DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Functional Programming in *Haskell*

— Winter 2025 —

Prof. Dr. Karl Stroetmann

February 7, 2025

Danger **Work in Progress!** Danger

Abstract

This is a very short introduction to functional programming with *Haskell*. It is not intended to be a Haskell course. My intention merely is for the reader to get a taste of what programming in *Haskell* feels like. If I succeed in convincing the reader that *Haskell* is a programming language that is both usefull and mind extending, then I consider my job done. Therefore, this short paper will just present some of the highlights of the programming language *Haskell* via examples that I hope the reader finds intriguing enough so that she feels inclined to read some of the outstanding books introducing Haskell in more depth. There are three books that I recommend for those readers who want to understand *Haskell* in more depth:

1. *Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming*

   by Rebecca Skinner [Ski23].

2. *Programming in Haskell*, second edition

   by Graham Hutton [Hut16].

3. *Learn You a Haskell for Great Good!*,

   by Miran Lipovača [Lip11].

Chapter 1

Introduction

In this introduction I will do two things:

1. First, I discuss those features of Haskell that set Haskell apart from other programming languages.
2. Second, I will present a few short example programs that give a first taste of Haskell.

1.1 Why Haskell is Different

Before we present any details of *Haskell*, let us categorize this programming language so that we have an idea about what to expect. *Haskell* has the following properties:

1. *Haskell* is a functional programming language.

   A functional programming language is any programming language that treats functions as first class citiziens:

   (a) A function can be given as an argument to another function.

   (b) A function can be produce a function as its result.

   A well known programming language that supports functional programming is *Python* and there are several books discussing functional programming in *Python*, e.g. [Lot22], [Mer15], and [Rei23].

2. *Haskell* is statically typed.

   Every variable in Haskell has a fixed type, which can not be changed. In this respect, *Haskell* is similar to the programming language *Java*. However, in contrast to *Java*, we do not have to declare the type of every variable and every function because most of the time the type of a variable can be inferred by the type system. Therefore, in Haskell we usually specify only the types of non-trivial functions.

   The benefit of this approach is that many type errors will already be caught by the compiler. This is in contrast to programs written in a dynamically typed language like *Python*, where type errors are only discovered at runtime.

3. *Haskell* is a pure functional programming language.

   Once a variable is assigned a value, this value can not be changed. For example, if we want to sort a list, we are not able to change the list data structure. All we can do is to compute a new list which contains the same elements as the old list and which, furthermore, is sorted.

   The property of being a pure language sets *Haskell* apart from most other programming languages. Even the language *Scala*, which is designed as a modern functional programming language, is not pure.

   What is the big deal about purity? On one hand, it forces the user to program in a declarative style. Although, in general, nobody likes to be forced to do something, there is a huge benefit in pure programming.

   (a) In a pure programming language, functions will always return the same result when they are called with the same arguments. This property is called referential transparency. This makes reasoning about code easier, as you can replace a function call with its result without changing the behavior of the program. Therefore the correctness of functions can be verified mathematically.

   (b) Since pure functions do not depend on or modify external state, their behavior is entirely predictable. The advantage is that testing becomes straightforward because functions can be tested in isolation without worrying about interactions with other parts of the system.

   (c) Compilers for pure languages can make aggressive optimizations, such as caching function results (memoization) or reordering computations, because they know that functions are side-effect-free. The advantage is that programs can often run faster.

   (d) Furthermore, concurrency becomes much easier to manage when functions do not use global variables and do not change their arguments.

4. *Haskell* is a compiled language similar to *Java* and C, but additionally offers an interpreter. Having an interpreter is beneficial for rapid prototyping. The property that *Haskell* programs can be compiled ensures that the resulting programs can befaster than, for example, *Python* programs.

5. *Haskell* is a lazy language. The programming language C is an eager language. If an expression of the form

   $$f(a_1, \cdots, a_n)$$

   has to be evaluated, first the subexpressions a_1, $\cdots$ a_n are evaluated. Let us assume that a_i is evaluated to to value x_i. The, $f(x_1, \cdots, x_n)$ is computed. This might be very inefficient. Consider the following contrived example:

```
int f(int x, int y) {
    if (x == 2) {
        return 42;
    }
    return 2 * y;
}
```

Let us assume that the expression

```
f(h(0), g(1))
```

needs to be evaluated and that the computation of `g(1)` is very expensive. In a C-program, the expressions and `h(0)` and `g(1)` will be both evaluated. If it turns out that `h(0)` is 2, then the evaluation of `g(1)` is not really necessary. Nevertheless, in `C` this evaluation takes place because `C` has an eager evaluation strategy. In contrast, an equivalent *Haskell* would not evaluate the expression `h(0)` and hence would be much more efficient.

6. *Haskell* is difficult to learn.

