technische universität
dortmund

Bachelorarbeit

Entwurf, Implementierung und Test einer Haskell-Bibliothek zur Validierung generischer Datenstrukturen

Jan Bessai

Mai 2011

Gutachter:
Dr. Christoph Schubert
Prof. Dr. Ernst-Erich Doberkat

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Softwaretechnologie (LS-10)
http://ls10-www.cs.tu-dortmund.de

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während meines bisherigen Studiums unterstützt und begleitet haben.
Mein besonderer Dank gilt den Mitarbeitern des Lehrstuhls für Softwaretechnologie und vornehmlich meinen Betreuern, Dr. Christoph Schubert und Prof. Dr. Ernst-Erich Doberkat, ohne deren Unterstützung diese Arbeit nicht möglich gewesen wäre.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADT .... Algebraic Data Type

AST .... Abstract Syntax Tree

DTD .... Document Type Definition

GHC .... Glasgow Haskell Compiler

HPC .... Haskell Program Coverage

SYB .... Scrap Your Boilerplate

XML .... Extensible Markup Language

1. Zusammenfassung

Wo auch immer Daten erhoben und verarbeitet werden, können diese irregulär oder fehlerhaft sein. Besonders Computersysteme können hiervon betriebskritisch gestört werden. Dementsprechend sind Maßnahmen notwendig, derartige Fehler und Irregularitäten frühzeitig zu erkennen und gegebenenfalls zu beheben. Diese Arbeit beschäftigt sich mit dem Entwurf, der Implementierung und dem Test einer Bibliothek zur Validierung generischer Datenstrukturen. Hierzu werden zunächst ausgewählte vorhandene Ansätze untersucht, um Anforderungen an einen neuen Ansatz zu gewinnen. Er basiert auf dem Konzept von Zippern, dessen formale Grundlagen nach der Anforderungsanalyse ausführlich behandelt werden. Die eingeführten Formalismen dienen sodann dem mathematischen Entwurf einer Bibliotheksschnittstelle samt Basisfunktionalität. Dieser Entwurf wird in der funktionalen Programmiersprache Haskell implementiert und ausführlich getestet. Abschließend wird die Anwendung der entwickelten Bibliothek an einem praxisnah gewählten Beispiel demonstriert.

2. Problemstellung und vorhandene Lösungsansätze

Es existiert bereits eine Fülle von Ansätzen zur Validierung, von denen im Folgenden einige ausgewählte problematisiert werden sollen, um Anforderungen an einen neuen Ansatz zu formulieren.

2.1. Typisierung

Viele Programmiersprachen erlauben es, eigene Datentypen zu definieren und somit Einschränkungen an die Eigenschaften von Daten zu machen. In der stark typisierten [Mar10] Sprache Haskell könnte die in Listing 2.1 gegebene Definition eines Datentyps für eine lineare Liste verwendet werden.

```
data Liste a = Element a | ElementUndNachfolger a (Liste a)
```

Listing 2.1: Eine nicht leere lineare Liste

Eine solche Liste speichert Elemente vom Typ a, die entweder alleine stehen oder ein Nachfolger-Element haben. Elemente ohne Daten lässt der Typ Liste nicht zu. Die Länge einer Liste kann mit einer einfachen Funktion bestimmt werden:

```
länge :: Liste a -> Integer
länge (Element _) = 1
länge (ElementUndNachfolger _ rest) = 1 + (länge rest)
```

Listing 2.2: Länge einer linearen Liste

Die Funktion länge bildet alle Eingaben vom Typ Liste auf positive **Integer** ab. Leere Listen sind unmöglich ohne den Typ zu verallgemeinern. Da dieser aber bereits a priori beim Compilieren feststehen muss, ist es nicht ohne weiteres möglich, auf A-Posteriori-Ereignisse zur Laufzeit (z.B. Einstellungen aus einer Konfigurationsdatei) einzugehen. Datentypen haben unter diesem Blickwinkel einen Charakter, der dem von Allaussagen gleichkommt, die, a priori getroffen, häufig a posteriori zu stark oder zu schwach sind.
Selbst wenn die Eigenschaften von Daten a priori bekannt sind, ist es manchmal inakzeptabel eine angemessene Kodierung in Datentypen vorzunehmen. So wäre es zum Beispiel für die mathematische Eigenschaft $n \in \mathbb{N}^{\geq 3}$ notwendig, die natürlichen Zahlen inklusive der auf ihnen definierten Operationen zu implementieren. Ein Extrembeispiel liefern Einschränkungen an Strings.
Listing 2.3 verdeutlicht den Kodierungsoverhead für einen Datentyp für die Menge aller Strings aus Großbuchstaben, die nicht „AB“ sind ($\{s \in \{A, B, \ldots Z\}^* | s \neq "AB"\}$).

2.2. Parser

Ein häufig gewählter Ansatzpunkt für Validierung ist das geschickte Parsen von Eingabedaten. So sind etwa moderne Compiler hierbei in der Lage, umfangreiche Analysen von

```
data Buchstabe = A | B | C | ... | X | Y | Z
data NichtA = B | C | D | ... | X | Y | Z
data NichtB = A | C | D | ... | X | Y | Z
data NichtAB =
  NichtStartA NichtA [Buchstabe]
  | NichtZweitesB Buchstabe NichtB [Buchstabe]
```

Listing 2.3: Strings außer ”AB”

Quellcode durchzuführen, um Programmierfehler zu entdecken. Ein Vorteil dieser Technik ist, dass sie genaue Anhaltspunkte über Fehlerstellen liefert. Überdies können inkonsistente Speicherzustände vermieden werden, da sie bereits während der Überführung in das vom Programm verwendete Format erkannt werden.
Der Quellcode von Parsern ist jedoch zumeist komplex, sodass das Hinzufügen von Validierungsfunktionalität und Fehlerbehandlungsmaßnahmen zu einem schwierigen Unterfangen werden kann. Die Vermischung von eigentlich unterschiedlichen Aspekten bringt erhebliche Nachteile für die Wartbarkeit, da Anpassungen an einen Aspekt immer auch den anderen beeinflussen. Oft wird für Parser auch auf Bibliotheken dritter (z.B. [Exp, Id3, Jso]) zurückgegriffen. Selbst wenn diese als Quellcode vorliegen, ist es zumeist indiskutabel aufwändig, sie programmspezifisch zu modifizieren. Noch problematischer wird dies, wenn, wie in Abbildung 2.1 gezeigt, mehrere Eingabeformate unterstützt werden: Hier müssen schlimmstenfalls für n Eingabeformate $\mathcal{O}(n)$ Parser angepasst werden.

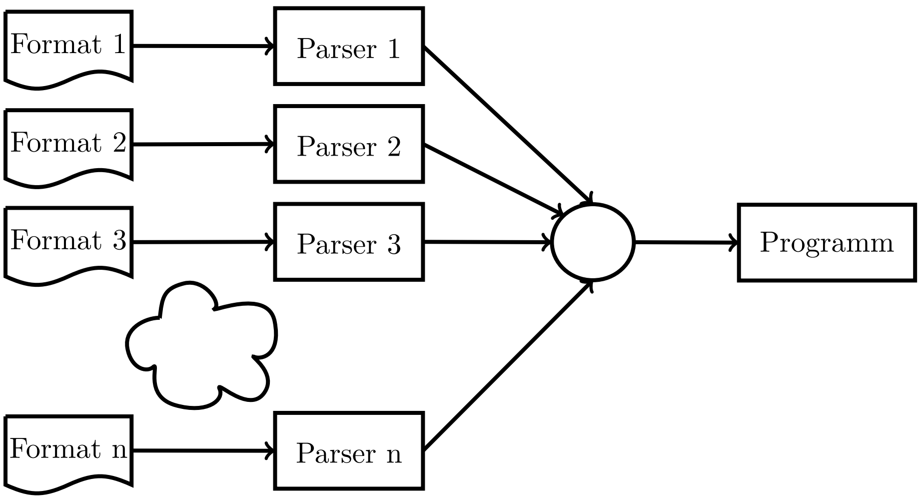

Abbildung 2.1.: Programm mit n unterschiedlichen Eingabeformaten

2.3. XML-Schema-Sprachen

Liegen Eingabedaten in einem XML-Format vor, so kann für die Validierung auf XML-Schema-Sprachen zurückgegriffen werden. Die gängigsten dieser Sprachen sind *XML DTD* [BPSM$^+$08], *XML Schema* [FW04] und *RELAX NG* [CM01]. Derartige Sprachen gestatten es, Validitätsbeschreibungen für XML-Dokumente zu definieren, welche anschließend, wie in [Har10] demonstriert, mit entsprechenden Werkzeugen geprüft werden können. Es wurde gezeigt, dass Schema-Sprachen in ihrer Ausdrucksstärke im Wesentlichen auf reguläre Baumgrammatiken beschränkt sind [MLMK05]. Diese Einschränkung erlaubt es, übersichtlich und effizient syntaktische Überprüfungen zu bewältigen. Jedoch ist semantische Validierung nur schwierig oder gar nicht modellierbar. Dies soll an einem Beispiel verdeutlicht werden. Listing 2.4 zeigt ein XML-Schema für eine Vortragsreihe, bei der einzelne Vorträge ein Thema, einen Start- und einen Endzeitpunkt haben.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="talks">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="talk">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="topic" type="xs:string"/>
              <xs:element name="start" type="xs:time"/>
              <xs:element name="end" type="xs:time"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 2.4: Schema für eine Vortragsreihe

Listing 2.5 zeigt ein Beispiel für eine XML-Datei mit zwei Vorträgen, die valide im Sinn des Schemas aus 2.4 ist.

```
<?xml version="1.0"?>
<talks>
  <talk>
    <topic>XML Schema Validation</topic>
    <start>10:00:00</start>
    <end>12:00:00</end>
  </talk>
  <talk>
    <topic>Haskell Validation</topic>
    <start>11:00:00</start>
    <end>13:00:00</end>
  </talk>
</talks>
```

Listing 2.5: Beispiel für eine XML-Datei mit Vorträgen

Die Vorträge aus Listing 2.5 überlappen sich zeitlich. Diese Überlappung ist für den syntaktischen Aufbau der XML-Datei und ihre Schemakonformität unerheblich. Semantisch kann sie jedoch unerwünscht sein, und es gibt keine sinnvolle Möglichkeit, sie mittels Schemavalidierung zu entdecken. Die theoretisch zugrundeliegende Ursache hierfür, welche an dieser Stelle nicht detailliert behandelt werden soll, ist, dass eine zu einem solchen Schema korrespondierende Baumgrammatik für jeden Vortrag Produktionsregeln zu allen noch erlaubten Vortragskombinationen definieren müsste, was nicht zu bewältigen ist. Zur Validierunslaufzeit dynamische Aspekte, wie etwa das Nachschlagen erlaubter Werte in einer Datenbank, sind ebenfalls nicht modellierbar.

2.4. Handgeschriebene Validierung

Selbstverständlich ist es immer möglich, anwendungsspezifischen Validierungscode per Hand, also ohne Verwendung zusätzlicher Werkzeuge als Hilfsmittel, zu schreiben. Während dieser Ansatz die meiste Flexibilität bietet, hat er auch einige Schwächen, die an einem Beispiel deutlich gemacht werden sollen.
Listing 2.6 zeigt eine Datenstruktur für den abstrakten Syntaxbaum (AST) einer einfachen imperativen Programmiersprache.

```haskell
data Pointer = Pointer Int deriving (Show)

data Value =
  ValueBool Bool
  | ValueStr String
  | ValueInt Int
  | ValuePtr Pointer
  deriving (Show)

data Identifier = Identifier String deriving (Show)

data Expression =
  Assignment Expression Expression
  | Variable Identifier Value
  | UninitializedVariable Identifier
  | Constant Value
  deriving (Show)

data Statement =
  Exp Expression
  | If Expression Statement
  | While Expression Statement
  | Block [Statement]
  deriving (Show)
```

Listing 2.6: Datenstruktur für den AST einer einfachen imperativen Programmiersprache

In der Sprache zu Listing 2.6 gibt es Strings, Integer, boolesche Werte und Pointer (Value). Pointer verweisen auf eine Speicheradresse, die als Integer abgespeichert wird. Ausdrücke (Expression) bestehen aus Konstanten mit einem Wert, uninitialisierten Variablen mit einem Identifier , der als String abgespeichert wird, Variablen mit einem Identifier und einem Wert, oder aus Zuweisungen zwischen zwei Ausdrücken. Eine Anweisung (Statement) ist ein einfacher Ausdruck, eine if-Abfrage mit einer Bedingung als Ausdruck und einer bedingten Anweisung, eine while-Schleife mit einem Kopf als Ausdruck und dem Schleifenrumpf als Anweisung, oder einem Block aus Anweisungen.
Es soll nun für eine konkrete Instanz des Syntaxbaumes, wie sie zum Beispiel ein Parser generieren könnte, geprüft werden, ob es einen Pointer mit dem Wert 0 als Speicheradresse gibt (im folgenden *Nullpointer* genannt). Hierzu wird zunächst in Listing 2.7 für prüfbare Daten ein Interface in Form einer Typklasse definiert.

```haskell
data Warning = NullPointerWarning deriving (Show)

class Warn a where
  warn :: a -> [Warning]
```

Listing 2.7: Typklasse für prüfbare Werte

Für alle prüfbaren Daten vom Typ a gibt es eine Funktion warn, die das jeweilige Datum auf eine Liste mit Warnungen (Warning) abbildet, welche auf *Nullpointer* hinweisen können (NullPointerWarning). Eine mögliche Instantiierung der Typklasse ist in Listing 2.8 angegeben.

```
instance Warn Pointer where
  warn (Pointer 0) = [NullPointerWarning]
  warn _ = []

instance Warn Value where
  warn (ValuePtr ptr) = warn ptr
  warn _ = []

instance Warn Expression where
  warn (Assignment l r) = warn l ++ warn r
  warn (Variable _ val) = warn val
  warn (UninitializedVariable _) = []
  warn (Constant val) = warn val

instance Warn Statement where
  warn (Exp e) = warn e
  warn (If e s) = warn e ++ warn s
  warn (While e s) = warn e ++ warn s
  warn (Block ss) = concatMap warn ss
```

Listing 2.8: Instanzen der Typklasse Warn

Für die Typen Statement, Expression und Value wird die Funktion warn rekursiv auf alle relevanten Teile angewandt, sodass ein konkreter Syntaxbaum von seiner Wurzel abwärts traversiert werden würde, und anschließend werden die Ergebnisse der einzelnen Aufrufe aufgesammelt. Die Traversierung stoppt, sobald ein Datum vom Typ Pointer erreicht wurde, welches anschließend überprüft wird und gegebenenfalls eine neue Warnung erzeugt.
Während für die eigentliche Überprüfung nur drei Zeilen Code geschrieben werden mussten, wurde für die Traversierung knapp mehr als das Vierfache an Quelltext benötigt. Für jede Veränderung am AST müsste der Traversierungscode sorgfältig überarbeitet werden, was bei komplexeren und tiefer verschachtelten Datenstrukturen langwierig und fehleranfällig werden kann. Für jedes zusätzliche Eingabeformat müssen neue Instanzen der Typklasse Warn angelegt werden. Die Flexibilität des Ansatzes wird also mit erheblichem Mehraufwand erkauft.

2.5. Objektorientierung

In objektorientierten Sprachen werden Daten durch den Zustand von Objekten modelliert. Der Zugriff auf den Zustand eines Objektes erfolgt in der Regel nicht durch direkte Manipulation von Daten, sondern unter Verwendung der vom Objekt zur Verfügung gestellten Methoden. Diese indirekte Schnittstelle zur Außenwelt ermöglicht es jedem Objekt stets einen validen Zustand zu gewährleisten, indem ungültige Modifikationen abgefangen werden. Wenn Objekte entsprechend modular entworfen werden, so begünstigt dies ein Design, bei dem jedes Objekt nur für die Validität seines eigenen lokalen Zustandes verantwortlich ist. Dies kann die Notwendigkeit für Traversierungen durch Zustandsräume mehrerer Objekte verringern. Allerdings macht der Überblick zu verschiedenen Techniken zur Validierung von Objekten in Java, der in [FGOG07] gegeben wird, deutlich, dass diese

Lokalität leicht zu einer ungewollten Durchmischung von Validierungscode mit der Implementierung der eigentlichen Objektfunktionalität führt. Die in [FGOG07] vorgestellten Ansätze lösen dieses Problem auf Kontrollflussebene, indem bevor und nachdem zustandsverändernde Methoden aufgerufen wurden Validierungsroutinen gestartet werden, die mit dem jeweils lokalen Objekt operieren, aber nicht zwangsweise zu diesem Objekt gehören. Der Ansatz dieser Arbeit ist es, auszunutzen, dass in der funktionalen Programmierung direkt mit der Struktur von Daten operiert werden kann, statt nur über indirekte Schnittstellen. Hierdurch wird die Traversierung verallgemeinerbar und soweit vereinfacht, dass die enge lokale Bindung von Validierungscode an die zu validierenden Daten gelockert wird.

2.6. Anforderungen an einen neuen Ansatz

Die vorangehende Betrachtung vorhandener Ansätze hat eine Reihe möglicher Anforderungen an einen neuen Ansatz offenbart:

Effiziente Kodierbarkeit:
Der resultierende Code soll einfach les- und überprüfbar sowie möglichst redundanzfrei sein.

Wartbarkeit und Erweiterbarkeit:
Es soll einfach sein, neue Datenformate zu unterstützen und vorhandene zu erweitern und zu verändern.

Ausdrucksstärke:
Es soll möglich sein, komplexe Überprüfungen ohne die Bindung an ein zu eng gewähltes theoretisches Konzept durchzuführen.

A posteriori wählbare Validierungskriterien:
Validierungskriterien sollen zur Laufzeit auswähl- und veränderbar sein.

Aufschluss über Fehlerstellen:
Die Position von Fehlerstellen in invaliden Daten soll auffindbar sein.

Automatische Traversierung:
Datenstrukturen sollen automatisch bis auf die Ebene validierbarer Bestandteile traversiert werden.

Trennbarkeit:
Die Validierungsfunktionalität soll klar als solche erkennbar in separaten Codeeinheiten realisierbar sein.

Hierbei sind die ersten drei Anforderungen nichtfunktionaler und die letzten vier funktionaler Natur. Darüber hinaus spielen die üblichen nichtfunktionalen Anforderungen an Software [Hof08], wie etwa Effizienz und Portierbarkeit, eine Rolle.

3. Typtheoretische Grundlagen und Zipper

In diesem Kapitel sollen zunächst einige theoretische Grundlagen für die Arbeit mit Funktoren und algebraischen Datentypen (im Folgenden ADTs genannt) erörtert werden. Anschließend wird das Konzept von Zippern zur Positionsmarkierung und -manipulation über die Einführung des Ableitungsoperators für ADTs erklärt. Gleichzeitig wird auf hierfür relevante Teile des *Scrap Your Boilerplate*-Ansatzes (SYB) eingegangen.

3.1. Definitionen und Schreibweisen

Als erstes soll eine Reihe von Definitionen und Schreibweisen aus [MFP91] übernommen und erweitert werden.

Identität:

$$\forall A\colon\ \mathrm{id} \in A \to A\colon \quad \mathrm{id}\, x = x \tag{3.1}$$

Bifunktor: Ein Bifunktor ist eine binäre Operation:

$$\forall (A, B, C, D, \varphi \in A \to B,\ \psi \in C \to D)\colon \quad \varphi \dagger \psi \in A \dagger C \to B \dagger D$$

Bifunktoren erhalten Identität und Verknüpfung:

$$\mathrm{id} \dagger \mathrm{id} = \mathrm{id} \tag{3.2}$$
$$f \dagger g \circ h \dagger j = (f \circ h) \dagger (g \circ j) \tag{3.3}$$

Monofunktor: Ein Monofunktor ist das unäre Gegenstück zum Bifunktor:

$$\forall (A, B, \varphi \in A \to B)\colon \quad \mathrm{F}\,\varphi \in \mathrm{F}\,A \to \mathrm{F}\,B \tag{3.4}$$

Auch hier bleiben Identität und Verknüpfung erhalten:

$$\mathrm{F}\,\mathrm{id} = \mathrm{id} \tag{3.5}$$
$$(\mathrm{F}\,f) \circ (\mathrm{F}\,g) = \mathrm{F}(f \circ g) \tag{3.6}$$

Die Verknüpfung von Monofunktoren ist definiert als:

$$\forall A\colon\ \mathrm{G}\,\mathrm{F}\,A = \mathrm{G}(\mathrm{F}\,A) \tag{3.7}$$

Die Verknüpfung von Monofunktoren mit einem Bifunktor erzeugt einen Monofunktor:

$$\forall A\colon (\mathrm{F} \dagger \mathrm{G})A = (\mathrm{F}\,A) \dagger (\mathrm{G}\,A) \tag{3.8}$$

Ein Funktor F ist applikativ, wenn es eine Funktion pure_F gibt, sodass:

$$\forall A\colon pure_F \in A \to \mathrm{F}\,A \tag{3.9}$$

Identitätsfunktor: Verwendet man die Identität id als Funktor I, so ergibt sich:

$$\forall A : \mathrm{I}\, A = A \tag{3.10}$$

Der Identitätsfunktor ist applikativ mit $\mathrm{pure}_\mathrm{I} = \mathrm{id}$.

