Fortgeschrittene funktionale Programmierung

LVA 185.A05, VU 2.0, ECTS 3.0
SS 2021

(Stand: 15.04.2021)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Lecture 4

Part IV: Advanced Language Concepts

- Chapter 12: Monads
  - Chap. 12.8: Recommended Reading: Basic, Advanced
- Chapter 13: Arrows
  - Chap. 13.7: Recommended Reading: Basic, Advanced

Outline in more Detail (1)

Part IV: Advanced Language Concepts

Outline in more Detail (2)

Chapter 12

Monads

Chapter 12.1

Motivation

Monad: The Mystic Type Constructor Class

```
class Monad m where
 (>>=) :: m a -> (a -> m b) -> m b
 ...
```

...is there any reason for the mystic aura around monads?

Compare monad with other type constructor classes:

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

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

Monad: The Mystic Type Constructor Class

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b
 return :: a -> m a
 (>>)   :: m a -> m b -> m b
 fail   :: String -> m a
 c >> k  =  c >>= \_ -> k
 fail s  =  error s
```

For comparison repeated:

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

Does the Name Itself

...give reason for a kind of mysticism?

Monad, derived from Greek *monas*, means:

– unit, unity (in German: Eins, Einheit).

Does the Usage of Monads

...(in other fields) give reason for a kind of mysticism?

Monads in Philosophy

Gottfried Wilhelm Leibniz (*1646 in Leipzig; †1716 in Hannover) used the monad notion as a counterpart of

- ‘atom’ denoting just as atom ‘something indivisable’

to ‘solve’ (more accurate possibly: tackle) the so-called

- body-soul problem (in German: Leib-Seele-Problem)

evolving from the body-soul dualism in the the classical formulation of René Descartes (*1596 in La Haye 50 km south of Tours, today Descartes; †1650 in Stockholm).

Monads in Category Theory

Eugenio Moggi introduced the monad notion to

- category theory

and used it for describing the

- semantics of programming languages.

in the realm of

- programming languages theory.

Eugenio Moggi. Computational Lambda Calculus and Monads. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.

Monads in Philosophy and Category Theory

Monads in Leibniz' Philosophy:

Definition (Gottfried Wilhelm Leibniz, 1714)

[Monadology, Paragraph 1]: The monad we want to talk about here is nothing else as a simple substance (German: Substanz), which is contained in the composite matter (German: Zusammengesetztes); simple means as much as: to be without parts.

Monads in Category Theory (cf. Saunders Mac Lane, 1971):

Definition (Eugenio Moggi, 1989)

[LICS'89]: A monad over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : Id_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ are natural transformations and the following equations hold:

$$\begin{array}{rcl} \mu_{TA}; \mu_A & = & T(\mu_a); \mu_A \\ \eta_{TA}; \mu_A & = & id_{TA} \quad = T(\eta_A); \mu_A \end{array}$$

... *"a monad is a monoid in the category of endofunctors."*

Monads in Functional Programming

...the monad notion became particularly popular in the field of functional programming (Philip Wadler, 1992) because (Haskell-style) monads

- allow to introduce some useful aspects of imperative programming such as sequencing into functional programming
- are well suited to smoothly integrate input/output into functional programming, as well as many other programming tasks and domains
- provide a suitable interface between functional programming and programming paradigms with side effects, in particular, imperative and object-oriented programming

...without breaking the functional paradigm!

These Capabilities let Monads

...appear to be a Suisse Knife of Functional Programming!

Monadic programming seems/is perfect for problems involving:

- Global state
  - Updating data during computation is often simpler than making all data dependencies explicit (the state monad).
- Huge data structures
  - No need for replicating a data structure that is not needed otherwise.
- Exception and error handling
  - The Maybe monad.
- ...
- Side-effects, explicit sequencing and evaluation orders
  - Canonical scenario: Input/output operations (the IO monad).

Good to Know

...the monad notion in functional programming lost its links to those in philosophy and category theory (almost) completely if there have been ever any tied ones, and hence, everything which might or might be considered a mystery or a miracle.

Rather than introducing a mystery, monads and monadic programming close a 'functional gap' between

- function application
- sequential function composition
- functorial mapping

Comparing Functorial and Monadic Mapping

- Functorial mapping:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
fmap k c = ... “(unpack, map, pack)“
```

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
(<*>) k c = ... “(unpack, unpack, map, pack)“
```

- Monadic mapping and sequencing:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>=) c k = ... “(unpack, map, repeat >>=)“
```

Why and How Monadic Sequencing? (1)

The associativity of `(>>=)` allows writing

```
(((((c >>= k) >>= k1) >>= k2) >>= k3) >>= k4)
```

more concisely:

```
c >>= k >>= k1 >>= k2 >>= k3 >>= k4
```

Double-checking types yields:

```
  c    >>=    k       >>=    k1       >>=    k2       >>=    k3       >>=    k4
:: m a :: a -> m b :: b -> m c :: c -> m d :: d -> m e :: e -> m g
```

```
:: m b
:: m c
:: m d
:: m e
:: m g
```

Why and How Monadic Sequencing? (2)

$$\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{k}}_{\texttt{:: a -> m b}} \;\texttt{>>=}\; \underbrace{\texttt{k1}}_{\texttt{:: b -> m c}} \;\texttt{>>=}\; \underbrace{\texttt{k2}}_{\texttt{:: c -> m d}} \;\texttt{>>=}\; \underbrace{\texttt{k3}}_{\texttt{:: d -> m e}} \;\texttt{>>=}\; \underbrace{\texttt{k4}}_{\texttt{:: e -> m g}} \;\texttt{:: m g}$$

$$\underbrace{\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{k}}_{\texttt{:: a -> m b}}}_{\texttt{c1}}$$

$$\underbrace{\underbrace{\texttt{c1}}_{\texttt{:: m b}} \;\texttt{>>=}\; \underbrace{\texttt{k1}}_{\texttt{:: b -> m c}}}_{\texttt{c2}}$$

$$\underbrace{\underbrace{\texttt{c2}}_{\texttt{:: m c}} \;\texttt{>>=}\; \underbrace{\texttt{k2}}_{\texttt{:: c -> m d}}}_{\texttt{c3}}$$

$$\underbrace{\underbrace{\texttt{c3}}_{\texttt{:: m d}} \;\texttt{>>=}\; \underbrace{\texttt{k3}}_{\texttt{:: d -> m e}}}_{\texttt{c4}}$$

$$\underbrace{\underbrace{\texttt{c4}}_{\texttt{:: m e}} \;\texttt{>>=}\; \underbrace{\texttt{k4}}_{\texttt{:: e -> m g}}}_{\texttt{c5}}$$

$$\underbrace{\texttt{c5}}_{\texttt{:: m g}}$$

Why and How Monadic Sequencing? (3)

$$\underbrace{\texttt{c}}_{\texttt{:: m a}} \texttt{ >>= } \underbrace{\texttt{k}}_{\texttt{:: a -> m b}} \texttt{ >>= } \underbrace{\texttt{k1}}_{\texttt{:: b -> m c}} \texttt{ >>= } \underbrace{\texttt{k2}}_{\texttt{:: c -> m d}} \texttt{ >>= } \underbrace{\texttt{k3}}_{\texttt{:: d -> m e}} \texttt{ >>= } \underbrace{\texttt{k4}}_{\texttt{:: e -> m g}} \texttt{ :: m g}$$

```
c   >>=  k   >>=  k1  >>=  k2  >>=  k3  >>=  k4
    ->>  c1  >>=  k1  >>=  k2  >>=  k3  >>=  k4
              ->>  c2  >>=  k2  >>=  k3  >>=  k4
                        ->>  c3  >>=  k3  >>=  k4
                                  ->>  c4  >>=  k4
                                          ->>  c5 :: m g
```

Why so Differently?

...why do functional composition and monadic sequencing look so differently?

Functional Composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)     -- (g . f) = \y -> g (f y)
```

Monadic Sequencing:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>=) c k = k “unpack c“              -- pseudo code
```

Or (using infix notation):

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
c >>= k = k “unpack c“                -- pseudo code
```

This Different Appearance is an Artifact!

The standard operator `(.)` for function composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)
```

...enables sequences of function applications applied R2L:

```
(k . (... . (h . (g . f))...)) x
                               ->> k (...(h (g (f x)))...)
```

We can define a dual operator `(;)` for function composition:

```
(;) :: (a -> b) -> (b -> c) -> (a -> c)
(f ; g) = (g . f)
```

...enabling sequences of function applications applied L2R:

```
((...((f ; g) ; h) ; ...) ; k) x
                               ->> k (...(h (g (f x)))...)
```

The Operator (;)

...suggests introducing another operator (>>;):

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x
```

enabling also sequences of function applications applied L2R:

(...(((x >>; f) >>; f1) >>; f2) >>; ... >>; fn)
  $\widehat{=}$ x >>; f >>; f1 >>; f2 >>; ... >>; fn

...where a value x is fed to the sequence of functions which are then applied one after the other to x (resp. its resulting images).

Opposing and Comparing

...non-monadic `(>>;)` and monadic `(>>=)` sequencing:

1. Ordinary Functional Sequencing from left to right:

   ```
   (>>;) :: a -> (a -> b) -> b
   x >>; f = f x
   ```

   ...enables L2R application sequences of the form:

   ```
   x >>; f >>; f1 >>; f2 >>; f3 >>; ... >>; fn
   ```

2. Monadic Functional Sequencing from left to right:

   ```
   (>>=) :: (Monad m) => m a -> (a -> m b) -> m b
   c >>= k = k "unpack c"
   ```

   ...enables L2R application sequences of the form:

   ```
   c >>= k >>= k1 >>= k2 >>= k3 >>= ... >>= kn
   ```

...reveals: There is no mystery at all!

Summing up

...the difference between `(>>;)` and `(>>=)` is a technical one:

```
(>>;) :: a -> (a -> b) -> b
x >>; f = f x
```

- The second argument `f` of `(>>;)` can directly be applied to its first argument `x`.
- This means, `(>>;)` is parametric polymorphic.

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
c >>= k = k “unpack c“
```

- The first argument `c` of `(>>=)` needs to be unpacked before its second argument `k` can be applied to it.
- The unpacking of the first argument is type specific.
- Hence, `(>>=)` can only be *ad hoc* polymorphic, and must be a member function of some type (constructor) class.
- This type constructor class is (called) `Monad`.

...again, except of this difference, no mystery!

Chapter 12.2

The Type Constructor Class Monad

The Type Constructor Class Monad

...monads are instances of the type constructor class Monad obeying the monad laws:

Type Constructor Class Monad

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

  c >> k   =  c >>= \_ -> k
  fail s   =  error s
```

Monad Laws

```
return x >>= f                  = f x                (ML1)
c >>= return                    = c                  (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g   (ML3)
```

Note

...monads must be 1-ary type constructors (like functors).

Intuitively, the monad laws require from (proper) monad instances:

- `return` is unit of `(>>=)`, i.e., it must pass its argument without any other effect (just as function `pure` of type constructor class `Applicative`) (ML1, ML2).
- `(>>=)` is associative, i.e., sequencings given by `(>>=)` must not depend on how they are bracketed (ML3).

Programmer obligation

- Programmers must prove that their instances of `Monad` satisfy the monad laws.

Note: Sequence operator `(>>=)`: Read as bind (Paul Hudak) or then (Simon Thompson). Sequence operator `(>>)`: Derived from `(>>=)`, read as sequence (Paul Hudak).

Type Constructor Class Monad in more Detail

```
class Monad m where
 -- 'Primary' functions (relevant for every monad)
 return :: a -> m a                -- Value 'lifting:' Ma-
                                   -- king a monadic value
 (>>=) :: m a -> (a -> m b) -> m b     -- Sequencing
 -- 'Secondary' functions (relevant for some monads)
 fail :: String -> m a             -- Error handling
 (>>) :: m a -> m b -> m b         -- Simplified sequencing
 -- Default implementations
```

$$\underbrace{\texttt{fail}\ \underbrace{\texttt{s}}_{\texttt{:: String}}}_{\texttt{:: m a}} \;=\; \underbrace{\texttt{error}\ \underbrace{\texttt{s}}_{\texttt{:: String}}}_{\texttt{:: m a}}$$

```
                                   -- Failing computation:
                                   -- Outputting s as errror
                                   -- error message
```

$$\underbrace{\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>}\; \underbrace{\texttt{k}}_{\texttt{:: m b}}}_{\texttt{:: m b}} \;=\; \underbrace{\underbrace{\texttt{c}}_{\texttt{:: m a}} \;\texttt{>>=}\; \underbrace{\texttt{\\\_ -> k}}_{\texttt{:: a -> m b}}}_{\texttt{:: m b}}$$

The Monad Laws in more Detail

...with added type information:

$$\underbrace{\underbrace{\overbrace{\texttt{:: a -> m a}}^{\texttt{return}}\ \overbrace{\texttt{:: a}}^{\texttt{x}}}_{\texttt{:: m a}}\ \texttt{>>=}\ \overbrace{\texttt{:: a -> m b}}^{\texttt{f}}}_{\texttt{:: m b}} \ = \ \underbrace{\overbrace{\texttt{:: a -> m b}}^{\texttt{f}}\ \overbrace{\texttt{:: a}}^{\texttt{x}}}_{\texttt{:: m b}} \qquad \texttt{(ML1)}$$

$$\underbrace{\overbrace{\texttt{:: m a}}^{\texttt{c}}\ \texttt{>>=}\ \overbrace{\texttt{:: a -> m a}}^{\texttt{return}}}_{\texttt{:: m a}} \ = \ \overbrace{\texttt{:: m a}}^{\texttt{c}} \qquad \texttt{(ML2)}$$

Associativity of (>>)

Lemma 12.2.2 (Associativity of (>>))

Monotonicity of (>>=) for some monad m implies that the default implementation of (>>) is associative, too, i.e.:

```
c1 >> (c2 >> c3) = (c1 >> c2) >> c3
```

Compared with the associativity statement of Lemma 12.2.2 for (>>), the left-hand side of (ML3) requiring the associativity of (>>=) looks 'ugly:'

```
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g   (ML3)
```

To improve on this, we introduce a new operator (>@>):

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c)
                                        -> (a -> m c)
f >@> g = \x -> (f x) >>= g
```

The Monad Laws in Terms of (>@>)

...using (>@>), the monad laws, especially the associativity requirement, look as natural and obvious as for (>>).

