Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Ein Stilprüfer für Curry

Ning Cheng

März 2019

betreut von
Prof. Dr. Michael Hanus
M.Sc. Finn Teegen

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Kiel, 28. März 2019

____

Ning Cheng

Abstract

Formatierung und Stil sind nicht nur ein wesentlicher Teil der Syntax von einigen Programmiersprachen, sondern grundsätzlich ein Aspekt der Programmierung, der vor allem den Entwickler interessieren sollte. Guter Stil bedeutet Lesbarkeit und womöglich Kompaktheit, was wiederum zu geringerer Fehleranfälligkeit und besserer Wart- und Erweiterbarkeit führt. Insbesondere Entwickler, die einen Code zum ersten Mal sehen, sollen möglichst leicht Zugang finden. Für die meisten Programmiersprachen existieren Style Guides, die den erwünschten Stil beschreiben. Diese Richtlinien automatisch zu überprüfen, erspart dem Entwickler Zeit und das Nachschlagen des bevorzugten Stils.

Inhalt dieser Arbeit ist ein Stilprüfer, der Programme der funktionallogische Programmiersprache Curry auf Stilbrüche überprüft und Korrekturvorschläge ausgibt. Es wird dazu auf den Quelltext und den sogenannten SpanAST zurückgegriffen, um sowohl unerwünschte Muster zu finden als auch in einzelnen und verschachtelten Konstrukten falsche Formatierung anhand der Positionsangaben abzufangen. Sollten Stilbrüche auftreten, werden Warnungen samt Hinweise ausgegeben. Zudem ist das Werkzeug über eine Konfigurationsdatei einstellbar.

Inhaltsverzeichnis

1. Einführung **1**

1.1. Motivation und Ziel 1

1.2. Existierende Tools 2

1.2.1. Hlint 3

1.2.2. CASC 3

1.3. Überblick der Arbeit 4

2. Grundlagen **5**

2.1. Curry 5

2.1.1. Datentypen 5

2.1.2. Typklassen 7

2.1.3. Pattern Matching 9

2.1.4. Einrückungen 10

2.2. SpanAST 11

2.2.1. Abstrakte Syntax Baum 11

3. Stilrichtlinien **15**

3.1. Curry Style Guide 15

3.2. Stilvorgaben im Tool 16

3.3. Kategorien und Anforderungen von Checks 16

3.3.1. Quellcode Checks 17

3.3.2. SpanAST Checks 17

3.4. Beispiel 19

4. Implementierung **21**

4.1. Infrastruktur und Idee 21

4.1.1. Struktur 21

4.1.2. Check State Monad 23

4.1.3. checkSrc 24

4.1.4. checkAST 24
4.2. Beispiel und Erweiterbarkeit 29
4.2.1. Beispiel If-Then-Else 30
4.2.2. Einbinden der Checks in `AST.curry` 32
4.2.3. Konfiguration 34
4.3. Benutzung 35
4.3.1. Eingabe 36
4.3.2. Ausgabe 37

5. Abschluss **39**
5.1. Zusammenfassung 39
5.2. Ausblick 39
5.2.1. Weitere Stilrichtlinien 40
5.2.2. Erweiterung der Konfiguration 40
5.2.3. Ausgabe 41
5.2.4. Autokorrektur 41

A. Auszüge **43**
A.1. Vollständiger Quelltext für `IfThenElse`.curry 43
A.2. Beispielhafte Ausgabe 46

B. Konfiguration **53**

C. Benutzeranleitung **55**

D. Modulstruktur **57**

1. Einführung

Für Programmiersprachen gibt es neben Syntax und Semantik, die formal definiert sind, noch die Frage vom guten Stil. Für die meisten Programmiersprachen existieren sogenannte Style Guides, die Richtlinien und Konventionen definieren, welche guten Stil fördern sollen. Während den Übersetzter nur korrekte Syntax interessiert, ist die Lesbarkeit von Programmen für den menschlichen Entwickler ein wichtiges Thema. Denn ohne Einhaltung von gutem Programmierstil, läuft das Programm Gefahr, fehleranfälliger und unübersichtlich zu werden. Das hindert nicht nur neue Entwickler daran, gut Zugang zu finden, aber schadet auch der Wartbarkeit und Erweiterbarkeit.

1.1. Motivation und Ziel

Um diese Punkte deutlich zu machen, betrachten wir folgendes Curry-Programm:

```
funcFirst x y            = if (x == True) then
      (func_second  y)  else
                          (    y )

func_second x = case x of
  "" -> "Silence..."
               x        -> x
                  ++ " Or so she says."
```

Dieses Programm ist für den menschlichen Leser zwar sehr schwierig zu verstehen, doch es ist syntaktisch korrekt und kompilierbar. Es existieren weder Formatierung noch Kommentare oder Signaturen. Zudem gibt es unnötige Strukturen wie überflüssige Klammern oder den Abgleich des booleschen Wertes `x` mit `True` in `funcFirst`. Es werden sowohl Camel Case (Wörter in Bezeichnern fangen mit großen Buchstaben

an) als auch Snake Case (Wörter werden in Bezeichnern mit Unterstrichen getrennt) verwendet. Werden Stilrichtlinien richtig umgesetzt, kann es ganz anders aussehen:

```
-- if first parameter is True, modify the String y with funcSecond
funcFirst :: Bool -> String -> String
funcFirst x y = if x
                  then funcSecond y
                  else y

-- adds " Or so she says" to String, if String is not emtpy
-- if the String is empty, return "Silence..."
funcSecond :: String -> String
funcSecond x = case x of
                 ""         -> "Silence..."
                 x          -> x ++ " Or so she says."
```

Guter Stil zielt also auf die Sauberkeit und Lesbarkeit von Programmen für den Entwickler ab. Diese Aspekte sind essentiell, um Wartbarkeit und Erweiterbarkeit zu gewährleisten. Auch die Fehleranfälligkeit von Seiten des Programmierers kann verringert werden. Darum ist es Ziel dieser Arbeit einen Stilprüfer für die Sprache Curry zu entwickeln, welches automatisiert auf Stilrichtlinien überprüft und den Nutzer warnt, falls gegen diese verstoßen wurde. Das Tool soll zudem einfach erweiterbar sein, um nicht nur alle Stilrichtlinien im jetzigen Curry Style Guide [4] umsetzbar zu machen, sondern auch, um im Falle der Erweiterung oder Änderung von diesem – zum Beispiel durch eine neue Curry-Version – leicht verwendbare Schnittstellen bereitzustellen. Dem Nutzer sollen Optionen durch eine Konfigurationsdatei zur Verfügung stehen, über die das Tool auf seine Bedürfnissen angepasst werden kann.

1.2. Existierende Tools

Es werden in den folgenden Abschnitten zwei bestehende Tools vorgestellt werden, an deren Vorbild sich diese Arbeit orientiert.

1.2.1. Hlint

Wie wir später noch beschreiben werden, ist Curry der funktionalen Programmiersprache Haskell syntaktisch sehr ähnlich. Hlint von Neil Mitchell[1] ist ein Tool, das Vorschläge zur Verbesserung von Haskell Quelltext macht. Es werden `.hs`-Dateien eingeben und der Nutzer erhält dann Warnungen und Vorschläge, wie das Programm verbessert werden kann. Es wird vor allem auf eine Vielzahl von überflüssigem Code geachtet und das Tool bietet die Möglichkeit, eigene `hints` zu schreiben. Eine Ausgabe kann beispielsweise folgendermaßen aussehen:

```
Warning: Use <$>
Found:
  fmap (map toUpper) getLine
Why not:
  map toUpper <$> getLine
```

1.2.2. CASC

CASC (Curry Automatic Style Checker) [6] ist ein Stilprüfer, der 2016 für Curry-Programmierer entwickelt wurde. Er prüft eine Reihe von Stilrichtlinien und stellt in Ansätzen Autokorrektur zur Verfügung. Auch CASC baut auf dem Curry Style Guide auf. Der Unterschied liegt darin, dass es zum Entwicklungszeitpunkt von CASC die Erweiterung des SpanAST (siehe Kapitel 2.2) noch nicht gab. Aus diesem Grunde baut das Tool auf einen eigenhändig um Positionen erweiterten AST auf und die Stilrichtlinien sind schwieriger zu implementieren. Weiterhin hat sich Curry um Typklassen erweitert, für welche Stilrichtlinien angedacht werden müssen.

Eine Beispielsausgabe von CASC im folgenden:

```
(6,0) Line is longer than 80 characters.
(9,5) Let: Equality signs are not aligned.
(9,9) Let: Keywords let and in are not aligned.
```

[1] N. Mitchell. *HLint Readme*. Available at `https://github.com/ndmitchell/hlint/blob/master/README.md`. 2019.

1.3. Überblick der Arbeit

Der Kern dieser Arbeit ist in drei Kapitel unterteilt. Zunächst werden in Kapitel 2 die für diese Arbeit wichtigsten Aspekte von Curry sowie der Aufbau des SpanAST, der eine essentielle Informationsquelle für unsere Checks darstellt, eingeführt. Im Kapitel 3 betrachten wir den Curry Style Guide sowie die ausgesuchten Stilrichtlinien dieser Arbeit und die Anforderungen ihrer zugehörigen Checks. Ein Beispiel soll verdeutlichen, wie eine Stilrichtlinie konkret aussehen kann. Dann werden in Kapitel 4 als erstes die Struktur des Programmes im Großen und wichtige Teile des Programmes wie die beiden Module `AST` und `Src` erklärt. In diesen werden die einzelnen Check-Gruppen auf die zu prüfende Datei angewandt. Um den Ablauf der Implementierung und das Einbinden eines konkreten Checks nachvollziehen zu können, wird ein solches Vorgehen am Beispiel der Formatierung von If-Then-Else Schritt für Schritt erklärt. Es folgt in Kapitel 5 eine Beschreibung der richtigen Nutzereingabe und der Ausgabe des Tools. Die Arbeit wird dann abgeschlossen mit einer Zusammenfassung und einem Ausblick.

2. Grundlagen

In diesem Kapitel werden zunächst die grundlegenden Konzepte von Curry vorgestellt, die für das weitere Verständnis der Implementierung benötigt werden. Des Weiteren wird auf die Struktur des abstrakten Syntaxbaumes (abstract syntax tree, AST) eingegangen, wonach wir den SpanAST – ein um Positionen erweiterter AST – betrachten, welche unerlässlich für diese Arbeit ist.

2.1. Curry

Curry ist eine funktionallogische Programmiersprache und gehört somit in die Klasse der deklarativen Sprachen. Curry-Programme sind in der Regel kompakter und besser verständlich als vergleichbare imperative Programme und dadurch weniger fehleranfällig. Der Stilprüfer, der Gegenstand dieser Arbeit ist, ist selbst auch in Curry geschrieben. Als eine solche Sprache enthält Curry sowohl funktionale Konzepte wie faule Auswertung und verschachtelte Ausdrücke als auch logische wie Nichtdeterminismus und freie Variablen. Curry erweitert die funktionale Sprache Haskell. Diese Tatsache spiegelt sich in der Syntax von Curry wieder, die große Ähnlichkeit zu Haskell aufweist. Eine ausführliche Beschreibung der Syntax und detaillierte Beispiele lassen sich jeweils im Curry-Report [2] und im Curry-Tutorial [1] nachlesen. In diesem Abschnitt werden einige wichtige und in dieser Arbeit verwendete Aspekte von Curry vorgestellt.

2.1.1. Datentypen

Neue Datentypen können in Curry mithilfe des Schlüsselwortes `data` in folgender Form deklariert werden:

$$\texttt{data T = C}_1\ \tau_{11}\ \ldots\ \tau_{1n_1}\ \texttt{|..|}\ \texttt{C}_k\ \tau_{k1}\ \ldots\tau_{kn_k}$$

Der neu deklarierte Datentyp ist `T` mit k neuen Konstruktoren $\texttt{C}_1$ ... $\texttt{C}_k$. Hierbei hat jedes $\texttt{C}_i$ den Typ

$$\tau_{i1} \texttt{ -> } \cdots \texttt{ -> } \tau_{in_i} \texttt{ -> T}$$

Beispielsweise kann ein binärer Baum für ganze Zahlen in Curry in der Form

```
data BinIntTree = Leaf Int | Node BinIntTree Int BinIntTree
```

dargestellt werden.

