1.1.4 Types

Mittwoch, 20. April 2016 11:00

Type = set of "similar" objects

e.g., Int, Bool, [Int], [[Int]], Int -> [Int], (Int, [Int], Int -> Int), ...

General idea: one uses a <u>type constructor</u> to create new types from already existing types

<u>Slide 13</u>

- (<u>tyconstr</u> <u>type</u>$_1$ ... <u>type</u>$_n$), $n \geq 0$

  ↑ type constructor, e.g. Int, Bool, Char, ... are pre-defined type constructors of arity 0

  More type constructors can be defined by the user (see later).
  Moreover, [...], ->, (...) are pre-defined type constructors with special syntax.

- [<u>type</u>] : [..] is a type constructor of arity 1

  stands for the type of lists where all elements have the same <u>type</u>

  e.g. [Int], [[Int]], ...

- (<u>type</u>$_1$ -> <u>type</u>$_2$) : -> is a type constructor of arity 2

  stands for the type of functions from <u>type</u>$_1$ to <u>type</u>$_2$

e.g. [Int] -> Bool, Int -> (Int -> Int), ...

- (type$_1$, ..., type$_n$), $n \geq 0$: (...) is a tuple constructor of arbitrary arity

  () is a type with only one object: ()

  (type) is the same as type:

  (5) — 5 is of type Int

  type Int = (Int)

  e.g.:
  (Int, [Int], Int -> Int) is a type

  (5, [1,2,3], square) is an expression of this type

- Var: a type variable is also a type.
  Needed for parametric polymorphism.

Parametric polymorphism

Polymorphism: one can apply the same function to arguments of different types

- parametric polymorphism (functional languages): the same implementation of a function is used for arguments of different

types

- ad hoc polymorphism: function has several different implementations (with the same function name). Type of the arguments determines which implementation is executed.

object-oriented languages

Nowadays, many languages have both forms of polymorphism (e.g., Haskell, Java since Java 5, ...)

First: parametric polymorphism (Slide 14)

```
id :: a -> a
id x = x
```

Type variable, can be instantiated by any type.

All occurrences of the same type variable in a type have to be instantiated in the same way.

Type of append (++):

```
(++) :: [a] -> [a] -> [a]
```

[1,2] ++ [3] = [1,2,3]

[True] ++ [False] = [True, False]

Now a function of type $\underline{type_1} \to \underline{type_2}$ can be applied to an argument of $\underline{type}$ if there is a most general substitution σ of type variables such that $\sigma(\underline{type_1}) = \sigma(\underline{type})$.

Result has type: $\sigma(\underline{type_2})$.

most general unifier of $\underline{type_1}$ and $\underline{type}$

Ex: [True] ++ [ ]

	[True]	[ ]	
actual types	[Bool]	[b]	
expected types	[a]	[a]	Result [a]

We can unify these types: $\sigma(a) = Bool$ $\sigma(b) = Bool$

Result: $\sigma([a]) = [Bool]$

[ ] ++ [True]

	[ ]	[True]
actual types	[b]	[Bool]
expected types	[a]	[a]

Same solution: $\sigma(a) = \sigma(b) = Bool$

Type definitions: introducing new types

Goal: define new type constructors

The declaration of new types is only possible on top-level (i.e., not in local declarations)

⤳ Slide 15: distinguish between topdecl and decl

↑ only on top level,

i.e., a program is a sequence of top declarations

Type synonyms (keyword "type")

```
type Position = (Float, Float)
type String = [Char]
type Pair a b = (a, b)
```

↑ introduces a new type constructor of arity 2

Haskell makes no difference between
Position, (Float, Float), or Pair Float Float

One can also introduce completely new types by the keyword data.
Here, the programmer provides a context-free grammar (in EBNF-like notation) to define a new type. (Slide 16)

```
data Color = Red | Yellow | Green
```

new type constructor of arity 0, i.e., Color is a type

data constructors of the type Color

```
Red :: Color
```

```
data MyBool = MyTrue | MyFalse
```

Now one can define algorithms with these new types and

use pattern matching, as for pre-defined types (Slide 16).

To print objects on the screen, Haskell uses a function "show" to convert them to strings. The function "show" is pre-defined for many types (Int, Bool, [Int], ...), but not for user-defined types.

To implement "show" for user-defined types:

- do it manually (needs "type classes", next lecture)
- generate a default implementation of "show" automatically: add "deriving Show" at the end of the data-declaration

---

We now show how to use "data" declarations for types with infinitely many objects (Slide 17):

```
data Nats = Zero | Succ Nats
```

Data constructors: Zero :: Nats
Succ :: Nats -> Nats

Succ can be used to construct arbitrary objects: its result type Nats is also among its argument types

Zero $\hat{=}$ 0
Succ Zero $\hat{=}$ 1
Succ (Succ Zero) $\hat{=}$ 2

One can also introduce new type constructors of arity > 0 (i.e., introduce new polymorphic types)

Ex: introduce data type for lists (Slide 18).

```
data List a = Nil | Cons a (List a)
```

↑ new type constr. of arity 1

The newly introduced type is List a

```
Nil :: List a
Cons :: a -> List a -> List a
```

```
len :: List a -> Nats
len Nil = Zero
len (Cons x xs) = Succ (len xs)
```

In a data-declaration

data tyconstr $var_1 \ldots var_n$ = ...

the right-hand side may not contain any type variables except $var_1, \ldots, var_n$.

