DHBW
Duale Hochschule
Baden-Württemberg

Funktionale Programmierung

Emil Schläger
Matrikelnummer:
inf23123@lehre.dhbw-stuttgart.de

Maxim Tausch
Matrikelnummer: 8395214
inf23142@lehre.dhbw-stuttgart.de

DHBW
Duale Hochschule
Baden-Württemberg

Inhaltsverzeichnis

1. Einführung

Die funktionale Programmierung stellt ein eigenständiges Programmierparadigma dar, das sich grundlegend von den imperativen Ansätzen unterscheidet, die in vielen Programmiersprachen dominieren. Im Mittelpunkt stehen dabei Funktionen als zentrale Abstraktion, deren Verhalten ausschließlich durch ihre Eingaben bestimmt ist und keine veränderlichen Zustände besitzen. Dieser Fokus auf Unveränderlichkeit und klare Abhängigkeiten führt zu Programmen, deren Ablauf mathematisch gut beschreibbar ist und deren Eigenschaften sich häufig präziser analysieren lassen.

In den vergangenen Jahren hat die funktionale Programmierung aufgrund der zunehmenden Bedeutung nebenläufiger und verteilter Systeme neues Interesse gewonnen. Da unveränderliche Datenstrukturen und zustandsfreie Funktionen weniger anfällig für typische Nebenläufigkeitsprobleme sind, bietet das funktionale Paradigma interessante theoretische und praktische Vorteile. Gleichzeitig stellt es jedoch auch Anforderungen an die Art und Weise, wie Probleme modelliert und programmiert werden, die für viele Entwicklerinnen und Entwickler zunächst ungewohnt sind.

Ziel dieser Hausarbeit ist es, die grundlegenden Prinzipien der funktionalen Programmierung systematisch darzustellen, ihre spezifischen Konzepte zu erläutern und ihre Anwendungsbereiche sowie Grenzen aufzuzeigen. Die Arbeit soll somit einen strukturierten Überblick bieten, der die Bedeutung funktionaler Ansätze in der aktuellen Softwareentwicklung einordnet.

2. Side Effects

Side Effects ist alles was eine Funktion macht, die nicht Teil ihres Outputs sind. Das Ändern von globalen Variablen, das Modifizieren von Objekten, das Schreiben in Dateien oder das Aufbauen von Netzwerkverbindungen sind alles Beispiele für Side Effects. Der Nachteil von Side Effects ist, dass sie den Zustand des Programms verändern können, was zu unerwartetem Verhalten führen kann. Dies erschwert das Testen und Debuggen von Code, da die Funktion nicht mehr isoliert betrachtet werden kann. Außerdem leidet die Transparenz des Codes, da der Leser nicht sofort erkennen kann, welche Teile des Codes den Zustand verändern. Der Leser muss den gesamten Kontext verstehen, um die Auswirkungen einer Funktion vollständig zu begreifen. Ein Beispiel für Side Effects ist eine Funktion, die eine globale Variable ändert:

```
static int counter = 0;

static void increment() {
    counter += 1;
}

public static void main(String[] args) {
    increment();
    System.out.println(counter);  // Ausgabe: 1
}
```

Listing 1: Side Effects in *prozedureller Programmierung*

Funktionale Programmierung hingegen strebt danach, Side Effects zu minimieren oder klar vom Rest des Codes zu trennen, um die Vorteile reiner Funktionen zu nutzen. *Pure Functions* sind Funktionen, die für dieselben Eingaben immer dieselben Ausgaben liefern und keine Side Effects haben:

$$f : A \to B, x \mapsto y$$

Die Funktion f erhält als Parameter einen Wert x vom Typ A und gibt einen Wert y vom Typ B zurück. Dabei wird x selbst nicht verändert, und es gibt keine anderen Auswirkungen auf den Zustand des Programms. Eine solche Funktion folgt also der mengentheoretischen Definition einer Funktion als deterministische Abbildung der Menge A in die Menge B.

Dieser Determinismus erleichtert das Verständnis, Testen und die Wiederverwendbarkeit von Code erheblich. Ein Beispiel für eine reine Funktion ist:

```
static int increment(int x) {
    return x + 1;
}

public static void main(String[] args) {
    int counter = increment(0);
    System.out.println(counter);  // Ausgabe: 1
}
```

Listing 2: Pure Function in *FP*

In dem Beispiel aus Listing 2 hat die Funktion `increment` keine Side Effects, da sie nur den Wert ihrer Eingabe um 1 erhöht, ohne den Zustand des Programms zu verändern. Außerdem erkennt man sofort, dass `increment(0)` immer `1` zurückgibt, unabhängig vom Kontext.