   You might ask yourself why *Haskell* hasn't been adopted more widely. After all, it has all these cool features mentioned above. The reason is that learning *Haskell* is a lot more difficult then learning a language like *Python* or *Java*. There are two reasons for this:

   (a) First, *Haskell* differs a lot from those languages that most people know.

   (b) In order to be very concise, the syntax of *Haskell* is quite different from the syntax of established programming languages.

   (c) *Haskell* requires the programmer to think on a very high level of abstraction. Many students find this difficult.

   (d) Lastly, and most importantly, *Haskell* supports the use of a number of concepts like, e.g. functors and monads from category theory. It takes both time and mathematical maturity to really understand these concepts.

   If you really want to understand the depth of *Haskell*, you have to dive into those topics. That said, while you have to understands both functors and monads, you do not have to understand category theory.

   (e) Fortunately, it is possible to become productive in *Haskell* without understanding functors and monads. Therefore, this lecture will focuss on those parts of *Haskell* that are more easily accessible.

1.2 A First Taste of Haskell

Computing All Prime Numbers

A prime number is a natural number p that is different from 1 and that can not be written as a product of two natural numbers a and b that are both different from 1. If we denote the set of all prime numbers with the symbol $\mathbb{P}$, we therefore have:

$$\mathbb{P} = \bigl\{ p \in \mathbb{N} \mid n \neq 1 \wedge \forall a, b \in \mathbb{N} : (a \cdot b = p \implies a = 1 \vee b = 1) \bigr\}$$

An efficient method to compute the prime numbers is the sieve of Eratosthenes. This is an algorithm used to find all primes up to a given number. The method works by iteratively marking the multiples of each prime number starting from 2. The numbers which remain unmarked at the end of the process are the prime numbers. For example, consider the list of integers from 2 to 30:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30.$$

We will now compute the set of all primes less or equal than 30 using the Sieve of Eratosthenes.

1. The first number in this list is 2 and hence 2 is prime:
   $$\mathbb{P} = \{2, \cdots\}.$$
2. We remove all multiples of 2 (i.e., 2, 4, 6, 8, ..., 30). This leaves us with the list
   $$3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29.$$
   The first remaining number 3 is prime. Hence we have
   $$\mathbb{P} = \{2, 3, \cdots\}$$
3. We remove all multiples of 3 (i.e., 3, 6, 9, 12, ..., 27). Note that some numbers (like 6 or 12, for instance) may already have been removed. This leaves us with the list
   $$5, 7, 11, 13, 17, 19, 23, 25, 29.$$
   The first remaining number 5 is prime. Hence we have
   $$\mathbb{P} = \{2, 3, 5, \cdots\}.$$
4. We remove all multiples of 5 (i.e., 5, 10, 15, 20, ...). This leaves us with the list
   $$7, 11, 13, 17, 19, 23, 29.$$
   The first remaining number 7 is prime. Hence we have
   $$\mathbb{P} = \{2, 3, 5, 7, \cdots\}.$$
5. Since the square of 7 is 49, which is greater that 30, there is no need to remove the multiples of 7, since all multiples $a \cdot 7$ for $a < 7$ have already been removed and all multiples $a \cdot 7$ for $a \geq 7$ are greater than 30. Hence the remaining numbers are all prime and we have found that the prime numbers less or equal than 30 are:
   $$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}.$$

```python
def sieve_of_eratosthenes(n):
    """Return a list of prime numbers up to n (inclusive)."""
    if n < 2:
        return []
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False  # 0 and 1 are not primes
    p = 2
    while p * p <= n:
        if is_prime[p]:
            for i in range(p * p, n + 1, p):
                is_prime[i] = False
        p += 1
    return [i for i, prime in enumerate(is_prime) if prime]
```

Figure 1.1: A *Python program to compute the prime numbers up to* n.

Figure 1.1 on page 4 shows a Python script that implements the Sieve of Eratosthenes to compute all prime numbers up to a limit `n`. This script first initializes the list `is_prime` to track prime status. It then iteratively marks the multiples of each prime number, finally collecting and printing all numbers that remain marked as prime. This script implements one optimization: If `p` is a prime, then only the multiples of `p` that have the form $a \cdot \mathtt{p}$ with $a \geq \mathtt{p}$ have to be removed from the list, since a product of the form $a \cdot \mathtt{p}$ with $a < \mathtt{p}$ has already been removed when removing multiples of a or, if a is not prime, of multiples of whatever prime is contained in a.

```
1  primes :: [Integer]
2  primes = sieve [2..]
3
4  sieve :: [Integer] -> [Integer]
5  sieve (p:ns) = p : sieve [n | n <- ns, mod n p /= 0]
```

Figure 1.2: Computing the prime numbers.

