

Functional Programming

WS 2021/22

Torsten Grust
University of Tübingen

Haskell Ramp-Up (Part 2)

- Haskell's system of types is extensible. Users may
 - introduce synonyms for existing types (using keyword `type`)
 - or
 - define entirely new types (using keywords `newtype` and `data`).
- We are focusing on `data` and **algebraic data types** now.

Algebraic Data Types (Sum-of-Product Types)

- Recall: `[]` and `(:)` are the value constructors for type `[a]`.
- Can define entirely new type `T` and its constructors `Ki`:

```
data T a1 a2 ... an = K1 b11 ... b1(n1)
                        | K2 b21 ... b2(n2)
                        |
                        | Kr br1 ... br(nr)
```

- Defines **type constructor** `T` and `r` **value constructors** `Ki` ($1 \leq i \leq r$) with types

$$K_i :: b_{i1} \rightarrow \dots \rightarrow b_{i(n_i)} \rightarrow T\ a_1\ a_2\ \dots\ a_n$$

`Ki`: identifier with uppercase first letter or symbol starting with a colon `(:)`.

Algebraic Data Types can be Sum Types

- Example (**sum type**, or: enumeration, choice):
no value constructor has any argument (all $n_i = 0$).

File: `weekday.hs`

```
-- A sum type (enumeration)

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

-- Is this day on a weekend?
weekend :: Weekday -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False

main :: IO ()
main = print (weekend Thu, weekend Sat)
```

Algebraic Data Types (**deriving**)

- Add `deriving (c, c, ...)` to `data` declaration to define canonical operations for the new data type:

<code>c</code> (class)	operations
<code>Eq</code>	equality (<code>==</code> , <code>/=</code>)
<code>Show</code>	printing (<code>show</code>)
<code>Ord</code>	ordering (<code><</code> , <code><=</code> , <code>max</code> , ...)
<code>Enum</code>	enumeration (<code>[x..y]</code> , ...)
<code>Bounded</code>	bounds (<code>minBound</code> , <code>maxBound</code>)

Algebraic Data Types can be Product Types

- Example (**product type**):

$r = 1$ (single constructor), with $n_1 = 2$ (pair).

File: `sequence.hs`

```
-- A product type (single constructor)

data Sequence a = S Int [a]
  deriving (Eq, Show)

fromList :: [a] -> Sequence a
fromList xs = S (length xs) xs

(+++) :: Sequence a -> Sequence a -> Sequence a
S lx xs +++ S ly ys = S (lx + ly) (xs ++ ys)

len :: Sequence a -> Int
len (S l _) = l

main :: IO ()
main = print $ len (fromList ['a'..'m'] +++ fromList ['n'..'z'])
```

Algebraic Data Types in General: Sum of Product Types

- Examples (**sum-of-product types**):

```
data Maybe a = Nothing           -- "optional a"
               | Just a

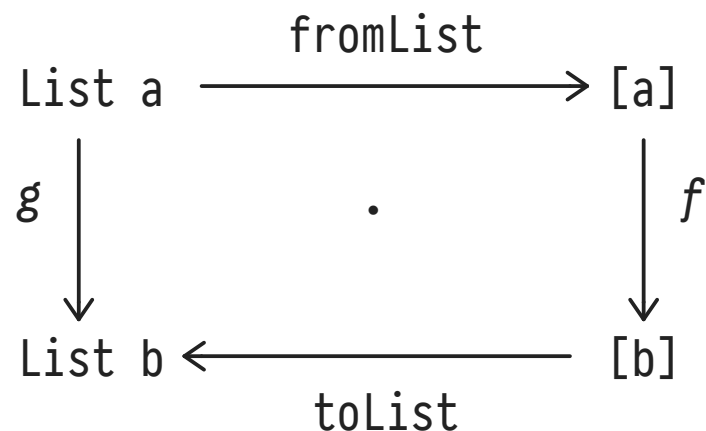
data Either a b = Left a         -- "either a or b"
                  | Right b

data List a = Nil               -- "list of a" (recursive A)
               | Cons a (List a)
```

Types `[a]` and `List a` are Isomorphic

- The built-in list type `[a]` is not special.