Jedoch kommt man in der Praxis nicht komplett ohne Side Effects aus, da Programme mit der Außenwelt interagieren müssen um einen Nutzen zu haben (z.B. Benutzereingaben, Datenbanken, Netzwerke). Es ist daher notwendig, Side Effects zu verwalten und zu kontrollieren, um die Vorteile der funktionalen Programmierung zu nutzen, ohne die Notwendigkeit von Side Effects zu vernachlässigen. Es ist außerdem möglich, Side Effects an den Rand des Programms zu verlagern, sodass der Großteil des Codes rein und frei von Side Effects bleibt. Dies kann beispielsweise duch den Einsatz von Datenbanken ermöglicht werden, indem der Zustand in der Datenbank gespeichert wird und die Hauptlogik des Programms rein bleibt.

2.1. Transparenz

Beim *OOP* ist Transparenz im Code oft eingeschränkt, da Methoden den internen Zustand von Objekten verändern zu können. Man muss den die gesamte Klasse und ihre Zustände verstehen, um die Auswirkungen einer einzelnen Methode zu verstehen. Das erschwert das Lesen und Warten, des Codes, da die Auswirkungen einer Methode nicht isoliert betrachtet werden können. Reine Funktionen hingegen können isoliert betrachtet und verstanden werden. Beim Beispiel Listing 1 muss man wissen, wechen Zustand die Variable `counter` hat, um zu verstehen, was das Ergebnis von `increment()` ist. Man kann erahnen, dass `increment()` den Zustand von Counter erhöht, aber man weiß nicht das konkrete Ergebnis ohne den aktuellen Zustand zu kennen. Hier ist das noch kein großes Problem, aber bei komplexeren Methoden und Klassen wird es schnell unübersichtlich, sodass nicht direkt klar ist, was eine Methode genau macht.

2.2. Concurrency

Unter *Concurrency* versteht man die Fähigkeit eines Programms, mehrere Aufgaben gleichzeitig auszuführen. Ein Problem im *OOP* ist der Umgang mit *Shared Mutable State*, also mit gemeinsam genutztem, veränderbarem Zustand. Wenn mehrere Threads gleichzeitig auf denselben Zustand zugreifen und diesen ändern, kann es zu Inkonsistenzen kommen. Das führt zu unerwartetem Verhalten, da die Threads sich gegenseitig beeinflussen. In *OOP* wird dieses Problem mit *Locks* gelöst, die sicherstellen, dass nur ein Thread gleichzeitig auf den Zustand zugreifen kann. Dies kann jedoch zu *Deadlocks* führen, bei denen zwei oder mehr Threads sich gegenseitig blockieren und nicht weiterarbeiten können. In der *funktionalen Programmierung* hingegen sind Daten *immutable*. Ein Thread in der *funktionalen Programmierung* bekommt also keinen globalen Zustand, den er ändern kann, sondern eine Kopie des Zustands. Wenn ein Thread eine Änderung vornehmen möchte, dann erstellt er eine neue Version des Zustands, anstatt den ursprünglichen Zustand zu ändern. Damit können Threads nicht mehr konkurrieren und sich gegenseitig beeinflussen. In der *funktionalen Programmierung* wird das Problem der *Concurrency* vermieden indem es keine gemeinsamen Zustände gibt.

Das Problem der *Concurrency* soll anhand eines Beispiels verdeutlicht werden. Im folgenden Beispiel wird die Summe der Quadrate der Zahlen von 0 bis $N - 1$ im *OOP* berechnet:

```
class Acc {
    long sum = 0;  // geteilter, veränderlicher Zustand

    void add(long v) {
        // ohne Lock: race-conditions möglich
        sum += v;
    }
}

static void worker(Acc acc, int[] items) {
    for (int i : items) {
        acc.add((long) i * i);
    }
}

public static void main(String[] args) throws InterruptedException {
    int N = 100_000;
    Acc acc = new Acc();
    int[] items = IntStream.range(0, N).toArray();
    int half = items.length / 2;

    Thread t1 = new Thread(() -> worker(acc, Arrays.copyOfRange(items, 0, half)));
    Thread t2 = new Thread(() -> worker(acc, Arrays.copyOfRange(items, half,
items.length)));
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println("OOP (ohne Lock) -> sum = " + acc.sum);
    // Erwartet: N*(N-1)*(2N-1)/6; ohne Lock meist kleiner/falsch
}
```

