1 Declarative Programming

Programming languages are divided into different paradigms. Programs written in traditional languages like Pascal or C are *imperative* programs that contain instructions to mutate state. Variables in such languages point to memory locations and programmers can modify the contents of variables using assignments. An imperative program contains commands that describe *how* to solve a particular class of problems by describing in detail the steps that are necessary to find a solution.

By contrast, *declarative* programs describe a particular class of problems itself. The task to find a solution is left to the language implementation. Declarative programmers are equipped with tools that allow them to abstract from details of the implementation and concentrate on details of the problem.

Hiding implementation details can be considered a handicap for programmers because access to low-level details provides a high degree of flexibility. However, a lot of flexibility implies a lot of potential for errors, and, more importantly, less potential for abstraction. For example, we can write more flexible programs using assembly language than using C. Yet, writing large software products solely in assembly language is usually considered impractical. Programming languages like Pascal or C limit the flexibility of programmers, e.g., by prescribing specific control structures for loops and conditional branches. This limitation increases the potential of abstraction. Structured programs are easier to read and write and, hence, large programs are easier to maintain if they are written in a structured way. Declarative programming is another step in this direction.[1]

The remainder of this chapter describes those features of declarative programming that are preliminary for the developments in this book, tools it provides for programmers to structure their code, and concepts that allow writing programs at a higher level of abstraction. We start in Section 1.1 with important concepts found in *functional* programming languages, viz., polymorphic typing of higher-order functions, demand-driven evaluation, and type-based overloading. Section **??** describes essential features of *logic* programming, viz., non-determinism, unknown values and built-in search

[1] Other steps towards a higher level of abstraction have been *modularization* and *object orientation* which we do not discuss here.

and the interaction of these features with those described before. Finally, we show how so called *constraint* programming significantly improves the problem solving capabilities for specific problem domains in Section **??**.

1.1 Functional programming

While running an imperative program means to *execute commands*, running a functional program means to *evaluate expressions*.

Functions in a functional program are functions in a mathematical sense: the result of a function call depends only on the values of the arguments. Functions in imperative programming languages may have access to variables other than their arguments and the result of such a "function" may also depend on those variables. Moreover, the values of such variables may be changed after the function call, thus, the meaning of a function call is not solely determined by the result it returns. Because of such side effects, the meaning of an imperative program may be different depending on the order in which function calls are executed.

An important aspect of functional programs is that they do not have side effects and, hence, the result of evaluating an expression is determined only by the parts of the expression – not by evaluation order. As a consequence, functional programs can be evaluated with different evaluation strategies, e.g., demand-driven evaluation. We discuss how demand-driven, so called lazy evaluation can increase the potential for abstraction in Subsection 1.1.2.

Beforehand, we discuss another concept found in functional languages that can increase the potential for abstraction: type polymorphism. It provides a mechanism for code reuse that is especially powerful in combination with higher-order functions: in a functional program functions can be arguments and results of other functions and can be manipulated just like data. We discuss these concepts in detail in Subsection 1.1.1.

Polymorphic typing can be combined with class-based overloading to define similar operations on different types. Overloading of type constructors rather than types is another powerful means for abstraction as we discuss in Subsection 1.1.3.

We can write purely functional programs in an imperative programming language by simply avoiding the use of side effects. The aspects sketched above, however, cannot be transferred as easily to imperative programming languages. In the remainder of this section we discuss each of these aspects in detail, focusing on the programmers potential to increase the level of abstraction.

1.1.1 Type polymorphism and higher-order functions

In Chapter **??** we have seen the definition of a function *size* that computes the size of a string. In Haskell strings are represented as lists of characters and we could define similar functions for computing the length of a list of numbers or the length of a list of Boolean values. The definition of such length functions is independent of the type of list elements. Instead of repeating the same definition for different types we can define the function *length* once with a type that leaves the type of list elements unspecified:

$$
\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] \quad\ = 0 \\
&\mathit{length}\ (\_ : l) = 1 + \mathit{length}\ l
\end{aligned}
$$

The type a used as argument to the list type constructor $[\,]$ represents an arbitrary type. There are infinitely many types for lists that we can pass to length, e.g., $[\mathit{Int}]$, String, $[[\mathit{Bool}]]$ to name a few.

Type polymorphism allows us to use type variables that represent arbitrary types, which helps to make defined functions more generally applicable.