Konstant-Funktor: Zu jedem Typen A gibt es den konstanten Monofunktor $\underline{\mathrm{A}}$ mit:

$$\forall (B, C, f) : f \in B \to C \Rightarrow \underline{\mathrm{A}}\, f = \mathrm{id} \tag{3.11}$$

$$\forall B \colon \nexists (C, D) \colon B \in C \to D \Rightarrow \underline{\mathrm{A}}\, B = A \tag{3.12}$$

Partielle Funktoranwendung: Durch partielle Anwendung auf Typen A und Funktionen f wird aus einem Bifunktor ein Monofunktor:

$$(A\dagger) = \underline{\mathrm{A}} \dagger \mathrm{I} \Rightarrow (A\dagger)B = A \dagger B \wedge (A\dagger)f = \mathrm{id} \dagger f \tag{3.13}$$

$$(\dagger A) = \mathrm{I} \dagger \underline{\mathrm{A}} \Rightarrow (\dagger A)B = B \dagger A \wedge (\dagger A)f = f \dagger id \tag{3.14}$$

$$(f\dagger) = f \dagger \mathrm{id} \Rightarrow (f\dagger) \circ (g\dagger) = (f \circ id \circ g \circ id) = (f \circ g) \tag{3.15}$$

Bottom-Typ: Der Bottom-Typ ist ein Typ, der keine Elemente enthält.

$$\forall f \colon \bot \circ f = \bot \tag{3.16}$$

$$\forall f \colon f \text{ ist strikt} \Rightarrow f \circ \bot = \bot \tag{3.17}$$

Void: Der Void-Typ **1** enthält nur das leere Element ().

$$\mathbf{1} = \{()\} \tag{3.18}$$

Die VOID-Funktion ist definiert als:

$$\forall x.\, \mathrm{VOID}\, x = () \tag{3.19}$$

Man kann konstante Funktionen ohne Argument mit dem Void-Typ modellieren:

$$f \colon \mathbf{1} \to A \tag{3.20}$$

Produkt: Das Produkt zweier Typen A und B ist definiert als:

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\} \tag{3.21}$$

Das mehrfache Produkt von $n \in \mathbb{N}$ Typen ergibt sich zu:

$$\prod_{i=1}^{0} A_i = \mathbf{1} \tag{3.22}$$

$$\prod_{i=1}^{1} A_i = A_1 \tag{3.23}$$

$$\prod_{i=1}^{n} A_i = A_1 \times A_2 \times \ldots A_n = \{(a_1, (a_2, (a_3 \ldots a_n) \ldots)) \mid a_i \in A_i,\, i \in \mathbb{N}^{\leq n}\} \tag{3.24}$$

Die Anwendung des Produktes der Funktionen $f_1 \dots f_n$ ist definiert als:

$$(f_1 \times f_2)(x, y) = (f_1 x, f_2 y) \tag{3.25}$$

$$\forall x\colon (\prod_{i=1}^{0} f_i)x = () \tag{3.26}$$

$$(\prod_{i=1}^{1} f_i)x_1 = f_1 x_1 \tag{3.27}$$

$$(\prod_{i=1}^{n} f_i)(x_1, (x_2, \dots x_n) \dots) = (f_1 x_1, (f_2 x_2, \dots f_n x_n) \dots) \tag{3.28}$$

Projektionsoperator: Die Projektionsoperatoren $\grave{\pi}$ und $\acute{\pi}$ wählen das erste und das zweite Element aus einem Tupel aus:

$$\grave{\pi}\,(x, y) = x \tag{3.29}$$

$$\acute{\pi}\,(x, y) = y \tag{3.30}$$

Applikationskombinator: Der Applikationskombinator $\vartriangle$ erzeugt ein Tupel aus Funktionsanwendungen:

$$(f \vartriangle g)x = (f\,x, g\,x) \tag{3.31}$$

$$f \circ g \vartriangle h \circ i = (f \circ g) \vartriangle (h \circ i) \tag{3.32}$$

Summe: Die Summe zweier Typen A und B ist definiert als:

$$A \mid B = \{\{0\} \times A\} \cup \{\{1\} \times B\} \cup \{\bot\} \tag{3.33}$$

Elemente einer Summe sind nummeriert. Die mehrfache Summe von $n \in \mathbb{N}$ Typen ergibt sich zu:

$$\sum_{i=1}^{0} A_i = \bot \tag{3.34}$$

$$\sum_{i=1}^{1} A_i = A_1 \tag{3.35}$$

$$\begin{aligned}\sum_{i=1}^{n} A_i &= A_1 \mid A_2 \mid \dots A_n = A_1 \mid \sum_{i=2}^{n} A_i \\ &= \{\bot\} \cup \{\{0\} \times A_1\} \cup \{\{1\} \times \{\{\bot\} \cup \{\{0\} \times A_2\} \cup \dots \{\{1\} \times A_n\}\} \dots\}\end{aligned} \tag{3.36}$$

Die Anwendung der Summe der Funktionen $f_1 \ldots f_n$ ist definiert als:

$$(f_1 \mid f_2)\bot = \bot \tag{3.37}$$
$$(f_1 \mid f_2)(0, x_1) = (0, f_1\, x_1) \tag{3.38}$$
$$(f_1 \mid f_2)(1, x_2) = (1, f_2\, x_2) \tag{3.39}$$
$$\forall x\colon \left(\sum_{i=1}^{0} f_i\right) x = \bot \tag{3.40}$$
$$\sum_{i=1}^{1} f_i = f_1 \tag{3.41}$$
$$\sum_{i=1}^{n} f_i = \left(f_1 \mid \sum_{i=2}^{n} f_i\right) \tag{3.42}$$

Selektionskombinator: Der Selektionskombinator wendet selektiv eine Funktion auf das Element einer Summe an und verwirft die zugehörige Nummerierung:

$$(f \triangledown g)\bot = \bot \tag{3.43}$$
$$(f \triangledown g)(0, x) = f\, x \tag{3.44}$$
$$(f \triangledown g)(1, y) = g\, y \tag{3.45}$$
$$f \circ g \triangledown h \circ i = (f \circ g) \triangledown (h \circ i) \tag{3.46}$$

Nummerierungskombinator: Die Nummerierungskombinatoren $\grave{\imath}$ und $\acute{\imath}$ nummerieren ein Element zum Einfügen in eine Summe:

$$\grave{\imath}\ x = (0, x) \tag{3.47}$$
$$\acute{\imath}\ x = (1, x) \tag{3.48}$$

Lambda-Ausdruck: Lambda-Ausdrücke sind unbenannte Funktionen:

$$\begin{aligned} x \in A, f \in A \to B\colon \lambda\, x\, .\, f\, x \in A \to B \\ y \in A\colon (\lambda\, x\, .\, f\, x)\, y = f\, y \end{aligned} \tag{3.49}$$

Lambda-Ausdrücke können auch über mehrere Variablen definiert werden:

$$\begin{aligned} x \in A, y \in B, f \in A \to B \to C\colon \lambda\, x\, y\, .\, f\, x\, y \in A \to B \to C \\ \lambda\, x\, y\, .\, f\, x\, y = \lambda\, x .\, \lambda\, y\, .\, f\, x\, y \end{aligned} \tag{3.50}$$

Boolescher Auswahloperator: Der boolesche Auswahloperator ist definiert als:

$$\begin{aligned} \forall A\colon ? \in (A \to \mathit{bool}) \to A \to A \mid A \\ p?\, a = \begin{cases} \bot, & \text{falls } p\, a = \bot \\ \grave{\imath}\, a, & \text{falls } p\, a = \mathit{true} \\ \acute{\imath}\, a, & \text{falls } p\, a = \mathit{false} \end{cases} \end{aligned} \tag{3.51}$$

bool kodiert hierbei Wahrheitswerte:

$$\begin{aligned} \forall A\colon (\mathit{true}, \mathit{false}) \in (A \to A \to A)^2 \\ \mathit{true}\, x\, y = x \\ \mathit{false}\, x\, y = y \end{aligned} \tag{3.52}$$

$$\mathit{bool} = \{\mathit{true}, \mathit{false}\} \tag{3.53}$$

Fixpunktoperator: Der Fixpunktoperator wendet eine Funktion rekursiv auf sich selbst an, bis sich ihr Ergebnis nicht mehr verändert:

$$\begin{aligned} \forall A\colon \mu &\in (A \to A) \to A \\ \mu f &= x, \quad x = f\,x \end{aligned} \tag{3.54}$$

Verknüpfungsgesetze: Für Summen und Produkte lassen sich unter Zuhilfenahme von Isomorphismen die üblichen Verknüpfungsgesetze (Kommutativität, Assoziativität und Distributivität) angeben:

$$\begin{aligned}
A \times B \cong B \times A, &\qquad f = f^{-1} = \acute{\pi} \vartriangle \grave{\pi} \\
A \mid B \cong B \mid A, &\qquad f = f^{-1} = \acute{i} \triangledown \grave{i} \\
A \times (B \times C) \cong (A \times B) \times C, &\qquad f = (\grave{\pi} \vartriangle (\grave{\pi} \circ \acute{\pi})) \vartriangle (\acute{\pi} \circ \acute{\pi}) \\
&\qquad f^{-1} = (\grave{\pi} \circ \grave{\pi}) \vartriangle ((\acute{\pi} \circ \grave{\pi}) \vartriangle \grave{\pi}) \\
A \mid (B \mid C) \cong (A \mid B) \mid C, &\qquad f = (\grave{i} \circ \grave{i}) \triangledown ((\acute{i} \circ \grave{i}) \triangledown \acute{i}) \\
&\qquad f^{-1} = (\grave{i} \triangledown (\grave{i} \circ \acute{i})) \triangledown (\acute{i} \circ \acute{i}) \\
(A \mid B) \times C \cong (A \times C) \mid (B \times C), &\qquad f = (\grave{\pi} \circ \grave{\pi}) \vartriangle ((\acute{\pi} \circ \grave{\pi}) \vartriangle \acute{\pi}) \\
&\qquad f^{-1} = (\grave{\pi} \vartriangle (\grave{\pi} \circ \acute{\pi})) \vartriangle (\acute{\pi} \circ \acute{\pi}) \\
C \times (A \mid B) \cong (A \mid B) \times C \cong (A \times C) \mid (B \times C) &\cong (C \times A) \mid (C \times B)
\end{aligned}$$

3.2. Algebraische Datentypen

Verwendet man die vorangegangene Notation, so kann man algebraische Datentypen gemäß [MFP91] allgemein als Fixpunkt eines Funktors ($\mu\,\mathrm{F}$) definieren.

Diese Definition lässt sich am besten anhand des Beispieles einer linearen Liste, welches schon in Listing 2.1 vorgestellt wurde, nachvollziehen. Analog zum Haskell-Datentyp aus Listing 2.1 definiert man einen Funktor LinList:

$$\textsc{LinList}\,X = \overbrace{A}^{\text{Element}} \mid \overbrace{A \times X}^{\text{ElementUndNachfolger}} \tag{3.55}$$

Die Fixpunktrekursion endet für diesen Funktor nicht, sodass ein Datentyp entsteht, der unendlich lange Listen zulässt:

$$\mu\,\textsc{LinList} = A \mid A \times (A \mid A \times (A \mid A \times (\dots))) \tag{3.56}$$

Listing 3.1 zeigt den Standrad-Haskell-Datentyp **Maybe**.

```
Maybe a = Nothing | Just a
```

Listing 3.1: Maybe

Maybe lässt sich ebenfalls einfach als Funktor darstellen:

$$\text{Maybe}\,X = \lfloor X \rfloor_{\mathbf{1}|} = 1 \mid X \tag{3.57}$$

Schränkt man sich auf Funktoren, die aus Summen und Produkten über Datentypen $A_1 \dots A_n$ bestehen, ein, erhält man nach Fixpunktbildung eine wichtige Klasse von Datentypen, die im Folgenden polynomiell genannt werden. Ein solcher Funktor kann mit

Hilfe der oben erläuterten Verknüpfungsgesetze dargestellt werden als:

$$\mathrm{F}\,X = \sum_{i=1}^{m} \prod_{j=1}^{k_i} B_{i,j}\,, \quad B_{i,j} \in \{X, \bot\} \cup \bigcup_{l=1}^{n} \{A_l\} \tag{3.58}$$

Die Menge aller polynomiellen Funktoren über den Variablen und Konstanten $B_{i,j}$ für $n \in \mathbb{N}$, $m \in \mathbb{N} \to \mathbb{N}$, $i \in \mathbb{N}^{\leq n}$, $j \in \mathbb{N}^{\leq m_i}$ sei im Folgenden mit dem Symbol $\mathbb{P}(B, n, m)$ oder auch kurz $\mathbb{P}(B)$ bezeichnet. Mit derart eingeschränkt definierten Datentypen sind kontextfreie Grammatiken beschreibbar. Die genauen Zusammenhänge wurden zum Beispiel in [BD07] untersucht. Hier soll der Sachverhalt an einer Beobachtung plausibel gemacht werden. Zunächst definiert man wie in [EP08] eine beliebige kontextfreie Grammatik $G = (V, T, R, S)$, bei der gilt:

$$V \cap T = \emptyset \quad \wedge \quad S \in V \quad \wedge \quad \forall P \to Q \in R : P \in V \quad \wedge \quad Q \in (V \cup T)^\star$$

Die rechten Seiten der Regeln dieser Grammatik kann man so umformen, dass in ihnen nur noch Variablen auftauchen, oder sie nur noch aus je einem Terminal bestehen. Hierzu führt man für jedes Terminal $a_n \in T$ eine neue Variable A_n in V ein und erhält so zunächst V'. Ersetzt man nun alle Terminale a_n in den rechten Seiten der Produktionen durch ihre entsprechenden Variablen A_n und erzeugt für jedes A_n eine neue Produktionsregel $A_n \to a_n$, erhält man R'. In der so umgeformten Grammatik $G' = (V', T, R', S)$ haben die Startregeln die Form $S \to Q \in V^\star$, und die Menge der Startregeln sei $R_S \subseteq R$. Ferner kann jede Variable A auf einer rechten Seite problemlos als Startsymbol einer neuen kontextfreien Grammatik $G_A = (V', T, R', A)$ interpretiert werden. Nun erzeugt man einen Funktor zu einem polynomiellen Datentyp:

$$\mathrm{F}\,S = \sum_{S \to Q \in R_S} \prod_{A \in Q} B_A, \quad B_A = \{\omega \in S(G_A)\}$$

Hierbei steht $S(G_A)$ für die von der Grammatik G_A erzeugte Sprache. Ein Wort ist genau dann in $S(G') = S(G)$ enthalten, wenn es aus einer von einer Startregel aus R_S vorgegebenen Anordnung der aus den Variablen auf der rechten Seite dieser Regel herleitbaren Wörter besteht. Ein Wort ist genau dann in $\mu\,\mathrm{F}$ enthalten, wenn es in einem Summanden von F enthalten ist, und diese enthalten genau die von Startregeln erlaubten Anordnungen von Wörtern als Produkte. Der einzige Unterschied sind die in S(G) fehlenden, aber aus $\mu\,\mathrm{F}$ problemlos entfernbaren Summennummerierungen. Die Konstruktion lässt sich rekursiv für die in F vorkommenden B_A-Typen fortsetzen, bis die neu eingeführten Variablen A_n berücksichtigt werden müssen. Dies kann z.B. geschehen, indem man neue Typen einführt, welche die Terminalsymbole unär als eindeutig interpretierbare Void-Folgen kodieren:

$$B_{A_n}\,\mathrm{F} = \prod_{i=1}^{n} \mathbf{1}$$

Die umgekehrte Konstruktion einer kontextfreien Grammatik aus einem polynomiellen Datentyp erfolgt analog, indem zu jedem Summanden des Funktors des Datentyps entsprechend der vorkommenden Produkte Startregeln gebildet werden und für die Untertypen entweder genauso verfahren wird, oder im Fall eines Typen aus einer Void-Folge ein Terminal erzeugt wird.

Die Gleichmächtigkeit mit kontextfreien Grammatiken macht polynomielle Datentypen äußerst ausdrucksstark. Sie korrespondieren außerdem mit den Datentypen, die Haskell normalerweise zur Verfügung stellt [Mar10].

3.3. Zipper

Zipper sind ein Entwurfsmuster, das dazu dient, komplexe Datenstrukturen zu traversieren und dabei jederzeit jede Position der Datenstruktur erreichen zu können. Sie wurden zum ersten Mal 1997 von Gérad Huet in [Hue97] beschrieben. Seitdem haben sich vor allem Conor McBride und seine Kollegen Michael Abbott, Thorsten Altenkirch und Neil Ghani mit den theoretischen Hintergründen zu Zippern beschäftigt [McB01, AAMG03, AAMG04, McB08]. 2010 wurde von Michael D. Adams in [Ada10] ein Weg gefunden, sie automatisch anwendbar zu machen.

Um Zipper zu verstehen, macht es Sinn, zunächst einen formalen Ableitungsoperator $\dfrac{\partial}{\partial X}$ auf Funktoren zu polynomiellen Datentypen zu definieren, wie dies z.B. in [McB01, AAMG04] geschieht:

$$\frac{\partial \bot}{\partial X} = \bot \tag{3.59}$$

$$\frac{\partial \mathbf{1}}{\partial X} = \bot \tag{3.60}$$

$$\frac{\partial X}{\partial X} = \mathbf{1} \tag{3.61}$$

$$\frac{\partial A \mid B}{\partial X} = \frac{\partial A}{\partial X} \mid \frac{\partial B}{\partial X} \qquad \text{(Summationsregel)} \tag{3.62}$$

$$\frac{\partial A \times B}{\partial X} = \frac{\partial A}{\partial X} \times B \mid A \times \frac{\partial B}{\partial X} \qquad \text{(Produktregel)} \tag{3.63}$$

$$\frac{\partial\, \mathrm{G} \circ \mathrm{F}}{\partial X} = \frac{\partial\, \mathrm{G}(\mathrm{F})}{\partial\, \mathrm{F}\, X} \times \frac{\partial\, \mathrm{F}}{\partial X} \qquad \text{(Kettenregel)} \tag{3.64}$$

Hieraus ergibt sich für beliebige Polynome unter Anwendung der Summations- und Produktregel (3.62, 3.63):

$$\begin{aligned}
\frac{\partial \sum_{i=1}^{n} \prod_{j=1}^{m_i} A_{i,j}}{\partial X} &= \sum_{i=1}^{n} \frac{\partial \prod_{j=1}^{m_i} A_{i,j}}{\partial X} \\
&= \sum_{i=1}^{n} \left(\frac{\partial A_{i,1}}{\partial X} \times \prod_{j=2}^{m_i} A_{i,j} \,\middle|\, A_{i,1} \times \frac{\partial A_{i,2}}{\partial X} \times \prod_{j=3}^{m_i} A_{i,j} \,\middle|\, \dots \,\middle|\, A_{i,1} \times A_{i,2} \times \dots A_{i,m_i-1} \times \frac{\partial A_{i,m_i}}{\partial X} \right) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{m_i} \underbrace{\left(A_{i,0} \times A_{i,1} \times \dots \frac{\partial \overbrace{A_{i,j}}^{\text{Loch}}}{\partial X} \times A_{i,j+1} \times \dots A_{i,m_i} \right)}_{\text{Kontext}}
\end{aligned} \tag{3.65}$$

Man nennt Typen, die noch abzuleiten sind, Loch. Die einzelnen aufsummierten Produktterme der Ableitung nennt man Kontext des zugehörigen Loches. Hat man einen Kontext und sein zugehöriges Loch, so braucht man dieses nur noch an der Stelle von $\frac{\partial A_{i,j}}{\partial X}$ einsetzen und kann so den zum Kontext gehörigen Summanden des abgeleiteten Typs rekonstruieren:

$$\sum_{i=1}^{n} \prod_{j=1}^{m_i} A_{i,j} = (\sum_{i=1}^{k-1} \prod_{j=1}^{m_i} A_{i,j}) \mid (A_{k,0} \times A_{k,1} \times \cdots \overset{A_{k,j}}{\xcancel{\frac{\partial A_{k,j}}{\partial X}}} \times A_{k,j+1} \times \dots A_{k,m_i}) \mid (\sum_{k+1=1}^{n} \prod_{j=1}^{m_i} A_{i,j})$$

Dieses Rückrechnen ist eine zentrale Operation, um bei der Traversierung von Datenstrukturen bereits besuchte Teile erneut zu besuchen. Zu diesem Zweck sei zunächst die Menge aller möglicher eines im Datentyp R durch mehrmaliges Ableiten erreichbaren direkten Vorgänger eines Loches vom Typ H definiert als:

$$\begin{aligned}\mathrm{PREDS}(H,R) = \{P \mid \exists k \in \mathbb{N} \colon \exists (A_1, \ldots A_k) \colon & \\ & P = A_1 \wedge H \in \mathrm{PARTS}(A_1) \wedge \\ & A_1 \in \mathrm{PARTS}(\mathrm{A}_2) \wedge \ldots \wedge A_{k-1} \in \mathrm{PARTS}(A_k) \wedge \\ & R = A_k\}\end{aligned} \tag{3.66}$$