Listing 3: Multithreading in *OOP*

Im Beispiel aus Listing 3 wird eine Klasse `Acc` definiert, die einen gemeinsamen, veränderlichen Zustand `sum` enthält. Zwei Threads werden gestartet, die jeweils die Quadrate der Zahlen in einem bestimmten Bereich berechnen und zur Summe hinzufügen. Da kein Lock verwendet wird, können Race-Conditions auftreten, was zu einem falschen Ergebnis führt. Wenn beide Threads gleichzeitig auf `self.sum` zugreifen und diesen Wert ändern, kann es passieren, dass eine Änderung die andere überschreibt, was zu einem inkonsistenten Zustand führt.

Im folgenden Beispiel wird dasselbe Problem in der *funktionalen Programmierung* gelöst:

```
static long square(int x) {
    return (long) x * x;  // reine Funktion, keine Seiteneffekte
}

public static void main(String[] args) {
    int N = 100_000;
    int[] items = IntStream.range(0, N).toArray();

    // map-ähnliche FP-Semantik mit parallelStream
    long total = Arrays.stream(items)
        .parallel()
        .mapToLong(Main::square)
        .sum();  // Aggregation erfolgt danach (thread-safe)

    System.out.println("FP (no shared mutation) -> sum = " + total);
}
```

Listing 4: Multithreading in *FP*

Im Beispiel aus Listing 4 wird die Funktion `square` definiert, die eine reine Funktion ist und keine Seiteneffekte hat. Mehrere Threads werden gestartet, die jeweils die Quadrate der Zahlen in einem bestimmten Bereich berechnen. Da es keinen gemeinsamen, veränderlichen Zustand gibt, treten keine Race-Conditions auf, und das Ergebnis ist konsistent.

3. Fundamentale Konzepte des FP

Funktionale Programmierung ist tief verwurzelt in den theoretischen Feldern der Kategorientheorie und des Lambda Kalküls. Auf diese theoretischen Grundlagen einzugehen, überschreitet den Umfang der Arbeit um ein Weites, weshalb wir nur die wichtigsten Begrifflichkeiten und Konzepte erörter werden.

3.1. Notation

Im Folgenden werden wir Signaturen von Funktionen basierend auf ihren Datentypen in folgender Form schreiben:

$$X \to Y$$

Hierbei ist X der Vektor der Datentypen der Parameter, und Y der Datentyp des Rückgabewerts der Funktion. Um Funktionstypen in Java darzustellen, nutzen wir die generische Klasse `Function` aus dem `java.util.function` Paket:

```
Function<X, Y>
```

Anzumerken ist, dass bei dieser Schreibweise vorausgesetzt wird, dass die beschriebene Funktion rein ist.

3.2. Higher-Order Functions

Als "Higher-Order Function" (HoF) wird jede Funktion bezeichnet, die entweder durch eine andere Funktion parameterisiert wird, oder eine Funktion als Rückgabewert besitzt. In Java können wir dies durch funktionale Interfaces realisieren:

```
int applyFToX(int x, Function<Integer, Integer> f) {
    return f.apply(x);
}
```

Dies ist ein Beispiel für den Syntax einer *HoF* in Java. Genutzt werden kann diese, indem man `applyFToX` den Bezeichner einer anderen Funktion übergibt:

```
int square(int x) {
    return x * x;
}

assert applyFToX(4, this::square) == 16;
```

3.2.1. Anonymous Functions

Besonders für kleine Funktionen, wie `square` im vorherigen Beispiel, kann es schnell verbos und unleserlich werden, jede dieser Funktionen seperat mit einem Bezeichner zu deklarieren. In den meisten Sprachen gibt es deshalb einen Weg, Funktionen ohne Bezeichner zu initialisieren, um sie direkt an HoFs zu übergeben. In Java geschieht dies durch den Lambda Syntax:

```
(arg1, arg2, arg3, ...) -> <result>
```

Das obige Beispiel kann demnach bedeutend kompakter geschrieben werden, ohne die Funktion `square` seperat zu deklarieren:

```
applyFToX(4, x -> x * x);
```

Anonyme Funktionen gibt es in beinahe allen modernen Programmiersprachen. In JavaScript beispielsweise ist der Syntax analog:

```
(x) => x * x
```

