Einsatz funktionaler Programmierung in der Webentwicklung

Wilhelm Büchner Hochschule
Alexander Pann
850621

25. April 2021

Inhaltsverzeichnis

1 Einführung

Funktionale Programmierung ist ein Programmierparadigma, dass auf Lambda Calculus[62] aufbaut, dass in den 1930er Jahren von Alonzo Church entwickelt wurde. Es werden dabei in einem Programm Funktionen angewandt und Funktionen zusammengesetzt. Dabei werden sogenannte Ausdrücke (englisch: Expression) durch Regeln ersetzt. Ein Ausdruck ist laut [48] rekursiv wie folgt definiert:

$$
\begin{aligned}
&< expression > :=< name > | < function > | < application > \\
&\quad < function > := \lambda < name > . < expression > \\
&< application > :=< expression >< expression >
\end{aligned}
$$

Ausdrücke können auch von runden Klammern umgeben sein. Eine Funktion *add*, die im imperativen Paradigma geschrieben ist, kann beispielsweise in Lambda Calculus wie folgt dargestellt werden (vgl. Folie 37 in [47]):

Imperative Schreibweise

```
function add(x, y) {
  if (y == 0) {
    return x
  } else {
    return add(x+1, y-1)
  }
}
```

Funktionale Schreibweise (Lambda Calculus)

```
ADD: λ x.λ y. (ISZERO Y)
  x
  (ADD   (SUCC X))
         (PRED Y))
```

1.1 Konzepte

In der funktionale Programmierung werden einige Begriffe verwendet, die aus der Informatik oder Mathematik stammen und hier kurz erklärt werden.

First-Class-Funktion/ Funktion höherer Ordnung

Eine Funktion höhere Ordnung (englisch: high-order function)[60] ist in der Mathematik und Informatik eine Funktion, die als Parameter ebenfalls Funktionen enthält und/oder als Rückgabewert eine Funktion zurückgibt. Im nachfolgenden Beispiel wird die bekannte map-Funktion dargestellt, an der eine Funktion f übergeben werden kann, die auf alle Elemente einer Liste angewandt wird:

map-Funktion (in Haskell implementiert)

```
map f []     = []
map f (x:xs) = (f x):map f xs

map (\x -> x * x) [1, 2, 3, 4, 5]
```

Rekursion

Bei der rekursiven Programmierung ruft sich eine Funktion selbst wieder auf (sie ist also rekursiv definiert). Es sollte dabei eine Abbruchbedingung geben, da es sonst zu einer (meist ungewollten) Endlosschleife kommt. Die folgende reverse-Funktion ist beispielsweise rekursiv definiert und dreht die Reihenfolge der Listenglieder um:

reverse-Funktion (in Haskell implementiert)

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse' xs ++ [x]

reverse [1, 2, 3, 4, 5] -- Ausgabe: [5, 4, 3, 2,
   1]
```

Strikte und verzögerte Auswertung

Auswertung bezeichnet den Vorgang, bei dem einem Ausdruck ein Wert zugewiesen wird. Bei der strikten Auswertung wird sofort ein Wert zugewiesen, bei der verzögerten Auswertung wird mit der Wertzuweisung eine gewisse Zeit gewartet. Mit letzt genannteren Technik können beispielsweise endlose Listen evaluiert werden. Im nachfolgenden Code wird der Sieb des Eratosthenes mit Hilfe von Rekursion und endlosen Listen implementiert:

sieve-Funktion (in Haskell implementiert)

```
sieve (p : xs) =
    p : sieve [x | x <- xs, x `mod` p /= 0]
primes = sieve [2..]

main = print $ take 5 primes
-- Ausgabe: 2, 3, 5, 7, 11
```

Typisierung

Das Ziel der Typisierung[7] von Objekten (z.B. Variablen, Funktionen, Objekte der objektorientierten Programmierung) ist das Vermeiden von Laufzeitfehlern. Es wird dabei zwischen typisierten und nicht typisierten Programmiersprachen unterschieden. Im folgenden Code werden bei der Fakultätsfunktion beispielsweise das Argument als Ganzzahl angegeben, damit keine Werte anderen Typen angegeben werden können, die zu einem Laufzeitfehler in der Berechnung führen würden (z.B. die Übergabe einer Zeichenkette):

factorial-Funktion (in Haskell implementiert)

```
factorial :: Integer -> Integer
factorial n = product [1..n]

factorial 5 -- Ausgabe: 120
```

Referenzielle Transparenz

Referenzielle Transparenz bedeutet, dass eine Programmiersprache bestimmte Ersetzungseigenschaften besitzt[51]. Die heutzutage übliche Definition besagt, dass ein Wert abhängig von seiner Umgebung und nicht vom Auswertungszeitpunkt abhängt. Variablen haben dann an verschiedenen Stellen ihres Geltungsbereichs immer den selben Wert. Der folgende Vergleich zeigt das diese Eigenschaft bei imperativen Programmiersprache nicht gültig ist:

Referenzielle Transparenz in Haskell

```
addOne:: Int -> Int
addOne x = x + 1
```

Fehlende referenzielle Transparenz in einer imperativen Programmiersprache