Type polymorphism is especially useful in combination with another feature of functional programming languages: higher-order functions. Functions in a functional program can not only map data to data but may also take functions as arguments or return them as result. Probably the simplest example of a higher-order function is the infix operator \$ for function application:

$$
\begin{aligned}
&(\$) :: (a \to b) \to a \to b \\
&f\ \$\ x = f\ x
\end{aligned}
$$

At first sight, this operator seems dispensable, because we can always write $f\ x$ instead of $f\ \$\ x$. However, it is often useful to avoid parenthesis because we can write $f\ \$\ g\ \$\ h\ x$ instead of $f\ (g\ (h\ x))$. Another useful operator is function composition:

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c) \\
&f \circ g = \lambda x \to f\ (g\ x)
\end{aligned}
$$

This definition uses a *lambda abstraction* that denotes an anonymous function. The operator for function composition is a function that takes two functions as arguments and yields a function as result. Lambda abstractions have the form $\lambda x \to e$ where x is a variable and e is an arbitrary expression. The variable x is the argument and the expression e is the body of the anonymous function. The body may itself be a function and the notation $\lambda x\ y\ z \to e$

is short hand for $\lambda x \to \lambda y \to \lambda z \to e$. While the first of these lambda abstractions looks like a function with three arguments, the second looks like a function that yields a function that yields a function. In Haskell, there is no difference between the two. A function that takes many arguments *is* a function that takes one argument and yields a function that takes the remaining arguments. Representing functions like this is called *currying*.[2]

There are a number of predefined higher-order functions for list processing. In order to get a feeling for the abstraction facilities they provide, we discuss a few of them here.

The predefined function *map* applies a given function to every element of a given list:

$$\begin{aligned} &map :: (a \to b) \to [a] \to [b] \\ &map\ f\ [\,] \quad\ = [\,] \\ &map\ f\ (x : xs) = f\ x : map\ f\ xs \end{aligned}$$

If the given list is empty, then the result is also the empty list. If it contains at least the element x in front of an arbitrary list xs of remaining elements, then the result of calling *map* is a non-empty list where the first element is computed using the given function f and the remaining elements are processed recursively. The type signature of *map* specifies that

- the argument type of the given function and the element type of the given list and
- the result type of the given function and the element type of the result list

must be equal. For example, *map length* `["Haskell","Curry"]` is a valid application of *map* because the a in the type signature of *map* can be instantiated with *String* which is defined as $[Char]$ and matches the argument type $[a]$ of *length*. The type b is instantiated with *Int* and, therefore, the returned list has the type $[Int]$. The application *map length* $[7,5]$ would be rejected by the type checker because the argument type $[a]$ of *length* does not match the type *Int* of the elements of the given list.

The type signature is a partial documentation for the function *map* because we get an idea of what *map* does whithout looking at its implementation. If we do not provide the type signature, then *type inference* deduces it automatically from the implementation.

Another predefined function on lists is *dropWhile* that takes a predicate, i.e., a function with result type *Bool*, and a list and drops elements from the list as long as they satisfy the given predicate.

[2] The term *currying* is named after the american mathematician and logician *Haskell B. Curry*.

$$
\begin{array}{l}
\mathit{dropWhile} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
\mathit{dropWhile}\ p\ [\,] \quad\ = [\,] \\
\mathit{dropWhile}\ p\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ \mathit{dropWhile}\ p\ xs\ \textbf{else}\ x : xs
\end{array}
$$

The result of *dropWhile* is the longest suffix of the given list that is either empty or starts with an element that does not satisfy the given predicate. We can instantiate the type variable a in the signature of *dropWhile* with many different types. For example, the function *dropWhile isSpace* uses a predefined function $\mathit{isSpace} :: \mathit{Char} \to \mathit{Bool}$ to remove preceding spaces from a string, *dropWhile* (<10) removes a prefix of numbers that are less than 10 from a given list, and *dropWhile* $((<10) \circ \mathit{length})$ drops short lists from a given list of lists, e.g., a list of strings. Both functions are defined as so called *partial application* of the function *dropWhile* to a single argument – an interesting programming style made possible by currying.

Polymorphic higher-order functions allow to implement recurring idioms independently of concrete types and to reuse such an implementation on many different concrete types.

1.1.2 Lazy evaluation

With lazy evaluation arguments of functions are only computed as much as necessary to compute the result of a function call. Parts of the arguments that are not needed to compute a result are not demanded and may contain divergent and/or expensive computations. For example, we can compute the length of a list without demanding the list elements. In a programming language with lazy evaluation like Haskell we can compute the result of the following call to the *length* function:

$$\mathit{length}\ [\bot, \mathit{fibonacci}\ 100]$$

Neither the diverging computation $\bot$ nor the possibly expensive computation *fibonacci* 100 are evaluated to compute the result 2.

This example demonstrates that lazy evaluation can be faster than eager evaluation because unnecessary computations are skipped. Lazy computations may also use less memory when different functions are composed sequentially:

```
do contents ← readFile "in.txt"
   writeFile "out.txt" ∘ concat ∘ map addSpace $ contents
  where addSpace c | c ≡ '.'  = ". "
                   | otherwise = [c]
```

This program reads the contents of a file `in.txt`, adds an additional space character after each period, and writes the result to the file `out.txt`. The function $concat :: [[a]] \to [a]$ concatenates a given list of lists into a single list. In an eager language, the functions *map addSpace* and *concat* would both evaluate their arguments completely before returning any result. With lazy evaluation, these functions produce parts of their output from partially known input. As a consequence, the above program runs in constant space and can be applied to gigabytes of input. It does not store the complete file `in.txt` in memory at any time.

