Functional Programming

WS 2021/22

Torsten Grust
University of Tübingen

Functors, Applicatives, and Monads

We will now discuss the `Functor`, `Applicative`, and `Monad` family of type classes. Each of these type classes (or: algebras) is more powerful than the last.

- `Monad`, in particular, will finally provide an answer to what lies behind the ominous `IO a` type—as in `main :: IO ()`—which somehow allows to cleanly integrate side effects 💀 (*e.g.*, I/O, state, non-determinism, …) into the pure Haskell language.

Type Class Functor

Type class Functor embodies the **application of a function to the elements (or: inside) of a structure, while leaving the structure (or: outside) alone.**

- Examples:
  - `map :: (a -> b) -> [a] -> [b]`
  - `mapTree :: (a -> b) -> Tree a -> Tree b`

- In general:

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

- **NB:** `f` is a type constructor that receives exactly one type argument. (Functor is also called a *constructor class*.)

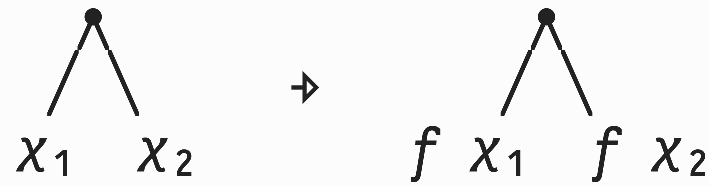
Functors

- **Example:** Instances of constructor class Functor:

```
-- map f [x1,…,xn] = [f x1,…,f xn].  Behaves like a functor. ✓
instance Functor [] where
  fmap = map

----------------------------------------------------------------

instance Functor Tree where
  fmap = mapTree
```



- Again **NB:** Both, [] and Tree, are **type constructors.** Applying them to a type t yields regular types [] t (≡ [t]) and Tree t.

Interlude: Kinds

- We know that Haskell distinguishes **types** and **type constructors**:
    - **Types** have values (*e.g.*, Bool has values True, False).
    - **Type constructors** do not (*e.g.*, no value has “type” Maybe).

- Type constructors build new types from existing types:

```
❶ Maybe Bool            ✔ is a type
❷ Maybe (Maybe Bool)    ✔ is a type
❸ Maybe Maybe           ✗ is nonsense
```

- We find a similar situation on the level of values:

```
❶ True                  ✔ is a value  (True :: Bool)
❷ not True              ✔ is a value  (not :: Bool -> Bool)
❸ not not               ✗ is nonsense
```

Kinds: “Types for Types”

Spot the correspondence:

- **On the value level, we have types** that describe which function applications make sense.
- **On the type level, we have <u>kinds</u>** that describe which type constructor applications make sense.

- Types and type constructors have **kinds** (read `*` as “*any type*”):

Kind	describes...	Examples
`*`	types	`Float`, `Bool`, `[Char]`, `[(Int,Int)]`
`* -> *`	unary type constructors	`Maybe`, `[]`
`* -> * -> *`	binary type constructors	`Either`, `(,)`, `(->)`

```
              * -> *
              ↑    ↑
type argument      resulting type
```

Kinds of Type Classes

- Type classes are also kinded to avoid nonsense constructions:

```
❶ Eq Bool            ✓ is a constraint
❷ Eq (Maybe Bool)    ✓ is a constraint
❸ Eq Maybe           ✗ is nonsense
```

- Kinds for type classes (again: read * as "*any type*"). In GHCi:

```
> :kind Eq
Eq :: * -> Constraint
> :kind Show
Show :: * -> Constraint
> :kind Functor
Functor :: (* -> *) -> Constraint   -- ⊛
```

⊛ Only unary type constructors can be instances of Functor!

Type Constructors of Kind `* -> * -> *`

- Curried notation for kinds like `* -> * -> *` (or: `* -> (* -> *)`) suggests that **type constructors can be partially applied.** Requires prefix notation for type constructors:

```
a -> b ≡ (->) a b
(a, b) ≡ (,) a b
```

- **Examples** (these yield type constructors of kind `* -> *`):

```haskell
type Flagged = (,) Bool       -- ≡ type Flagged a = (Bool, a)
type Indexed = (->) Int       -- ≡ type Indexed a = Int -> a
type MayFail e = Either e     -- MayFail e a: computation may
                              -- yield a or fail with an error e
```

More Functor Instances

- If the kind of `Either` *e* is `* -> *`, we should be able to define an instance of `Functor` for it:

  data Either *e a* = **Left** *e* | **Right** *a*

  ⇒ fmap operates on the second (last) argument *a* of the type constructor

- Works indeed:

```
instance Functor (Either e) where
  fmap _ (Left err) = Left err
  fmap f (Right x)  = Right (f x)
```

- Functors `f` thus need *not* be containers (like `[]`, `Tree`), but can also describe **values** (of type *a*) **in a context `f`**.