Our own sum-of-product type `List a` has the same structure and can fully replace `[a]`:



$$g \equiv \text{toList} \cdot f \cdot \text{fromList}$$

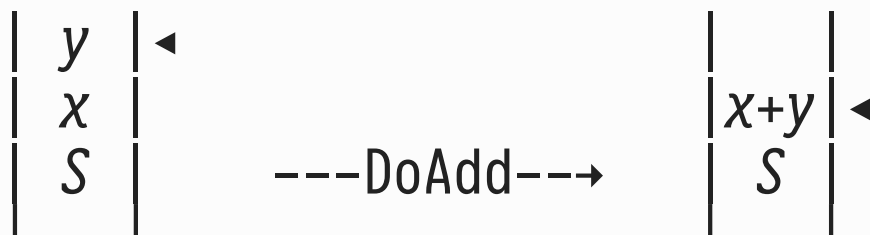
A Super-Simple Stack Machine

- Operations **Push** and **DoAdd** act on the machine's stack (let S denote some arbitrary stack contents):

1



2



- Once all operations have been processed, the top of the stack (denoted \triangleleft) holds the answer of the machine.

Type Classes

- Haskell's type system implements **type classes**, the instances of which implement a common set of operations, each in a type-specific fashion.

Type classes allow for **ad-hoc polymorphism** (or **overloading**).

- **Example:** We want to express equality for all sorts of types (`Int`, `String`, `(a,b)`, `[a]`, `Exp a`):
 1. Want to continue to use the single symbol `==` (not `eqInt`, `eqString`, ...).
 2. Obviously, the type-specific implementations of `==` need to differ.
 3. Some types may not be able to implement `==` at all (consider `a -> b`).

Type Classes

- A **type class** C defines a family of n functions f_i (“methods”) which all **instances** of C must implement:

```
class C a where                                -- class name C: Uppercase
  f1 :: t1
    ⋮
  fn :: tn
```

- Read: “If type a is an instance of C , then all methods f_i are implemented for a .”
- The types t_i must mention type a .
- For any f_i , the class may provide a **default** definition (that instances may overwrite).

Type Classes

- **Example** (type class `Eq` defines what it means for type `a` to support equality comparisons):

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x /= y = not (x == y)      -- default definitions
  x == y = not (x /= y)
```

- If type `a` wants to support equality (be a member of class `Eq`), defining either `==` or `/=` suffices.

Type Classes: Class Constraints

- A class constraint

class constraint

$$\begin{array}{l} \overbrace{e :: \mathcal{C} \ a \Rightarrow t} \\ e = \dots \end{array}$$

(where t mentions a) says that expression e has type t *only if* type a is an instance of class \mathcal{C} .

\Rightarrow In the definition of e we may use the methods of class \mathcal{C} on values of type a .

Type Classes: Class Inheritance

- Defining

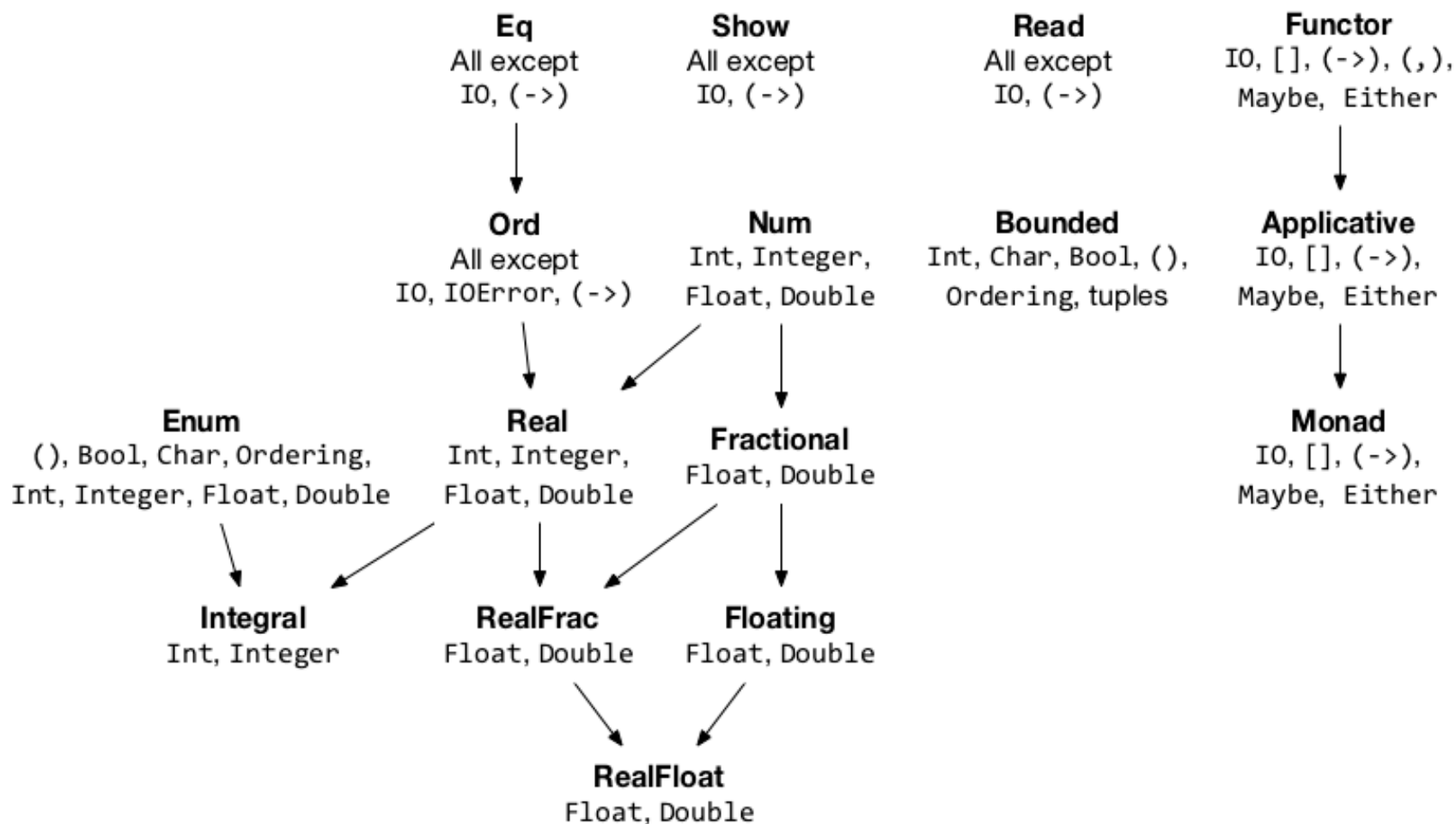
```
class ( $C_1\ a, C_2\ a, \dots$ )  $\Rightarrow C\ a$  where ...
```

makes type class C a **subclass** of the classes C_i . C inherits all C_i methods.

- The class constraint $C\ a \Rightarrow t$ thus implies the larger constraint $(C_1\ a, C_2\ a, \dots, C\ a) \Rightarrow t$:

Writing the type $f :: \text{Ord } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$ abbreviates $f :: (\text{Eq } a, \text{Ord } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}$ and function f may, e.g., use \leq as well as $=$ on value of type a .

Type Classes: Class Inheritance



Inheritance of standard Haskell type classes ($\rightarrow \equiv$ superclass of)

Type Classes: Class Instances

- Now: Define type-specific behavior for the class methods (\Rightarrow **overloading**).
- Implementing all methods of class C makes t an **instance** of C :