Lemma 12.2.3

If (>>=) and `return` of some monad `m` are associative and unit of (>>=), respectively, then we have:

```
return >@> f      = f                     (ML1')
f >@> return      = f                     (ML2')
(f >@> g) >@> h = f >@> (g >@> h)         (ML3')
```

Intuitively

- `return` is unit of (>@>) (ML1′, ML2′).
- (>@>) is associative (ML3′).

A Law linking Classes Monad and Functor

...type constructors, which shall be proper instances of both Monad and Functor must satisfy law MFL:

```
fmap g xs  =  xs >>= return . g                    (MFL)
         ( =  do x <- xs; return (g x) )
```

(regarding the do-notation, refer to Chapter 12.3.)

Selected Utility Functions for Monads (1)

```
(=<<)         :: Monad m => (a -> m b) -> m a -> m b
f =<< x       =  x >>= f

sequence      :: Monad m => [m a] -> m [a]
sequence      =  foldr mcons (return [])
                   where mcons p q = do l  <- p
                                        ls <- q
                                        return (l:ls)

sequence_     :: Monad m => [m a] -> m ()
sequence_     =  foldr (>>) (return ())

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

mapM_         :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as =  sequence_ (map f as)
```

Selected Utility Functions for Monads (2)

```
mapF        :: Monad m => (a -> b) -> m [a] -> m [b]
mapF f x    =  do v <- x; return (f v)
   -- equals map on lists, i.e., for picking [] as m
joinM        :: Monad m => m (m a) -> m a
joinM x    =  do v <- x; v
 -- equals concat on lists, i.e., for picking [] as m
```

...and many more (see e.g., library Monad).

Lemma 12.2.4

1. `mapF (f . g) = mapF . mapF g`
2. `joinM return = joinM . mapF return`
3. `joinM return = id`

Chapter 12.3

Syntactic Sugar: The do-Notation

The do-Notation

...the monadic operations `(>>=)` and `(>>)` allow very much as functional composition `(.)`

- to explicitly specify the sequencing of (fitting) operations.

Both functional and monadic sequencing introduce

- an imperative flavour into functional programming.

The syntactic sugar of the so-called

- do-notation

replacing `(>>=)` and `(>>)` allows to express this imperative flavour of monadic sequencing syntactically even more compelling and concise.

Relating Monadic Operations and do-Notation

...four conversion rules allow converting sequences of monadic operations composed of

– (>>=) and (>>)

into equivalent ('<=>') sequences of

– do-blocks

and vice versa.

Intuitively

Recall:

```
(>>=) :: m a -> (a -> m b) -> m b
(>>)  :: m a -> m b -> m b
```

Then:

$$\underbrace{\texttt{dc v}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{\texttt{:: (a -> m b)}}\ \texttt{->>}\ \underbrace{\texttt{f v}}_{\texttt{:: m b}}$$

$$\texttt{"<=> do}\ \underbrace{\texttt{x}}_{\texttt{:: a}}\ \texttt{<-}\ \underbrace{\texttt{dc v}}_{\texttt{:: m a}}\texttt{;}\ \underbrace{\texttt{y}}_{\texttt{:: b}}\ \texttt{<-}\ \underbrace{\texttt{f x}}_{\texttt{:: m b}}\texttt{;}\ \underbrace{\texttt{return y}}_{\texttt{:: m b}}\texttt{"}$$

$$\underbrace{\texttt{dc v}}_{\texttt{:: m a}}\ \texttt{>>}\ \underbrace{\texttt{dc}'\ \texttt{v}'}_{\texttt{:: m b}}\ \texttt{->>}\ \underbrace{\texttt{dc v}}_{\texttt{:: m a}}\ \texttt{>>=}\ \underbrace{\texttt{\_ -> dc}'\ \texttt{v}'}_{\texttt{:: (a -> m b)}}$$

$$\texttt{"<=> do}\ \underbrace{\texttt{\_}}_{\texttt{:: a}}\ \texttt{<-}\ \underbrace{\texttt{dc v}}_{\texttt{:: m a}}\texttt{;}\ \underbrace{\texttt{y}}_{\texttt{:: b}}\ \texttt{<-}\ \underbrace{\texttt{dc}'\ \texttt{v}'}_{\texttt{:: m b}}\texttt{;}\ \underbrace{\texttt{return y}}_{\texttt{:: m b}}\texttt{"}$$

with `dc`, $\texttt{dc}'$ some data constructors of type constructor `m`.

The Conversion Rules

```
(R1) do e <=> e

(R2) do e1;e2;...;en <=> e1 >>= \_ -> do e2;...;en
                     <=> e1 >> do e2;...;en

(R3) do let decl_list;e2;...;en <=> let decl_list
                                    in do e2;...;en

(R4) do pattern <- e1;e2;...;en <=>
        let ok pattern = do e2;...;en
            ok _       = fail "..."
            in e1 >>= ok
```

...and as a special case of the 'pattern' rule (R4):

```
(R4') do x <- e1;e2;...;en <=>
         e1 >>= \x -> do e2;...;en
```

Notes on the Conversion Rules

Intuitively

- (R2): If the return value of an operation is not needed, it can be moved to the front.
- (R3): A `let`-expression storing a value can be placed in front of the `do`-block.
- (R4): Return values bound to a pattern require a supporting function that handles the pattern matching and the execution of the remaining operations, or that calls `fail`, if the pattern matching fails.

  Note: It is rule (R4) which necessitates `fail` as a monadic operation in `Monad`. Overwriting this operation allows a monad-specific exception and error handling.

Illustrating the do-Notation

...using the monad laws as example.

A) The monad laws using (>>=) and (>>):

```
return a >>= f                = f a                   (ML1)
c >>= return                  = c                     (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g      (ML3)
```

B) The monad laws using do-notation:

```
do x <- return a; f x       =  f a                (ML1)
do x <- c; return x         =  c                  (ML2)
do x <- c; y <- f x; g y   =
            do y <- (do x <- c; f x); g y       (ML3)
```

Semicolons vs. Linebreaks in do-Notation

B) do-notation in 'one' line (w/ ';', no linebreaks):

```
do x <- return a; f x      =  f a                           (ML1)
do x <- c; return x        =  c                             (ML2)
do x <- c; y <- f x; g y   =
            do y <- (do x <- c; f x); g y                   (ML3)
```

C) do-notation in 'several' lines (w/ linebreaks, no ';'):

```
do x <- return a
   f x            =  f a                                    (ML1)
do x <- c
   return x       =  c                                      (ML2)
do x <- c
   y <- f x
   g y            =  do y <- (do x <- c
                                 f x)
                        g y                                 (ML3)
```

Chapter 12.4

Monad Examples

Predefined Monads in Haskell

We consider a selection of predefined monads:

- Identity monad
- List monad
- Maybe monad
- Map monad
- State monad
- Input/Output monad

...but there are many more of them predefined in Haskell:

- Writer monad
- Reader monad
- Failure monad
- ...

As a Rule of Thumb

...when making a 1-ary type constructor a monad, then:

- `(>>=)` will be defined to unpack the value of the first argument, map the second argument over it, and return the packed result this yields.
- `return` will be defined in the most straightforward way to lift the argument value to its monadic counterpart.
- `(>>)` and `fail` are usually not to be implemented afresh. Usually, their default implementations provided in type constructor class `Monad` are just fine.

  If the default implementations of `(>>)` and `fail` are used, this means for

  - `(>>)`: the first argument is evaluated and dropped, the second argument is evaluated and returned as result (makes sense for some monads like the IO-monad).
  - `fail`: the computation stops by calling `error` with some appropriate error message.

Chapter 12.4.1

The Identity Monad

The Identity Monad

...making the 1-ary type constructor `Id` an instance of `Monad` (conceptually the simplest monad):

```
newtype Id a = Id a

instance Monad Id where
  (Id x) >>= f   =  f x
  return         =  Id
```

Note:

- `Id`: 1-ary type constructor, i.e., if a is a type variable, then `Id a` denotes a type.
- `Id`: 1-ary data (or value) constructor, i.e., if `x :: a`, then `Id x` is a value of type `Id a`: `Id x :: Id a`.
- `(>>)`, `fail` implicitly defined by default implementations.
- ```
  (>>=)  :: Id a -> (a -> Id b) -> Id b
  return :: a -> Id a
  (>>)   :: Id a -> Id b -> Id b
  ```
  ```

Proof Obligation: The Monad Laws

Lemma 12.4.1.1 (Soundness of Identity Monad)

The `Id` instance of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

...`Id` is thus a proper instance of `Monad`, the so-called identity monad.

The Identity Monad Operations in more Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
```

The instance declaration for `Id` with added type information:

```
instance Monad Id where
```

$$\underbrace{\texttt{Id x}}_{\texttt{:: Id a}}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{\texttt{:: a -> Id b}}\ =\ \underbrace{\texttt{f x}}_{\texttt{:: Id b}}\ \texttt{-- yields an (Id b)-value}$$

$$\texttt{return}\ \underbrace{\texttt{x}}_{\texttt{:: a}}\ =\ \underbrace{\texttt{Id x}}_{\texttt{:: Id a}}\ \texttt{-- yields an (Id a)-value}$$

Recall the overloading of `Id` (`newtype Id a = Id a`):

- `Id` followed by `x`: `Id` is data (or value) constructor (`Id` $\widehat{=}$ `Id`).
- `Id` followed by `a` or `b`: `Id` is type constructor (`Id` $\widehat{=}$ `Id`).

Note

Intuitively

- The identity monad maps a type to itself.
- It represents the trivial state, in which no actions are performed, and values are returned immediately.
- It is useful because it allows to specify computation sequences on values of its type (cf. Chapter 12.5.1)

Moreover

- The operation `(>@>)` boils down to forward composition of functions `(>.>)` ($\widehat{=}$ `(>>;)`) for the identity monad:

```
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
g >.> f = f . g = g ; f
```

- Forward composition of functions `(>.>)` is associative with unit element `id`.