Hierbei ist $\mathrm{PARTS}(A)$ die Menge aller Bestandteile des polynomiellen Funktors A:

$$\mathrm{PARTS}(A) = \{B_{i,j} \mid \exists n \in \mathbb{N}, m \in \mathbb{N} \to \mathbb{N} \colon A \in \mathbb{P}(B, n, m)\} \tag{3.67}$$

Außerdem sei die Zerlegung einer Ableitung in linke und rechte Nachbarn der zugehörigen Löcher realisiert durch die Funktion LR:

$$P \in \mathbb{P}(A, n, m) \colon \quad \mathrm{LR}\left(\frac{\partial P}{\partial X}\right) = \sum_{i=1}^{n} \sum_{j=1}^{m_i} \left(\lfloor \prod_{k=1}^{j-1} A_{i,k} \rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m_i} A_{i,k} \rfloor_{\mathbf{1}|} \right) \tag{3.68}$$

Haskell ist statisch getypt [Mar10], sodass alle Typen zur Compilezeit bekannt sein müssen. Dementsprechend kann vom Compiler eine Abbildung von Typen auf eindeutige Identifikatoren (z.B. natürliche Zahlen) generiert werden. Dieser Mechanismus ermöglicht die Typidentifikation zur Laufzeit und gewährleistet die Wohldefiniertheit der im Folgenden verwendeten Funktionen. Im hier verwendeten Modell ist die Abbildungsvorschrift für die Menge der im Programm vorkommenden polynomiellen Typen $\mathbb{T}_n = \{A_k | \exists k \in \mathbb{N}^{\leq n} \colon A_k \in P(B_k, n_k, m_k)\}$ gegeben als:

$$\begin{aligned} &\forall A_k \in \mathbb{T}_n \colon \ \mathrm{tag} \in A_k \to \mathbb{N}^{\leq n} \\ &x \in A_k \Leftrightarrow \mathrm{tag}\, x = k \end{aligned} \tag{3.69}$$

Eine entsprechende Variante ist auch zur Compilezeit auf Typebene möglich:

$$\begin{aligned} &\mathrm{TAG} \in \mathbb{T}_n \to \mathbb{N}^{\leq n} \\ &A_k \in \mathbb{T}_n \colon \ \mathrm{TAG}\, A_k = k \end{aligned} \tag{3.70}$$

Ein Datentyp für die Liste der bei mehrfacher Ableitung des Typs R bis zum Loch vom Typ H verwendeten zerlegten Kontexte ist jetzt darstellbar als:

$$\text{Contexts } H\, R = \mathbf{1} \mid \bigcup_{P \in \mathrm{PREDS}(H,R)} \{\mathrm{TAG}(P)\} \times LR\left(\frac{\partial P}{\partial H}\right) \times \text{Contexts } P\, R \tag{3.71}$$

Zur Darstellung von Contexts $H\ R$ wird nicht auf die Summe über Typen zurückgegriffen, da sonst für die passende Nummerierung die Menge $\mathrm{PREDS}(H, R)$ vollständig berechnet werden müsste. Die Wohldefiniertheit von Funktionen auf der Typvereinigung wird durch die TAG-Funktion sichergestellt, die keinen zwei unterschiedlichen Vorgängern die selbe natürliche Zahl zuordnet, sodass Kontexte zu unterschiedlichen Vorgängern P immer disjunkt sind. Für die Funktionen $f \in H \to H'$ und $g \in R \to R'$ ergibt Contexts:

$$\begin{aligned}\text{Contexts}\ f\,g = \text{id} \mid \lambda & \left(\bigcup_{P \in \text{PREDS}(H,R)} \{\text{TAG}(P)\} \times \text{LR}\left(\frac{\partial P}{\partial H}\right) \times \text{Contexts}\ P\,R \right) . \\ & \bigcup_{P' \in \text{PREDS}(f\,H, g\,R)} \{\text{TAG}(P')\} \times \text{LR}\left(\frac{\partial P'}{\partial f\,H}\right) \times \text{Contexts}\ P'\,(g\,R)\end{aligned} \tag{3.72}$$

Hierdurch bleiben die Funktoreigenschaften erhalten:

$$\begin{aligned}\text{Contexts id id} = \text{id} \mid \lambda & \left(\bigcup_{P \in \text{PREDS}(H,R)} \{\text{TAG}(P)\} \times \text{LR}\left(\frac{\partial P}{\partial H}\right) \times \text{Contexts}\ P\,R \right) . \\ & \bigcup_{P' \in \text{PREDS}(\text{id}\ H, \text{id}\ R)} \{\text{TAG}(P')\} \times \text{LR}\left(\frac{\partial P'}{\partial\, \text{id}\ H}\right) \times \text{Contexts}\ P'\,(\text{id}\ R) \\ = \text{id} &\end{aligned}$$

$$\begin{aligned}&(\text{Contexts}\ f\,g) \circ (\text{Contexts}\ h\,j) \\ &= (\text{Contexts}\ f\,g) \circ (\text{id} \mid \lambda(\bigcup_{P \in \text{PREDS}(H,R)} \{\text{TAG}(P)\} \times \text{LR}\left(\frac{\partial P}{\partial H}\right) \times \text{Contexts}\ P\,R\,) . \\ &\qquad \bigcup_{P' \in \text{PREDS}(h\,H, j\,R)} \{\text{TAG}(P')\} \times \text{LR}\left(\frac{\partial P'}{\partial h\,H}\right) \times \text{Contexts}\ P'\,(j\,R)) \\ &= \text{id} \mid \lambda \left(\bigcup_{P \in \text{PREDS}(H,R)} \{\text{TAG}(P)\} \times \text{LR}\left(\frac{\partial P}{\partial H}\right) \times \text{Contexts}\ P\,R \right) . \\ &\qquad \bigcup_{P' \in \text{PREDS}(f \circ h\,H, g \circ j\,R)} \{\text{TAG}(P')\} \times \text{LR}\left(\frac{\partial P'}{\partial f \circ h\,H}\right) \times \text{Contexts}\ P'\,(g \circ j\,R) \\ &= \text{Contexts}\,(f \circ h)\,(g \circ j)\end{aligned}$$

Ein Zipper besteht aus einem Loch vom Typ H und seiner zugehörigen Liste von Kontexten im Datentyp R:

$$\text{Zipper}\ H\,R = H \times \text{Contexts}\ H\,R \tag{3.73}$$

Auch Zipper sind Funktoren. Mit $f \in H \to H'$, $g \in R \to R'$, $h \in H'' \to H$ und $i \in R'' \to R$ erhält man:

$$\begin{aligned}\text{Zipper}\ f\,g &= f \times \text{Contexts}\ f\,g \\ \text{Zipper id id} &= \text{id} \times \text{Contexts id id} = \text{id} \times \text{id} = \text{id} \\ (\text{Zipper}\ f\,g) \circ (\text{Zipper}\ h\,j) &= (f \circ h) \times \text{Contexts}\,(f \circ h)\,(g \circ j) = \text{Zipper}\,(f \circ h)\,(g \circ j)\end{aligned}$$

3.4. Operationen auf Zippern

Ein Zipper kann mit toZipper erzeugt werden:

$$\begin{aligned}&\text{toZipper} \in R \to \text{Zipper}\ R\,R \\ &\text{toZipper} = (\text{id} \vartriangle \lambda\,x\,.\,\mathbf{1})\end{aligned} \tag{3.74}$$

Nun ist es möglich, den Zipper mit einheitlichen Funktionen zu manipulieren und so durch die Datenstruktur zu navigieren:

$\mathsf{down}_{\mathsf{Left}}$ Der Bestandteil, der im aktuellen Loch eines Zippers am weitesten links steht, wird zum neuen Loch.

$$\begin{aligned} &R \in \mathbb{P}(B, j, k),\ H \in \mathbb{P}(A, n, m),\ H' \in \mathrm{PARTS}(H)\colon \\ &\mathrm{down}_{\mathrm{Left}} \in \mathrm{Zipper}\ H\ R \to \lfloor \mathrm{Zipper}\ H'\ R \rfloor_{\mathbf{1}|} \end{aligned} \tag{3.75}$$

up Die Datenstruktur, in die das aktuelle Loch eingebettet war, wird zum neuen Loch.

$$\begin{aligned} &R \in \mathbb{P}(B, j, k),\ H' \in \mathbb{P}(A, n, m),\ H \in \mathrm{PARTS}(H')\colon \\ &\mathrm{up} \in \mathrm{Zipper}\ H\ R \to \lfloor \mathrm{Zipper}\ H'\ R \rfloor_{\mathbf{1}|} \end{aligned} \tag{3.76}$$

right Das aktuelle Loch wird durch seinen rechten Nachbarn aus der Datenstruktur, in der es eingebettet war, ersetzt.

$$\begin{aligned} R \in \mathbb{P}(B, j, k),\ P \in \mathbb{P}(A, n, m),\ i \in \mathbb{N}^{\leq n},\ k \in \mathbb{N}^{\leq m_i}\colon& \\ \mathrm{right} \in \mathrm{Zipper}\ A_{i,k}\ R \to \lfloor \mathrm{Zipper}\ A_{i,k+1}\ R \rfloor_{\mathbf{1}|}& \end{aligned} \tag{3.77}$$

left Das aktuelle Loch wird durch seinen linken Nachbarn ersetzt.

$$\begin{aligned} R \in \mathbb{P}(B, j, k),\ P \in \mathbb{P}(A, n, m),\ i \in \mathbb{N}^{\leq n},\ k \in \mathbb{N}^{\leq m_i}\colon& \\ \mathrm{left} \in \mathrm{Zipper}\ A_{i,k}\ R \to \lfloor \mathrm{Zipper}\ A_{i,k-1}\ R \rfloor_{\mathbf{1}|}& \end{aligned} \tag{3.78}$$

Der Vollständigkeit halber sollen die Manipulationsfunktionen nun noch ausgeschrieben werden.

$$\begin{aligned} &R \in \mathbb{P}(B, j, k),\ H \in \mathbb{P}(A, n, m),\ H' \in \mathrm{PARTS}(H)\colon \\ &\mathrm{down}_{\mathrm{Left}} \in \mathrm{Zipper}\ H\ R \to \lfloor \mathrm{Zipper}\ H'\ R \rfloor_{\mathbf{1}|} \\ &\mathrm{down}_{\mathrm{Left}} = (\grave{i} \circ \grave{\pi} \triangledown \acute{i} \circ (\acute{\pi} \circ \grave{\pi}\ \vartriangle\ \acute{\pi}))\ \mathrm{dZero?} \circ (\mathrm{d} \circ \grave{\pi}\ \vartriangle\ \acute{i} \circ (\mathrm{tag} \vartriangle \mathrm{lr} \circ \grave{\pi}\ \vartriangle\ \acute{\pi})) \end{aligned}$$

$\mathrm{down}_{\mathrm{Left}}$ erzeugt mit lr die Zerlegung des Loches in einen neuen Kontext und stellt diese zusammen mit dem Typidentifikator des Loches der Kontextliste voran.

$$\begin{aligned} H \in P(A, n, m)\colon\ \mathrm{lr} \in H \to \bigcup_{i=1}^{n} \mathrm{LR}\left(\frac{\partial H}{\partial A_{i,1}}\right)& \\ \mathrm{lr} = \sum_{i=1}^{n} \Big(\grave{i} \circ (\grave{i} \circ \mathrm{VOID} \vartriangle\ (\acute{i} \circ \acute{\pi} \triangledown \grave{i} \circ \mathrm{VOID}))\ \mathrm{isTuple?}\Big)& \\ \forall A\colon\ \mathrm{isTuple} \in A \to \mathit{bool}& \\ \mathrm{isTuple}\, x = \begin{cases} \mathit{true} & \text{falls } \exists (B \times C)\colon A = (B \times C) \wedge x \in A \\ \mathit{false} & \text{sonst} \end{cases}& \end{aligned}$$

d traversiert den polynomiellen Funktor und extrahiert den ersten Produktterm mit selectFirst als neues Loch. Fehlschläge hierbei werden mit dZero überprüft.

$$
\begin{aligned}
\mathrm{d} &\in \sum_{i=1}^{n} \prod_{j=1}^{m_i} A_{i,j} \to \lfloor \bigcup_{i=1}^{n} A_{i,1} \rfloor_{\mathbf{1}|} \\
\mathrm{d} &= \mu(\lambda f\,.\,((\mathrm{selectFirst} \bigtriangledown f) \bigtriangledown \ \mathrm{selectFirst})\, \mathrm{isSum?})
\end{aligned}
$$

$$
\begin{aligned}
&\forall A\colon\ \mathrm{isSum} \in A \to \mathit{bool} \\
&\mathrm{isSum}\, x = \begin{cases} \mathit{true} & \text{falls } \exists (B \mid C)\colon A = (B \mid C) \wedge x \in A \\ \mathit{false} & \text{sonst} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{selectFirst} &\in \prod_{i=1}^{n} A_i \to \lfloor A_1 \rfloor_{\mathbf{1}|} \\
\mathrm{selectFirst} &= \left(\grave{\imath} \circ \mathrm{VOID} \bigtriangledown \acute{\imath} \circ ((\grave{\pi} \bigtriangledown \ \mathrm{id})\, \mathrm{isTuple?})\right) \mathrm{isVoid?}
\end{aligned}
$$

$$
\begin{aligned}
&\forall A\colon\ \mathrm{isVoid} \in A \to \mathit{bool} \\
&\mathrm{isVoid}\, x = \begin{cases} \mathit{true} & \text{falls } x = () \\ \mathit{false} & \text{sonst} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{dZero} &\in (\mathbf{1} \mid A) \times B \to \mathit{bool} \\
\mathrm{dZero} &= (\lambda x\,.\,\mathit{true} \bigtriangledown \lambda x\,.\,\mathit{false}) \circ \grave{\pi}
\end{aligned}
$$

$$R \in \mathbb{P}(B, j, k),\ H' \in \mathbb{P}(A, n, m),\ H \in \mathrm{PARTS}(H')\colon$$

$$\mathrm{up} = ((\lambda x\,.\ \mathrm{traverseAndCombine}\,(\grave{\pi}\, x)\, (\mathrm{selectFirstContext}\ x) \bigtriangleup \ \mathrm{tail} \circ \acute{\pi}\,) \bigtriangledown \acute{\pi}\,)\, \mathrm{contextAvailable?}$$

Zunächst wird mit contextAvailable geprüft, ob noch ein Vorgänger-Kontext vorhanden ist. Wenn ja, wird der Kopf aus der Kontextliste mit tail entfernt, und mit selectFirstContext ausgewählt, um anschließend mit traverseAndCombine wieder mit dem aktuellen Loch verschmolzen zu werden.

$$
\begin{aligned}
\mathrm{contextAvailable} &\in A \times (\mathbf{1} \mid B) \to \mathit{bool} \\
\mathrm{contextAvailable} &= (\lambda x\,.\,\mathit{false} \bigtriangledown \lambda x\,.\,\mathit{true}) \circ \acute{\pi} \\
\mathrm{tail} &\in (\mathbf{1} \mid (A \times B \times (1 \mid C))) \to (\mathbf{1} \mid C) \\
\mathrm{tail} &= (\grave{\imath} \circ \mathrm{id} \bigtriangledown \acute{\pi} \circ \acute{\pi}\,) \\
\mathrm{selectFirstContext} &\in (A \times (\mathbf{1} \mid (B \times C \times D))) \to C \\
\mathrm{selectFirstContext} &= \grave{\pi} \circ \acute{\pi} \circ \acute{\pi} \circ \acute{\pi}
\end{aligned}
$$

Die Funktion traverseAndCombine durchläuft die innere Summe des aktuellen Kontextes, um ihre Nummerierungen zu verwerfen bis der Inhalt gefunden wurde, der dann mit dem aktuellen Loch verschmolzen wird.

$$
\begin{aligned}
\mathrm{traverseAndCombine} \in H \to\ & \sum_{i=1}^{n} \sum_{j=1}^{m_i} \left(\lfloor \prod_{k=1}^{j-1} A_{i,k} \rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m_i} A_{i,k} \rfloor_{\mathbf{1}|} \right) \to \\
& \sum_{i=1}^{n} \bigcup_{j=1}^{m_i} (A_{i,1} \times A_{i,2} \ldots A_{i,j-1} \times H \times A_{i,j+1} \times A_{i,j+2} \ldots A_{i,m_i})
\end{aligned}
$$

$$\mathrm{traverseAndCombine}\, h = \sum_{i=1}^{n} (\mu(\lambda f\,.\,((\mathrm{combine}\, h \bigtriangledown f) \bigtriangledown \ \mathrm{combine}\, h)\, \mathrm{isSum?}))$$

Durch combine wird das Loch wieder in den Kontext eingesetzt. Hierzu werden lcombine für das linke Produkt des Kontextes und rcombine für das rechte verwendet. Die Unterscheidung ist nötig, da das rechte Produkt in die Klammern des linken eingeschachtelt werden muss. Die Struktur des Ergebnisses von combine hängt davon ab, ob das linke und/oder das rechte Produkt vorhanden war(en).

$$\begin{aligned}
\text{combine} \in H \to \left(\lfloor \prod_{k=1}^{n} A_k \rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=1}^{m} B_k \rfloor_{\mathbf{1}|} \right) \to\ & \{(A_1 \times A_2 \ldots A_n \times H \times B_1 \times B_2 \ldots B_m)\} \cup \\
& \{(H \times B_1 \times B_2 \ldots B_m)\} \cup \\
& \{(A_1 \times A_2 \ldots A_m \times H)\} \cup \\
& \{H\}
\end{aligned}$$

$$\text{combine}\; h\, x = \text{lcombine}\,(\text{rcombine}\; h\,(\acute{\pi}\, x))\,(\grave{\pi}\, x)$$

$$\text{lcombine} \in \prod_{k=1}^{m} B_k \to \lfloor \prod_{k=1}^{n} A_k \rfloor_{\mathbf{1}|} \to \{(A_1 \times A_2 \ldots A_n \times B_1 \times B_2 \ldots B_m)\} \cup \{(B_1 \times B_2 \ldots B_m)\}$$

$$\text{lcombine}\, r = (\lambda\, x \,.\, r) \triangledown \mu(\lambda\, f \,.\, (((\text{id} \times f) \triangledown (\text{id} \vartriangle \lambda\, x \,.\, r))\, \text{isTuple?}))$$

$$\text{rcombine} \in H \to \lfloor \prod_{k=1}^{m} B_k \rfloor_{\mathbf{1}|} \to \{(H \times B_1 \times B_2 \ldots B_m)\} \cup \{H\}$$

$$\text{rcombine}\, h = (\lambda\, x \,.\, h) \triangledown (\lambda\, x \,.\, h \vartriangle \text{id})$$

Wenn ein Kontext verfügbar ist, so versucht die Funktion right aus diesem mit extractRightHoleOuter das Loch rechts vom aktuellen zu extrahieren und mit shiftRight das aktuelle Loch eine Stelle nach rechts zu schieben. Die restliche Funktion setzt den Zipper wieder richtig zusammen.

$$\begin{aligned}
& R \in \mathbb{P}(B, j, k),\ P \in \mathbb{P}(A, n, m),\ i \in \mathbb{N}^{\leq n},\ k \in \mathbb{N}^{\leq m_i}: \\
& \text{right} \in \text{Zipper}\ A_{i,k}\, R \to \lfloor \text{Zipper}\ A_{i,k+1}\, R \rfloor_{\mathbf{1}|} \\
& \text{right} = (((\grave{i} \circ \grave{\pi} \triangledown \acute{i} \circ (\acute{\pi} \circ \grave{\pi} \vartriangle \acute{\pi}))\, \text{dZero?} \circ \\
& \qquad\qquad (\text{extractRightHoleOuter} \circ \text{selectFirstContext} \vartriangle \\
& \qquad\qquad\quad \acute{i} \circ (\text{selectTag} \vartriangle \lambda\, x \,.\, (\text{shiftRight}\,(\grave{\pi}\, x)\,(\text{selectFirstContext}\; x)) \vartriangle \text{tail} \circ \acute{\pi}))) \triangledown \\
& \qquad\quad \grave{i} \circ \text{VOID}) \\
& \qquad\quad \text{contextAvailable?}
\end{aligned}$$

Von extractRightHoleOuter wird zunächst die äußere Summe eines Kontextes durchlaufen und nach dem inneren Produkt gesucht. Wird stattdessen eine innere Summe gefunden, so wird diese mit extractRightHoleInner auf ihr inneres Produkt durchsucht. Summennummerierungen werden bei der Suche verworfen. Wurde ein inneres Produkt gefunden, so wird mit extractRightHole der am weitesten links stehende Teil des zweiten Produktterms extrahiert, welcher der rechte Nachbar des jeweils aktuellen Loches ist.

