Haskell tc ucs 0 - Zusammenfassung Programmierung und Modellierung

Haskell Typeclasses Cheat Sheet

Function Synonyms

Functor	Applicative	Monad	List ([])	Fn. ((->) r)
fmap/(<$>)	liftA	liftM	map	(.)
	pure	return	(:[])	const
	(<*>)	ap	(<*>)	(<*>)
		(>>=)	(>>=)	(>>=)

Functor	Applicative	Monad	List ([])	Fn. ((->) r)
	(*>)	(>>)	(>>)	(>>)
		join	concat	join
		(=<<)	concatMap	(=<<)
	liftA2	liftM2	liftA2	liftA2
	liftA3	liftM3	liftA3	liftA3
	sequenceA	sequence	sequence	sequence
	traverse	mapM	traverse	traverse
	for	forM	for	for

Functor

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

Laws

identity	fmap id ≡ id
distributivity	fmap (f . g) ≡ fmap f . fmap g

instances – fmap examples

```
[]          fmap f [x,y,z]       ≡ [f x, f y, f z]
Maybe       fmap f (Just x)      ≡ Just (f x)
IO          fmap length getLine  ≡ 4  - input: "test"
((->) r)    fmap f g             ≡ f . g
(Either a)  fmap f (Left x)      ≡ (Left x)
            fmap f (Right x)     ≡ (Right (f x))
((,) a)     fmap f (x,y)         ≡ (x,f y)
```

Functions

Data.Functor

```
<$>  :: Functor f                f <$> x       ≡ fmap f x
     => (a -> b)                 f <$> Just x  ≡ Just (f x)
     -> f a -> f b               f <$> Left x  ≡ Left x

<$   :: Functor f                (x <$)        ≡ fmap (const x)
     => a -> f b -> f a          x <$ [y,z]    ≡ [x,x]

$>   :: Functor f                ($> x)        ≡ fmap (const x)
     => f a -> b -> f b          [y,z] $> x    ≡ [x,x]

void :: Functor f                void f        ≡ () <$ f
     => f a -> f ()              void [x,y]    ≡ [(),()]
```

Applicative

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

Laws

identity	pure id <*> v ≡ v
composition	pure (.) <*> u <*> v <*> w ≡ u <*> (v <*> w)
homomorphism	pure f <*> pure x ≡ pure (f x)
interchange	u <*> pure y ≡ pure ($ y) <*> u

Instances – pure examples

```
[]          pure x ≡ [x]
Maybe       pure x ≡ Just x
IO          pure x ≡ x "inside" IO
((->) r)    pure x ≡ const x
(Either a)  pure x ≡ Right x
```

Instances – apply (<*>) examples

```
[]                      [f,g] <*> [x,y]  ≡ [f x, g x, f y, g y]
          [(+),(*)] <*> [x,y] <*> [z]    ≡ [x+z,y+z,x*z,y*z]

Maybe              Just f <*> Just x     ≡ Just (f x)
                   Just f <*> Nothing    ≡ Nothing

IO          pure (++) <*> getLine <*> getLine

((->) r)            (f <*> g) x          ≡ f x (g x)
              (f <*> g <*> h) x          ≡ f x (g x) (h x)

(Either a)   Right f <*> Right x         ≡ Right (f x)
             Left e1 <*> Right x         ≡ Left e1
             Right f <*> Left e2         ≡ Left e2
             Left e1 <*> Left e2         ≡ Left e1
```

Functions

```
(*>)    :: Applicative f                    (*>)  ≡ flip (<*)
        => f a -> f b -> f b        [x] *> [y,z]  ≡ [y,z]

(<*)    :: Applicative f                    (<*)  ≡ liftA2 const
        => f a -> f b -> f a        [x] <* [y,z]  ≡ [x,x]

liftA   :: Applicative f               liftA f x  ≡ fmap f x
        => (a -> b)                liftA f [x,y]  ≡ [f x,f y]
        -> f a -> f b            liftA f Nothing  ≡ Nothing

liftA2  :: Applicative f      liftA2 (+) (Just x) (Just y)
        => (a -> b -> c)                  ≡
        -> f a -> f b -> f c          Just (x + y)

liftA3  :: Applicative f      liftA3 f [x,y] [z] [w,u]
        => (a -> b -> c -> d)             ≡
        -> f a -> f b             [ f x z w, f x z u
        -> f c -> f d             , f y z w, f y z u ]
```