In a lazy language, we can build complex functions from simple parts that communicate via intermediate data structures without sacrificing memory efficiency. The simple parts may be reused to form other combinations which increases the modularity of our code.

Infinite data structures

With lazy evaluation we can not only handle large data efficiently, we can even handle unbounded, i.e., potentially infinite data. For example, we can compute an approximation of the square root of a number x as follows:

$$
\begin{aligned}
&sqrt :: Float \to Float \\
&sqrt\ x = head \circ dropWhile\ inaccurate \circ iterate\ next\ \$\ x \\
&\quad \textbf{where}\ inaccurate\ y = abs\ (x - y * y) > 0.00001 \\
&\qquad\quad next\ y \qquad\ = (y + x\ /\ y)\ /\ 2
\end{aligned}
$$

With lazy evaluation we can split the task of generating an accurate approximation into two sub tasks:

1. generating an unbounded number of increasingly accurate approximations using Newton's formula and

2. selecting a sufficiently accurate one.

Approximations that are more accurate than the one we select are not computed by the function *sqrt*. In this example we use the function *iterate* to generate approximations and *dropWhile* to dismiss inaccurate ones. If we decide to use a different criterion for selecting an appropriate approximation, e.g., the difference of subsequent approximations, then we only need to change the part that selects an approximation. The part of the algorithm that computes them can be reused without change. Again, lazy evaluation promotes modularity and code reuse.

In order to see another aspect of lazy evaluation we take a closer look at the definition of the function *iterate*:

$$iterate :: (a \to a) \to a \to [a]$$
$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$

Conceptually, the call $iterate\ f\ x$ yields the infinite list

$$[x, f\ x, f\ (f\ x), f\ (f\ (f\ x)), \ldots]$$

The elements of this list are only computed if they are demanded by the surrounding computation because lazy evaluation is *non-strict*. Although the argument x is duplicated in the right-hand side of *iterate* it is evaluated at most once because lazy evaluation is *sharing* the values that are bound to variables once they are computed. If we call $sqrt\ (fibonacci\ 100)$ then the call *fibonacci* 100 is only evaluated once, although it is duplicated by the definition of *iterate*.

Sharing of sub computations ensures that lazy evaluation does not perform more steps than a corresponding eager evaluation because computations bound to duplicated variables are performed only once even if they are demanded after they are duplicated.

1.1.3 Class-based overloading

Using type polymorphism as described in Subsection 1.1.1 we can define functions that can be applied to values of many different types. This is often useful but sometimes insufficient. Polymorphic functions are agnostic about those values that are represented by type variables in the type signature of the function. For example, the *length* function behaves identically for every instantiation for the element type of the input list. It cannot treat specific element types different from others.

While this is a valuable information about the *length* function, we sometimes want to define a function that works for different types but can still take different instantiations of the polymorphic arguments into account. For example, it would be useful to have an equality test that works for many types. However, the type

$$(\equiv) :: a \to a \to Bool$$

would be a too general type for an equality predicate $\equiv$. It requires that we can compare arbitrary types for equality, including functional types which might be difficult or undecidable.

Class-based overloading provides a mechanism to give functions like $\equiv$ a reasonable type. We can define a *type class* that represents all types that support an equality predicate as follows:

$$\begin{array}{l}\textbf{class } \mathit{Eq}\ a \textbf{ where}\\ \quad (\equiv) :: a \to a \to \mathit{Bool}\end{array}$$

This definition defines a type class *Eq* that can be seen as a predicate on types in the sense that the *class constraint* $\mathit{Eq}\ a$ implies that the type a supports the equality predicate $\equiv$. After the above declaration, the function $\equiv$ has the following type:

$$(\equiv) :: \mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}$$

and we can define other functions based on this predicate that inherit the class constraint:

$$\begin{array}{l}(\not\equiv) :: \mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}\\ x \not\equiv y = \neg\ (x \equiv y)\\ \mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool}\\ x \in [\,] \quad\ = \mathit{False}\\ x \in (y : ys) = x \equiv y \vee x \in ys\end{array}$$

Here, the notation $x \in xs$ is syntactic sugar for $\mathit{elem}\ x\ xs$, $\neg$ denotes negation and $\vee$ disjunction on Boolean values.

In order to provide implementations of an equality check for specific types we can *instantiate* the *Eq* class for them. For example, an *Eq* instance for Booleans can be defined as follows.

$$\begin{array}{l}\textbf{instance } \mathit{Eq}\ \mathit{Bool} \textbf{ where}\\ \quad \mathit{False} \equiv \mathit{False} = \mathit{True}\\ \quad \mathit{True}\ \equiv \mathit{True}\ = \mathit{True}\\ \quad \_\ \quad \equiv \_\ \quad = \mathit{False}\end{array}$$

Even polymorphic types can be given an *Eq* instance, if appropriate instances are available for the polymorphic components. For example, lists can be compared if their elements can.

