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

Bachelorarbeit

Ein Tool zur automatischen Dokumentationsgenerierung für Curry-Programme

Kai-Oliver Prott

26. September 2018

Betreut durch Prof. Dr. Michael Hanus
und M. Sc. Finn Teegen

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

_____ _____

Ort, Datum Unterschrift

Zusammenfassung

Werkzeuge zur Erzeugung einer Dokumentation auf Basis des Quellcodes stehen für viele moderne Programmiersprachen zur Verfügung. Auch für Curry existiert ein solches Werkzeug, welches jedoch nicht alle Eigenschaften der Sprache unterstützt. Des weiteren ist die Syntax von Curry ähnlich zu der von Haskell, jedoch unterscheidet sich CurryDoc stark von dem entsprechenden Werkzeug für Haskell (Haddock).

In dieser Arbeit wird die grundlegende Überarbeitung der bestehenden Implementierung von CurryDoc vorgestellt. Dabei werden die vorgenommenen Änderungen am Frontend der beiden Curry-Compiler *PAKCS* und *KICS2* und die einzelnen Schritte der Dokumentationserzeugung in CurryDoc beschrieben.

Inhaltsverzeichnis

1. Einleitung

Für die Benutzung und Pflege von Softwarebibliotheken ist eine gute Dokumentation unverzichtbar. Dem Anwender bietet sie eine kompakte Darstellung der Funktionen einer Software und bei der Wartung erleichtert sie das Verständnis des Programmcodes. Eine nur aus dem Code generierte Übersicht ist jedoch in der Regel nicht ausreichend. Durch Einbeziehung der Kommentare aus dem Quellcode kann die Dokumentation durch den Programmierer um Erläuterungen zur Benutzung und dem Verhalten von Programmen ergänzt werden. Eine strukturierte Darstellung der damit gewonnenen Informationen ist eine gute Basis für die Dokumentation von Software.

1.1. Ziel

Das Ziel dieser Arbeit ist die (Weiter-) Entwicklung eines Werkzeugs für die automatische Dokumentationsgenerierung aus Curry-Programmen. Dabei soll sich der Dokumentationsstil aus Gründen der Ähnlichkeit zwischen Curry und Haskell an Haddock (siehe Abschnitt 2.1) orientieren und eine Unterstützung für Markdown (siehe Unterabschnitt 3.5.2) bieten. Zudem sollen die kürzlich eingeführten Typklassen für Curry auch dokumentiert werden können. Als Ausgabeformate der Dokumentation beschränken wir uns auf HTML und LaTeX, die Erweiterung um weitere Formate soll jedoch einfach möglich sein. CurryDoc soll als Paket über den Curry-Package-Manager (siehe Abschnitt 3.5) zur Verfügung stehen.

1.2. Aufbau

Das zweite Kapitel dieser Arbeit stellt drei Tools zur automatischen Dokumentationsgenerierung für verschiedene Programmiersprachen vor. Im dritten Kapitel werden einige Grundlagen behandelt, die für diese Arbeit relevant sind. Danach folgt im vierten Kapitel die Beschreibung der Änderungen am Curry-Frontend und der Implementierung von CurryDoc. Dabei wird erst ein Überblick über die Implementierung präsentiert und im Anschluss werden die Phasen der Dokumentationsgenerierung genauer vorgestellt. Im letzten Kapitel wird ein Fazit gezogen und ein Ausblick auf weiterführende Arbeiten gegeben.

2. Existierende Tools

Für viele Programmiersprachen gibt es bereits Tools zur Erzeugung einer Dokumentation aus dem Programmcode. Im folgenden möchte ich Haddock, JavaDoc und CurryDoc vorstellen und dabei auf die Unterschiede eingehen.

2.1. Haddock

Mit Haddock kann eine Dokumentation für Programme der Programmiersprache Haskell erzeugt werden. Dabei können als Zielformat sowohl HTML als auch LaTeX Dateien erzeugt werden. Im folgenden möchte ich auf die wichtigsten Features von Haddock eingehen. Details können zum Beispiel im *User Guide* unter `https://www.haskell.org/haddock/doc/html/index.html` nachgelesen werden.

Die Dokumentation erfolgt pro Modul, wobei standardmäßig nur exportierte Programmdeklarationen eines Moduls dokumentiert werden. Diese Deklarationen werden in einer Quellcode-ähnlichen Formatierung dargestellt und der Programmierer kann zusätzliche Anmerkungen durch speziell geformte Kommentare im Code hinzufügen.

Es gibt zwei verschiedene Möglichkeiten zum Einleiten von Haddock-Kommentaren, dabei sind sowohl einzeilige, als auch mehrzeilige Kommentare möglich:

- `-- |` oder `{- |`
- `-- ^` oder `{- ^`

Wenn mehrere hintereinander stehende Kommentare in die Dokumentation aufgenommen werden sollen, dann muss nur der erste Kommentar gekennzeichnet werden. Alle weiteren Kommentare werden dann auch von Haddock berücksichtigt, solange bis eine Zeile ohne Kommentar auftritt.

Auf welche Deklaration sich ein Kommentar bezieht wird durch die Art des Kommentars („ | “ oder „ ^“) entschieden. Ein Kommentar mit „ | “ bezieht sich dabei immer auf das als nächstes im Programmcode folgende syntaktische Element, wohingegen sich „ ^ “ auf das vorherige syntaktische Element bezieht.