$$\begin{aligned}
\text{extractRightHoleOuter} \in{}& \sum_{i=1}^{n}\sum_{j=1}^{m_i}\left(\lfloor\prod_{k=1}^{j-1} A_{i,k}\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m_i} A_{i,k}\rfloor_{\mathbf{1}|}\right) \to \lfloor\bigcup_{i=1}^{n}\bigcup_{j=2}^{m_i} A_{i,j}\rfloor_{\mathbf{1}|}\\
\text{extractRightHoleOuter} ={}& (\mu(\lambda f\,.\,((\text{extractRightHoleInner} \triangledown f)\triangledown\ \text{extractRightHole})\ \text{isSum?})\triangledown\\
& \text{extractRightHole})\\
& \text{isSum?}\\
\text{extractRightHoleInner} \in{}& \sum_{j=1}^{m}\left(\lfloor\prod_{k=1}^{j-1} A_k\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m} A_k\rfloor_{\mathbf{1}|}\right) \to \lfloor\bigcup_{j=2}^{m} A_j\rfloor_{\mathbf{1}|}\\
\text{extractRightHoleInner} ={}& \mu(\lambda f\,.\,((\text{extractRightHole} \triangledown f)\triangledown\ \text{extractRightHole})\ \text{isSum?})\\
\text{extractRightHole} \in{}& \left(\lfloor\prod_{k=1}^{n} A_k\rfloor_{\mathbf{1}|} \times \lfloor\prod_{k=1}^{n} B_k\rfloor_{\mathbf{1}|}\right) \to \lfloor B_1\rfloor_{\mathbf{1}|}\\
\text{extractRightHole} ={}& \left(\grave{\imath} \circ \text{VOID} \triangledown \acute{\imath} \circ ((\grave{\pi} \triangledown \text{id})\ \text{isTuple?})\right) \circ \acute{\pi}
\end{aligned}$$

Die Funktion shiftRight verwendet das aktuelle Loch, um den aktuellen Kontext eine Stelle nach rechts zu schieben. Hierzu wird das Loch in den linken Produktterm am Ende eingefügt und mittels cutFirst das erste Element als neues Loch aus dem rechten Produktterm entfernt.

$$\begin{aligned}
\text{shiftRight} \in{}& H \to \sum_{i=1}^{n}\sum_{j=1}^{m_i}\left(\lfloor\prod_{k=1}^{j-1} A_{i,k}\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m_i} A_{i,k}\rfloor_{\mathbf{1}|}\right) \to\\
& \sum_{i=1}^{n}\sum_{j=1}^{m_i}\left(\lfloor A_{i,1} \times A_{i,2} \times \ldots A_{i,j-1} \times H\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+2}^{m_i} A_{i,k}\rfloor_{\mathbf{1}|}\right)\\
\text{shiftRight} ={}& \sum_{i=1}^{n}\sum_{j=1}^{m}(\text{lcombine}\ h \circ \grave{\pi}\ \vartriangle\ \text{cutFirst} \circ \acute{\pi})\\
\text{cutFirst} \in{}& \lfloor\prod_{k=1}^{n} A_k\rfloor_{\mathbf{1}|} \to \lfloor\prod_{k=2}^{n} A_k\rfloor_{\mathbf{1}|}\\
\text{cutFirst} ={}& \grave{\imath} \circ \text{VOID} \triangledown \acute{\imath} \circ ((\text{VOID} \triangledown \acute{\pi})\ \text{isTuple?})
\end{aligned}$$

Analog zu right lässt sich left angeben.

$$\begin{aligned}
& R \in \mathbb{P}(B,j,k),\ P \in \mathbb{P}(A,n,m),\ i \in \mathbb{N}^{\leq n},\ k \in \mathbb{N}^{\leq m_i}:\\
& \text{left} \in \text{Zipper}\ A_{i,k}\ R \to \lfloor \text{Zipper}\ A_{i,k-1}\ R\rfloor_{\mathbf{1}|}\\
& \text{left} = (((\grave{\imath} \circ \grave{\pi} \triangledown \acute{\imath} \circ (\acute{\pi} \circ \grave{\pi}\ \vartriangle\ \acute{\pi}))\ \text{dZero?} \circ\\
& \qquad (\text{extractLeftHoleOuter} \circ \text{selectFirstContext}\ \vartriangle\\
& \qquad\quad \acute{\imath} \circ (\text{selectTag}\ \vartriangle\ \lambda x\,.\,(\text{shiftLeft}\,(\grave{\pi}\,x)\,(\text{selectFirstContext}\ x))\ \vartriangle\ \text{tail} \circ \acute{\pi})))\triangledown\\
& \qquad \grave{\imath} \circ \text{VOID})\\
& \qquad \text{contextAvailable?}
\end{aligned}$$

Auch extractLeftHoleOuter und extractLeftHoleInner funktionieren wie ihre Gegenstücke für right. Allerdings ist extractLeftHole komplexer als extractRightHole, da auf Grund der Klammerung zuerst das linke innere Produkt des Kontextes vollständig durchlaufen werden muss.

$$\begin{aligned}
\text{extractLeftHoleOuter} &\in \sum_{i=1}^{n}\sum_{j=1}^{m_i}\left(\lfloor\prod_{k=1}^{j-1} A_{i,k}\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m_i} A_{i,k}\rfloor_{\mathbf{1}|}\right) \to \lfloor\bigcup_{i=1}^{n}\bigcup_{j=1}^{m_i-1} A_{i,j}\rfloor_{\mathbf{1}|} \\
\text{extractLeftHoleOuter} &= \ (\mu(\lambda\, f\, .\, ((\text{extractLeftHoleInner} \mathbin{\triangledown} f) \mathbin{\triangledown}\ \text{extractLeftHole})\,\text{isSum?}) \mathbin{\triangledown} \\
&\quad\ \text{extractLeftHole}) \\
&\quad\ \text{isSum?} \\
\text{extractLeftHoleInner} &\in \sum_{j=1}^{m}\left(\lfloor\prod_{k=1}^{j-1} A_k\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m} A_k\rfloor_{\mathbf{1}|}\right) \to \lfloor \bigcup_{j=1}^{m-1} A_j\rfloor_{\mathbf{1}|} \\
\text{extractLeftHoleInner} &= \mu(\lambda\, f\, .\, ((\text{extractLeftHole} \mathbin{\triangledown} f) \mathbin{\triangledown}\ \text{extractLeftHole})\,\text{isSum?}) \\
\text{extractLeftHole} &\in \left(\lfloor\prod_{k=1}^{n} A_k\rfloor_{\mathbf{1}|} \times \lfloor\prod_{k=1}^{m} B_k\rfloor_{\mathbf{1}|}\right) \to \lfloor A_k\rfloor_{\mathbf{1}|} \\
\text{extractLeftHole} &= \Big(\grave{\imath} \circ \text{VOID} \mathbin{\triangledown}\ \Big(\mu(\lambda\, f\, .\, (f \circ \acute{\pi} \mathbin{\triangledown} \acute{\imath} \circ \text{id})\,\text{isTuple?})\Big)\Big) \circ \grave{\pi}
\end{aligned}$$

Das Problem, das linke Tupel vollständig traversieren zu müssen, wirkt sich ebenfalls auf die Komplexität von cutLast aus, welches ansonsten von shiftLeft benutzt wird wie cutFirst aus shiftLeft.

$$\begin{aligned}
\text{shiftLeft} &\in H \to\ \sum_{i=1}^{n}\sum_{j=1}^{m_i}\left(\lfloor\prod_{k=1}^{j-1} A_{i,k}\rfloor_{\mathbf{1}|} \times \lfloor \prod_{k=j+1}^{m_i} A_{i,k}\rfloor_{\mathbf{1}|}\right) \to \\
&\qquad\ \sum_{i=1}^{n}\sum_{j=1}^{m_i}\left(\lfloor\prod_{k=1}^{j-2} A_{i,k}\rfloor_{\mathbf{1}|} \times \lfloor H \times \prod_{k=j+1}^{m_i} A_{i,k}\rfloor_{\mathbf{1}|}\right) \\
\text{shiftLeft} &= \sum_{i=1}^{n}\sum_{j=1}^{m}(\text{cutLast} \circ \grave{\pi} \mathbin{\triangledown}\ \text{rcombine}\ h\ \circ \acute{\pi}\,) \\
\text{cutLast} &\in \lfloor\prod_{k=1}^{n} A_k\rfloor_{\mathbf{1}|} \to \lfloor\prod_{k=1}^{n-1} A_k\rfloor_{\mathbf{1}|} \\
\text{cutLast} &= \Big((\mu(\lambda\, f\, .\, (\grave{\pi}\ \vartriangle\ f \circ \acute{\pi}\,))) \mathbin{\triangledown} \grave{\pi}\,)\,\text{rightIsTuple?} \mathbin{\triangledown} \grave{\imath}\ \circ \text{VOID}\Big)\ \text{isTuple?} \\
\text{rightIsTuple} &= \text{isTuple} \circ \acute{\pi}
\end{aligned}$$

Zusätzlich zu den Bewegungsoperationen ist es möglich, mit getHole und setHole den Wert des aktuellen Loches zu erhalten und zu verändern:

$$\begin{aligned}
\text{getHole} &\in \text{Zipper}\ H\ R \to H \\
\text{getHole} &= \grave{\pi}
\end{aligned} \tag{3.79}$$

$$\begin{aligned}
\text{setHole} &\in \text{Zipper}\ H\ R \times H \to \text{Zipper}\ H\ R \\
\text{setHole} &= (\acute{\pi}\ \vartriangle\ \acute{\pi} \circ \grave{\pi}\,)
\end{aligned} \tag{3.80}$$

Durch Navigation nach oben ist es möglich, wieder zu der Stelle zu kommen, an der der Zipper erzeugt wurde:

$$\begin{aligned}
\text{fromZipper} &\in \text{Zipper HOLEOF}(R)\ R \to R \\
\text{fromZipper} &= \mu\,(\lambda\, f\, .\, (f \circ \acute{\pi}\ \circ \text{up} \mathbin{\triangledown}\ \text{getHole})\ \text{contextAvailable?})
\end{aligned} \tag{3.81}$$

Hierbei ergibt sich die Besonderheit, dass das jeweilige Loch die Originaldatenstruktur oder ein beliebiges aus ihr ableitbares Loch sein kann. Dem kann mit der Funktion HOLEOF Rechnung getragen werden, die eine Datenstruktur auf eine Menge mit allen zu ihr denkbaren Löchern abbildet:

$$\begin{aligned} \mathrm{HOLEOF} &\in \mathbb{P}(A) \rightarrow \mathcal{P}(\mathbb{P}(A)) \\ \mathrm{HOLEOF}(R) &= \{H \mid \mathrm{PREDS}(H, R) \neq \{\} \vee H = R\} \end{aligned} \tag{3.82}$$

$\mathcal{P}(\mathbb{P}(A))$ bezeichnet die Potenzmenge von $\mathbb{P}(A)$, weil beliebige Mengen von polynomiellen Funktoren erzeugt werden können.

3.5. Umsetzung in Haskell

Das hier angegebene mathematische Modell beschreibt im Wesentlichen die Umsetzung in Haskell [Ada10]. Es musste allerdings, da Haskell von Haus aus keine Vereinigungstypen unterstützt, auf Allquantifizierung der Kontextinhalte ausgewichen werden. Die TAG-Funktion wurde mit dem *Scrap Your Boilerplate*-Ansatz [LP03] realisiert, der die Typidentifikation mit der compilergenerierbaren Typklasse Typeable realisiert. Der ständigen Veränderung des jeweiliges aktuellen Loches wurde mit GenericQ-Anfragefunktionen begegnet. Diese können mittels mkQ erzeugt werden. In der Notation dieser Arbeit entspricht das:

$$\forall A_k \in \mathbb{T}_n\colon\ \mathrm{GenericQ}\ R \in A_k \rightarrow R \tag{3.83}$$

$$\begin{aligned} &\forall R, \forall A_k \in \mathbb{T}_n, X \in \mathbb{T}_n\colon \\ &\ \mathrm{mkQ} \in R \times (X \rightarrow R) \rightarrow \mathrm{GenericQ}\ R \\ &\ \mathrm{mkQ} = \lambda\, z\, .\, (\acute{\pi}\, z \nabla\, (\lambda\, x\, .\, \grave{\pi}\ z))\, \mathrm{cast}_{\mathrm{TAG}(X)}? \end{aligned} \tag{3.84}$$

$$\begin{aligned} &\forall A_k \in \mathbb{T}_n\colon\ \mathrm{cast}_n \in \mathbb{N} \rightarrow A_k \rightarrow \mathit{bool} \\ &\ \mathrm{cast}_n\ x = \begin{cases} true & \text{falls tag } x = n \\ false & \text{sonst} \end{cases} \end{aligned} \tag{3.85}$$

Eine von mkQ erzeugte generische Anfrage versucht mit cast festzustellen, ob ihr Parametertyp dem Parametertyp der mkQ übergebenen Funktion entspricht. Stimmen die Typen überein, so wird die Funktion auf den Parameter angewandt und das Ergebnis zurückgegeben. Stimmen die Typen nicht, so wird auf den mkQ als erstes übergebenen Standardwert ausgewichen.

4. Entwurf

Im vorherigen Kapitel wurde gezeigt, wie die Manipulationsfunktionen auf Zippern dazu benutzt werden können, auf einheitlichem Weg durch beliebige Datenstrukturen zu navigieren. Ziel dieses Kapitels ist es, eine Funktionalität zu entwerfen, mit der die Traversierung weitestgehend automatisiert werden kann. Hierzu werden zunächst die Unterschiede und Gemeinsamkeiten der Preorder- und Breitendurchlaufstrategie [DD02] untersucht, um anschließend eine einheitliche Schnittstelle für möglichst allgemeine Durchlaufstrategien zu schaffen. Diese Schnittstelle wird zusammen mit der ihr zugehörigen Funktionalität mathematisch modelliert. Abschließend folgt ein Entwurf von einfachen Preorder- und Breitendurchläufen mit Ergebnislisten als Teil der Basisfunktionalität der Bibliothek.

4.1. Durchlaufstrategien

Die Traversierungsfunktionen auf Zippern ermöglichen es, Datenstrukturen als Bäume zu betrachten. Hierbei hat jeder Knoten im Baum eine geordnete Liste von Kindern, die über $\text{down}_{\text{Left}}$ zu erreichen und mittels right und left zu durchlaufen ist. Der Rückweg von einem Kind zu seinem Elternelement erfolgt mit up.

Preorder-Durchlauf: Beim Preorder-Durchlauf wird zunächst das Elternelement untersucht und anschließend die Liste mit Kindern abgearbeitet. Die Traversierung erfolgt zunächst in die Tiefe, da die Abarbeitung benachbarter Kinder bei den verbleibenden Aufgaben hinten angereiht wird und und ein Element nur untersucht wird, wenn sein linker Nachbar vollständig mit all seinen Kindern abgearbeitet wurde.

Breitendurchlauf: Beim Breitendurchlauf werden zuerst benachbarte Elternelemente besucht und der Besuch ihrer Kinder jeweils bei den verbleibenden Aufgaben hinten eingereiht. Die Baumstruktur wird, wie in [DD02] für binäre Suchbäume veranschaulicht, schichtweise in die Breite abgetragen.

Den Durchlaufstrategien ist gemeinsam, dass jeweils ausgehend vom aktuell besuchten Knoten eine Liste mit noch abzuarbeitenden Aufgaben aktualisiert wird. Sie unterscheiden sich hauptsächlich in der Reihenfolge, in der neue Aufgaben eingefügt werden. Obwohl bei den einfachen Durchlaufstrategien die Aufgabe beim Besuchen eines Knotens immer die selbe ist, wäre es durchaus denkbar, dass diese je nach gefundenem Knoten wechselt. So könnte zum Beispiel die Traversierung beim Auffinden eines Fehlers vollständig abgebrochen werden oder es könnten bestimmte Teile einer Datenstruktur mit einer anderen Durchlaufstrategie verarbeitet werden. Daher erscheint es sinnvoll, nicht nur eine Liste mit noch zu besuchenden Knoten zu verwalten, sondern die Liste aus Aufgaben, also Knoten und den auf ihnen anzuwendenden Besuchsfunktionen, bestehen zu lassen. Eine solche Besuchsfunktion bildet dann wiederum den ihr zugeordneten Knoten auf ein Besuchsergebnis und eine aktualisierte Aufgabenliste ab. Die Traversierung endet, sobald keine Aufgaben mehr anstehen. Die Besuchsergebnisse können völlig unterschiedlicher Natur sein. Es kann sein, dass es sich einfach um Werte handelt, die anschließend in einer

Liste oder einer Menge aufgesammelt werden sollen, aber es kann auch sein, dass beim Besuch Ein-/Ausgabeoperationen durchgeführt werden sollen. Auch ausgefeiltere Szenarien, wie die direkte Weiterverarbeitung in einem (endlichen) Automaten, sind vorstellbar.

4.2. Schnittstellenbeschreibung

Im vorherigen Abschnitt wurde gezeigt, dass es wesentlich ist, eine Liste aus zu besuchenden Knoten und den ihnen zugeordneten Besuchsfunktionen zu verwalten. Diese Liste wird, wenn die Traversierung beendet ist, leer. Die zu besuchenden Knoten sind Zipper. Der Funktor für eine Liste aus Aufgaben kann also zunächst angegeben werden als:

$$R \in \mathbb{P}: \quad \text{TaskList}'\ R\,A = \lfloor(\bigcup_{H \in \text{HOLEOF}(R)} \text{Task}'\ H\,R\,A) \times \text{TaskList}'\ R\,A\rfloor_{\mathbf{1}|} \tag{4.1}$$

Aufgaben bestehen aus einem Zipper, einer Besuchsfunktion und, um die Typvereinigung in den Aufgabelisten disjunkt zu machen, dem Identifikator des Loches des Zippers. Besuchsfunktionen bilden einen Knoten und eine Aufgabenliste auf ein Besuchsergebnis vom Typ A und eine neue Aufgabenliste ab. Aufgaben können also dargestellt werden als:

$$R \in \mathbb{P}\colon H \in \text{HOLEOF}(R)\colon \tag{4.2}$$
$$\text{Task}'\ H\,R\,A = \text{TAG}(H) \times \text{Zipper}\ H\,R \times \text{Visit}'\ H\,R\,A \tag{4.3}$$
$$\text{Visit}'\ H\,R\,A = (\text{Zipper}\ H\,R \times \text{TaskList}'\ R\,A) \to (A \times \text{TaskList}'\ R\,A) \tag{4.4}$$

Eine Besuchsfunktion sollte in der Lage sein, Seiteneffekte wie Ein-/ Ausgabeoperationen zu bewirken. In Haskell werden solche Seiteneffekte durch Monaden realisiert [Mar10]. Datentypen M, für die mindestens die Funktionen bind $(>>=)$ und return definiert sind, sind Monaden:

$$\forall A\colon\ \text{return} \in A \to M\ A \tag{4.5}$$
$$\forall A, B\colon (>>=) \in M\ A \to (A \to M\ B) \to M\ B \tag{4.6}$$

$$\forall f \in A \to B\colon\ \text{return}\ x\ >>=\ f = f\,x \tag{4.7}$$
$$m\ >>=\ \text{return}\ = m \tag{4.8}$$
$$m\ >>=\ (\lambda\, x\,.\, f\,x\ >>=\ g) = (m\ >>=\ f)\ >>=\ g \tag{4.9}$$

Man fordert von allen Monaden Funktoreigenschaften:

$$\forall f \in A \to B\colon\ M\,f \in M A \to M B \tag{4.10}$$
$$M\,f = \lambda\, m\,.\, m\ >>=\ \lambda\,.\,x\ \text{return}\,(f\,x) \tag{4.11}$$

Die Funktoreigenschaften sind konsistent, da

$$\begin{aligned} M\ \text{id} &= \lambda\, m\,.\, m\ >>=\ \lambda\,.\,x\ \text{return}\,(\text{id}\ x) \\ &= \lambda\, m\,.\, m\ >>=\ \text{return} \\ &\overset{4.8}{=} \lambda\, m\,.\, m = \text{id} \end{aligned}$$

$$
\begin{aligned}
\forall (f,g) \in (A \to B)^2 \colon\ M\, f \circ M\, g &= (\lambda\, m\, .\, m \ >>= \ \lambda\, .\, x \text{ return}\,(f\, x)) \\
&\quad \circ (\lambda\, m\, .\, m \ >>= \ \lambda\, .\, x \text{ return}\,(g\, x)) \\
&= \lambda\, m\, .\, ((m \ >>= \ \lambda\, .\, x \text{ return}\,(g\, x)) \ >>= \ \lambda\, .\, x \text{ return}\,(f\, x)) \\
&\overset{4.9}{=} \lambda\, m\, .\, (m \ >>= \ \lambda\, x\, .\, \text{return}\,(g\, y) \ >>= \ \lambda\, y\, .\, \text{return}\,(f\, z)) \\
&\overset{4.7}{=} \lambda\, m\, .\, (m \ >>= \ \lambda\, x\, .\, \text{return}\,(f \circ g)\, x) \\
&\overset{4.11}{=} M\,(f \circ g)
\end{aligned}
$$

Somit kann man beim Entwurf den Besuchsfunktionen mehr Spielraum gewähren, indem man sie um einen Funktor F erweitert:

$R \in \mathbb{P}$:

$$\text{TaskList } F\, R\, A = \lfloor (\bigcup_{H \in \text{HOLEOF}(R)} \text{Task } F\, H\, R\, A) \times \text{TaskList } F\, R\, A \rfloor_{\mathbf{1}|} \tag{4.12}$$

$R \in \mathbb{P}$: $H \in \text{HOLEOF}(R)$:

$$\text{Task } F\, H\, R\, A = \text{TAG}(H) \times \text{Zipper } H\, R \times \text{Visit } F\, H\, R\, A \tag{4.13}$$

$$\text{Visit } F\, H\, R\, A = (\text{Zipper } H\, R \times \text{TaskList } F\, R\, A) \to (A \times \text{TaskList } F\, R\, A) \tag{4.14}$$

Die Aufgabenliste wird schrittweise verarbeitet. Ein Schritt besteht zunächst daraus, die vorderste Aufgabe zu extrahieren und auszuführen. Ist die Aufgabenliste leer, so kann dies mit einer VOID-Rückgabe signalisiert werden, ansonsten sollten das Ergebnis und die aktualisierte Aufgabenliste bereitgestellt werden. Hierfür soll die Funktion runTask verwendet werden:

$$
\begin{aligned}
&\text{runTask} \in \text{TaskList } F\, R\, A \to \lfloor F\,(A \times \text{TaskList } F\, R\, A) \rfloor_{\mathbf{1}|} \\
&\text{runTask} = \grave{i} \circ \text{id} \triangledown \acute{i} \circ (\lambda\, l\, .\, (((\text{getFunction} \circ \grave{\pi}\,)\, l) \circ (\text{getZipper} \circ \grave{\pi} \ \vartriangle \ \acute{\pi}\,))\, l)
\end{aligned}
\tag{4.15}
$$

Hierbei wählen getFunction und *getZipper* die Besuchsfunktion und den Zipper aus einer Aufgabe aus:

$$
\begin{aligned}
&\text{getFunction} \in \text{Task } F\, H\, A\, R \to \text{Visit } F\, H\, R\, A \\
&\text{getFunction} = \acute{\pi} \circ \acute{\pi}
\end{aligned}
\tag{4.16}
$$

$$
\begin{aligned}
&\text{getZipper} \in \text{Task } F\, H\, A\, R \to \text{Zipper } H\, R \\
&\text{getZipper} = \grave{\pi} \circ \acute{\pi}
\end{aligned}
\tag{4.17}
$$

Wenn die Ergebnisse bereitgestellt sind, müssen sie aufgesammelt werden. Auch dies kann schrittweise passieren. Dafür muss immer der bisherige Sammlungszustand zusammen mit dem neuen Ergebnis in einen neuen Sammlungszustand überführt werden. Sonderfälle stellen hierbei die Schaffung eines Anfangszustandes und der Umgang mit der abgearbeiteten Aufgabenliste dar. Beide Sonderfälle können vereint werden, indem der initiale Sammlungszustand geschaffen wird, wenn die Liste vollständig abgearbeitet wurde, und die Ergebnisse in der umgekehrten Reihenfolge ihrer Bereitstellung in die Sammlung eingefügt werden. Es ist zu beachten, dass die Ergebnisse bei ihrer Bereitstellung jeweils in einen Funktor eingebaut sind. Die Schnittstelle einer Funktion zum Sammeln von Ergebnissen ergibt sich also zu:

$$\text{Collect } F\, A\, C = \lfloor F\,(A \times C) \rfloor_{\mathbf{1}|} \to C \tag{4.18}$$

Nun können runTask und eine Funktion aus Collect verbunden werden und rekursiv aufgerufen, bis sich ihr Ergebnis nicht mehr verändert, weil die Aufgabenliste abgearbeitet wurde.

$$\begin{aligned} \text{zeverything} &\in \text{Collect}\ F\, A\, C \to \text{TaskList}\ F\, R\, A \to C \\ \text{zeverything}\ f &= \mu(\lambda\, g\, .\, f\ \circ\ (\mathrm{id} \mid (F\,(\mathrm{id} \times g)))\ \circ\ \text{runTask}) \end{aligned} \tag{4.19}$$

Die Wahl der Rekursionsreihenfolge bewirkt, dass jeweils die erste Applikation von runTask mit der letzten Applikation der übergebenen Funktion f korrespondiert. Der innere Rekursionsaufruf von g wird in den Funktor F eingebettet, damit Seiteneffekte auch zwischen den Rekursionsaufrufen möglich sind und alle Typen zueinander passen. Allgemein entspricht die Rekursionsstruktur der eines Hylomorphismus wie er in [MFP91] beschrieben wird. Diese Darstellung bringt insbesondere den Vorteil, dass runTask und f einzeln und ohne Rücksicht auf ihre spätere rekursive Verwendung entwickelt und getestet werden können. Der Funktionsname zeverything ist an die Benennung der High-Level-Funktionen zreduce, zeverywhere und zsomewhere, die zur Transformation von Zippern in *Scrap Your Zippers* dienen [Ada10], und an die ähnliche Funktion everything aus *Scrap Your Boilerplate* [LP03] angelehnt.

4.3. Basisfunktionalität

Mit der im vorherigen Abschnitt festgelegten Schnittstelle ist es möglich, Tiefen- und Breitensuche mit Ergebnislisten als Standardfunktionalität der Bibliothek zu entwerfen und mitzuliefern.

Collect-Funktion für Ergebnislisten

$$\text{ResultList}\ A = \lfloor A \times \text{ResultList}\ A \rfloor_{\mathbf{1}|} \tag{4.20}$$

$$\begin{aligned} &\text{collectList} \in \text{Collect}\ \mathrm{I}\, \lfloor A \rfloor_{\mathbf{1}|}\ (\text{ResultList}\ A) \\ &\text{collectList} = \grave{i}\ \circ \text{VOID} \triangledown (\lambda\, x\, .\, ((\lambda\, y\, .\, \acute{\pi}\, x) \triangledown (\lambda\, y\, .\, \acute{i}\, x))\, (\grave{\pi}\ x)) \end{aligned} \tag{4.21}$$

Als Funktor wird, da im einfachsten Fall kein Funktor gebraucht wird, der Identitätsfunktor verwendet. Wenn keine Liste und kein einzufügendes Ergebnis existieren, wird eine leere Liste zurückgegeben. Ansonsten wird das Ergebnis, falls vorhanden, am Anfang der übergebenen Liste eingefügt.

Preorder-Durchlauf

$$\begin{aligned} &\text{preorder} \in \text{GenericQ}\, \lfloor A \rfloor_{\mathbf{1}|} \to \text{Visit}\ F\, H\, R\, \lfloor A \times \text{Zipper}\ H\, R \rfloor_{\mathbf{1}|} \\ &\text{preorder}\ q = \text{pure}_F \circ \\ &\qquad \lambda\, x\, .\, (\, (((\mathrm{id} \vartriangle\ \lambda\, y\, .\, \grave{\pi}\ x) \\ &\qquad\qquad \vartriangle\ \ \lambda\, y\, .\, ((\, \text{maybeCons}\, ((\text{down}_{\text{Left}}\ \circ \grave{\pi}\,)\, x)\, (\text{preorder}\ q) \circ \\ &\qquad\qquad\qquad \text{maybeCons}\, ((\text{right}\ \circ \grave{\pi}\,)\, x)\, (\text{preorder}\ q) \circ \acute{\pi}\,)\, x)) \\ &\qquad\qquad \triangledown\, ((\mathrm{id} \vartriangle\ \lambda\, y\, .\, \grave{\pi}\ x) \\ &\qquad\qquad\quad \vartriangle\ \lambda\, y\, .\, ((\text{maybeCons}\, ((\text{right}\ \circ \grave{\pi}\,)\, x)\, (\text{preorder}\ q) \circ \acute{\pi}\,)\, x))) \\ &\qquad\quad \circ\ q\ \circ\ \text{getHole} \circ \grave{\pi}\,)\, x \end{aligned} \tag{4.22}$$

Bei preorder wird als Funktor, um möglichst viel Variabilität zu schaffen, ein beliebiger applikativer Funktor F zugelassen. Mit der übergebenen generischen Anfrage wird versucht, das aktuelle Loch zu validieren. Wenn das Loch nicht kompatibel (z.B. aufgrund eines unpassenden Typs) zur Anfragefunktion ist, kann diese Void zurückgeben und damit signalisieren, dass die Suche auch auf den Kindern des Loches fortgesetzt werden soll. Andernfalls werden nur die rechten Nachbarn betrachtet. Sowohl Kinder als auch rechte Nachbarn werden wieder mit preorder durchsucht. Die Funktion maybeCons erstellt, wenn die Kinder existieren, die passenden Aufgaben und stellt sie der Liste verbleibender Aufgaben voran:

$$\begin{aligned}\text{maybeCons} \in \lfloor \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \to \text{Visit } F\, H\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \to \\ \text{TaskList } F\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \\ \text{maybeCons } z\, f\, l = ((\lambda\, x \,.\, l) \triangledown \acute{i} \circ (((\text{tag} \circ \text{getHole}) \vartriangle (\text{id} \vartriangle \lambda\, y \,.\, f)) \vartriangle \lambda\, y \,.\, l))\, z\end{aligned} \tag{4.23}$$

Breitendurchlauf

$$\begin{aligned}&\text{breadthfirst} \in \text{GenericQ } \lfloor A \rfloor_{\mathbf{1}|} \to \text{Visit } F\, H\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \\ &\text{breadthfirst } q = \text{pure}_F \circ \\ &\quad \lambda\, x \,.\, (\, (((\text{id} \vartriangle \lambda\, y \,.\, \grave{\pi}\; x) \\ &\qquad \vartriangle \lambda\, y \,.\, ((\, \text{maybeAppend } ((\text{down}_{\text{Left}} \circ \grave{\pi}\,)\, x)\, (\text{breadthfirst } q) \circ \\ &\qquad\qquad \text{maybeAppend } ((\text{right} \circ \grave{\pi}\,)\, x)\, (\text{breadthfirst } q) \circ \acute{\pi}\,)\, x)) \\ &\qquad \triangledown\, ((\text{id} \vartriangle \lambda\, y \,.\, \grave{\pi}\; x) \\ &\qquad\quad \vartriangle \lambda\, y \,.\, ((\text{maybeAppend } ((\text{right} \circ \grave{\pi}\,)\, x)\, (\text{breadthfirst } q) \circ \acute{\pi}\,)\, x))) \\ &\qquad \circ\, q \circ \text{getHole} \circ \grave{\pi}\,)\, x\end{aligned} \tag{4.24}$$

Die Funktion breadthfirst ist fast identisch zu preorder, jedoch werden hier neue Aufgaben, nachdem maybeAppend die Aufgabenliste bis an ihr Ende durchlaufen hat, mit insert eingefügt:

$$\begin{aligned}\text{maybeAppend} \in \lfloor \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \to \text{Visit } F\, H\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \to \\ \text{TaskList } F\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \\ \text{maybeAppend } z\, f\, l = ((\lambda\, x \,.\, l) \triangledown \\ (\lambda\, x \,.\, (\mu\, \lambda\, g \,.\, ((\text{insert } x\, f) \triangledown \acute{i} \circ (\text{id} \times g)))\, l))\, z\end{aligned} \tag{4.25}$$

$$\begin{aligned}\text{insert} \in \text{Zipper } H\, R \to \text{Visit } R\, H\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|} \to \\ \mathbf{1} \to \text{TaskList } F\, R\, \lfloor A \times \text{Zipper } H\, R \rfloor_{\mathbf{1}|}\end{aligned} \tag{4.26}$$

$$\text{insert } z\, f = \acute{i} \circ (((\lambda\, y \,.\, (\text{tag} \circ \text{getHole})\, z) \vartriangle ((\lambda\, y \,.\, z) \vartriangle \lambda\, y \,.\, f)) \vartriangle \grave{i} \circ \text{id})$$

5. Implementierung

Im letzten Kapitel wurden eine Schnittstelle und eine Basisimplementierung entworfen, deren Umsetzung in Haskell in diesem Kapitel beschrieben werden soll. Hierfür wird zunächst kurz auf die Modulstruktur und die verwendeten Bibliotheken eingegangen. Anschließend werden die Besonderheiten bei der Umsetzung der mathematischen Beschreibung in Quellcode erläutert.

5.1. Modulstruktur

In Haskell ist es, wie in den meisten Programmiersprachen, üblich, Bibliotheksmodule hierarchisch zu ordnen [Mar10]. Die hier erstellte Bibliothek arbeitet mit Datenstrukturen, die kompatibel mit *Scrap your Boilerplate* sind. *Scrap Your Boilerplate*-Pakete sind in der Regel in Unterhierarchien von Data.Generics angeordnet. Dementsprechend wurde Data.Generics.Validation als Modulname gewählt. Ferner wurden zum automatischen Testen die Module Data.Generics.Validation.Test und Main erstellt. Der Quellcode wurde mit englischen Kommentaren im Haddock-Stil [MW10] versehen.

5.2. Verwendete Bibliotheken

Neben den im Haskell-Standard [Mar10] definierten Bibliotheken wurden noch weitere verwendet.

Haskell Basisbibliothek Dem frei verfügbaren und plattformunabhängigen Glasgow Haskell Compiler (*GHC*) [Gla], welcher de-facto Standard für die Haskellentwicklung ist, liegt die Bibliothek *base* [Bas] bei. Die hier entwickelte Bibliothek wurde mit der Version *base-4.2.0.2* erstellt und getestet. Die in Listing 5.1 aufgeführten Importe stammen dort her und dienen dem Umgang mit Funktoren (1) sowie der Berechnung von Fixpunkten (2).

```
1 import Control.Applicative
2 import Data.Function
```

Listing 5.1: Importe aus *base-4.2.0.2*

Transformers Die Bibliothek *transformers* [Tra] erweitert den Haskellstandard um nützliche Funktionen für die Komposition von Funktoren. Sie wurde in Version *transformers-0.2.2.0* für die Importe Data.**Functor**.Compose und Data.**Functor**.Identity verwendet.

Scrap Your Boilerplate In Kapitel 3.5 wurde die Verwendung von *Scrap Your Boilerplate* erörtert. Die zugehörige Bibliothek *syb* kommt in Version 0.2 [SYB] zum Einsatz, um generische Anfragen zu ermöglichen. Sie ist ebenfalls notwendige Abhängigkeit von *Scrap Your Zippers*. Die Einbindung erfolgt mittels **import** Data.Generics, und die Verwendung erfordert außerdem die gängige Compilererweiterung *RankNTypes* [GHCb, HUG] für die mehrfach polymorphe Verwendung von GenericQ.

Scrap Your Zippers Basierend auf [Ada10] wurde von Michael D. Adams die Bibliothek *syz* [SYZ] entwickelt, die Zipper für alle Datenstrukturen aus der Typklasse Data bereitstellt. Sie wurde in Version 0.2.0.0 verwendet, mit **import** Data.Generics.Zipper eingebunden und bildet die wichtigste Grundlage für diese Arbeit.

QuickCheck Für den automatisierten Softwaretest wird *QuickCheck* in Version 2.4.0.1 [Qui] benötigt.

5.3. Umsetzung

Listing 5.2 zeigt zunächst die für die Umsetzung deklarierten Typen.

```haskell
1  -- Types
2
3  -- | A list of tasks running in a functor of type @f@ on zippers with
       root type
4  -- @r@ producing results typed @a@.
5  type TaskList f r a = [Task f r a]
6
7  -- | A single traversal Task.
8  data Task f r a = Task
9    { getZipper :: Zipper r -- ^ Current position
10   , getFunction :: Visit f r a -- ^ Function to apply to @zipper@
11   }
12
13 -- | Function to visit a zipper and complete a single task.
14 type Visit f r a = Zipper r -- ^ The zipper to visit
15   -> TaskList f r a -- ^ List of remaining tasks
16   -> f (a, TaskList f r a) -- ^ Result and updated list of remaining
         tasks
17
18 -- | Function to collect results.
19 type Collect f a c =
20   Compose Maybe (Compose f ((,) a)) c -- ^ Result and collection in
         previous state or nothing
21   -> c -- ^ updated or initialized collection
```

Listing 5.2: Typdeklaration

Die Typen entsprechen konzeptionell TaskList, Task, Visit und Collect aus Kapitel 4.2. Es ergibt sich jedoch der wichtige Unterschied, dass der Typparameter für das aktuelle Loch jeweils entfällt, da *Scrap Your Zippers* den Lochtypen vollständig hinter dem Zipper-Interface verbirgt. Dies vereinfacht vor allem den Typen Task, da die Identifikatoren zur disjunkten Typvereinigung in einer tieferen Schicht verborgen werden. In Zeile 20 wurde für Collect ausgenutzt, dass **Maybe** komponiert mit einem Produkt einen Funktor ergibt.

Mathematisch lässt sich dies wie folgt zeigen:

$$
\begin{aligned}
\forall f \in A \to B\colon\ (\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id}))f &\in \lfloor R \times A \rfloor_{\mathbf{1}|} \to \lfloor R \times B \rfloor_{\mathbf{1}|} \\
(\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id}))\, f &= \mathrm{id} \mid (\underline{\mathrm{R}} \times f) \\
(\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id}))\ \mathrm{id} &= \mathrm{id} \mid (\underline{\mathrm{R}} \times \mathrm{id}) \\
&= (\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id})) = \mathrm{id} \\
((\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id}))\, f) \circ ((\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id}))\, g) &= (\mathrm{id} \mid (\underline{\mathrm{R}} \times f)) \circ (\mathrm{id} \mid (\underline{\mathrm{R}} \times g)) \\
&= (\mathrm{id} \circ \mathrm{id}) \mid (\underline{\mathrm{R}} \circ \underline{\mathrm{R}} \times f \circ g) \\
&= \mathrm{id} \mid (\underline{\mathrm{R}} \times f \circ g) \\
&= (\mathrm{Maybe} \circ (\underline{\mathrm{R}} \times \mathrm{id}))\, (f \circ g)
\end{aligned}
$$

Listing 5.3 beinhaltet die Umsetzung der Kernfunktionalität zeverything und runTask.

```
1  -- Core functions
2
3  -- | Traverse a zipper and collect the results.
4  zeverything :: (Functor f, Data r) =>
5    Collect f a c -- ^ Function used to collect results
6    -> Visit f r a -- ^ Initial visitation function
7    -> Zipper r -- ^ Position to start with
8    -> c -- ^ Collection of results
9  zeverything cf tf z =
10  fix (\ f -> cf . fmap f . runTask) $ [(Task z tf)]
11
12 ---- Internal functions
13
14 -- | Run the first task and return its results.
15 runTask :: (Functor f, Data r) =>
16   TaskList f r a -- ^ Available tasks
17   -> Compose Maybe (Compose f ((,) a)) (TaskList f r a) -- ^ Result
       and new list of tasks
18 runTask ts =
19   case ts of
20     [] -> Compose Nothing
21     ((Task z f):ts') -> Compose . Just . Compose $ f z ts'
```

Listing 5.3: Kernfunktionalität

Die Funktion zeverything (Zeilen 4-10) bekommt im Gegensatz zu der in Kapitel 4.2 entworfenen Variante keine Aufgabenliste, sondern eine initiale Besuchsfunktion und einen Zipper als Startposition. Die Konstruktion der Aufgabenliste hieraus wird der Übersichtlichkeit halber vor dem Anwender verborgen. Der Fixpunktoperator μ entspricht in Haskell der Funktion fix aus Data.Function. Die oben erklärte Komposition von Maybe und dem Produkt vereinfacht die innere Anwendung der Funktion f zu einer Funktoranwendung, die in Haskell mit `fmap :: Functor f => (a -> b) -> f a -> f b` stattfindet. Die Funktion runTask (Zeilen 14 ff.) lässt sich in Haskell durch **case**-Abfragen mit Patternmatching lesbarer darstellen als mit den in der mathematischen Notation verwendeten Operatorprimitiven.