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

Note the class constraint $\mathit{Eq}\ a$ in the instance declaration for $\mathit{Eq}\ [a]$. The first occurrence of $\equiv$ in the second rule of the definition of $\equiv$ for lists is the equality predicate for values of type a while the second occurrence is a recursive call to the equality predicate for lists.

Although programmers are free to provide whatever instance declarations they choose, type-class instances are often expected to satisfy certain laws. For example, every definition of $\equiv$ should be an equivalence relation—reflexive, symmetric and transitive—to aid reasoning about programs that use $\equiv$. More specifically, the following properties are usually associated with an equality predicate.

$$x \equiv x$$
$$x \equiv y \Rightarrow y \equiv x$$
$$x \equiv y \wedge y \equiv z \Rightarrow x \equiv z$$

Defining an *Eq* instance where $\equiv$ is no equivalence relation can result in highly unintuitive program behaviour. For example, the *elem* function defined above relies on reflexivity of $\equiv$. Using *elem* with a non-reflexive *Eq* instance is very likely to be confusing. The inclined reader may check that the definition of $\equiv$ for Booleans given above is an equivalence relation and that the *Eq* instance for lists also satisfies the corresponding laws if the instance for the list elements does.

Class-based overloading provides a mechanism to implement functions that can operate on different types differently. This allows to implement functions like *elem* that are not fully polymorphic but can still be applied to values of many different types. This increases the possibility of code reuse because functions with similar (but not identical) behaviour on different types can be implemented once and reused for every suitable type instead of being implemented again for every different type.

Overloading type constructors

An interesting variation on the ideas discussed in this section are so called *type constructor classes*. In Haskell, polymorphic type variables can not only abstract from types but also from type constructors. In combination with class-based overloading, this provides a powerful mechanism for abstraction.

Reconsider the function $map :: (a \to b) \to [a] \to [b]$ defined in Subsection 1.1.1 which takes a polymorphic function and applies it to every element of a given list. Such functionality is not only useful for lists. A similar operation can be implemented for other data types too. In order to abstract from the data type whose elements are modified, we can use a type variable to represent the corresponding type constructor.

In Haskell, types that support a map operation are called *functors*. The corresponding type class abstracts over the type constructor of such types and defines an operation *fmap* that is a generalised version of the *map* function for lists.

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$

Like the *Eq* class, the type class *Functor* has a set of associated laws that are usually expected to hold for definitions of *fmap*:

$$\begin{aligned} &\mathit{fmap}\ \mathit{id} &&\equiv \mathit{id} \\ &\mathit{fmap}\ (f \circ g) &&\equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g \end{aligned}$$

Let us check whether the following *Functor* instance for lists satisfies these laws.

$$\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}$$
$$\quad \mathit{fmap} = \mathit{map}$$

We can prove the first law by induction over the list structure. The base case considers the empty list:

$$\begin{aligned} & \mathit{map}\ \mathit{id}\ [\,] \\ \equiv\ & \quad \{\ \text{definition of } \mathit{map}\ \} \\ & [\,] \\ \equiv\ & \quad \{\ \text{definition of } \mathit{id}\ \} \\ & \mathit{id}\ [\,] \end{aligned}$$

The induction step deals with an arbitrary non-empty list:

$$\begin{aligned} & \mathit{map}\ \mathit{id}\ (x : xs) \\ \equiv\ & \quad \{\ \text{definition of } \mathit{map}\ \} \\ & \mathit{id}\ x : \mathit{map}\ \mathit{id}\ xs \\ \equiv\ & \quad \{\ \text{definition of } \mathit{id}\ \} \\ & x : \mathit{map}\ \mathit{id}\ xs \\ \equiv\ & \quad \{\ \text{induction hypothesis}\ \} \\ & x : \mathit{id}\ xs \\ \equiv\ & \quad \{\ \text{definition of } \mathit{id} \text{ (twice)}\ \} \\ & \mathit{id}\ (x : xs) \end{aligned}$$

We conclude $\mathit{map}\ \mathit{id} \equiv \mathit{id}$, hence, the *Functor* instance for lists satisfies the first functor law. The second law can be verified similarly.

As an example for a different data type that also supports a map operation consider the following definition of binary leaf trees[3].

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

[3] Binary leaf trees are binary trees that store values in their leaves.

A binary leaf tree is either empty, a leaf storing an arbitrary element, or an inner node with left and right sub trees. We can apply a polymorphic function to every element stored in a leaf using *fmap*:

```
instance Functor Tree where
  fmap _ Empty      = Empty
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Fork l r) = Fork (fmap f l) (fmap f r)
```

The proof that this definition of *fmap* satisfies the functor laws is left as an exercise. More interesting is the observation that we can now define non-trivial functions that can be applied to both lists and trees. For example, the function $fmap\ (length \circ dropWhile\ isSpace)$ can be used to map a value of type $[String]$ to a value of type $[Int]$ and also to map a value of type *Tree String* to a value of type *Tree Int*.