Chapter 12.4.2

The List Monad

The List Monad

...making the 1-ary type constructor `[]` an instance of `Monad`:

```
instance Monad [] where
  xs >>= f = concat (map f xs)   -- concat, map:
  return x = [x]                 -- Standard Prelude
  fail s   = []
```

Note:

- `[]`: 1-ary type constructor, i.e., if `a` is a type variable, then `[a]` ($\hat{=}$ `[] a`) denotes a type.
- `[]`: 1-ary data (or value) constructor, i.e., if `x :: a`, then `[x]` is a value of type `[a]`: `[x] :: [a]`; in particular, `[]` is a value, the empty list, i.e., `[] :: [a]`
- `(>>)` is implicitly defined by its default implementation; the default implementation of `fail` is overwritten.
- ```
  (>>=)  :: [] a -> (a -> [] b) -> [] b
  return :: a -> [] a
  (>>)   :: [] a -> [] b -> [] b
  ```
  ```

Proof Obligation: The Monad Laws

Lemma 12.4.2.1 (Soundness of List Monad)

The [] instance of Monad satisfies the three monad laws ML1, ML2, and ML3.

...[] is thus a proper instance of Monad, the so-called identity monad.

For convenience, we recall from the Standard Prelude:

```
concat :: [[a]] -> [a]
concat lss = foldr (++) [] lss
concat [[1,2,3],[4],[5,6]] ->> [1,2,3,4,5,6]
```

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs
map (*2) [1,2,3] ->> [2,4,6]
```

The List Monad Operations in more Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for [] with added type information:

$$\texttt{instance Monad [] where}$$

$$\underbrace{\texttt{xs}}_{::\,[]\,a}\ \texttt{>>=}\ \underbrace{\texttt{f}}_{::\,a\,\texttt{->}\,[]\,b} = \underbrace{\texttt{concat}\ \underbrace{\texttt{(map f xs)}}_{::\,[]\,([]\,b)}}_{::\,[]\,b} \quad \texttt{-- yields a [b]-list}$$

$$\texttt{return}\ \underbrace{\texttt{x}}_{::\,a} = \underbrace{\texttt{[x]}}_{::\,[]\,a} \quad \texttt{-- yields the singleton list [x]}$$

$$\underbrace{\texttt{fail s}}_{::\,\texttt{String}} = \underbrace{\texttt{[]}}_{::\,[]\,a} \quad \texttt{-- yields the empty list []}$$

Example: Applying the Monad Operations

```
ls = [1,2,3] :: [] Int
f = \n -> [(n,odd(n))] :: Int -> [] (Int,Bool)
g = \n -> [x*n | x <- [1.5,2.5,3.5]] :: Int -> [] Float
h = \n -> [1..n] :: Int -> [] Int

h 3 >>= f
  ->> ls >>= f
  ->> concat [ [(1,True)], [(2,False)], [(3,True)] ]
  ->> [(1,True),(2,False),(3,True)] :: [] (Int,Bool)

h 3 >>= g
  ->> ls >>= g
  ->> concat [ [ x*n | x <- [1.5,2.5,3.5] ] | n <- [1,2,3] ]
  ->> concat [ [1.5*1,2.5*1,3.5*1], [1.5*2,2.5*2,3.5*2],
               [1.5*3,2.5*3,3.5*3] ]
  ->> concat [ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] ]
  ->> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] Float
```

The Example in More Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for [] with added type information:

```
instance Monad [] where
  xs   >>=   f      = concat (map f xs)    -- yields a [b]-list
:: [] a  :: a -> [] b         :: [] ([] b)
                               :: [] b
  return x        = [x]   -- yields the singleton list [x]
       :: a        :: [] a
  fail s          = []         -- yields the empty list []
   :: String      :: [] a
```

Examples:

```
ls = [1,2,3] :: [] Int
f = \n -> [(n,odd(n))] :: Int -> [] (Int,Bool)
g = \n -> [x*n | x <- [1.5,2.5,3.5]] :: Int -> [] Float
h = \n -> [1..n] :: Int -> [] Int

h 3 >>= f ->> ls >>= f ->> concat [ [(1,True)], [(2,False)], [(3,True)] ]
          ->> [(1,True),(2,False),(3,True)] :: [] (Int,Bool)

h 3 >>= g ->> ls >>= g ->> concat [ [ x*n | x <- [1.5,2.5,3.5] ] | n <- [1,2,3] ]
          ->> concat [ [1.5*1,2.5*1,3.5*1], [1.5*2,2.5*2,3.5*2], [1.5*3,2.5*3,3.5*3] ]
          ->> concat [ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] ]
          ->> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] Float
```

Reconsidering the List Monad Implementation

...the list monad could have equivalently been implemented by:

```
instance Monad [] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  [] >>= f = []
  return x = [x]
  fail s   = []
```

Recall: The operations (>>=) and return of the list monad have types:

```
(>>=)  :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

List Monad and List Comprehension

...the list monad and list comprehension are closely related:

```
do x <- [1,2,3]
   y <- [4,5,6]
   return (x,y)
->> [(1,4),(1,5),(1,6),
     (2,4),(2,5),(2,6),
     (3,4),(3,5),(3,6)]
```

In fact, the following expressions are equivalent:

Proposition 12.4.2.2

```
[(x,y) | x <- [1,2,3], y <- [4,5,6] ] <=>
                                  do x <- [1,2,3]
                                     y <- [4,5,6]
                                     return (x,y)
```

...list comprehension is syntactic sugar for monadic syntax!

List comprehension: Syntactic Sugar

...for monadic syntax.

We have:

Lemma 12.4.2.3

```
[f x | x <- xs] <=> do x <- xs; return (f x)
```

Lemma 12.4.2.4

```
[a | a <- as, p a] <=>
  do a <- as; if (p a) then return a else fail ""
```

Exercise 12.4.2.5

Prove by stepwise evaluation the equivalences stated in:

1. Proposition 12.4.2.2
2. Lemma 12.4.2.3
3. Lemma 12.4.2.4

Chapter 12.4.3

The Maybe Monad

The Maybe Monad

...making the 1-ary type constructor Maybe a monad:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return         = Just
  fail s         = Nothing
```

Note:

- ```
  (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
  return :: a -> Maybe a
  (>>)   :: Maybe a -> Maybe b -> Maybe b
  ```
- The Maybe monad is useful for computation sequences that can produce a result, but might also produce an error.
  ```

Proof Obligation: The Monad Laws

Lemma 12.4.3.1 (Soundness of Maybe Monad)

The Maybe instance of Monad satisfies the three monad laws ML1, ML2, and ML3.

...Maybe is thus a proper instance of Monad, the so-called maybe monad.

Recall that Maybe is also an instance of Functor:

```
instance Functor Maybe where
 fmap f Nothing  = Nothing
 fmap f (Just x) = Just (f x)
```

Lemma 12.4.3.2 (MFL Soundness of Maybe Mo/Fu)

The Maybe instances of Monad and Functor satisfy law MFL (of Chap. 12.2).

The Maybe Monad Operations in More Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for `Maybe` with added type information:

`instance Monad Maybe where`

$$\underbrace{\texttt{Just x}}_{::\,\text{Maybe a}}\ \texttt{>>=}\ \underbrace{\texttt{k}}_{::\,\text{a -> Maybe b}}\ =\ \underbrace{\texttt{k x}}_{::\,\text{Maybe b}}\qquad \texttt{-- yields a Just-value}$$

$$\underbrace{\texttt{Nothing}}_{::\,\text{Maybe a}}\ \texttt{>>=}\ \underbrace{\texttt{k}}_{::\,\text{a -> Maybe b}}\ =\ \underbrace{\texttt{Nothing}}_{::\,\text{Maybe b}}\qquad \texttt{-- yields the Nothing-value}$$

$$\texttt{return}\ \underbrace{\texttt{x}}_{::\,\text{a}}\ =\ \underbrace{\texttt{Just x}}_{::\,\text{Maybe a}}\qquad \texttt{-- yields the Just-value}$$

$$\texttt{fail}\ \underbrace{\texttt{s}}_{::\,\text{String}}\ =\ \underbrace{\texttt{Nothing}}_{::\,\text{Maybe a}}\qquad \texttt{-- yields the empty list}$$

Example: Error Handling: (1)

...or: How to compose functions with monadic value ranges.

Let f′, g′ be two functions of type:

```
f′ :: a -> b
g′ :: b -> c
```

Obviously, composing f′ and g′ sequentially is straightforward:

```
h′ :: a -> c
h′ = (g′ . f′)

h′ x ->> (g′ . f′) x ->>  g′ (f′ x)
```

Example: Error Handling (2)

If the computations of f' and g' can fail, this can be taken care of by replacing f' and g' by two new functions f and g embedding the computation into the Maybe type:

```
f :: a -> Maybe b                       -- f replaces f′
g :: b -> Maybe c                       -- g replaces g′
```

Unlike f' and g', however, f and g can not straightforwardly be sequentially composed:

```
h :: a -> Maybe c                    -- "h = (g . f)":
h x = case (f x) of               -- Composing f and g
       Nothing -> Nothing          -- requires nested
       Just y  -> case (g y) of    -- case clauses
                   Nothing -> Nothing
                   Just z  -> Just z
```

Though possible, the explicit nesting of cases to sequentially compose f and g is inconvenient and tedious.

Example: Error Handling (3)

Step 1: Hiding nestings.

...embedding f' and g' into the Maybe type gets a lot easier by exploiting the monad property of Maybe: Using the monadic sequencing operations for composing f and g allows:

```
h :: a -> Maybe c                    -- "h = (g . f)"
h x = f x >>= \y -> g y >>= \z -> return z
```

or, equivalently, using the do notation:

```
h :: a -> Maybe c                    -- "h = (g . f)"
h x = do y <- f x
         z <- g y
         return z
```

...the 'nasty' error checks are now hidden in the implementation of the bind operation (>>=) of the maybe monad.

Example: Error Handling (4)

Step 2: Hiding the bind operation (>>=).

Note that the sequence of monad operations:

```
f x >>= \y -> g y >>= \z -> return z
```

can be simplified to:

```
f x >>= \y -> g y >>= \z -> return z
    <=> (simplification by currying)
f x >>= \y -> g y >>= return
   <=> (monad law for return)
f x >>= \y -> g y
   <=> (simplification by currying)
f x >>= g
```

Hence, `h x` (“= `g (f x)`”) is equivalent to `f x >>= g`.

Example: Error Handling (5)

...making use of this observation and introducing function:

```
composeM :: Monad m => (b -> m c) ->
                              (a -> m b) -> (a -> m c)
(g `composeM` f) x = f x >>= g
```

allows an even more pleasing notation for composing f and g:

```
h :: a -> Maybe c                    -- "h = (g . f)"
h = (g `composeM` f)
```

Hence, we get:

```
(g `composeM` f)
```

as the monadic notational counterpart of sequentially composing f' and g':

```
(g' . f')
```

Example: Error Handling (6)

Overall: Using monadic sequencing

`f x >>= g` (or equivalently: `(g 'composeM' f) x`)

for embedding the composition of f' and g' into the `Maybe` type preserves the original syntactical form of composing f' and g':

$$(g' \; . \; f') \; x = g' \; (f' \; x)$$

in almost a 1-to-1 kind:

```
(g `composeM` f) x = f x >>= g
```

Chapter 12.4.4

The Either Monad

Exercise 12.4.4.1 The Either Monad

1. Make the type constructor `(Either a)` a monad.
2. Provide (most general) type information for the defining equations of the monad operations `(>>=)`, `(>>)`, `return`, and `fail` of `(Either a)`.
3. Prove that `(Either a)` satisfies the monad laws.
4. Does your implementation of the `(Either a)` monad instance and the implementation of the `(Either a)` functor instance of Chapter 10.3.4 satisfy the law FML (of Chap. 12.2)? Prove or provide a counter-example.

Chapter 12.4.5

The Map Monad

The Map Monad

...making the 1-ary type constructor ((->) d) a monad:

```
instance Monad ((->) d) where
  h >>= f  = \x -> f (h x) x
  return x = \_ -> x
```

Note: (d for domain, r for range)

```
(>>=)  :: ((->) d) r -> (r -> ((->) d) r') -> ((->) d) r'
return :: r -> ((->) d) r
(>>)   :: ((->) d) r -> ((->) d) r' -> ((->) d) r'
```

Proof obligation: The monad laws

Lemma 12.4.5.1 (Soundness of Map Monad)

The ((->) d) instance of Monad satisfies the three monad laws ML1, ML2, and ML3.

...((->) d) is thus a proper instance of Monad, the so-called map monad.

Example (w/ String, Int, (Bool,String) for d, r, r′, resp.) (1)

```
(>>=) :: ((->) d) r -> (r -> ((->) d) r′) -> ((->) d) r′
  ( ≙ (>>=) :: (d -> r) -> (r -> (d -> r′)) -> (d -> r′) )
h >>= f  = \x -> f (h x) x

h_length :: ((->) String) Int
  ( ≙ h_length :: String -> Int )
h_length = length
f_cp_p :: Int -> ((->) String) ((,) Bool String)
  ( ≙ f_cp_p :: Int -> (String -> (Bool,String) )
f_cp_p n s = (,) (mod n 2 == 1) (copy n s)
 where copy n s = if n > 0 then s ++ " " ++ copy (n-1) s else ""
g :: ((->) String) ((,) Bool String)
  ( ≙ g :: String -> (Bool,String) )
g = \s -> f_cp_p (h_length s) s
 ( ≙ g s = (mod (length s) 2 == 1,copy (length s) s) )

h_length >>= f_cp_p
 ->> (\x -> f_cp_p (h_length x) x)     ( = g )

(h_length >>= f_cp_p) "Fun"
 ->> ... ->> (True,"Fun Fun Fun")
```

Example (w/ String, Int, (Bool,String) for d, r, r′, resp.) (2)

...in more detail:

```
h_length >>= f_cp_p
 ->> (\x -> f_cp_p (h_length x) x)
  = g    ( :: String -> (Bool,String) )

(h_length >>= f_cp_p) "Fun"
 ->> (\x -> f_cp_p (h_length x) x) "Fun"
  = g "Fun"
 ->> (mod (length "Fun") 2 == 1,copy (length "Fun") "Fun")
 ->> (mod 3 2 == 1,copy 3 "Fun")
 ->> (True,"Fun Fun Fun")       ( :: (Bool,String) )
```

Example (w/ String, Int, (Bool,String) for d, r, r′, resp.) (3)

```
(>>=)  :: ((->) d) r -> (r -> ((->) d) r′) -> ((->) d) r′
h >>= f  = \x -> f (h x) x

return :: r -> ((->) d) r (≙ return :: Int -> ((->) String) Int)
return x = \_ -> x         ≙ return :: Int -> (String -> Int))

return 0 = \_ -> 0    ( :: String -> Int )

return 0 >>= f_cp_p
 ->> \x -> f_cp_p ((return 0) x ) x
 ->> \x -> f_cp_p (\_ -> 0) x) x ( :: String -> (Bool,String) )

(return 0 >>= f_cp_p) "Fun"
 ->> (\x -> f_cp_p ((return 0) x ) x) "Fun"
 ->> f_cp_p ((return 0) "Fun" ) "Fun"
 ->> f_cp_p ((\_ -> 0) "Fun") "Fun"
 ->> f_cp_p 0 "Fun"
 ->> (mod 0 2 == 1,copy 0 "Fun")
 ->> (False,"")        ( :: (Bool,String) )

(return 1 >>= f_cp_p) "Fun" ->> ... ->> (True,"Fun")
(return 2 >>= f_cp_p) "Fun" ->> ... ->> (False,"Fun Fun")
(return 3 >>= f_cp_p) "Fun" ->> ... ->> (True,"Fun Fun Fun")
```

Example (w/ String, Int for d, r, resp.) (4)

```
(>>=)  :: ((->) d) r -> (r -> ((->) d) r') -> ((->) d) r'
h >>= f  = \x -> f (h x) x

return :: r -> ((->) d) r (≙ return :: Int -> ((->) String) Int)
return x = \_ -> x          ≙ return :: Int -> (String -> Int))

return 3 = \_ -> 3    ( :: String -> Int )

h_length >>= return
 ->> \x -> return (h_length x) x
 ->> \x -> return (length x) x
 ->> \x -> (\_ -> length x) x    ( :: String -> Int )

(h_length >>= return) "Fun"
 ->> (\x -> (return (h_length x) x)) "Fun"
 ->> return (h_length "Fun") "Fun"
 ->> return (length "Fun") "Fun"
 ->> return 3 "Fun"
 ->> (\_ -> 3) "Fun"
 ->> 3    ( :: Int )
```

Exercise 12.4.5.2

1. Recall the monad operations:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
```

   Add (most general) type information for the instance declaration of `((->) d)`:

```
instance Monad ((->) d) where
  h   >>=  f   =  \x -> f (h x) x
  return x     =  \_ -> x
```

2. Evaluate stepwise:
   2.1 `(return 2 >>= f_cp_p) "Fun"`
   2.2 `(h_length >>= return) "Fun Prog"`
   2.3 `(h_length >>= return >>= f_cp_p) "Fun"`

Chapter 12.4.6

The State Monad

Objective: Modelling Global State, Side-Effects

...by means of functions, so-called state transformers, which, applied to some current state s yield a new state s′ together with some additional result at the side.

Key: The state monad of an appropriate state type:

```
newtype State st a = St (st -> (st,a))
```

where

- `State` : 2-ary type constructor (bundling `st` and `a`).
- `st`, `a`: Type variables (concrete types inserted for `st` and `a` are the actual state type of interest and the type of some additional result of state transformers, resp.).
- `St (st -> (st,a))`: State values capsulating state transformers mapping 'old' to 'new' states plus delivering some additional result.

State Transformers

...map (or: transform) global (internal program) states of a type `st` into (possibly modified) new states of the same type `st` computing additionally a result of some type `a`.

In more detail:

State transformers are mappings `m` of type:

```
m :: st -> (st,a)
```

mapping states `s :: st` to pairs of (possibly modified result) states `s' :: st` and values `x :: a`:

$$\underbrace{\texttt{m s}}_{\texttt{:: st}} \;\texttt{->>}\; (\underbrace{\texttt{s'}}_{\texttt{:: st}} \texttt{,} \underbrace{\texttt{x}}_{\texttt{:: a}})$$

The State Monad

...making the 1-ary type constructor (State st) resulting from partially evaluating the 2-ary type constructor State

```
newtype (State st) a = St (st -> (st,a))
```

a monad:

```
instance Monad (State st) where
 (St h) >>= f = St (\s -> let (s',x) = h s
                              St f'  = f x
                          in f' s')

 return x = St (\s -> (s,x))
```

Note: The sequence operation (>>) and fail inherit their default implementations of type constructor class Monad.

Stepwise developing bind operation (>>=) (1)

```
(>>=) :: (State st) a -> (a -> (State st) b) -> (State st) b
(St h) >>= f = St g
  where g :: st -> (st,b)
    ⇒ g = "apply h, then apply f to h's result" ⇐
  wrt given maps h :: st -> (st,a)
                 f :: a  -> (State st) b
                 where values of type (State st) b look like:
                 St k :: (State st) b with k :: st -> (st,b)
  ensuring St g :: (State st) b is of type (State st) b
  as required.
```

This might look confusing at first sight but we are well familar with the pattern "apply h, then apply f to h's result" from sequentially composing functions:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . h) x = f (h x)
```

Let us thus look into this pattern in more detail...

Stepwise developing bind operation (>>=) (2)

Recall how two functions `f` and `h` are sequentially composed:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . h) x = f (h x)
```

The sequential composition `(f . h)` of `f` and `h` applies `f` to the result yielded by `h` applied to `x`: This "apply `f` to `h`'s result" gets even more obvious by introducing name `y` for the result `h` yields applied to `x` and passing this name as argument to `f`:

```
(f . h) x = let y = h x
                z = f y
            in z
```

Note: `y` denotes the intermediate result yielded by `h` applied to `x`. `y` as intermediate result is passed as argument to `f` yielding `z`, which is already the result of sequentially composing `f` and `h`.

Stepwise developing bind operation (>>=) (3)

The sequential composition (f . h) of f and h is itself a function: let's name it g. This gets obvious by defining (f . h) pointfree:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . h = g where g :: (a -> c)
                g = \x -> let y = h x
                              z = f y
                          in z
```

Note: This definition is nothing else as the answer to asking how to define the sequential composition (f . h) of two functions f and h we could have started our considerations of (f . h) with:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . h = g
   where g :: a -> c
      ⇒ g = "apply h, then apply f to h's result" ⇐
   wrt given maps h :: a -> b
                  f :: b -> c
                  where values of type c look like:
                  k :: c (with k w/out further inner structure)
```

Cp. the two patterns and note their similarity:

Pattern 1: Sequential composition of `f` and `h`:

```
 (.) :: (b -> c) -> (a -> b) -> (a -> c)
 f . h = g
    where g :: a -> c
          g = "apply h, then apply f to h's result"
    wrt given maps h :: a -> b
                   f :: b -> c
                   where values of type c look like:
                   k :: c (with k w/out further inner structure)
```

Pattern 2: Monadic composition of (`St h`) and `f`:

```
(>>=) :: (State st) a -> (a -> (State st) b) -> (State st) b
(St h) >>= f = St g
    where g :: st -> (st,b)
          g = "apply h, then apply f to h's result"
    wrt given maps h :: st -> (st,a)
                   f :: a  -> (State st) b
                   where values of type (State st) b look like:
                   St k :: (State st) b with k :: st -> (st,b)
    ensuring St g :: (State st) b as required.
```

This means

...if we understand sequential composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . h = g where g :: (a -> c)
                g = \x -> let y = h x   -- apply h
                              z = f y   -- then apply f to
                          in z          -- h's result
```

we understand monadic composition, too: Composing a monadic value (`St h`) capsulating a state transformer `h` and a state transformer producing function `f` yields eventually a value (`St g`) of another monadic type being the result the monadic composition of (`St h`) and `f`:

```
(>>=) :: (State st) a -> (a -> (State st) b) -> (State st) b
(St h) >>= f = St g
             where g ::  st -> ( st,b)
                   g = "apply h, then apply f to h's result"
             wrt given maps h and f...
```

Of course, the details of monadic composition are more complex than for sequential composition because the involved types are more complex...

Getting bind (>>=) done!

```
(>>=) :: (State st) a -> (a -> (State st) b) -> (State st) b
  (St h) >>= f = St g
    where g :: st -> (st,b)
          g = (\s -> let (s′,x)  = h s    -- Apply h
              :: st      St f′   = f x    -- then apply f to
                         (s′′,y) = f′ s′  -- (part of) h’s
                     in (s′′,y))          -- result giving f′
                      :: (st,b)           -- and f′ to the rest
                                          -- of h’s result
```

Note: The two functions

1) `h :: (st -> (st,a))` 2) `f :: a -> (State st) b`

involved in monadic composition for the state monad are applied one after the other and yield as intermediate result a third function

3) `f′ :: st -> (st,b)`

that, applied to another intermediate result, completes a fourth function

4) `g :: st -> (st,b)`

which, capsulated in state value `St g`, is the result of monad. compos.!

Constructing g in three steps (1)

Note: `g = \s -> let ... in (s'',y) :: st -> (st,b)` is constructed in 3 steps:

```
g :: st -> (st,b)
g = (\s -> let (s',x) = h s       -- 1) Apply h,
                St f' = f x       -- 2) then apply f to
               (s'',y) = f' s'    -- (part of) h's
           in (s'',y))            -- result giving f',
```

$$\underbrace{\texttt{s}}_{\texttt{:: st}} \qquad \underbrace{\texttt{(s'',y)}}_{\texttt{:: (st,b)}}$$

```
                                  -- 3) and then f' to the
                                  -- rest of h's result
                                  -- giving (s'',y).
```

1) State transformer `h` is applied to `s :: st` yielding a pair `(s',x) :: (st,a)` of an intermediate new state `s'` and an additional value `x`.
2) Applied to `x :: a`, `f` yields a monadic value `St f' :: (State st) b` capsulating a new state transformer `f' :: st -> (st,b)`.
3) `f'` is applied to the intermediate new state `s' :: st` yielding the pair `(s'',y) :: (st,b)` with final state `s''` and additional value `y` as result of the monadic composition of `(St h)` and `f` as required.

Constructing g in three steps (2)

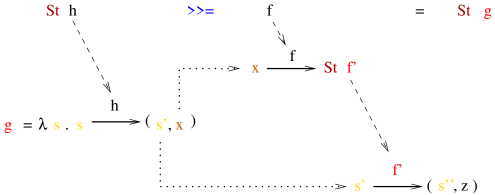
In summary, there are two intermediate results showing up in the course of constructing g:

1. a pair (s′,x) of an intermediate new state s′ and some value x,
2. an intermediate new state transformer function f′ capsulated in a (State st b) value (St f′)!

Mission accomplished: Bind (>>=) done!

