Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Lecture A4: Classes and Their Instances

Doaitse Swierstra

Utrecht University

August 19, 2010

Overloading versus Polymorphism

Many functions can be applied to values of different types.

Overloading versus Polymorphism

Many functions can be applied to values of different types. There are two underlying principles:

- A polymorphic function works on some specific data structure, without making explicit use of properties of the elements contained. Examples are polymorphic functions such as: *length*, *concat* en *map*. Also $\circ$ is such a function; it can compose two functions, without knowing the types of the argument, the result, or the intermediate value.

Overloading versus Polymorphism

Many functions can be applied to values of different types. There are two underlying principles:

- A polymorphic function works on some specific data structure, without making explicit use of properties of the elements contained. Examples are polymorphic functions such as: *length*, *concat* en *map*. Also $\circ$ is such a function; it can compose two functions, without knowing the types of the argument, the result, or the intermediate value.
- An overloaded function can deal with a collection of different types; actually these are different functions, all with the same name. Examples are $+$, for which one version adds *Int*'s and another *Float*'s. The operator $\leqslant$ has versions for *Int*, *Float*, *Char*, tuples and lists.

Classes

A class definition specifies a set of overloaded function names with their types. Instances of a class define the specific function definitions. The types *Int* and *Float* are members of the class *Num*.

Classes

A class definition specifies a set of overloaded function names with their types. Instances of a class define the specific function definitions. The types *Int* and *Float* are members of the class *Num*.

The operator $(+)$ is defined for each type that is an instance of the class *Num*.

Classes

A class definition specifies a set of overloaded function names with their types. Instances of a class define the specific function definitions. The types *Int* and *Float* are members of the class *Num*.

The operator $(+)$ is defined for each type that is an instance of the class *Num*.This fact is expressed by the type of the operator $+$:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

Classes

A class definition specifies a set of overloaded function names with their types. Instances of a class define the specific function definitions. The types *Int* and *Float* are members of the class *Num*.

The operator $(+)$ is defined for each type that is an instance of the class *Num*.This fact is expressed by the type of the operator $+$:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

One can read this as: "$+$ has type $a \rightarrow a \rightarrow a$ provided there is proof $Num\ a$ that a in an instance of *Num*".

Classes

A class definition specifies a set of overloaded function names with their types. Instances of a class define the specific function definitions. The types Int and $Float$ are members of the class Num.

The operator $(+)$ is defined for each type that is an instance of the class Num.This fact is expressed by the type of the operator $+$:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

One can read this as: "$+$ has type $a \rightarrow a \rightarrow a$ provided there is proof $Num\ a$ that a in an instance of Num".

One can think of $Num\ a \Rightarrow$ as an extra implicit argument, containing the actual definition of the specific $(+)$.

... other classes ...

Other classes which occur quite frequent are Eq (types for which there is equality) and Ord (types which can be compared) Operators specified by these classes are e.g.:

$$(\equiv) :: Eq \quad a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(\leqslant) :: Ord \; a \Rightarrow a \rightarrow a \rightarrow Bool$$

Class and Instance declarations

- Besides the standard classes (such as *Num*, *Eq* and *Ord*) you can define your own classes
- This is a very powerful technique to parametrize your code without too much effort
- Many Haskell libraries depend extensively on these concepts

The first version of the class *Num*

A simple variant of the class *Num* from the *Prelude* is:

```
class Num a where
  (+), (−), (∗), (/) :: a→a→a
  negate             :: a→a
```

class Declarations

A class declaration consists of the following elements:

- the keyword **class**;
- the name of the class (Num in our case);
- a type-variable (a in our case);
- the keyword **where**;
- type declarations for operators en functions, in which the type parameter of the class may occur. The functions can be polymorphic.