Ein solcher Baum besteht also entweder aus einem Blatt, an der eine ganze Zahl als Wert steht, oder zwei Teilbäumen. Er ist dann also ein Knoten, an dem auch ein Zahlenwert liegt. Da Bäume abgesehen von ganzen Zahlen allerdings auch für andere Typen verwendet werden, liegt es nahe, dass wir einen Datentyp definieren, der den Typ Baum verallgemeinert. Curry bietet hier Typvariablen an, sodass ein neuer Datentyp folgende Form hat:

$$\texttt{data T } \alpha_1 \ \ldots \ \alpha_m \texttt{ = C}_1 \ \tau_{11} \ \ldots \ \tau_{1n_1} \texttt{ |..| C}_k \ \tau_{k1} \ \ldots \tau_{kn_k}$$

Die Konstruktoren $\texttt{C}_1$ ... $\texttt{C}_k$ haben den Typ

$$\tau_{i1} \texttt{ -> } \cdots \texttt{ -> } \tau_{in_i} \texttt{ -> T } \alpha_1 \ldots \alpha_n$$

Ein Typ τ_{ij} ist hierbei aus Typkonstruktoren und den Typvariablen $\alpha_1 \ldots \alpha_n$ aufgebaut. Ein allgemeiner Binärbaum kann dann mit

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)
```

deklariert werden. Dann kann der Binärbaum aus dem obigen Beispiel einfach folgendermaßen definiert werden:

```
type BinIntTree = BinTree Int
```

Mit `type` wird in Curry ein Typsynonym angegeben, also ist `BinaryIntTree` äquivalent zu `BinaryTree Int` und kann austauschbar verwendet werden.

```
class (S₁ α, ... , Sₘ α) => C α where
  f₁ :: K₁ => τ₁₁ -> ··· -> τ₁ₖ₁
  ⋮
  fₙ :: Kₙ => τₙ₁ -> ··· -> τₙₖₙ

  fₗ ... (Implementierung)
  ⋮
  f_g ... (Implementierung)
```

Listing 1: Allgemeine Form einer Typklasse, mit $m, n, k \geq 0$ und $\{l, \ldots, g\} \subseteq [1, n]$.

2.1.2. Typklassen

Seit der Version 2.0.0 können in Curry Typklassen in der gleichen Art wie Haskell erstellt werden. Typklassen erlauben das Überladen von Funktionen. Dies bedeutet, dass auf alle Typen, die Instanzen der Klasse bilden, die Klassenmethoden angewandt werden können. Weiterhin können auf diesen Klassenmethoden aufbauend weitere Funktionen überladen werden, ohne dass für diese Instanzen angegeben werden müssen, indem die Implementierung in der Klasse selbst angegeben wird. Sollten wir eine Typklasse als Kontext für Typen von weiteren Funktionen angeben, sind diese auch überladbar. Die allgemeine Form einer Typklasse ist in Listing 1 aufgeführt. Der zu definierenden Klassenname wird mit `C` bezeichnet und die Typvariable mit α. Diese heißt auch Klassenvariable und muss im Typ von jeder Funktion auftauchen. Jedes $\mathtt{S}_i$ ist ein weiterer Klassenname, diese sind Kontexte für `C`. Sie sind Superklassen von `C`, es dürfen aber keine Zyklen in der Hierarchie entstehen. Alle $\mathtt{f}_i$ sind Funktionen der Klasse und haben jeweils die Kontexte `C` α und `K`, wobei `C` α implizit gilt. Diese Kontexte dürfen die Klasse selbst allerdings nicht weiter einschränken. Zu den Funktionen gehören zusätzlich die Typen τ_{i1} `->` $\cdots$ `->` τ_{ik_i}, in denen die Typvariable mindestens einmal auftauchen muss. Die Implementierung der Funktionen kann in der Klasse selbst angeben werden. Eine Instanz der Klasse wird dann folgendermaßen deklariert:

```
instance (C₁ αₐ ··· Cₛ α_b) => C (T α_c ··· α_d) where
  fₑ ... (Implementierung)
  ⋮
  f_f ... (Implementierung)
```

mit $\mathtt{C}_i$ Klassennamen und $\{\alpha_a, \cdots, \alpha_b\} \subseteq \{\alpha_c, \cdots, \alpha_d\}$. Jedes `fi` ist wiederum eine

Funktion und $\{e, \cdots, f\} \subseteq [1, n]$. In dieser Form kann dann die Klasse `Eq` beispielsweise definiert werden, die die Vergleichsmethode `(==)` überlädt:

```
class Eq a where
    (==) :: a -> a -> Bool
```

Für den Typ `Bool` kann die Instanz dann so aussehen:

```
instance Eq Bool where
  (==) True True     = True
  (==) False False   = True
  (==) False True    = False
  (==) True False    = True
```

Wir können nun auch die Funktion

```
elem :: Eq a => a -> [a] -> Bool
elem e (x:xs) = if (==) x e
                  then True
                  else elem e xs
elem _ []      = False
```

überladen, da wir `Eq` als Kontext angeben können. Weiterhin ist es möglich, innerhalb der Typklasse weitere vorimplementierte Methoden anzugeben, die nicht von der Instanz implementiert werden müssen. Falls gewünscht, können diese jedoch überschrieben werden,

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  (/=) x y = not ((==) x y)
```

`(/=)` basiert auf `(==)` und kann somit auf jeden Typ angewandt werden, solange diese `(==)` implementiert. Typklassen können auch mit partiellen Typkonstruktoren anstelle von Typvariablen arbeiten und sind in dem Fall eine Typekonstruktorklasse (Beispiel Listing 2).

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)

instance Functor [] where
  fmap = map
```

Listing 2: Typkonstruktorklasse `Functor`

2.1.3. Pattern Matching

Curry bietet wie alle funktionalen Programmiersprachen Pattern Matching an. Pattern Matching erlaubt die Implementierung einer Funktion als eine Menge von Gleichungen in der Form

$$
\begin{array}{l}
\texttt{f x}_{11} \ \dots \ \texttt{x}_{1n} = \texttt{e}_1 \\
\vdots \\
\texttt{f x}_{k1} \ \dots \ \texttt{x}_{kn} = \texttt{e}_k
\end{array}
$$

wobei $\texttt{e}_i$ der Ausdruck von der jeweiligen Gleichung, $\texttt{x}_{ij}$ Parameter sowie `f` der Name der Funktion sind. Beim Pattern Matching wird nun das Muster der Parameter von links nach rechts ausprobiert und die Gleichungen von oben nach unten abgearbeitet. In Haskell würde dann die erste passende Regel angewandt werden. Da Curry Nichtdeterminismus unterstützt, werden für alle passende Regel die Berechnung durchgeführt, sodass mehrere Ergebnisse möglich sind. Weil Nichtdeterminismus jedoch nicht in dem Tool zur Verwendung kommt, wird hier nicht weiter darauf eingegangen. Mit Pattern Matching kann dann beispielsweise `map` folgendermaßen definiert werden:

```
map _ []      = []
map f (l:ls)  = (f l) : (map f ls)
```

während der Compiler `[]` und `:` als Konstruktoren und somit bestimmte Muster erkennt, können auch Variablen wie `f` angegeben werden, die zwar für das Muster nicht relevant sind, jedoch für den Ausdruck. Falls auch dies nicht der Fall ist, kann ein Platzhalter, ein sogenannter „Wildcard“, `_` eingesetzt werden.

2.1.4. Einrückungen

Einrückungen sind in Curry wesentliche Bestandteile der Syntax und dienen der Strukturierung. Anstelle von geschweiften Klammern zum Zuordnen von Coderümpfen zu Schlüsselwörtern können diese Listen an Ausdrücken und Deklarationen durch eine größere Einrückungstiefe als ihr Schlüsselwort ausgezeichnet werden. Die folgende Funktion

```
f x = h x where { g y = y + 1 ; h z = (g z) * 2 }
```

kann beispielsweise dann auch

```
f x = h x
  where g y = y + 1
        h z = (g z) * 2
```

geschrieben werden. Dadurch verringert sich die Verbosität des Codes und Lesbarkeit wird gefördert. Syntaktisch sind Einrückungen an bestimmte Regeln gebunden, beispielsweise muss in den meisten Konstrukten der Rumpf ausgerichtet sein, doch wie tief die Einrückung ist oder ob Gleichheitszeichen in Funktionsgleichungen ausgerichtet sind, ist nicht entscheidend. Weiterhin werden viele weitere Strukturierungen – zum Beispiel wo Umbrüche stattfinden – nicht vom Compiler überprüft, da diese nicht für die Semantik relevant sind. Dadurch ist folgender Code syntaktisch korrekt:

```
f x y (z:zs)    = return z
f x y [] = do
  if x then
      return (y*2) else return y
```

Durch das Definieren und Einhalten von Richtlinien für solche Fälle kann die Sauberkeit und Lesbarkeit des Codes verbessert werden. Darum handelt es sich bei einem Großteil der Checks im Stilprüfer um Überprüfungen für Einrückungen und Formatierungen. Dann kann das obige Beispiel wie folgt aussehen:

```
f x y (z:zs)    = return z
f x y []        = do
  if x then
         return (y*2)
       else
         return y
```

```
func1 x = if x then 1 else 0
```

Listing 3: Programm example.curry

2.2. SpanAST

Der abstrakte Syntaxbaum ist eine wichtige Information, die wir zur Formatierungsüberprüfung und Erkennung unerwünschter Muster brauchen. Im Folgenden soll kurz der Baum selbst vorgestellt werden, sowie auf die Anwendung im Tool eingegangen werden.

2.2.1. Abstrakte Syntax Baum

Der abstrakte Syntax Baum (abstract syntax tree, AST) ist eine Darstellung des Programmes in Form eines Baumes. Wie der Name schon andeutet, wird der syntaktische Aufbau eines Programmes repräsentiert. Der Abstract Syntax Tree wird im Frontend-Schritt des Übersetzers erstellt und anschließend analysiert, um im Verlauf des Kompilierens darauf zurückzugreifen. Im Baum finden sich dann keine Klammern, Kommentare und ähnliche, für die Syntax irrelevante, Zeichen und Informationen. Verschiedene Konstrukte, wie Ausdrücke, Deklarationen und Gleichungen bestehen selbst aus weiteren Konstrukten, die die Verschachtelungen in einem Programm widerspiegelt. Diese Unterkonstrukte sind die Kindknoten in der Baumdarstellung. Für das Programm example.curry (Listing 3) ergibt sich die in Listing 4) dargestellte Repräsentation.

Ein `Module` besteht aus seinem Namen, weiteren Informationen wie die Importdeklarationen und einer Liste von Top-Level-Deklarationen. In dieser findet sich dann unsere Funktion als eigenes Element mit `func1` als Namen. Eine Funktionsdeklaration besteht aus einer Reihe von Gleichungen (Regeln) `Equation`, die aus einer linken und rechten Seite zusammengesetzt sind. Auf der rechten Seite steht dann als oberstes Konstrukt unser If-Then-Else mit seinen Kindern, die hier nur eine Variable sowie zwei Literale sind. Diese sind Ausdrücke sowie If-Then-Else selbst und dadurch hat der Konstruktor `IfThenELse` für `Expression` folgende Form:

```
IfThenElse (Expression a) (Expression a) (Expression a)
```

Während die Informationen des Baumes essentiell für die meisten Stilüberprüfungen sind, fehlt jedoch eine wesentliche Information: die Positionen der Konstrukte

```
Module
  []
  (ModuleIdent ["example"])
  Nothing
  [ImportDecl (ModuleIdent ["Prelude"]) False Nothing Nothing]
  [FunctionDecl
    ()
    (Ident "func1" 0)
    [Equation
      (FunLhs
        (Ident "func1" 0)
        [VariablePattern
          ()
          (Ident "x" 1)])
      (SimpleRhs
        (IfThenElse
          (Variable
            ()
            (QualIdent Nothing (Ident "x" 1)))
          (Literal () (Int 1))
          (Literal () (Int 0)))
        [])
    ]
  ]
```

Listing 4: abstrakter Syntaxbaum `example.ast`

und Schlüsselwörter. Wir brauchen diese immer, um die Positionen eines eventuellen Stilbruchs angeben zu können. Noch viel wichtiger ist allerdings die Tatsache, dass Formatierungsüberprüfungen die Positionen aller Schlüsselwörter und Rümpfe eines Konstruktes benötigen, um Einrückungen und Ausrichtung abzugleichen. Hier ist die Erweiterung Curry-ast-2.0.0, die im Zuge der Bachelorarbeit von Kai Prott [5] entstanden ist und den abstrakten Syntax Baum um so genannte Spaninformationen erweitert, gut geeignet. Eine Spaninformation hat die Form:

```
data SpanInfo = NoSpanInfo
              | SpanInfo Span [Span]
```

Ein Span enthält entweder Informationen oder sagt nichts aus (`NoSpanInfo`). Hierbei gehört der erste Span zum gesamten Konstrukt. Die Liste an Spans sind Positionsangaben für die Schlüsselwörter des Konstrukts, also hier `if`, `then` und `else`. Diese Liste kann entsprechend auch leer sein. Ein Span selber wird wie folgt definiert:

```
data Span = NoSpan
          | Span Position Position
```

Die Positionen sind die Start- und Endpunkte des zugehörigen syntaktischen Elementes. Eine Position selbst besteht aus Zeilen- und Spaltennummer.