Es verbleiben noch die Basisfunktionalitäten preorder, breadthfirst und collectList für die Traversierung und das Aufsammeln von Ergebnissen in Listen (Listing 5.4).

```
-- Basic Visit and Collect functions

-- | Perform a single preorder (top-down, left-right) search step.
-- Stop descending the current path if the supplied query is successful.
-- Attach the current position to the query result.
-- Can use any applicative functor @f@.
preorder :: (Applicative f, Data r) =>
  GenericQ (Maybe a) -- ^ Query to be applied
  -> Visit f r (Maybe (a, Zipper r)) -- ^ Step result
preorder q z zs = pure $
  case query q z of
    Nothing ->
      (Nothing, maybeCons (down' z) (preorder q) . maybeCons (right z) (preorder q) $ zs)
    (Just res) -> (Just (res, z), maybeCons (right z) (preorder q) zs)
  where
    maybeCons z f l = case z of
      (Just x) -> (Task x f) : l
      Nothing -> l

-- | Perform a single breadthfirst (left-right, top-down) search step.
-- Stop descending the current path if the supplied query is successful.
-- Attach the current position to the query result.
-- Can use any applicative functor @f@.
breadthfirst :: (Applicative f, Data a) =>
  GenericQ (Maybe r) -- ^ Query to be applied
  -> Visit f a (Maybe (r, Zipper a)) -- ^ Step result
breadthfirst q z zs = pure $
  case query q z of
    Nothing ->
      (Nothing, maybeAppend (down' z) (breadthfirst q) . maybeAppend (right z) (breadthfirst q) $ zs)
    (Just res) -> (Just (res, z), maybeAppend (right z) (breadthfirst q) zs)
  where
    maybeAppend z f l = case z of
      (Just x) -> l ++ [Task x f]
      Nothing -> l

-- | Collect results in a list.
-- Uses the identity functor.
collectList :: Collect Identity (Maybe a) [a]
collectList s = case getCompose s of
  Nothing -> []
  (Just x) ->
    case (runIdentity . getCompose) x of
      (Just res, rs) -> (res:rs)
      (Nothing, rs) -> rs
```

Listing 5.4: Basisfunktionen

Die Funktionen preorder (Zeilen 7-18), breadthfirst (Zeilen 25-36) und collect (Zeilen 40 ff.) sind ebenfalls analog zu ihrem Entwurf in Kapitel 4.3 implementiert. Auch hier ergibt sich ein Lesbarkeitsgewinn durch Patternmatching und **case**-Abfragen. Das Anhängen an eine Liste erfolgt durch den Haskelloperator (++) :: [a] −> [a] −> [a], was maybeAppend vereinfacht (Zeile 34). Die Funktion $\mathrm{down}_{\mathrm{Left}}$ entspricht in der *Scrap Your Zippers*-Bibliothek down', da down dort das am weitesten rechts befindliche Kind als neues Loch setzt (Zeilen 13 und 30). Für die Komposition einer generischen Anfrage und getHole stellt *Scrap Your Zippers* die Funktion query :: GenericQ b −> Zipper a −> b zur Verfügung. Der Quellcode befindet sich noch einmal vollständig im Anhang (Listing A.1).

6. Test

Die Fehlerfreiheit von Datenstrukturen kann nur sichergestellt werden, wenn die hierzu verwendete Bibliothek selbst fehlerfrei ist. Daher wird in diesem Kapitel die Implementierung eines automatisierten Tests für das in den vorangegangenen Kapiteln entwickelte Modul Data.Generics.Validation beschrieben. Hierbei wird auf die dafür relevanten Teile der Haskell-Bibliothek *QuickCheck* [CH00, Qui] eingegangen.

6.1. Modulstruktur

Der Test ist auf zwei Dateien aufgeteilt. Das Modul Data.Generics.Validation.Test enthält den eigentlichen Testcode, der in *QuickCheck* in Form von *Properties* (Eigenschaften) von Funktionen beschrieben ist, welche selbst wieder Funktionen sind. Ferner gibt es ein Modul Main, das nichts weiter tut, als die von *QuickCheck* aufgesammelten Eigenschaften passend aufzurufen und hierfür ein ausführbares Programm bereit zu stellen (Listing 6.1).

```haskell
1  module Main where
2
3  import Test.QuickCheck
4  import qualified Data.Generics.Validation.Test
5
6  args = stdArgs
7    { maxSuccess = 999
8    , maxSize = 999
9    }
10
11 main = do
12   Data.Generics.Validation.Test.properties (quickCheckWithResult args)
```

Listing 6.1: Main

Bei Programmaufruf und Einstieg in die Main-Funktion wird zum Abarbeiten der einzelnen Eigenschaften die Funktion quickCheckWithResult verwendet, welche die Testergebnisse und eventuelle Fehler in der Konsole ausgibt. Der Aufruf wird mit den Standardargumenten von *QuickCheck* parametrisiert, welche so verändert wurden, dass 999 erfolgreiche Tests pro Eigenschaft notwendig sind und zufällig generierte Datenstrukturen jeweils bis zu 999 einzelne Elemente haben können (Zeilen 6-9).
Listing 6.2 zeigt den Anfang des Modules Data.Generics.Validation.Test.

```haskell
{-# LANGUAGE TemplateHaskell, DeriveDataTypeable, RankNTypes #-}
module Data.Generics.Validation.Test
  ( properties
  ) where

import Data.Generics.Validation

import Test.QuickCheck
import Test.QuickCheck.All
```

```
10
11 import Data.Generics
12 import Data.Generics.Zipper
13
14 import Data.Functor.Compose
15 import Data.Functor.Identity
16 import Control.Monad
17 import Data.Maybe
18 import Data.List (unfoldr)
19
20 properties = $(forAllProperties)
```

Listing 6.2: Einleitung des Testmoduls

Es werden zunächst die Compilererweiterungen *TemplateHaskell* [SJ02], *DerivingDataTypeable* [GHCa] und *RankNTypes* [GHCb] aktiviert (Zeile 1). *TemplateHaskell* dient hierbei dazu, *QuickCheck* den Testcode auf Funktionen, deren Name mit prop_ beginnt, durchsuchen zu lassen und für diese die Funktion properties zu generieren (Zeile 20), welche alle aufgesammelten Tests durchführt. *DerivingDataTypeable* ermöglicht es, vom Compiler automatisch die Instanziierung der Typklassen Typeable und Data vorzunehmen, die den in Kapitel 3.5 beschriebenen Mechanismus zur Typidentifikation umsetzen und es der *Scrap Your Zippers*-Bibliothek ermöglichen, die einzelnen Bestandteile der polynomiellen Datentypen zu analysieren. *RankNTypes* dient erneut dazu, die mehrfache Polymorphie von GenericQ-Anfragen in Haskell ausdrücken zu können.
Importiert werden das zu testende Modul Data.Generics.Validation (Zeile 6), die *QuickCheck*-Bibliothek (Zeilen 8 und 9), *Scrap Your Boilerplate* und *Scrap Your Zippers* (Zeilen 11 und 12) sowie einige Funktionen aus *transformers* für die Funktorkomposition (Zeile 14), den Identitätsfunktor (Zeile 15), den Umgang mit Monaden (Zeile 16), den bequemeren Umgang mit dem *Maybe*-Typ (Zeile 17) und **unfoldr** auf Listen, was später zu einer Referenzimplementierung der Breitensuche dient.

6.2. Test-Eigenschaften

Die einzelnen implementierten Bibliotheksfunktionen werden von *QuickCheck* mit randomisierten Testdaten aufgerufen und auf im Test definierte Eigenschaften (*Properties*) geprüft. In diesem Abschnitt soll die Implementierung dieser Definition vorgestellt werden.

Eigenschaften von runTask: Die Funktion runTask soll, wenn sie für eine leere Liste aufgerufen wird den Wert **Nothing** liefern. Dies wird mit der Testeigenschaft prop_emptyTaskList aus Listing 6.3 überprüft.

```
1 -- | Check that runTask terminates for empty lists.
2 prop_emptyTaskList = isNothing . getCompose $ runTask emptyList
3   where
4     emptyList :: TaskList Identity Int Int
5     emptyList = []
```

Listing 6.3: Eigenschaft von runTask für leere Listen

Der Typ der einzelnen Aufgaben ist, weil die Liste leer ist, frei wählbar und wird in Zeile 4 willkürlich auf TaskList Identity **Int Int** gesetzt.
In Listing 6.4 wird runTask auf eine nicht leere Liste angewandt. Diese besteht aus

zwei Aufgaben. Die erste Aufgabe beinhaltet einen Zipper auf eine von *QuickCheck* zufällig erzeugte Datenstruktur (Parameter td, Zeile 8) und eine Besuchsfunktion, welche den Zipper unverändert zusammen mit einer leeren Liste verbleibender Aufgaben zurückgibt (Zeile 16). Die zweite Aufgabe ist eine Dummyaufgabe (Zeilen 1-3), die, sollte sie ausgewertet werden, den Test sofort fehlschlagen lässt.

```
1  -- | Return a dummy task to fill task lists.
2  dummyTask :: Task f r a
3  dummyTask = error "This task may not be used"
4
5  -- | Check that runTask uses the first elements visit function.
6  prop_runTaskStep ::
7    ( Arbitrary a, Data a, Show a, Eq a) => a -> Bool
8  prop_runTaskStep td =
9    case getCompose $ runTask [(Task (toZipper td) visit),
       dummyTask] of
10     Just x ->
11       case (runIdentity . getCompose $ x) of
12         (res, []) -> res == td
13         _ -> False
14     _ -> False
15   where
16     visit z ts = Identity (fromZipper z, [])
```

Listing 6.4: Eigenschaft von runTask für nicht leere Listen

Als Ergebnis werden der übergebene Zipper und eine leere Liste erwartet, sonst war der Test nicht erfolgreich (Zeilen 8 und 9).

Eigenschaften von collectList: Ähnlich wie bei runTask werden bei collectList auch zwei Fälle unterschieden. Im Initialfall wird **Nothing** übergeben und eine leere Liste als Ausgabe erwartet (Listing 6.5).

```
-- | Checks that collectList creates a new List.
prop_collectListInitial =
  null . collectList $ nothingInt
  where
    nothingInt :: Compose Maybe (Compose Identity ((,) (Maybe
       Int))) [Int]
    nothingInt = Compose Nothing
```

Listing 6.5: Eigenschaft von collectList im Initialfall

Auch hier erfolgt die Einbettung von **Nothing** in den erwarteten Funktor willkürlich. Gibt es schon eine beliebige erstellte Liste und ein einzufügendes Element, so soll *collectList* das einzufügende Element der Liste voranstellen, wenn es nicht den Wert **Nothing** hat. Dieser zweite Fall wird mit prop_collectListStep für von *QuickCheck* beliebig erzeugte Elemente und Listen überprüft (Listing 6.6).

```
-- | Checks that collectList filters Nothing values and updates
   the result list.
prop_collectListStep x xs =
  case x of
    (Just i) -> (i:xs) == (collectList $ wrapped x xs)
    Nothing -> xs == (collectList $ wrapped x xs)
  where
```

```
7       wrapped x xs = Compose . Just . Compose $ Identity (x, xs)
```

Listing 6.6: Eigenschaft von collectList für vorhandene Listen

Vorhandene Elemente werden der Einfachheit halber in den Identitätsfunktor eingebaut (Zeile 7), wobei auch jeder andere Funktor hätte gewählt werden können.

Eigenschaften von preorder und breadthfirst: Um preorder und breadthfirst zu testen, macht es Sinn, sie auf zufälligen Zippern starten zu lassen. Damit *QuickCheck* randomisiert Zipper erzeugen kann, muss Zipper zu einer Instanz der Typklasse Arbitrary gemacht werden, und im Fehlerfall müssen die den Fehler erzeugenden Zipper ausgegeben werden können (Listing 6.7).

```
1  -- | Obtain a Zipper at an arbitrary position inside an arbitrary
       data structure.
2  instance (Arbitrary a, Data a) => Arbitrary (Zipper a) where
3    arbitrary = do
4      d <- arbitrary
5      dirs <- listOf (elements [down', left, right, up])
6      return $ foldl tryToMove (toZipper d) dirs
7      where
8        tryToMove pos dir = case dir pos of
9          Nothing -> pos
10         (Just pos') -> pos'
11
12 -- | Show an arbitrary Zipper by showing its hole
13 instance (Show a) => Show (Zipper a) where
14   show = query gshow
```

Listing 6.7: Typinstanzen für Arbitrary und Show von Zipper

Ein zufälliger Zipper wird erzeugt, indem zunächst eine zufällige Datenstruktur erzeugt wird (Zeile 4). Anschließend wird eine zufällige Liste aus Bewegungsoperationen erstellt (Zeile 5) und hieraus alle möglichen Bewegungen der Reihe nach auf einem Zipper mit der zufälligen Struktur als Wurzel ausgeführt (Zeile 6). Der Code befindet sich in einem **do**-Block, da die arbitrary-Funktion monadisch ist, weil sie als Seiteneffekt den Zustand eines Zufallsgenerators aktualisiert. Damit beliebige Zipper angezeigt werden können, müssen sie in einen String verwandelbar sein, also Instanzen der Typklasse **Show** sein. In Zeile 14 wird dies realisiert, indem die *Scrap Your Boilerplate*-Funktion gshow auf ihr Loch angewandt wird, welche beliebige Instanzen der Typklasse Data in Strings umwandeln kann.
Als nächstes wird eine geeignete Testdatenstruktur benötigt, auf der die zufälligen Zipper operieren. Listing 6.8 zeigt eine solche Struktur und ihre Arbitrary-Instanz.

```
-- | Nested test data structure
data TestData a b c =
  Void -- ^ No content
  | Simple a -- ^ Just a single element of type a
  | Product a b -- ^ Product of two elements
  | Triple a b c -- ^ Triple of three elements
  | Four a b c a -- ^ Four elements
  | List a (TestData a b c) -- ^ List of an a and TestData
  | HaskellTuple (a, b) -- ^ Haskell version of products
  | HaskellList [a] -- ^ Haskell version of lists
  deriving (Eq, Data, Typeable, Show)
```

```
12
13 -- | Construct arbitrary TestData
14 instance (Arbitrary a, Arbitrary b, Arbitrary c) =>
15   Arbitrary (TestData a b c) where
16   arbitrary =
17     oneof [ return Void
18           , liftM Simple arbitrary
19           , liftM2 Product arbitrary arbitrary
20           , liftM3 Triple arbitrary arbitrary arbitrary
21           , liftM4 Four arbitrary arbitrary arbitrary arbitrary
22           , liftM2 List arbitrary arbitrary
23           , liftM HaskellTuple arbitrary
24           , liftM HaskellList arbitrary
25           ]
```

Listing 6.8: TestData und Arbitrary-Instanz

TestData wurde möglichst komplex und verschachtelt definiert, um die in Haskell vorkommenden Typen möglichst umfassend zu simulieren. Der Compiler wird in Zeile 11 angewiesen, automatisch Instanzen der Typklassen **Eq**, Data, Typeable und **Show** zu erzeugen, sodass TestData-Werte verglichen, von *Scrap Your Boilerplate* erfasst und in Strings verwandelt werden. Zufällige TestData-Werte werden erzeugt, indem ein zufälliger Konstruktor gewählt und mit

```
liftMn :: (a1 -> .. an -> r) -> m a1 -> .. m an -> m r
```

auf die hintereinander ausgeführten monadischen Operationen zur Erzeugung zufälliger Parameter angewandt wird.

In Kapitel 4.1 wurde bereits festgestellt, dass sich die preorder und breadthfirst im Wesentlichen nur in der Reihenfolge unterscheiden, in der sie in die Aufgabenliste neue Aufgaben einfügen. Dementsprechend kann für beide Funktionen der selbe grundsätzliche Testrahmen (Listing 6.9) benutzt werden.

```
type IntTestData = TestData (TestData Int Int Int) Int Int
type SelectFunction =
  [Task Identity IntTestData (Maybe (Int, Zipper IntTestData))]
  -> Task Identity IntTestData (Maybe (Int, Zipper IntTestData))

-- | Check a single search step of a preorder or breadthfirst search.
_prop_searchStep ::
  (GenericQ (Maybe Int) -> Visit Identity IntTestData (Maybe (Int, Zipper IntTestData))) -- ^ Search function
  -> SelectFunction -- ^ Selects the task for down from a list
  -> SelectFunction -- ^ Selects the task for right from a list
  -> SelectFunction -- ^ Selects the task for down or right from an incomplete list
  -> Maybe Int -- ^ Static simulated query result
  -> Zipper (TestData (TestData Int Int Int) Int Int) -- ^ Zipper at test position
  -> Bool
_prop_searchStep search downTask rightTask downOrRightTask qres zipper =
  case (qres, search (const qres) zipper [dummyTask]) of
    (Nothing, (Identity (Nothing, ts@[t1, t2, t3]))) ->
```

```
18        (isJust $ down' zipper) && check up (downTask ts)
19        && (isJust $ right zipper) && check left (rightTask ts)
20      (Nothing, (Identity (Nothing, ts@[t1, t2]))) ->
21        (check up (downOrRightTask ts) && (isNothing $ right zipper))
22        || (check left (downOrRightTask ts) && (isNothing $ down' zipper))
23      (Nothing, (Identity (Nothing, [dt]))) ->
24        (isNothing $ right zipper) && (isNothing $ down' zipper)
25      (Just res, (Identity ((Just (res', z)), ts@[t1, t2]))) ->
26        res == res' && checkHoles z zipper
27        && check left (downOrRightTask ts) && (isJust $ right zipper)
28      (Just res, (Identity ((Just (res', z)), [dt]))) ->
29        res == res' && checkHoles z zipper && (isNothing $ right zipper)
30      _ -> False
31    where
32      check back t =
33        case (back . getZipper) t of
34          (Just x) -> checkHoles x zipper
35          Nothing -> False
36      checkHoles z1 z2 =
37        (getHoleInt z1 == getHoleInt z2)
38        && (getHoleTestDataInner z1 == getHoleTestDataInner z2)
39        && (getHoleIntTestData z1 == getHoleIntTestData z2)
40      getHoleInt :: Zipper a -> Maybe Int
41      getHoleInt = getHole
42      getHoleTestDataInner :: Zipper a -> Maybe (TestData Int Int Int)
43      getHoleTestDataInner = getHole
44      getHoleIntTestData :: Zipper a -> Maybe (IntTestData)
45      getHoleIntTestData = getHole
```

Listing 6.9: Gemeinsamer Testrahmen für preorder und breadthfirst

Der Testrahmen wird mit der zu testenden Suchfunktion (Zeile 8), Auswahlfunktionen (Zeilen 9-11), einem simulierten Anfrageergebnis (Zeile 10) und dem zu besuchenden Knoten parametrisiert. Der zu durchsuchende Datentyp ist TestDataInt (Zeile 1), eine verschachtelte Version von TestData mit **Int**-Parametern. Gesucht wird nach **Int**-Elementen. Die Auswahlfunktionen selektieren aus einer nach dem Knotenbesuch aktualisierten Aufgabenliste die neu eingefügte Aufgabe für das Kindelement des besuchten Knotens (Zeile 9), das rechte Nachbarelement (Zeile 10) oder, falls eines der Elemente nicht vorhanden ist, die für das jeweils vorhandene Element eingefügte Aufgabe (Zeile 11). Ihrem Typ wurde, um die Lesbarkeit zu erhöhen, das Synonym SelectFunction (Zeilen 2-4) gegeben. Nun wird eine Suche auf dem übergebenen Zipper durchgeführt (Zeile 16). Hierbei wird eine Anfragefunktion verwendet, die konstant das simulierte Suchergebnis qres liefert. Als verbleibende Aufgabenliste wird eine Liste verwendet, die nur den bereits besprochenen dummyTask enthält, weil preorder und breadthfirst die Werte in dieser Liste nicht untersuchen sollen und ein Fehler auftritt, wenn dummyTask ausgewertet wird. Für alle möglichen Werte, die die so aufgerufene Suchfunktion annimmt, muss gelten, dass das Ergebnis zum aktuell besuchten Knoten mit dem simulierten Ergebnis qres identisch ist. Ferner

muss die neue Aufgabenliste die passenden Aufgaben in der passenden Reihenfolge enthalten. War das Anfrageergebnis qres kein Treffer (**Nothing**), müssen der Kindknoten des aktuellen Knotens und der rechte Nachbarknoten in der neuen Aufgabenliste vorkommen, falls diese existieren (Zeilen 17-24). Andernfalls muss der rechte Nachbarknoten eingefügt werden (Zeilen 25-29). Dass die richtigen Knoten eingefügt wurden, wird mit check (Zeilen 32-35) geprüft, indem mit der jeweils passend gewählten Operation back von der eingefügten Position einen Schritt zurück gegangen wird und mit checkHoles sichergestellt wird, dass die Lochwerte der so erreichten Position mit der Ursprungsposition übereinstimmen. Die Funktion checkHoles probiert hierbei passend getypte Anfragen (Zeilen 40ff.) für alle möglichen Lochtypen, die in einer IntTestData-Datenstruktur vorkommen können. Als Funktor wird erneut der Identitätsfunktor gewählt, weil er besonders einfach zu handhaben ist.
Die Tests für preorder und breadthfirst müssen jetzt nur noch den Testrahmen mit passenden Auswahlfunktionen instantiieren (Listing 6.10).

```
1  -- | Checks a single preorder step on a given data structure with a given
2  -- query result.
3  prop_preorderStep :: Maybe Int -> Zipper (TestData (TestData Int Int Int) Int Int) -> Bool
4  prop_preorderStep =
5    _prop_searchStep preorder head (head . tail) head
6
7  -- | Checks a single breadthfirst step on a given data structure with a given
8  -- query result.
9  prop_breadthfirstStep :: Maybe Int -> Zipper (TestData (TestData Int Int Int) Int Int) -> Bool
10 prop_breadthfirstStep =
11   _prop_searchStep breadthfirst last (head . tail . reverse) (last)
```

Listing 6.10: Tests für preorder und breadthfirst

Die preorder-Funktion fügt jeweils die Aufgabe für den Kindknoten als erstes und die für den rechten Nachbarknoten als zweites in die Aufgabenliste ein (Zeile 5), während die breadthfirst-Funktion sie als Letztes bzw. Vorletztes (Zeile 11) einfügt.

