+)  <*> [1,2] <*> []
[]
> pure (*)  <*> [1,2,3] <*> [1,2]
[1,2,2,4,3,6]
```

# The Applicative Aspect

- Consider the following definition:
  - `prods :: [Int] -> [Int] -> [Int]`
  - `prods as bs = [ a * b | a <- as, b <- bs ]`
- The comprehension has to name the members of the arguments.
- With `Applicative` instances we can avoid this naming.
- Instead we use a more applicative style:
  - `prods :: [Int] -> [Int] -> [Int]`
  - `prods as bs = pure (*) <*> as <*> bs`

# Example: IO

## Haskell

```
instance Applicative IO where
    -- pure :: a -> IO a
    pure = return
    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    (<*>) ig ix = do { g <- ig; x <- ix; return (g x) }
```

# Example: IO

## Haskell

```
instance Applicative IO where
    -- pure :: a -> IO a
    pure = return
    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    (<*>) ig ix = do { g <- ig; x <- ix; return (g x) }

getChars :: Int -> IO String
getChars 0 = return []
getChars i = pure (:) <*> getChar <*> getChars (i-1)
```

# Effectful Programming

- We generalised `map` for list arguments:
  - We got `fmap` for `Functor` arguments.
- We generalised `Functor` mapping to `Applicative` mapping:
  - Maps now have multiple arguments.
  - The operands of <*> may now also have effects:
    - They may fail or succeed;
    - They may perform I/O;
    - They may have zero, one, or several results.
- Regardless of the result, we can use an applicative style.
- We can write useful generic functions for applicative functors:
  - `sequenceA :: Applicative f => [f a] -> f [a]`
  - `sequenceA [] = pure []`
  - `sequenceA (a:as) = pure (:) <*> a <*> sequenceA as`

# One More Example

## Haskell

```
getChars :: Int -> IO String
getChars n = sequenceA (replicate n getChar)
```

# Applicative Laws

## Identity Preservation

- `pure id <*> x = x`
- `pure (g x) = pure g <*> pure x`
- `x <*> pure y = pure (\g -> g y) <*> x`
- `x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z`

# Applicative Laws

## Function Application Preservation

- `pure id <*> x = x`
- `pure (g x) = pure g <*> pure x`
- `x <*> pure y = pure (\g -> g y) <*> x`
- `x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z`

# Applicative Laws

## Order of Evaluation For Pure Operands Doesn't Matter

- `pure id <*> x = x`
- `pure (g x) = pure g <*> pure x`
- `x <*> pure y = pure (\g -> g y) <*> x`
- `x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z`

# Applicative Laws

## Associativity of <*>

- `pure id <*> x = x`
- `pure (g x) = pure g <*> pure x`
- `x <*> pure y = pure (\g -> g y) <*> x`
- `x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z`

# <$>

- If all laws are satisfied, `fmap g x` is equal to `pure g <*> x`.
- There's an infix operator for `fmap`:
  - `g <$> x` for `fmap g x` or `pure g <*> x`.
- Lets you write:
  - $g \texttt{ <\$> } x_1 \texttt{ <*> } \cdots \texttt{ <*> } x_n$.

# Question Time

Questions Anybody?

# For Next Friday

- Study [Hutton 2016, Sec 12.1–12.2].

# Acknowledgements


- Partially based on [Hutton 2016, Chapter 12].
- For pronounciation of <*> and friends, see
  - https://wiki.haskell.org/Pronunciation.

# Bibliography I



Hutton, Graham [2016]. *Programming in Haskell*. Second. Cambridge University Press. ISBN: 978-1-316-62221-1.

# About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.