The type class *Functor* can not only be instantiated by polymorphic *data* types. The partially applied type constructor $\rightarrow$ for function types is also an instance of *Functor*:

```
instance Functor (a →) where
  fmap = (∘)
```

For $f \equiv (a \rightarrow)$ the function *fmap* has the following type.

$$fmap :: (b \rightarrow c) \rightarrow (a \rightarrow)\ b \rightarrow (a \rightarrow)\ c$$

If we rewrite this type using the more conventional infix notation for $\rightarrow$ we obtain the type $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$ which is exactly the type of the function composition operator $(\circ)$ defined in Subsection 1.1.1. It is tempting to make use of this coincidence and define the above *Functor* instance without further ado. However, we should check the functor laws in order to gain confidence in this definition. The proofs can be found in Appendix A.1.

Type constructor classes provide powerful means to overload functions. This results in increased potential for code reuse – sometimes to a surprising extend. For example, we can implement an instance of the *Functor* type class for type constructors like $(a \rightarrow)$ where we would not expect such possibility at first sight. The following subsection presents another type class that can be instantiated for many different types leading to a variety of different usage scenarios that can share identical syntax.

Monads

Monads are a very important abstraction mechanism in Haskell – so important that Haskell provides special syntax to write monadic code. Besides syntax, however, monads are nothing special but instances of an ordinary type class.

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m\ \textbf{where}\\&\quad \mathit{return} :: a \to m\ a\\&\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

Like functors, monads are unary type constructors. The *return* function constructs a monadic value of type $m\ a$ from a non-monadic value of type a. The function $\gg\!=$, pronounced *bind*, takes a monadic action of type $m\ a$ and a function that maps the wrapped value to another monadic action of type $m\ b$. The result of applying $\gg\!=$ is a combined monadic action of type $m\ b$.

The first monad that programmers come across when learning Haskell is often the *IO* monad and in fact, a clear separation of pure computations without side effects and input/output operations that incorporate external state was the main reason to add monads to Haskell. In Haskell, functions that interact with the outside world return their results in the *IO* monad, i.e., their result type is wrapped in the type constructor *IO*. Such functions are often called *IO* actions to emphasise their imperative nature and distinguish them from pure functions. There are predefined *IO* actions *getChar* and *putChar* that read one character from standard input and write one to standard output respectively.

$$\begin{aligned}&\mathit{getChar} :: \mathit{IO}\ \mathit{Char}\\&\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\ ()\end{aligned}$$

The *IO* action *putChar* has no meaningful result but is only used for its side effect. Therefore, it returns the value $()$ which is the only value of type $()$.

We can use these simple *IO* actions to demonstrate how to write more complex monadic actions using the functions provided by the type class *Monad*. For example, we can use $\gg\!=$ to sequence the actions that read and write one character:

$$\begin{aligned}&\mathit{copyChar} :: \mathit{IO}\ ()\\&\mathit{copyChar} = \mathit{getChar} \gg\!= \lambda c \to \mathit{putChar}\ c\end{aligned}$$

This combined action will read one character from standard input and directly write it back to standard output, when it is executed. It can be written more conveniently using Haskell's **do**-notation as follows.

$$
\begin{aligned}
&\mathit{copyChar} :: \mathit{IO}\ () \\
&\mathit{copyChar} = \mathbf{do}\ c \leftarrow \mathit{getChar} \\
&\qquad\qquad\quad \mathit{putChar}\ c
\end{aligned}
$$

In general, $\mathbf{do}\ x \leftarrow a; f\ x$ is syntactic sugar for $a \gg\!= \lambda x \rightarrow f\ x$ and arbitrarily many nested calls to $\gg\!=$ can be chained like this in the lines of a **do**-block. The imperative flavour of the special syntax for monadic code highlights the historical importance of input/output for the development of monads in Haskell.

It turns out, however, that monads can do much more than just sequence input/output operations.

For example, we can define a *Monad* instance for lists and use **do**-notation to elegantly construct complex lists from simple ones.

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Monad}\ [\,]\ \mathbf{where} \\
&\quad \mathit{return}\ x = [x] \\
&\quad l \gg\!= f \quad = \mathit{concat}\ (\mathit{map}\ f\ l)
\end{aligned}
$$

The *return* function for lists yields a singleton list and the $\gg\!=$ function maps the given function on every element of the given list and concatenates all lists in the resulting list of lists. We can employ this instance to compute a list of pairs from all elements in given lists.

$$
\begin{aligned}
&\mathit{pair} :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ (a, b) \\
&\mathit{pair}\ \mathit{xs}\ \mathit{ys} = \mathbf{do}\ x \leftarrow \mathit{xs} \\
&\qquad\qquad\qquad\quad\ \ y \leftarrow \mathit{ys} \\
&\qquad\qquad\qquad\quad\ \ \mathit{return}\ (x, y)
\end{aligned}
$$

For example, the call $\mathit{pair}\ [0,1]\ [\mathit{True}, \mathit{False}]$ yields a list of four pairs, viz., $[(0, \mathit{True}), (0, \mathit{False}), (1, \mathit{True}), (1, \mathit{False})]$. We can write the function *pair* without using the monad operations[4] but the definition with **do**-notation is arguably more readable.