```
data Position = NoPos
              | Position Int Int
```

Um auf das obige Beispiel zurückzukommen, wird der Konstruktor `IfThenElse` von `Expression a` um eine `SpanInfo` erweitert:

```
IfThenElse SpanInfo (Expression a) (Expression a) (Expression a)
```

Der SpanAst von unserem Beispiel sieht entsprechend (abgekürzt) aus wie Listing 5.

Mit diesen Informationen können wir also sehr einfach den SpanAST des zu prüfenden Codes einlesen und mithilfe von Pattern Matching den Baum von der Wurzel aus (`Module`) traversieren und an jedem Knoten per Pattern Matching zum einen entscheiden, welche Checks angewendet werden, und zum anderen wie weiter im Baum abgestiegen wird. Wie dies im Detail abläuft wird in Kapitel 4.1.4 beschrieben.

```
Module
  (SpanInfo ...)
  []
  (ModuleIdent NoSpanInfo ["example"])
  Nothing
  [ImportDecl
    NoSpanInfo (ModuleIdent NoSpanInfo ["Prelude"])
    False Nothing Nothing]
  [FunctionDecl
    (SpanInfo ...)
    ()
    (Ident (SpanInfo ...) "func1" 0)
    [Equation
      (SpanInfo ...)
      (FunLhs
        (SpanInfo ...)
        (Ident (SpanInfo ...) "func1" 0)
        [VariablePattern (SpanInfo ...) () (Ident (SpanInfo ...) "x" 1)])
      (SimpleRhs (SpanInfo ...)
        (IfThenElse
          (SpanInfo
            (Span (Position 1 11) (Position 1 28))
            [ Span (Position 1 11) (Position 1 12)     --if
            , Span (Position 1 16) (Position 1 19)     --then
            , Span (Position 1 23) (Position 1 26)     --else
            ]
          )
          (Variable
            (SpanInfo (Span (Position 1 14) (Position 1 14)) [])
            ()
            (QualIdent (SpanInfo ...) Nothing (Ident (SpanInfo ...) "x" 1)))
          (Literal (SpanInfo (Span (Position 1 21) (Position 1 21)) [])
                   () (Int 1))
          (Literal (SpanInfo (Span (Position 1 28) (Position 1 28)) [])
                   () (Int 0)))
          []
      )
    ]
  ]
```

Listing 5: abgekürzter abstrakter Syntaxbaum mit Span-Erweiterung

3. Stilrichtlinien

Formatierung und Stil sind nicht nur ein wesentlicher Teil der Syntax von Curry, sondern grundsätzlich ein Aspekt der Programmierung, der vor allem den Programmierer interessieren sollte. Guter Stil bedeutet Lesbarkeit und womöglich Kompaktheit (im Falle von Curry besonders, da geschweifte Klammern ausgelassen werden können, siehe Kapitel 2.1.4), was wiederum zu geringerer Fehleranfälligkeit und besserer Wartbarkeit führt. Insbesondere Entwickler, die einen Code zum ersten Mal sehen, sollen möglichst leicht Zugang finden.

3.1. Curry Style Guide

Die Stilrichtlinien dieser Arbeit basieren auf den Vorgaben des Curry Style Guides [4] der Arbeitsgruppe „Programmiersprachen und Übersetzerkonstruktion" der Christian-Albrechts-Universität zu Kiel und erweitern diese um einige Spezialfälle. Dieser ist angelehnt an den Haskell Style Guide von Johann Tibell [7]. Stilrichtlinien werden in fünf Gruppen unterteilt, von denen wir einige nicht in das Tool aufnehmen, wozu im nächsten Abschnitt noch mehr erwähnt wird.

Generelle Formatierung enthält Richtlinien wie maximale Zeilenlänge, dem richtigen Einsatz von Leerzeichen beziehungsweise Leerzeilen, Einrückungsstandard und Ähnliches.

Formatierung einzelner Sprachkonstrukte sind konstruktspezifische Stilhinweise. Das können beispielsweise die Formatierung der Schlüsselwörter in If-Then-Else oder Umbruchsrichtlinien für Datendeklarationen oder Listendefinitionen sein. Diese geben Regel für Ausrichtung und Einrückung für individuelle Konstrukte im Detail an.

Kommentare sollen unter anderem in englischer Sprache geschrieben und jede Top-Level-Definition kommentiert sein.

Namensgebung zielt darauf ab, sinnvolle Namen in Camel Case zu benutzen, die für Top-Level-Definition ausreichend lang und aussagekräftig sein sollen. Für lokale Definitionen hingegen sollen bestimmte Buchstaben(kombinationen) für bestimmte Typen/Arten von Parametern benutzt werden, wie etwa `n` und `m` für `Integer`.

Compiler-Warnungen Weiterhin gibt der Compiler einige Warnungen aus, die bei gutem Code nicht auftreten sollen. Diese sind oft auch Stilbrüche und sollten vermieden werden. Das können beispielsweise die Warnungen sein, die auftreten, wenn Tab-Zeichen benutzt werden.

3.2. Stilvorgaben im Tool

Während es einfach ist Zeilenlänge und Tabs zu überprüfen, sind leere Zeilen und Leerzeichen schwieriger zu überprüfen. Im ersten Fall betrachten wir nur, ob zwischen Top-Level-Deklarationen, mindestens einer vorhanden ist. Überflüssige Leerzeilen sowie Leerzeichen prüfen wir nicht allgemein, zumal der Programmierer aus vielen Gründen mehr Leerzeichen/-zeilen als vorgegeben brauchen wird, um beispielsweise Ausrichtung und Einrückung zu berücksichtigen sowie womöglich Lesbarkeit. Wir betrachten eher Spezialfälle wie Trailing Spaces. Es werden überwiegend Formatierungsrichtlinien einzelner Sprachkonstrukte aufgenommen. Weiterhin überprüfen wir auch auf zusammengesetzte Konstrukte sowie die Einrückung vom überliegenden Block ausgehend. Ob Top-Level-Deklarationen kommentiert sind oder diese richtig formatiert sind betrachten wir im Rahmen dieser Arbeit nicht. Weil es sehr schwierig und subjektiv ist, „sinnvolle“ Namen zu prüfen und Camel Case zu erkennen, ist es weitgehend unmöglich, diese Stilrichtlinien automatisch zu prüfen. Wir müssten dafür über das Wissen verfügen, was ein „Wort“ ist. Möglich wäre es die Namen darauf zu prüfen, ob nicht Snake Case vorliegt oder Parameternamen klein anfangen. Weiterhin wollen wir noch Fälle aufnehmen, wie das Eliminieren von überflüssigen Code wie `(b==True)` oder `if b then True else False`. Im beiden Fällen würde `b` schon ausreichen.

3.3. Kategorien und Anforderungen von Checks

Wir können die Stilrichtlinien in dieser Arbeit grundsätzlich in zwei Phasen unterteilen. Einige können direkt an dem Quellcode überprüft werden, die meisten benötigen aller-

dings sowohl die Information der syntaktischen Konstrukte als auch die Positionen. Im ersten Fall betrachten wir den Quellcode als eine Liste von Tupeln, in dem Zeilennummer und Inhalt der Zeile als String aufgenommen sind. Im zweiten Fall lesen wir den SpanAST ein und finden per Pattern Matching Konstrukte, auf die wir die entsprechenden Checks anwenden. Hier unterscheiden wir noch grob zwischen Formatierung und „schlechtem“ (überflüssigem) Code. Es muss nur für die Formatierung der linke Rand für die Einrückung ausgerechnet werden, da dieser für andere Checks nicht relevant ist. Daher benötigt ein Formatierungscheck eigentlich etwas mehr Informationen als ein standard AST-Check.

3.3.1. Quellcode Checks

Nachfolgend werden die im Zuge dieser Arbeit implementierten Checks auf dem Quellcode aufgeführt.

Zeilenlänge Die maximale Zeichenanzahl einer Zeile soll auf 80 Zeichen beschränkt sein. Dabei soll dieser Wert konfigurierbar sein; bei breiteren Monitoren wären zum Beispiel auch 120 Zeichen akzeptabel.

Tabs Tabulatorzeichen sollen nicht verwendet werden, Einrückungen sollen mittels Leerzeichen erfolgen. Viele Editoren rücken standardmäßig mit Tabs ein, weshalb für dieses Whitespace-Zeichen ein eigener Check existiert.

Trailing Spaces Leerzeichen am Ende einer Zeile sind überflüssig.

Whitespace Abgesehen von Tabs (abgedeckt durch den obigen Check) und Leerzeichen sollen im Code keine weiteren Whitespace-Zeichen auftauchen.

3.3.2. SpanAST Checks

Die meisten der implementierten Checks benötigen den SpanAST. Hier lassen sich komplexere Strukturen einfacher überprüfen als rein mit dem Quellcode.

Im Folgenden sind die Checks aufgelistet, welche die Formatierung des Codes betrachten:

IfThenElse Sofern es passt, können alle Schlüsselwörter in einer Zeile geschrieben werden. Andernfalls gibt es zwei Fälle: (1) `else` muss in eine eigene Zeile umgebrochen und an dem `then` ausgerichtet werden oder (2) sowohl `then` als auch `else` werden umgebrochen und beide vom linken Rand oder dem `if` um 2 Leerzeichen eingerückt.

Do Ausdrücke in einem `do`-Block sollen zueinander ausgerichtet sein. Der erste Ausdruck darf dabei direkt hinter dem `do` stehen oder umgebrochen und dann 2 Zeichen vom linken Rand oder dem `do` eingerückt werden

Let Für `let` und `in` gelten: Rümpfe sollen ausgerichtet sein; fange hinter dem Schlüsselwort an oder brich um und rücke um 2 Zeichen vom linken Rand beziehungsweise dem Schlüsselwort ein. Dabei sollen `let` und `in` immer ausgerichtet sein, es sei denn das Konstrukt passt in eine Zeile.

Case Das Konstrukt `case e of` soll sich in einer Zeile befinden, alle Alternativen sollen ausgerichtet sein und entweder hinter `case e of` anfangen oder um zwei Zeichen vom linken Rand oder dem `case` eingerückt sein.

Guard Richte `|` und `=` aus, rücke um 2 vom linken Rand beziehungsweise der Funktion ein.

Signatur Jede Funktion sollte eine Signatur haben und diese sollte direkt über der Funktionsdefinition stehen.

`=` und `|` Falls eine Funktionsdefinition mehrere Muster hat, richte sowohl `=` als auch `|` aus, sofern Guards in der Definition auftauchen.

Leerzeilen Zwischen Top-Level-Deklarationen soll mindestens eine Leerzeile stehen und am Ende der Datei keine überflüssigen.

Auch unerwünschte Muster lassen sich anhand des SpanAST überprüfen:

`== True` Da `boolV == True` äquivalent zu `boolV` ist, kann der Ausdruck zu `boolV` vereinfacht werden. Analog gilt dies für `(==) False b` und `not b`.

`then True else False` Der Ausdruck `if boolV then True else False` ist äquivalent zu `boolV`. Analog kann der Check für `if boolV then False else True` mit `not boolV` ausgedrückt werden.

3.4. Beispiel

Um einen besseren Einblick in eine konkrete Stilrichtlinie zu gewinnen, sehen wir uns das Beispiel If-Then-Else an. Es werden im Style Guide drei Formatierungen akzeptiert:

```
f x = if x then 0 else 42

f x = if x
        then 0
        else 42

f x = if x then 0
           else 42
```

Passt das gesamte If-Then-Else Konstrukt in eine Zeile, dann darf es so stehen bleiben. Ansonsten muss entweder beim `else` umgebrochen werden oder sowohl beim `then` als auch beim `else`. In beiden Fällen müssen die Schlüsselwörter `then` und `else` ausgerichtet sein. Dies klingt zunächst simpel, aber es gibt dadurch eine Reihe an Formatierungen, die vermieden werden müssen. Beispielsweise gibt es folgende Fälle:

```
f = if
      ... then
      ... else ...

f = if ... then ... else
                      ...


f = if
    ...
      then ...
      else ...
```

Erlaubt wären dafür zum Beispiel:

```
f = if
      ...
```

```
        then
          ...
        else
          ...

    f = if ... then
                 ...
               else
                 ...
```

Diese Fälle werden nicht spezifiziert. Man kann erkennen, dass es nicht ausreicht mit den Schlüsselwortpositionen die Formatierung zu prüfen. Es werden zusätzlich die exakten Positionen der nachfolgenden Ausdrücke gebraucht. Andersherum ist es für die Ausdrücke, die in einem If-Then-Else verschachtelt sind, wichtig, wie das umliegende Konstrukt formatiert ist.