The instance *Num Int*

In an instance declaration we provide definitions for the functions and operators of the class for a specific type:

The instance *Num Int*

In an instance declaration we provide definitions for the functions and operators of the class for a specific type:

```
instance Num Int where
  (+)      = primPlusInt
  (-)      = primMinusInt
  (*)      = primMulInt
  (/)      = primDivInt
  negate   = primNegInt
```

Instance Declarations

An instance declaration contains the following parts:

- the keyword **instance**;
- the name of a declared class (Num);
- the type for which we are declaring an instance (Int);
- the keyword **where**;
- definitions for the functions and operators of the class for this specific type

Instance Declarations

An instance declaration contains the following parts:

- the keyword **instance**;
- the name of a declared class (Num);
- the type for which we are declaring an instance (Int);
- the keyword **where**;
- definitions for the functions and operators of the class for this specific type

An instance actually is a tuple of function values, sometimes referred to as a dictionary

Num Float

```
instance Num Float where
  (+)    = primPlusFloat
  (-)    = primMinusFloat
  (*)    = primMulFloat
  (/)    = primDivFloat
  negate = primNegFloat
```

Defining New Instances

We can make the rational numbers an instance of *Num* too:

```
data Rational = Rat (Int, Int)
instance Num Rational where
  Rat (x, y) + Rat (p, q) = simplify (Rat (x * q + y * p, y * q))
  Rat (x, y) - Rat (p, q) = simplify (Rat (x * q - y * p, y * q))
  Rat (x, y) * Rat (p, q) = simplify (Rat (x * p, y * q))
  Rat (x, y) / Rat (p, q) = simplify (Rat (x * q, y * p))
  negate (Rat (x, y))     = Rat (negate x, y)
simplify (Rat x y) = Rat (x `div` f) (y `div` f)
                     where f = gcd x y
```

How Haskell deals with constants

The class mechanism allows different versions of $(+)$ to co-exist.

How Haskell deals with constants

The class mechanism allows different versions of $(+)$ to co-exist.

Thus far constants cannot co-exist, and we have to write:

Int:	3
Float:	3.0
Ratio:	$\mathit{Rat}\ (3, 1)$
Complex:	$\mathit{Comp}\ (3.0, 0.0)$

If we define $half = Rat\ (1,2)$, we cannot write:

$$3 * half$$

because $half$ has type $Ratio$, whereas 3 has type Int.

Num revisited

To solve this problem the class *Num* contains one more function: *fromI nteger*.

```
class Num a where
  (+),(-),(*),(/) :: a→a→a
  negate          :: a    →a
  fromInteger     :: Int  →a
```

Num revisited

To solve this problem the class *Num* contains one more function: *fromI nteger*.

```
class Num a where
  (+), (−), (∗), (/) :: a → a → a
  negate             :: a      → a
  fromInteger        :: Int    → a
```

For each instance of *Num* we thus have to specify how an *Int* value can be converted to an *a* value.

For our own instances of Num ...

we define our own specific conversion function:

```
instance Num Rational where
  ....
  fromInteger n = Rat (n, 1)
```

Eq

The *Prelude* contains the definition of *Eq*:

```
class Eq a where
    (≡), (≢) :: a→a→Bool
```

and definitions for all the standard types

```
instance Eq Int where
    x ≡ y = primEqInt x y
    x ≢ y = ¬ (x ≡ y)
instance Eq Float where
    x ≡ y = primEqFloat x y
    x ≢ y = ¬ (x ≡ y)
instance Eq Char where
    x ≡ y = ord x ≡ ord y
    x ≢ y = ¬ (x ≡ y)
instance Eq Bool  where
    True  ≡  True  = True
    False ≡  False = True
    True  ≡  False = False
    False ≡  True  = False
    x ≢ y          = ¬ (x ≡ y)
```

Eq revisited

class $Eq\ a$ **where**

$$(\equiv), (\not\equiv) :: a \to a \to Bool$$
$$x \not\equiv y = \neg\ (x \equiv y)$$
$$x \equiv y = \neg\ (x \not\equiv y)$$

If we define an instance and omit either $\equiv$ or $\not\equiv$, then the default definition is taken.

Eq revisited

class $Eq\ a$ **where**

$$(\equiv), (\not\equiv) :: a \to a \to Bool$$
$$x \not\equiv y = \neg\ (x \equiv y)$$
$$x \equiv y = \neg\ (x \not\equiv y)$$

If we define an instance and omit either $\equiv$ or $\not\equiv$, then the default definition is taken.

Do not forget to provide at least one!

Using Eq instances in defining new instances

We can make $Rational$ an instance of Eq

instance $Eq\ Rational$ **where**

$$Rat\ (x, y) \equiv Rat\ (p, q) = x * q \equiv y * p$$

Using Eq instances in defining new instances

We can make $Rational$ an instance of Eq

instance $Eq\ Rational$ **where**
$$Rat\ (x, y) \equiv Rat\ (p, q) = x * q \equiv y * p$$

The $\equiv$ in the left hand side of the definition refers to equality for $Rational$'s which is being defined here, whereas the $\equiv$ in the right hand side refers to equality for Int's.

