Technische Hochschule Mittelhessen

Kernel-Architekturen in Programmiersprachen

Projektarbeit

Eine Analyse der grundlegenden Übersetzungsmechanik der Sprache PureScript

Autor:
Daniel Kreck

Dozent:
Prof. Dr. D. Herzberg

26. Juli 2018

Inhaltsverzeichnis

1 Zielsetzung und Vorgehen

In diesem Kapitel wird die Projektaufgabe des Moduls Kernel-Architekturen in Programmiersprachen, im Fachbereich MNI der Technischen Hochschule Mittelhessen, beschrieben.

1.1 Kontext

Im Rahmen des Moduls Kernel-Architekturen in Programmiersprachen beschäftigen wir uns mit Sprachen dieser Art. Dies sind Sprachen, welche aus einem kleinen Sprachkern aufgebaut und meist funktionaler Natur sind.

Der Dozent Prof. Dr. Dominikus Herzberg entwickelte zu Lehrzwecken die Kernel-basierte konkatenative Sprache Consize in der Programmiersprache Clojure. Diese galt es zum Einstieg in das Modul in einer Sprache der eigenen Wahl zu reimplementieren. Prüfungsleistung kann es sein, Consize um weitere Sprachfunktionalitäten zu erweitern oder eine Analyse einer anderen Kernel-basierten Programmiersprache vorzunehmen.

1.2 Problem

Als individuelle Projektaufgabe soll sich mit einer Kernel-basierten Sprache beschäftigt und ihr Übersetzungsmechanismus beschrieben werden. Das dient dem Zweck zu ergründen wie dieser aussieht und aufgebaut ist, welche Sprachkonstrukte zur Abbildung genutzt werden und damit auch zu erforschen, wie eine sich in der Praxis befindliche Kernel-basierte Sprache strukturiert ist und im Kern funktioniert. Es geht also nicht um weniger, als zu erforschen, wie gut man sich einer fremden Sprache unter einem potentiell fremden Paradigma nähern kann, wenn man den Zielcode analysiert, der auf ein bereits bekanntes Paradigma aufsetzt.

1.3 Aufgabe

Dem Problem wird mit der Programmiersprache Purescript, welche nach Javascript kompiliert, begegnet. Ziel ist es Muster in der Übersetzung von PureScript nach JavaScript zu erkennen sowie zu beschreiben und sich dadurch der Sprache PureScript zu nähern.

Der Beschreibungsprozess ist wie folgt aufgebaut. Die aufgegriffenen Spachelemente von Purescript werden erklärt und damit wird kein Wissen in der Sprache selbst vorausgesetzt, um sie zu verstehen. Im Anschluss wird die Übersetzung nach Javascript aufgezeigt und diese Abbildung beschrieben. Das hat zur Folge, dass die Kapitel thematisch aufeinander aufbauen müssen. Als erstes steht bspw. das Modulkonzept von PureScript, gefolgt von Funktionen, ohne Typdeklarationen einzuführen. Diese werden erst genutzt, wenn Typen im nächsten Kapitel eingeführt wurden. Allgemeine Begriffe oder Definitionen der Informatik, insbesondere der Softwareentwicklung, werden allerdings vorausgesetzt.

Um im Vorgehen systematisch zu sein, wird sich anhand der Gliederung des Buches „PureScript by Example“ von Phil Freeman orientiert. Die größten und wichtigsten Themenblöcke bilden in dieser Ausarbeitung die Kapitel. Kleinere oder von ihrer Abbildung her uninteressantere Themenbereiche fließen entweder innerhalb eines Kapitels am Rande ein oder werden nicht betrachtet. Ein Beispiel für einen wichtigen Themenblock, der ein Kapitel aufspannt, sind Typen, untergliedert in einfache Typen, Algebraische Typen und Typklassen. Ein Beispiel für einen Themenbereich, der nur am Rande einfließt, wäre bspw. Pattern Matching.

2 | Module und Bezeichner

Ein PureScript Programm ist eine Sammlung von Modulen. Module definieren Namensräume und sind wiederum eine Sammlung von Funktionen, Typen sowie Typklassen für einen bestimmten Zweck. In PureScript werden Module durch das `module` Schlüsselwort beschrieben. Ein Modul kann über `import` weitere Funktionalität aus anderen Modulen importieren und nutzen sowie über eine Exportliste entscheiden, welche der eigenen Funktionalität für externe Module sichtbar ist [4]. Durch Listing 2.1 wird dieses Konzept am einfachen Beispiel der Identitätsfunktion demonstriert. Im Gegensatz zu Haskell wird die Prelude in PureScript wie jedes andere Modul behandelt und muss damit explizit importiert werden[1]. Sie definiert elementare Operationen, wie z.B. bereits die Addition oder Multiplikation, sofern es sich um die standardmäßige Prelude handelt, da dieses Konzept es ebenso ermöglicht eine eigene Prelude zu definieren [3].

```
1 module Main (exportedIdentity) where
2
3 import Prelude
4
5 exportedIdentity x = x
6 notExportedIdentity x = x
```

Listing 2.1: Einfache Moduldefinition mit Imports und Exports in PureScript

Das Modul mit der Bezeichnung *Main* beinhaltet zwei Funktionen - zweimal die Definition der Identitätsfunktion. Die eine wird durch die Angabe zwischen Modulbezeichnung und `where` Schlüsselwort, welches einen neuen Codeblock einführt, exportiert. Die andere bleibt lediglich innerhalb des Moduls zugreifbar. Wird keine Exportliste angegeben, werden implizit alle Bestandteile des Moduls exportiert.

In Listing 2.2 ist die Übersetzung des vorangegangenen PureScript-Beispiels abgebildet. Man erkennt durch die Direktive in der ersten Zeile, dass JavaScript im Strict Mode ausgeführt werden soll. In diesem ist es bspw. nicht möglich undeklarierte Variablen zu nutzen [14]. Für den Import und Export von Modulen wird auf die Node.js Funktionen `require` und `exports` zurückgegriffen. Dies bedeutet, dass Module JavaScript Objekte werden, welche die zu exportierenden Member des Moduls kapseln [1].

[1] In diesem Fall ist der Import nicht erforderlich, da keine Funktionalitäten aus der Prelude genutzt werden - ihr Import dient einzig zur Demonstration des Importmechanismus von Modulen

```
"use strict";
var Prelude = require("../Prelude");

var notExportedIdentity = function (x) { return x; };
var exportedIdentity = function (x) { return x; };

module.exports = {
  exportedIdentity: exportedIdentity
};
```

Listing 2.2: Übersetzung der einfachen Moduldefinition nach JavaScript

Nach dem Übersetzungsvorgang liegt im jeweiligen Ausgabeverzeichnis ein Verzeichnis, welches den Modulnamen, hier *Main*, trägt und in diesem mindestens zwei weitere Dateien - eine JavaScript-Datei (index.js) mit dem entsprechenden Zielcode und eine Datei im JSON-Format (externs.json), die Metainformationen enthält. Das übersetzte Prelude-Module befindet sich mit gleicher Struktur ebenfalls im Ausgabeverzeichnis, was dessen Import in Listing 2.2 durch die Pfadangabe bereits andeutet.

PureScript Funktionen werden zu JavaScript Funktionen übersetzt, wobei deren ursprüngliche Bezeichnungen, sofern keine Namenskonflikte entstehen, versucht werden beizubehalten [7, S. 10]. Nutzt man zum Beispiel reservierte Bezeichner in JavaScript als Bezeichner in PureScript werden sie durch das Voranstellen zweier Dollar Symbole maskiert. Wird eine Bezeichnung in PureScript gewählt, welcher in JavaScript unzulässig ist, wird sie durch ein einfaches Dollar Symbol maskiert. In Listing 2.3 und 2.4 sind diese beiden Fälle exemplarisch aufgeführt [7, S. 144].

```
null = []
example' = 100
```

Listing 2.3: Name Generation PS

```
var $$null = [];
var example$prime = 100;
```

Listing 2.4: Name Generation JS

3 Kommentare

In PureScript gibt es zwei Arten von Kommentaren - Zeilen- sowie Blockkommentare. Der PureScript Compiler ignoriert diese nicht wie üblich während der Übersetzung, sondern wandelt sie in ihr JavaScript-Pendant um und bettet sie in den Zielcode ein. Leerzeichen werden ebenfalls eins zu eins übernommen, wie in Listing 3.1 und 3.2 dargestellt.

```
-- single line comment

{- multiline comment which can
  continue for many lines
-}
```

Listing 3.1: Kommentare PS

```
// single line comment

/**
*  multiline comment which can
*    continue for many lines
*/
```

Listing 3.2: Kommentare JS

Kommentare, die mit einem Pipe-Zeichen beginnen, werden als Dokumentation gesehen und erscheinen in Tools wie Pursuit[1]. Im Gegensatz zu Haskell muss bei mehreren Zeilenkommentaren untereinander, jede Zeile, die als Dokumentation dienen soll, mit dem Pipe-Zeichen beginnen [5].

[1]Pursuit ist ein Webdienst, der es einem erlaubt die API Dokumentation von PurseScript nach Paketen, Modulen, Funktionen usw. zu durchsuchen - https://pursuit.purescript.org/

4 | Funktionsdefinition und -applikation

Funktionen in PureScript werden auf JavaScript Funktionen abgebildet und können auf oberster Ebene eines Moduls definiert werden, indem der Funktionsbezeichner gefolgt von beliebig vielen Funktionsargumenten vor dem Gleichheitszeichen und der Funktionskörper hinter diesem geschrieben werden [7, S. 17]. Als Beispiel diene hierzu die Funktion *addTwo* in Listing 4.1. Alle weiteren Ausführungen beziehen sich ebenfalls auf dieses und Listing 4.2 (gekürzt um Strict Mode Direktive und Exports), welches dessen Übersetzung nach JavaScript aufzeigen soll.

```
1  module Main where
2
3  import Prelude ((+))
4
5  addTwo a b = a + b
6  constant = addTwo 2 3
```

Listing 4.1: Funktionen in PureScript

```
1  var Data_Semiring = require("../Data.Semiring");
2  var Prelude = require("../Prelude");
3
4  var addTwo = function (dictSemiring) {
5    return function (a) {
6      return function (b) {
7        return Data_Semiring.add(dictSemiring)(a)(b);
8      };
9    };
10 };
11
12 var constant = addTwo(Data_Semiring.semiringInt)(2)(3);
```

Listing 4.2: Funktionen in JavaScript

Im Vergleich zu ihrem Gegenstück in JavaScript nehmen Funktionen in PureScript immer nur ein Argument an - wie auch in Haskell [6, 8, Kap. Higher order functions, Abschn. Curried functions]. Unsere Funktion *addTwo*, welche offensichtlich zwei Argumente

anzunehmen scheint, ist ein Beispiel für eine **Curried Function**[1]. Aus diesem Aspekt erschließt sich die Übersetzung nach Javascript in den Zeilen fünf bis neun und auch die schrittweise Funktionsapplikation in der letzten Zeile.

Bleibt zu erläutern wie Operationen wie die Addition, und wie die Funktionsapplikation in JavaScript umgesetzt werden. Ersteres wird auf den Abschnitt „5 Typen und Typklassen“ vertagt, da zuvor Typen und Typklassen eingeführt werden müssen. Für jetzt reicht es aus den Zielcode rund um das Modul *Data.Semiring*[2] aus dem Blickwinkel zu betrachten, dass die Addition bekannterweise eine typabhängige Operation darstellt, was zu dem zusätzlichen Zielcode führt, der die bereits erläuterte Funktionalität umschließt. In diesem Fall bezieht er sich auf den Integer-Typ, weil in der Funktionsapplikation in Zeile 6 bzw. 12 des jeweiligen Listings Ganzzahlen genutzt werden und der Compiler durch Typinferenz die passende typabhängige Übersetzung vorgenommen hat - *Data_Semiring.semiringInt* als typgebundene Berechnungsfunktion in der Funktion *addTwo*.

Die Funktionsapplikation wird in PureScript durch das einfache hintereinanderschreiben (auch Juxtaposition genannt) des Funktionsbezeichners und den Argumenten durchgeführt [5]. Einfache Leerzeichen dienen als Trennsymbol, wie in Zeile sechs des Listings 4.1 zu sehen. Funktionsapplikationen in PureScript werden als Funktionsapplikationen in JavaScript aufgelöst [7, S. 10]. Die Auswertung des Funktionsaufrufs von *addTwo* führt zu einem Wert. Da PureScript eine rein funktionale Programmiersprache ist und nur Funktionen kennt, handelt sich bei *constant* nicht um eine Variable an die der Wert gebunden wird, sondern um eine konstante Funktion.

4.1 Sonderfall Main-Modul und Main-Funktion

PureScript erwartet als Programmeinstiegspunkt ein Modul mit dem Namen *Main*, in dem eine Funktion namens *main* definiert sein muss. Die reine Übersetzung (*pulp build*) tangiert nachfolgende Ausführungen nicht. Sie beziehen sich lediglich auf die Ausführung des Zielcodes (*pulp run*).

Einer Main-Funktion in einem anderweitig benannten Modul wird keine besondere Bedeutung zu Teil. Bei der Ausführung wird ein Main-Modul ohne Main-Funktion beanstandet, ebenso wie dessen komplettes Fehlen. Zusätzlich muss die Main-Funktion entweder vom Typ *Effect.Effect* oder *Control.Monad.Eff.Eff* sein. Alle genannten Fehler können über Schalter ausgeschaltet bzw. ignoriert werden.

Ein korrektes Main-Modul wird standardmäßig zu einer Datei mit der Bezeichnung *output.js*, die sich auf der gleichen Ebene wie das Ausgabeverzeichnis, welche die übersetzten Module enthält, übersetzt. In dieser wird die Main-Funktion in der letzten Zeile gestar-

[1] https://de.wikipedia.org/wiki/Currying

[2] ist an die Algebraische Struktur des Halbringes aus der Mathematik angelehnt - https://de.wikipedia.org/wiki/Halbring_(Algebraische_Struktur)

tet, nachdem alle sonstigen Module definiert worden sind. Sie ist übersetzt als einfacher Funktionsaufruf ohne Argumente [7, S. 10].

4.2 Lessons Learned

- Funktionen in PureScript sind Funktionen in JavaScript
- Durch den Einsatz von Currying nehmen alle Funktionen (syntaktischen Zucker außen vor gelassen) lediglich ein Argument entgegen - für partielle Anwendung ist daher keine spezielle Syntax notwendig
- Die Funktionsanwendung geschieht über die Juxtaposition von Funktion und ihren Argumenten - Klammern dienen nur der Präzedenzsteuerung

5 | Typen und Typklassen

In diesem Kapitel werden wesentliche Bestandteile des Typsystems von PureScript vorgestellt und demonstriert, wie diese zur Laufzeit in JavaScript repräsentiert werden. Alle Typinformationen sind zur Laufzeit nicht mehr vorhanden. Das Typsystem garantiert allerdings, dass der Typ eines Ausdrucks mit seiner Repräsentation zur Laufzeit kompatibel ist, sodass ein vom Compiler akzeptiertes Programm zur Laufzeit nicht aufgrund eines Typfehlers abstürzt [7, S. 144, 148].

5.1 Einfache Datentypen

In PureScript sind drei Datentypen enthalten, die direkt auf JavaScript Typen abgebildet werden. Sie werden im Modul *Prim* definiert, welches implizit von jedem anderen Modul importiert wird. Es handelt sich um folgende Typen, inkl. Angabe eines Literals als Beispiel [7, S. 16]:

1. Number - `1.0`
2. String - `"Ich bin eine Zeichenkette"`
3. Boolean - `true`

Darüber hinaus sind in PureScript weitere Typen fest eingebaut, wie:

- Int - `1`
- Char - `'a'`
- Array - `[1, 2, 3]`
- Record -

```
{ name: "Phil", interests: ["PureScript", "JavaScript"] }
```

- Function (benannt) - `add a b = a + b`
- Function (anonym) - `\a b -> a + b`

PureScript's Datentypen *Int* und *Char* werden in JavaScript letztlich wieder als *Number* und *String* repräsentiert. Arrays in PureScript sind Arrays in JavaScript, allerdings mit

der Einschränkung, dass alle Elemente vom gleichen Typ sein müssen (homogen). Records werden in PureScript in JavaScript's Objekt-Literal-Schreibweise angegeben und sind in JavaScript vom Typ *object* und damit heterogener Natur. PureScript Funktionen, ob benannt oder anonym, sind vom JavaScript Datentyp *function* [7, S. 16–18, 145].