Python besitzt den sogenannten lambda-Syntax:

```
lambda x: x * x
```

3.2.2. Currying

Wie im vorherigen Kapitel erwähnt, können *Higher-Order Functions* auch Funktionen zurückgeben. Ein Anwendungsfall für dieses Pattern ist eine Art "Konstruktor" für Funktionen. Dies lässt sich gut aufzeigen am Beispiel der vorher eingeführten `square` Funktion: Es soll nun nicht nur quadriert werden, sondern der Exponent soll konfigurierbar sein. Die triviale Lösung hierfür ist eine zweistellige Funktion

$$(\text{int}, \text{int}) \to \text{int},$$

wo der Exponent ein weiterer Parameter ist:

```
int power(int base, int exponent) {
    return (int) Math.pow(base, exponent);
}
```

Wollen wir nun eine Funktion mit dem selben Exponenten häufiger verwenden, können wir die Funktion `power` auch interpretieren als eine Funktion

$$\text{int} \to (\text{int} \to \text{int}),$$

die den Exponenten als Parameter nimmt, und eine Funktion zurückgibt, welche das Potenzieren zu diesem Exponenten durchführt:

```
Function<Integer, Integer> cPower(int exponent) {
    return base -> (int) Math.pow(base, exponent);
}
```

Wir können `cPower` nun nutzen, um mehrere Exponentialfunktionen zu erstellen:

```
Function<Integer, Integer> square = cPower(2);
Function<Integer, Integer> cube = cPower(3);
```

```
assert square.apply(2) == power(2, 2);
assert cube.apply(2) == power(2, 3);
```

Diese Re-Interpretation einer Funktion mit mehreren Parametern als eine *Higher-Order Function* nennt sich *Currying* [1]. Zu bemerken ist, dass die zurückgegebene Funktion den Kontext `exponent` beibehält, obwohl sie den Scope der Funktion `c_power` verlässt. Sie "captured" die Variable `exponent`. Capturing ist ein Weg, wie (immutable) State zwischen Funktionen weitergereicht werden kann. **TODO: irgendwas zitieren für den Bullshit den ich da labere (should be like 90% correct)**

3.3. Monaden

Das Pattern der Monade ist der Weg, wie deterministisch mit Seiteneffekten umgegangen werden kann. Monaden an sich sind ein Konzept aus der Kategorientheorie und entsprechend tief theoretisch verwurzelt, mit verschiedensten äquivalenten Definitionen. Wir wollen Monaden aber nur aus der Perspektive eines Software-Engineers betrachten, und werden deshalb theoretische Grundlagen auslassen. Erwähnt sei, dass folgende Definition einer Monade auf einem sogenanten *Kleisli Tripel* basiert **TODO: CITATION**.

3.3.1. Definition

Eine Monade kann definiert werden als ein Parameterisierter Datentyp $M\langle T\rangle$, der Methoden mit den Folgenen Signaturen bereitstellt [2]:

$$\text{unit} : T \to M\langle T\rangle \tag{1}$$

$$\text{bind} : (M\langle A\rangle,\ A \to M\langle B\rangle) \to M\langle B\rangle \tag{2}$$

Damit M tatsächlich eine Monade ist, muss `unit` als Neutrales Element bezüglich der `bind` Operation agieren (3) `bind` assoziativ sein (4) [3]:

$$\begin{aligned} \text{bind}(\text{unit}(a), f) &\equiv f(a) \\ \text{bind}(a, \text{unit}) &\equiv a \end{aligned} \tag{3}$$

$$\text{bind}(\text{bind}(a, f), g) \equiv \text{bind}(a, \text{bind}(f(a), g)) \tag{4}$$

3.3.2. Maybe Monade

Die Rolle der Methoden `unit` und `bind` können gut anhand des Beispiels der sogenannten "Maybe Monade" demonstriert werden. Diese abstrahiert den Side-Effect der möglichen Nicht-Existenz des enkapsulierten Wertes. Listing Listing 5 ist eine beispielhafte, rudimentäre Implementierung der Maybe Monade. Diese benutzt das Sprach-Feature von "Sealed Interfaces", die seit Java 17 unterstützt werden. **TODO: Anhang**

```java
public sealed interface Maybe<T> permits Maybe.Just, Maybe.Nothing {
      static <T> Maybe<T> unit(T value) {
          if (value == null) return new Nothing<>();
          return new Just<>(value);
      }

      <S> Maybe<S> bind(Function<T, Maybe<S>> f);

      // map: (Maybe<A>, A -> B) -> Maybe<B>
      default <S> Maybe<S> map(Function<T, S> f) {
          return bind(value -> Maybe.unit(f.apply(value)));
      }

      // Implementierung der `Just` Variante
      final class Just<T> implements Maybe<T> {
          private final T value;

          public <S> Maybe<S> bind(Function<T, Maybe<S>> f) {
              return f.apply(this.value);
          }
          // weitere Methoden - s. Anhang
      }

      // Implementierung der `Nothing` Variante
      final class Nothing<T> implements Maybe<T> {
          public <S> Maybe<S> bind(Function<T, Maybe<S>> f) {
              return new Nothing<>();
          }
          // weitere Methoden - s. Anhang
      }
}
```