```
int ONE = 1; // globale Variable

int addOne(int x)
{
  return x + ONE;
  // ONE koennte bereits veraendert worden sein
}
```

Rein funktionale Datenstrukturen

Rein funktionale Datenstrukturen[40] sind im Gegensatz zu normalen Datenstrukturen nicht veränderbar. Dies hat zum Vorteil, dass sie vollkommen persistent abgespeichert werden können, threadsicher sind und Objekte schnell kopiert werden können. Verzögerte Auswertungen und die Technik der Memoisierung sind daher laut dem Autor oft notwendig um keine Effizienzeinbußungen zu erleiden.

1.2 Abgrenzung zu anderen Paradigmen

Laut [61] gibt es beinahe 30 nützliche Programmierparadigmen, die hier nicht alle mit funktionaler Programmierung verglichen werden können. In deren Arbeiten beschrieben sie auch, dass die Beziehungen zwischen Programmiersprachen und Paradigmen nicht eindeutig sind. Moderne Programmiersprachen wie z.B. Java oder C++ realisieren ein oder mehrere Paradigmen. Diese Paradigmen wiederum bestehen aus einer Menge von Konzepten. Der Autor teilt in der Abbildung 1 die Paradigmen ein:

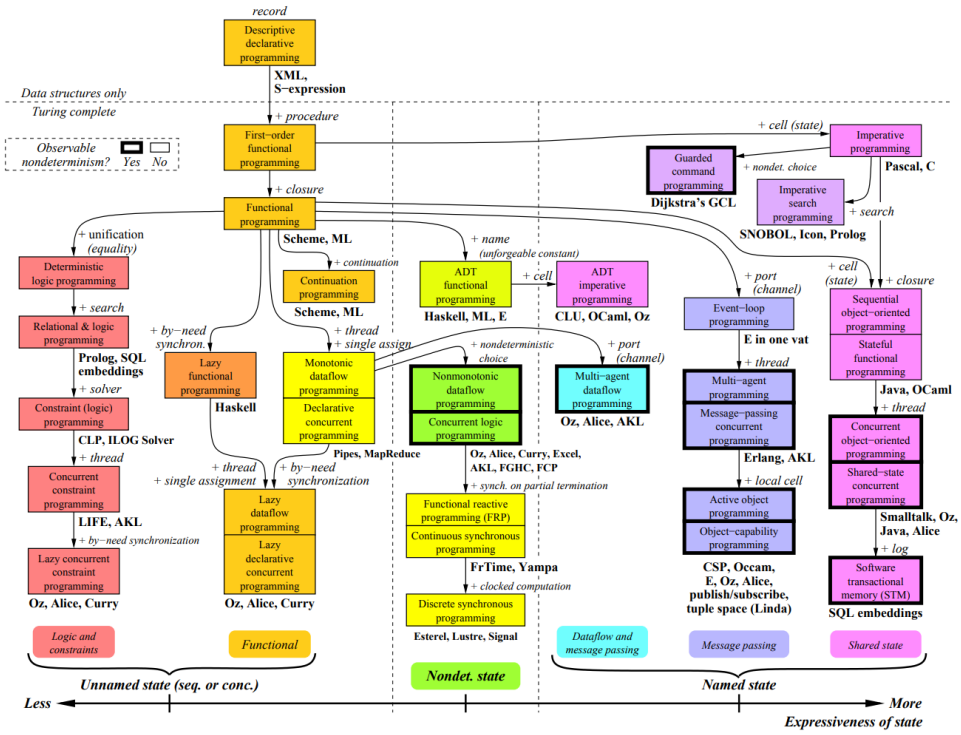

Figure 2. Taxonomy of programming paradigms

Abbildung 1: Taxonomie der Paradigmen (vgl. [61] Figure 2)

Einer der Eigenschaften, mit denen man Paradigmen einordnen kann, ist beobachtbarer Nichtdeterminismus. Dies bedeutet, dass die Ausführung des Programms zur Zeit der Spezifikation noch nicht feststeht, sondern von einem Teil des Laufzeitsystems, nämlich dem Scheduler, abhängig ist. Der Nichtdeterminismus ist sichtbar, wenn der Benutzer verschiedene Ergebnisse sieht, obwohl er die gleiche Konfiguration verwendet. Eine weitere Eigenschaft eines Paradigmas ist die Fähigkeit, Zustände zu unterstützen. Dabei werden Sequenzen von Werten über längere Zeit gespeichert. Zustände können dabei drei Richtungen unterschieden werden: benannt oder unbenannt, deterministisch oder nichtdeterministisch und sequenziell oder gleichzeitig. Wie in Abbildung 1 ersichtlich, ist funktionale Programmierung nicht sehr expressiv in Bezug auf Zustände. Auch ist ersichtlich, dass die funktionale Programmierung von der deklarativen Programmierung abstammt, die im Kontrast zur imperativen Programmierung steht. Bei der deklarativen Programmierung wird die Logik einer Berechnung nicht durch den Kontrollfluss beschrieben[28]. Außerdem kann funktionale Programmierung auch mit nebenläufiger Programmierung kombiniert werden sowie mit Nichtdeterminismus, wie man am Paradigma funktionale reaktive Programmierung sehen kann, dass in einem späteren Kapitel beschrieben wird.

2 Praxis

Nachdem im vorherigen Kapitel die theoretischen Grundlagen gelegt wurden, beschreibt dieses Kapitel die praxisbezogen Aspekte.

2.1 Allgemeine Anwendungsgebiete

Funktionale Programmierung ist ein Forschungszweig im Bereich der Programmiersprachentheorie und wird auch an vielen Universitäten beim Bachelor der Computerwissenschaften gelehrt. Eine Studie über die verwendeten Programmiersprache in der amerikanischen Forschung und Industrie[46] hat gezeigt, dass funktionale Programmierung speziell im akademischen Bereich weit verbreitet ist. Bei der Analyse von gewählten Programmiersprachen für Kurse im Zeitraum 2010-2013 wurden zu 32.772% funktionale Programmiersprachen benutzt, darunter die Sprachen Scheme (Rang 4), Haskell (Rang 6), ML (Rang 7), Lisp(8), OCAML (Rang 10), SML (Rang 13) und CAML (21). Funktionale Programmierung wurde schon in den 1980er Jahren verwendet, als die Firma Ericsson Erlang entwickelte, um damit fehlertolerante Telekommunikationssysteme zu entwickeln[2]. Auch Facebook verwendet funktionale Programmierung[44]. Scheme wurde für einige Anwendungen des frühen Apple Macintosh verwendet[24][20]. OCaml wurde zur finanziellen Analyse verwendet[34]. Haskell wurde bereits in Bereich wie Raumfahrt, Hardwaredesign und Webprogrammierung verwendet[1]. Auch für medizinische Anwendungen[43] wurde dieses Paradigma bereits verwendet. Weitere Sprachen, die in der Industrie bereits verwendet wurden, sind Scala[36], F#[3], Lisp[19], Standard ML[27] sowie Clojure [63].

2.2 Einsatz in der Webentwicklung

2.3 Vorteile

Die Vor- und Nachteile der funktionalen Programmierung sind ein schlecht erforschtes Gebiet, daher können nicht alle Argumente in den nächsten zwei Kapitel objektiv bewertet werden. Einige der folgenden Argumente basieren auf Tatsachen, andere wiederum basieren auf Meinungen von Nutzern verschiedenster Internetplattformen und sollten daher genau überprüft werden.

Reine Funktionen sind deterministisch und haben keine Nebeneffekte. Daher sind sie referenziell transparent. Folglich können Programme leichter verstanden werden, getestet und Fehler gefunden werden. Auch Parallele/nebenläufige Programmierung wird dadurch einfacher.Durch Funktionskomposi-

tion können ein oder mehrere Funktionen zu einer neuen Funktion kombiniert werden um eine neue Aufgabe zu erledigen. Dabei werden keine gemeinsamen Zustände verwendet, sondern unveränderliche Datenstrukturen. Bei gemeinsamen Zuständen ist die Reihenfolge der Funktionsaufrufe entscheidend und kann somit zu Fehlern führen. Zusätzlich ist die Signatur von reinen Funktionssignaturen aussagekräftiger, da diese Methoden keine Nebeneffekte haben können. Durch eine verzögerte Auswertung werden nur die Werte berechnet, die wirklich benötigt werden.