In PureScript gibt es keine Repräsentation von JavaScript's `null` oder `undefined` Werten[1]. Das bedeutet, dass PureScript Ausdrücke, ob nun vom Datentyp *Int*, *String*, *Array*, *Record* usw., zur Laufzeit immer zu einer **Nicht-Null** Repräsentation in JavaScript evaluiert werden [7, S. 145].

5.2 Algebraische Datentypen

Algebraische Datentypen (kurz: ADT) sind ein Bestandteil des Typsystems von PureScript und eng verwandt mit Pattern Matching, da jeder Wertkonstruktor mit seinen Argumenten als Pattern genutzt werden kann. Sie sind ein Weg einen neuen Datentyp zu deklarieren, der aus anderen Typen zusammengesetzt ist. In PureScript werden sie mit dem Schlüsselwort `data` eingeführt, gefolgt von einer Bezeichnung (Typkonstruktor) für den ADT. Nach dem Gleichheitszeichen folgen die Wertkonstruktoren. In Listing 5.1 sind drei Sorten von Algebraischen Datentypen dargestellt [7, S. 57–59].

```
data Person = Person String Int

data Maybe a = Nothing | Just a

data List a = Nil | Cons a (List a)
```

Listing 5.1: Beispiel ADT

Der erste Typ mit dem Namen *Person* hat genau einen Wertkonstruktor mit dem Namen *Person*, der zwei Argumente mit den Typen *String* für den Namen einer Person und *Int* für deren Alter hat. In diesem Fall ist es üblich, dass der Typ- und Wertkonstruktor gleich benannt sind. *Maybe* ist ein Beispiel für einen ADT, dessen Typkonstruktor (*Maybe*) einen Typparameter[2] hat (*a*) und der Alternativen aufspannt, also mehrere Wertkonstruktoren besitzt. Das Pipe-Symbol kann als Oder gelesen werden und trennt die Wertkonstruktoren. Für *Maybe* heißt dies, dass ein Wert vom Typ *Maybe* entweder nichts ist (*Nothing*) oder ein Etwas (*Just*) von dem Typen *a*. *List* ist wie *Maybe* ein ADT aus der PureScript Prelude und sagt aus, dass eine Liste entweder leer ist oder einen Kopf und einen Rest hat, der wieder eine Liste ist. Damit ist *List* ein Beispiel für einen ADT, welcher über sich selbst rekursiv definiert ist [7, S. 57–58].

[1] sind in JavaScript gleich vom Wert, aber unterschiedlich im Datentyp - *null* ist vom Datentyp *object* und *undefined* von *undefined* [9]

[2] das hat zur Folge, dass im Gegensatz zu bspw. Person, wo ein Wert vom Typ Person wäre, ein Wert nicht direkt vom Typ Maybe sein kann, sondern Im Fall von *Nothing* von `Maybe a` und bei *Just* von `a -> Maybe a`

5.2.1 Umsetzung von Algebraischen Datentypen in JavaScript

In diesem Unterabschnitt wird der ADT *List* aus Listing 5.1 zur Erläuterung des Übersetzungsmusters von Algebraischen Datentypen nach JavaScript exemplarisch genutzt. *Person* und *Maybe* folgen dem gleichen Muster, weshalb auf deren Übersetzung in Listing 5.2 verzichtet wird.

```
1  var Nil = (function () {
2    function Nil() {
3
4    };
5    Nil.value = new Nil();
6    return Nil;
7  })();
8
9  var Cons = (function () {
10   function Cons(value0, value1) {
11     this.value0 = value0;
12     this.value1 = value1;
13   };
14   Cons.create = function (value0) {
15     return function (value1) {
16       return new Cons(value0, value1);
17     };
18   };
19   return Cons;
20 })();
```

Listing 5.2: Übersetzung nach JavaScript von ADT's am Beispiel von *List*

In Listing 5.2 ist zu sehen, dass für jeden Wertkonstruktor eines ADT ein neuer JavaScript Objekttyp (Klasse) angelegt wird. Der Compiler erzeugt für Wertkonstruktoren mit Argumenten in JavaScript Objekt-Konstruktoren mit der entsprechenden Anzahl Argumente. Bei *Cons* ist das erste Argument der Kopf und das zweite der Rest. Der Compiler erzeugt für die Objektinstantiierung Hilfsfunktionen, welche eingesetzt werden können, anstatt den *new*-Operator explizit nutzen zu müssen. Bei Konstruktoren mit keinen Argumenten, wie bei *Nil*, wird ein Attribut *value* angeboten, um eine wiederholte Neuinstantiierung zu vermeiden. Bei Konstruktoren mit mindestens einem Argument, wie bei *Cons*, wird eine Funktion *create* bereitgestellt, welches die Argumente mit Currying annimmt und den entsprechenden Konstruktor aufruft [7, S. 145–146].

5.2.2 Wenn der Programmierer eine optimierte Übersetzung explizit veranlassen kann

In PureScript gibt es einen Sonderfall von Algebraischen Datentypen, die mit dem Schlüsselwort `newtype` eingeführt werden. Newtypes dürfen nur exakt einen Wertkonstruktor mit exakt einem Argument definieren. Wie der Name schon aussagt, hat das den Zweck einen neuen Namen für einen bestehenden Datentyp zu vergeben, ohne die Laufzeitrepräsentation des unterliegenden Datentyps zu ändern [7, S. 60].

```
1 newtype Euro = Euro Number
2 newtype Dollar = Dollar Number
3
4 fEuro :: Euro -> Euro
5 fEuro euro = euro
6
7 fDollar :: Dollar -> Dollar
8 fDollar dollar = dollar
```

Listing 5.3: Beispiel Newtypes

Im Gegensatz zu reinen Typaliasen, die mit dem Schlüsselwort `type` definiert werden, garantieren Newtypes dadurch, dass sie eine besondere Form von ADT's sind, Typsicherheit. In Listing 5.3 sind zwei NewType's definiert - Euro und Dollar. Es ist bspw. nicht möglich einen Wert vom Typ *Dollar* in die Funktion *fEuro* zu übergeben, da Euro und Dollar inkompatible Typen sind [7, S. 60].

Listing 5.4 veranschaulicht den Unterschied bzgl. der Übersetzung nach JavaScript, wenn man *Euro* als gewöhnlichen Algebraischen Datentyp oder Newtype definiert.

```
1  // as newtype
2  var Euro = function (x1) {
3    return x1;
4  };
5
6  // as common ADT
7  var Euro = (function () {
8    function Euro(value0) {
9      this.value0 = value0;
10   };
11   Euro.create = function (value0) {
12     return new Euro(value0);
13   };
14   return Euro;
15 })();
```

Listing 5.4: Übersetzung nach JavaScript von Newtypes

Newtypes haben zur Laufzeit wie ADT's eine Repräsentation, die durch den Newtype Charakter im Vergleich zu ADT's allerdings Overhead vermeidet. Dadurch eignen sich Newtypes beim Design von Bibliotheken. Ihr Typ zur Laufzeit entspricht dem ihres Arguments [7, S. 146–147].

5.3 Typklassen

Typen können Typklassen zugeordnet werden. Ist das der Fall, definiert die Typklasse für ihre Typen ein bestimmtes Verhalten. Das am nähesten zutreffende Pendant zu Typklassen in der OOP-Welt sind Interfaces [8, Kap. Types and Typeclasses, Abschn. Typeclasses 101].

```
class Eq a where
  eq :: a -> a -> Boolean

data Ordering = LT | GT | EQ

class Eq a <= Ord a where
  compare :: a -> a -> Ordering
```

Listing 5.5: Beispiel Typklassen

Beispielsweise gibt es in der Prelude eine Typklasse *Ord*, welche für all ihre untergeordneten Typen aussagt, dass sie einerseits vergleichbar sind und andererseits wie der Vergleich im konkreten Fall auszusehen hat [7, S. 65]. In Java bspw. entspräche dies in etwa dem Interface *Comparable*. Listing 5.5 zeigt die Definition der Typklasse *Ord*, die eine Funktion *compare* deklariert, um zwei Werte bei Typen, welche eine totale Ordnung unterstützen, zu vergleichen. Von *compare* wird ein Wert vom Typ *Ordering*[3] zurückgegeben, der drei Wertkonstruktoren als Alternative spezifiziert - *LT*, wenn das erste Argument kleiner als das zweite ist, *GT*, wenn es größer ist und *EQ* wenn beide gleich sind [7, S. 66]. Die Typklasse *Ord* ist durch die Typvariable *a* parametrisiert. Typvariablen stehen als Platzhalter für einen beliebigen konkreten Typen. Man erkennt sie daran, dass sie mit keinem Großbuchstaben anfangen, wie es für Typen vorgeschrieben ist, aber an der Stelle von Typen stehen. In Sprachen wie Java kommen Generics Typvariablen am nächsten. Funktionen, die mit Typvariablen definiert sind, werden als polymorphe Funktion bezeichnet, das Konzept im Allgemeinen als Parametrisierte Polymorphie [8, Kap. Types and Typeclasses, Abschn. Typevariables].

Die Typklasse *Eq* deklariert eine Funktion *eq*[4], um zwei Werte auf Gleichheit zu prüfen [7, S. 65]. *Eq* ist damit die Superklasse von *Ord*, da ein Typ, um ein Mitglied von *Ord* zu sein, zuerst ein Mitglied von *Eq* sein sollte. Denn wenn „`a == b`“ unter den Gesetzen von *Eq* gilt, sollte „`compare a b`“ zu *EQ* ausgewertet werden. Diese Super- zu Subklassen Beziehung wird durch den rückwärtsgewandten Doppelstrichpfeil in der Definition von Typklassen ausgedrückt [7, S. 77].

[3] bei *Ordering* handelt es sich um einen Algebraischen Datentyp
[4] der Operator == ist ein Alias für *eq*

5.3.1 Umsetzung von Typklassen in JavaScript am Beispiel der Typklasse Semiring

Wie in Abschnitt „4 Funktionsdefinition und -applikation“ angekündigt, wird das Additions-Beispiel an dieser Stelle wieder aufgegriffen und weiter ins Detail gegangen im Hinblick auf die dort genutzte Typklasse *Semiring*. In Listing 5.6 ist die Typklasse mit der Bezeichnung *Semiring* aus dem Prelude Modul *Data.Semiring* aufgeführt.

```
1 class Semiring a where
2   add  :: a -> a -> a
3   zero :: a
4   mul  :: a -> a -> a
5   one  :: a
```

Listing 5.6: Auszug PureScript Prelude - Typklasse Data.Semiring

Die Typklasse *Semiring* definiert vier Funktionen: die für unser Beispiel notwendige Addition, die Multiplikation und zwei konstante Funktionen *zero* und *one*. Wir möchten an dieser Stelle nicht in die Mathematik abtauchen, allerdings sei gesagt, dass die 0 das neutrale[5] Element in der Addition und das absorbierende[6] Element in der Multiplikation ist - während in der Multiplikation das neutrale Element die 1 ist. Damit bilden die Triple $(\mathbb{N}_0, +, 0)$ und $(\mathbb{N}, \cdot, 1)$ unter dem Assoziativgesetz jeweils einen Monoid[7] und zusammen einen Halbring[8], wobei das Kommutativgesetz für die Addition zusätzlich gilt und für beide das Distributivgesetz.

```
1 instance semiringInt :: Semiring Int where
2   add = intAdd
3   zero = 0
4   mul = intMul
5   one = 1
```

Listing 5.7: Auszug PureScript Prelude - Typinstanz Data.SemiringInt

Ein Typ kann zu einer Instanz einer Typklasse werden, wenn er ihr spezifiziertes Verhalten unterstützt. Beispielsweise ist der Typ *Int* eine Instanz („gehört zu“) der Typklasse *Eq*, da er vergleichbar ist und diese Typklasse die notwendigen Operationen für den Vergleich definiert [8, Kap. Making our own Types and Typeclasses, Abschn. Derived instances]. Ebenso ist der Typ *Int* eine Instanz der Typklasse *Semiring*, wie in Listing 5.7 zu sehen. Man kann dort erkennen wie die Typvariable *a* durch den konkreten Typ *Int* ersetzt wird und für die vier Funktionen der Typklasse deren typabhängige Implementierung angegeben ist. In PureScript werden Typinstanzen benannt (hier: *semiringInt*), um die Lesbarkeit des korrespondierenden JavaScript-Codes zu erhöhen [7, S. 64].

[5] https://de.wikipedia.org/wiki/Neutrales_Element
[6] https://de.wikipedia.org/wiki/Absorbierendes_Element
[7] https://de.wikipedia.org/wiki/Monoid
[8] https://de.wikipedia.org/wiki/Halbring_(Algebraische_Struktur)

In der Prelude gibt es neben der Datei Semiring.purs noch eine Datei Semiring.js - siehe Listing 5.8 und 5.9. In Ersterer wird über PureScripts Foreign Function Interface ein Typ für die korrespondierende JavaScript-Funktion *intAdd*[9] deklariert. In der zweiten Datei wird die Funktion in JavaScript definiert und exportiert.

```
foreign import intAdd :: Int
 -> Int -> Int
```

Listing 5.8: Foreign Function PS

```
exports.intAdd = function (x)
 {
  return function (y) {
    /*jshint bitwise: false*/
    return x + y | 0;
  };
};
```

Listing 5.9: Foreign Function JS

Nach der Übersetzung durch den Compiler landet der Inhalt der Datei Semiring.js in der Datei foreign.js im Ausgabeverzeichnis für die kompilierten Module.

```
1  // ../Data.Semiring/index.js
2  var $foreign = require("./foreign");
3
4  var Semiring = function (add, mul, one, zero) {
5    this.add = add;
6    this.mul = mul;
7    this.one = one;
8    this.zero = zero;
9  };
10
11 var semiringInt = new Semiring($foreign.intAdd, $foreign.intMul, 1,
       0);
12
13 var add = function (dict) {
14   return dict.add;
15 };
```

Listing 5.10: Auszug aus übersetzen Modul Data.Semiring

Wie in Listing 5.10 zu sehen, wird für die Typklasse *Semiring* ein Java-Script Objekt-Typ angelegt, indem eine Konstruktor-Funktion generiert wird [11]. Anschließend wird das Objekt *semiringInt* erzeugt, welches in Abschnitt „4 Funktionsdefinition und -applikation“,

[9]das binäre Oder dient dazu, dass auch bei Berechnungsungenauigkeiten das Ergebnis wieder eine korrekt gerundete Ganzzahl ist, da JavaScript Zahlen immer 64-Bit Gleitkommazahlen sind und die Genauigkeit für Ganzzahlen nur bis zu einer bestimmten Länge garantiert wird [10]

Listing 4.2 als „Type Class Dictionary“ genutzt wurde und damit die Implementierung der Funktionen der Typklasse für die gewählte Instanz bereithält [7, S. 146–147]. Dieses Listing wird an dieser Stelle durch Listing 5.11 nochmals zur besseren Nachvollziehbarkeit wiederholt aufgeführt.

```
1  var Data_Semiring = require("../Data.Semiring");
2  var Prelude = require("../Prelude");
3
4  var addTwo = function (dictSemiring) {
5    return function (a) {
6      return function (b) {
7        return Data_Semiring.add(dictSemiring)(a)(b);
8      };
9    };
10 };
11
12 var constant = addTwo(Data_Semiring.semiringInt)(2)(3);
```

Listing 5.11: Funktionen in JavaScript - Kopie

Der Compiler umschließt die Funktionsdefinitionen um eine weitere Funktion zur Einführung dieses Type Class Dictionaries als weiteres Argument und ruft explizit die Funktion *add* aus der Datei Data.Semiring/index.js auf, welche auf die Funktion *add* aus dem Type Class Dictionary names *semiringInt* zugreift. Mit diesem Mechanismus werden in PureScript Funktionen mit einem Typklassen-Constraint zur Laufzeit in JavaScript repräsentiert [7, S. 148].

Würde man anstatt *add* die Funktion *mul* aus der Typklasse nutzen, würde sich Zeile 7 wie folgt ändern: „`return Data_Semiring.mul(dictSemiring)(a)(b);`“. Rechnet man hingegen mit Fließkommazahlen anstatt Ganzzahlen, sähe Zeile 12 so aus: „`var constant = addTwo(Data_Semiring.semiringNumber)(2.0)(3.0);`“. Das Übersetzungsmuster scheint also bestehen zu bleiben und es wird entweder nur eine andere Funktion aus dem Type Class Dictionary gewählt oder ein anderes Typ Class Dictionary, durch die Verwendung einer anderen Typklasse in PureScript, erzeugt.