Super-classes

In the class definition of Ord we can require that the types which will be instances of Ord have to be instances of Eq too:

$$
\begin{array}{l}
\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where} \\
\quad (\leqslant), (<), (\geqslant), (>) :: a \rightarrow a \rightarrow Bool \\
\quad max, min \qquad\qquad\quad :: a \rightarrow a \rightarrow a
\end{array}
$$

Forall function expect $\leqslant$ have default definitions, expressed in terms of $\leqslant$ and $\equiv$.

The class Num

Num also requires its instances to be instance of Eq:

```
class Eq a ⇒ Num a where
  (+), (−), (∗), (/) :: a→a→a
  negate             :: a→a
  fromInteger        :: Int→a
```

There is no semantic reason for this; it is a way of expressing the intentions of the library designers.

Parametrised instances

Instance declarations can be parametrised (think of it as "functions from dictionaries to dictionaries")

Parametrised instances

Instance declarations can be parametrised (think of it as "functions from dictionaries to dictionaries") By giving a single definition we can make all lists an instance of Eq:

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = True \\
\quad [\,] \equiv (y : ys) = False \\
\quad (x : xs) \equiv [\,] = False \\
\quad (x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys
\end{array}
$$

One may read the first line as: 'if a is an instance of Eq,then so is $[a]$ an instance of Eq'.

Parametrised instances

Instance declarations can be parametrised (think of it as "functions from dictionaries to dictionaries") By giving a single definition we can make all lists an instance of Eq:

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = True \\
\quad [\,] \equiv (y : ys) = False \\
\quad (x : xs) \equiv [\,] = False \\
\quad (x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys
\end{array}
$$

The last line is remarkable; we have two calls to $\equiv$ in the right hand side, but these have different types:

Parametrised instances

Instance declarations can be parametrised (think of it as "functions from dictionaries to dictionaries") By giving a single definition we can make all lists an instance of Eq:

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = True \\
\quad [\,] \equiv (y : ys) = False \\
\quad (x : xs) \equiv [\,] = False \\
\quad (x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys
\end{array}
$$

The last line is remarkable; we have two calls to $\equiv$ in the right hand side, but these have different types:

- one compares two elements and is taken from the parameter dictionary
- the other is a recursive call to the function being defined

Another Example

The *Prelude* defines a lexicographic ordering between lists:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Ord}\ a \Rightarrow \mathit{Ord}\ [a]\ \textbf{where} \\
\quad [\,] \leqslant ys \qquad\qquad\quad = \mathit{True} \\
\quad (x : xs) \leqslant [\,] \qquad\quad = \mathit{False} \\
\quad (x : xs) \leqslant (y : ys) = x < y \vee (x \equiv y \wedge xs \leqslant ys)
\end{array}
$$

More than one instance parameter

Instances can specify several preconditions (parameters). We use it to define equality for tuples:

$$\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)\ \textbf{where}$$
$$\quad (x,y) \equiv (u,v) = x \equiv u \wedge y \equiv v$$

Elements of type (a,b) can be compared provided a and b can be compared, i.e. are instances of eq.

Attention!

The reserved symbol $\Rightarrow$ can be used in types, in instance-declarations and in class-declarations:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is a type-declaration: f is a function with type $a \rightarrow a$ provided a is a type in the class Num.

Attention!

The reserved symbol $\Rightarrow$ can be used in types, in instance-declarations and in class-declarations:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is a type-declaration: f is a function with type $a \rightarrow a$ provided a is a type in the class Num.
- **instance** $Eq\ a \Rightarrow Eq\ [a]$ is an instance-declaration: $[a]$ is an instance of Eq provided a is.

Attention!

The reserved symbol $\Rightarrow$ can be used in types, in instance-declarations and in class-declarations:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is a type-declaration: f is a function with type $a \rightarrow a$ provided a is a type in the class Num.
- **instance** $Eq\ a \Rightarrow Eq\ [a]$ is an instance-declaration: $[a]$ is an instance of Eq provided a is.
- **class** $Eq\ a \Rightarrow Ord\ a$ is a class-declaration: all instances of the new class Ord have to be an instance of Eq too.