```
    f = if ...
          then ...
          else do
            ...
            ...
            ...

    f = if ... then ... else do
      ...
      ...
      ...
```

Es ist meist der Fall, dass sich die Einrückung vom Rumpf eines Konstrukts an dem linken Rand des umliegenden Blocks ausrichten darf. Im ersten Fall ist dies das `else`, da dort umgebrochen wurde. Der zweite Fall sollte vermieden werden, da es gegen die Richtlinie für If-Then-Else verstößt. Dies kann jedoch vorkommen, da der Check in der Konfiguration ausgeschaltet ein kann. Dann ist der linke Rand 1 bei dem Funktionsanfang, da nie umgebrochen wurde. Es müssen also mehr Situationen bedacht werden, als intuitiv aus dem Style Guide zu entnehmen. Mehr Details werden in Kapitel 4.1.4 erklärt, da das Berechnen dieser linken Ränder dort stattfindet.

4. Implementierung

Ziel dieser Arbeit ist die Programmierung eines Werkzeuges, das Curry Quellprogramme auf die Einhaltung von Stilrichtlinien überprüft und entsprechende Fehlermeldung und Korrekturvorschläge ausgibt. Die Liste dieser Stilrichtlinien soll konfigurierbar und leicht erweiterbar sein. In diesem Kapitel sollen der Entwurf und die Grundidee der Implementierung vorgestellt werden. Anschließend soll Anhand eines Beispieles die konkrete Umsetzung einer Überprüfung im Code vorgestellt werden, sodass sich eine Erweiterung an diesem Abschnitt orientieren kann. Zum Schluss wird auf die Handhabung des Tools sowie die Form der Eingabe und die Struktur des Ausgabe eingegangen.

4.1. Infrastruktur und Idee

In diesem Abschnitt wird auf die grundlegende Struktur und den Entwurf des Programmes eingegangen. Vor allem die CheckState-Monade, in der wir überwiegend arbeiten, sowie die Phasen CheckAST und CheckSrc werden näher betrachtet, da diese die eigentliche Prüfung ausmachen.

4.1.1. Struktur

Das Programm arbeitet mit der State-Monade, um das Durchreichen der Konfiguration und das Aufsammeln der Stilbruchnachrichten zu verstecken und automatisch weiterzugeben. In Abb. 4.1 kann eine graphische Darstellung der Infrastruktur im großen gesehen werden. Nach dem Einlesen von Konfiguration und zu prüfende Datei in der IO-Monade in dem Hauptmodul arbeiten wir die Checks in der CheckState-Monade ab. Die Datei, falls vorhanden, wird sowohl als Quellcode als auch als SpanAST geholt und in das Check-Modul gereicht. Hierbei werden Quellcode und SpanAST nur eingelesen, wenn es mindestens einen Check gibt, der das jeweilige Format benötigt. Andernfalls

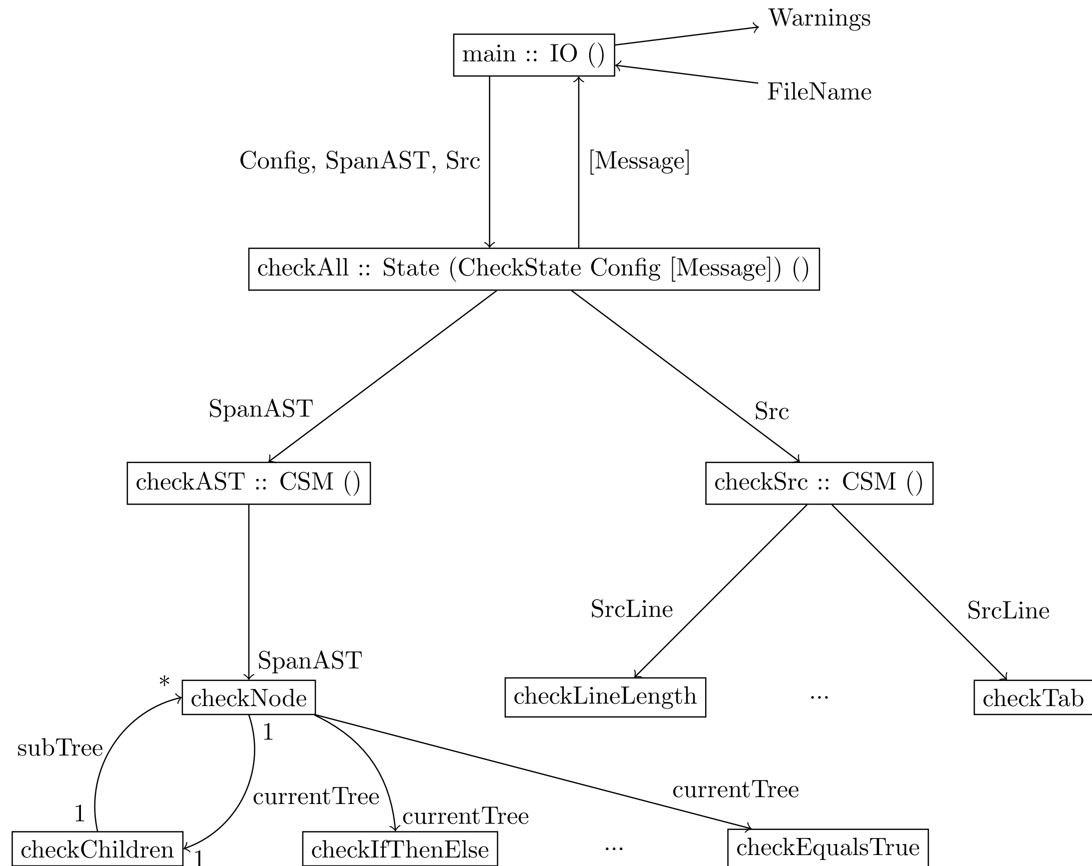


Abbildung 4.1.: Darstellung der Abhängigkeiten und Struktur des Programmes

wird durch das Traversieren des SpanAST dieser trotz der faulen Auswertung von Curry komplett eingelesen. Im Check werden dann die beiden großen Prüfphasen CheckSrc und CheckAST verbunden. Während der Quellcode Zeile für Zeile durchgeprüft wird, arbeiten wir auf dem SpanAST rekursiv, um an jedem Knoten Checks durchzuführen und die Kinderknoten aufzurufen. Die Rückgabe ist die Liste an Nachrichten (Messages), die in der Monade aufgesammelt wurden, die dann gefiltert, sortiert und als String formatiert an die IO Monade zurückgeliefert wird. Die Funktionen, die von `checkAST` und `checkSrc` verwendet werden, arbeiten auch in der Check-State-Monade `CSM ()`. Auf diese wird in Kapitel 4.1.2 eingegangen. Der detailliertere Aufbau von den beiden Phasen finden sich auch in den nachfolgenden Unterabschnitten.

4.1.2. Check State Monad

Es wird die State-Monade verwendet, um die Nachrichten, die die Checks ausgeben, zu sammeln. Weiterhin müssen wir Informationen – beispielsweise die Konfiguration oder Dateinamen – weiterreichen und können dies in der State versteckt und somit einfach implementieren. Die CheckState-Monade (`CSM ()`) besitzt einen CheckState der folgendermaßen definiert ist:

```
data CheckState = CheckState { fileName :: String
                             , config :: Config
                             , messages :: [Message]
                             }
```

Die Nachrichten bestehen aus einem `Span` zur Positionsangabe, sowie zwei `Doc`-types, die die Warnung und den Verbesserungshinweis beinhalten. Die Konfiguration ist ein Record, der selbst unter anderem die maximale Zeilenlänge und auch weitere Records enthält. Der Record `checklist` ist zum Beispiel ein Teil von `config`, der für jeden Check einen Boolean enthält. Mehr dazu wird im Abschnitt Konfiguration beschrieben. Mit der Funktion

```
report :: Message -> CSM ()
report m = modify $ \cs -> cs { messages = m : messages cs }
```

können wir Nachrichten aufnehmen. Diese Nachrichten werden wie bei einem Stack immer vorne angefügt. Erst nach dem Ausführen der monadischen Aktion wird nach Zeilen- und Spaltennummer sortiert und eventuell gefiltert. Die für die State-Monade vordefinierte Funktion `modify` erwartet eine Methode, die den State angewandt verändert. Hier geben wir sie als anonyme Funktion weiter. Um die Konfiguration zu lesen, benutzen wir

```
getConfig :: CSM Config
getConfig = do s <- get
               return $ config s
```

Die Funktion `get` liefert uns den Zustand als Rückgabewert und aus diesem selektieren wir die Konfiguration. Mithilfe dieser Funktion wird zusätzlich eine Hilfsfunktion erstellt, die die `checklist` zurückgibt:

```
getCheckList :: CSM CheckList
getCheckList = do c <- getConfig
                  return $ checks c
```

4.1.3. checkSrc

Die Hauptfunktion zur Quellcodeüberprüfung `checkSrc` ist im Modul `Check.Src` zu finden. Die Funktion hat folgende Form:

```
checkSrc :: [SrcLine] -> CSM ()
checkSrc src =
  mapM_ (\srcl -> do conf <- getCheckList
                     whenM (selector_1 conf) $ check_1 src
                     ⋮
                     whenM (selector_n conf) $ check_n src
        ) src
```

Wir erhalten den Quellcode als eine Liste von Tupeln der Form `(Int,String)`, wobei es sich um Paare von Zeilennummern und dem eigentlichen Code handelt. Wir wenden die Checks, die jeweils eine Zeile abprüfen, mit `mapM_` auf jede Zeile an. Diese sind in einer anonymen Funktion monadisch gebunden. Es wird aus der Konfiguration die `checklist` entnommen und für jeden Check über den Quellcode, der in der Konfiguration aktiviert ist, der konkrete Check angewandt. Sollte der Check nicht auf `True` gesetzt sein, so erhalten wir für `whenM_` automatisch `return ()` zurück.
Es wird für jeden Stilbruch eine Nachricht aufgegeben. Dies bedeutet für das Beispiel der Prüfung für das Auftauchen von Tabs, dass es eine Nachricht für jeden gefundenen Tab gibt.

4.1.4. checkAST

Die Funktion `checkASt` ruft die Checkfunktion `checkNode` am Knoten (`Module`) auf. Diese Checkfunktion, die den Check für den jeweiligen Konstrukttypen aussucht und anwendet sowie die Traversierfunktion `checkChildren` gehören zu der Typkonstruktorklasse `Checkable`:

```haskell
class Checkable c where
  checkNode :: Checks a -> Int -> c a -> CSM ()
  checkChildren :: Int -> Checks a -> c a -> CSM ()
```

Diese beiden Funktionen rufen sich gegenseitig auf, um so den Baum abzusteigen. Dabei sucht `checkNode` und wendet die richtigen Checks über einen Knoten an und reicht diesen nach `checkChildren`. Diese Methode sucht sich mit Pattern Matching die Unterkonstrukte aus und führt für jedes dieser wiederum `checkNode` aus. Zudem wird noch die Einrückungstiefe in jedem Schritt berechnet, worauf später näher eingegangen wird.

Der Record `Checks` enthält für jeden im AST zu prüfenden Konstrukttypen einen Check. Wir benutzen es, um unsere Checks nach diesen Typen zu sortieren und zusammenzufassen. So können sie einfach durch die Traversierfunktion gereicht werden. Der Record sieht dann folgendermaßen aus:

```haskell
checks :: Checks a
checks = Checks (\e i -> do checkConf selector_11 check_11  e i
                            ⋮
                            checkConf selector_1n check_1n e i)
                ⋮
                (\e i -> do checkConf selector_k1 check_k1 e i
                            ⋮
                            checkConf selector_kn check_kn e i)
```

wobei $n, k \geq 0$. Die Hilfsfunktion `checkConf` erhält einen Selektor sowie den Check und die Parameter `e` und `i`. Es wird dann in der Konfiguration nachgeguckt, ob der Check auf wahr gesetzt ist, und falls dies der Fall ist, der Check auf `e` und `i` angewandt. Der Parameter `e` ist das aktuelle Konstrukt und `i` die aktuelle Einrückung beziehungsweise der linke Rand. Mit dieser Struktur können wir nun in der jeweiligen Instanz von Checkable für einen Konstrukttypen in der Funktion `checkNode` den richtigen Check auswählen und anwenden. Anschließend wird das Konstrukt samt `checks` in die Traversierfunktion gegeben. Die Traversierfunktion nimmt für jeden Konstruktor eines Konstrukts die Kinder (Unterkonstrukte) und ruft für diese `checkNode` mit der aktualisierten Einrückungstiefe und `checks` auf (siehe Listing 6).

Die Einrückung `i` wird benötigt, um Fälle wie

```
instance Checkable AST-Datentyp D where
  checkChildren i c e@(Konstruktor_D,1 sI con_11 ... con_1n) =
    do checkNode c (newIndent getLi sI con_11 i) con_11
         ⋮
         checkNode c (newIndent getLi sI con_1n i) con_1n
  ⋮
  checkChildren i c e@(Konstruktor_D,k sI con_k1 ... con_kn) =
    do checkNode c (newIndent getLi sI con_k1 i) con_k1
         ⋮
         checkNode c (newIndent getLi sI con_kn i) con_kn

  checkNode c i e = do (selector c) e i
                       checkChildren i c e
```

Listing 6: Eine Instanz von Checkable für einen Typen aus dem AST: Es wird in `checkChildren` für jeden Konstruktor `checkNode` auf die Kinder angewandt sowie die Einrückung `i` neu berechnet und weitergereicht. In `checkNode` hingegen wird mit dem `selector` die richtigen Checks (gebunden) aus `c` genommen, auf den Knoten angewandt und `checkChildren` aufgerufen.

```
func x = if x
            then return ()
            else do
              a
              b
              c
```

in Betracht zu ziehen (siehe `do`). Grundsätzlich orientiert sich der Rumpf von Konstrukten an ihren Schlüsselwörtern:

```
func x = if x
          then return ()
          else do
                 a
                 b
                 c


func x = if x
          then return ()
```

```
else do a
        b
        c
```

Es ist jedoch möglich vom Rand aus einzurücken. Da dieser je nach Verschachtelung und Formatierung variiert, nutzen wir `newIndent` (siehe Listing 7) zum Neuberechnen der Einrückungstiefe.

Wir übergeben eine Funktion `f`, die aus dem SpanInfo des aktuellen Konstrukts eine Zeilenposition extrahiert. Standardmäßig ist dies einfach die Startzeile des Konstrukts. Falls unser Kind `a` sich auf der gleichen Zeile wie das Überkonstrukt befindet, so ändert sich der Rand nicht (es wird Elternrand benutzt). Andernfalls ist der neue Rand die Spaltennummer des Kindes, da unser Kind umgebrochen sein muss. Nicht zu verwechseln ist, dass dieser Rand bei der Prüfung des Kindes benötigt wird, um im konkreten Check den Rumpf zu überprüfen. Dieser Rand gilt also für die Enkelkinder. Für Spezialfälle sieht es allerdings komplizierter aus.

Wir betrachten dazu den Fall If-Then-Else. Zum Verständnis muss beachtet werden, dass die richtige Positionierung von `if`, `then` und `else` und ihren Ausdrücken – sofern der Check angeschaltet war – vor dem Traversieren der Kinder ausgeführt wurde. Die Ränder, die wir ausrechnen, werden innerhalb der Teilausdrücke verwendet und spielen keine Rolle für `if then else` selbst. Es muss dann unterschieden werden:

1. Das Kind ist der Ausdruck hinter dem `if`. Wir rechnen wie üblich, da `if` immer in der Startzeile des Konstrukts liegt. Der Elternrand ist die, die in `checkChildren` gereicht wurde und wir prüfen, ob das Kind in der Startzeile steht.

2. Das Kind ist der `then`-Ausdruck. Wir vergleichen nicht mit der Startzeile, sondern mit der Zeile in der sich `then` befindet, da wir immer wissen wollen, ob der

```
newIndent :: HasSpanInfo a =>
             (SpanInfo -> Int) -> SpanInfo -> a -> Int -> Int
newIndent f sI a i = if (f sI) == (getLi (getSpanInfo a))
                       then i
                       else getCol (getSpanInfo a)
```

Listing 7: Die Funktion aktualisiert die Einrückungstiefe durch Überprüfung, ob der aktuelle Knoten sich auf der gleichen Zeile wie sein Elter befand

```
func x = if x
|          then return ()
|          else do
|          |  a
|          |  b
|          |  c
1          12
```

Abbildung 4.2.: Beispiel zur Verdeutlichung, wie Ränder berechnet werden

Ausdruck umgebrochen wurde. Dies erreichen wir, indem wir ein Positionsrückgabefunktion reingeben, die die Zeilenposition von `then` findet. Falls sich `then` in der gleichen Zeile wie `if` befindet, dann rechnen wir mit der alten Einrückungstiefe des `if`. Ansonsten steht `then` nicht in der Startzeile zusammen mit `if`. Dann ist der Elternrand nicht der von `if`, sondern die Spaltenposition von `then`.

Tatsächlich wird der Rand nicht gebraucht, falls `if` und `then` in einer Zeile stehen, da zuvor im konkreten Check des Eltern If-Then-Else Richtlinien erzwungen werden, die besagen, dass `if` und `then` nur in eine Zeile gehören, wenn auch der `then`-Ausdruck in der Zeile steht. Dadurch wird der Rand korrekt berechnet, wenn er gebraucht wird. Wo er nicht gebraucht wird, können wir trotzdem standardmäßig rechnen. Falls der Check für `if then else` deaktiviert ist, können wir weiterhin mit dem linken Rand arbeiten, da dieser dann richtig ist, auch wenn die Gesamtformatierung eventuell gegen die Stilrichtlinien verstößt.

3. Das Kind ist der `else`-Ausdruck. Analog zum `then` wird nun mit der Zeile von `else` verglichen. Steht `else` in der gleichen Zeile wie das `if`, benutzen wir den Elternrand von if. Falls dies nicht der Fall, ist werden zudem die Zeilenpositionen von `else` und `then` verglichen. Sind sie auf einer Zeile, nehmen wir die Spaltenposition von `else`. Ansonsten steht `else` in der einer eigenen Spalte, dann nimm als Elternrand die `else` Spalte. Auch hier gilt: `if` und `else` sollen nach Stilrichtlinie nur in eine Zeile, wenn der gesamte `if then else` Ausdruck mit seinen Unterausdrücken in eine Zeile passt. Dieser Check könnte, wie bereits erwähnt, deaktiviert sein, darum muss trotzdem für die ersten beiden Fälle die Einrückung ausgerechnet werden.

Zur Verdeutlichung kommen wir nochmal zum obigen Beispiel zurück (siehe Abb. 4.2). Der Rand von `func` ist 1. Da wir den If-Then-Else Ausdruck nicht umgebrochen haben,

ist der Rand von If-Then-Else weiterhin 1. Würden wir nun die Traversierung auf das `if then else` ausführen, wäre der Elternrand 1 und wir betrachten wir die Kinder:

If Wir könnten das `x`, welches in der gleichen Zeile wie `if` steht, umbrechen und an die Spaltenposition 3 schieben. Dies wäre eigentlich nicht erlaubt, könnte aber passieren, falls If-Then-Else vorher nicht geprüft wurde. Dann wäre der neue Rand 3 für die Unterkonstrukte von `x`, wenn es welche gegeben hätte. Da das `x` eine Variable ist, ist es ein Blatt. Da nicht umgebrochen wurde, ist linke Rand jedoch weiterhin 1.

Then Die Schlüsselwörter `then` und `else` befinden sich in eigenen Zeilen. Wir überprüfen nun, ob `return ()` in der gleichen Zeile wie `then` steht (anstelle mit `if`). Da dies der Fall ist, wird der alte Rand (Elternrand) an den Ausdruck weitergereicht, dieser ist 12, die Spaltenposition von `then`. Auch der Ausdruck von `then` hat keine weiteren Unterausdrücke, sodass wir den Rand eigentlich nicht brauchen.

Else Analog rechnen wir für das `do`. Da `do` in der Zeile von `else` steht, ist der neue Rand 12. Das bedeutet, dass innerhalb unseres `else`-Ausdrucks dieser Rand gilt. Dies ist unser `do {a; b; c}`. Wir wissen also, dass der Rand 12 ist und dadurch ist es erlaubt `a; b; c` von dort aus einzurücken. Sie stehen dann an Position 14.

Für verschiedene Konstrukte müssen verschiedene Fälle in Betracht gezogen und entsprechend in der Traversierungsphase zugeschneidert implementiert werden.

4.2. Beispiel und Erweiterbarkeit

In diesem Abschnitt soll an einem konkreten Implementierungsbeispiel eines Checks der Ablauf des Integrierens eines neuen Checks gezeigt werden. Es wird wieder das Beispiel von If-Then-Else aufgefasst. Abgesehen von der eigentlichen Implementierung muss der Check in der Datei `AST.curry` eingebunden und aufgerufen werden. Eventuell muss die Klasse Checkable neue Typen instanziiert werden und der Record `checks` erweitert werden. Für jeden neuen Check wird dann in der Konfiguration eine neue Einstellung definiert und das Einlesen von dieser aus der Datei ermöglicht. Der Typ `Config` erhält dann auch einen weiteren `Bool`.

```
checkIfThenElse :: Expression a -> Int -> CSM ()
checkIfThenElse e _ =
  case e of
    (IfThenElse sI expr1 expr2 expr3) ->
      checkIfThenElse' sI
                       (getSpanInfo expr1)
                       (getSpanInfo expr2)
                       (getSpanInfo expr3)
    _                               -> return ()
```

Listing 8: Hauptfunktion für den Formatierungscheck für If-Then-Else

4.2.1. Beispiel If-Then-Else

Ein Check auf dem AST beginnt meist mit einer Methode, die den eigentlichen Check aufruft. Dabei werden mithilfe von Pattern Matching nur die Konstrukte ausgewählt, die geprüft werden sollen. In diesem Fall ist es nur das If-Then-Else von allen möglichen `Expression`s. Dadurch können wir in `AST.curry` alle Prüfungen auf `Expression` zusammenfassen und auf jeden Knoten anwenden, der ein `Expression` ist. Es wird zudem auch der linke Rand als `Integer` durchgegeben. Dieser wird jedoch für If-Then-Else nicht gebraucht wird, da die Ausdrücke sich hier nur an den Schlüsselwörtern orientieren dürfen. Wir reichen die SpanInfo `sI` von den Schlüsselwörtern und die der Unterausdrücke weiter (siehe Listing 8). Das Pattern Matching erlaubt uns, die Positionen, die wir benötigen, direkt beim Definieren der Funktion zu spezifizieren. Wir nutzen dann Guards, um verschiedene Situationen abzufangen (Listing 9). Falls sich das gesamte Konstrukt in einer Zeile befindet, wird nicht weiter geprüft. Ist dies nicht der Fall, aber das Konstrukt befindet sich bis zum `else` in einer Zeile, so ist der `else`-Ausdruck umgebrochen. Die ist nicht erlaubt ist und es wird eine Warnung und Hinweis aufgenommen.

Sind `if` und `then` in einer Zeile und `else` in einer anderen (was wir wissen, da die vorige Option nicht eintraf) sowie der `then`-Ausdruck nicht unerlaubterweise in Zeile mit `else`, so ist es die zweite akzeptierte Formatierung. Die Spaltenpositionen von `then` und `else` werden zur weiteren Prüfung in `checkIfThenElseInTwoLines` (für Details siehe Anhang A.1) gegeben. Der Span `sp` wird für die Positionsangabe in der Nachricht gebraucht. Die Hilfsmethode `checkBreakIndent` prüft immer, ob der Ausdruck hinter einem Schlüsselwort richtig eingerückt ist, falls dieser umgebrochen wurde.