Listing 5: Rudimentäre Implementierung der Maybe Monade

Dieses Interface implementiert beide Methoden einer Monade. Die Implementierung von `bind` als Methode eines Objektes ermöglicht die Nutzung der Maybe Monade durch das aneinander-ketten von `bind` aufrufen wie folgt:

```java
int val = 4;
Maybe<String> result = Maybe.unit(val)
    .bind(x -> Maybe.unit(x - 2))
    .bind(x -> Maybe.unit(String.valueOf(x)));
assert result.getValue().equals("2");
```

Diese Kette an Operationen verändert erst einen Integer, und konvertiert ihn dann in einen String. Diese Aufgabe ist trvial genug, dass auch ein rein prozedureller Ansatz ohne unvorhergesehene Fehler durchlaufen könnte. Dies ändert sich allerdings, wenn man die Aufgabe umdreht: Es soll zuerst ein String von stdin eingelesen werden, dann in einen Integer konvertiert und schlussendlich verarbeitet werden.

```
Maybe<Integer> result = Maybe.unit(input.nextLine())
    .bind(s -> s.matches("\\d+")
        ? Maybe.unit(Integer.parseInt(s))
        : Maybe.nothing())
    .bind(x -> Maybe.unit(x - 2));
```

Gibt der Nutzer eine valide Zahl ein (dies wird überprüft durch die `matches` Methode mit einem regulären Ausdruck), enthält `result.getValue()` das korrekte Ergebnis als Integer. Tut der Nutzer dies allerdings nicht, gibt `result.isPresent()` `false` zurück. In diesem Fall könnte man beispielsweise dem Benutzer eine Fehlermeldung anzeigen. Die hier gezeigte Implementierung ist der Übersichtlichkeit halber rudimentär gehalten - in einer tatsächlichen Codebase sollte die Klasse weitere API Methoden enthalten (zum beispiel eine Funktion $\text{getValueOrDefault} : M\langle T\rangle \to T$, die im Falle einer Nicht-Existenz einen Default Wert zurück gibt anstatt eine Exception zu werfen), um Entwicklern eine sinnvolle Nutzung der `Maybe` Klasse mit semantischer Relevanz zu ermöglichen.

3.4. Monaden in der Praxis

Monadische Strukturen finden sich in beinahe allen modernen Programmiersprachen (außer halt Python lol). Die Maybe Monade beispielsweise manifestiert sich in Java als die Klasse `Optional`, und in Rust als der `Option` Datentyp. Ein weiteres Prominentes beispiel ist die sogenannte "IO Monade", die in JavaScript als die Klasse `Promise`, und in Java und Rust als Klasse bzw. der Datentyp `Future` realisiert ist. Die IO Monade enkapsuliert den Side-Effekt, dass ein Wert möglicherweise erst in der Zukunft existiert. Sie findet häufig Anwendung in Web Applikationen, wo Daten über ein Netzwerk geladen werden müssen.

4. Beispiel

- ein Beispiel von Kapitel 2 aufgreifen und mit FP fixen
- idealerweise auf alle Konzepte eingehen

TODO: Vielleicht ist das ein verständliches Beispiel?

OOP:

```
# OOP-Stil
import threading

class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance
        self.lock = threading.Lock()  # Thread-safety nötig

    def deposit(self, amount):
        with self.lock:
            if amount < 0:
                raise ValueError("Cannot deposit negative amount")
            self.balance += amount

    def withdraw(self, amount):
        with self.lock:
            if amount > self.balance:
                raise ValueError("Insufficient funds")
            self.balance -= amount

    def transfer(self, other, amount):
        with self.lock:
            self.withdraw(amount)
            other.deposit(amount)
```

TODO: Das wäre dann der Fix in FP (aber ich glaub definitiv nicht fertig???)