2.4 Nachteile

Reine Funktionen können einfach definiert werden, allerdings ist das Kombinieren schwierig. Die fortgeschrittene Mathematikterminologie schreckt zusätzlich Neulinge ab. Für viele Menschen fühlt sich Rekursion auch nicht natürlich an. Dem hinzu kommt, dass durch das Fehlen der Fähigkeit Daten zu verändern, ein Muster verwendet wird, bei dem Daten während dem Kopieren aktualisiert werden. Außerdem mischen sich reine Funktionen und Eingabe/Ausgabe nicht gut. Fern kann die Verwendung von nicht veränderbaren Werten und Rekursion zu Performanceproblemen führen wie z.B. Arbeitsspeicherauslastung und Geschwindigkeitsprobleme.

2.5 Umstieg vom imperativen Paradigma

In einer Studie [9] aus dem Jahr 2012 wurden anhand von 5 Teilnehmern, die bereits Erfahrung im imperativen Paradigma hatten, die Probleme beobachtet, die beim Umstieg zum funktionalen Paradigma entstehen (die gewählte Sprache war Haskell). Es wurde festgestellt, dass 58% der Probleme zur Kompilierungszeit, 24% zur Laufzeit und 18% beim Debugging auftraten. Die meisten Probleme traten beim Aufrufen von Funktionen auf aufgrund des strikten Typsystems. Auch mussten Funktionen öfter aufgerufen werden als in äquivalenten imperativen Programmen. Bei Fehlern mussten oft Informationsquellen gelesen werden, bevor der Fehler behoben werden konnte. Auch während dem Programmierung musste das Codieren gestoppt werden um Informationen von externen Quellen nachzulesen. Die Studie kam zum Schluss, dass Studenten während der Aufgabenstellung sehr abhängig waren von Informationsquellen. Codebeispiele wurden verwendet um Kompilationsfehler zu beseitigen und die Informationsquellen mit der höchsten Erfolgsraten wurden nur selten referenziert.
Frühere Studien beschäftigen sich mit der Frage, wie man Studenten funktionale Programmiersprachen lernen kann. Die Forschung beschäftigte sich dabei

mit der Integration funktionaler Programmiersprachen in Einführungskursen der Informatik. Befürworter kamen zum Schluss, dass dies zu einem einfacher lernbaren Paradigmenwechsel führt[16] und hilft, besser strukturierte Programme zu schreiben. Dies wurden anhand der effizienten Benutzung von Abstraktion[26] und Fehlerrate während des Entwurfs des Programms[13] gemessen.
Gegner kamen zum Schluss, dass Studenten das Konzept der Funktionen höherer Ordnung [8], den Typ von funktionalen Ausdrücken[13], die Evaluierung von Iteration und rekursiven Funktionen[26][49] und die Bedeutung von Fehlermeldungen [26] nicht verstanden.

3 Funktional reaktive Programmierung

Laut [42], ist die funktional reaktive Programmierung (FRP) ein deklarativer Ansatz um reaktive Anwendung zu schreiben, bei der sich mit der Zeit änderte Werte im Vordergrund stehen, Im Vergleich zu den operationalen Beschreibungen wie auf einzelne Werte zu reagieren ist. Es gibt verschiedene Ansätze, bei denen zwischen einer gemischt diskreten und fortlaufenden Zeitnotation unterschieden wird. Auch werden Eingabe und Ausgabe und andere Effekte unterschiedlich gehandhabt. Diese Beschreibung passiert auf Yampa[10], einer funktional reaktiven domainspezifischen Programmiersprache für Haskell. Es wird hier von System ausgegangen bei denen FRP ein hybrides System beschreibt, dass mit Daten arbeitet, die sich im Laufe der Zeit verändert. Das Hauptkonzept ist hierbei das Signal, dass als Funktion von Zeit nach Werten eines Typs gesehen werden kann:

$$Signal\alpha \approx Time \to \alpha$$

Eine Signalfunktion ist eine Funktion von einem Signal zu einem anderen Signal:

$$Signal\alpha\beta \approx Signal\alpha \to Signal\beta$$

Zeit ist kontinuierlich und wird als nicht negative reelle Zahl dargestellt. Signale können auch Eingabedaten wie Tastatureingaben oder Mausbewegungen darstellen. Es wird zwischen zwei Arten von FRP Frameworks unterschieden: klassische FRP und Arrowized FRP. Beim klassischen FRP werden Programme mit Hilfe von Signalen oder ähnlich Notation strukturiert und repräsentieren somit zeitabhängige Daten. Programmen mit Arrowized FRP werden mit kau-

salen Funktionen zwischen Signalen oder Signalfunktionen definiert, die mit der externen Welt auf oberste Ebene verbunden sind. Arrowized FRP erlauben modularen, deklarativen und effizienten Code und trennen auch Eingaben und Ausgaben vom FRP-Code selbst (referentielle Transparenz).

4 Funktionale Programmierung in nicht funktionalen Sprachen

Viele Allzwecksprachen wie z.B: Java, C++ oder Python unterstützen teilweise funktionale Programmierung[64][32][29]. In diesem Kapitel werden einige Konzepte anhand der Programmiersprache Java erklärt.

4.1 First-Class-Funktionen und Funktionen höherer Ordnung

Diese beiden Konzepte werden oft mit Hilfe von sogenannten Lambda Ausdrücken umgesetzt. Dies sind anonyme Funktionen, die als unabhängiger Block im Code herumgereicht werden können (z.B. als Argument für andere Funktionen). Sie werden in manchen Programmiersprachen auch Closures genannt. Der folgende Code verwendet einen Lambda-Ausdruck, um alle Elemente einer Liste auszugeben:

Lambda in Java-Code