```
instance C t where
  f1 = <def of f1>      -- all fi may be provided, minimal
  ⋮                      -- complete definition must be provided
  fn = <def of fn>      -- types must match definition of C
```

- Class constraint $C\ t$ is satisfied from now on.

- **Example:**

```
instance Eq Bool where
  x == y = (x && y) || (not x && not y)
```



Type Classes: Class Instances

- An instance definition for **type constructor** t may formulate type constraints for its argument types a , b , ...:

```
instance ( $C_1\ a$ ,  $C_2\ a$ ,  $C_3\ b$ , ...) =>  $C\ (t\ a\ b\ \dots)$  where  
...
```

- **Example:**

```
-- print sequences as «3|[10,20,30]»  
instance (Show a) => Show (Sequence a) where  
  show (Sequence l xs) = "«" ++ show l ++ "|" ++ show xs ++ "»"
```

-  This makes use of two other **Show** instances:
 1. `instance Show Int`
 2. `instance (Show a) => Show [a]`

Type Classes: Deriving Class Instances

- Automatically make user-defined data types (`data ...`) instances of classes $C_i \in \{\text{Eq}, \text{Ord}, \text{Enum}, \text{Bounded}, \text{Show}, \text{Read}\}$:

```
data T a1 a2 ... an = ...      -- } regular algebraic
                        | ...      -- } data type definition
deriving (C1, C2, ...)
```

<i>C</i>	Semantics of derived instance
Eq	for all sum-of-prod types, equality of constructors, recursive equality of components
Ord	for all sum-of-prod types, lexicographic ordering of constructors in <code>data</code> definition
Enum	only for sum types, <i>n</i> th constructor mapped to <i>n</i> -1
Bounded	only for sum types, <code>minBound</code> / <code>maxBound</code> \equiv first/last constructor
Show	<code>show</code> generates syntactically correct Haskell presentation
Read	<code>read</code> reads string generated by <code>Show</code> instance