5.3.2 Bestätigung des Übersetzungsmusters von Typklassen

Am Anfang dieses Kapitels wurden Typklassen am Beispiel der Typklasse *Ord* erläutert. In diesem Unterabschnitt soll durch den Vergleich von zwei Werten eines bestimmten Typs das Übersetzungsmuster aus dem vorherigen Unterabschnitt 5.3.1 bestätigt werden. Durch die Listings 5.12 und 5.13 ist erkennbar, wie die Benutzung der Typklasse *Ord* zu dem gleichen Übersetzungsmuster wie bei der Typklasse *Semiring* führt.

Es wird logischerweise ein anderes Type Class Dictionary (*ordInt*) genutzt und andere Funktionen aufgerufen, da in Listing 5.12 die Größer-Als-Vergleichsoperation (>) genutzt

wird. Das Muster ist jedoch identisch. Auf Implementierungsdetails abseits der gezeigten Top-Ebene der Übersetzung sei an dieser Stelle verzichtet.

```
module Main where

import Prelude

compGT x y = y > x

constant = compGT 2 3
```

Listing 5.12: Beispiel mit Typklasse Ord in PureScript

```
var Data_Ord = require("../Data.Ord");
var Prelude = require("../Prelude");

var compGT = function (dictOrd) {
  return function (x) {
    return function (y) {
      return Data_Ord.greaterThan(dictOrd)(y)(x);
    };
  };
};

var constant = compGT(Data_Ord.ordInt)(2)(3);
```

Listing 5.13: Übersetzung nach JavaScript

5.3.3 Wenn Compileroptimerungen das Übersetzungsmuster durchbrechen

Das vorgestellte Übersetzungsmuster wird vom PureScript Compiler nicht strikt verfolgt. Es gibt Situation, wo er Optimierungen ansetzt. Eine wird im Folgenden vorgestellt.

```
var Data_Ord = require("../Data.Ord");
var Prelude = require("../Prelude");

var compGT = function (x) {
  return function (y) {
    return y > x;
  };
};

var constant = compGT(2)(3);
```

Listing 5.14: Übersetzung nach JavaScript mit anderem Muster

Wenn man Funktionen mit einer Typdeklaration ausstattet, anstatt, dass der Comiler den Typ durch Typinferenz herausfinden muss, wird wenn möglich ein optimiertes Übersetzungsmuster genutzt. In Listing 5.14 ist dargestellt, wie das vorgestellte Beispiel mit der Typklasse Ord aus Listing 5.12 übersetzt wird, wenn man eine Typdeklaration wie folgt angibt: „`compGT :: Int -> Int -> Boolean`“.

Es entfällt vollständig die Nutzung des Type Class Dictionaries und der damit verbundenen Funktionsdefinitionen und -applikationen. Im Beispiel der Addition, in Verbindung mit der Typklasse *Semiring* aus Listing 4.1, verhält es sich ähnlich. Es wird mit Typdeklaration gemäß des Musters von Listing 5.14 übersetzt und der Code zur Addition („`return a + b | 0;`“), der ursprünglich über einen Foreign Function Import genutzt wurde (Listing 5.9), wird direkt in die innerste, durch Currying erzeugte, Funktion gesetzt.

5.3.4 Lessons Learnd

- Typklassen in PureScript werden als Object Types (Klassen) in JavaScript abgebildet (sind aber vom Konzept keine Klassen im Sinne von OOP!).
- Typinstanzen in PureScript sind Objekte in JavaScript, welche in diesem Kontext auch als Type Class Dictionaries bezeichnet werden. Sie bekommen bei Erzeugung die entsprechenden Funktionen über den Konstruktor übergeben.
- Die Bezeichnungen von Typklassen oder Typinstanzen werden ebenso für ihre Pendants in JavaScript genutzt. Dabei tragen die Bezeichner einer Typinstanz im ersten Teil immer den Namen ihrer Typklasse, gefolgt vom Datentyp für den sie stehen.
- Bei Gebrauch von Typinferenz erzeugt der Compiler Code, in dem Type Class Dictionaries genutzt werden und die für die zusätzliche äußerste Funktion beim Currying von Funktionen verantwortlich sind.
- Funktionen mit Typdeklaration werden vom Compiler, sofern möglich, ohne die Nutzung von Type Class Dictionaries übersetzt.

6 Rekursion und Funktionen höherer Ordnung

Rekursion ist in der funktionalen Programmierung die Basistechnik schlechthin um wiederholende Operationen auszuführen. In rein funktionalen Sprachen wie PureScript gibt es keine veränderlichen Zustände und damit auch keine Schleifen, wie man sie aus der imperativen Programmierung kennt. Das begründet sich aus der Tatsache, dass man in funktionalen Sprachen Berechnungen durchführt, indem man vornehmlich beschreibt was etwas ist, anstatt wie man etwas macht [7, S. 31–32].

In diesem Kapitel wird demonstriert, wie unterschiedliche Formen der Rekursion nach JavaScript übersetzt werden und welche Optimierungen der Compiler dabei durchführt. Darüber hinaus wird eine prominente Funktion (*fold*) aus der funktionalen Welt vorgestellt, welche zwar das Konzept der Rekursion intern nutzt, aber für die Anwendung durch den Nutzer davon abstrahiert.

6.1 Rekursion und deren Übersetzung nach JavaScript

In diesem Abschnitt soll einerseits veranschaulicht werden, wie man Rekursion in PureScript ausdrückt und andererseits, wie sie in JavaScript vom Compiler umgesetzt wird. Dazu werden zwei Beispiele herangezogen, die Definition der Fibonacci-Funktion und die rekursive Definition der Summenberechnung über eine Liste. Wie im Abschnitt der Algebraischen Datentypen bereits erwähnt, ist eine Liste ein ADT, der über sich selbst rekursiv definiert ist und damit ist die Struktur der Liste prädestiniert für Rekursion in Verbindung mit Pattern Matching. Im Beispiel des Listings 6.1 wird die Listendefinition aus der Bibliothek von PureScript genutzt. Für unsere jetzigen Zwecke ist es hinreichend, sie sich so vorzustellen, wie wir sie in Listing 5.1 definiert haben.

In Listing 6.1 ist zunächst die Fibonacci-Funktion (`fib`) und anschließend die Summenfunktion (`sum`) definiert, gefolgt von einem Verwendungsbeispiel, deren Resultate in Konstanten abgelegt werden. Der Wert von *resFib* entspricht 89 und von *resSum* 6 im Beispiel. Bei bspw. *sum* ist zu beachten, dass die Angabe von Mustern, um zwei Strukturen zu ver-

gleichen, eingesetzt werden, um die Abbruchbedingung und Berechnungsvorschrift für die Rekursion zu definieren. Dies entspricht den Wertkonstruktoren des ADT's der Liste - eine Liste kann entweder leer sein oder einen Kopf, gefolgt von einem Rest haben, wobei der Rest wieder eine Liste ist. Der im Pattern verwendete Doppelpunkt ist ein Infix-Alias für den Wertkonstruktor *Cons*. Die Summe der Elemente einer leeren Liste ist 0, die Summe einer nicht leeren Liste ist der Wert des Kopfs der Liste (des ersten Elements) und der Wert der Summe über den Rest der Liste.

```
1  module Main where
2
3  import Prelude
4  import Data.List
5
6  fib :: Int -> Int
7  fib 0 = 1
8  fib 1 = 1
9  fib n = fib (n - 1) + fib (n - 2)
10
11 sum :: List Int -> Int
12 sum Nil = 0
13 sum (head : tail) = head + sum (tail)
14
15 resFib = fib 10
16 resSum = sum (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Listing 6.1: Rekursion in PureScript

Die Summenfunktion ist in dieser Definitionsart eine linear rekursive Funktion. Das heißt, sie hat eine nachhängende Operation neben dem rekursiven Aufruf - in diesem Fall die Addition des Vorgängerelements. Der PureScript Compiler übersetzt diese Art von Funktionen wie in Listing 6.2 dargestellt. Rekursive Funktionen, bei denen die nachhängende Operation ein weiterer rekursiver Aufruf ist, werden auf die gleiche Weise übersetzt und als allgemeine rekursive oder einfach nur rekursive Funktionen bezeichnet, da sie keiner Beschränkung unterliegen. Eine Funktion dieser Form ist die Fibonacci-Funktion. In Listing 6.2 sei wegen der Länge auf die Imports und Aufrufbeispiele verzichtet.

```
1  var fib = function (v) {
2    if (v === 0) {
3      return 1;
4    };
5    if (v === 1) {
6      return 1;
7    };
8    return fib(v - 1 | 0) + fib(v - 2 | 0) | 0;
9  };
10
```

```
var sum = function (v) {
  if (v instanceof Data_List_Types.Nil) {
    return 0;
  };
  if (v instanceof Data_List_Types.Cons) {
    return v.value0 + sum(v.value1) | 0;
  };
  throw new Error("Failed pattern match at Main line 15, column 1 -
    line 15, column 23: " + [ v.constructor.name ]);
};
```

Listing 6.2: Übersetzung von Rekursion nach JavaScript

Beide Funktionen haben im Wesentlichen das gleiche Übersetzungsmuster. Das Pattern Matching wird in If-Abfragen umgewandelt. Die Abbruchbedingungen werden allesamt auf diese Weise definiert. Bei der Summenfunktion kommt die Definition eines Errors hinzu, falls die Eingabe nicht das Muster der Datenstruktur Liste erfüllt. Wenn man von den Bezeichnungen einmal absieht, entspricht die Übersetzung von Rekursion ziemlich genau der Form, wie sie ein Programmierer manuell in JavaScript verwendet hätte.

6.2 Optimierte Übersetzung von Endrekursion

Die übliche Umsetzung von Rekursion in Programmiersprachen wie JavaScript hat einen entscheidenden Makel. Wenn die Eingaben zu groß werden, die Verschachtlung von Funktionsaufrufen zu tief und der Stack dann seine maximale Größe überschreitet, endet dies in einem Fehler - StackOverflow. Um unbegrenzte Eingabe bei Rekursion dennoch nutzen zu können, wird eine Technik namens „Tail Recursion Optimization“ (kurz: TCO) von PureScript eingesetzt. Die rekursive Funktion wird bei der Übersetzung in eine while-Schleife umgewandelt, was den Speicherplatzverbrauch von der Rekursionstiefe unabhängig macht, nämlich konstant. Dies gelingt nur, wenn die rekursive Funktion endrekursiv ist. Das heißt dem rekursiven Aufruf folgen keine weiteren Operationen, weshalb kein neuer Stackframe allokiert werden müsste. Viele linear rekursive Funktionen können in endrekursive Funktionen umformuliert werden, was diese Optimierungstechnik trotz der Einschränkung der Endrekursion wertvoll macht [7, S. 42–43].

```
module Main (sumTailRec) where

import Prelude
import Data.List

sumTailRec :: List Int -> Int
sumTailRec lst = sumTailRec' lst 0

```

```
sumTailRec' :: List Int -> Int -> Int
sumTailRec' Nil acc = acc
sumTailRec' (head : tail) acc = sumTailRec' (tail) (head + acc)
```

Listing 6.3: Endrekursion in PureScript

Listing 6.3 zeigt die Umformulierung der Summenfunktion in eine endrekursive Form. Eine gängige Technik zur Umformung ist die Nutzung eines akkumulierenden Parameters. Bei der Funktionsdefinition wird ein zusätzlicher Parameter verwendet, durch den die Ergebnisse eines jeden rekursiven Aufrufs aufgehäuft werden, anstatt dafür den Return Value der Funktion zu nutzen [7, S. 43]. Dadurch, dass die Berechnung und deren Resultat über den Parameter über die rekursiven Aufrufe hinweg mitgezogen wird, muss der Stackframe im Gegensatz zu einer linear rekursiven Funktion nicht aufbewahrt werden, um beim Aufstieg aus der Rekursion die Berechnungen mit den jeweiligen Werten des primären Parameters durchzuführen. Damit die Funktion für den Anwender die gleiche Signatur behält, wird meist die eigentliche rekursive Berechnung in einer außerhalb des Moduls nicht sichtbaren Hilfsfunktion ausgeführt.

In Listing 6.4 ist die Übersetzung der endrekursiven Summenfunktion auf Listen dargestellt. Die Abkürzung *tco* in Variablennamen steht mutmaßlich für „Tail Call Optimization“[1].

```
var sumTailRec$prime = function ($copy_v) {
  return function ($copy_acc) {
    var $tco_var_v = $copy_v;
    var $tco_done = false;
    var $tco_result;
    function $tco_loop(v, acc) {
      if (v instanceof Data_List_Types.Nil) {
        $tco_done = true;
        return acc;
      };
      if (v instanceof Data_List_Types.Cons) {
        $tco_var_v = v.value1;
        $copy_acc = v.value0 + acc | 0;
        return;
      };
      throw new Error("<pattern error message skipped by author>");
    };
    while (!$tco_done) {
      $tco_result = $tco_loop($tco_var_v, $copy_acc);
    };
    return $tco_result;
  };
```

[1] https://en.wikipedia.org/wiki/Tail_call