The story does not end here. The data type for binary leaf trees also has a natural *Monad* instance:

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Monad\ Tree}\ \mathbf{where} \\
&\quad \mathit{return} = \mathit{Leaf} \\
&\quad t \gg\!= f = \mathit{mergeTrees}\ (\mathit{fmap}\ f\ t)
\end{aligned}
$$

This instance is similar to the *Monad* instance for lists. It uses *fmap* instead of *map* and relies on a function *mergeTrees* that computes a single tree from a tree of trees.

[4] $\lambda \mathit{xs}\ \mathit{ys} \rightarrow \mathit{concat}\ (\mathit{map}\ (\lambda x \rightarrow \mathit{concat}\ (\mathit{map}\ (\lambda y \rightarrow [(x, y)])\ \mathit{ys}))\ \mathit{xs})$

$$
\begin{aligned}
&\mathit{mergeTrees} :: \mathit{Tree}\ (\mathit{Tree}\ a) \to \mathit{Tree}\ a \\
&\mathit{mergeTrees}\ \mathit{Empty} \quad\ = \mathit{Empty} \\
&\mathit{mergeTrees}\ (\mathit{Leaf}\ t) \quad = t \\
&\mathit{mergeTrees}\ (\mathit{Fork}\ l\ r) = \mathit{Fork}\ (\mathit{mergeTrees}\ l)\ (\mathit{mergeTrees}\ r)
\end{aligned}
$$

Intuitively, this function takes a tree that stores other trees in its leaves and just removes the *Leaf* constructors of the outer tree structure. So, the $\gg\!=$ operation for trees replaces every leaf of a tree with the result of applying the given function to the stored value.

Now we benefit from our choice to provide such a general type signature for the function *pair*. We can apply the same function *pair* to trees instead of lists to compute a tree of pairs instead of a list of pairs. For example, the call $\mathit{pair}\ (\mathit{Fork}\ (\mathit{Leaf}\ 0)\ (\mathit{Leaf}\ 1))\ (\mathit{Fork}\ (\mathit{Leaf}\ \mathit{True})\ (\mathit{Leaf}\ \mathit{False}))$ yields the following tree with four pairs.

$$
\begin{aligned}
&\mathit{Fork}\ (\mathit{Fork}\ (\mathit{Leaf}\ (0, \mathit{True})) \\
&\qquad\qquad (\mathit{Leaf}\ (0, \mathit{False}))) \\
&\qquad (\mathit{Fork}\ (\mathit{Leaf}\ (1, \mathit{True})) \\
&\qquad\qquad (\mathit{Leaf}\ (1, \mathit{False})))
\end{aligned}
$$

Like functors, monads allow programmers to define very general functions that they can use on a variety of different data types. Monads are more powerful than functors because the result of the $\gg\!=$ operation can have a different structure than the argument. When using *fmap* the structure of the result is always the same as the structure of the argument – at least if *fmap* satisfies the functor laws.

The *Monad* type class also has a set of associated laws. The *return* function must be a left- and right-identity for the $\gg\!=$ operator which needs to satisfy an associative law.

$$
\begin{aligned}
&\mathit{return}\ x \gg\!= f \quad\ \equiv f\ x \\
&m \gg\!= \mathit{return} \quad\ \equiv m \\
&(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

These laws ensure a consistent semantics of the **do**-notation and allow equational reasoning about monadic programs. The verification of the monad laws for the list instance is left as an exercise for the reader. The proof for the *Tree* instance is in Appendix A.2.

Summary

In this section we have seen different abstraction mechanisms of functional programming languages that help programmers to write more modular and

reusable code. Type polymorphism (Section 1.1.1) allows to write functions that can be applied to a variety of different types because they ignore parts of their input. This feature is especially useful in combination with higher-order functions that allow to abstract from common programming patterns to define custom control structures like, e.g., the *map* function on lists. Lazy evaluation (Subsection 1.1.2) increases the modularity of algorithms because demand driven evaluation often avoids storing intermediate results which allows to compute with infinite data. With class-based overloading (Subsection 1.1.3) programmers can implement one function that has different behaviour on different data types such that code using these functions can be applied in many different scenarios. We have see two examples for type constructor classes, viz., functors and monads and started to explore the generality of the code they allow to write. Finally, we have seen that equational reasoning is a powerful tool to think about programs and their correctness.