More Functor Instances

- Make type constructors Flagged, Indexed instances of Functor:

```
instances Functor Flagged where          -- Flagged a ≡ (Bool, a)
  fmap f (b, x) = (b, f x)


instance Functor Indexed where           -- Indexed a ≡ Int -> a
  fmap f g = f . g
```

  - Check: Do these still fit with our intuition that `fmap :: (a -> b) -> f a -> f b` applies a function to a value in context `f`? ✔

Stacked Functors

Since functors have kind `* -> *` we can build “stacks” of functors f_i applied to some initial type t (of kind $*$):

$$f_n\ (\cdots\ (f_2\ (f_1\ t))\ \cdots\ )$$

- **Example:** stack of depth $n = 4$ with functors (outer to inner) f_4 = [∘], f_3 = (String,∘), f_2 = Maybe ∘, f_1 = [∘] (of Char):

```haskell
-- (Excerpt of) characters in "The Force Awakens"
tfa :: [(String, Maybe String)]
tfa =  [("Rey"  , Nothing),
        ("Han"  , Just "Solo"),
        ("Finn", Nothing),
        ("Kylo", Just "Ren")   ]
```

⇒ Can use n-fold composition of fmap to “reach into” nested structure and apply a function to contained values at depth n.

Functor Laws

- Any Functor is expected to adhere to the two **functor laws:**

  1. `fmap id ≡ id`
  2. `fmap` *f* `. fmap` *g* `≡ fmap (`*f* `.` *g*`)`

- The two laws capture the essence of the functor idea: `fmap` applies a function to values *inside* the structure (container, context), leaving the structure alone.

- ⚠️ Haskell does *not* enforce these laws. Our implementations of `fmap` are *expected* to behave as shown above.

Deriving Functor Instances

- Note that the Functor instance for (Pred i) is generic and could have been **derived automatically.** General “recipe”:
    - To implement `fmap :: (a -> b) -> f a -> f b` for functor `f`:
        1. Apply function to all contained values of type `a` in the structure.
        2. Recurse into substructures of type `f a`.
        3. Leave everything else untouched.

💡 Haskell can derive Functor instances for algebraic datatypes with language extension DeriveFunctor (use GHC compiler pragma `{-# LANGUAGE DeriveFunctor #-}`).

Type Class Applicative

Essence of a functor `f a`: we can use `fmap` to apply a function to the insides of a structure/context. Now, type class `Applicative`:

- **Both**, the function to apply and its argument(s), reside in a structure/context. Two steps to apply (via operator `<*>`, read: "*tie fighter*"):

  1. extract function and argument(s) from their structures,
  2. place result in structure again:

- Compare (**NB:** `fmap` *f e* may also be written as *f* `<$>` *e*):

```
  ($)   ::                        (a -> b) ->   a ->   b
 (<$>)  :: Functor f =>           (a -> b) -> f a -> f b
 (<*>)  :: Applicative f => f (a -> b) -> f a -> f b
```

Type Class Applicative

Haskell's type class Applicative:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

- **Notes:**
  - Every Applicative also is a Functor (f is also referred to as *applicative functor*), *i.e.*, we need fmap.
  - pure *f* places regular function (any value, actually) *f* into a trivial structure/context. Thus:

    ```
    fmap f e = pure f <*> e
    ```

Applicative: Function *and* Argument in Structure/Context

- With Functor and fmap we had **one bit of structure** that we needed to preserve. With Applicative and (<*>) we have **two bits of structure** that we need to combine:
  - Applicative embodies
    1. **function application** on the level of values, *and*
    2. **combination** on the level of structures:

```
                                  =                preserve

fmap ::     (a -> b) -> f a -> f b

(<*>) :: f (a -> b) -> f a -> f b

                                  ⊕                combine
```

Combining Structure/Context

Examples (of structure/context combination ⊕):

```
Prelude> Just (+ 2) <*> Just 40
Just 42
Prelude> Just (+ 2) <*> Nothing
Nothing
Prelude> Nothing <*> Just 40
Nothing
Prelude> Nothing <*> Nothing
Nothing
Prelude> [(+ 1),(* 10)] <*> [1,2,3]
[2,3,4,10,20,30]
Prelude> ([True], (+ 2)) <*> ([False], 40)
([True,False],42)
Prelude> (True, (+ 2)) <*> (False, 40)     ⚡

        how to combine two Bools?
```

Interlude: Type Class `Monoid`

Type class `Monoid a` represents *combinable* values of type `a` with a neutral element:

```
class Semigroup a => Monoid a where
  mempty  :: a                -- neutral element for mappend
  mappend :: a -> a -> a      -- associative combine/⊕, also: (<>)
  mconcat :: [a] -> a         -- x₁ <> x₂ <> ⋯ <> xₙ
```

- **Examples** of monoids (*neutral element*, ⊕) ≡ (`mempty`, `(<>)`):
    - (`0`, `(+)`), (`1`, `(*)`)
    - (`True`, `(&&)`), (`False`, `(||)`)
    - (`empty`, `union`) [see module `Data.Set`: (∅, ∪)]
    - (`[]`, `(++)`)

Sample Applicative Instances

```
instance Applicative Maybe where
  pure x = Just x

  Just f <*> Just x = Just (f x)
  _      <*> _      = Nothing
-----------------------------------------------------------------
instance              Applicative ((,) c) where
  pure x =

  (c1, f) <*> (c2, x) =
-----------------------------------------------------------------
instance Applicative [] where
  pure x =

  fs <*> xs =
```

An Application of Applicative: Input Validation

- The upcoming example uses Applicative operator variant *>:

```
                   =
(*>) :: f a -> f b -> f b          preserve value

                   ⊕               combine structures
```

- Default definition of *> in class Applicative:

```
                const id u = id   id v = v    value of v preserved ✓

u *> v = pure (const id) <*> u <*> v
                      ⊕
                                ⊕
                                        structure of u, v combined ✓
```

combined, but simply yields structure of u since
pure builds a neutral structure: pure id <*> x ≡ x