```
23 };
24
25 var sumTailRec = function (lst) {
26   return sumTailRec$prime(lst)(0);
27 };
```

Listing 6.4: Übersetzung von Endrekursion nach JavaScript

Wir fangen bei der Erklärung der Variablen an. Die Erläuterung basiert auf einem Beispielaufruf mit einer Liste der Form: `Cons 1 (Cons 2 (Cons 3 Nil))`. Durch den Aufruf der Funktion *sumTailRec* wird die Liste an die Funktion *sumTailRec$prime* zzgl. des Akkumulatorwerts `0` übergeben (Zeile 25 - 27). Die Funktion *sumTailRec$prime* enthält die eigentliche Geschäftslogik und Übersetzung der Endrekursion in eine while-Schleife. Dazu definiert sie zwei Argumentvariablen aufgrund des Curryings und drei Variablen innerhalb der innersten Funktion (Zeile 1 - 5). Nachfolgende Aufzählung beschreibt die Intension hinter diesen fünf Variablen und deren Initialisierung für unseren Beispielaufruf.

$copy_v: enthält die initiale Liste, die als Argument übergeben wurde, ändert seinen Werte nie und ist für die weitere Betrachtung nicht mehr von Belang - `Cons 1 (Cons 2 (Cons 3 Nil))`

$copy_acc: enthält den initialen Akkumulatorwert und wird bei jedem Schleifendurchlauf mit dem neuen Akkumulatorwert beschrieben - `0`

$tco_var_v: bekommt initial die Liste aus *$copy_v* übergeben und wird bei jedem Schleifendurchlauf mit der kürzer werdenden Liste beschrieben - `Cons 1 (Cons 2 (Cons 3 Nil))`

$tco_done: wird als Schleifenbedingung (Flag) und damit zum Ausstieg aus dieser genutzt, sobald die Summe vollständig berechnet wurde - `false`

$tco_result: enthält im letzten Schleifendurchlauf die endgültige Summe und gibt sie anschließend zurück bei Funktionsende - `undefined`

Im Anschluss zu diesen fünf Variablen wird die Funktion *$tco_loop* definiert, welche die eigentliche Geschäftslogik und Pattern Matching und Berechnungsvorschriften enthält (Zeile 6 bis 17). Sie gleicht der naiven Übersetzung der Summenfunktion in Listing 6.2 und beschreibt, was später wiederholt ausgeführt werden soll.

Darauf folgt eine while-Schleife, welche die Funktion *$tco_loop* mit den Variablen *$tco_var_v* und *$copy_acc* aufruft (Zeile 18 - 20). Dies wird solange wiederholt, bis die Variable *$tco_done* aus der Funktion *$tco_loop* heraus auf `true` gesetzt wurde. Zum Abschluss bekommt *$tco_result* die endgültige Summe zugewiesen, die Schleife wird verlassen und der Wert von *$tco_result* zurückgegeben.

Nachdem alle beteiligten Komponenten erklärt wurden, kommen wir nun zur Dynamik des Aufrufs. Nebenstehende Aufzählung begleitet die folgenden Erläuterungen für den

Beispielaufruf mit einer Liste von `Cons 1 (Cons 2 (Cons 3 Nil))`. Die Aufzählung zeigt, welche Werte die Variablen nach einem Schleifendurchlauf und Aufruf der Funktion *$tco_loop* haben. Variablen, die in einem Schleifendurchlauf keine Wertveränderung erfahren haben, sind nicht aufgeführt.

Bei jedem Schleifendurchlauf wird die Funktion *$tco_loop* mit der immer kürzer werdenden Liste (*$tco_var_v*) und dem aktuellen Summenwert (*$copy_acc*) aufgerufen. Beide Variablen werden bei jedem Schleifendurchlauf aktualisiert, bis die Liste leer ist, die Abbruchbedingung greift und der Wert des Akkumulators (*$copy_acc*) einmalig am Ende in *$tco_result* gespeichert wird, bevor die Schleife verlassen wird.

1. **$tco_var_v:** `Cons 2 (Cons 3 Nil)`
   $copy_acc: `1`
   $tco_result: `undefined`
2. **$tco_var_v:** `Cons 3 Nil`
   $copy_acc: `3`
   $tco_result: `undefined`
3. **$tco_var_v:** `Nil`
   $copy_acc: `6`
   $tco_result: `undefined`
4. **$tco_done:** `true`
   $tco_result: `6`

Wenn die Liste nicht leer ist (Zeile 11 - 15), wird die zustandsbehaftete Variable *$tco_var_v*, welche mit der Eingabeliste initialisiert ist, auf den Rest der Liste (*v.value1*) gesetzt. Die Variable nähert sich also bei jedem Schleifendurchlauf dem Ende der Liste und damit der Abbruchbedingung an, indem der Kopf „abgeschnitten“ wird. Die wechselnden Zwischensummen werden in der Akkumulator Variable *$copy_acc* abgelegt. Daraufhin wird nichts zurückgegeben[2].

Wenn die Liste leer ist (Abbruchbedingung), wird nicht nur der Endwert der Summe zurückgegeben, sondern ebenso das Flag *$tco_done* von `false` auf `true` gesetzt. Dieses wird später in Zeile 18 dazu genutzt, die while-Schleife zu verlassen und anschließend das Endergebnis (*$tco_result*) zurückzugeben.

Die Variable *$tco_result* ist also bis auf diesen Schlussmoment uninitialisiert[2] und im Gegensatz zu Listing 6.2 wird für den Fall, dass man noch nicht am Ende der Liste angelangt ist, nicht die Zwischensumme ständig durch den Funktionsaufruf zurückgegeben, sondern nur einmal beim Erfüllen der Abbruchbedingung, wo es dann die Endsumme ist. Die sich verändernde Liste und Summe werden über die Nutzung von Seiteneffekten, mittels außerhalb der innersten Funktion (*$tco_loop*) definierten Variablen, geschrieben und gelesen. Im letzten Schleifendurchlauf enthält *$copy_acc* die Endsumme, übergibt sie an den Parameter *acc*, welcher durch den Fall der Abbruchbedingung als Returnwert der innersten Funktion wiederum *$tco_result* zugewiesen wird, um ihn schlussendlich als Endergebnis des gesamten Aufrufs zurück zu liefern.

Aber die Optimierung von Endrekursion hat auch seine Grenzen. Wechselseitig endrekursive Funktionen werden in Purescript nicht optimiert bei der Übersetzung. Bei-

[2] genau genommen wird bei einem `return` ohne nachfolgenden Ausdruck `undefined` zurückgegeben und eine nicht initialisierte Variable ist implizit ebenso mit `undefined` belegt [12, 13]

spielsweise werden die beiden endrekursiven Funktionen aus Listing 6.5 auch in JavaScript durch Rekursion repräsentiert. Auf die genaue Übersetzung sei an dieser Stelle verzichtet. Sie gestaltet sich analog zu der aus Listing 6.2.

```
1  module Main where
2
3  import Prelude
4
5  isEven :: Int -> Boolean
6  isEven 0 = true
7  isEven 1 = false
8  isEven n = isOdd (n - 1)
9
10 isOdd :: Int -> Boolean
11 isOdd 0 = false
12 isOdd 1 = true
13 isOdd n = isEven (n - 1)
```

Listing 6.5: Wechselseitige Endrekursion in PureScript

6.3 Funktionen höherer Ordnung

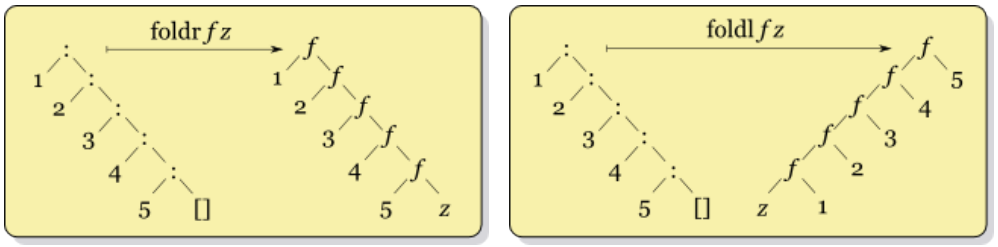
Nachdem wir in den letzten Abschnitten Rekursion als einziges Wiederholungskonzept in rein funktionalen Sprachen kennengelernt und gesehen haben, wie effizient Endrekursion nach JavaScript übersetzt wird, erscheint es zunächst unumgänglich sich der rekursiven Programmierung in der Praxis bedienen zu müssen. Allerdings sind in der funktionalen Welt Funktionen bereits vorhanden, mit denen sich Aufgaben wie die Berechnung der Summe aus den Elementen einer Liste formulieren lassen, ohne explizit Rekursion nutzen zu müssen. Die hier gemeinten Funktionen werden Funktionen höherer Ordnung genannt. Sie zeichnen sich dadurch aus, dass sie eine Funktion als Argument annehmen oder eine Funktion zurückgeben können. Die Summe der Elemente einer Liste kann bspw. mit der Funktion *fold* realisiert werden, die in allen gängigen funktionalen Programmiersprachen in etwa diesen Namen trägt, wodurch die Absicht des eigenen Quellcodes für andere Programmierer viel schneller zu erfassen ist, als wenn man blanke Rekursion nutzen würde [7, S. 44]. In Listing 6.7 wird die Definition der Summenfunktion unter zu Hilfenahme von *fold* demonstriert und die Funktion genauer erklärt. Bevor wir allerdings dazu kommen, sei noch ein weiterer Vorteil von Funktionen höherer Ordnung erwähnt. Es ist mit ihnen leicht die Geschäftslogik vom Kontext zu trennen - dem Code, der z.B. nötig ist, um über eine bestimmte Datenstruktur zu traversieren. Das heißt, eine Funktion wie *fold* definiert intern wie z.B. eine Liste durchlaufen werden soll, und nimmt die eigentliche Berechnungsfunktion als Argument an. Dies führt dazu, dass wir ohne großen Aufwand und Coderedundanzen, in diesem Fall nicht nur die Summe, sondern bspw. auch das Produkt

der Elemente berechnen könnten. In OO-Sprachen, in denen Funktionen keine First-Class-Citizens[3] sind, realisiert man so etwas durch das Entwurfsmuster Strategie.

```
class Foldable f where
  foldr :: forall a b. (a ->
  b -> b) -> b -> f a -> b
  foldl :: forall a b. (b ->
  a -> b) -> b -> f a -> b
```

Listing 6.6: Typklasse Foldable

Alle Datenstrukturen, die man falten kann, sind Instanzen der Typklasse *Foldable*. In Listing 6.6 ist diese mit zwei wesentlichen Funktionsdeklarationen auszugsweise dargestellt. Der Vorgang des Faltens beschreibt einen Prozess, indem eine Datenstruktur von Werten zu einem Wert reduziert wird. Wie dieser Prozess gestaltet wird, entscheidet eine Funktion, die als Argument der *fold*-Funktion übergeben wird. Die Funktion *foldl* faltet die Datenstruktur von links aus gesehen zu einem Wert und *foldr* von rechts aus [15, 7, S. 40–42]. Beide Faltfunktionen nehmen als erstes Argument eine Funktion an. Diese nimmt einen Wert vom Typ a an und einen Wert vom Typ b. Ersterer ist ein Wert aus der Datenstruktur, die als drittes Argument angeben wird ($f\ a$)[4]. Zweiterer dient als Akkumulator für jeden Berechnungsschritt und wird letztlich von der Faltfunktion zurückgegeben. Zusätzlich muss ein Initialwert für den Akkumulator als zweites Argument übergeben werden. Je nach Berechnungsintension ist der Akkumulatorwert ein neutrales Element in der jeweiligen Berechnung. Abbildung 6.1 veranschaulicht das Falten anhand von einer Liste von Ganzzahlen (hier 1-5) als Datenstruktur. Der Doppelpunkt steht für *Cons*, `[]` beschreibt die leere Liste (*Nil*), z ist der Akkumulatorwert und f die Funktion, welche die Berechnung bei jedem Faltschritt definiert.



(a) Falten von rechts (b) Falten von links

Abbildung 6.1: Veranschaulichung des Faltens anhand von Listen [15]

Das Faltbeispiel der Summen- und Produktfunktion aus Listing 6.7 ist semantisch kompatibel mit Abbildung 6.1, auch wenn Arrays statt Listen angewendet werden. Sowohl die Summe als auch das Produkt werden über *foldl* ausgedrückt. Als Funktion und Akkumulator dient bei der Summe die binäre Funktion `+` und `0` als neutrales Element, bei dem Produkt `*` und `1`.

[3] in denen sie nicht wie andere Werte Typen haben können, und nicht wie andere Werte als Funktionsargument genutzt werden können

[4] f steht für einen Funktor eines beliebigen Typs a (Typkonstruktor). Dazu kommen wir im nächsten Kapitel detailliert. Für den Moment stellen wir uns f einfach als Datenstruktur oder Container vor.

```
module Main where

import Prelude
import Data.Foldable

sum :: forall a f. Foldable f => Semiring a => f a -> a
sum = foldl (+) zero

product :: forall a f. Foldable f => Semiring a => f a -> a
product = foldl (*) one

resSum = sum [1, 2, 3]
resPro = product [1, 2, 3]
```

Listing 6.7: Falten in PureScript

Listing 6.8 zeigt am Beispiel des Faltens, wie Funktionen höherer Ordnung nach JavaScript übersetzt werden können. Da es sich bei *Foldable* um eine Typklasse handelt, ist das Übersetzungsmuster gemäß des in Abschnitt „5.3 Typklassen“ beschriebenen.

```
var Prelude = require("../Prelude");
var Data_Foldable = require("../Data.Foldable");
var Data_Semiring = require("../Data.Semiring");

var sum = function (dictFoldable) {
  return function (dictSemiring) {
    return Data_Foldable.foldl(dictFoldable)(Data_Semiring.add(
   dictSemiring))(Data_Semiring.zero(dictSemiring));
  };
};

var product = function (dictFoldable) {
  return function (dictSemiring) {
    return Data_Foldable.foldl(dictFoldable)(Data_Semiring.mul(
   dictSemiring))(Data_Semiring.one(dictSemiring));
  };
};

var resSum = sum(Data_Foldable.foldableArray)(Data_Semiring.
   semiringInt)([ 1, 2, 3 ]);
var resPro = product(Data_Foldable.foldableArray)(Data_Semiring.
   semiringInt)([ 1, 2, 3 ]);
```

Listing 6.8: Falten in JavaScript

Es wird also analog die Typklasse *Foldable* aus PureScript als Object Type (Klasse) in JavaScript repräsentiert. Die Typinstanz für die Datenstruktur Array ist ein Objekt (Type Class Dictionary) mit der Bezeichnung *foldableArray*, welches ähnlich wie in Abschnitt 5.3 Listing 5.10 beim Objekt *semiringInt* die Implementierung des Faltens als Foreign Function Import zugewiesen bekommt. Dieses Type Class Dictionary wird als Argument übergeben, zzgl. einer Berechnungsfunktion und der Datenstruktur (Listing 6.8, je Zeile 17 und 18). Listing 6.9 zeigt, wie das Falten in JavaScript implementiert ist.

```
1  exports.foldrArray = function (f) {
2    return function (init) {
3      return function (xs) {
4        var acc = init;
5        var len = xs.length;
6        for (var i = len - 1; i >= 0; i--) {
7          acc = f(xs[i])(acc);
8        }
9        return acc;
10     };
11   };
12 };
13
14 exports.foldlArray = function (f) {
15   return function (init) {
16     return function (xs) {
17       var acc = init;
18       var len = xs.length;
19       for (var i = 0; i < len; i++) {
20         acc = f(acc)(xs[i]);
21       }
22       return acc;
23     };
24   };
25 };
```

Listing 6.9: Falten als Native-Implementierung in JavaScript

Es wird deutlich, dass das Falten nicht mit Rekursion, sondern Schleifen umgesetzt ist. Den Unterschied zwischen von links und rechts Falten, erkennt man an den jeweiligen Schleifenköpfen. Das Array *xs* wird einmal von rechts und einmal von links durchlaufen. Der Parameter *init* hält den Initialwert des Akkumulators, dessen Variable *acc* heißt.

Die Übergabe und Rückgabe von Funktionen in Funktionen höherer Ordnung gestaltet sich als intuitiv, denn in JavaScript sind Funktionen ebenfalls First-Class-Citizens. Die Berechnungsfunktion namens *f* wird einfach als Argument übergeben und in den Zeilen 7 und 20 jeweils mit dem aktuellen Array-Element und Akkumulatorwert aufgerufen.

7 | Funktoren, Applicatives, Monaden

In der Prelude werden drei Typklassen definiert, die wesentlich für die Handhabung von Seiteneffekten in PureScript sind - *Functor*, *Applicative* und *Monad* [7, S. 68].

Rein funktionale Programmiersprachen kennen keine Seiteneffekte - zumindest keine Unkontrollierten. Das bedeutet Funktionen können „nicht einfach so“ einen Zustand, wie den Inhalt einer Variable, ändern. Sie liefern bei gleichen Parametern, bei jedem Aufruf, immer das gleiche Resultat (siehe Abbildung 7.1). Ein wesentlicher Vorteil von dieser selbst auferlegten Beschränkung ist, dass es viel leichter als in imperativen Sprachen ist, Beweisführungen, oder zumindest Schlussfolgerungen, über die Korrektheit von Funktionen anzustellen. Dies fängt bereits bei den Vorzügen von unveränderlichen gegenüber veränderlichen Datenstrukturen an - gerade bei Nebenläufigkeit. Wenn Funktionen allerdings keine Zustände ändern können, gelangt man spätestens bei IO-Operationen an den Punkt, dass irgendwie mit der Welt außerhalb des eigenen Programms interagiert werden muss. Dies geht auch in rein funktionalen Sprachen nur über Seiteneffekte - allerdings Seiteneffekte, die nur in eigenen Berechnungskontexten, abgeschottet vom Rest unseres Programms, auftreten dürfen [8, Kap. Input and Output].

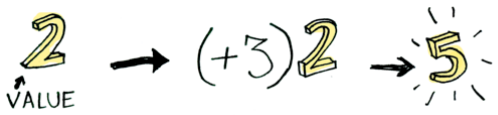


Abbildung 7.1: Wert und einfache Funktionsanwendung mit ihm [2]

Genau an dieser Stelle halten Konzepte wie Monaden aus der Mathematik (Kategorientheorie) in der Informatik Einzug. Der Weg zum Verständnis von Monaden führt über Funktoren und Applicatives, da aufgrund der Typklassenhierarchie eine Monade auch immer ein Applicative und ein Applicative ein Funktor ist.

7.1 Funktoren

Die Typklasse Funktor definiert eine Klasse für alle Typkonstruktoren *f*, die eine map-Operation unterstützen - siehe Listing 7.1. Die parametrisierten Typen

```
class Functor f where
  map :: forall a b. (a -> b) ->
  f a -> f b
```

Listing 7.1: Typklasse Functor

Array oder Maybe sind bspw. Funktoren,
da sie eine Instanz zur Typklasse Funktor
definieren (dazu später mehr) [7, S. 29, 33]. Die Map-Funktion nimmt als erstes Argument eine Funktion von einem Typ a in einen Typ b an, als zweites einen Typ f von a und gibt einen Typ f von b zurück, indem sie die Funktion von a nach b auf den Typ f von a anwendet. Damit handelt es sich bei *map* um eine Funktion höherer Ordnung.

Funktoren werden sich gerade in diesem Szenario oftmals als Container vorgestellt. Eine präzisere Vorstellung wäre es sich einen Funktor als Berechnungskontext vorzustellen, in dem Seiteneffekte auftreten dürfen [8, Kap. Functors, Applicative Functors and Monoids, Abschn. Functors redux]. Denn über den konkreten Berechnungskontext (Wertkonstruktor eines Funktors) entscheidet sich, was das Resultat einer Funktionsanwendung sein wird [2].

Bevor wir uns im Anschluss einem konkreten Beispiel mit Funktoren in PureScript und dessen Übersetzung nach JavaScript widmen, soll im nächsten Unterabschnitt das beschriebene Konzept zunächst visualisiert werden.

7.1.1 Funktoren in Bildern am Beispiel von Maybe

Bei der Visualisierung des Funktorkonzepts wird sich des Algebraischen Datentyps *Maybe* als Beispiel bedient. Listing 7.2 wiederholt noch einmal die Definition von *Maybe* mit dessen Wertkonstruktoren *Nothing* und *Just*.

Um eine Funktion auf den Wert in einem Funktor anwenden zu können, ohne den Wert vorher auszupacken, bedarf es einer weiteren Funktion, in der definiert ist, wie die Funktionsanwendung durchgeführt werden soll - und das macht *map*.

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor
  Maybe where
  map fn (Just x) = Just (fn x)
  map _  _        = Nothing
```

Listing 7.2: ADT Maybe u. Funktorinstanz

Anhand der Wertkonstruktoren, des Berechnungskontextes, wird entschieden, welche Auswirkungen die Anwendung einer Funktion hat. Abbildung 7.2 veranschaulicht dies. Je nachdem, ob ein *Just* von etwas oder ein *Nothing* vorliegt, muss die Anwendung einer Funktion, nennen wir sie *fn*, unterschiedliche Auswirkungen haben (Seiteneffekt). In Listing 7.2 ist die Funktorinstanz von Maybe zur Typklasse Funktor aus Listing 7.1 aufgeführt.