A Proofs

A.1 Functor laws for $(a \rightarrow)$ instance

This is a proof of the functor laws

$$\begin{aligned} &\mathit{fmap}\ \mathit{id} &&\equiv \mathit{id} \\ &\mathit{fmap}\ (f \circ g) &&\equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g \end{aligned}$$

for the *Functor* instance

$$\begin{aligned} &\textbf{instance}\ \mathit{Functor}\ (a \rightarrow)\ \textbf{where} \\ &\quad \mathit{fmap} = (\circ) \end{aligned}$$

The laws are a consequence of the fact that functions form a monoid under composition with the identity element *id*.

$$\begin{aligned} & \mathit{fmap}\ \mathit{id}\ h \\ \equiv & \quad \{\ \text{definition of } \mathit{fmap}\ \} \\ & \mathit{id} \circ h \\ \equiv & \quad \{\ \text{definition of } (\circ)\ \} \\ & \lambda x \rightarrow \mathit{id}\ (h\ x) \\ \equiv & \quad \{\ \text{definition of } \mathit{id}\ \} \\ & \lambda x \rightarrow h\ x \\ \equiv & \quad \{\ \text{expansion}\ \} \\ & h \\ \equiv & \quad \{\ \text{definition of } \mathit{id}\ \} \\ & \mathit{id}\ h \end{aligned}$$

This proof makes use of the identity $(\lambda x \rightarrow f\ x) \equiv f$ for every function f. The second law is a bit more involved as it relies on associativity for function composition.

$$\begin{aligned} & \mathit{fmap}\ (f \circ g)\ h \\ \equiv & \quad \{\ \text{definition of } \mathit{fmap}\ \} \\ & (f \circ g) \circ h \\ \equiv & \quad \{\ \text{associativity of } (\circ)\ \} \\ & f \circ (g \circ h) \end{aligned}$$

$$
\begin{aligned}
&\equiv \quad \{ \text{ definition of } \mathit{fmap} \text{ (twice) } \} \\
&\quad \mathit{fmap}\ f\ (\mathit{fmap}\ g\ h) \\
&\equiv \quad \{ \text{ reduction } \} \\
&\quad (\lambda x \to \mathit{fmap}\ f\ (\mathit{fmap}\ g\ x))\ h \\
&\equiv \quad \{ \text{ definition of } (\circ) \} \\
&\quad (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ h
\end{aligned}
$$

Now it is only left to verify that function composition is indeed associative:

$$
\begin{aligned}
&\quad (f \circ g) \circ h \\
&\equiv \quad \{ \text{ definition of } (\circ) \text{ (twice) } \} \\
&\quad \lambda x \to (\lambda y \to f\ (g\ y))\ (h\ x) \\
&\equiv \quad \{ \text{ reduction } \} \\
&\quad \lambda x \to f\ (g\ (h\ x)) \\
&\equiv \quad \{ \text{ reduction } \} \\
&\quad \lambda x \to f\ ((\lambda y \to g\ (h\ y))\ x) \\
&\equiv \quad \{ \text{ definition of } (\circ) \text{ (twice) } \} \\
&\quad f \circ (g \circ h)
\end{aligned}
$$

A.2 Monad laws for *Tree* instance

This is a proof of the monad laws

$$
\begin{aligned}
&\mathit{return}\ x \gg\!= f \quad \equiv f\ x \\
&m \gg\!= \mathit{return} \quad \equiv m \\
&(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

for the *Monad* instance

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad\ Tree}\ \textbf{where} \\
&\quad \mathit{return} = \mathit{Leaf} \\
&\quad t \gg\!= f = \mathit{mergeTrees}\ (\mathit{fmap}\ f\ t) \\
&\mathit{mergeTrees} :: \mathit{Tree}\ (\mathit{Tree}\ a) \to \mathit{Tree}\ a \\
&\mathit{mergeTrees}\ \mathit{Empty} \quad = \mathit{Empty} \\
&\mathit{mergeTrees}\ (\mathit{Leaf}\ t) \quad = t \\
&\mathit{mergeTrees}\ (\mathit{Fork}\ l\ r) = \mathit{Fork}\ (\mathit{mergeTrees}\ l)\ (\mathit{mergeTrees}\ r)
\end{aligned}
$$

for the data type

$$
\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)
$$

The left-identity law follows from the definitions of the functions *return*, $\gg\!=$, *fmap*, and *mergeTrees*.

$$
\begin{aligned}
& \mathit{return}\ x \gg\!= f \\
\equiv\ & \quad \{ \text{ definitions of } \mathit{return} \text{ and } \gg\!= \} \\
& \mathit{mergeTrees}\ (\mathit{fmap}\ f\ (\mathit{Leaf}\ x)) \\
\equiv\ & \quad \{ \text{ definition of } \mathit{fmap} \} \\
& \mathit{mergeTrees}\ (\mathit{Leaf}\ (f\ x)) \\
\equiv\ & \quad \{ \text{ definition of } \mathit{mergeTrees} \} \\
& f\ x
\end{aligned}
$$

We prove the right-identity law by induction over the structure of m. The *Empty* case follows from the observation that $\mathit{Empty} \gg\!= f \equiv \mathit{Empty}$ for every function f, i.e., also for $f \equiv \mathit{return}$.