```
checkIfThenElse' :: SpanInfo -> SpanInfo ->
                    SpanInfo -> SpanInfo -> CSM ()
checkIfThenElse'
  (SpanInfo
    sp
    [ Span (Position lpi pi) _
    , Span (Position lpt pt) _
    , Span (Position lpe pe) _
    ]
  )
  sI1@(SpanInfo (Span _ (Position li ci)) _)
  sI2@(SpanInfo (Span _ (Position lt ct)) _)
  sI3@(SpanInfo (Span _ (Position le ce)) _)
  |lpi == le                = return ()
  |lpi == lpe                = do report (...)
  |lpi == lpt && lt /= lpe = do checkIfThenElseInTwoLines sp pt pe
                                checkBreakIndent "then" lpt pt sI2
                                checkBreakIndent "else" lpe pe sI3
  |lt == lpe                = do report (...)
  |li == lpt                = do report (...)
  |li /= lpt && lt /= lpe   = do checkIfThenElseInThreeLines sp pi pt pe
                                checkBreakIndent "if" lpi pi sI1
                                checkBreakIndent "then" lpt pt sI2
                                checkBreakIndent "else" lpe pe sI3
```

Listing 9: Abgekürzte Funktion `checkIfThenElse'`, welche verschiedene Formatierungen von If-Then-Else abfängt und verarbeitet

Anderenfalls fängt die Zeile mit dem **`then`**-Ausdruck an und er steht vor dem **`else`**. Dies wird gemeldet. Der nächste Fall tritt ein, wenn die **`if`**-Bedingung vor **`then`** steht (auch nach de Umbrechen), was ebenfalls ein Verstoß ist. Der letzte Fall ist die erlaubte Formatierung, in der alle Schlüsselwörter in einer eigenen Zeile liegen. Auch hier werden die Schlüsselwortpositionen übergeben und für alle Ausdrücke die Einrückung überprüft.

In `checkIfThenElseInTwoLines` und `checkIfThenElseInThreeLines` wird die korrekte Ausrichtung der Schlüsselwörter überprüft und gegebenenfalls Fehler gemeldet.

4.2.2. Einbinden der Checks in `AST.curry`

Nach dem Importieren der Hauptfunktion, erweitern wir `checks` in der anonymen Funktion für `Expressions` um `checkIfThenElse`:

```
import Check.AST.Indent.IfThenElse    (checkIfThenElse)

checks :: Checks a
checks =
  Checks ⋮
         (\e i -> do ⋮
                     checkConf ifThenElse checkIfThenElse e i
                     ⋮

         )
         ⋮
```

Dementsprechend muss der Typ `Checks` um Funktionen erweitert werden, falls der geprüfte Typ neu ist. Wie wir gleich sehen werden, ist dies für die Instanziierung der `Checkable`-Klasse nicht zu vermeiden. Um `checkConf` nutzen zu können, müssen wir die Konfiguration noch erweitern und den neuen Selektor auch hier angeben (siehe Kapitel 4.2.3). Der `checkNode` an einem Baumknoten sieht dann folgendermaßen aus:

```
instance Checkable Expression where
  ⋮
  checkNode c i e = do (expr c) e i
                       checkChildren i c e
```

Bisher wurden alle Konstrukttypen als Checkable instanziiert, die Konstrukttypen als Kinder haben können, die überprüft werden, sodass wir sie richtig traversieren. Falls wir auf einen Typen treffen, der uns noch nicht interessiert, wird `return ()` zurückgegeben statt `checkNode` aufzurufen. Gibt es noch keine `Checkable` Instanz für den Typ, den wir mit dem neuen Check prüfen, muss diese definiert werden. Falls es weiterhin Typen gibt, von denen keine `Checkable` Instanzen existieren, aber unser neue Typ ein Kind sein könnte, müssen hier ebenfalls entsprechende Instanzen angegeben werden, damit sie traversiert werden können. Jeder neu instanziierter Typ muss entsprechend im Record `Checks` als Funktion aufgenommen werden, weil die Klasse diesen Typ verwendet. Um Typen zu instanziieren, die eigentlich nur traversiert werden, gibt es eine Funktion

```
instance Checkable Expression where
  ⋮
  checkChildren i c (IfThenElse sI exp1 exp2 exp3 ) =
    do checkNode c (newIndent getLi sI exp1 i) exp1 -- if expression
       if ((getLi sI) == (getThenLi sI)) -- then expression
         then checkNode c
                        (newIndent getLi sI exp2 i)
                        exp2
         else checkNode c
                        (newIndent getThenLi sI exp2 (getThenCol sI))
                        exp2
       if ((getLi sI) == (getElseLi sI)) -- else expression
         then checkNode c (newIndent getLi sI exp3 i) exp3
         else
           if ((getThenLi sI) == (getElseLi sI))
             then checkNode c
                            (newIndent getThenLi sI exp3 (getThenCol sI))
                            exp3
             else checkNode c
                            (newIndent getElseLi sI exp3 (getElseCol sI))
                            exp3
  ⋮
```

Listing 10: Ausschnitt aus der `Checkable` Instanz für den Typen Expression für If-Then-Else: `exp1` ist der `if`-Ausdruck und die Einrückung wird wie in den meisten Fällen mit dem `if`-Rand berechnet. Für `exp2` und `exp3` müssen erst Umbrüche überprüft werden, sodass der Elternrand immer ein anderer sein kann.

`default`, die immer `return ()` zurückgibt. Diese können wir in `checks` verwenden, um einen Check zu simulieren. Diesen verwenden wir in `checkNode` für Typen, die (noch) nicht tatsächlich geprüft werden.

Wir haben nun die Implementierung der `checkNode` betrachtet. Falls für den Typen, der von dem neuen Check geprüft wird, eine Checkable Instanz gebildet werden muss, muss `checkChildren` auch definiert werden. Wichtig ist es, für jeden Typen zu wissen, welche Kinder weiter zu traversieren sind. Für nicht instanziierte Typen, die wir auch nicht für den neuen Check brauchen, wenden wir nicht `checkNode` an. Dann muss für jede neue Instanz das Berechnen der Einrückungstiefe zugeschnitten werden (siehe Listing 10). Da bereits in Kapitel 4.1.4 mehr zu diesem Problem erklärt worden ist, werden wir in diesem Abschnitt nicht weiter darauf eingehen.

4.2.3. Konfiguration

Die Konfiguration `Config` ist ein Record, der die maximale Zeilenlänge, den Flag für Hinweisausgabe sowie eine `Checklist` beinhaltet. Diese ist selbst ein Record, der aus boolschen Werten für jeden Check besteht, um zu markieren, ob diese geprüft werden sollen. Es wird zu Beginn – noch vor dem Einlesen der zu prüfenden Datei – die Konfigurationsdatei eingelesen und in einer `Config` abgespeichert. Diese Typen und Funktionen müssen wir erweitern, um einen neuen Check aufzunehmen.

Zunächst muss in der `Types.curry` der Typ `Checklist` erweitert werden, welcher Teil von `Config` ist. Wir geben dem Check hier den Selektor `ifThenElse`:

```
data CheckList = CheckList
                { 
                ⋮
                , ifThenElse   :: Bool
                ⋮
                }
```

Weiterhin gibt es zwei Methoden, mit denen wir prüfen, ob mindestens ein AST- beziehungsweise Src-Check auf wahr gesetzt ist. Je nach Typ des Checks müssen wir eine erweitern:

```
anyAST :: Config -> Bool
anyAST con =
  let c = checks con
  in foldr (||) False
      [
      ⋮
      ,ifThenElse  c
      ⋮
      ]
```

Wir verwenden zur Initialisierung der Konfiguration beim Einlesen eine Standardkonfiguration `defaultConfig`, die abgesehen von speziellen Einstellungen wie Zeilenlänge alle Checks auf `True` setzt. Diese muss entsprechend des neuen `Checklist` Typs neu definiert werden. Mit dieser Standardkonfiguration können wir nun aus der Konfigurationsdatei einlesen. Eine Option wird eingefügt, hier `ifThenElse`:

```
⋮
ifThenElse    = 1
⋮
```

Für diese Art der Einstellung gilt: Ist der Wert 0, so wird der Check in der `config` deaktiviert und wird nicht geprüft. In jedem anderen Fall ist der Check angeschaltet aufgrund der Standardeinstellungen. Es wird dabei davon ausgegangen, dass der Nutzer weiß, wie die Konfiguration auszusehen hat, sodass das Einlesen problemlos möglich ist.

Der Ausschnitt der Einlesefunktion für If-Then-Else findet sich in Listing 11. Es wird Zeile für Zeile durchlaufen: beginnt die Zeile mit `"ifThenElse"` und endet auf `"0"`, wird der Schlüssel `ifThenElse` in der `Checklist` auf `False` gesetzt. Falls der Check noch spezifischere Einstellung benötigt, können sie in ähnlicher Form eingelesen werden. Diese werden direkt in `Config` definiert und nicht in der `Checklist`.

4.3. Benutzung

In dem folgenden Abschnitt wird die Handhabung des Tools beschrieben. Es wird vor allem darauf eingegangen, wie die Nutzereingabe formuliert werden soll und wie Ausgabe der Warnungen aussieht. Im Anhang finden sich mehr Informationen zu der Konfigurationsdatei (Anhang B) und eine Benutzeranleitung (Anhang C).