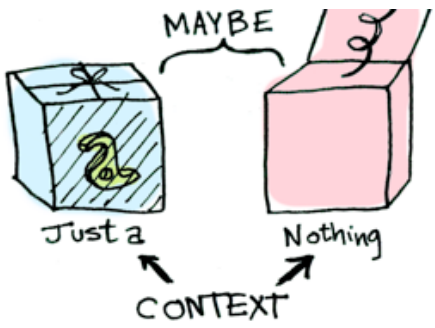


Abbildung 7.2: Maybe Context [2]

Im Folgenden besprechen wir beide Definitionen von Map, je nachdem welches Pattern greift. Es beginnt mit dem Fall, dass Map eine Funktion *fn* bekommt und ein *Just* mit einem Wert x eines Typen a. Gehen wir zum besseren Verständnis davon aus, der Typ wäre in einem konkreten Fall *Int* und wir würden eine Funktion auf einen im Berechnungskontext gebundenen Wert *2* anwenden wollen, die diesen mit einem freien Wert *3* addiert (siehe Abbildung 7.3).

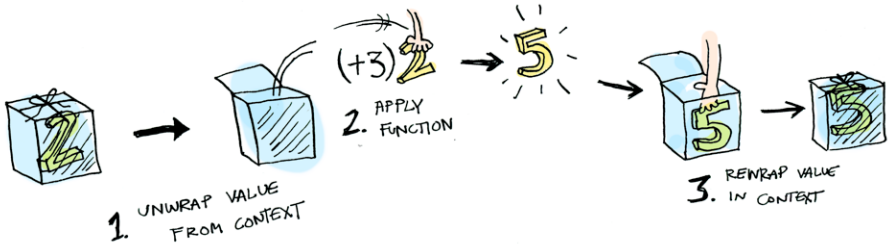


Abbildung 7.3: Funktionsanwendung auf ein Just von 2 [2]

In dem Fall ist *map* so definiert, dass intern die Funktion auf den Wert $x = 2$ angewendet wird. Anschließend wird das Ergebnis, hier $x = 5$, über den Wertkonstruktor *Just* in einem neuen Berechnungskontext (Funktor) gekapselt und dieser zurückgegeben.

Wenn der andere Fall eintritt, dass *map* eine Funktion bekommt und ein *Nothing*, muss *map* abweichend definiert sein. Denn auf was sollte die Funktion angewendet werden, wenn *Nothing* ja gar keinen Wert intern bereithält.

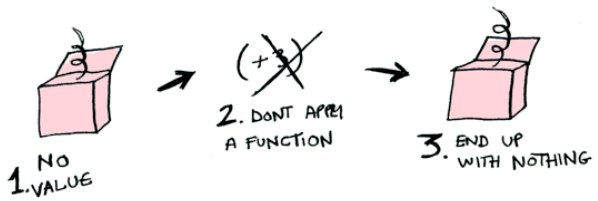


Abbildung 7.4: Funktionsanwendung auf Nothing [2]

Wie in der Definition von *map* in Listing 7.2 zu sehen und durch Abbildung 7.4 visualisiert, ist das vorgeschriebene Verhalten also, dass ohne Funktionsanwendung wieder ein *Nothing* zurückgegeben wird.

Typinstanzen der Typklasse *Functor* müssen darüber hinaus zwei Gesetzmäßigkeiten bei der Definition ihrer *map*-Funktion einhalten:

- Identity Law: `map identity xs = xs`
- Composition Law: `map g (map f xs) = map (g <<< f ) xs`

Diese Gesetze garantieren, dass nicht der Berechnungskontext (Funktor) verändert wird, sondern nur der Wert. Das Gesetz der Identität sagt aus, dass wenn die Identitätsfunktion auf einen Funktor *xs* angewendet wird, der originale Funktor zurückgegeben werden muss. Das macht Sinn, da die Identitätsfunktion ihre Eingabe nicht modifiziert. Das Gesetz der

Komposition sagt aus, dass die Funktionsanwendung einer Funktion f auf einen Funktor xs über *map* und die anschließende Funktionsanwendung einer weiteren Funktion g auf dieselbe Weise über den resultierenden Funktor, das gleiche Resultat erzeugen muss, wie wenn man die Komposition der beiden Funktion g und f direkt auf den Funktor über *map* angewendet hätte [2, S. 69]. Der Infix-Operator `<<<` beschreibt die Backwards-Komposition zweier Funktionen (g nach f) [7, S. 28].

7.1.2 Funktoren in PureScript

Dieser Unterabschnitt soll die Nutzung von Funktoren in PureScript an Beispielen verdeutlichen. Die Übersetzung nach JavaScript der hier aufgeführten Beispiele wird im nächsten Unterabschnitt aufgezeigt und besprochen.

Zunächst greifen wir in Listing 7.3 die, durch Bilder illustrierten, Beispiele mit dem Funktor *Maybe* des letzten Unterabschnitts auf. Die unäre Funktion wird jeweils auf ein *Just* von 2 und auf ein *Nothing* angewendet. Durch das in der Definition von *map* verwendete Pattern Matching wird der Wert 2 aus dem Funktor geholt, die Funktion mit ihm als Argument aufgerufen und das Ergebnis dessen in einen neuen Funktor gesteckt. Bei *Nothing* wird ohne Funktionsanwendung *Nothing* zurückgegeben.

```
1 module Main where
2
3 import Prelude
4 import Data.Maybe
5
6 x = map (\x -> x + 3) (Just 2) -- x = (Just 5)
7 y = map (\x -> x + 3) Nothing  -- y = Nothing
```

Listing 7.3: Beispiel mit Funktoren in PS

Arrays und auch Funktionen sind ebenfalls Funktoren. In Listing 7.4 wird zunächst eine unäre Funktion auf ein Array von Ganzzahlen angewendet. Wenn *map* als Wert und Funktor eine Funktion bekommt, wirkt sich dies wie eine Funktionskomposition aus.

```
1 module Main where
2
3 import Prelude
4 import Data.Array
5
6 a = map (\x -> x + 1) [1, 2, 3, 4, 5] -- a = [2,3,4,5,6]
7 b = map (\x -> x + 3) (\x -> x + 2)   -- (b 1) = 6
```

Listing 7.4: Weitere Funktoren in PS

Schauen wir uns zur Verdeutlichung der genauen Mechanik einmal die Implementierung der Typklasse Funktor im Falle der Instanzen für Arrays und Funktionen an. In Listing

7.5 sind die Typinstanzen und die Implementierung der Funktion *compose* angegeben. Diese wird in der Realität über mehrere Ebenen der Typklassenhierarchie geschliffen. Das soll an dieser Stelle aber nicht Gegenstand der Betrachtung sein. Die Implementierung der *map*-Funktion für Arrays wird über einen Foreign Function Import aus JavaScript bereitgestellt (Listing 7.6).

```
instance functorArray :: Functor
  Array where
  map = arrayMap

instance functorFn :: Functor
  ((->) r) where
  map = compose

-- imported over several modules
compose f g x = f (g x)
```

Listing 7.5: Typinstanzen für Arrays und Funktionen PS

```
exports.arrayMap = function (f) {
  return function (arr) {
    var l = arr.length;
    var result = new Array(l);
    for (var i = 0; i < l; i++) {
      result[i] = f(arr[i]);
    }
    return result;
  };
};
```

Listing 7.6: Native JavaScript Impl.

Bezüglich der Mechanik bei Arrays lässt sich nun sagen, dass ein neues Array (Funktor) *result* erzeugt wird, die Funktion f auf jedes Element des alten Arrays (Funktors) *arr* angewendet wird und das Resultat dieser Funktionsanwendung jeweils im neuen Funktor *result* abgelegt wird.

Die Mechanik bei Funktionen ist, wie es vom Verhalten der *map*-Anwendung bereits zu vermuten war, die der Komposition von Funktionen, wie sie auch in anderen Sachverhalten eingesetzt wird. Es wird eine Funktion f entgegen genommen, eine weitere Funktion g als Funktor und eine neue Funktion (Funktor) zurückgegeben, in der zunächst g auf den Wert x angewendet wird und anschließend f auf das Ergebnis davon. Das heißt in unserem konkreten Fall wird die Addition mit dem Wert 2 vor der Addition mit dem Wert 3 durchgeführt.

7.1.3 Übersetzung nach JavaScript

Im Folgenden wird die Übersetzung der Codebeispiele mit Funktoren des vorangegangenen Unterabschnitts dargestellt und erläutert. In Listing 7.8 ist die Übersetzung von dem Maybe-Listing 7.3 aufgeführt, in Listing 7.10 die des Array- und Funktionen-Listings 7.4.

Da es sich bei *Functor* um eine Typklasse handelt, ist das Übersetzungsmuster gemäß des in Abschnitt „5.3 Typklassen“ beschriebenen. Es wird also analog die Typklasse *Func-*

tor aus PureScript als Object Type (Klasse) in JavaScript repräsentiert. Die Typinstanz für den ADT *Maybe* ist ein Objekt (Type Class Dictionary) mit der Bezeichnung *functorMaybe*, welches die Implementierung der *map*-Funktion für *Maybe* definiert. In dieser wird auf die Übersetzung der Wertkonstruktoren *Just a* und *Nothing* des ADT *Maybe* zurückgegriffen. Deren Übersetzung gestaltet sich analog zur Übersetzung der Werkkonstruktoren des Listen-ADT's aus Listing 5.2 aus Abschnitt „5.2 Algebraische Datentypen". Zum besseren Verständnis der Mechanik seien die zuvor genannten Entitäten dennoch in Listing 7.7 aufgeführt. Auf weiterführende Beschreibungen wird allerdings verzichtet. Zur Reduzierung der Länge des Listings sind an geeigneten Stellen Zeilenumbrüche entfernt worden.

```
1  // ../Data.Functor/index.js
2  var Functor = function (map) { this.map = map; };
3
4  var map = function (dict) { return dict.map; };
5
6  // ../Data.Maybe/index.js
7  var functorMaybe = new Data_Functor.Functor(function (v) {
8    return function (v1) { // v1 ist ein Objekt, v eine Funktion
9      if (v1 instanceof Just) { return new Just(v(v1.value0)); };
10     return Nothing.value;
11   };
12 });
13
14 var Nothing = (function () { // kann oben als v1 auftreten
15   function Nothing() { };
16   Nothing.value = new Nothing();
17   return Nothing;
18 })();
19
20 var Just = (function () { // kann oben als v1 auftreten
21   function Just(value0) { this.value0 = value0; };
22   Just.create = function (value0) { return new Just(value0); };
23   return Just;
24 })();
```

Listing 7.7: Übersetzung der Typklasse *Functor* und Typinstanz *functorMaybe*

Richten wir unser Augenmerk nun auf Listing 7.8. Es wird in Zeile 4 ein neues *functorMaybe* Type Class Dictionary (Objekt in JavaScript, Typinstanz in PureScript) erzeugt. Dieses bekommt[1] in diesem Fall eine unäre Additionsfunktion (oben *v* genannt) und ein Objekt *Just* (oben *v1*) mit dem Wert 2 (oben *v1.value0*). Das Type Class Dictionary *functorMaybe* wird der freien Funktion *map* des Moduls *Data.Functor* als Argument übergeben.

[1] aufgrund des Currying's wird auch in diesen und weiteren Beispielen partielle Anwendung in JavaScript genutzt, weshalb manche Formulierung unpräzise ist, aber es steht gerade anderes im Vordergrund

Diese Funktion ruft anschließend die Klassenfunktion *map* auf, deren Implementierung die übergebene Additionsfunktion und das Funktorobjekt entsprechend des bereits zuvor beschriebenen Mechanismuses des Maybe-Funktors verwertet (Zeile 9, Listing 7.7). Der Fall mit dem Wertkonstruktor *Nothing* in Zeile 10 bis 12 aus Listing 7.8 ist im Aufrufverhalten gleich, wird aber unterschiedlich verwertet (Zeile 10, Listing 7.7).

```
1  var Data_Functor = require("../Data.Functor");
2  var Data_Maybe = require("../Data.Maybe");
3  var Data_Semiring = require("../Data.Semiring");
4  var Prelude = require("../Prelude");
5
6  var x = Data_Functor.map(Data_Maybe.functorMaybe)(function (x1) {
7    return x1 + 3 | 0;
8  })(new Data_Maybe.Just(2));
9
10 var y = Data_Functor.map(Data_Maybe.functorMaybe)(function (x1) {
11   return x1 + 3 | 0;
12 })(Data_Maybe.Nothing.value);
```

Listing 7.8: Übersetzung des Maybe-Funktorenbeispiels nach JS

Kommen wir nun zur Übersetzung von Arrays und Funktionen als Funktoren in Listing 7.10. Da Arrays ein in die Sprache PureScript direkt eingebauter Datentyp sind, ist das Aufrufverhalten einfacher als bei *Maybe*. Das Type Class Dictionary *functorArray* bekommt ohne Umschweife bei Erzeugung aus der Funktorklasse die Implementierung von *arrayMap* (Listing 7.6) übergeben. Da die exakte Umsetzung des Funktors *functorFn*, wie bereits zuvor erwähnt, weitere Typklassen einführen würde, sei auch an dieser Stelle in Listing 7.9 nur die Implementierung in JavaScript der *compose*-Funktion angeben. Das Aufrufverhalten gestaltet sich jedoch wieder analog zu dem von *Maybe*.

```
var functorArray = new Functor(
  $foreign.arrayMap);

var compose = function (f) {
  return function (g) {
    return function (x) {
      return f(g(x)); }; }; });
```

Listing 7.9: Auszug Funktor Array u. Function

```
1  var a = Data_Functor.map(Data_Functor.functorArray)(function (x) {
2    return x + 1 | 0;
3  })([ 1, 2, 3, 4, 5 ]);
4
5  var b = Data_Functor.map(Data_Functor.functorFn)(function (x) {
6    return x + 3 | 0;
7  })(function (x) {
8    return x + 2 | 0;
9  });
```

Listing 7.10: Übersetzung des Beispiels mit Array und Funktion als Funktor nach JS

7.2 Applicatives

Applicatives sind eine Subklasse von Funktoren und erweitern diese um zwei weitere Funktionen.

- apply - wie map, nur, dass auch die Funktion, die angewendet werden soll, bereits in einem Funktor steckt [7, S. 84]
- pure - um Werte in einen Funktor zu betten, in einen Berechnungskontext zu heben [7, S. 86]

In PureScript steht die Typklasse *Applicative* nicht direkt unter *Functor* in der Hierarchie, sondern *Functor* wird zunächst durch eine Typklasse names *Apply* erweitert und diese anschließend von *Applicative*. In Listing 7.11 wird dies dargestellt und auf welcher Hierarchieebene welche der beiden Funktionen wie definiert ist.

```
1 class Functor f where
2   map :: forall a b. (a -> b) -> f a -> f b
3
4 class Functor f <= Apply f where
5   apply :: forall a b. f (a -> b) -> f a -> f b
6
7 class Apply f <= Applicative f where
8   pure :: forall a. a -> f a
```

Listing 7.11: Typklasse Applicative

Bevor wir uns im Anschluss einem konkreten Beispiel mit Applicatives in PureScript und dessen Übersetzung nach JavaScript widmen, soll im nächsten Unterabschnitt das beschriebene Konzept zunächst visualisiert werden.