FP:

```
from dataclasses import dataclass
from typing import NamedTuple, Optional

# --- Funktionaler Datentyp ---
@dataclass(frozen=True)
class Account:
    owner: str
    balance: float


# --- Result Monad (vereinfacht) ---
class Result(NamedTuple):
    value: Optional[Account]
```

```
    error: Optional[str]

    @staticmethod
    def ok(value): return Result(value, None)
    @staticmethod
    def err(error): return Result(None, error)


# --- Pure Functions ---
def deposit(account: Account, amount: float) -> Result:
    if amount < 0:
        return Result.err("Negative deposit not allowed")
    new_acc = Account(account.owner, account.balance + amount)
    return Result.ok(new_acc)

def withdraw(account: Account, amount: float) -> Result:
    if amount > account.balance:
        return Result.err("Insufficient funds")
    new_acc = Account(account.owner, account.balance - amount)
    return Result.ok(new_acc)

def transfer(a_from: Account, a_to: Account, amount: float) -> Result:
    result = withdraw(a_from, amount)
    if result.error:
        return result
    result2 = deposit(a_to, amount)
    if result2.error:
        return result2
    return Result.ok((result.value, result2.value))
```

TODO: FP Concurrency Beispiel

Concurrency in FP:

```
from concurrent.futures import ThreadPoolExecutor

acc1 = Account("Alice", 1000)
acc2 = Account("Bob", 500)

with ThreadPoolExecutor() as executor:
    future1 = executor.submit(lambda: deposit(acc1, 100))
    future2 = executor.submit(lambda: withdraw(acc2, 50))
    r1, r2 = future1.result(), future2.result()

print(r1, r2)
```

5. ggf. Comparison mit anderen Paradigmen (OOP & Procedural)

6. Zusammenfassung

Die vorliegende Hausarbeit hat die grundlegenden Prinzipien und Konzepte der funktionalen Programmierung beleuchtet und ihre Relevanz in der modernen Softwareentwicklung aufgezeigt. Im Gegensatz zur imperativen oder objektorientierten Programmierung, die oft auf veränderlichen Zuständen und Seiteneffekten basiert, stellt die funktionale Programmierung reine Funktionen und unveränderliche Daten in den Mittelpunkt.

Wie im Kapitel zu Kapitel 2 *Side Effects* dargelegt, führt die Vermeidung von Seiteneffekten zu deterministischem Code, der leichter zu testen, zu verstehen und zu warten ist. Die klare Trennung von Berechnung und zustandsverändernden Operationen erhöht die Transparenz und Robustheit von Softwaremodulen.

Ein wesentlicher Vorteil der funktionalen Programmierung zeigt sich im Umgang mit Nebenläufigkeit (*Concurrency*). Durch den Verzicht auf *Shared Mutable State* entfallen typische Fehlerquellen wie Race Conditions oder Deadlocks, die in der objektorientierten Programmierung oft komplexe Synchronisationsmechanismen erfordern.

Darüber hinaus wurden mächtige Abstraktionswerkzeuge wie *Higher-Order Functions*, *Currying* und *Monaden* vorgestellt. Diese Konzepte ermöglichen es, wiederkehrende Muster elegant zu lösen und Seiteneffekte – die in realen Anwendungen unvermeidbar sind – kontrolliert und typsicher zu handhaben.

Abschließend lässt sich festhalten, dass die funktionale Programmierung nicht nur ein theoretisches Konstrukt ist, sondern praktische Lösungen für aktuelle Herausforderungen der Softwareentwicklung bietet. Auch wenn der reine funktionale Stil eine Umgewöhnung erfordert, finden sich viele dieser Konzepte mittlerweile auch in populären Multi-Paradigmen-Sprachen wieder, was die wachsende Bedeutung dieses Ansatzes unterstreicht.

Literaturverzeichnis

[1] B. Milewski, *Category Theory for Programmers*. 2019. Accessed: Dec. 08, 2025. [Online]. Available: http://www.blurb.com/b/9621951-category-theory-for-programmers-new-edition-hardco

[2] E. Moggi, “Notions of computation and monads,” *Information and Computation*, vol. 93, no. 1, pp. 55–92, July 1991, doi: 10.1016/0890-5401(91)90052-4.

[3] K. Slechten, “A gentle introduction to monads.” Accessed: Nov. 04, 2025. [Online]. Available: https://kristofsl.medium.com/a-gentle-introduction-to-monads-bc583d41d95