Standard classes

The prelude defines the following classes:

- *Eq*, the class of type which can be compared;
- *Ord*, the class of types which have an ordering relation;
- *Num*, the class of types with have numeric properties;
- *Enum*, the class of types which can be enumerated;
- *Ix*, the class of types which can be used as an array index;
- *Read* and *Show*, the classes of types which have an external representation.

Enum

Enum is defined as:

```
class Ord a ⇒ Enum a where
  enumFrom        :: a        → [a]
  enumFromThen    :: a → a    → [a]
  enumFromTo      :: a → a    → [a]
  enumFromThenTo  :: a → a → a → [a]
```

Use of *Enum*

Example uses:

$$
\begin{aligned}
\mathit{enumFrom}\ 4 &= [4, 5, 6, 7, 8, \ldots \\
\mathit{enumFromThen}\ 4\ 6 &= [4, 6, 8, \ldots \\
\mathit{enumFromTo}\ \texttt{'c'}\ \texttt{'f'} &= [\texttt{'c'}, \texttt{'d'}, \texttt{'e'}, \texttt{'f'}] \\
\mathit{enumFromThenTo}\ 1.0\ 1.5\ 3.0 &= [1.0, 1.5, 2.0, 2.5, 3.0]
\end{aligned}
$$

Use of *Enum*

Example uses:

$$
\begin{aligned}
&\mathit{enumFrom}\ 4 &&= [4,5,6,7,8,\ldots\\
&\mathit{enumFromThen}\ 4\ 6 &&= [4,6,8,\ldots\\
&\mathit{enumFromTo}\ \texttt{'c'}\ \texttt{'f'} &&= [\texttt{'c'},\texttt{'d'},\texttt{'e'},\texttt{'f'}]\\
&\mathit{enumFromThenTo}\ 1.0\ 1.5\ 3.0 &&= [1.0,1.5,2.0,2.5,3.0]
\end{aligned}
$$

The functions are used to "desugar" the list notations $[x\,..]$, $[x,y\,..]$, $[x\,..\,y]$ and $[x,y\,..\,z]$

Enum Int and *Enum Char*

```
instance Enum Int where
  enumFrom       i   = iterate (1+) i
  enumFromThen i j = iterate ((j - i)+) i
```

Default definition for *enumFromTo* and *enumFromThenTo*

```
class Ord a ⇒ Enum a where
  ....
  enumFromTo       i j = takeWhile (j ⩾) (enumFrom i)
  enumFromThenTo i i' j
      | i' > i         = takeWhile (j ⩾)
                                   (enumFromThen i i')
      | otherwise      = takeWhile (j ⩽)
                                   (enumFromThen i i')
```

Default definition for *enumFromTo* and *enumFromThenTo*

```
class Ord a ⇒ Enum a where
  ....
  enumFromTo       i j = takeWhile (j ⩾) (enumFrom i)
  enumFromThenTo i i' j
      | i' > i         = takeWhile (j ⩾)
                                   (enumFromThen i i')
      | otherwise      = takeWhile (j ⩽)
                                   (enumFromThen i i')
```

This explains why instances of *Enum* have to be instances of *Ord* too.

Ix

```
class Ord a ⇒ Ix a where
  range   :: (a,a)       → [a]
  index   :: (a,a) → a → Int
  inRange :: (a,a) → a → Bool
```

Int is an instance of *Ix*

instance *Ix Int* **where**

$$
\begin{aligned}
&\mathit{range}\ (i,j) &&= [i\,..\,j] \\
&\mathit{index}\ (i,j)\ n &&= n - i \\
&\mathit{inRange}\ (i,j)\ n &&= i \leqslant n \wedge n \leqslant j
\end{aligned}
$$

Show

Types which have external (i.e. *String* representations, are instance of *Show*

> $show :: Show\ a \Rightarrow a \rightarrow String$

Show

Types which have external (i.e. *String* representations, are instance of *Show*

> $show :: Show\ a \Rightarrow a \rightarrow String$

All elementary types and composed types, such as lists and tuples, are instance of *Show*.