```
readCheckList :: [String] -> CheckList -> CheckList
readCheckList [] checkl
  = checkl
readCheckList (l:ls) checkl
  ⋮
  | isPrefixOf "ifThenElse" l && isSuffixOf "0" l
  = readCheckList ls $ checkl {ifThenElse = False}
  ⋮
  | otherwise
  = readCheckList ls checkl
```

Listing 11: Ausschnitt der Funktion readCheckList, Option für If-Then-Else

4.3.1. Eingabe

Das Tool kann aus der ausführbaren Datei `Main` aufgerufen werden. Um das Programm auszuführen muss

```
cypm exec /path/to/Main <filename>
```

verwendet werden. Hierbei erlaubt `cypm exec` dem Tool beim eventuellen Generieren und Einlesen des Abstract Syntax Trees die Module richtig zuzuordnen. Es folgt der Dateipfad der ausführbaren Datei sowie der Dateipfad des zu prüfenden Currycodes. Zudem wird vorausgesetzt, dass sich der Nutzer im Hauptmodul des Programmes befindet, sodass der AST der jeweiligen Datei korrekt gefunden werden kann. Sollte die Datei in der Form `example.curry` angegeben werden, wird genau diese Datei gesucht. Im Falle `example` wird angenommen, dass dies der Dateiname ist und die Datei wie eine `.curry` behandelt.

Es ist sehr zu empfehlen, die zu prüfende Datei vor der Stilprüfung erfolgreich zu kompilieren. Andernfalls kommt es zu Fehlern, da die AST-Datei für Checks auf dieser fehlerfrei vorliegen muss. Die Konfigurationsdatei sollte sich im gleichen Ordner wie die ausführbare Datei befinden. Es wird unter dem Namen `config` gesucht. Sollte diese nicht vorhanden sein, wird automatisch die Standardeinstellung verwendet. Falls Einstellungen in der Konfigurationsdatei zudem inkorrekt verwendet oder ausgedrückt werden, werden diese ignoriert und für diese Fälle ebenfalls Standardeinstellungen genommen. Der Nutzer kann in der Konfiguration Prüfungen ausstellen:

```
exampleCheck = 0
```

`0` deaktiviert den Check, während sowohl für `1` als auch jede andere Eingabe der Check aktiviert bleibt (da dies die Standardeinstellung ist). Wenn mehr Informationen benötigt werden, gibt es entsprechende Optionen in der `config`:

```
--maximal length allowed for a line
maxLineLength = 80
```

Dabei wird davon ausgegangen, dass mithilfe der Kommentare die Einstellungen intuitiv handzuhaben sind. Das bedeutet in diesem Fall, dass nur ganze und positive Zahlen verwendet werden. Weiterhin gibt es die Möglichkeit, nur Warnungen ausgeben zu lassen:

```
--switch for output of hint
hints = 0
```

Gewöhnlich wird für jeden Stilbruch ein Korrekturvorschlag beziehungsweise die Angabe der erwarteten Formatierung des Stils mitgeliefert.

4.3.2. Ausgabe

Die Ausgabe bei einem gefundenem Stilbruchs wie beispielsweise Listing 12 besteht aus Position, Warnung und Hinweis (Listing 13). Zunächst werden Start- und Endposition des Konstrukts beziehungsweise der Bereich des Stilbruchs ausgegeben. Intern ist dies ein `Span`. Das ist vor allem praktisch, um direkt aus dem SpanAST diese Information in die `Message` zu geben. Es folgen zwei Nachrichten. Die `Warnung` sollte möglichst knapp und generell sein. Der `Hinweis` soll dem Nutzer genauer mitteilen, gegen welche Stilrichtlinie verstoßen wurde und was erlaubte Lösungen sind. Diese Nachrichten sind von dem Typ `Doc` aus dem Packet `wl-pprint-0.0`, welcher uns erlaubt die Texte einfacher zu manipulieren. Wir können die Schlüsselwörter in den Nachrichten, die uns interessieren sowie `Warning` und `Hint` farblich markieren. In `Check.curry` wird aus der ausgeführten State-Monade, die `checkAST` und `checkSrc` abgearbeitet hat, die Liste `Messages` selektiert, per Quicksort nach Position sortiert und in Stringformat für die Ausgabe aufbereitet.

Das Tool gibt beim Einlesen der Datei zunächst aus, ob sich die Datei im Ordner befindet und bricht gegebenenfalls ab. Dann wird versucht, die Konfiguration einzulesen, und wenn nötig die Standardeinstellungen übernommen, worüber die Ausgabe den Nutzer informiert. Danach werden der SpanAST und die Quelldatei geladen. Ist der Code fehlerhaft, gibt das Lesen des SpanAST Fehler aus, da erfolgreich kompiliert werden muss, und das Tool bricht ab. Dies kann vermieden werden, wenn die Datei vorher richtig kompiliert und debugged worden ist. Andernfalls geht es hiernach in die

```
85
86 f1 :: Integer
87 f1 = if True then 1 else
88   2
85
```

Listing 12: Beispiel für ein Stilbruch im If-Then-Else Konstrukt

```
line 87, 6 ; line 88, 3
Warning :
wrong formatting else expression
Hint :
do not break else expression if fitting if-then-else in one line
```

Listing 13: Beispielhafte Ausgabe beim Stilbruch für Listing 12

Stilprüfphase, die am Ende die Liste der Nachrichten ausgibt. Ist das Tool fertig wird die Ausgabe mit

```
Checks done
```

abgeschlossen.

5. Abschluss

In diesem Kapitel wird noch einmal Überblick über die vorgenommenen und erreichten Ziele gewährt. Weiterhin gehen wir auf Erweiterungsmöglichkeiten und Verbesserungen ein.

5.1. Zusammenfassung

Ziel dieser Arbeit war es ein Tool zu entwickeln, das Stilrichtlinien für Curry automatisch prüfen kann. Dieses baut auf dem Curry Style Guide auf und soll Stileinhaltung und leichteres Überprüfen von Curry-Programmen fördern. Für jede Programmiersprache ist es wichtig und zu empfehlen, sinnvolle Formatierungen und Richtlinien zu definieren, die Code für den Menschen lesbarer und strukturierter und damit weniger fehleranfällig und verständlicher zu gestalten. Der Stilprüfer wurde selbst komplett in Curry geschrieben und ist auch zum Überprüfen dieser Sprache gedacht. Die ausgesuchte Reihe an Stilrichlinien (siehe Kapitel 3.3), die die drei großen Gruppen Quelltext, Formatierung und überflüssigen Code abdecken, sind in dem Tool implementiert und aufgenommen. Das Tool ist in der Lage, farbige Ausgaben zu Stilbrüchen auszugeben, die sowohl Warnungen als auch Hinweise beinhalten (siehe Kapitel 4.3.2 oder Anhang A.2). Es können Einstellungen vorgenommen werden, die dem Nutzer erlauben, Prüfungen oder Hinweise an und auszuschalten, sowie Zeilenlängenbegrenzung selbst anzugeben.

5.2. Ausblick

Das Tool ist einsatzfähig und die Struktur komplett. Es sind Schnittstellen zur einfachen Implementierung weiterer Stilrichtlinien vorhanden. Im Folgenden werden einige der interessanteren Erweiterungsmöglichkeiten für das Tool betrachtet und an welchen bereits vorhanden Tools für andere Sprachen sich diese anlehnen können.

5.2.1. Weitere Stilrichtlinien

Aufgrund des Zeitrahmens dieser Arbeit wurden zwar diverse und wichtige Stilrichlinien implementiert, doch natürlich beschränkt sich das Tool nicht auf diese. Anhand der Schritte im Beispiel Kapitel 4.2.1, kann sehr einfach weitere Prüfungen eingefügt werden. Diese könnten – um einige zu nennen – von folgender Art sein:

Curry Styleguide Im Curry Style Guide [4] finden sich zur Formatierung noch weitere Richtlinien für Konstrukte, die in diese Arbeit nicht aufgenommen wurden. Diese Checks sind analog zu implementieren und gehören zweifellos zu den wichtigeren Stilrichlinien. Andere Regeln wie das Nutzen von Camel Case kann, wie im Style Guide Abschnitt (Kapitel 3.1) beschrieben, noch eingebaut werden.

Hlint Im Tool Hlint finden sich wenig Formatierungsprüfungen, dafür aber ein großes Spektrum an „schlechtem“ (überflüssiger) Code. Viele sind nach Curry übertragbar und sollten übernommen werden.

Kommentare Es kann nützlich sein zu prüfen, ob Top-Level-Deklarationen kommentiert sind, wie es im Style Guide vorgeschlagen ist. Zudem könnte versucht werden, diese auf die richtige Syntax von Curry-Doc zu prüfen.

5.2.2. Erweiterung der Konfiguration

Die Konfiguration kann um viele weitere Einstellungen erweitert werden. Beispielsweise könnten eine Option für die farbige Ausgabe oder grundsätzliche Nutzereinstellungen wie ein Debugging-Modus hinzugefügt werden. Zudem wäre es sinnvoll für das Aufrufen des Tools optionale Argumente mitzugeben, die größere Einstellungen in der Konfiguration überschreiben, sodass der Nutzer nicht unbedingt die Konfigurationsdatei ändern muss. Einige Beispiele im Folgenden:

Konfiguration

- Farbige oder Monotone Ausgabe
- Farbschema zur Auswahl stellen
- verschiedene Ausgabeformate
- Ausgabe der einzelne Schritte des Tool ausschalten
- Alle AST/Src Checks togglen
- Debugging Modus

- Zeilenbereiche angeben, die ungeprüft bleiben sollen.

Kommandozeilenargumente

- Konfiguration ignorieren
- Hinweise togglen
- Alle AST/Src Checks togglen
- Manuell einzelne Checks ausstellen

5.2.3. Ausgabe

In Zusammenspiel mit Kapitel 5.2.2 könnten dem Nutzer andere Ausgabeformate zur Verfügung gestellt werden, wenn beispielsweise die Positionen deutlicher hervorgehoben sein sollen und dafür die Schlüsselwörter gar nicht. Wie überschaubar die Ausgaben sind, ist subjektiv. Diese Erweiterung könnte dem Entwickler Möglichkeit geben, eine für ihn angenehmeres Format zu verwenden. Es wäre zudem für den Nutzer hilfreich, die tatsächlichen Codebereiche auszugeben, die vom Stilbruch betroffen sind (am Beispiel des Übersetzers).

5.2.4. Autokorrektur

Autokorrektur ist ein wünschenswertes Ziel, da dem Nutzer Zeit und das Nachgucken wie eine akzeptierte Formatierung aussieht erspart bleibt, wenn das Tool den Code automatisch in den bevorzugten Stil umwandeln könnte. In CASC [6] wurde dies schon ansatzweise implementiert. Es gibt zudem weitere Tools, die keine Stilprüfungen durchführen, sondern für einen eingegebenen Code bestimmte Konstrukte direkt formatiert wieder ausgeben. Es gibt jedoch einiges zu Bedenken, da wir teilweise Änderungen am SpanAST vornehmen müssten. In der Theorie kann aus dem SpanAST der Quelltext dann (modifiziert) rekonstruiert werden. Da jedoch Änderungen am SpanAST bedeuten, dass nach jedem Schritt der Baum neu berechnet werden muss. Das verschiebt eventuell die Struktur des Programms komplett und ist nicht trivial umzusetzen. Die Prüfungen und somit Änderungen am Quelltext dürften dann hinterher erst durchgeführt werden.

Nichtsdestotrotz wäre dies zwar eine große, aber hilfreiche Erweiterung des Stilprüfers.

A. Auszüge

A.1. Vollständiger Quelltext für `IfThenElse.curry`

Dies ist der vollständige Formatierungscheck für If-Then-Else in Curry.

```
module Check.AST.Indent.IfThenElse where

import Types

import Curry.SpanInfo
import Curry.Span
import Curry.Position
import Curry.Types
import Text.Pretty

-- per patternmatching apply actual check only on ifthenelse constructs
checkIfThenElse :: Expression a -> Int -> CSM ()
checkIfThenElse e _ =
  case e of
    (IfThenElse sI expr1 expr2 expr3) -> checkIfThenElse' sI
                                                          (getSpanInfo expr1)
                                                          (getSpanInfo expr2)
                                                          (getSpanInfo expr3)
    _                                 -> return ()

-- check if various situations:
-- all in one line
-- if - else in one line, else expression not
-- if - then in one line, else and else expression in another -> twolines
-- if - then in one line, then expression and else in one
-- if and then not in one line, if expression and then in one
-- if then else in different lines, if and then expressions not right in front
-- if then and else
checkIfThenElse' :: SpanInfo -> SpanInfo -> SpanInfo -> SpanInfo -> CSM ()
checkIfThenElse'
```

```
(SpanInfo
  sp
  [ Span (Position lpi pi) _
  , Span (Position lpt pt) _
  , Span (Position lpe pe) _
  ]
) sI1@(SpanInfo (Span _ (Position li ci)) _)
  sI2@(SpanInfo (Span _ (Position lt ct)) _)
  sI3@(SpanInfo (Span _ (Position le ce)) _)
|lpi == le                = return ()
|lpi == lpe               = do report (Message
                                         sp
                                         ( text "wrong formatting"
                                           <+> colorizeKey "else expression"
                                         )
                                         ( text "do not break"
                                           <+> colorizeKey " else expression"
                                           <+> text "if writing"
                                           <+> colorizeKey "if-then-else"
                                           <+> text "in one line"
                                         ))
|lpi == lpt && lt /= lpe = do checkIfThenElseInTwoLines sp pt pe
                              checkBreakIndent "then" lpt pt sI2
                              checkBreakIndent "else" lpe pe sI3
|lt == lpe                = do report (Message
                                         sp
                                         ( text "wrong formatting"
                                           <+> colorizeKey "else"
                                         )
                                         ( colorizeKey "else"
                                           <+> text " should start in seperate line"
                                         ))
|li == lpt                = do report (Message
                                         sp
                                         ( text "wrong formatting"
                                           <+> colorizeKey "then"
                                         )
                                         ( colorizeKey "then"
                                           <+> text " should start in seperate line"
                                         ))
|li /= lpt && lt /= lpe   = do checkIfThenElseInThreeLines sp pi pt pe
                              checkBreakIndent "if" lpi pi sI1
                              checkBreakIndent "then" lpt pt sI2
                              checkBreakIndent "else" lpe pe sI3
```