$$
\begin{aligned}
& \mathit{Empty} \gg\!= f \\
\equiv\ & \quad \{ \text{ definition of } \gg\!= \} \\
& \mathit{mergeTrees}\ (\mathit{fmap}\ f\ \mathit{Empty}) \\
\equiv\ & \quad \{ \text{ definition of } \mathit{fmap} \} \\
& \mathit{mergeTrees}\ \mathit{Empty} \\
\equiv\ & \quad \{ \text{ definition of } \mathit{mergeTrees} \} \\
& \mathit{Empty}
\end{aligned}
$$

The *Leaf* case follows from the left-identity law because $\mathit{return} \equiv \mathit{Leaf}$.

$$
\begin{aligned}
& \mathit{Leaf}\ x \gg\!= \mathit{return} \\
\equiv\ & \quad \{ \text{ definition of } \mathit{return} \} \\
& \mathit{return}\ x \gg\!= \mathit{return} \\
\equiv\ & \quad \{ \text{ first monad law} \} \\
& \mathit{return}\ x \\
\equiv\ & \quad \{ \text{ definition of } \mathit{return} \} \\
& \mathit{Leaf}\ x
\end{aligned}
$$

The *Fork* case makes use of the induction hypothesis and the observation that $\mathit{Fork}\ l\ r \gg\!= f \equiv \mathit{Fork}\ (l \gg\!= f)\ (r \gg\!= f)$

$$
\begin{aligned}
& \mathit{Fork}\ l\ r \gg\!= f \\
\equiv\ & \quad \{ \text{ definition of } \gg\!= \} \\
& \mathit{mergeTrees}\ (\mathit{fmap}\ f\ (\mathit{Fork}\ l\ r)) \\
\equiv\ & \quad \{ \text{ definition of } \mathit{fmap} \} \\
& \mathit{mergeTrees}\ (\mathit{Fork}\ (\mathit{fmap}\ f\ l)\ (\mathit{fmap}\ f\ r)) \\
\equiv\ & \quad \{ \text{ definition of } \mathit{mergeTrees} \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Fork}\ (\mathit{mergeTrees}\ (\mathit{fmap}\ f\ l))\ (\mathit{mergeTrees}\ (\mathit{fmap}\ f\ r)) \\
\equiv\ & \quad \{\ \text{definition of } \gg= \text{ (twice)}\ \} \\
& \mathit{Fork}\ (l \gg= f)\ (r \gg= f)
\end{aligned}
$$

Now we can apply the induction hypothesis.

$$
\begin{aligned}
& \mathit{Fork}\ l\ r \gg= \mathit{return} \\
\equiv\ & \quad \{\ \text{previous derivation}\ \} \\
& \mathit{Fork}\ (l \gg= \mathit{return})\ (r \gg= \mathit{return}) \\
\equiv\ & \quad \{\ \text{induction hypothesis (twice)}\ \} \\
& \mathit{Fork}\ l\ r
\end{aligned}
$$

Finally we prove assiciativity of $\gg=$ by structural induction. The *Empty* case follows from the above observation that $\mathit{Empty} \gg= f \equiv \mathit{Empty}$ for every function f.

$$
\begin{aligned}
& (\mathit{Empty} \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{above observation for } \mathit{Empty} \text{ (twice)}\ \} \\
& \mathit{Empty} \\
\equiv\ & \quad \{\ \text{above observation for } \mathit{Empty}\ \} \\
& \mathit{Empty} \gg= (\lambda x \to f\ x \gg= g)
\end{aligned}
$$

The *Leaf* case follows again from the first monad law.

$$
\begin{aligned}
& (\mathit{Leaf}\ y \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{definition of } \mathit{return}\ \} \\
& (\mathit{return}\ y \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{first monad law}\ \} \\
& f\ y \gg= g \\
\equiv\ & \quad \{\ \text{first monad law}\ \} \\
& \mathit{return}\ y \gg= (\lambda x \to f\ x \gg= g) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{return}\ \} \\
& \mathit{Leaf}\ y \gg= (\lambda x \to f\ x \gg= g)
\end{aligned}
$$

The *Fork* case uses the identity $\mathit{Fork}\ l\ r \gg= f \equiv \mathit{Fork}\ (l \gg= f)\ (r \gg= f)$ that we proved above and the induction hypothesis.

$$
\begin{aligned}
& (\mathit{Fork}\ l\ r \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{property of } \gg=\ \} \\
& \mathit{Fork}\ (l \gg= f)\ (r \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{property pf } \gg=\ \} \\
& \mathit{Fork}\ ((l \gg= f) \gg= g)\ ((r \gg= f) \gg= g) \\
\equiv\ & \quad \{\ \text{induction hypothesis}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Fork}\ (l \gg\!= (\lambda x \to f\ x \gg\!= g))\ (r \gg\!= (\lambda x \to f\ x \gg\!= g)) \\
\equiv\quad & \{\ \text{property of } \gg\!= \ \} \\
& \mathit{Fork}\ l\ r \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

This finishes the proof of the three monad laws for the *Tree* instance.