Figure 1.2 on page 5 shows a *Haskell* program to compute **all** primes. Yes, you have read that correct. It doesn't compute the primes up to a given number, but rather it computes all primes.

The first thing to note is that line 1 and line 4 are type annotations. They have only been added to aid us in understanding the program. If we would drop these lines, the program would still work. Hence, we have an efficient 2-line program to compute the prime numers. Lets discuss this program line by line.

(a) Line 1 states that the function `primes` returns a list of `Integer`s. `Integer` is the type of all arbitrary precision integers. The fact that we have enclosed the type name `Integer` in the square brackets "`[`" and "`]`" denotes that the result has the type *list* of `Integer`.

(b) In line 2, the expression "`[2..]`" denotes the list of all integers starting from 2. Since *Haskell* is lazy, it is able to support infinite data structures. The trick is that these lists are only evaluated as much as they are needed. As long as we do not inspect the complete list, everything works fine.

(c) Line 2 calls the function `sieve` with the argument `[2..]`. *Haskell* uses prefix notation for calling a function. If f is a function an a_1, a_2, and a_3 are arguments of this function, then the invocation of f with these arguments is written as

$$f\ a_1\ a_2\ a_3$$

Note that the expression

$$f(a_1, a_2, a_3)$$

denotes something different: This expression would apply the function f to a single argument, which is the triple (a_1, a_2, a_3).

(d) Line 4 declares the type of the function `sieve`. This function takes one argument, which is a list of `Integer`s and returns a list of `Integer`s.

(e) Line 5 defines the function `sieve` that takes a list of numbers l that has the following properties:

- The list l is a sorted ascendingly.
- If p is the first element of the list l, then p is a prime number.
- The list l does not contain multiples of any number q that is less than p:

$$\forall q \in \mathbb{N} : \bigl(q < p \rightarrow \forall n \in \mathbb{N} : n \cdot q \notin l\bigr).$$

Given a list l with these properties, the expression

```
sieve l
```

returns a list of all prime number that are greater or equal than p, where p is the first element of l:

$$\texttt{sieve}\, l = [\, q \in l \mid q \in \mathbb{P}\,].$$

Hence, when `sieve` is called with the list of all natural number greater or equal than 2 it will return the list of all prime numbers, since 2 is the smallest prime numer.

There is a lot going on in the definition of the function `sieve`. We will discuss this definition in minute detail.

1. The function `sieve` is defined via matching. We will discuss matching in more detail in the next chapter. For now we just mention that the expression

   ```
   (p:ns)
   ```

   matches a list with first element `p`. The remaining elements are collected in the list `ns`. For example, if `l = [2..]`, i.e. if l is the list of all natural numbers greater or equal than 2, then the variable `p` is bound to the number 2, while the variable `ns` is bound to the list of all natural numbers greater or equal than 3, i.e. `ns = [3..]`.

2. The right hand side of the function definition, i.e. the part after the symbol "=" defines the value that is computed by the function `sieve`. This value is computed by calling the function "`:`", which is also known as the cons function because it constructs a list. The operator "`:`" takes two arguments. The first argument is a value `u` of some type a and the second argument `us` is list of elements of the same type a. An expression of the form

   ```
   u : us
   ```

   then returns a list where `u` is the first elements and `us` are the remaining elements. For example, we have

   ```
   1 : [2, 3, 4] = [1, 2, 3, 4].
   ```

3. The recursive invocation of the function sieve takes a list comprehension as ist first argument. the expression

   ```
   [n | n <- ns, mod n p /= 0]
   ```

   computes the list of all those number `n` from the list `ns` that have a non-zero remainder when divided by the prime number `p`, i.e. this list contains all those number from the list `ns` that are not multiples of `p`. There are two things to note here concerning the syntax of *Haskell*:

- Functions are written with prefix notation. For example, we first write the function name `mode` followed by the arguments `m` and `p`. In *Python* the expression

  ```
  mod m p
  ```

  would have been written as `m % p`.
- The operator "/=" expresses inequality, i.e. the *Haskell* expression

  `a /= b` would be written as `a != b`

  in the programming language *Python*.

Putting everything together, the *Haskell* expression

```
[n | n <- ns, mod n p /= 0]
```

is therefore equivalent to the *Python* expression

```
[n for n in ns if n % p != 0].
```

Bibliography

[Hut16] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, San Francisco, CA, 2011.

[Lot22] Steven F. Lott. *Functional Python Programming - Third Edition*. Packt Publishing, 3rd edition, 2022.

[Mer15] David Mertz. *Functional Programming in Python*. Packt Publishing, 2015.

[Rei23] James L. Reid. *Python Functional Programming: A Hands-on Guide To Write Clean & Powerful Applications*. Independently Published, 2023.

[Ski23] Rebecca Skinner. *Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming*. Pragmatic Bookshelf, 2023.