```
  -- |otherwise

-- then and else should alway be aligned
checkIfThenElseInTwoLines :: Span -> Int -> Int -> CSM ()
checkIfThenElseInTwoLines sp pt pe =
  unlessM (pt == pe) (report (Message
                                sp
                                ( colorizeKey "then"
                                  <+> text "and"
                                  <+> colorizeKey "else"
                                  <+> text "wrong alignement"
                                  )
                                ( text "align"
                                  <+> colorizeKey "then"
                                  <+> text "and"
                                  <+> colorizeKey "else"
                                  )
                              )
                      )

-- see above, but also, if then starts in another line, indent by 2
checkIfThenElseInThreeLines :: Span -> Int -> Int -> Int -> CSM ()
checkIfThenElseInThreeLines sp pi pt pe =
  if pt == pe
    then unlessM (pe-pi == 2) (report (Message
                                        sp
                                        ( colorizeKey "then"
                                          <+> text "and"
                                          <+> colorizeKey "else"
                                          <+> text "not properly indented"
                                        )
                                        ( text "indent by 2 from"
                                          <+> colorizeKey "if"
                                        )
                                      )
                              )
    else report (Message
                  sp
                  ( colorizeKey "then"
                    <+> text "and"
                    <+> colorizeKey "else"
                    <+> text "not aligned"
                    )
                    ( text "align"
```

```
                        <+> colorizeKey "then"
                        <+> text "and"
                        <+> colorizeKey "else"
                        )
                  )

-- gets keyname as string, keypositions and the expression to check
-- if the expression is in next line, check indent
checkBreakIndent :: String -> Int -> Int -> SpanInfo -> CSM ()
checkBreakIndent s l c sI = do
  unlessM (l == (getLi sI))
          (unlessM ((getCol sI)==(c+2))
                   (report (Message
                             (getSpan sI)
                             ( colorizeKey "expression"
                               <+> text "wrongly indented"
                             )
                             ( colorizeKey(s ++ " expression")
                               <+> text "should be indented by 2 from"
                               <+> colorizeKey s
                             )
                           )
                   )
          )
```

A.2. Beispielhafte Ausgabe

Es ergibt sich für folgende Testdatei, die möglichst viele akzeptierte Formate sowie Formatierungsfehler für If-Then-Else abdecken soll, die darauffolgende Ausgabe. Hierbei sind nur Formatierungschecks angeschaltet (es wird also ignoriert, ob jede Top-Level-Deklaration eine Signatur besitzt).

ExampleTests.curry

```
-- Single Funktions with if then else for outermost expression --

func1 x = if x then 1 else 0

-- error : then else alignment
func2 x = if x
           then 1
```

```
            else 0

func3 x = if x then 1
               else 0

func4 x =
  if x then 1
       else 0

-- error: formatting else
func5 x = if x
  then 1 else 0

func6 x = if x
            then
              1
            else
              0

-- error: else expr no break
funcIf x =
  if x then 1 else
    0

-- error: indentation then expr
funcIf x =
  if x then
    10000001010101
       else 42

funcIf x =
  if x then
         10000001010101
       else
         42

funcIf x =
  if
    x
    then 1
    else
      0

-- error: start else in own line
```

```
ff = if
       True then 1              else 2

-- error: start else in own line
ff = if True
        then 1 else 2

-- error: start then in own lines
ff = if
       True
       then
        1 else 2

ff = if True then
               1
             else
               2

-- error: fit in one line
ff = if True then 1 else
  2

-- error: start else in own line
ff = if
       True then
             1 else 2


-- if then else expression nested in other constructs --

-- where

func6 x = f x
  where
    f x = if x then 1
               else 0

-- error: indentation then expression, else expression
f x = if x
        then do
          return ()
          return ()
        else do if
                  x
```

```
                  then
                     return ()
                  else
                   return ()


-- if then else expressions with nested childconstructs --

-- do

-- error: formatting
funcIf x =
  if x then return 43 else do
    return 42

funcIf x =
  if x then return 43
       else do
         return 42

-- case

function4 x y = if x
                  then
                    case y of
                      1 -> 0
                      0 -> 1
                  else 2

function5 x y = if x
                  then case y of
                    1 -> 0
                    0 -> 1
                  else 2

--error: indentation case
function4 x y = if x
                  then
                      case y of
                        1 -> 0
                        0 -> 1
                  else 2
```

Ausgabe des Stilprüfers für ExampleTests.curry

```
line 6, 11 ; line 8, 18
Warning :
then and else not aligned
Hint :
align then and else

line 18, 11 ; line 19, 15
Warning :
wrong formatting else
Hint :
else should start in own line

line 29, 3 ; line 30, 5
Warning :
wrong formatting else expression
Hint :
do not break else expression if fitting if-then-else in one line

line 35, 5 ; line 35, 18
Warning :
expression wrongly indented
Hint :
then expression should be indented by 2 from then

line 53, 6 ; line 54, 37
Warning :
wrong formatting else
Hint :
else should start in own line

line 57, 6 ; line 58, 21
Warning :
wrong formatting else
Hint :
else should start in own line
```

```
line 61, 6 ; line 64, 16
Warning :
wrong formatting else
Hint :
else should start in own line

line 72, 6 ; line 73, 3
Warning :
wrong formatting else expression
Hint :
do not break else expression if fitting if-then-else in one line

line 76, 6 ; line 78, 22
Warning :
wrong formatting else
Hint :
else should start in own line

line 98, 22 ; line 98, 30
Warning :
expression wrongly indented
Hint :
then expression should be indented by 2 from then

line 100, 20 ; line 100, 28
Warning :
expression wrongly indented
Hint :
else expression should be indented by 2 from else

line 109, 3 ; line 110, 13
Warning :
wrong formatting else expression
Hint :
do not break else expression if fitting if-then-else in one line
```

```
line 135, 23 ; line 137, 30
Warning :
expression wrongly indented
Hint :
then expression should be indented by 2 from then

Checks done
```

B. Konfiguration

```
-- configuration for stylecheckerTool

-- maximal length allowed for a line
maxLineLength = 80

-- switch for hint output
hints = 1

-- src checks
lineLength                = 1
tabs                  = 1
trailingSpaces = 1

-- indent and alignment
ifThenElse     = 1
case           = 1
do             = 1
let            = 1
guard          = 1
functionRhs    = 1

-- top level declarations
signatures     = 1
blankLines     = 1

-- superfluos/bad code
equalstrue     = 1
```

C. Benutzeranleitung

Zur Inbetriebname muss zunächst PAKCS oder KiCS2 auf dem System installiert sein.

Nachdem das Repository[1] geklont wurde, muss `cypm install` aufgerufen werden, um alle benötigten Pakete zu installieren.

Dann wird in das Verzeichnis `src/` gewechselt und dort

```
cypm curry :l Main.curry :save :quit
```

aufgerufen, wodurch die ausführbare Datei `Main` erstellt wird.

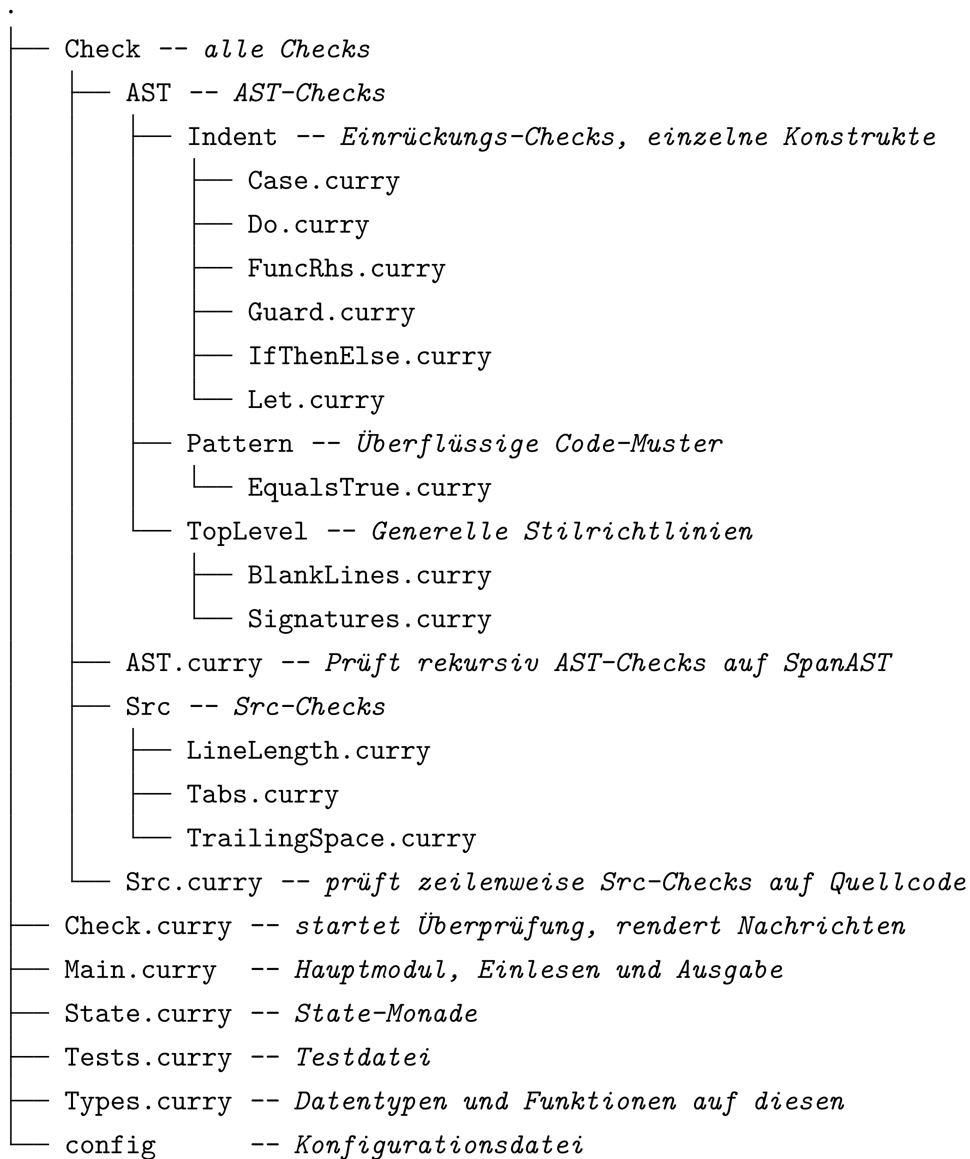
Nun wird in das Hauptmodul des prüfenden Programmes gewechselt. Hiernach kann mit

```
cypm exec /path/to/Main <filename>
```

die zu prüfende Datei mit dem Tool aufgerufen werden. Dabei ist `<filename>` die Curry-Datei mit allen eventuellen Verzeichnisebenen innerhalb des Programmes.

[1] https://git.ps.informatik.uni-kiel.de/theses/2018/2018-ncheng-ba

D. Modulstruktur



Literatur

[1] S. Antoy und M. Hanus. *Curry: A Tutorial Introduction*. Available at `http://www.curry-language.org`. 2014.

[2] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`. 2016.

[4] B. Peemöller. *Curry Style Guide*. Available at `https://www.informatik.uni-kiel.de/~mh/curry/curry-style-guide.html`. 2016.

[5] K.-O. Prott. „Ein Tool zur automatischen Dokumentationsgenerierung für Curry-Programme“. In: *Bachelorarbeit. Christian-Albrechts-Universität zu Kiel* (2018).

[6] K. Rahf. „Überprüfung von Stilrichtlinien für deklarative Programme“. In: *Masterarbeit. Christian-Albrechts-Universität zu Kiel* (2016).

[7] J. Tibell. *Haskell Style Guide*. Available at `https://github.com/tibbe/haskell-style-guide`. 2015.