```
(>>=) :: (State st) a -> (a -> (State st) b) -> (State st) b
  (St h) >>= f = St g
    where g :: st -> (st,b)
          g = (\s -> let (s',x)  = h s    -- 1) Apply h,
             :: st       St f'   = f x    -- 2) then apply f
                         (s'',y) = f' s'  -- to (part of) h's
                     in (s'',y))          -- result giving f',
                        :: (st,b)
                                          -- 3) and then f' to
                                          -- the rest of h's
                                          -- result giving (s'',y).
```

This effect of the bind operation can be visualized as follows:



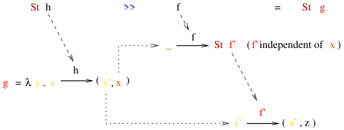
Getting the remaining State monad op's done!

Having defined bind (>>=), we are left with defining return, sequence (>>), and fail:

```
return :: a -> (State st) a
return x = St g
 where g :: st -> (st,a)
       g = \s -> (s,x)
```

For sequence (>>) and fail we'll go ahead with their default implementations of type constructor class Monad, i.e.:

```
(>>) :: (State st) a -> (State st) b -> (State st) b
(St h) >> f = (St h) >>= \_ -> f
```




```
fail :: String -> (State st) b
fail s = error s
```

Getting done with the State monad!

$$\texttt{instance Monad (State st) where}$$

$$\underbrace{\texttt{(St h)}}_{\texttt{:: st -> (st,a)}} \;\texttt{>>=}\; \underbrace{\texttt{f}}_{\texttt{:: a -> (State st) b}}$$

$$\texttt{= St (}\underbrace{\texttt{\textbackslash s}}_{\texttt{:: st}}\texttt{ -> let (s}'\texttt{,x) = h s}$$

$$\texttt{St f}' \texttt{ = f x}$$

$$\texttt{in } \underbrace{\texttt{f}'\texttt{ s}'}_{\texttt{:: (st,b)}}\texttt{)}$$

$$\texttt{return } \underbrace{\texttt{x}}_{\texttt{:: a}} \texttt{ = St (}\underbrace{\texttt{\textbackslash s}}_{\texttt{:: st}}\texttt{ -> }\underbrace{\texttt{(s,x)}}_{\texttt{:: (st,a)}}\texttt{)}$$

...with types:

```
(>>=) :: (State st) a -> (a -> (State st) b) -> (State st) b
return :: a -> (State st) a
(>>) :: (State st) a -> (State st) b -> (State st) b
fail :: String -> (State st) a
```

Or, more concisely, w/out type information:

```
instance Monad (State st) where
 (St h) >>= f = St (\s -> let (s',x) = h s
                              St f'  = f x
                          in f' s')

 return x = St (\s -> (s,x))
```

Once again, the State Monad in more Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
c >>= k = ... :: m b
return :: (Monad m) => a -> m a
return x = ... :: m a
```

The instance declaration for (State st) with added type information:

```
instance Monad (State st) where
      St h       >>=         f
:: (State st) a    :: a -> (State st) b
                 = St (\s -> let ... in f′ s′)   -- constructing
                      :: st       :: (st,b)      -- a proper state
                         :: st -> (st,b)         -- value using h
                         :: (State st) b         -- and f.
   return x  =  St (\s -> (s,x))    -- constructing a proper
          :: a    :: (State st) a   -- state value using x
:: (State st) a                     -- in the simplest way.
```

Proof Obligation: The Monad Laws

Lemma 12.4.6.1 (Soundness of the State Monad)

The `(State st)` instance of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

...`(State st)` is thus a proper instance of `Monad`, the so-called state monad.

State′: The Specialized State Monad

...specialized for a concrete state type CStT (‘Concrete State Type’) (e.g., Int, [String],...):

```
newtype State' a = St' (CStT -> (CStT,a))

instance Monad State' where
  St' m >>= f = St' (\cs -> let (cs',x) = m cs
                    :: CStT       St' f'  = f x
                              in f' cs')
                              ::] (CStT,b)

  return x     = St' (\cs -> (cs,x))
       :: a          :: CStT :: (CStT,a)
```

Note: State′ is a 1-ary type constructor whereas State is a 2-ary type constructor.

Proof Obligation: The Monad Laws (State′)

Lemma 12.4.6.2 (Soundness of Spec. State Monad)

The State′ instance of Monad satisfies the three monad laws ML1, ML2, and ML3.

...(State′) is thus a proper instance of Monad, the so-called specialized state monad.

Note: For State′ the types of the monad operations (>>=), return, and (>>) boil down to:

```
(>>=)  :: State' a -> (a -> State' b) -> State' b
return :: a -> State' a
(>>)   :: State' a -> State' b -> State' b
```

The State Monad Reconsided (1)

...sometimes also renaming helps getting things clear(er).

Think of `st_otw` as a type variable where the values of appropriate concrete types for `st_otw` describe or model the

- state of the world (st_otw).

The bind operation `(>>=)` of state monad `(State st_otw)` then allows us to transform current states of the world into new states of the world, i.e., to

- transform (the description of) the state of the world it is currently in into (the description of) the world it is in after the transformation, i.e., (the description of) the new state the world is in afterwards.

This suggests that state transformers are of the type:

```
state_transformer :: st_otw -> st_otw
```

...class `Monad` makes this a bit more complex as shown next.

The State Monad Reconsidered (2)

```
newtype (State st_otw) a = St (st_otw -> (st_otw,a))

instance Monad (State st_otw) where
 St h >>= f
  = St (\current_state ->
        let (intermediate_state,x) = h current_state
            St g = f x
            (new_state,z) = g intermediate_state
        in (new_state,z)
 return x = St (\current_state -> (new_state,x))
     where new_state = current_state
```

where

```
(>>=) :: (State st_otw) a -> (a -> (State st_otw) b) ->
                                      (State st_otw) b
return :: a -> (State st_otw) a
```

Finally

...recall (or note) that we find the same pattern when sequentially composing functions (note particularly the similarity of the definitions of the left-to-right sequencing operations (>>=) and (;)):

```
(g . f) = (f ; g) = \x -> let intermediate = f x
                              z = g intermediate
                          in z
```

Obviously:

```
(g . f) y =
(f ; g) y =
(\x -> let intermediate = f x; z = g intermediate in z) y
          = z
          = g (f y)
```

Chapter 12.4.7

The Input/Output Monad

The Input/Output Monad

```
instance Monad IO where    (Impl. intern. hidden)
 (>>=)  :: IO a -> (a -> IO b) -> IO b
 return :: a -> IO a
 (>>)   :: IO a -> IO b -> IO b
 fail   :: String -> IO a
```

Note:

- IO-values are so-called IO-commands (or commands).
- Commands have a procedural effect (i.e., reading or writing) and a functional effect (i.e., computing a value).
- (>>=): With p, q commands, p >>= q is a composed command that first executes p, thereby performing a read or write operation and yielding an a-value x as result; subsequently q is applied to x, thereby performing a read or write operation and yielding a b-value y as result.
- return: Lifts an a-value to an IO a-value w/out performing any input or output operation.

Proof Obligation: The Monad Laws

Lemma 12.4.7.1 (Soundness of I/O Monad)

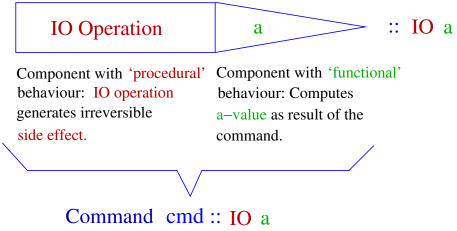
The `IO` instance of `Monad` satisfies the three monad laws ML1, ML2, and ML3.

...`IO` is thus a proper instance of `Monad`, the so-called input/output (I/O) monad.

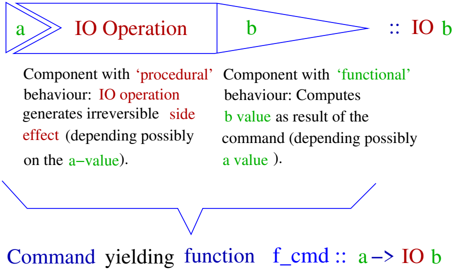
Note: The implementation of the input/output monad is internally hidden; it is thus the compiler writer who is in charge for proving Lemma 12.4.7.1.

Illustrating the Nature of Commands

Command `cmd :: IO a`

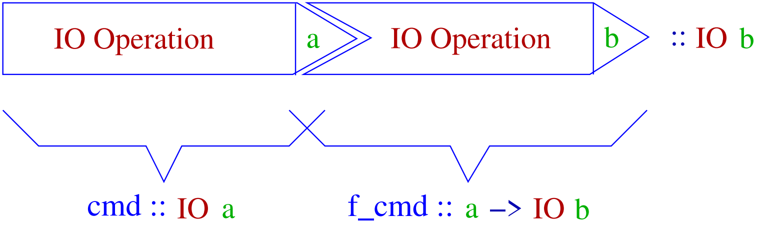


Command yielding function `f_cmd :: a -> IO b`



Illustrating

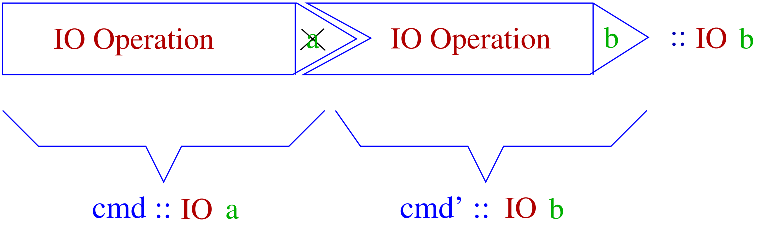
...the operational meaning of `(cmd >>= f_cmd)`:



`cmd >>= f_cmd` $\widehat{=}$ `cmd >>= \x -> f_cmd x`

Illustrating

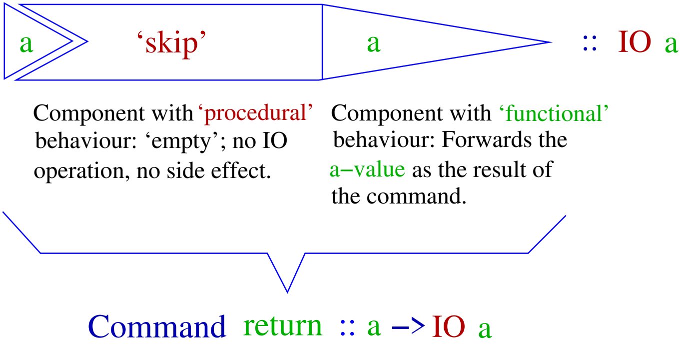
...the operational meaning of (`cmd >> cmd'`):



$$\texttt{cmd >> cmd'} \;\;\widehat{=}\;\; \texttt{cmd >> \textbackslash\_ -> cmd'}$$

Illustrating

...the operational meaning of `return`:



The Type

...of all read commands is

- (IO a) (for type instances a whose values can be read).

  The a-value into which the read value is transformed serves as the (formally required and actually wanted) result of read operations.

...of all write commands is

- (IO ()), where () is the singleton null tuple type with the single unique element ().

  () as (the one and only) value of the null tuple type () serves as the formally required result of write operations.

The I/O Monad viewed as a State Monad

...the input/output monad is similar in spirit to the state monad: It passes around the "state of the world!"

For a suitable type `World` whose values represent the

- states of the world

interactive programs (or IO-programs) can informally be considered functions of a type `IO` with:

- "`type IO = (World -> World)`"

In order to reflect that interactive programs do not only modify the state of the world but may also return a result, e.g., the `Int`-value of a sequence of characters that has been read from the keyboard and interpreted as an integer, this leads to changing the informal type of IO-programs from `IO` to `(IO a)`:

- "`type IO a = (World -> (World,a))`"

The Input/Output Monad (1)

...allows switching from a batch-like handling of input/output:

Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, p. 245.

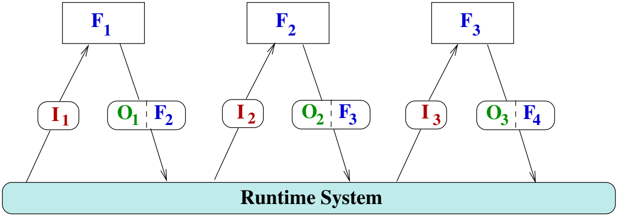
where

- all input data must be provided at the very beginning
- there is no interaction between a program and a user (i.e., once called there is no opportunity for the user to react on a program's response and behaviour)

to a...

The Input/Output Monad (2)

...truly interactive handling of input/output in terms of sequentially composed dialogue components, while preserving referential transparency as far as possible:



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, p. 253.

Note that input/output operations are a major source for side effects: read statements e.g. will yield different values for every call causing unavoidably the loss of referential transparency.