Eigenschaften von zeverything: Die zeverything-Funktion besteht nur aus der Verknüpfung von Funktionen und ist dementsprechend korrekt, wenn die verknüpften Funktionen korrekt sind. Dennoch soll sie auch getestet werden. Hierzu eignen sich Referenzimplementierungen 6.11 eines Preorder-Durchlaufs und einer Breitensuche auf dem bereits vorgestellten TestData-Typ. Bei den Referenzimplementierungen werden die in Kapitel 2.4 vorgestellten Probleme erneut deutlich, da sehr genau auf die zu durchsuchende Datenstruktur eingegangen werden muss und ein Aufschluss über den Kontext von Fehlerstellen nicht gegeben wird.

```
-- | Manual preorder search of Int TestData
toPreorder :: TestData a b a -> [a]
toPreorder (List x y) = (x:(toPreorder y))
toPreorder (Void) = []
toPreorder (Simple x) = [x]
toPreorder (Product x y) = [x]
toPreorder (Triple x y z) = [x, z]
```

```
8 toPreorder (Four x y z x2) = [x, z, x2]
9 toPreorder (HaskellTuple (x, y)) = [x]
10 toPreorder (HaskellList xs) = xs
11
12 -- | Manual breadthfirst search of Int TestData
13 toBreadthFirst :: TestData a b a -> [a]
14 toBreadthFirst x = unfoldr bfsStep [x]
15   where
16     bfsStep toDo = case toDo of
17       ((List x y):xs) -> Just (x, xs ++ [y])
18       ((Void):xs) -> bfsStep xs
19       ((Simple x):xs) -> Just (x, xs)
20       ((Product x y):xs) -> Just (x, xs)
21       ((Triple x y z):xs) -> Just (x, xs++[(Simple z)])
22       ((Four x y z x2):xs) -> Just (x, xs++[(Triple z y x2)])
23       ((HaskellTuple (x, y)):xs) -> Just (x, xs)
24       ((HaskellList []):xs) -> bfsStep xs
25       ((HaskellList [x]):xs) -> Just (x, xs)
26       ((HaskellList (x:ys)):xs) -> bfsStep (xs++[(Simple x),
            (HaskellList ys)])
27       [] -> Nothing
```

Listing 6.11: Referenzimplementierungen von preorder und breadthfirst auf TestData

Beide Implementierungen liefern für Werte vom Typ TestData a b a Listen mit allen vorkommenden Teilwerten vom Typ a. Die toPreorder-Funktion bedient sich einer einfachen Fallunterscheidung und ruft sich, wenn angebracht, rekursiv selbst auf (Zeile 3). Dagegen ist die toBreadthFirst-Funktion aufgrund der eher iterativen Struktur der Breitensuche etwas komplizierter. Sie verwendet eine Liste mit noch abzuarbeitenden Teildaten und iteriert auf dieser Liste mit unfodlr :: (b −> (**Maybe** (a, b))−> b −> [a] aus Data.**List**. Dabei wird die bfsStep-Funktion (Zeilen 16ff.) immer wieder mit der abzuarbeitenden Liste aufgerufen, bis sie **Nothing** zurückgibt. Gibt sie nicht **Nothing** zurück, wird der erste Teil ihrer Rückgabe in die Ergebnisliste übernommen und der zweite Teil als ihre nächste Eingabe verwendet. In [MFP91] wird dieses Rekursionsmuster als Anamorphismus detaillierter beschrieben. Die bfsStep-Funktion terminiert die Suche mit **Nothing**, wenn die Liste noch abzuarbeitender Teildaten leer ist (Zeile 27) und hängt ansonsten die je nach Fall unterschiedlichen zu betrachtenden Teildaten hinten an die noch ausstehende Liste an.
Das Testen von zeverything gegen die beiden Referenzimplementierungen ist relativ einfach (Listing 6.12)

```
_prop_manualSearch ::
  (TestData Int Char Int -> [Int]) -- ^ Handwritten search
      function
  -> (GenericQ (Maybe Int) -> Visit Identity (TestData Int Char
      Int) (Maybe (Int, Zipper (TestData Int Char Int)))) -- ^
      Search function
  -> (TestData Int Char Int) -- ^ Test data structure
  -> Bool
_prop_manualSearch manualSearch search td =
  manualSearch td == (map fst $ zeverything collectList (search
      q) (toZipper td))
  where
    q :: GenericQ (Maybe Int)
```

```haskell
10        q = mkQ Nothing Just
11
12  -- | Checks if zeverything collectList preorder is the same as manual
13  -- preorder on TestData Int Int Int
14  prop_manualPreorder :: TestData Int Char Int -> Bool
15  prop_manualPreorder =
16    _prop_manualSearch toPreorder preorder
17
18  -- | Checks if zeverything collectList breadthfirst is the same as manual
19  -- breadthfirst on TestData Int Int Int
20  prop_manualBreadthfirst :: TestData Int Char Int -> Bool
21  prop_manualBreadthfirst =
22    _prop_manualSearch toBreadthFirst breadthfirst
```

Listing 6.12: Testen von zeverything gegen Referenzimplementierungen

Für TestData **Int Char Int** wird eine Suche nach allen **Int**-Werten gestartet und die Ergebnisse der Referenzsuche mit denen von zeverything mit collectList und dem korrespondierenden Suchalgorithmus verglichen, wobei die auf die Fundstellen fokussierten Zipper ignoriert werden (_prop_manualSearch, Zeilen 1-9). Die *Properties* für preorder (Zeilen 12-16) und breadthfirst (Zeilen 18 ff.) tun nichts weiter als passend parametrisiert _prop_manualSearch aufzurufen.

Der Testquellcode befindet sich noch einmal vollständig im Anhang (A.2, A.3).

6.3. Testergebnisse

Das im vorherigen Abschnitt implementierte Testprogramm läuft für die in der Arbeit verwendete Implementierung fehlerfrei durch (Ausgabe in Listing 6.13).

```
=== prop_emptyTaskList on ./Data/Generics/Validation/Test.hs:23 ===
+++ OK, passed 999 tests.
=== prop_runTaskStep on ./Data/Generics/Validation/Test.hs:33 ===
+++ OK, passed 999 tests.
=== prop_collectListInitial on ./Data/Generics/Validation/Test.hs:46 ===
+++ OK, passed 999 tests.
=== prop_collectListStep on ./Data/Generics/Validation/Test.hs:53 ===
+++ OK, passed 999 tests.
=== prop_preorderStep on ./Data/Generics/Validation/Test.hs:149 ===
+++ OK, passed 999 tests.
=== prop_breadthfirstStep on ./Data/Generics/Validation/Test.hs:155 ===
+++ OK, passed 999 tests.
=== prop_manualPreorder on ./Data/Generics/Validation/Test.hs:200 ===
+++ OK, passed 999 tests.
=== prop_manualBreadthfirst on ./Data/Generics/Validation/Test.hs:206 ===
+++ OK, passed 999 tests.
```

Listing 6.13: Testergebnis

Der Test wurde mit dem Programm Haskell Program Coverage (HPC) [HPC, OGS08] einer Code-Coverage-Analyse unterzogen, um seine Vollständigkeit sicherzustellen. Hierzu

module	**Top Level Definitions**			**Alternatives**			**Expressions**		
	%	**covered / total**		**%**	**covered / total**		**%**	**covered / total**	
module Data.Generics.Validation	100%	5/5		100%	14/14		99%	121/122	
Program Coverage Total	100%	5/5		100%	14/14		99%	121/122	

Abbildung 6.1.: Ergebnis der Code-Coverage-Analyse des Tests mit HPC

wurde das Main-Modul mit *GHC* und dem Compiler-Flag *-fhpc* compiliert. Anschließend wurde mit dem Befehl *hpc markup* ein Report über die beim Testaufruf erreichten Quellcodeteile generiert. Abbildung 6.1 zeigt das Ergebnis dieses Reports. Es ist ersichtlich, dass der Code des Moduls Data.Generics.Validation beim Testen vollständig ausgeführt wurde.

7. Anwendungsbeispiel

Die Funktionsweise der entwickelten Bibliothek und ihre Vorteile sollen in diesem Kapitel an einem kleinen praxisnahen Beispiel verdeutlicht werden. Hierzu wird zunächst ein mögliches Datenmodell für ein Internetforum skizziert. Anschließend wird demonstriert, wie unter Verwendung von zeverything die Forendaten auf Viren geprüft werden könnten.

7.1. Datenmodell für ein Internetforum

Das Datenmodell zu einem Internetforum könnte wie in Listing 7.1 angegeben aussehen.

```haskell
data Forum = Forum
  { admins :: [User]
  , groups :: [Group]
  , topics :: [Topic]
  } deriving (Data, Typeable, Show)

data User = User
  { name :: String
  , nickname :: String
  , postcount :: Int
  , profilepicture :: File
  } deriving (Data, Typeable, Show)

data Group = Group
  { moderator :: User
  , users :: [User]
  } deriving (Data, Typeable, Show)

data Topic = ComplexTopic
  { title :: String
  , subTopics :: [Topic]
  }
  | SimpleTopic
  { title :: String
  , posts :: [Post]
  } deriving (Data, Typeable, Show)

data Post = Post
  { authorName :: String
  , text :: String
  , date :: String
  , attachments :: [File]
  , reply :: Maybe Post
  } deriving (Data, Typeable, Show)

```

```
36 data File = File
37   { filename :: String
38   , content :: String
39   } deriving (Data, Typeable, Show)
```

Listing 7.1: Datenmodell eines Internetforums

Ein solches Forum besteht aus einer Liste von Nutzern, die Administratoren sind, einer Liste von Gruppen und einer Liste von Themen (Zeilen 1-5). Benutzer haben einen Namen, ein Pseudonym (nickname), eine Anzahl von geschriebenen Foreneinträgen (postcount) und ein Profilbild (Zeilen 7-12). Gruppen haben einen Moderator und eine Liste von Benutzern, die der Gruppe zugeordnet sind (Zeilen 14-17). Themen können komplexe Themen mit einem Titel und einer Liste von Unterthemen sein, oder einfache Themen mit einem Titel und einer Liste aus Foreneinträgen (Zeilen 19-26). Foreneinträge vermerken den Namen ihres Autors, haben einen Beitragstext, einen Zeitstempel, eine Liste von hochgeladenen Dateien (attachments) und eventuell einen Antwortbeitrag (Zeilen 28-34). Dateien bestehen aus einem Dateinamen und dem Dateiinhalt (content), der als String abgespeichert wird (Zeilen 36-39).

7.2. Validierung hochgeladener Dateien

Es soll sichergestellt werden, dass keine der als Profilbild oder Anhang zu einem Forenbeitrag hochgeladenen Dateien einen Virus beinhaltet. Zum Überprüfen der Dateien kann zum Beispiel der frei verfügbare Virenscanner ClamAV [Cla] zum Einsatz kommen. Dieser bietet das aus der Kommandozeile startbare Programm *clamavscan* an, das bei Aufruf von

```
clamavscan --infected -
```

einen String aus seiner Standardeingabe liest, den Inhalt auf Viren überprüft, eventuelle Warnmeldungen in der Standardausgabe zurückgibt und mit seinem Systemrückgabecode das Vorhandensein eines Virus signalisiert.

Nun sollen mit zeverything alle Daten vom Typ File in einem Forum gefunden werden, ihr Dateiinhalt mit dem Virenscanner überprüft werden und anschließend eine Liste mit allen infizierten Dateien und ihren Positionen in der Forendatenstruktur zurückgegeben werden. Als Traversierungsalgorithmus kann ein einfacher Preorder-Durchlauf eingesetzt werden, der bereits implementiert wurde. Beim Aufsammeln der Ergebnisse soll jeweils der Virenscanner gestartet werden, und es sollen nur Ergebnisse mit infizierten Dateien übernommen werden. Listing 7.2 zeigt den hierfür erforderlichen Quellcode.

```
antiVirusCommand = "clamavscan"
antiVirusArguments = ["--infected", "-"]

data VirusPosition = VirusPosition
  { file :: File
  , position :: Zipper Forum
  , message :: String
  }

collectViruses :: Collect IO (Maybe (File, Zipper Forum)) (IO
    [VirusPosition])
collectViruses s =
  case getCompose s of
    Nothing -> return []
```

```
14      Just state -> do
15        (result, results) <- getCompose state
16        case result of
17          Nothing -> results
18          Just (file, position) -> do
19            (exitCode, avMsg, avErr) <- readProcessWithExitCode
                  antiVirusCommand antiVirusArguments (content file)
20            case exitCode of
21              ExitSuccess -> results
22              _ -> fmap ((:) (VirusPosition file position (avMsg ++
                  avErr))) results
23
24  findViruses = zeverything collectViruses (preorder fileQuery)
25    where
26      fileQuery :: GenericQ (Maybe File)
27      fileQuery = mkQ Nothing Just
```

Listing 7.2: Überprüfen der Forendatenstruktur auf Viren

In den Zeilen 1 und 2 werden zunächst noch einmal das Kommando zum Starten des Virenscanners und die zugehörigen Parameter definiert. Die Datenstruktur VirusPosition besteht aus der infizierten Datei, einem Zipper zur Markierung ihrer Position und der Ausgabe des Virenscanners (Zeilen 4-8). Die Funktion collectViruses ist eine Collect-Funktion für zeverything. Sie operiert in der **IO**-Monade, um beliebige Ein-/Ausgabeoperationen als Seiteneffekt durchführen zu können. Aufzusammelnde Ergebnisse des Preorder-Durchlaufs sind Dateien und ihre Positionen als Zipper in der Foren-Datenstruktur. Als Rückgabe wird eine Liste von VirusPosition-Daten erstellt. Beim initialen Aufruf von collectViruses wird eine leere Liste zurückgegeben (Zeile 13). Anschließend wird, falls der Preorder-Durchlauf im aktuellen Schritt ein File-Datum geliefert hat, die Funktion readProcessWithExitCode aus *base* verwendet, um den Virenscanner zu starten und den Dateiinhalt an seine Standardeingabe zu senden (Zeile 19). Daraufhin liegen die Systemrückgabe, die Standardausgabe und die Standardfehlerausgabe des Virenscanners in den Variablen exitCode, avMsg und avErr vor. Signalisiert der Systemrückgabecode, dass ein Virus gefunden wurde, wird ein VirusPosition-Datum zur Datei, ihrer Position sowie der Virenscanner-Ausgabe erstellt und zur Liste der infizierten Dateien hinzugefügt (Zeile 22). Die Funktion findViruses ruft jetzt einfach zeverything mit collectViruses und einem Preorder-Durchlauf, der generische Anfragen nach Dateien stellt, auf. Mit ihr kann ein Forum vollständig auf Viren durchsucht werden.

Der vollständige Beispielquellcode inklusive verwendeter Importe befindet sich noch einmal im Anhang A.4.

8. Abschließende Bewertung und Ausblick

Im Rahmen dieser Arbeit wurde eine Haskell-Bibliothek zur Validierung generischer Datenstrukturen erstellt. Hierzu wurden zunächst aus der Problematisierung bereits vorhandener Ansätze Anforderungen an einen neuen Ansatz gewonnen (Kapitel 2). Es hat sich herausgestellt, dass es für Validierungsaufgaben zentral ist, beliebige Datenstrukturen auf einheitlichem Weg traversieren zu können. Die theoretischen Grundlagen hierfür wurden in Kapitel 3 in Form von algebraischen Datentypen, Zippern und Operationen auf Zippern erörtert. Mit ihnen als Rüstzeug war es möglich, eine mathematische Beschreibung der Schnittstelle, die für verschiedene Durchlaufstrategien gebraucht wird, zu formulieren (Kapitel 4). Diese Schnittstelle wurde in Kapitel 5 in Haskell umgesetzt. Die Implementierung wurde im darauf folgenden Kapitel einem umfassenden automatisierten Test unterzogen, wobei die klare, schlanke und sauber getrennte mathematische Schnittstellenbeschreibung äußerst hilfreich war. Ein abschließendes praxisnahes Beispiel hat verdeutlicht, dass die entwickelte Bibliothek den formulierten Anforderungen (2.6) gerecht werden kann:

Effiziente Kodierbarkeit:
Selbst komplexe und verschachtelte Datenstrukturen können mit wenigen Zeilen Quellcode auf Fehler überprüft werden.

Wartbarkeit und Erweiterbarkeit:
Eine Erweiterung der Bibliotheksfunktionen ist problemlos möglich. Details der Datenstruktur finden sich nirgendwo im Validierungscode wieder, und notwendige Anpassungen bei Änderungen werden vom Compiler vorgenommen.

Ausdrucksstärke:
Zipper funktionieren auf beliebigen polynomiellen Funktoren und damit insbesondere auch auf allen Daten, die sich durch kontextfreie Grammatiken beschreiben lassen. Durch die Einbettung in Funktoren sind während der Validierung sogar Seiteneffekte wie Ein-/Ausgabefunktionen möglich.

A posteriori wählbare Validierungskriterien:
Die Validierungskriterien müssen nicht zur Compilezeit bekannt sein und können sogar von einem extern aufgerufenen Programm, wie z.B. einem Virenscanner, abhängen.

Aufschluss über Fehlerstellen:
Die Position von Fehlerstellen wird mit Zippern vollständig beschrieben.

Automatische Traversierung:
Es ist nicht notwendig, Traversierungscode fallabhängig von einzelnen Bestandteilen einer Datenstrukturen zu erstellen.

Trennbarkeit:
Der Validierungscode kann an beliebigen Stellen im Programm untergebracht werden und braucht lediglich die Definition der Datenstruktur und die hier entwickelte Bibliothek zu importieren.

Es ergeben sich einige interessante Fragestellungen für weitere Arbeiten. Hierzu gehört insbesondere die Laufzeit- und Speichereffizienz des vorgestellten Ansatzes. Auch der praktische Einsatz in einem größeren Kontext, wie einer verteilten Applikation mit einer großen Datenbank, wäre ein interessantes Unterfangen. Darüber hinaus existiert noch eine Fülle an weiteren Durchlaufstrategien, um die die Bibliothek erweitert werden könnte. Für große Datenmengen könnte sogar über den Einsatz randomisierter Strategien nachgedacht werden, die sich problemlos einbinden lassen sollten. Die Arbeit hat also einen Beitrag zu einem interessanten und weitreichenden Themenfeld leisten können.