```
List<String> wochentage=new
   ArrayList<String>();
wochentage.add("Montag");
wochentage.add("Dienstag");
wochentage.add("Mittwoch");
wochentage.add("Donnerstag");
wochentage.add("Freitag");
wochentage.forEach(
    (tag)->System.out.println(tag)
);
```

4.2 Streams

Streams wurden in Java eingeführt um Operationen auf Arrays und Listen zu erlauben. Ähnliche Konzepte können auch in anderen Programmiersprachen gefunden werden. Ein Stream repräsentiert dabei eine Sequenz von Elemen-

ten. Operationen geben als Rückgabewert wieder einen Stream zurück, sodass Operationen aneinandergereiht werden können. Das folgende Beispiel zeigt wie Streams im Code geschrieben werden:

Streams in Java

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

4.3 Reine Funktionen

Reine Funktionen haben keine Nebeneffekte und verändern somit auch keine Zustände. Das folgende Beispiel filtert eine Liste von Namen ohne die Liste dabei zu verändern:

Reine Funktionen in Java

```
List<String> result =
    names.stream()
    .filter(name -> name.equals("Max"))
    .map(String::toLowerCase)
    .collect(java.util.stream.Collectors.toList());
```

4.4 Immutabilität

Immutabilität ist eine der Grundprinzipien der funktionalen Programmierung. Die meisten Programmiersprache haben einige sogenannte persistente Datenstrukturen in die Sprache eingebaut. Zu den bekanntesten Strukturen, die nicht veränderbar sein können, zählen Zeichenketten, Listen, Mengen und Maps.

4.5 Referenzielle Transparenz

Referenzielle Transparenz bedeutet, dass eine Funktion immer die gleichen Werte zurückgeben muss, wenn sie mit den gleichen Argumenten aufgerufen wird. Funktionen, die beispielsweise auf unveränderbaren Zeichenketten aufgerufen werden, erfüllen diese Forderung:

```
"hallo".replace('a', 'e')
```

5 Funktionale Programmiersprachen zur Webentwicklung

Die in diesem Kapitel vorgestellten Programmiersprachen sind teilweise keine reinen funktionalen Programmiersprachen sondern unterstützen mehrere Paradigmen. Sie können alle zur Web-Front-End-entwicklung verwendet werden und auch fast alle zur Back-End-Entwicklung. Da es eine sehr große Menge an Programmiersprachen gibt, stellt dieses Kapitel nur die bedeutendsten Sprachen vor.

5.1 Elm

Elm[12] ist eine statische, stark typisierte Programmiersprache zur Erstellung von graphischen Benutzeroberflächen im Webbrowser. Sie wurde 2012 von Evan Czaplicki entwickelt. Sie ist rein funktional und verursacht fast keine Laufzeitfehlerausnahmen in der Praxis. Sie baut auf Arrowized FRP auf und ist auch mit HTML, CSS und JavaScript kompatibel. Elmprogramme basieren auf der Model-View-Update-Architektur[17]:

Model der Zustand der Applikation

View eine Funktion, die das Modell in HTML umwandelt

Update eine Funktion, die das Modell basierend auf Nachrichten aktualisiert

5.2 Reason ML

Reason[15] ist eine funktionale, stark typisierte Programmiersprache, die von Jordan Walke 2016 designed wurde. Sie ist eine Syntaxerweiterung der Programmiersprache OCaml. Sie interagiert mit existierenden JavaScriptbibliotheken und unterstützt speziell das Front-End-Framework React. Sie kann zu OCaml oder JavaScript transpiliert werden.

5.3 ClojureScript

ClojureScript[31] ist ein Compiler, der eine Clojure[21] ähnliche Sprache nach JavaScript übersetzt. Sie wurde basierend auf Clojure entwickelt, einem stark

typisierten, dynamischem Lisp-Dialekt, der von Rich Hickey 2007 entwickelt wurde.

5.4 F#

F#[50] ist eine stark typisierte Programmiersprache, die mehrere Paradigmen unterstützt, darunter imperative, objektorientierte und funktionale Methoden. Sie wurde von der F# Software Foundation, Microsoft und der OpenSource-Community entwickelt. Sie gehört zur *Meta Language*-Familie und kann mit dem Compiler Fable zu JavaScript transpiliert werden.

5.5 OCaml

OCaml[35] ist eine Programmiersprache die mehrere Paradigmen unterstützt (funktional, imperativ, modular, objektorientiert). Sie erweitert den Caml-Dialekt von ML mit objektorientierten Features. Sie wurde 1996 von Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy und Ascánder Suárez entwickelt. Es kann mit einem Transpiler JavaScript-Code erzeugt werden.

5.6 Haskell

Haskell[30] ist eine statisch typisierte, rein funktionale Programmiersprache mit Typinterferenz und verzögerter Auswertung. Sie wurde 1990 von Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, Simon Peyton Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman und Philip Wadler entwickelt. Sie kann mit verschiedenen Compilern wie dem GHCJS oder UHC zu JavaScript transpiliert werden. Eine spezielle Implementierung von Haskell ist Haste, die ebenfalls nach JavaScript transpiliert.

5.7 PureScript

PureScript ist eine stark typisierte, rein funktionale Programmiersprache, die 2013 von Phil Freeman entwickelt wurde. Sie ist nach Haskell modelliert, verfolgt eine strikte Auswertung und kompiliert nach JavaScript. Sie wurde als Alternative zu den Versuchen entwickelt, einen Compiler für Haskell zu schreiben, der den Code nach JavaScript transpilieren soll.

5.8 Scala

Scala[39] ist eine statisch streng typisierte, funktionale und objektorientierte Programmiersprache, die von Martin Odersky designed wurde und vom Programming Methods Laboratory of École polytechnique fédérale de Lausanne entwickelt wurde. Sie kann mit Scala.js zu JavaScript transpiliert werden.

5.9 JavaScript

JavaScript ist eine Event gesteuerte, funktionale und imperative Programmiersprache, die die ECMAScript-Spezifikation[14] implementiert. Sie zählt zu den Haupttechnologien des Webs und dient auch als Zielsprache für die meisten Programmiersprachen in diesem Kapitel. Sie ist dynamisch typisiert und wurde 1995 von Brendan Eich entwickelt.

5.10 Elixir

Elixir[65] ist eine dynamische, funktionale, nebenläufige Programmiersprache, die auf der BEAM Virtual Machine läuft und eine Implementierung der Programmiersprache Erlang ist. Sie wurde 2011 von José Valim entwickelt und kann zum Teil mit den Compiler ElixirScript nach JavaScript umgewandelt werden.

5.11 Kotlin

Kotlin[25] ist eine statisch typisierte Programmiersprache, die 2011 von der Firma JetBrains entwickelt wurde. Sie unterstützt die folgenden Paradigmen: objektorientiert, funktional, imperativ, Block strukturiert, deklarativ, generisch, reflektiv, nebenläufig. Kotlin wurde hauptsächlich für die Java Virtual Machine entworfen, kann jedoch für die verschiedensten Plattformen verwendet werden. Es steht auch ein JavaScript-Transpiler zur Verfügung.

6 Funktionale Tools, Datenstrukturen und Bibliotheken in JavaScript

Die folgenden Tools, Datenstrukturen und Bibliotheken wurden aufgrund der Anzahl an Sternen auf der populären Entwicklerseite Github[4] gewählt, die die Popularität einer Software wiederspiegeln. Es werden nur Projekte mit mindestens 1000 Sterne aufgelistet. Um die Länge der Liste nicht zu sprengen, werden funktional reaktive Bibliotheken nicht behandelt.

6.1 Ramda

Ramda[58] ist eine funktionale Standardbibliothek für JavaScript, die rein funktional aufgebaut wird. Immutabilität und effektfreihe Funktionen stehen im Vordergrund. Funktion in Ramda sind gecurried, dh. Funktionen mit mehreren Argumenten werden in eine Reihe von Funktionen mit jeweils einem Argument umgewandelt. Durch diese Eigenschaft können viele Funktionen aneinandergereiht werden und so als Pipeline verwendet werden. Als Datenstrukturen werden einfache JavaScript-Objekte verwendet. Performance steht auch im Vordergrund der Implementation.

6.2 Folktale