Error message 1: Cannot derive instance

You get this error message when you need (implicitly) an instance for a type but you did not declare one:

```
? (1,2,3) == (4,5,6)
ERROR: Cannot derive instance in expression
*** Expression        : (1,2,3) == (4,5,6)
*** Required instance : Eq (Int,Int,Int)
```

Error message 1: Cannot derive instance

You get this error message when you need (implicitly) an instance for a type but you did not declare one:

```
? (1,2,3) == (4,5,6)
ERROR: Cannot derive instance in expression
*** Expression        : (1,2,3) == (4,5,6)
*** Required instance : Eq (Int,Int,Int)
```

Here the standard prelude does not contain an instance declaration of *Eq* for triples. GHC is more permissive however.

Error message 1: Cannot derive instance

You get this error message when you need (implicitly) an instance for a type but you did not declare one:

```
? (1,2,3) == (4,5,6)
ERROR: Cannot derive instance in expression
*** Expression        : (1,2,3) == (4,5,6)
*** Required instance : Eq (Int,Int,Int)
```

Here the standard prelude does not contain an instance declaration of *Eq* for triples. GHC is more permissive however.

The solution lies in adding the missing instance declaration.

... continued ...

You also get this error message if you try to compare functions, since functions types are not instances of *Eq*:

```
? tail == drop 1 ERROR: Cannot derive instance
in expression *** Expression : tail == drop 1
*** Required instance : Eq ([a]->[a])
```

... continued ...

The same problem arises if an instance declaration is a special case of a more general case.

... continued ...

The same problem arises if an instance declaration is a special case of a more general case. The *Prelude* defines tuples as an instance of Eq:

$$\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)\ \textbf{where}$$
$$\quad (x,y) \equiv (u,v) = x \equiv u \wedge y \equiv v$$

... continued ...

The same problem arises if an instance declaration is a special case of a more general case. The *Prelude* defines tuples as an instance of Eq:

$$\begin{array}{l}\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)\ \textbf{where} \\ \quad (x,y) \equiv (u,v) = x \equiv u \wedge y \equiv v\end{array}$$

We now try define rational numbers as a tuple, and provide an instance for this case:

$$\begin{array}{l}\textbf{type}\ Rational = (Int, Int) \\ \textbf{instance}\ Eq\ Rational\ \textbf{where} \\ \quad (x,y) \equiv (u,v) = x * v \equiv u * y\end{array}$$

...

For $(1,2) \equiv (2,4)$ it is no longer possible to decide which definition for $\equiv$ to take here?

...

For $(1,2) \equiv (2,4)$ it is no longer possible to decide which definition for $\equiv$ to take here? Hence the compiler reports:

```
ERROR "file" (line 12):
Overlapping instances for class "Eq"
*** This instance : Eq (Int,Int)
*** Overlaps with : Eq (a,a)
*** Common instance : Eq (Int,Int)
```

...

For $(1,2) \equiv (2,4)$ it is no longer possible to decide which definition for $\equiv$ to take here? Hence the compiler reports:

```
ERROR "file" (line 12):
Overlapping instances for class "Eq"
*** This instance : Eq (Int,Int)
*** Overlaps with : Eq (a,a)
*** Common instance : Eq (Int,Int)
```

This problem can be solved here by defining:

$$\textbf{data}\ \mathit{Rational} = \mathit{Rat}\ (\mathit{Int}, \mathit{Int})$$

Problem 3: Unresolved overloading

Sometimes the compiler has insufficient information available to decide which instance to take:

```
? fromInteger 1 + fromInteger 2 ERROR:
Unresolved overloading *** type : Num a =>
a
```

Problem 3: Unresolved overloading

Sometimes the compiler has insufficient information available to decide which instance to take:

```
? fromInteger 1 + fromInteger 2 ERROR:
Unresolved overloading *** type : Num a =>
a
```

For *fromInteger* we can take the *Int* version or the *Float* version.

Problem 3: Unresolved overloading

Sometimes the compiler has insufficient information available to decide which instance to take:

```
? fromInteger 1 + fromInteger 2 ERROR:
Unresolved overloading *** type : Num a =>
a
```

For *fromInteger* we can take the *Int* version or the *Float* version. Since this is confusing the system will **default** to *Int* in such cases.

