Functional Programming

SS 2025

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].
- We can define an entirely new type T and its constructors K_i:

$$
\begin{aligned}
\textbf{data}\ T\ a_1\ a_2\ \ldots\ a_n = {} & K_1\ b_{11}\ \ldots\ b_{1(n1)} \\
\mid {} & K_2\ b_{21}\ \ldots\ b_{2(n2)} \\
\vdots {} & \\
\mid {} & K_r\ b_{r1}\ \ldots\ b_{r(nr)}
\end{aligned}
$$

  - Defines **type constructor** T and r **value constructors** K_i ($1 \leqslant i \leqslant r$) with types

$$K_i :: b_{i1} \to \cdots \to b_{i(ni)} \to T\ a_1\ a_2\ \ldots\ a_n$$

K_i: 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:

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

Algebraic Data Types can be Product Types

- Example (**product type**):
  r = 1 (single constructor), with n_1 = 2 (pair).

File: sequence.hs

```haskell
-- 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**):

```haskell
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 ⚠)
            | 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]:

$$\begin{tikzcd} \texttt{List a} \arrow[r, "\texttt{fromList}"] \arrow[d, "g"'] \arrow[dr, phantom, "."] & {\texttt{[a]}} \arrow[d, "f"] \\ \texttt{List b} & {\texttt{[b]}} \arrow[l, "\texttt{toList}"] \end{tikzcd}$$

$g \equiv \texttt{toList . } f \texttt{ . fromList}$

A Super-Simple Stack Machine

- Operations Push and DoAdd act on the machine's stack (S denotes arbitrary stack contents, ◂ is the top of the stack):

❶

```
|   |           |  x  |◂
| S |◂ --Push x--→ |  S  |
└───┘           └─────┘
```

❷

```
| y |◂          |     |
| x |           | x+y |◂
| S | ---DoAdd--→ |  S  |
└───┘           └─────┘
```

- Once all operations have been processed, the top of the stack 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 (*i.e.*, be a member of class Eq), defining either == or /= suffices.

Type Classes: Class Constraints

- A **class constraint**

$$\overbrace{\mathtt{}\ }^{\textbf{class constraint}}$$

```
e :: C a => t
e = …
```

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

⇒ In the definition of *e* (here: ⋯) we may use the methods of class *C* on values of type *a*.

Type Classes: Class Inheritance

- Defining

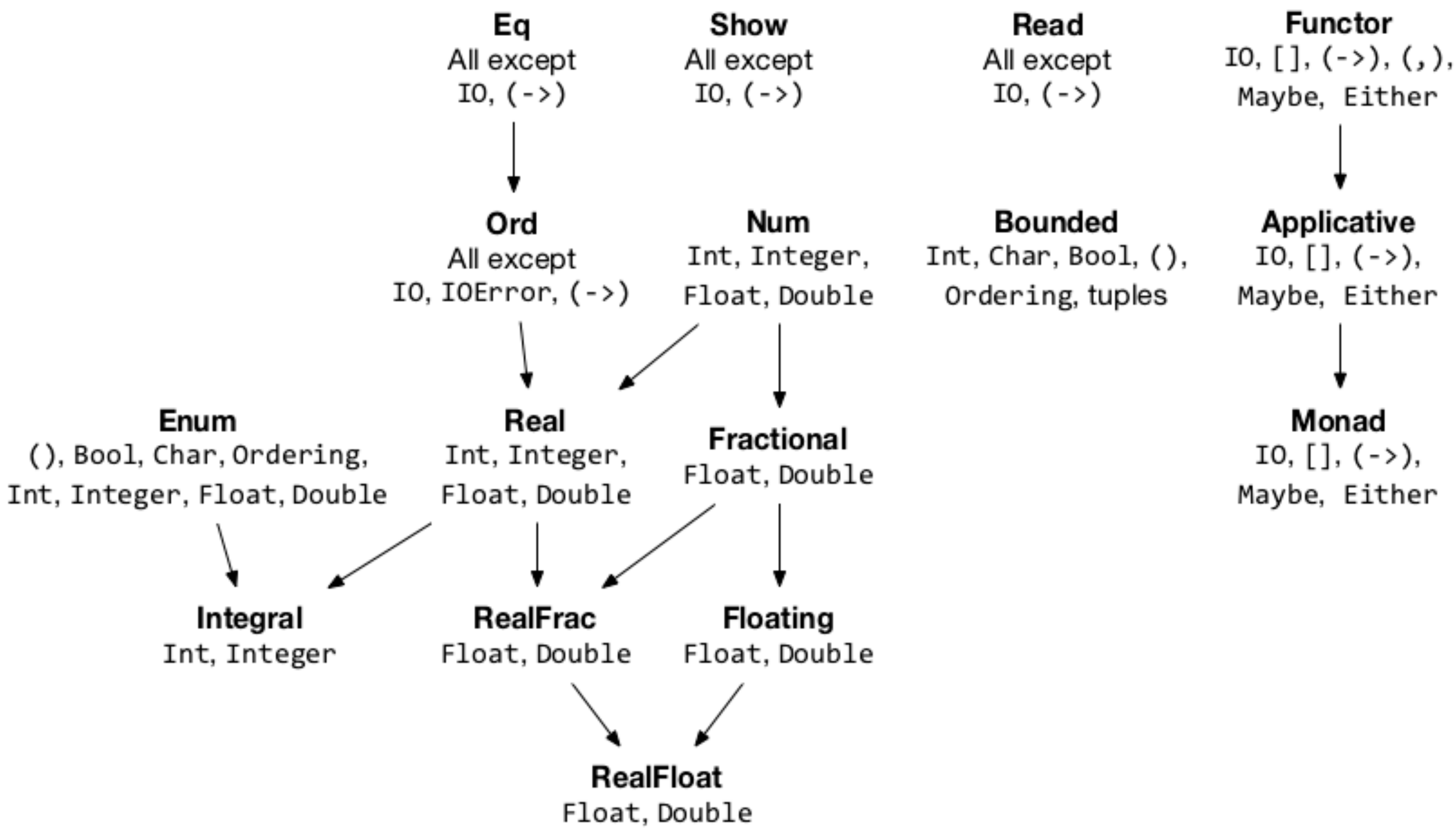
```
class (C₁ a, C₂ a, …) => C a where …
```

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 $(C_1\ a, C_2\ a, \ldots, C\ a) \Rightarrow t$:

    Writing the type `f :: Ord a => a -> a -> Bool` abbreviates `f :: (Eq a, Ord a) => a -> a -> Bool` and function `f` may, e.g., use `<=` as well as `==` on values of type `a`.

Type Classes: Class Inheritance



Inheritance of standard Haskell type classes (→ ≡ superclass of)

Type Classes: Class Instances

- Now: Define type-specific behavior for the class methods (⇒ **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 (C₁ a, C₂ a, C₃ b, …) => C (t a b …) 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 $C_i \in$ {Eq, Ord, Enum, Bounded, Show, Read}:

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

C	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 data definition
Enum	only for sum types, nth constructor mapped to n-1
Bounded	only for sum types, minBound/maxBound ≡ first/last constructor
Show	show generates syntactically correct Haskell presentation
Read	read reads string generated by Show instance