7.2.1 Applicatives in Bildern am Beispiel von Maybe

Bei der Visualisierung des Konzepts von Applicatives wird sich, wie zuvor bei Funktoren, des Algebraischen Datentyps *Maybe* als Beispiel bedient. In Listing 7.12 ist die Typinstanz für *Maybe* als Funktor nochmals wiederholt und die neuen Typinstanzen für die Typklassen *Apply* und *Applicative* aufgeführt.

```
1 instance functorMaybe :: Functor Maybe where
2   map fn (Just x) = Just (fn x)
3   map _  _        = Nothing
```

```
4
5  instance applyMaybe :: Apply Maybe where
6    apply (Just fn) x = fn <$> x
7    apply Nothing   _ = Nothing
8
9  instance applicativeMaybe :: Applicative Maybe where
10   pure = Just
```

Listing 7.12: Applicative-Intanz Maybe

An der Definition der Typinstanz *applyMaybe* ist zu sehen, dass die Funktion *apply* ihr erstes Argument in einem Funktor haben möchte. Das ist der Unterschied zur Funktion *map* bei Funktoren. Durch Abbildung 7.5 wird das Vorgehen visualisiert dargestellt. Für den Fall, dass das erste Argument eine Funktion in einem *Just* ist und das zweite Argument ein Wert in einem Funktor, werden beide durch das Pattern Matching ausgepackt. Anschließend wird die Funktion durch *map*, vertreten durch seinen Infix-Alias *<$>*, auf den Wert angewendet und das Resultat wieder in einen Funktor eingebettet [7, S. 84].

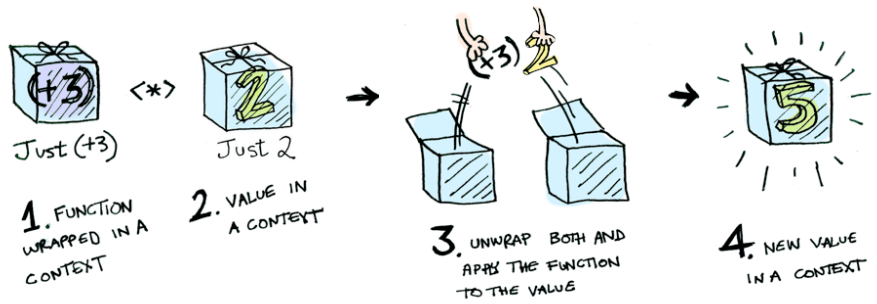


Abbildung 7.5: Funktionsanwendung aus einem Just auf ein Just von 2 [2]

Für den Fall, dass das erste Argument ein *Nothing* ist, wird, wie bei Funktoren auch, wieder ein *Nothing* zurückgeliefert. Bei Funktoren gab es keinen Wert, auf den die Funktion *fn* hätte angewendet werden können. Jetzt gibt es keine Funktion *fn*, da das erste Argument ein *Nothing* ist und den Wert ohne Funktionsanwendung einfach wieder zurückzugeben, würde zu einem Typfehler führen. Denn das zweite Argument ist vom Typ *a* bevor es durch eine Funktion von *a* nach *b* zu einem Typ von *b* wird. Ohne diese Funktion zu haben, bleibt uns keine andere Option als Nothing zu erzeugen.

In der Typinstanz *applicativeMaybe* wird die konstante Funktion *pure* für *Maybe* definiert, indem sie den Wertkonstruktor *Just* zugewiesen bekommt.

Die Typinstanzen der Typklasse *Applicative* bzw. *Apply* müssen bei der Implementierung der Funktionsdeklarationen ebenso wie Funktoren einige Gesetze einhalten. Diese sollen an dieser Stelle aber nicht im Detail besprochen werden.

7.2.2 Applicatives in PureScript

Dieser Unterabschnitt soll die Nutzung von Applicatives und vor allem ihre Vorzüge im Vergleich zu Funktoren an Beispielen verdeutlichen. Die Übersetzung nach JavaScript der hier aufgeführten Beispiele wird im nächsten Unterabschnitt besprochen.

In Listing 7.13 ist der bereits erwähnte Unterschied zu Funktoren demonstriert. Dort wird neben der Funktion *apply* auch deren Infix-Alias <*> genutzt. Im Gegensatz zu *map* dürfen sich die Funktionen ebenfalls bereits in einem Funktor befinden.

```
1 module Main where
2
3 import Prelude
4 import Data.Maybe
5
6 a = apply (Just \x -> x + 3) (Just 2)              -- (Just 5)
7 b = [(\x -> x * 2), (\x -> x + 2)] <*> [1, 2, 3] -- [2,4,6,3,4,5]
8 c = [(\x y -> x * y), (\x y -> x + y)] <*> pure 2  <*> [1, 2, 3]
```

Listing 7.13: Beispiel mit Applicatives in PS

Zeile 6 zeigt, dass *apply* im Grunde wie *map* funktioniert, wie wir auch schon anhand ihrer Definition erkennen konnten. Das Beispiel ist im Resultat äquivalent mit dem in Listing 7.3 bei Funktoren. Dadurch, dass die Funktion in einem Funktor sein darf, eröffnen sich aber auch Möglichkeiten, welche die Mächtigkeit von einfachen Funktoren übersteigen. Ein erstes Beispiel dafür ist in Zeile 7 zu sehen. Ein Array ist ebenso ein Funktor und kann damit, gefüllt mit Funktionen, genutzt werden. Als Auswirkung wird zunächst die erste Funktion auf die Elemente des Arrays, dass als Wert gedient hat, angewendet und im Anschluss die zweite Funktion auf dieselben Ursprungswerte. Als Ergebnis entsteht ein Array, dass die Resultate der beiden Funktionsanwendungen in sich vereint.

In Zeile 7 wird kurz die praktische Nutzung von *pure* demonstriert, was einen einfachen Wert in den Berechnungskontext hebt. Die vorherigen unären Funktionen sind jetzt binär, damit der Wert 2 und jeweils ein Element des Arrays wie zuvor in Zeile 6 miteinander verrechnet werden können. Es entsteht demnach das gleiche Array als Ergebnis.

Es deutete sich bereits an, dass die praktische Ausnutzung des Unterschieds von Funktoren zu Applicatives etwas mit der Anzahl der Argumente einer Funktion zu tun haben könnte. Dies soll nachfolgend, durch Listing 7.14 begleitet, bestätigt werden.

Wenn wir bei Funktoren eine unäre Funktion auf einen Wert in einem Berechnungskontext anwenden, erhalten wir einen Wert in einem neuen Berechnungskontext (Zeile 1). Wird allerdings mindestens eine binäre Funktion angewendet, ergibt dies durch die partielle Anwendung eine Funktion, die um ein Argument kürzer ist (Zeile 2). Dies ist kein wirklicher Unterschied zu Zeile 1, denn dort wurde durch die partielle Anwendung eine Funktion von einem Argument auf eine Funktion von keinem Argument reduziert, also auf

eine konstante Funktion bzw. Wert. In beiden Fällen haben wir prinzipiell das Problem: „Und wie solls jetzt **weitergehen**?“. Entweder wir legen selbst Hand an und holen unseren Wert oder Funktion aus dem Berechnungskontext durch Pattern Matching heraus, verwerten ihn und heben ihn wieder in einen neuen Kontext hoch (Zeile 4 - 6) oder wir bedienen uns etwas Vorgefertigtem.

```
1 a = (\x -> x + 3) <$> (Just 2)    -- a = (Just 5)
2 b = (\x y -> x + y) <$> (Just 2) -- b = (\y -> 2 + y)
3
4 c :: Maybe (Int -> Int) -> Int -> Maybe Int -- (c b 3) = (Just 5)
5 c (Just f) y = Just(f y)
6 c Nothing y = Nothing
7
8 d = lift2 (\x y -> x + y) (Just 2) (Just 3) -- d = (Just 5)
```

Listing 7.14: Grenzen von Funktoren

In Zeile 8 wird durch die Funktion *lift2* aus dem Modul *Control.Apply* eine Funktion, die zwei Argumente annimmt, mit diesen Argumenten in einen Berechnungskontext gehoben, die Funktion auf diese angewendet und das Resultat in einen neuen Berechnungskontext gepackt. Schön und gut, aber was ist, wenn die Funktion drei Argumente annimmt? In diesem Modul gibt es Lift-Funktionen für Funktionen bis zu fünf Argumenten, aber etwas umständlich ist es dennoch noch und wir sind nicht in der Lage Funktionen mit einer beliebigen Anzahl Argumente zu behandeln.

Schauen wir uns in Listung 7.15 exemplarisch an, wie die Lift-Funktionen definiert sind. Es fällt auf, dass die Definition einem strikten Schema folgen, was wir statisch unendlich so weiterführen könnten. Alternativ könnten wir uns aber auch eine Funktion *lift1* definieren und diese durch mehrfache partielle Anwendung auf Funktionen einer beliebigen Anzahl Argumente anwenden - solange wie es für einen spezifischen Anwendungsfall von Nöten ist.

```
1 lift2 :: forall a b c f. Apply f => (a -> b -> c) -> f a -> f b ->
     f c
2 lift2 f a b = f <$> a <*> b
3
4 lift3 :: forall a b c d f. Apply f => (a -> b -> c -> d) -> f a ->
     f b -> f c -> f d
5 lift3 f a b c = f <$> a <*> b <*> c
```

Listing 7.15: Auszug aus *Control.Apply*

Die Funktion *lift1* gibt es bereits und nennt sich *apply*. Sie ist neben der Funktion *pure*, die wenn man so will Funktionen mit keinen Argumenten in einen Berechnungskontext hochhebt, das was die Typklasse *Applicative* ausmacht und alles was wir brauchen.

Listing 7.16 zeigt zur Verdeutlichung nochmal die Funktionsdeklaration von *apply* und

anhand der Beispiele aus Listing 7.14 wie sie sich bequem deklarativ, ohne Boilerplate-Code, durch Applicatives formulieren lassen.

```
1 -- apply :: forall a b. f (a -> b) -> f a -> f b
2
3 a = (\x y -> x + y) <$> (Just 2) <*> (Just 3) -- a = (Just 5)
4 b = (\x y z -> x + y + z) <$> (Just 2) <*> (Just 3) <*> (Just 5)
5 -- b = (Just 10)
```

Listing 7.16: Aneinanderkettung von Berechnungen mit Applicatives in PS

Applicatives sind also einfachen Funktoren bei Funktionen mit mehr als einem Argument überlegen. Durch die Kombination von *map* und *apply* können Funktionen mit beliebiger Anzahl ungebundener Argumente auf die gleiche Anzahl von Werten, die bereits in einem Berechnungskontext gebunden sind, angewendet werden. Das Ergebnis der sukzessive partiellen Funktionsanwendung ist wie bei Funktoren ein Wert in einem neuen Berechnungskontext. So scheint es kein Zufall zu sein, dass Funktoren in dem Modul *Data* definiert sind und Applicatives im Modul *Control*, da sie zusätzlich doch eine generische Art und Weise vorgeben, wie herkömmliche Funktionen elegant mit eingepackten Werten umgehen können. Sie erlauben im Gegensatz zu einfachen Funktoren eine Aneinanderreihung von Berechnungsschritten unter dem Standpunkt der funktionalen Sicht auf Seiteneffekte.

7.2.3 Übersetzung nach JavaScript

In diesem Unterabschnitt wird die Übersetzung einiger Codebeispiele mit Applicatives des vorangegangenen Unterabschnitts dargestellt und erläutert. Diese sind in Listing 7.19 nochmal zusammengefasst aufgeführt.

Da es sich bei Applicatives um eine Typklasse handelt, ist das Übersetzungsmuster gemäß des in Abschnitt „5.3 Typklassen“ beschriebenen. Es wird also analog die Typklasse *Applicative* aus PureScript als Object Type (Klasse) in JavaScript repräsentiert. Die Typinstanz für den ADT *Maybe* ist ein Objekt (Type Class Dictionary) mit der Bezeichnung *applicativeMaybe*. Da in der Übersetzung auf die Wertkonstruktoren *Just a* und *Nothing* des ADT *Maybe* und auf die von Funktoren zurückgegriffen wird, lohnt ein Blick in Listing 7.7. Applicatives stehen in der Typklassenhierarchie unter *Apply* und *Functor*. Diese Beziehung wird über die Übersetzung ebenfalls deutlich. Zur Reduzierung der Länge der nachfolgenden Listings sind an geeigneten Stellen Zeilenumbrüche entfernt worden.

In Listing 7.17 ist die Übersetzung der Typklassen *Apply* und *Applicative* nach JavaScript dargestellt. In *Apply* wird dazu auf die Übersetzung der Typklasse *Functor* zurückgegriffen und in *Applicative* auf die von *Apply*. An dieser Stelle wird also die Beziehung zwischen diesen Typklassen deutlich und auch, dass ein Applicative auch immer ein Funktor ist, da es über die gleichen Fähigkeiten verfügt.

```
// ../Control.Apply/index.js
var Apply = function (Functor0, apply) {
  this.Functor0 = Functor0;
  this.apply = apply;
};
var apply = function (dict) { return dict.apply; };

// ../Control.Applicative/index.js
var Applicative = function (Apply0, pure) {
  this.Apply0 = Apply0;
  this.pure = pure;
};
var pure = function (dict) { return dict.pure; };
```

Listing 7.17: Übersetzung der Typklassen *Apply* und *Applicative*

Listing 7.18 zeigt, analog zu Funktoren in Listing 7.7, die Typinstanzen für *Apply* und *Applicative* im Falle des ADT's *Maybe*.

```
// ../Data.Maybe/index.js
var applyMaybe = new Control_Apply.Apply(
  function () { return functorMaybe; }, function (v) {
  return function (v1) { // v1 ist ein Objekt, v eine Funktion, ..
    if (v instanceof Just) { // nun auch in einem Objekt gekapselt
      return Data_Functor.map(functorMaybe)(v.value0)(v1);
    };
    if (v instanceof Nothing) {
      return Nothing.value;
    };
    throw new Error("<pattern error message skipped by author>");
  };
});

var applicativeMaybe = new Control_Applicative.Applicative(
  function () { return applyMaybe; }, Just.create);
```

Listing 7.18: Übersetzung der Typinstanzen für *Maybe*

Das Vorgehen unterscheidet sich nicht wesentlich von dem, welches wir bei Funktoren vorgefunden haben. Wegen der Typklassenhierarchie wird auch hier wieder auf die Typinstanzen des übergeordneten Typs zugegriffen. Die Funktion *pure* wird zu einer Erstellung von einem *Just* (Zeile 16). Die Funktion *apply* ist, im Falle eines *Just*, über den Aufruf der Funktion *map* von Funktoren definiert, indem es dieser die ausgepackte Funktion (*v.value0*) und den noch eingepackten Wert (*v1*) übergibt. Im Fall von *Nothing* passiert das gleiche wie bei Funktoren. Da es für einen anderen Fall kein definiertes Verhalten gibt,

wird in einem solchen ein Fehler geworfen.