How to repair

Usually such problems can be solved by providing a type annotation:

```
? fromInteger 1 + fromInteger (2 :: Int) 3
```

The class *Functor*

The function $map :: (a \to b) \to [a] \to [b]$ is generalised in the *Prelude*e to:

```
class Functor f where
  fmap :: (a→b)→f a→f b
```

This captures the pattern that values of type $f\ a$ somehow "contain" values of type a, which can be converted to values of type b.

The class *Functor*

The function $map :: (a \to b) \to [a] \to [b]$ is generalised in the *Prelude*e to:

```
class Functor f where
  fmap :: (a→b)→f a→f b
```

This captures the pattern that values of type $f\ a$ somehow "contain" values of type a, which can be converted to values of type b. "Contains" should be taken liberally:

```
instance Functor ((→) c) where
  fmap :: (a→b)→(c→a)→(c→b)
  fmap f c2a = f ∘ c2a
```

The class *Functor*

The function $map :: (a \to b) \to [a] \to [b]$ is generalised in the *Prelude*e to:

```
class Functor f where
  fmap :: (a→b)→f a→f b
```

This captures the pattern that values of type $f\ a$ somehow "contain" values of type a, which can be converted to values of type b. "Contains" should be taken liberally:

```
instance Functor ((→) c) where
  fmap :: (a→b)→(c→a)→(c→b)
  fmap f c2a = f ∘ c2a
```

So the operator $\circ$ is actually the *map* equivalent for functions!

Monad

A second important class which is used in many places is:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a

  ma >> mb = ma >>= \_keyop -> mb
  fail s = error s
```

Monad

A second important class which is used in many places is:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a

  ma >> mb = ma >>= \_keyop -> mb
  fail s = error s
```

The **do**-notation is desugared into calls of these functions.

sequence and *sequence_* can be generalised

$$
\begin{array}{ll}
\mathit{sequence} & :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ [a] \\
\mathit{sequence} & = \mathit{foldr}\ \mathit{mcons}\ (\mathit{return}\ [\,]) \\
& \quad \textbf{where}\ \mathit{mcons}\ p\ q = p \gg\!= \\
& \qquad \lambda x \to q \gg\!= \\
& \qquad \lambda y \to \mathit{return}\ (x : y)) \\
\mathit{sequence_} & :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ () \\
\mathit{sequence_} & = \mathit{foldr}\ (\gg)\ (\mathit{return}\ ()) \\
\mathit{mapM} & :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [a] \\
\mathit{mapM}\ f\ \mathit{vs} & = \mathit{sequence}\ (\mathit{map}\ f\ \mathit{vs}) \\
\mathit{mapM_} & :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ () \\
\mathit{mapM_}\ f\ \mathit{vs} & = \mathit{sequence_}\ (\mathit{map}\ f\ \mathit{vs}) \\
(=\!\!\ll) & :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to m\ a \to m\ b \\
f =\!\!\ll x & = x \gg\!= f
\end{array}
$$

IO

In the "secret" part of the prelude *IO* has been made an instance of *Monad*:

```
data World = ...    -- a secret type
data IO a  = IO (World→(a, World))
```

IO

In the "secret" part of the prelude *IO* has been made an instance of *Monad*:

```
data World = ...    -- a secret type
data IO a   = IO (World→(a, World))
instance Functor IO where
fmap f (IO sea) = IO (mapFst ∘ sea)
   where mapFst (x, y) = (f x, y)
```

IO

In the "secret" part of the prelude *IO* has been made an instance of *Monad*:

```
data World = ...    -- a secret type
data IO a   = IO (World→(a, World))
instance Functor IO where
fmap f (IO sea) = IO (mapFst ∘ sea)
   where mapFst (x, y) = (f x, y)
instance Monad IO where
   (IO sea) >>= a2seb = IO (λwkeyop→let (a, nw) = sea w
                                      in a2seb a nw)
   return a           = IO ((, ) a)
```

IO

In the "secret" part of the prelude *IO* has been made an instance of *Monad*:

```
data World = ...    -- a secret type
data IO a   = IO (World→(a, World))
instance Functor IO where
fmap f (IO sea) = IO (mapFst ∘ sea)
   where mapFst (x, y) = (f x, y)
instance Monad IO where
   (IO sea) >>= a2seb = IO (λwkeyop→let (a, nw) = sea w
                                      in a2seb a nw)
   return a           = IO ((, ) a)
```

In this way we have created a "safe" interface to the outside world, which can only be accessed sequentially.