Examples: Simple IO Programs (1)

...a question/response interaction with a user:

```
ask :: String -> IO String
ask question = do putStrLn question
                  getLine

interAct :: IO ()
interAct =
    do name <- ask "May I ask your name?"
       putStrLine ("Welcome " ++ name ++ "!")
```

Examples: Simple IO Programs (2)

...input/output from and to files:

```
type FilePath = String  -- file names according
                        -- to the conventions of
                        -- the operating system

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath -> IO String
isEOF      :: FilePath -> IO Bool

interAct :: IO ()
interAct = do putStr "Please input a file name: "
              fname <- getLine
              contents <- readFile fname
              putStr contents
```

Examples: Simple IO Programs (3)

...the sequence of input/output commands with local declarations within a do-construct

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   let rev1 = reverse line1
                   let rev2 = reverse line2
                   putStrLn rev2
                   putStrLn rev1
```

is equivalent to the following one without:

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   putStrLn (reverse line2)
                   putStrLn (reverse line1)
```

Examples: Simple IO Programs (4)

...sequences of (canonic) monadic operations:

```
writeFile "testFile.txt" "Hello File System!"
  >> putStr "Hello World!" >> putStr "Oh, yeah."
```

can be replaced by their equivalent do-expressions:

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
   putStr "Oh, yeah."
```

Examples: Simple IO Programs (5)

...note the sometimes subtle differences in the representation of values of output and non-output types.

Output types:

```
Main>putStr ('a':('b':('c':[])))
        ->> abc :: IO ()
```

```
Main>putChar (head ['x','y','z'])
        ->> x :: IO ()
```

Non-output types:

```
Main>('a':('b':('c':[])))
        ->> "abc" :: [Char]
```

```
Main>head ['x','y','z']
        ->> 'x' :: Char
```

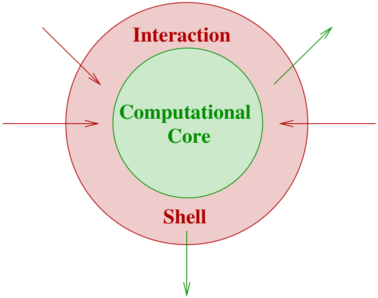
```
Main>print "abc"
        ->> "abc" :: IO ()
```

```
Main>print 'x'
        ->> 'a' :: IO ()
```

Monadic Input/Output in Haskell

...allows us to conceptually think of a Haskell program as being composed of a

- purely functional computational core
- procedural-like interaction shell.



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell.* Pearson, 2004, p. 89.

The Conceptual Separation

...of functions belonging to the

- computational core (pure functions)
- interaction shell (impure functions, i.e., performing input/output operations causing side effects).

is achieved by assigning different types to them:

- `Int`, `Real`, `String`,... vs. `IO` Int, `IO` Real, `IO` String,...

with the type constructor `IO` a pre-defined monad.

The monadic implementation of input/output allows us

- precisely specify the evaluation order of functions of the interaction shell (i.e., basic input/output primitives provided by Haskell) by using the monadic sequencing operations `(>>=)` and `(>>)`.

...see e.g. lecture notes of LVA 185.A03 Funktionale Programmierung for further details and examples.

Chapter 12.5

Monadic Programming

Monadic Programming

…we consider three examples for illustration:

1. Folding trees by adding the values of their numerical labels.
2. Numbering tree labels (and overwriting the original labels).
3. Renaming tree labels by the number of their occurrences.

The first two examples are handled

- without
- with

monads in order to oppose and illustrate the relative merits of the two programming styles.

Chapter 12.5.1

Folding Trees

The Setting

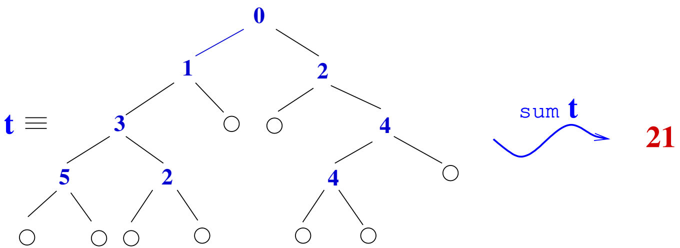
Given:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Objective:

- Write a function that computes the sum of the values of all labels of a tree of type `Tree Int`.

  Illustration:



For Comparison

...we consider three approaches:

1. w/out monads
2. w/ monads
3. w/ monads followed by unpacking the monadic result.

1st Approach: Straightforward w/out Monads

...using a recursive function:

```
sum :: Tree Int -> Int
sum Nil              = 0
sum (Node n t1 t2) = n + sum t1 + sum t2
```

Note:

- The evaluation order of the right-hand term of the (non-trivial) defining equation of `sTree` is not fixed; only data dependencies need to be respected.
- This leaves interpreter and compiler a degree of freedom in picking an evaluation order.
- This freedom can not be broken by a programmer by using a specific right-hand side term:

```
sum (Node n t1 t2) = n + sum t1 + sum t2
sum (Node n t1 t2) = sum t2 + n + sum t1
...
sum (Node n t1 t2) = sum t2 + sum t1 + n
```

2nd Approach: Using the Identity Monad

...using the identity monad Id:

```
sum' :: Tree Int -> Id Int
sum' Nil = return 0
sum' (Node n t1 t2) =
 do s2  <- sum' t2        -- Evaluating right subtree
    num <- return n       -- Bounding n :: Int to num
    s1  <- sum' t1        -- Evaluating left subtree
    return (s2+num+s1) -- Yielding Id (num+s1+s2) ::
                          -- Id Int as result
```

Note:

- The evaluation order of the defining 'equations' for s2, n, and s1 is explicitly fixed; there is no degree of freedom for the sequence in which values are bound to them.
- Changing their order allows the programmer to enforce a different evaluation order.
- Note, this does not apply to evaluating s2+num+s1.

Recall

...the definition of the identity monad `Id`:

```
newtype Id a = Id a

instance Monad Id where
 (Id x) >>= f = f x
 return       = Id
```

...and the overloading of `Id`:

- `Id`: 1-ary type constructor, i.e., if a is a type variable, then `Id a` denotes a type.
- `Id`: 1-ary data (or value) constructor, i.e., if `x :: a`, then `Id x` is a value of type `Id a`: `Id x :: Id a`.

Illustrating the Imperative Flavour of sum′

...unlike sum, sum′ enjoys an ‘imperative’ flavour quite similar to sequentially sequencing assignment statements of some imperative programming language:

Imperative	Monadic
`s2  := sumTree t2;`	`do s2  <- sumTree t2`
`s1  := sumTree t1;`	`   s1  <- sumTree t1`
`num := n;`	`   num <- return n`
`return (s2+s1+num);`	`   return (s2+s1+num)`

Note: Just for folding a tree, a monadic approach might be considered too ‘heavy’ and a foldable approach with tree an instance of class Foldable more lightweight. If, however, for some reason it is important that subtrees are folded in a particular order, this can be achieved by the monadic approach, however, not by the foldable one.

3rd Approach: Unpacking the Monadic Result

...to this end we introduce an extraction function unpacking a monadic value:

```
extract :: Id a -> a
extract (Id x) = x
```

This allows function sum″ yielding again an Int-value (instead of a monadic one):

```
sum'' :: Tree Int -> Int
sum'' = extract . sum'
```

Example:

```
t = (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil))

sum'' t ->> (extract . sum') t
        ->> extract (sum' t)
        ->> extract (Id 15)
        ->> 15
```

Chapter 12.5.2

Numbering Tree Labels

The Setting

Given:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Objective:

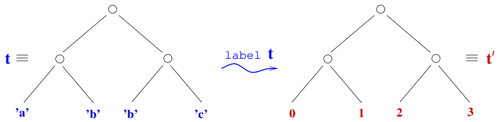
– Replace the labels of leafs by continuous natural numbers.

Illustration: The tree value `t :: Tree Char`:

```
t = Branch (Branch (Leaf 'a') (Leaf 'b'))
           (Branch (Leaf 'b') (Leaf 'c'))
```

shall be transformed into the tree value `t' :: Tree Int`:

```
t' = Branch (Branch (Leaf 0) (Leaf 1))
            (Branch (Leaf 2) (Leaf 3))
```



For Comparison

...we consider two approaches:

1. w/out monads
2. w/ monads

1st Approach: Straightforward w/out Monads

...using a pair of functions, one of which a recursive supporting function:

```
label :: Tree a -> Tree Int
label t = snd (lab t 0)

lab :: Tree a -> Int -> (Int, Tree Int)
lab (Leaf a) n = (n+1, Leaf n)
lab (Branch t1 t2) n
    = let (n1,t1') = lab t1 n
          (n2,t2') = lab t2 n1
      in (n2, Branch t1' t2')
```

Note: The solution is simple and straightforward but passing the counter value `n` through the incarnations of `lab` is tedious and intricate.

2nd Approach: Using the Spec. State Monad (1)

...using the pattern of the specialized state monad State′:

```
newtype Label a = Lab (Int -> (Int,a))

instance Monad Label where
 Lab lt >>= flt = Lab $ \n -> let (n',x)  = lt n
                                  Lab lt' = flt x
                              in lt' n'
 return x       = Lab (\n -> (n,x))
```

Note:

- The $-operator in the defining equation of `(>>=)` can be replaced by bracketing: `(\n -> let ... in lt' n')`.
- For the state monad `Label` the monad operations `(>>=)` and `return` have the types:

```
(>>=)  :: Label a -> (a -> Label b) -> Label b
return :: a -> Label a
```

2nd Approach: Using the Spec. State Monad (2)

...the renaming of labels is now achieved by using:

```
label' :: Tree a -> Tree Int
label' t = let Lab lt = lab' t
           in snd (lt 0)

lab' :: Tree a -> Label (Tree Int)
lab' (Leaf a) = do n <- get_label
                   return (Leaf n)
lab' (Branch t1 t2) = do t1' <- lab' t1
                         t2' <- lab' t2
                         return (Branch t1' t2')

get_label :: Label Int
get_label = Lab (\n -> (n+1,n))
```

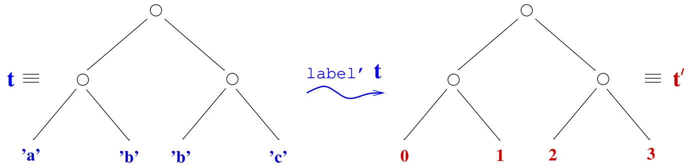
2nd Approach: Using the Spec. State Monad (3)

Example: Applying `label′` to tree value `t`:

```
t = Branch (Branch (Leaf ‘a‘) (Leaf ‘b‘))
           (Branch (Leaf ‘b‘) (Leaf ‘c‘))
```

...we get as desired:

```
label′ t ->> Branch (Branch (Leaf 0) (Leaf 1))
                    (Branch (Leaf 2) (Leaf 3))
         ≡   t′
```



Chapter 12.5.3

Renaming Tree Labels

The Setting

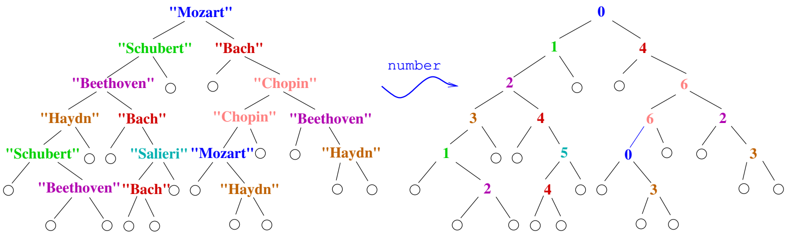
Given:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Objective:

– Rename labels of equal a-value by the same natural number.

Illustration:



Ultimate Goal

...a function `number` of type

```
number :: Eq a => Tree a -> Tree Int
```

solving this task using the state monad `State`.

Towards the Monadic Approach (1)

We start defining:

```
number_tree :: Eq a => Tree a -> State a (Tree Int)
number_tree Nil = return Nil
number_tree (Node x t1 t2) =
                 = do num <- number_node x
                      nt1 <- number_tree t1
                      nt2 <- number_tree t2
                      return (Node num nt1 nt2)
```

...post-poning the implementation of number_node.

Towards the Monadic Approach (2)

Additionally, we introduce a table type

```
type Table a = [a]
```

for storing pairs of the form

(<string>,<number of occurrences>)

In particular, the list (or table) value

```
[True,False]
```

encodes that `True` represents (or is associated with) `0` and `False` with `1`.

Mon. Approach: Using the State Monad (1)

...using the pattern of the state monad `State st`:

```
newtype State a b = St (Table a -> (Table a, b))

instance Monad (State a) where
  (St st) >>= f
     = St (\tab -> let (tab′,y)      = st tab
                       (St transf) = f y
                   in transf tab′)
   return x = St (\tab -> (tab, x))
```

Intuitively:

- Computing b-values: The (functional) result
- Updating tables: The side effect

...of the monadic operations.

Mon. Approach: Using the State Monad (2)

...providing the post-poned implementation of `number_node`:

```
number_node :: Eq a => a -> (State a) Int
number_node x = St (num_node x)

num_node :: Eq a => a -> (Table a -> (Table a, Int))
num_node x table
  | elem x table = (table,       lookup x table)
  | otherwise    = (table ++ [x], length table)
  -- num_node yields the position of x in the table:
  -- if x is stored in the table, using lookup; if
  -- not, after adding x to the table using length.

lookup :: Eq a => a -> Table a -> Int
lookup x table = ... -- Homework: Completing the
                     -- implementation of lookup.
```

Mon. Approach: Using the State Monad (3)

Putting the pieces together, number_tree is fully defined:

```
number_tree :: Eq a => Tree a -> State a (Tree Int)
number_tree Nil = return Nil
number_tree (Node x t1 t2)
                 = do num <- number_node x
                      nt1 <- number_tree t1
                      nt2 <- number_tree t2
                      return (Node num nt1 nt2)
```

Note, for every value `t :: Eq a => Tree a`, e.g., the tree of the illustrating example, we can conclude (functional and hence) type correctness:

```
number_tree t :: State a (Tree Int)
               ≡ (State a) (Tree Int)
               ≡ ((State a) (Tree Int))
```

Mon. Approach: Using the State Monad (4)

...introducing and using the extraction function:

```
extract :: State a b -> b
extract (St st) = snd (st [])
```

we get the implementation of the initially envisioned function number:

```
number :: Eq a => Tree a -> Tree Int
number = extract . number_tree
```

Chapter 12.6

Monad-Plusses

Chapter 12.6.1

The Type Constructor Class MonadPlus

The Type Constructor Class MonadPlus

...monads with a 'plus' operation and a 'zero' element, which is a unit for 'plus' and a zero for (>>=), can be instances of the type constructor class MonadPlus obeying the monad-plus laws:

Type Constructor Class MonadPlus

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Monad-Plus Laws

```
m >>= (\_ -> mzero) = mzero      (MPL1)
mzero >>= m         = mzero      (MPL2)
m `mplus` mzero     = m          (MPL3)
mzero `mplus` m     = m          (MPL4)
```

Note

...`MonadPlus` instances are monads and thus must satisfy in addition to the monad-plus laws also all monad laws.

Intuitively, the monad-plus laws require from (proper) monad-plus instances:

– `mzero` is left-zero and right-zero for `(>>=)`.
– `mzero` is left-unit and right-unit for `mplus`.

Programmer obligation:

– Programmers must prove that their instances of `MonadPlus` satisfy the monad and monad-plus laws.

Note: The `IO` monad can not be made an instance of `MonadPlus` because it is lacking an appropriate 'zero' element.

Chapter 12.6.2

The List Monad-Plus

The List Monad-Plus

...making the 1-ary type constructor [] an instance of MonadPlus:

```
instance MonadPlus [] where        -- note the over-
  mzero = []                       -- loading of Id
  mplus = (++)
```

Proof obligation: The Monad-Plus Laws

Lemma 12.6.2.1 (Soundness of List Monad-Plus)

The [] instance of MonadPlus satisfies all monad and monad-plus laws.

...[] is thus a proper instance of MonadPlus, the so-called list monad-plus.

Chapter 12.6.3

The Maybe Monad-Plus

The Maybe Monad-Plus

...making the 1-ary type constructor Maybe an instance of MonadPlus:

```
instance MonadPlus Maybe where
  mzero                 = Nothing
  Nothing `mplus` ys = ys
  xs `mplus` ys       = xs
```

Proof obligation: The Monad-Plus Laws

Lemma 12.6.3.1 (Soundness of Maybe Monad-Plus)

The Maybe instance of MonadPlus satisfies all monad and monad-plus laws.

...Maybe is thus a proper instance of MonadPlus, the so-called maybe monad-plus.

Chapter 12.7

Summary

Summary

Monads (i.e., instances of the type constructor class `Monad`) combine features of

- functors and functional composition/sequencing:
  `(>>=) :: m a -> (a -> m b) -> m b`
  $\texttt{c >>= k >>= k}' \texttt{ >>= k}'' \texttt{ >>= } \ldots$

Monads are thus well-suited for

- structuring and ordering the steps of a computation

because the monadic sequencing operations `(>>=)` and `(>>)`

- allow specifying the order of computations explicity.
- offer an adequately high abstraction by decoupling the data type forming a monad (instance) from the structure of computation.
- support equational reasoning, e.g., in terms of the monad laws.

Monads

...are often considered of being fanned by an aura of something

- mystic, wondrous that is difficult to grasp and lets monads appear the Holy Grail of functional programming (*'once I will have understood monads, I will have understood functional programming'*).

This (slightly odd) image of monads might be due to the origin and ties of the monad notion to (possibly often difficult considered) fields like

- philosophy, category theory, programming languages theory and semantics.

Recall

Monads in Leibniz' Philosophy:

Definition (Gottfried Wilhelm Leibniz, 1714)

[Monadology, Paragraph 1]: The monad we want to talk about here is nothing else as a simple substance (German: Substanz), which is contained in the composite matter (German: Zusammengesetztes); simple means as much as: to be without parts.

Monads in Category Theory (cf. Saunders Mac Lane, 1971):

Definition (Eugenio Moggi, 1989)

[LICS'89]: A monad over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : Id_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ are natural transformations and the following equations hold:

$$\begin{array}{rcll} \mu_{TA};\mu_A & = & T(\mu_a);\mu_A & \\ \eta_{TA};\mu_A & = & id_{TA} & = T(\eta_A);\mu_A \end{array}$$

…*"a monad is a monoid in the category of endofunctors."*

But Remember

...the monad notion in functional programming (in Haskell, too) lost its connection to the monad notion in philosophy and category theory (almost) completely, and hence, everything which might or might be considered a mystery or miracle.

Rather than introducing a mystery, monads and monadic sequencing in functional programming close a 'functional gap' between function application, sequential function composition, and functorial mapping.

On the Closing of a 'Functional Gap' (1)

...smashing the myth behind functional programming monads.

- Function application ('mapping over'):

  ```
  ($) :: (a -> b) -> a -> b
  g $ x = g x
  ```

  - Special case (m a for a, m b for b):

    ```
    ($) :: (m a -> m b) -> m a -> m b
    g $ x = g x
    ```

- Sequential function composition ('sequencing'):

  ```
  (.) :: (b -> c) -> (a -> b) -> (a -> c)
  (f . g) x = f (g x)
  ```

  - Special case (m a for a, m b for b, m c for c):

    ```
    (.) :: (m b -> m c) -> (m a -> m b) -> (m a -> m c)
    (f . g) x = f (g x)
    ```

...one implementation fits all types: Parametric polymorphism

On the Closing of a ‘Functional Gap’ (2)

- Functorial mapping (‘mapping over’):

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
fmap g c = ... ‘(unpack, map, pack)’

(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
(<*>) k c = ... ‘(unpack, unpack, map, pack)’
```

- (Monadic) mapping plus sequencing:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>=) c k = k ‘‘unpack c’’
                    ‘(unpack, map, repeat >>=)’
```

...type-specific instance implementations required for 1-ary type constructors: *Ad hoc* polymorphism

Commonalities of Functions at a Glimpse

...compare (same color means 'correspond to each other'):

```
(.) :: (b -> c) -> (a -> b ) -> (a -> c)
(f . g) x = f (g x)

(;) :: (a -> b ) -> (b -> c) -> (a -> c)
(f ; g) = g . f                                    -- pointfree

(>>;) :: a -> (a -> b) -> b
x >>; f = f x                         -- Non-monadic operations
```

```
(>>.) :: Monad m => (m b -> m c) -> (m a -> m b) -> (m a -> m c)
(>>.) = (.)                               -- Monadic operations

(>>=) :: Monad m => m a -> (a -> m b) -> m b
m >>= k = k 'unpack m'

(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >@> g = \x -> (f x) >>= g                        -- pointfree
```

Chapter 12.8

References, Further Reading

Chapter 12: Basic Reading (1)

- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 7, Monaden)
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.2, The Monad Class; Chapter 18.3, The MonadPlus Class; Chapter 18.4, State Monads)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 10.6, Class and Instance Declarations – Monadic Types)

Chapter 12: Basic Reading (2)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 13, A Fistful of Monads; Chapter 14, For a Few Monads More)
- Simon Peyton Jones, Philip Wadler. *Imperative Functional Programming*. In Conference Record of the 20 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 71-84, 1993.
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 18, Programming with monads)
- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.

Chapter 12: Selected Advanced Reading (1)

- A (Reasonably) Comprehensive List of Tutorials on Monads: `haskell.org/haskellwiki/Monad_tutorials`.
- John Launchbury, Simon Peyton Jones. *State in Haskell*. Lisp and Symbolic Computation 8(4):293-341, 1995.
- Martin Odersky. *Funktionale Programmierung*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 599-612, 2006. (Kapitel 5.3, Funktionale Komposition: Monaden, Beispiele für Monaden)

Chapter 12: Selected Advanced Reading (2)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 7, I/O – The I/O Monad; Chapter 14, Monads; Chapter 15, Programming with Monads; Chapter 16, Using Parsec – Applicative Functors for Parsing; Chapter 18, Monad Transformers; Chapter 19, Error Handling – Error Handling in Monads)
- Philip Wadler. *Monads for Functional Programming*. In Johan Jeuring, Erik Meijer (Eds.), *1st Int. Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 24-52, 1995.
- Philip Wadler. *How to Declare an Imperative*. ACM Computing Surveys 29(3):240-263, 1997.

Chapter 12: Selected Advanced Reading (3)

- Simon Peyton Jones. *Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell*. In Tony Hoare, Manfred Broy, Ralf Steinbruggen (Eds.), Engineering Theories of Software Construction, IOS Press, 47-96, 2001 (Presented at the 2000 Marktoberdorf Summer School).
- Wouter S. Swierstra, Thorsten Altenkirch. *Beauty in the Beast: A Functional Semantics for the Awkward Squad*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007), 25-36, 2007.

Chapter 12: Background Reading

- René Descartes. *Meditationes de prima philosophia*. 1641.
- Gottfried Wilhelm Leibniz. *Monadology* (Original in French). 90 Paragraphen, 1714.
- Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-V., 1971 (2nd edition, 1998).
- Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
- Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
- Thomas Petricek. *What We Talk about when We Talk about Monads*. The Art, Science, and Engineering of Programming 2(3), Article 12, 1-27, 2018.

Chapter 13

Arrows

Chapter 13.1

Motivation

Motivation

...monads do not always suffice.

The higher-order type constructor class `Arrow`

– complements the type class `Monad`

with a complementary mechanism for

– composing and sequencing functions

which support 2-ary type constructors and is useful e.g. for:

– electronic circuits modelling (this chapter)
– functional reactive programming (cf. Chapter 18).

Chapter 13.2

The Type Constructor Class Arrow

The Type Constructor Class Arrow

Arrows are instances of the type constructor class Arrows obeying the arrow laws:

```
class Arrow a where
 pure  :: (b -> c) -> a b c
    -- equivalently: pure :: ((->) b c) -> a b c
 (>>>) :: a b c -> a c d -> a b d
 first :: a b c -> a (b,d) (c,d)
```

Note:

- pure allows embedding of ordinary maps into the constructor class Arrow (the role of pure for maps is similar to the role of return in class Monad for values of type a).
- (>>>) serves the composition of computations.
- first has as an analogue on the level of ordinary functions: The function firstfun with
  `firstfun f = \(x,y) -> (f x, y)`

The Arrow Laws

...proper instances of Arrow must satisfy the following nine arrow laws:

Arrow Laws

```
pure id >>> f = f                          (ArrL1): identity
f >>> pure id = f                          (ArrL2): identity
(f >>> g) >>> h = f >>> (g >>> h)          (ArrL3): associa-
                                                    tivity

pure (g . f) = pure f >>> pure g           (ArrL4): functor
                                                    composition

first (pure f) = pure (f × id)             (ArrL5): extension
first (f >>> g) = first f >>> first g      (ArrL6): functor
first f >>> pure (id × g) = pure (id × g) >>> first f
                                           (ArrL7): exchange
first f >>> pure fst = pure fst >>> f      (ArrL8): unit
first (first f) >>> pure assoc = pure assoc >>> first f
                                           (ArrL9): association
```

Utility Functions for Arrows (1)

The product map ×:*)

```
(×) :: (a -> a′) -> (b -> b′) -> (a,b) -> (a′,b′)
(f × g) ~(a,b) = (f a, g b)
```

Regrouping arguments via assoc, unassoc, and swap:*)

```
assoc :: ((a,b),c) -> (a,(b,c))
assoc ~(~(x,y),z) = (x,(y,z))
unassoc :: (a,(b,c)) -> ((a,b),c)
unassoc ~(x,~(y,z)) = ((x,y),z)
swap :: (a,b) -> (b,a)
swap ~(x,y) = (y,x)
```

The dual analogue of first, map second:

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = pure swap >>> first f >>> pure swap
```

*) Refer to Chapter 2.5.1 for lazy patterns like ~(a,b).