Die Übersetzung der in Listing 7.19 zusammengefassten Codebeispiele werden im Folgenden nacheinander besprochen. Es werden nur für das Verständnis nötige Imports einmalig beim ersten Listing aufgeführt.

```
a = [(\x -> x * 2), (\x -> x + 3)] <*> [1, 2, 3]
b = (\x y -> x + y) <$> (Just 2) <*> (Just 3)
c = (\x y z -> x + y + z) <$> (Just 2) <*> (Just 3) <*> (Just 5)
```

Listing 7.19: Applicatives in PS

Die Übersetzung des Beispiels unter Verwendung von Arrays ist in Listing 7.21 dargestellt. Wie bereits im Abschnitt über Funktoren erwähnt, ist das Aufrufverhalten beim fest in die Sprache eingebauten Typ *Array* einfacher als bei Typen wie *Maybe*. Analog wie in Listing 7.9 bereits dargestellt, bekommt das Type Class Dictionary *applyArray* bei Erzeugung aus der Applyklasse direkt die Implementierung von *arrayApply* übergeben. Diese ist in Listing 7.20 aufgeführt.

```
exports.arrayApply = function (fs) {
  return function (xs) {
    var l = fs.length;
    var k = xs.length;
    var result = new Array(l*k);
    var n = 0;
    for (var i = 0; i < l; i++) {
      var f = fs[i];
      for (var j = 0; j < k; j++) {
        result[n++] = f(xs[j]);
      }
    }
    return result;
  };
};
```

Listing 7.20: Auszug Control.Apply

Das Beispiel soll verdeutlichen wie ein einfacher Aufruf von *apply* realisiert wird. In Zeile 7 wird ein neues *applyArray* Type Class Dictionary (Objekt) erzeugt. Dieses hält die beiden, zuvor in einem Array abgelegten, Funktionen. Darüber hinaus bekommt es das Array von drei Werten übergeben. Das Type Class Dictionary wird der freien Funktion *apply* des Moduls *Control.Apply* als Argument überreicht. Diese Funktion ruft anschließend die Klassenfunktion *apply* auf, was zum Aufruf von *arrayApply* aus Listing 7.20 führt.

```
var Control_Apply = require("../Control.Apply");
var Data_Functor = require("../Data.Functor");
var Data_Maybe = require("../Data.Maybe");

var a = Control_Apply.apply(Control_Apply.applyArray)
([ function (x) {
  return x * 2 | 0;
}, function (x) {
  return x + 3 | 0;
} ])([ 1, 2, 3 ]);
```

Listing 7.21: Übersetzung von Applicatives mit Arrays nach JS

In der Funktion *arrayApply* wird das Array von Funktionen *fs* und das Array von Werten *xs* angenommen. In einer ersten Schleife wird sich die jeweilige Funktion herausgenommen. In einer zweiten geschachtelten Schleife wird diese Funktion auf alle Argumente angewendet und die Resultate in einem neuen Array gespeichert. Darauf folgt die Anwendung aller weiteren Funktionen auf dieselbe Weise.

Im nachfolgenden Listing 7.22 soll verdeutlicht werden, wie ein Aufruf von *map* in Verbindung mit einem Aufruf von *apply* übersetzt wird.

```
var b = Control_Apply.apply(Data_Maybe.applyMaybe)
(Data_Functor.map(Data_Maybe.functorMaybe)
(function (x) {
  return function (y) {
    return x + y | 0;
  };
})(new Data_Maybe.Just(2)))(new Data_Maybe.Just(3));
```

Listing 7.22: Übersetzung von Applicatives mit Maybe nach JS

Die *map*-Funktion bekommt das *Just* von 2 und dieses wird an den Parameter *x* gebunden. Im Anschluss bekommt die *apply*-Funktion die in einem *Just* gekapselte partiell aufgerufene Funktion des *map*-Aufrufs sowie das *Just* von 3 und ersetzt den Parameter *y* mit diesem.

Bei weiteren *apply*-Aufrufen wird, wie in Listing 7.23 zu sehen, ein weiteres *applyMaybe* Objekt nach dem gleichen Schema in den Aufruf integriert.

```
var c = Control_Apply.apply(Data_Maybe.applyMaybe)
(Control_Apply.apply(Data_Maybe.applyMaybe)
(Data_Functor.map(Data_Maybe.functorMaybe)
(function (x) {
  return function (y) {
    return function (z) {
      return (x + y | 0) + z | 0;
    };
  };
})(new Data_Maybe.Just(2)))(new Data_Maybe.Just(3)))(new Data_Maybe
  .Just(5));
```

Listing 7.23: Übersetzung von Applicatives mit Maybe nach JS

7.3 Monaden

Monaden sind eine Subklasse von Applicatives und erweitern diese um eine Funktion *bind*. Kurz zur Wiederholung und zur Intention von *bind*:

- Funktoren erlauben es durch *map*, eine Funktion auf einen Wert in einem Funktor anzuwenden.
- Applicatives erlauben es durch *apply*, eine Funktion in einem Funktor auf einen Wert in einem Funktor anzuwenden. Dadurch wird die Aneinanderreihung von Funktionsberechnungen möglich.
- Monaden erlauben es durch *bind*, eine Funktion entgegen zu nehmen, die einen kontextlosen Wert nimmt und einen Wert in einem Kontext (Monade) zurückliefert. Die anzuwendende Funktion darf also im Gegensatz zu *map* oder *apply* bereits einen eingepackten Wert zurückliefern. Dadurch wird die Aneinanderreihung von Funktionsberechnungen ermöglicht, wo in jeweils jeder Funktionsberechnung entschieden werden kann, welche Berechnung auf der Basis des Ergebnisses der vorangegangen Berechnung ausgeführt werden soll. Der resultierende spezifische Berechnungskontext hängt also lediglich von der Auswertung der Funktion ab, nicht von der Definition des Pattern Matchings des Funktors (siehe bspw. Listing 7.25).

In PureScript gibt es auf der gleichen Hierarchieebene wo sich sich die Typklasse *Applicative* befindet eine Typklasse *Bind*, welche die Funktion *bind* deklariert. Die Typklasse Monad erweitert eine Ebene darunter diese beiden Typklassen und bringt deren Funktionalität damit zusammen, wie in Listing 7.24 zu sehen [7, S. 101]. In der Bibliothek wird im Zusammenhang mit Monaden der Funktor in deren Funktionsdeklaration mit dem Buchstaben m anstatt f ausgedrückt, was aber keinen Unterschied macht.

```
1  class Functor f where
2    map :: forall a b. (a -> b) -> f a -> f b
3
4  class Functor f <= Apply f where
5    apply :: forall a b. f (a -> b) -> f a -> f b
6
7  class Apply f <= Applicative f where
8    pure :: forall a. a -> f a
9
10 class Apply m <= Bind m where
11   bind :: forall a b. m a -> (a -> m b) -> m b
12
13 class (Applicative m, Bind m) <= Monad m
```

Listing 7.24: Typklasse Monad

Bevor wir uns im Anschluss einem konkreten Beispiel mit Monaden in PureScript und dessen Übersetzung nach JavaScript widmen, soll im nächsten Unterabschnitt das beschriebene Konzept zunächst visualisiert werden.

7.3.1 Monaden in Bildern am Beispiel von Maybe

Bei der Visualisierung des Konzepts von Monaden wird sich, wie zuvor bei Funktoren und Applicatives, des Algebraischen Datentyps *Maybe* als Beispiel bedient. In Listing 7.25 ist die Typinstanz für *Maybe* als Funktor und Applicative nochmals wiederholt und die neuen Typinstanzen für die Typklassen *Bind* und *Monad* aufgeführt.

```
1  instance functorMaybe :: Functor Maybe where
2    map fn (Just x) = Just (fn x)
3    map _  _        = Nothing
4
5  instance applyMaybe :: Apply Maybe where
6    apply (Just fn) x = fn <$> x
7    apply Nothing   _ = Nothing
8
9  instance applicativeMaybe :: Applicative Maybe where
10   pure = Just
11
12 instance bindMaybe :: Bind Maybe where
13   bind (Just x) k = k x
14   bind Nothing  _ = Nothing
15
16 instance monadMaybe :: Monad Maybe
```

Listing 7.25: Monad-Instanz Maybe

An der Typinstanz *bindMaybe* ist zu sehen, dass wir einen Wert in dem Monadentyp *Just* oder *Nothing* erwarten. Durch Abbildung 7.6 wird das Vorgehen für den ersten Fall visualisiert dargestellt. Die Funktion *bind* bekommt einen Wert in einer Monade (Funktor) als erstes Argument. Durch Pattern Matching wird der Wert ausgepackt. Das zweite Argument ist eine beliebige Funktion, die einen Wert außerhalb eines Funktors erwartet und einen Wert in einem Funktor zurückliefert. Der Rückgabewert der Funktionsanwendung ist gleichzeitig der Rückgabewert von *bind*.

Das bedeutet die Funktion, die als Argument angeben wird, entscheidet **wie** es **weitergeht** - was für ein Funktor genau zurückgegeben wird. Bei den Funktionen *map* und *apply* war das durch deren Fallunterscheidung bereits entschieden. Für den Fall von *Maybe* in Listing 7.25 ist in Zeile 13 nicht entschieden, dass wenn wir ein *Just* von etwas bekommen auch wie bei *map* und *apply* ein *Just* resultieren muss (Zeile 2 und 6). Die Funktion *k* könnte auch ein *Nothing* zurückliefern. In Abbildung 7.6 ist bspw. eine Funktion *half* definiert, wie in Listing 7.26 dargestellt.

```
1 half :: Int -> Maybe Int
2 half x = if even x
3   then Just (div x 2)
4   else Nothing
```

```
5
6 retNothing = Just 20 >>= half >>= half >>= half
```

Listing 7.26: Funktion mit Monad-Anwendung

Durch *half* entscheidet sich, ob der Aufruf von *bind* ein *Just* oder ein *Nothing* zurückliefern wird. Wie in Zeile 6 des Listings zu sehen, kann man dadurch eine direkte Verkettung von *bind*-Aufrufen angeben. Die Funktion *bind* wird dort durch deren Infix-Alias »= vertreten [7, S. 101]. Aus dem ersten Aufruf resultiert ein *Just 10*, wird im zweiten Aufruf von *bind* ausgepackt und erneut in die Funktion *half* gesteckt. Das Resultat davon ist *Just 5*, was bei, dritten Aufruf von *bind* in einem *Nothing* endet. Alle erneuten Anwendungen würden immer in einem *Nothing* enden, da die Funktion *half* per Typinstanzdefinition nicht mehr angewendet, sondern gleich *Nothing* zurückgegeben wird.

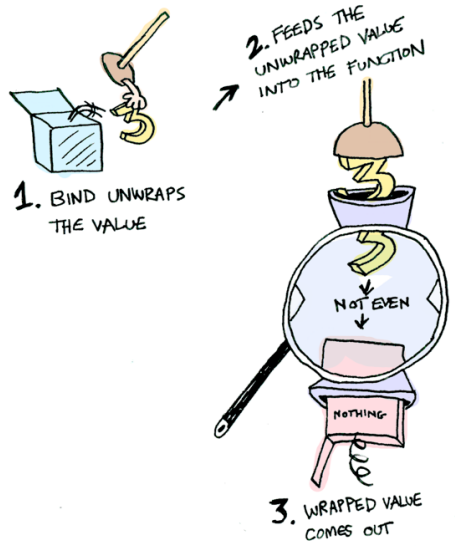


Abbildung 7.6: Funktionsanwendung auf ein Just von 3, wo der Ausgang der Funktionsanwendung, die Rückgabe von *bind* bestimmt [2]

Das bedeutet, während in der Aneinanderreihung von Berechnungen bei Applicatives die Ausgangsfunktion statisch war und der Fokus eher auf sich verändernden Werten lag (Listing 7.16), stehen bei Monaden sich potenziell verändernde Funktionsangaben im Vordergrund. Der Ausgangswert ist hier statisch und die resultierenden Werte sind von den angegebenen Funktionen abhängig. In Listing 7.26 Zeile 6 könnte bspw. auch eine andere Funktion des gleichen Funktortyps an die Stelle eines oder mehrerer *half*-Angaben treten.

Die Typinstanzen der Typklasse *Monad* bzw. *Bind* müssen bei der Implementierung der Funktionsdeklarationen ebenso wie Funktoren und Applicatives einige Gesetze einhalten. Diese werden im Nächsten Unterabschnitt mit der Einführung der Do-Notation besprochen.

7.3.2 Monaden in PureScript

Dieser Unterabschnitt soll die Nutzung von Monaden in PureScript an einem Beispiel verdeutlichen. Dessen Übersetzung nach JavaScript wird im nächsten Unterabschnitt besprochen.

In Listing 7.27 ist das Beispiel mit Monaden des vorherigen Unterabschnitts aus Listing 7.26 etwas angepasst. Anstatt, dass beim dritten Aufruf der Funktion *half* ein Nothing resultiert, sorgt der zwischengeschobene Aufruf der Funktion *decIfOdd* dafür, dass ein *Just 2* als Ergebnis berechnet werden kann. Anstatt explizit *Just 20* wie zuvor zu verwenden, wird an dieser Stelle der Wert 20 über *pure* in den Berechnungskontext gehoben.

```
1  module Main where
2
3  import Prelude
4  import Data.Maybe
5
6  even x = mod x 2 == 0
7
8  decIfOdd :: Int -> Maybe Int
9  decIfOdd x = if even x
10   then Just(x)
11   else Just(x - 1)
12
13 half :: Int -> Maybe Int
14 half x = if even x
15   then Just (div x 2)
16   else Nothing
17
18 -- Ergebniskette: Just 20 - Just 10 - Just 5 - Just 4 - Just 2
19 ret = pure 20 >>= half >>= half >>= decIfOdd >>= half
```

Listing 7.27: Funktionen mit Monad-Anwendung

Hiermit sei noch einmal der Unterschied zu Applicatives verdeutlicht, dass bei Monaden, in der Aneinanderreihung von Funktionen, entschieden werden kann, welche Berechnung auf der Grundlage des Ergebnisses der vorherigen Berechnung ausgeführt werden soll. Aufgrund dessen, dass ein Funktor einen Typ hat, müssen die Funktionen einer Aneinanderreihung alle vom gleichen Funktor, wie hier z.B. *Maybe*, sein.

Für den *bind*-Funktionsalias bietet PureScript syntaktischen Zucker in Form der Do-Notation an. In Listing 7.28 ist der äquivalente Aufruf in Do-Notation zu dem des vorherigen Listings 7.27 aufgeführt.

```
1  ret = do
2    zwanzig <- pure 20                    -- Syntax pattern:
3    zehn <- half zwanzig                  -- x >>= f = do y <- x
4    fuenf <- half zehn                    --              f y
5    vier <- decIfOdd fuenf
6    half vier
```

Listing 7.28: Funktionen mit Monad-Anwendung in Do-Notation

Für Monaden gelten drei Gesetze, welche sich unmittelbar auf mögliche Formulierungen in der Do-Notation auswirken und damit recht deutlich veranschaulicht werden können:

- Right Identiy: `x >>= pure = x`
- Left Identiy: `pure x >>= f = f x`
- Associative Composition: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Das Right-Identiy-Gesetz sagt aus, dass *pure* weggelassen werden kann, wenn er der letzte Ausdruck ist [7, S. 102]. Es macht also keinen Unterschied, ob man in Listing 7.28 den Ausdruck in Zeile 6 zuvor an eine weitere Konstante gebunden (z.B. *zwei*) und dann in der nächsten Zeile *pure zwei* angefügt hätte.

Das Left-Identity-Gesetz sagt aus, dass *pure* weggelassen werden kann, wenn er der erste Ausdruck ist [7, S. 102–103]. Es macht also keinen Unterschied, ob man in Listing 7.28 den Ausdruck in Zeile 2 nicht an die Konstante *zwanzig* gebunden hätte, sondern die *20* anstelle der *zwanzig* direkt in Zeile 3 hinter die Funktion *half* schreiben würde.

Das Associative-Composition-Gesetzt sagt aus, dass wenn man eine Abfolge von monadischen Funktionsanwendungen hat, es keinen Unterschied macht, wie diese miteinander verschachtelt sind [7, S. 103]. Bezüglich der Do-Notation hat das die Auswirkung, dass *c1* und *c2* aus Listing 7.29 äquivalent sind.

```
c1 = do
  y <- do
    x <- m1
    m2
  m3

c2 = do
  x <- m1
  y <- m2
  m3
```

Listing 7.29: Associative Composition

7.3.3 Übersetzung nach JavaScript

In diesem Unterabschnitt wird die Übersetzung des Codebeispiels mit der *Maybe*-Monade des vorangegangenen Unterabschnitts dargestellt und erläutert.

Da es sich bei Monaden um eine Typklasse handelt, ist das Übersetzungsmuster gemäß des in Abschnitt „5.3 Typklassen“ beschriebenen. Es wird also analog die Typklasse *Monad* aus PureScript als Object Type (Klasse) in JavaScript repräsentiert. Die Typinstanz für den ADT *Maybe* ist ein Objekt (Type Class Dictionary) mit der Bezeichnung *monadMaybe*. Da in der Übersetzung auf die Wertkonstruktoren *Just a* und *Nothing* des ADT *Maybe* und auf die von Funktoren zurückgegriffen wird, lohnt ein Blick in Listing 7.7. Monaden stehen in der Typklassenhierarchie unter *Applicative* und *Bind*. Diese Beziehung wird über die Übersetzung ebenfalls deutlich. Zur Reduzierung der Länge der nachfolgenden Listings sind an geeigneten Stellen Zeilenumbrüche entfernt worden.