Literaturverzeichnis

[AAMG03] Abbott, Michael ; Altenkirch, Thorsten ; McBride, Conor ; Ghani, Neil: Derivatives of containers. In: *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications*. Berlin, Heidelberg : Springer-Verlag, 2003. – ISBN 3–540–40332–9, 16–30

[AAMG04] Abbott, Michael ; Altenkirch, Thorsten ; McBride, Conor ; Ghani, Neil: Delta for Data: Differentiating Data Structures. In: *Fundamenta Informaticae* 65 (2004), August, 1–28. `http://portal.acm.org/citation.cfm?id=1227143.1227145`, Abruf: 14. Apr. 2010

[Ada10] Adams, Michael D.: Scrap Your Zippers: A Generic Zipper for Heterogeneous Types. In: *ICFP '10: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming* (2010), Sept., 13–24. `https://www.cs.indiana.edu/~adamsmd/papers/scrap_your_zippers/ScrapYourZippers-2010.pdf`, Abruf: 23. Feb. 2011. ISBN 978–1–4503–0251–7

[Bas] *Haskell-Basisbibliothek - base-4.2.0.2*. `http://hackage.haskell.org/packages/archive/base/4.2.0.2/doc/html/`, Abruf: 22. Mai 2011

[BD07] Badouel, Eric ; Djeumen, Rodrigue: Modular Grammars and Splitting of Catamorphisms / INRIA. Version: 2007. `http://hal.inria.fr/inria-00175793/PDF/RR-6313.pdf`, Abruf: 21. Mai 2011. Institut National des Sciences Appliquées de Rennes - Université de Rennes I, 2007 (RR-6313). – Research Report

[BPSM+08] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M. ; Maler, Eve ; Yergeau, François: Extensible Markup Language (XML) 1.0 (Fifth Edition). Version: 2008. `http://www.w3.org/TR/xml/`, Abruf: 24. Feb. 2011. 2008. – Forschungsbericht

[CH00] Claessen, Koen ; Hughes, John: QuickCheck: a lightweight tool for random testing of Haskell programs. In: *SIGPLAN Not.* 35 (2000), Sept., 268–279. `http://portal.acm.org/citation.cfm?id=351240.351266`, Abruf: 29. Dez. 2010

[Cla] *Virenscanner ClamAV*. `http://www.clamav.net/`, Abruf: 26. Mai 2011

[CM01] Clark, James ; Makoto, Murata: RELAX NG Specification. Version: 2001. `http://www.relaxng.org/spec-20011203.html`, Abruf: 24. Feb. 2011. 2001. – Forschungsbericht

[DD02] Doberkat, Ernst-Erich ; Dissmann, Stefan: *Einführung in die objektorientierte Programmierung mit Java*. München Wien : Oldenbourg, 2002. – ISBN 3486253425

[EP08] Erk, Katrin ; Priese, Lutz: *Theoretische Informatik*. Springer-Verlag, 2008. – ISBN 1645216

[Exp] *Expat - Eine XML-Parser Bibliothek*. http://expat.sourceforge.net/, Abruf: 23. Jan. 2011

[FGOG07] Froihofer, Lorenz ; Glos, Gerhard ; Osrael, Johannes ; Goeschka, Karl M.: Overview and Evaluation of Constraint Validation Approaches in Java. In: *Proceedings of the 29th International Conference on Software Engineering* (2007), Mai, 313–322. http://portal.acm.org/citation.cfm?id=1248864, Abruf: 6. Jan. 2011

[FW04] Fallside, David C. ; Walmsley, Priscilla: XML Schema Part 0: Primer Second Edition. Version: 2004. http://www.w3.org/TR/xmlschema-0/, Abruf: 24. Feb. 2011. 2004. – Forschungsbericht

[GHCa] *Glasgow Haskell Compiler - Deriving Data Typeable*. http://www.haskell.org/ghc/docs/7.0.2/html/users_guide/deriving.html#deriving-typeable, Abruf: 25. Mai 2011

[GHCb] *Glasgow Haskell Compiler - RankNTypes*. http://www.haskell.org/ghc/docs/7.0.3/html/users_guide/other-type-extensions.html, Abruf: 22. Mai 2011

[Gla] *Glasgow Haskell Compiler*. http://www.haskell.org/ghc/, Abruf: 22. Mai 2011

[Har10] Harold, Elliotte R.: The Java XML Validation API. In: *IBM developerWorks* (2010), Februar. http://www.ibm.com/developerworks/xml/library/x-javaxmlvalidapi.html, Abruf: 24. Feb. 2011

[Hof08] Hoffmann, Dirk W.: *Software-Qualität*. Springer-Verlag, 2008. – ISBN 3540763228

[HPC] *Haskell Program Coverage (HPC)*. http://projects.unsafeperformio.com/hpc/, Abruf: 25. Mai 2011

[Hue97] Huet, Gérard: The Zipper. In: *Journal of Functional Programming* 7 (1997), Sept., 549–554. http://portal.acm.org/citation.cfm?id=969867.969872, Abruf: 14. Apr. 2010

[HUG] *HUGS Haskell Compiler - Rank2Types*. http://cvs.haskell.org/Hugs/pages/users_guide/quantified-types.html, Abruf: 22. Mai 2011

[Id3] *id3lib - Eine MP3-Tag-Parser-Bibliothek*. http://id3lib.sourceforge.net/, Abruf: 23. Jan. 2011

[Jso] *Json - Eine JSON-Parser-Bibliothek für Haskell*. http://hackage.haskell.org/package/json, Abruf: 23. Jan. 2011

[LP03] Lämmel, Ralf ; Peyton Jones, Simon: Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In: *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)* 38 (2003), März, Nr. 3, 26–37. http://research.microsoft.com/en-us/um/people/simonpj/Papers/hmap/hmap.ps, Abruf: 29. Dez. 2010

[Mar10] MARLOW, Simon: Haskell 2010 Language Report. Version: 2010. `http://haskell.org/definition/haskell2010.pdf`, Abruf: 23. Jan. 2011. 2010. – Forschungsbericht

[McB01] MCBRIDE, Conor: *The Derivative of a Regular Type is its Type of One-Hole Contexts*. `http://strictlypositive.org/diff.pdf`. Version: 2001, Abruf: 14. Apr. 2010

[McB08] MCBRIDE, Conor: Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In: *SIGPLAN Not.* 43 (2008), January, 287–295. `http://doi.acm.org/10.1145/1328897.1328474`, Abruf: 14. Apr. 2010

[MFP91] MEIJER, Erik ; FOKKINGA, Maarten M. ; PATERSON, Ross: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. London, UK : Springer-Verlag, 1991. – ISBN 3–540–54396–1, 124–144

[MLMK05] MURATA, Makoto ; LEE, Dongwon ; MANI, Murali ; KAWAGUCHI, Kohsuke: Taxonomy of XML schema languages using formal language theory. In: *ACM Transactions on Internet Technology (TOIT)* 5 (2005), November, 660–704. `http://portal.acm.org/citation.cfm?id=1111631`, Abruf: 23. Feb. 2011

[MW10] MARLOW, Simon ; WAERN, David: Haddock User Guide. Version: 2010. `http://www.haskell.org/haddock/doc/html/index.html`, Abruf: 22. Mai. 2011. 2010. – Forschungsbericht

[OGS08] O'SULLIVAN, Bryan ; GOERZEN, John ; STEWART, Donald B.: *Real World Haskell*. O'Reilly Media, 2008 `http://book.realworldhaskell.org/`. – ISBN 9780596514983

[Qui] *QuickCheck-Bibliothek - QuickCheck-2.4*. `http://hackage.haskell.org/package/QuickCheck-2.4`, Abruf: 22. Mai 2011

[SJ02] SHEARD, Tim ; JONES, Simon P.: Template Meta-programming for Haskell. In: *ACM SIGPLAN Haskell Workshop 02*, ACM Press, 2002, 1–16

[SYB] *Scrap Your Boilerplate-Bibliothek - syb-0.3*. `http://hackage.haskell.org/package/syb-0.3`, Abruf: 22. Mai 2011

[SYZ] *Scrap Your Zippers-Bibliothek - syz-0.2.0.0*. `http://hackage.haskell.org/package/syz`, Abruf: 22. Mai 2011

[Tra] *Transfomers-Bibliothek - transformers-0.2.2.0*. `http://hackage.haskell.org/package/transformers`, Abruf: 22. Mai 2011

Index

Anhang A.

Quellcodes

```haskell
{-# LANGUAGE RankNTypes #-}

module Data.Generics.Validation
  (
  -- * Types
    TaskList
  , Task (..)
  , Visit
  , Collect
  -- * Core functions
  , zeverything
  , runTask
  -- * Basic Visit and Collect functions
  , preorder
  , breadthfirst
  , collectList
  ) where

import Control.Applicative
import Data.Functor.Compose
import Data.Functor.Identity
import Data.Function

import Data.Generics
import Data.Generics.Zipper

-- Types

-- | A list of tasks running in a functor of type @f@ on zippers with
    root type
-- @r@ producing results typed @a@.
type TaskList f r a = [Task f r a]

-- | A single traversal Task.
data Task f r a = Task
  { getZipper :: Zipper r -- ^ Current position
  , getFunction :: Visit f r a -- ^ Function to apply to @zipper@
  }

-- | Function to visit a zipper and complete a single task.
type Visit f r a = Zipper r -- ^ The zipper to visit
  -> TaskList f r a -- ^ List of remaining tasks
  -> f (a, TaskList f r a) -- ^ Result and updated list of remaining
      tasks
```

```haskell

-- | Function to collect results.
type Collect f a c =
  Compose Maybe (Compose f ((,) a)) c  -- ^ Result and collection in
      previous state or nothing
  -> c -- ^ updated or initialized collection

-- Core functions

-- | Traverse a zipper and collect the results.
zeverything :: (Functor f, Data r) =>
  Collect f a c -- ^ Function used to collect results
  -> Visit f r a -- ^ Initial visitation function
  -> Zipper r -- ^ Position to start with
  -> c -- ^ Collection of results
zeverything cf tf z =
 fix (\ f -> cf . fmap f . runTask) $ [(Task z tf)]

---- Internal functions

-- | Run the first task and return its results.
runTask :: (Functor f, Data r) =>
  TaskList f r a -- ^ Available tasks
  -> Compose Maybe (Compose f ((,) a)) (TaskList f r a) -- ^ Result
      and new list of tasks
runTask ts =
  case ts of
    [] -> Compose Nothing
    ((Task z f):ts') -> Compose . Just . Compose $ f z ts'

-- Basic Visit and Collect functions

-- | Perform a single preorder (top-down, left-right) search step.
-- Stop descending the current path if the supplied query is
    successful.
-- Attach the current position to the query result.
-- Can use any applicative functor @f@.
preorder :: (Applicative f, Data r) =>
  GenericQ (Maybe a) -- ^ Query to be applied
  -> Visit f r (Maybe (a, Zipper r)) -- ^ Step result
preorder q z zs = pure $
  case query q z of
    Nothing ->
      (Nothing, maybeCons (down' z) (preorder q) . maybeCons (right z)
          (preorder q) $ zs)
    (Just res) -> (Just (res, z), maybeCons (right z) (preorder q) zs)
  where
    maybeCons z f l = case z of
      (Just x) -> (Task x f) : l
      Nothing -> l

-- | Perform a single breadthfirst (left-right, top-down) search step.
-- Stop descending the current path if the supplied query is
    successful.
-- Attach the current position to the query result.
```

```haskell
-- Can use any applicative functor @f@.
breadthfirst :: (Applicative f, Data a) =>
  GenericQ (Maybe r) -- ^ Query to be applied
  -> Visit f a (Maybe (r, Zipper a)) -- ^ Step result
breadthfirst q z zs = pure $
  case query q z of
    Nothing ->
      (Nothing, maybeAppend (down' z) (breadthfirst q) . maybeAppend
          (right z) (breadthfirst q) $ zs)
    (Just res) -> (Just (res, z), maybeAppend (right z) (breadthfirst
        q) zs)
  where
    maybeAppend z f l = case z of
      (Just x) -> l ++ [Task x f]
      Nothing -> l

-- | Collect results in a list.
-- Uses the identity functor.
collectList :: Collect Identity (Maybe a) [a]
collectList s = case getCompose s of
  Nothing -> []
  (Just x) ->
    case (runIdentity . getCompose) x of
      (Just res, rs) -> (res:rs)
      (Nothing, rs) -> rs
```

Listing A.1: Validation.hs

```haskell
{-# LANGUAGE TemplateHaskell, DeriveDataTypeable, RankNTypes #-}
module Data.Generics.Validation.Test
  ( properties
  ) where

import Data.Generics.Validation

import Test.QuickCheck
import Test.QuickCheck.All

import Data.Generics
import Data.Generics.Zipper

import Data.Functor.Compose
import Data.Functor.Identity
import Control.Monad
import Data.Maybe
import Data.List (unfoldr)

properties = $(forAllProperties)

-- | Check that runTask terminates for empty lists.
prop_emptyTaskList = isNothing . getCompose $ runTask emptyList
  where
    emptyList :: TaskList Identity Int Int
    emptyList = []

```

```
-- | Return a dummy task to fill task lists.
dummyTask :: Task f r a
dummyTask = error "This task may not be used"

-- | Check that runTask uses the first elements visit function.
prop_runTaskStep ::
  (Arbitrary a, Data a, Show a, Eq a) => a -> Bool
prop_runTaskStep td =
  case getCompose $ runTask [(Task (toZipper td) visit), dummyTask] of
    Just x ->
      case (runIdentity . getCompose $ x) of
        (res, []) -> res == td
        _ -> False
    _ -> False
  where
    visit z ts = Identity (fromZipper z, [])

-- | Checks that collectList creates a new List.
prop_collectListInitial =
  null . collectList $ nothingInt
  where
    nothingInt :: Compose Maybe (Compose Identity ((,) (Maybe Int)))
        [Int]
    nothingInt = Compose Nothing

-- | Checks that collectList filters Nothing values and updates the
    result list.
prop_collectListStep x xs =
  case x of
    (Just i) -> (i:xs) == (collectList $ wrapped x xs)
    Nothing -> xs == (collectList $ wrapped x xs)
  where
    wrapped x xs = Compose . Just . Compose $ Identity (x, xs)

-- | Obtain a Zipper at an arbitrary position inside an arbitrary data
    structure.
instance (Arbitrary a, Data a) => Arbitrary (Zipper a) where
  arbitrary = do
    d <- arbitrary
    dirs <- listOf (elements [down', left, right, up])
    return $ foldl tryToMove (toZipper d) dirs
    where
      tryToMove pos dir = case dir pos of
        Nothing -> pos
        (Just pos') -> pos'

-- | Show an arbitrary Zipper by showing its hole
instance (Show a) => Show (Zipper a) where
  show = query gshow

-- | Nested test data structure
data TestData a b c =
  Void -- ^ No content
  | Simple a -- ^ Just a single element of type a
  | Product a b -- ^ Product of two elements
```

```haskell
  | Triple a b c -- ^ Triple of three elements
  | Four a b c a -- ^ Four elements
  | List a (TestData a b c) -- ^ List of an a and TestData
  | HaskellTuple (a, b) -- ^ Haskell version of products
  | HaskellList [a] -- ^ Haskell version of lists
  deriving (Eq, Data, Typeable, Show)

-- | Construct arbitrary TestData
instance (Arbitrary a, Arbitrary b, Arbitrary c) =>
  Arbitrary (TestData a b c) where
  arbitrary =
    oneof [ return Void
          , liftM Simple arbitrary
          , liftM2 Product arbitrary arbitrary
          , liftM3 Triple arbitrary arbitrary arbitrary
          , liftM4 Four arbitrary arbitrary arbitrary arbitrary
          , liftM2 List arbitrary arbitrary
          , liftM HaskellTuple arbitrary
          , liftM HaskellList arbitrary
          ]

type IntTestData = TestData (TestData Int Int Int) Int Int
type SelectFunction =
  [Task Identity IntTestData (Maybe (Int, Zipper IntTestData))]
  -> Task Identity IntTestData (Maybe (Int, Zipper IntTestData))

-- | Check a single search step of a preorder or breadthfirst search.
_prop_searchStep ::
  (GenericQ (Maybe Int) -> Visit Identity IntTestData (Maybe (Int, Zipper IntTestData))) -- ^ Search function
  -> SelectFunction -- ^ Selects the task for down from a list
  -> SelectFunction -- ^ Selects the task for right from a list
  -> SelectFunction -- ^ Selects the task for down or right from an incomplete list
  -> Maybe Int -- ^ Static simulated query result
  -> Zipper (TestData (TestData Int Int Int) Int Int) -- ^ Zipper at test position
  -> Bool
_prop_searchStep search downTask rightTask downOrRightTask qres zipper =
  case (qres, search (const qres) zipper [dummyTask]) of
    (Nothing, (Identity (Nothing, ts@[t1, t2, t3]))) ->
      (isJust $ down' zipper) && check up (downTask ts)
      && (isJust $ right zipper) && check left (rightTask ts)
    (Nothing, (Identity (Nothing, ts@[t1, t2]))) ->
      (check up (downOrRightTask ts) && (isNothing $ right zipper))
      || (check left (downOrRightTask ts) && (isNothing $ down' zipper))
    (Nothing, (Identity (Nothing, [dt]))) ->
      (isNothing $ right zipper) && (isNothing $ down' zipper)
    (Just res, (Identity ((Just (res', z)), ts@[t1, t2]))) ->
      res == res' && checkHoles z zipper
      && check left (downOrRightTask ts) && (isJust $ right zipper)
    (Just res, (Identity ((Just (res', z)), [dt]))) ->
      res == res' && checkHoles z zipper && (isNothing $ right zipper)
```

```
      _ -> False
  where
    check back t =
      case (back . getZipper) t of
        (Just x) -> checkHoles x zipper
        Nothing -> False
    checkHoles z1 z2 =
      (getHoleInt z1 == getHoleInt z2)
      && (getHoleTestDataInner z1 == getHoleTestDataInner z2)
      && (getHoleIntTestData z1 == getHoleIntTestData z2)
    getHoleInt :: Zipper a -> Maybe Int
    getHoleInt = getHole
    getHoleTestDataInner :: Zipper a -> Maybe (TestData Int Int Int)
    getHoleTestDataInner = getHole
    getHoleIntTestData :: Zipper a -> Maybe (IntTestData)
    getHoleIntTestData = getHole

-- | Checks a single preorder step on a given data structure with a given
-- query result.
prop_preorderStep :: Maybe Int -> Zipper (TestData (TestData Int Int Int) Int Int) -> Bool
prop_preorderStep =
  _prop_searchStep preorder head (head . tail) head

-- | Checks a single breadthfirst step on a given data structure with a given
-- query result.
prop_breadthfirstStep :: Maybe Int -> Zipper (TestData (TestData Int Int Int) Int Int) -> Bool
prop_breadthfirstStep =
  _prop_searchStep breadthfirst last (head . tail . reverse) (last)

-- | Manual preorder search of Int TestData
toPreorder :: TestData a b a -> [a]
toPreorder (List x y) = (x:(toPreorder y))
toPreorder (Void) = []
toPreorder (Simple x) = [x]
toPreorder (Product x y) = [x]
toPreorder (Triple x y z) = [x, z]
toPreorder (Four x y z x2) = [x, z, x2]
toPreorder (HaskellTuple (x, y)) = [x]
toPreorder (HaskellList xs) = xs

-- | Manual breadthfirst search of Int TestData
toBreadthFirst :: TestData a b a -> [a]
toBreadthFirst x = unfoldr bfsStep [x]
  where
    bfsStep toDo = case toDo of
      ((List x y):xs) -> Just (x, xs ++ [y])
      ((Void):xs) -> bfsStep xs
      ((Simple x):xs) -> Just (x, xs)
      ((Product x y):xs) -> Just (x, xs)
      ((Triple x y z):xs) -> Just (x, xs++[(Simple z)])
      ((Four x y z x2):xs) -> Just (x, xs++[(Triple z y x2)])
```

```haskell
        ((HaskellTuple (x, y)):xs) -> Just (x, xs)
        ((HaskellList []):xs) -> bfsStep xs
        ((HaskellList [x]):xs) -> Just (x, xs)
        ((HaskellList (x:ys)):xs) -> bfsStep (xs++[(Simple x), (HaskellList ys)])
        [] -> Nothing

_prop_manualSearch ::
  (TestData Int Char Int -> [Int]) -- ^ Handwritten search function
  -> (GenericQ (Maybe Int) -> Visit Identity (TestData Int Char Int) (Maybe (Int, Zipper (TestData Int Char Int)))) -- ^ Search function
  -> (TestData Int Char Int) -- ^ Test data structure
  -> Bool
_prop_manualSearch manualSearch search td =
  manualSearch td == (map fst $ zeverything collectList (search q) (toZipper td))
  where
    q :: GenericQ (Maybe Int)
    q = mkQ Nothing Just

-- | Checks if zeverything collectList preorder is the same as manual
-- preorder on TestData Int Int Int
prop_manualPreorder :: TestData Int Char Int -> Bool
prop_manualPreorder =
  _prop_manualSearch toPreorder preorder

-- | Checks if zeverything collectList breadthfirst is the same as manual
-- breadthfirst on TestData Int Int Int
prop_manualBreadthfirst :: TestData Int Char Int -> Bool
prop_manualBreadthfirst =
  _prop_manualSearch toBreadthFirst breadthfirst
```

Listing A.2: Test.hs

```haskell
module Main where

import Test.QuickCheck
import qualified Data.Generics.Validation.Test

args = stdArgs
  { maxSuccess = 999
  , maxSize = 999
  }

main = do
  Data.Generics.Validation.Test.properties (quickCheckWithResult args)
```

Listing A.3: ValidationTest.hs

```haskell
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Generics
import Data.Generics.Zipper
import Data.Generics.Validation
```

```

import Data.Functor.Compose

import System.Process
import System.Exit

data Forum = Forum
  { admins :: [User]
  , groups :: [Group]
  , topics :: [Topic]
  } deriving (Data, Typeable, Show)

data User = User
  { name :: String
  , nickname :: String
  , postcount :: Int
  , profilepicture :: File
  } deriving (Data, Typeable, Show)

data Group = Group
  { moderator :: User
  , users :: [User]
  } deriving (Data, Typeable, Show)

data Topic = ComplexTopic
  { title :: String
  , subTopics :: [Topic]
  }
  | SimpleTopic
  { title :: String
  , posts :: [Post]
  } deriving (Data, Typeable, Show)

data Post = Post
  { authorName :: String
  , text :: String
  , date :: String
  , attachments :: [File]
  , reply :: Maybe Post
  } deriving (Data, Typeable, Show)

data File = File
  { filename :: String
  , content :: String
  } deriving (Data, Typeable, Show)

antiVirusCommand = "clamavscan"
antiVirusArguments = ["--infected", "-"]

data VirusPosition = VirusPosition
  { file :: File
  , position :: Zipper Forum
  , message :: String
  }

```

```haskell
collectViruses :: Collect IO (Maybe (File, Zipper Forum)) (IO
    [VirusPosition])
collectViruses s =
  case getCompose s of
    Nothing -> return []
    Just state -> do
      (result, results) <- getCompose state
      case result of
        Nothing -> results
        Just (file, position) -> do
          (exitCode, avMsg, avErr) <- readProcessWithExitCode
              antiVirusCommand antiVirusArguments (content file)
          case exitCode of
            ExitSuccess -> results
            _ -> fmap ((:) (VirusPosition file position (avMsg ++
                avErr))) results

findViruses = zeverything collectViruses (preorder fileQuery)
  where
    fileQuery :: GenericQ (Maybe File)
    fileQuery = mkQ Nothing Just
```

Listing A.4: Forum.hs

Eidesstattliche Versicherung

______________________________ ____________________

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

____

____

____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

__________________________ _______________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____________________________ _________________________

Ort, Datum Unterschrift