Functor and Applicative common idioms

```
f <$> x <*> y  ≡ pure f <*> x <*> y ≡ fmap f x <*> y
f <$ g <*> x   ≡ pure f <* g <*> x
x <*> y $> z   ≡ x <*> y *> pure z
```

Monad

```haskell
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Laws

left identity	return x >>= f ≡ f x
right identity	m >>= return ≡ m
associativity	(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)

Instances – return examples

```
[]          return x ≡ [x]
Maybe       return x ≡ Just x
IO          return x ≡ x "inside" IO
((->) r)    return x ≡ const x
(Either a)  return x ≡ Right x
```

Instances – bind (>>=) examples

```
[]                       xs >>= f  ≡ concatMap f xs
          [x,y] >>= replicate 3    ≡ [x,x,x,y,y,y]

Maybe                Just x >>= f  ≡ f x
         Just [x] >>= listToMaybe  ≡ Just x

IO               getLine >>= putStrLn

((->) r)              (f >>= g) x  ≡ g (f x) x
          (tail >>= (++)) [x,y]    ≡ [y,x,y]

(Either a)          Right x >>= f  ≡ f x
                    Left e1 >>= f  ≡ Left e1
```

Instances – then (>>) examples

```
[]           [x,y] >> [z,w,u]    ≡ [z,w,u,z,w,u]

Maybe        Nothing >> Just x   ≡ Nothing
              Just x >> Just y   ≡ Just y

IO           putStrLn "Username:" >> getLine

((->) r)          (f >> g) x     ≡ g x

(Either a)   Right x >> Right y  ≡ Right y
             Left e1 >> Right y  ≡ Left e1
             Right x >> Left e2  ≡ Left e2
             Left e1 >> Left e2  ≡ Left e1
```

Do notation

do notation	≈	desugarized do notation
do patA <- action1		action1 >>= \patA ->
action2		action2 >>
patB <- action3		action3 >>= \patB ->
action4		action4

Functions

```
mapM :: Monad m                 mapM f ≡ sequence . map f
     => (a -> m b)
     -> [a] -> m [b]

sequence :: Monad m            sequence [Just x, Just y]
         => [m a]                          ≡
         -> m [a]                    Just [x,y]

(>=>) :: Monad m          (f >=> g) x ≡ f x >>= g
      => (a -> m b)
      -> (b -> m c)         iterate (+1) >=> replicate 2 $ 0
      -> a -> m c            ≡ [0,0,1,1,2,2,3,3,4,4,5,5...]

foldM :: Monad m          foldM f a [x, y, z]
      => (a -> b -> m a)             ≡ do a1 <- f a x
      -> a -> [b]                         a2 <- f a1 y
      -> m a                              f a2 z

replicateM :: Monad m             replicateM 2 [x,y]
           => Int -> m a                   ≡
           -> m [a]              [[x,x],[x,y],[y,x],[y,y]]

when, unless :: Monad m     when verbose (putStrLn "msg")
             => Bool
             -> m ()        unless quiet (putStrLn "msg")
             -> m ()

forever :: Monad m          forever (getLine >>= putStrLn)
        -> m a -> m b               (← poor man's cat)

<fn>_ :: Monad m            sequence_ ≡ void sequence
      => ... -> m ()            mapM_ ≡ void mapM
```

Monoid

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a     -- NOTE: (<>) infix synonym
```

Laws

identity	mempty <> x ≡ x ≡ x <> mempty
associativity	x <> (y <> z) ≡ (x <> y) <> z

Instances – `mempty` and `mappend` (<>) examples

```
Ordering   mempty ≡ EQ         EQ <> GT ≡ GT ≡ GT <> LT
()         mempty ≡ ()         () <> () ≡ ()
[a]        mempty ≡ []         xs <> ys ≡ xs ++ ys
Maybe a    mempty ≡ Nothing    Just m <> Just n
                                     ≡ Just (m <> n)
```