Folktale[41] ist eine funktionale Bibliothek für JavaScript und TypeScript. Zusätzlich zur Funktionskomposition fokusiert sich die Bibliothek auf die bessere Fehlerbehandlung durch die Typen *Maybe* und *Result*. Nebenläufigkeit wird durch den Typ Task implementiert, bei dem asynchrone Aktionen kombiniert werden können die auch automatischen Ressourcenmanagment unterstützen.

6.3 Iodash/FP

Iodash/FP[56] ist die funktionale Variante der Javascriptbibliothek Iodash, die den Umgang mit verschieden Datentypen erleichtern solle. Sie fokussiert sich auf das iterieren von Arrays, Objekten und Zeichenketten, das Manipulierung und Testen von Werten sowie das Erstellen von Kompositionsfunktionen. Funktionen in Iodash/FP sind automatisch curried und können auch einzeln als Modul importiert werden.

6.4 101

101[33] ist eine JS-Hilfsbibliothek, die funktionale Methoden unterstützt. Sie verwendet Standard JavaScript-Methoden außer für das funktionale Paradigma. Jede Methode kann auch einzeln importiert werden.

6.5 Most

Most[11] ist ein Akronym und steht für Monadic streams for reactive programming. Most.js ist eine Framework für reaktive Programmierung. Es können dabei asynchrone Operation auf Ströme von Werten angewandt werden wie z.B. WebSocketnachrichten oder DOM-Events. Es werden auf zeitabhängige Werte wie z.B der aktuelle Wert eines Eingabefelds unterstützt ohne dabei auf

die Gefahren von Nebeneffekten und veränderbaren gemeinsamen Zuständen geplagt zu sein.

6.6 fp-ts

fp-ts[6] ist eine typisierte funktionelle Bibliothek, die in TypeScript geschrieben ist. Sie wurde inspiriert durch Haskell, PureScript und Scala und versucht bekannte Muster und Abstraktionen den Entwicklern anzubieten, die von funktionalen Programmiersprachen bekannt sind.

6.7 Immutable-js

Immutable-js[55] bietet viele nicht veränderbare persistente Datenstrukturen an. Dazu zählen: List, Stack, Map, OrderedMap, Set, OrderedSet und Record. Es wird auch eine verzögerte evaluierte Variante von Seq (Sequenz) angeboten.

6.8 Immer

Immer[54] ist eine Bibliothek, die es erlaubt mit nicht veränderbaren Zuständen zu arbeiten. Sie verwendet den copy-on-write-Mechanismus. Hierbei werden alle Änderungen an einem temporären Zustände angewandt, welcher ein Stellvertreter des aktuellen Zustandes ist. Nachdem alle Mutationen abgeschlossen sind, wird der nächste Zustand abhängig von den Mutation produziert, die am temporären Zustand vorgenommen wurden.

6.9 Mori

Mori[38] bildet eine Brücke zu ClojureScript persistenten Datenstrukturen und unterstützt auch APIs für normales JavaScript. Alle Morikollektionen unterstützen die ES6 basierende Iteration.

6.10 Baobab

Baobab[45] ist eine Bibliothek für JavaScript und TypeScript und unterstützt persistente und nicht veränderbare Datenbäume, die sogenannte Cursor unterstützen, mit denen Daten abgefragt und verändert werden können. Sie wurde inspiriert von funktionalen Zippern[23] sowie Cursors in der Programmiersprache Om.

6.11 List

List[18] bietet eine rein funktionale Alternative zu Arrays. Durch die Nichtveränderbarkeit der Listen erhöht sich die Sicherheit. Dadurch das Mutationen nicht erlaubt sind, kann die Performance teilweise stark optimiert werden. Die Funktionen der Bibliothek sind auch gecurried und können aneinandergereiht werden.

6.12 Fantasy Land

Fantasy Land[52] implementiert bekannte algebraische Strukturen für JavaScript: Setoid, Ord, Semigroupoid, Category, Semigroup, Monoid, Group, Filterable, Functor, Contravariant, Apply, Applicative, Alt, Plus, Alternative, Foldable, Traversable, Chain, ChainRec, Monad, Extend, Comonad, Bifunctor und Profunctor.

6.13 Sanctuary

Sanctuary[59] ist eine funktionale JavaScriptbibliothek, die durch Haskell und PureScript inspiriert wurde. Sie ist strikter als Ramda und bietet ähnliche Funktionen. Es wurden die beiden Datentypen *Maybe* und *Either* implementiert, die auch mit Fantasy Land kompatibel sind.

6.14 Monet.js

Monet.js[57] ist eine weitere funktionale Bibliothek für JavaScript. Sie bietet eine Menge von sogenannten Monaden und andere nützliche Funktionen an.

6.15 Fluture

Fluture[53] bietet eine Kontrollstruktur an, die ähnlich wie Promises, Tasks und Deferreds ist und Futures genannt wird. Sie repräsentiert Werte von erfolgreichen/fehlgeschlagenen und asynchronen Operationen. Sie werden verzögert ausgeführt und verhalten sich wie Monaden.

6.16 Crocks

Crocks[4] ist eine Kollektion von populären algebraischen Datentypen, die in der funktional Programmierung verwendet werden.

7 Fazit

Funktionale Programmierung hat ihren Ursprung in der Forschung und hat inzwischen auch die Welt der Softwareentwicklung als alternatives Softwareparadigma erreicht. Auch wenn das Erlernen dieses Paradigmas schwierig ist, besonders wenn ein Umstieg oder Dazulernen von einem anderen Paradigma wie der imperativen Programmierung erfolgt, können funktionale Ansätze auch in modernen Allzweckprogrammiersprachen gefunden werden. Zudem haben sich aus bekannten funktionalen Programmiersprachen wie zum Beispiel Haskell viele neue Programmiersprachen entwickelt, die auch für die Webentwicklung verwenden werden können. Auch gibt es viele Bibliotheken für die Programmiertsprache JavaScript, die als Zielsprache für viele anderen Webprogrammiersprachen dient. Dabei werden teilweise nur bestimmte Teile des funktionalen Paradigmas übernommen wie z.B. dessen verwendete Datenstrukturen oder die Unveränderbarkeit.

8 Zukünftige Entwicklungen

Große Hoffnung bei funktionalen Programmierung wird gelegt in die Verwendung von sogenannten abhängigen Typen[22]. Darunter werden Typen verstanden, die im Gegensatz zu einfachen Typen wie Produkten, Funktionsbereichen oder natürlichen Zahlen, von Werten abhängig sind. Dies lässt sich am Besten an einem Beispiel demonstrieren:

append-Funktion in Idris

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append []         ys = ys
append (x :: xs) ys = x :: append xs ys
```