Utility Functions for Arrows (2)

Derived operators for arrows:

```
(***) :: Arrow a => a b c -> a b' c' ->
                                  a (b,b') (c,c')
f *** g = first f >>> second g

(&&&) :: Arrow a => a b c -> a b c' -> a b (c,c')
f &&& g = pure (_-> (b,b)) >>> (f *** g)

idA :: Arrow a => a b b
idA = pure id
```

Chapter 13.3

The Map Arrow

The Map Arrow

...making the 2-ary type constructor (->) an instance of Arrow:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f × id
```

where

```
(×) :: (b -> c) -> (d -> e) -> (b,d) -> (c,e)
(f × g)~(bv,dv) = (f bv, g dv) :: (c,e)
```

Note: Defining `first f = \(b,d) -> (f b, d)` is equivalent.

Proof obligation: The arrow laws

Lemma 13.3.1 (Arrow Laws for (->))

The (->) instance of Arrows satisfies the 9 arrow laws.

...(->) is thus a proper instance of Arrow, the so-called map arrow.

The Map Arrow in More Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (->) an instance of Arrow means constructor a equals (->):

instance Arrow (->) where

$$\underbrace{\texttt{pure f}}_{\texttt{:: (->) b c}} = \underbrace{\texttt{f}}_{\texttt{:: (->) b c}}$$

$$\underbrace{\texttt{f}}_{\texttt{:: (->) b c}} \;\texttt{>>>}\; \underbrace{\texttt{g}}_{\texttt{:: (->) c d}} = \underbrace{\texttt{g . f}}_{\texttt{:: (->) b d}}$$

$$\underbrace{\texttt{first f}}_{\texttt{:: (->) b c}} = \underbrace{\texttt{f} \times \texttt{id}}_{\texttt{:: (->) (b,d) (c,d)}}$$

Recall: Defining `first` by `first f = \(b,d) -> (f b, d)` is equivalent.

Note

```
(>>>) :: Arrow a => a b c -> a c d -> a b d
```

...introduces composition for 2-ary type constructors.

This means, for the map instance of class `Arrow`:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f × id
```

arrow composition boils down to:

– ordinary functional composition, i.e.: `(>>>) = (.)`

Chapter 13.4

Application: Modelling Electronic Circuits

A Notion of Computation

The map add introduces a notion of computation:

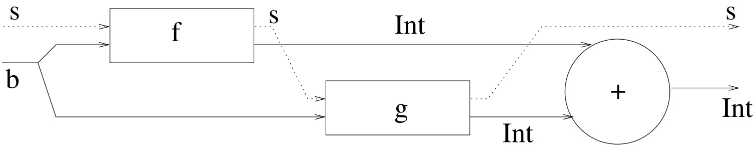
```
add :: (b -> Int) -> (b -> Int) -> (b -> Int)
add f g z = f z + g z
```

...which can be generalized in various ways, e.g., to

- state transformers
- non-determinism
- map transformers
- simple automata

for modelling electronic circuits.

Illustration:



Towards Modelling Electronic Circuits (1)

...generalizing add to state transformers:

```
type State s i o = (s,i) -> (s,o)

addST :: State s b Int -> State s b Int ->
                                     State s b Int
addST f g (s,z) = let (s′,x)  = f (s,z)
                      (s″,y) = g (s′,z)
                  in (s″,x+y)
```

Towards Modelling Electronic Circuits (2)

...generalizing add to non-determinism:

```
type NonDet i o = i -> [o]

addND :: NonDet b Int -> NonDet b Int ->
                                      NonDet b Int
addND f g z = [ x+y | x <- f z, y <- g z ]
```

Towards Modelling Electronic Circuits (3)

...generalizing add to map transformers:

```
type MapTrans s i o = (s -> i) -> (s -> o)

addMT :: MapTrans s b Int -> MapTrans s b Int ->
                                         MapTrans s b Int
addMT f g m z = f m z + g m z
```

Towards Modelling Electronic Circuits (4)

...generalizing add to simple automata:

```
newtype Auto i o = A (i -> (o, Auto i o))

addAuto :: Auto b Int -> Auto b Int -> Auto b Int
addAuto (A f) (A g)
   = A (\z -> let (x,f') = f z
                  (y,g') = g z
              in (x+y), addAuto f' g'))
```

Putting all this together

...allows us

- modelling of synchronous circuits (with feedback loops).

Note:

- The preceding examples have in common that there is a type $A \rightsquigarrow B$ of computations, where inputs of type A are transformed into outputs of type B.
- The type class `Arrow` yields a sufficiently general interface to describe these commonalities uniformly and to encapsulate them in a class.

Returning to the Application

...we are now going to make the previously introduced types instances of the type constructor class `Arrow`. To this end, we reintroduce them as new types (using `newtype`):

```
newtype State s i o = ST ((s,i) -> (s,o))

newtype NonDet i o  = ND (i -> [o])

newtype MapTrans s i o = MT ((s -> i) -> (s -> o))

newtype Auto i o = A (i -> (o, Auto i o))
```

The State Transformer Arrow

...making (State s) an instance of Arrow:

```
newtype State s i o = ST ((s,i) -> (s,o))

instance Arrow (State s) where
 pure f         = ST (id × f)
 ST f >>> ST g = ST (g . f)
 first (ST f)  = ST (assoc . (f × id) . unassoc)
```

The State Transformer Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (State s) an instance of Arrow means type constructor variable a is set to (State s):

```
newtype State s i o = ST ((s,i) -> (s,o))

instance Arrow (State s) where
  pure f           = ST (id × f)
  :: (->) b c        :: (State s) b c
      ST f          >>>      ST g        =    ST (g . f)
:: (State s) b c   :: (State s) c d   :: (State s) b d
  first (ST f)          =  ST (assoc . (f × id) . unassoc)
   :: (State s) b c            :: (State s) (b,d) (c,d)
```

The Non-Determinism Arrow

...making NonDet an instance of Arrow:

```
newtype NonDet i o  = ND (i -> [o])

instance Arrow NonDet where
 pure f          = ND (\b -> [f b])
 ND f >>> ND g = ND (\b -> [d | c <- f b, d <- g c])
 first (ND f)  = ND (\(b,d) -> [(c,d) | c <- f b])
```

The Non-Determinism Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making `NonDet` an instance of `Arrow` means type constructor variable `a` is set to `NonDet`:

```
NonDet i o  = ND (i -> [o])
```

```
instance Arrow NonDet where
  pure f          = ND (\b -> [f b])
  :: (->) b c        :: NonDet b c
     ND f     >>>    ND g     = ND (\b -> [d | c <- f b, d <- g c])
:: NonDet b c   :: NonDet c d                :: NonDet b d
  first (ND f)       =    ND (\(b,d) -> [(c,d) | c <- f b])
     :: NonDet b c                   :: NonDet (b,d) (c,d)
```

The Map Transformer Arrow

...making `(MapTrans s)` an instance of `Arrow`:

```
newtype MapTrans s i o = MT ((s -> i) -> (s -> o))

instance Arrow (MapTrans s) where
 pure f          = MT (f .)
 MT f >>> MT g = MT (g . f)
 first (MT f)   = MT (zipMap . (f x id) . unzipMap)
```

where

```
zipMap     :: (s -> a, s -> b) -> (s -> (a,b))
zipMap h s = (fst h s, snd h s)

unzipMap   :: (s -> (a,b)) -> (s -> a, s -> b)
unzipMap h = (fst . h, snd . h)
```

The Map Transformer Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (MapTrans s) an instance of Arrow means type constructor variable a is set to (MapTrans s):

```
MapTrans s i o = MT ((s -> i) -> (s -> o))
```

```
instance Arrow (MapTrans s) where
  pure f          =    MT (f .)
  :: (->) b c     :: (MapTrans s) b c
        MT f        >>>          MT g         =      MT (g . f)
:: (MapTrans s) b c   :: (MapTrans s) c d   :: (MapTrans s) b d
  first (MT f)          =     MT (zipMap . (f x id) . unzipMap)
 :: (MapTrans s) b c              :: (MapTrans s) (b,d) (c,d)
```

The Automata Arrow

...making `Auto` an instance of `Arrow`:

```
newtype Auto i o = A (i -> (o, Auto i o))

instance Arrow Auto where
 pure f      = A (\b -> (f b, pure f))
 A f >>> A g = A (\b -> let (c,f') = f b
                            (d,g') = g c
                        in (d, f' >>> g'))
 first (A f) = A (\(b,d) -> let (c,f') = f b
                            in ((c,d),first f'))
```

The Automata Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making Auto an instance of Arrow means type constructor variable a is set to Auto:

```
Auto i o = A (i -> (o, Auto i o))
instance Arrow Auto where
  pure f       = A (\b -> (f b, pure f)
  :: (->) b c          :: Auto b c
    A f    >>>    A g     = A (\b -> let (c,f') = f b
                                          (d,g') = g c
                                      in (d, f' >>> g')))
:: Auto b c  :: Auto c d                :: Auto b d
  first (A f)     =    A (\(b,d) -> let (c,f') = f b
                                      in ((c,d),first f'))
     :: Auto b c               :: Auto (b,d) (c,d)
```

Proof Obligation: The Arrow Laws

Lemma 13.4.1 (Soundness: Arrow Laws)

The state transformer, non-determinism, map transformer, and automata instances of `Arrow` satisfy the arrow laws and are thus proper arrows.

Last but not least, it is worth noting

....that each of the considered variants of add results as a specialization of general combinator addA with the corresponding arrow-type:

```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> pure (uncurry (+))
```

Chapter 13.5

An Update on the Haskell Type Class Hierarchy

An Update on the Haskell Type Class Hierarchy

...Haskell is a research vehicle and, hence, a moving target:

Haskell'98

Functor
fmap

↓

Applicative
pure
(<*>)

Monad
(>>=)
(>>)
return
fail

↓

MonadPlus
mZero
mPlus

Arrow
pure
(>>>)
first

Haskell'98 Onwards

Functor
fmap
(<$) :: f a –> f b –> f a
(<$) = fmap . const

↓

Applicative
pure
(<*>)
(*>) :: f a –> f b –> f b
a1 *> a2 = (id <$ a1) <*> a2
(<*) :: f a –> f b –> f a
(<*) = liftA2 const

↓

Monad
(>>=)
(>>)
return
fail

↓

MonadPlus
mZero
mPlus

Category
id :: cat a a
(.) :: cat b c –> cat a b –> cat a c

↓

Arrow
arr :: (b –> c) –> (b ‘arr‘ c)
first :: (b ‘arr‘ c) –> ((b,d) ‘arr‘ ((c,d))
second :: (b ‘arr‘ c) –> ((d,b) ‘arr‘ (d,c))
(***) :: (b ‘arr‘ c) –> (b’ ‘arr‘ c’) –> ((b,b’) ‘arr‘ (c,c’))
(&&&) :: (b ‘arr‘ c) –> (b ‘arr‘ c’) –> (c,c’))

where ‘arr‘ is a two–ary type variable

...for more information, check out:

https://wiki.haskell.org/Typeclassopedia

Chapter 13.6

Summary

Summing up

- Functions and programs often contain components that are 'function-like' 'w/out being just functions.'
- `Arrows` define a common interface for coping w/ the "notion of computation" of such function-like components.
- Monads are a special case of arrows.
- Like monads, arrows allow to meaningfully structure the computation process of programs.
- Arrow combinators operate on 'computations', not on values. They are point-free in distinction to the 'common case' of functional programming.
- Analoguous to the monadic case a `do`-like notational variant makes programming with arrow operations often easier and more suggestive (cf. literature hint at the end of the chapter), whereas the pointfree variant is more useful and advantageous for proof-theoretic reasoning.

Chapter 13.7

References, Further Reading

Chapter 13: Basic Reading

- John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37:67-111, 2000.
- Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001), 229-240, 2001.
- Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 201-222, 2003.

Chapter 13: Selected Advanced Reading

Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.

Concluding Note

...for additional information and details refer to

- full course notes

available in TUWEL and at the homepage of the course at:

http://www.complang.tuwien.ac.at/knoop/ffp185A05_ss2021.html

Assignment for Thursday, 22 April 2021

...independent study of Part IV, Chapters 12 and 13 and of Central and Control Questions IV for self-assessment and as a basis of the flipped classroom session on 04/22/2021:

Lecture, Flipped Classroom	Topic Lecture	Topic Flip. Classr.
Thu, 03/04/2021, 4.15-6.00 pm	P. I, Ch. 1 P. II, Ch. 2	n.a. / Prel. Mtg.
Thu, 03/11/2021, 4.15-6.00 pm	P. IV, Ch. 7, 8 P. II, Ch. 3	P. I, Ch. 1 P. II, Ch. 2
Thu, 03/25/2021, 4.15-6.00 pm	P. II, Ch. 4 P. IV, Ch. 9–11, 14	P. IV, Ch. 7, 8 P. II, Ch. 3
Thu, 04/15/2021, 4.15-6.00 pm	**P. IV, Ch. 12, 13**	**P. II, Ch. 4** **P. IV, Ch. 9–11, 14**
Thu, 04/22/2021, 4.15-6.00 pm	**P. III, Ch. 5, 6**	**P. IV, Ch. 12, 13**
Thu, 04/29/2021, 4.15-6.00 pm	P. V, Ch. 15, 16	P. III, Ch. 5, 6
Thu, 05/20/2021, 4.15-6.00 pm	P. V, Ch. 17, 18 P. VI, Ch. 19, 20	P. V, Ch. 15, 16