Functions

```
(<>) :: Monoid a => a -> a -> a         (<>) ≡ mappend

mconcat :: Monoid a          mconcat ≡ foldr (<>) mempty
        => [a] -> a          mconcat [[x],[y]] ≡ [x,y]
```

Foldable

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

NOTE: Foldable instances can also be defined by `foldMap`

Instances – `foldr` examples

```
[]                  foldr f z [x,y] ≡ x `f` (y `f` z)
Maybe          foldr f x (Just y) ≡ x `f` y
(Either a)     foldr f x (Left y) ≡ x
((,) a)             foldr f x (y,z) ≡ x `f` z
```

Functions

```
foldMap :: ( Foldable t          foldMap f ≡ fold . map f
           , Monoid m )    foldMap f [x,y] ≡ f x <> f y
        => (a -> m)         foldMap f (Just x) ≡ f x
        -> t a -> m        foldMap Nothing ≡ mempty

fold :: ( Foldable t                   fold ≡ foldMap id
        , Monoid m )          fold [x,y,z] ≡ x <> (y <> z)
     -> t m -> m              fold Nothing ≡ mempty

foldl :: (Foldable t)    foldl f z [x,y] ≡ (x `f` y) `f` z
      => (b -> a -> b)
      -> b -> t a -> b

toList :: (Foldable t) => t a -> [a]    toList (x,y) ≡ [y]
null :: (Foldable t) => t a -> Bool      null Nothing ≡ True
length :: (Foldable t) => t a -> Int    length [x,y] ≡ 2
concat :: Foldable t => t [a] -> [a]          concat ≡ fold

elem,notElem :: (Foldable t, Eq a) => a -> t a -> Bool
maximum, minimum :: (Foldable t, Ord a) => t a -> a
sum,     product :: (Foldable t, Num a) => t a -> a
asum :: (Foldable t, Alternative f) => t (f a) -> f a
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
and, or   :: Foldable t => t Bool -> Bool
any, all  :: Foldable t => (a -> Bool) -> t a -> Bool
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

Operators (grouped by precedence)

6 (++): mappend	<>	
4 (&&): fmap, repl. value	<$>, <$, $>	
apply, apply discard	<*>, *>, <*	
3 (||): choice	<|>	
1 ($): bind, then	(>>=), (>>)	
rv. bind, kleisli c.		(=<<), (>=>), (<=<)

NOTE: left and right alignments indicate left- and right- associativity

Traversable

```
class (Functor t, Foldable t) => Traversable t
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
```

NOTE: It is only necessary to define either `traverse` or `sequenceA`.

Equivalences (laws are ommited here)

```
traverse f ≡ sequenceA . fmap f
 sequenceA ≡ traverse id
```

Instances – `traverse` examples

```
[]              traverse f [x,y] ≡
                  (:) <$> f x <*> ((:) <$> f y <*> [])
Maybe         traverse f (Just x) ≡ Just <$> f x
(Either a)   traverse f (Right x) ≡ Right <$> f x
             traverse f (Left e1) ≡ pure (Left e1)
((,) a)           traverse f (x,y) ≡ (,) x <$> f y
```

Instances – `sequenceA` examples

```
[]        sequenceA [x,y] ≡ (:) <$> x <*> ((:) <$> y <*> [])
Maybe     sequenceA (Just x)  ≡  Just <*> x
Either    sequenceA (Right x) ≡ Right <*> x
((,) a)      sequenceA (x,y) ≡ (,) x <$> y
```

Functions

```
for  :: (Traversable t, Applicative f)   flip for ≡ traverse
     => t a -> (a -> f b) -> f (t b)

mapM :: (Traversable t, Monad m)
     => (a -> m b) -> t a -> m (t b)

sequence :: (Traversable t, Monad m)
         => t (m a) -> m (t a)
```

Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Laws

identity	mempty <|> x ≡ x ≡ x <|> mempty
associativity	x <|> (y <|> z) ≡ (x <|> y) <|> z

Instances – `empty` and choice (<|>) examples

```
[a]        empty ≡ []                  xs <|> ys ≡ xs ++ ys
Maybe a    empty ≡ Nothing     Just x <|> Just y ≡ Just x
                              Nothing <|> Just y ≡ Just y
```