Die hier implementierte append-Funktion hängt eine Liste an die andere Liste an. Der verwendete Datentyp *Vec* hat zwei Argumente: n ist die Längen der Liste und a der Typ. In der Funktion selbst werden beim Rückgabewert die Längen der beiden Listen addiert und als neue Länge des Rückgabetypes definiert.
Eine Programmiersprache die abhängigen Typen unterstützt ist beispielsweise die rein funktionale Programmiersprache Idris[5], die der Programmiersprache Haskell ähnlich ist und auch für die Webentwicklung verwendet werden kann. Sie wurde 2007 von Erwin Brady entwickelt.

Auch der vermehrte Einsatz im Bereich Künstliche Intelligenz (KI) und maschinelles Lernen wäre denkbar. Beispielsweise könnte funktionale Program-

mierung in semantischen Netzen und Web-Ontologien angewandt werden, die KI-basiert sind. Die Web Ontology Language[37] unterstützt dieses Paradigma bereits seit über 10 Jahren.

Literatur

[1] anonym. Haskell in industry, 2020. URL `https://wiki.haskell.org/Haskell_in_industry`.

[2] Joe Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.

[3] Johan Astborg. *F# for Quantitative Finance*. Packt Publishing Ltd, 2013.

[4] Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[6] Giulio Canti. fpts. `https://github.com/gcanti/fp-ts`.

[7] Luca Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28(1): 263–264, 1996.

[8] Manuel MT Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. 2004.