In Listing 7.30 ist die Übersetzung der Typklassen *Bind* und *Monad* aus Listing 7.24 nach JavaScript dargestellt. In *Bind* wird dazu auf die Übersetzung der Typklasse *Ap-*

ply zurückgegriffen und in *Monad* auf die von *Applicative* und *Bind*. An dieser Stelle wird also die Beziehung zwischen diesen Typklassen deutlich und auch, dass eine Monade auch immer ein Applicative und damit ebenfalls ein Funktor ist, da sie über die gleichen Fähigkeiten verfügt.

```
1  // ../Control.Bind/index.js
2  var Bind = function (Apply0, bind) {
3    this.Apply0 = Apply0;
4    this.bind = bind;
5  };
6  var bind = function (dict) { return dict.bind; };
7
8  // ../Control.Monad/index.js
9  var Monad = function (Applicative0, Bind1) {
10   this.Applicative0 = Applicative0;
11   this.Bind1 = Bind1;
12 };
```

Listing 7.30: Übersetzung der Typklassen *Bind* und *Monad*

Listing 7.31 zeigt, die Übersetzung der Typinstanzen für *Bind* und *Monad* im Falle des ADT's *Maybe* aus Listing 7.25.

```
1  // ../Data.Maybe/index.js
2  var bindMaybe = new Control_Bind.Bind(
3  function () {
4    return applyMaybe;
5  }, function (v) {  // v1 ist ein Objekt, v eine Funktion, die
6    return function (v1) { // einen ausgepackten Wert erwartet
7      if (v instanceof Just) {
8        return v1(v.value0);
9      };
10     if (v instanceof Nothing) {
11       return Nothing.value;
12     };
13     throw new Error("<pattern error message skipped by author>");
14   };
15 });
16
17 var monadMaybe = new Control_Monad.Monad(
18 function () {
19   return applicativeMaybe;
20 }, function () {
21   return bindMaybe;
22 });
```

Listing 7.31: Übersetzung der Typinstanzen für *Maybe*

Das Vorgehen unterscheidet sich nicht wesentlich von dem, welches wir bei Funktoren und Applicatives vorgefunden haben. Wegen der Typklassenhierarchie wird auch hier wieder auf die Typinstanzen des übergeordneten Typs zugegriffen. Die Funktion *bind* nimmt ein JavaScript-Objekt als Funktor (*v1*) und eine Funktion (*v*) entgegen. Im Falle von *Just* wird der Wert im Objekt ausgepackt der Funktion für eine Berechnung übergeben und deren bereits eingepackte Rückgabe anschließend einfach direkt zurückgegeben. Im Fall von *Nothing* passiert das gleiche wie bei Funktoren und Applicatives. Da es für einen anderen Fall kein definiertes Verhalten gibt, wird in einem solchen ein Fehler geworfen.

In Listing 7.32 ist die Übersetzung des Beispiels mit dem ADT *Maybe* als Monade aus Listing 7.27 nach JavaScript aufgeführt. Auf die Übersetzung der genutzten Berechnungsfunktion sei an dieser Stelle verzichtet.

```
1 var ret =
2   Control_Bind.bind(Data_Maybe.bindMaybe)
3   (Control_Bind.bind(Data_Maybe.bindMaybe)
4     (Control_Bind.bind(Data_Maybe.bindMaybe)
5       (Control_Bind.bind(Data_Maybe.bindMaybe)
6         (Control_Applicative.pure(Data_Maybe.applicativeMaybe)
7   (20))(half))(half))(decIfOdd))(half);
```

Listing 7.32: Übersetzung von Monaden mit Maybe nach JS

Das Type Class Dictionary *bindMaybe* wird der freien Funktion *bind* des Moduls *Control.Bind* als Argument überreicht. Diese Funktion ruft anschließend die Klassenfunktion *bind* auf. Der Aufruf von *pure* geschieht auf äquivalente Weise. Die Funktionen sind in umgekehrter Reihenfolge zu ihren Argumenten für die Funktionswendung verschachtelt. Soll heißen, der Wert 20 wird von der innersten Funktion *pure* (Zeile 6) angenommen und darauf folgt das erste *half* für die zweitinnerste Funktion (Zeile 5) usw.

Listing 7.33 zeigt die Übersetzung des Beispiels mit dem ADT *Maybe* als Monade aus Listing 7.28 in Do-Notation nach JavaScript.

```
1  var ret = Control_Bind.bind(Data_Maybe.bindMaybe)
2  (Control_Applicative.pure(Data_Maybe.applicativeMaybe)(20))
3  (function (v) {
4    return Control_Bind.bind(Data_Maybe.bindMaybe)(half(v))
5    (function (v1) {
6      return Control_Bind.bind(Data_Maybe.bindMaybe)(half(v1))
7      (function (v2) {
8        return Control_Bind.bind(Data_Maybe.bindMaybe)(decIfOdd(v2))
9        (function (v3) {
10         return half(v3);
11       });
12     });
13   });
```

```
14 });
```

Listing 7.33: Übersetzung von Monaden mit Maybe nach JS in Do-Notation

Es ist zu erkennen, dass sich ein ähnliches Muster wie bei einfacher *bind*-Anwendung ergibt. Die Abweichungen resultieren aus dem Aspekt, dass die Zwischenergebnisse der Aufrufe in der Do-Notation benannt sind, was über Funktionen mit einem Argument realisiert wird. Deshalb ist die Reihenfolge der verschachtelten Funktionen und Type Class Dictionaries wieder wie sie auch in PureScript angegeben war.

7.4 Lessons Learned

- Funktoren erlauben es durch *map*, eine Funktion auf einen Wert in einem Funktor anzuwenden. Ein Weitermachen ist nicht definiert.
- Applicatives erlauben es durch *apply*, eine Funktion in einem Funktor auf einen Wert in einem Funktor anzuwenden. Ein **Weitermachen** ist definiert, aber dieses nimmt keinen Bezug auf das Ergebnis einer vorangegangen Berechnung.
- Monaden erlauben es durch *bind*, eine Funktion entgegen zu nehmen, die einen kontextlosen Wert nimmt und einen Wert in einem Kontext (Funktor bzw. hier Monade) zurückliefert. Ein **Weitermachen** ist definiert und um zu entscheiden **wie** weitergemacht werden soll, kann das Ergebnis einer vorangegangenen Berechnung als Ausgangswert genutzt werden.

8 | IO - Hallo Welt

PureScript unterscheidet zwischen nativen und nicht-nativen Seiteneffekten. Unter nativen Seiteneffekten werden solche verstanden, die aus JavaScript herrühren. Beispielsweise zählt darunter die Ein- und Ausgabe über die Konsole, Generierung von Zufallszahlen, Exceptions oder auch DOM Manipulationen. Ein Beispiel für einen Nicht-nativen Seiteneffekt wäre die Nutzung des ADT's *Maybe* [7, S. 107–108].

Die Monade, welche mit nativen Seiteneffekte umgehen soll, nennt sich *Eff* und ist im Modul *Control.Monad.Eff* definiert. An dieser Stelle sei erwähnt, dass wir in diesem Abschnitt nur an der Oberfläche kratzen werden, um den nachfolgenden, in Listing 8.1 dargestellten, Code verstehen zu können. Für Details sei auf die angegebene Literatur verwiesen.

```
1  module Main where
2
3  import Prelude (Unit, show, bind, (<>))
4  import Control.Monad.Eff (Eff)
5  import Control.Monad.Eff.Console (CONSOLE, log)
6  import Control.Monad.Eff.Random (RANDOM, random)
7
8  main :: forall eff. Eff (console :: CONSOLE, random :: RANDOM | eff
       ) Unit
9  main = do
10   n <- random
11   log ((show n) <> " - Hello world!")
```

Listing 8.1: Hello World Beispiel

Fangen wir bei der Typdeklaration der Main-Funktion an. Im Gegensatz zu Haskell, kann in PureScript sehr granular bestimmt werden, welche Seiteneffekte *main* haben darf. Die Funktion wird die Konsole nutzen und zusätzlich Zufallszahlen generieren [7, S. 108–109]. Das Akronym *eff* steht für *Extensible Effects*. Die Typdeklaration beschreibt vor dem Pipe-Zeichen, welche Berechnungen mit Seiteneffekten auftreten dürfen. Solange diese unterstützt werden, können jedoch auch noch anderweitige Seiteneffekte auftreten. Die Typdeklaration liest sich in diesem konkreten Fall wie folgt: „Die Funktion *main* ist eine Berechnung mit Seiteneffekten, die in einem Berechnungskontext ausgeführt werden, welcher Ein- und Ausgabeoperationen auf der Konsole und die Generierung von Zufallszahlen

unterstützt, und jede andere Art von Seiteneffekten, und wo ein Wert vom Typ *Unit* zurückgegeben wird.“ [7, S. 109–110]. Ohne darauf näher eingehen zu wollen - diese Technik, nennt sich in PureScript *Row Polymorphism* und steht in engem Bezug zu PureScript's Kind-System. So wie Werte nach Typen klassifiziert werden können, können Typen nach ihrer Art (kind) klassifiziert werden [7, S. 110–112].

Man kann den Typ der Funktion auch ohne *Row Polymorphism* ausdrücken. Dies wird dann (eingedeutscht) geschlossene Zeile von Effekten genannt, da die Row-Variable *eff* nicht genutzt wird. Die Typdeklaration sähe so aus:
`main :: Eff (console :: CONSOLE, random :: RANDOM) Unit.`
Dies hat den Vorteil, dass nicht aus Versehen eine Subberechnung von einem Seiteneffekt eines anderen Typs auftreten kann und den Nachteil, dass wir immer alle Seiteneffekte ins Detail festlegen müssen [7, S. 112].

Die Monade *Eff* ist im Grunde aufgebaut wie wir es bereits bei *Maybe* im vorangegangen Kapitel über Funktoren, Applicatives und Monaden gesehen haben. Sie stellt allerdings mehr eine typsichere API für Berechnungen mit Seiteneffekten dar, da es das Ziel ist effizienten JavaScript Code zu generieren. Die Geschäftslogik der Funktionen wird fast ausschließlich über Foreign Function Imports realisiert und ist damit in JavaScript geschrieben [7, S. 109].

In Listing 8.2 ist, unter Verzicht auf manche Imports, die Übersetzung des Hallo-Welt-Beispiels nach JavaScript dargestellt.

```
1 var Control_Monad_Eff_Console = require("../Control.Monad.Eff.
    Console");
2 var Control_Monad_Eff_Random = require("../Control.Monad.Eff.Random
    ");
3 var Data_Show = require("../Data.Show");
4
5 var main = function __do() {
6   var v = Control_Monad_Eff_Random.random();
7   return Control_Monad_Eff_Console.log(
8     Data_Show.show(Data_Show.showNumber)(v) + " - Hello world!")();
9 };
```

Listing 8.2: Übersetzung Hello World Beispiel

Für die Generierung von Zufallszahlen zwischen 0 und 1 in Zeile 6 kommt folgender JavaScript Code zum Einsatz: `exports.random = Math.random;`. Um aus dieser Zufallszahl (v) eine Zeichenkette zu machen, wird die Funktion *showNumber* der Typklasse *Show* genutzt. Diese Funktion ist ebenfalls in JavaScript implementiert und ruft, ein paar Details außer Acht gelassen, im Endeffekt JavaScript's *toString()*-Methode auf. Im nachfolgendem Listing wird der ADT der Console und die Implementierung der Funktion *log* aufgezeigt. Der „Konsolen-Effekt“ repräsentiert Berechnungen, die die Konsole nutzen und *Effect* ist ein *Kind*, kein Typ. Die Log-Funktion zum Schreiben einer Zeichenkette auf die

Konsole ist auf offensichtliche Weise in JavaScript implementiert.

```
foreign import data CONSOLE ::
   Effect

foreign import log :: forall
   eff. String -> Eff
  (console :: CONSOLE | eff)
   Unit
```

Listing 8.3: Foreign Function PS

```
exports.log = function (s) {
  return function () {
    console.log(s);
    return {};
  };
};
```

Listing 8.4: Foreign Function JS

9 | Reflexion

Anhand dieser Ausarbeitung zeigt sich deutlich, dass funktionale Konzepte wie Typklassen, Funktionen höherer Ordnung oder auch Funktoren auf Konzepte der Objektorientierten-Welt in einer intuitiven Art und Weise abgebildet werden können. Die Typsicherheit ist im Zielcode in JavaScript natürlich nicht mehr gegeben. Viele der funktionalen Sprachbestandteile gehen Hand in Hand, was in ihrem mathematischen Fundament begründet sein könnte. Beispielsweise stellten sich Typklassen und Algebraische Datentypen durch die Analyse als ein fundamentales Konstrukt in PureScript heraus - das Schmierwerk im Getriebe in funktionalen Sprachen.

Es zeigte sich allerdings auch, dass um Sprachen wie PureScript zu verstehen, die Hürde, welche anfangs zu nehmen ist, um einiges höher liegt als sie es bei einem OOP-Pendant wie Java oder JavaScript ist. Es müssen bspw. mindestens die in dieser Ausarbeitung vorgestellten Konzepte verstanden sein, um das Hallo Welt Beispiel des letzten Kapitels annähernd erfassen zu können. Dies liegt zugegebenermaßen an der Philosophie wie man mit Seiteneffekten umgehen möchte, um durch eine selbst auferlegte Beschränkung letztendlich einen Mehrwert generieren zu können. Deshalb erscheint das typische Hallo Welt Beispiel mit IO-Interaktionen als denkbar ungeeignet, um als Einsteigerbeispiel in einer funktionalen Sprache benutzt zu werden. Man kommt von IO aufgrund der Seiteneffekte unweigerlich zu Monaden, wofür man mindestens Funktoren verstanden haben sollte und darüber zu Algebraischen Datentypen und Typklassen.

Durch das Aufzeigen, was sich hinter zunächst kompliziert erscheinenden Sprachbestandteilen aus PureScript verbirgt, konnte die Sprache in meinen Augen wesentlich besser erschlossen werden, als wenn man sie nur konzeptionell betrachtet hätte. Der objektorientierte Zielcode ist verständlich und besteht in seinem Kern auch aus recht einfachen Konstrukten. Etwas visuell undurchsichtig wird es meist nur aufgrund des Curryings von Funktionen und der damit verbundenen Verschachtlung.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] Node.js API. *Modules*. URL: https://nodejs.org/api/modules.html (besucht am 13. 06. 2018).

[2] Aditya Bhargava. *Functors, Applicatives, And Monads In Pictures*. URL: http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html (besucht am 25. 06. 2018).

[3] PureScript Documentation. *Differences from Haskell*. URL: https://github.com/purescript/documentation/blob/master/language/Differences-from-Haskell.md (besucht am 13. 06. 2018).

[4] PureScript Documentation. *Modules*. URL: https://github.com/purescript/documentation/blob/master/language/Modules.md (besucht am 12. 06. 2018).

[5] PureScript Documentation. *Syntax*. URL: https://github.com/purescript/documentation/blob/master/language/Syntax.md (besucht am 16. 06. 2018).

[6] PureScript Documentation. *Types*. URL: https://github.com/purescript/documentation/blob/master/language/Types.md (besucht am 16. 06. 2018).

[7] Phil Freeman. *PureScript by Example. Functional Programming for the Web*. 24. Sep. 2017. PDF version.

[8] Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. 2011. URL: http://learnyouahaskell.com/chapters (besucht am 16. 06. 2018). Free online version.

[9] w3schools. *JavaScript Data Types*. URL: https://www.w3schools.com/js/js_datatypes.asp (besucht am 22. 06. 2018).

[10] w3schools. *JavaScript Numbers*. URL: https://www.w3schools.com/js/js_numbers.asp (besucht am 23. 06. 2018).

[11] w3schools. *JavaScript Object Constructors*. URL: https://www.w3schools.com/js/js_object_constructors.asp (besucht am 20. 06. 2018).

[12] w3schools. *JavaScript return Statement*. URL: https://www.w3schools.com/jsref/jsref_return.asp (besucht am 23. 06. 2018).

[13] w3schools. *JavaScript undefined Property*. URL: https://www.w3schools.com/jsref/jsref_undefined.asp (besucht am 23. 06. 2018).

[14] w3schools. *JavaScript Use Strict*. URL: https://www.w3schools.com/js/js_strict.asp (besucht am 13. 06. 2018).

[15] Haskell Wiki. *Fold*. URL: https://wiki.haskell.org/Fold (besucht am 28. 06. 2018).