Type Classes

- Type class = set of types
- Elements of a type class are called instances
- functions in these instances may have the same name, but different implementations
  (ad-hoc polymorphism, overloading)

```
(==), (/=) :: a -> a -> Bool
```

2 == 3 is False

[1,2] == [1,2] is True

True == False is False

But: == should not be applied to functions
(e.g., for functions on numbers or lists, equality of functions is undecidable)

Solution: We introduce a type class Eq that contains all types whose elements can be checked for equality
(e.g., Eq contains Int, Bool, [Int], ...
but not Int -> Int etc.)

(==), (/=) :: Eq a => a -> a -> Bool

Context (Eq a): restricts the instantiation of the type var. a to types from the type class Eq

Slide 19:

class tyconstr var where {cdecl₁ ; ... ; cdeclₙ}

- tyconstr: name of the new type class
- var: type variable stands for the types of the new type class
- cdecl₁ ; ... ; cdeclₙ: declarations which are used in all types of the new type class

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

} default declarations that

$(==), (/=) :: a \to a \to Bool$

x /= y = not $(x == y)$

} default declarations that can be overwritten by declarations in each instance type of the class Eq.

instance tyconstr instype where { idecl$_1$; ...; idecl$_n$ }

means that the type instype is an instance / a member of the type tyconstr

declarations for this particular type instype which overwrite the default implementations of the class tyconstr

```
instance Eq Int where
  (==) = primEqInt
```

primEqInt: pre-defined equality on integers

To evaluate

2 == 3 : one uses primEqInt

2 /= 3 is not (2 == 3) (use default implementation of Eq)

Now we can define type classes and declare that certain types are members / instances of certain typeclasses.

⇒ We can now restrict any type by a context (Slide 20):

Context: (tyconstr$_1$ var$_1$, ..., tyconstr$_n$ var$_n$)

means that each type variable var$_i$ may only be instantiated

by a type from type class $tyconstr_i$.

Contexts cannot only occur in type declarations, but also in instance- and class-declarations.

Contexts in instance declarations:

(context means that a must be a type from the class Eq)

```
instance Eq a => Eq [a] where
   [] == []         = True
   (x:xs) == (y:ys) = x == y && xs == ys
   _  ==  _         = False
```

(x == y: calls (==) for elements of type a)

(xs == ys: calls (==) on [a], i.e., this is a recursive call)

We have several implementations of (==). It depends on the types of the arguments which implementation is executed (ad-hoc polymorphism).

```
instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (x',y') = x == x' && y == y'
```

Haskell has several pre-defined type classes (like Eq). E.g., there is a type class Show that contains all types that can be converted to Strings and shown on the screen.

```
class Show a where
```

```
show :: a -> String
  ⋮
```

To declare that a certain type is a member of the typeclass Show, up to now we added "deriving Show" to the corresponding data declaration.

↗ adds a corresponding instance declaration and default implementation for the functions of the class.
Similarly "deriving Eq" etc.

But we could also define "show" ourselves:

```
data List a = Nil | Cons a (List a)

instance Show a => Show (List a) where

    show Nil = "[ ]"
    show (Cons x xs) = show x ++ " : " ++ show xs
```

Contexts in Class Declarations:

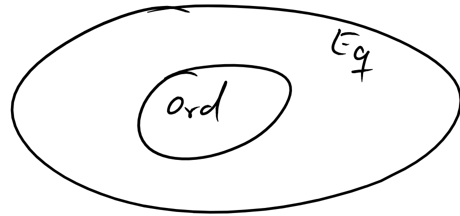
needed for a hierarchical organisation of typeclasses

Ex: Type class Ord for types whose elements can be ordered, i.e., here we have functions like

$>$, $<$, $>=$, $<=$

If a type is a member of Ord, then it should also be a member of Eq:

If a type is a member of Ord, then it should also be a member of Eq:



⇒ Ord should be a subclass of Eq

Context ensures that a is from the class Eq, i.e., Ord ⊆ Eq

```
class Eq a => Ord a where
  (>), (<), (>=), (<=) :: a -> a -> Bool
  x < y = x <= y && x /= y
```

Type classes are also useful for overloading arithmetic operators like +, −, *, ...

+ can be used for Int, Float, ...
2 —— " ————

2 + 3.5

type class for numbers, contains Int, Float, ...

```
class (Eq a, Show a) => (Num a) where
  ⋮
```

Type of 2: Num a => a

i.e.: 2 has any type a where a is a member of the typeclass Num