[9] Christopher Chambers, Sheng Chen, Duc Le, and Christopher Scaffidi. The function, and dysfunction, of information sources in learning functional programming. *Journal of Computing Sciences in Colleges*, 28(1): 220–226, 2012.

[10] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, 2003.

[11] User "cujojs". most. `https://github.com/cujojs/most`.

[12] Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 30, 2012.

[13] Alireza Ebrahimi. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41 (4):457–480, 1994.

[14] Draft Standard ECMA. Ecmascript language specification. 1999.

[15] J David Eisenberg. *Web Development with ReasonML: Type-safe, Functional Programming for JavaScript Developers*. Pragmatic Bookshelf, 2019.

[16] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[17] Simon Fowler. Model-view-update-communicate: Session types meet the elm architecture. *arXiv preprint arXiv:1910.11108*, 2019.

[18] User "funkia". list. `https://github.com/funkia/list`.

[19] Paul Graham. Beating the averages, 2005.

[20] Anne. Hartheimer. Programming a text editor in macscheme+toolsmith, 1987. URL `https://web.archive.org/web/20110629183752/http://www.mactech.com/articles/mactech/Vol.03/03.1/SchemeWindows/index.html`.

[21] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, 2008.

[22] Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.

[23] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[24] John Hughes. Multitasking and macscheme, 1987. URL `http://www.mactech.com/articles/mactech/Vol.03/03.12/Multitasking/index.html`.

[25] Dmitry Jemerov and Svetlana Isakova. *Kotlin in action*. Manning Publications Company, 2017.

[26] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *JFP*, 3:49–65, 1993.

[27] Timo Lilja. Functional programming in communications security. In *Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–2, 2007.

[28] John W Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[29] Steven Lott. *Functional python programming*. Packt Publishing Ltd, 2015.

[30] Simon Marlow et al. Haskell 2010 language report. *Available on: http://www. haskell. org/onlinereport/haskell2010*, 2010.

[31] Mark McGranaghan. Clojurescript: Functional programming for javascript platforms. *IEEE Internet Computing*, 15(6):97–102, 2011.

[32] Brian McNamara and Yannis Smaragdakis. Functional programming in c++. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 118–129, 2000.

[33] Tejesh Mehta. 101. `https://github.com/tjmehta/101`.

[34] Yaron Minsky and Stephen Weeks. Caml trading-experiences with functional programming on wall street. *Journal of Functional Programming*, 18(4):553, 2008.

[35] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. Ö'Reilly Media, Inc.", 2013.

[36] Lee Momtahan. Scala at edf trading: implementing a domain-specific language for derivative pricing with scala. In *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming As a Means, Not an End*, page 1, 2009.

[37] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. Owl 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27(65):159, 2009.

[38] David Nolen. mori. `https://github.com/swannodette/mori`.

[39] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[40] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[41] User "origamitower". folktale. `https://github.com/origamitower/folktale`.

[42] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. *ACM SIGPLAN Notices*, 51(12):33–44, 2016.

[43] Christian L Petersen, Matthias Gorges, Dustin Dunsmuir, Mark Ansermino, and Guy A Dumont. Experience report: functional programming of mhealth applications. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 357–362, 2013.

[44] Christopher Piro and Eugene Letuchy. Functional programming at facebook. In *Commercial Users of Functional Programming Conference*, volume 10, 2009.

[45] Guillaume Plique. baobab. `https://github.com/Yomguithereal/baobab`.

[46] Ben Arfa Rabai et al. Programming language use in us academia and industry. *Informatics in Education*, 14(2):143–160, 2015.

[47] N. Richards. The lambda calculus and the javascript, 2012. URL `https://www.slideshare.net/normanrichards/the-lambda-calculus-and-the-javascript`.

[48] Raúl Rojas. A tutorial introduction to the lambda calculus. *arXiv preprint arXiv:1503.09060*, 2015.

[49] Judith Segal. Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, 22(5):385–411, 1994.

[50] Chris Smith. *Programming F#: A comprehensive guide for writing simple code to solve complex problems*. Ö'Reilly Media, Inc.", 2009.

[51] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

[52] Open Source. fantasy-land. `https://github.com/fantasyland/fantasy-land`, .

[53] Open Source. Fluture. `https://github.com/fluture-js/Fluture`, .

[54] Open Source. immer. `https://github.com/immerjs/immer`, .

[55] Open Source. immutable-js. `https://github.com/immutable-js/immutable-js`, .

[56] Open Source. lodash/fp. `https://github.com/lodash/lodash/wiki/FP-Guide`, .

[57] Open Source. monet.js. `https://github.com/monet/monet.js`, .

[58] Open Source. Ramda. `https://github.com/ramda/ramda`, .

[59] Open Source. sanctuary. `https://github.com/sanctuary-js/sanctuary`, .

[60] Simon Thompson. *Type theory and functional programming.* Addison Wesley, 1991.

[61] Peter Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:616–621, 2009.

[62] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

[63] Akhil Wali. *Clojure for machine learning.* Packt Publishing Ltd, 2014.

[64] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming.* Ö'Reilly Media, Inc.", 2014.

[65] Brock Wilcox. The elixir programming language. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 49–60, 2008.