Mit Haddock können Datentyp, Typklassen und Funktionsdeklarationen dokumentiert werden. Die Dokumentation für einzelne Parameter kann direkt in der Typsignatur einer Funktion erfolgen. Dem Modulkopf zugeordnete Kommentare werden als Dokumentation für das gesamte Modul interpretiert. Innerhalb dieser Kommentare können verschiedene allgemeine Informationen wie Kurzbeschreibung, Copyright oder Lizenz angegeben werden. Diese Informationen müssen in einer festen Reihenfolge auftauchen. Ein Beispiel befindet sich in Listing 2.1

In Haddock wird die Reihenfolge der Deklarationen in der generierten Dokumentation durch die Reihenfolge der Bezeichner in der Exportliste bestimmt. Zusätzlich können Überschriften für Abschnitte und geschachtelte Unterabschnitte angegeben werden. Ein mit „`-- *`“ beginnender Kommentar wird dabei als Überschrift für alle in der Exportliste nachfolgen Bezeichner gewählt. Mit jedem weiteren „`*`“ wird die „Tiefe“ (Abschnitt, Unterabschnitt, ...) der Überschrift festgelegt. Falls keine Exportliste vorhanden ist können die Überschriften auch an der gewünschten Stelle im Code zwischen den Deklarationen eingefügt werden. Wenn ein Modul reexportiert wird, dann fügt Haddock einen Verweis auf die Dokumentation des importierten Moduls in die Dokumentation des importierenden Moduls ein.

Des Weiteren kann auch Text in die Dokumentation eingefügt werden, der zu keiner bestimmten Deklaration gehört. Außerdem bietet Haddock Unterstützung für *Markdown* und weitere Formatierungsmöglichkeiten.

Listing 2.1: Beispielkommentare für Haddock

```
{-|
  Module      : Main
  Description : Short description
  Copyright   : (c) Kai Prott, 2018
  License     : MyLicense
  Maintainer  : example@mail.com
  Stability   : experimental
  Portability : POSIX

  Here could be a longer description of this module.
-}
module Main (
  -- * First section
  List(..),
  -- * Last section
  AClass(..)) where

-- | Documentation for the 'List' datatype
data List a = Nil             -- ^ Doc for the 'Nil'  constructor
            | Cons a (List a) -- ^ Doc for the 'Cons' constructor
  deriving (Eq, Show)
-- ^ Also documentation for 'List'
```

```haskell
-- | Documentation for 'main'
--   This will also be included
main = print "Hello World"

-- | Class documentation
class AClass a where

  -- | Parameters can be documented like this:
  aMethod :: a      -- ^ first Parameter
          → String -- ^ second Parameter
          → String -- ^ result
```

2.2. Javadoc

Javadoc ist ein Dokumentationswerkzeug für Java-Programme und ermöglicht die Erzeugung von HTML-Dokumentation und anderer Formate aus dem Quellcode. Ausführliche Informationen befinden sich unter der URL https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

Wie auch bei Haddock kann der Programmierer weitere Informationen zu einer Methode, Variable, Klasse, Interface oder einem Paket durch speziell formatierte Kommentare hinzufügen. Javadoc-Kommentare sind reguläre mehrzeilige Java-Kommentare, die jedoch mit einem weiteren Stern („`/**`“ statt „`/*`“) beginnen und vor der entsprechenden Deklaration stehen.

Bei Methoden, Klassen und Interfaces soll der erste Absatz des Kommentars eine Kurzbeschreibung sein. Danach folgt eine längere Beschreibung, die auch mehrere Absätze lang sein darf. Am Ende folgt ein Abschnitt für verschiedene, so genannte „Javadoc Tags“, mithilfe derer zum Beispiel Parameter und Autor dokumentiert werden können. Die Dokumentation von Variablen erfolgt ähnlich. Nur der dritte Abschnitt mit den Javadoc Tags entfällt. Die Dokumentation eines Pakets erfolgt in einer `package-info.java` Datei.

Auch innerhalb der Beschreibungstexte können manche Javadoc Tags genutzt werden um zum Beispiel Code zu formatieren oder auf andere Teile der Dokumentation zu verweisen. Zudem kann der Programmierer eigene Tags erstellen und zur Dokumentation verwenden.

Im Gegensatz zu Haddock unterstützt Javadoc kein Markdown, sondern eine Teilmenge verschiedener HTML-Tags zur Formatierung der Ausgabe.

Listing 2.2: Beispielkommentare für Javadoc

```
/**
 * Short description
 *
 * Details
 * @author Kai Prott
 * @version 1.0
 */
public class Main {

    /**
     * Main programm.
     *
     * Prints a text to the console
     *
     * @param args Command line parameters
     */
    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

2.3. CurryDoc

CurryDoc ist zur Erstellung einer Dokumentation aus Curry-Programmen gedacht. Als Ausgabeformate werden HTML, LaTeX und CDOC unterstützt, wobei letzteres eine (unvollständige) abstrakte Repräsentation der Dokumentation des Programms ist. CDOC wird als Eingabeformat für andere Tools im Curry-Universum benötigt.

Auch wenn die Syntax von Curry mehr der von Haskell als der von Java ähnlich sieht, so orientiert sich die Dokumentationssyntax von CurryDoc eher an Javadoc. CurryDoc-Kommentare werden mit `---` eingeleitet und müssen direkt vor der Funktions- oder Datentypdefinition stehen, auf die sich der Kommentar beziehen soll. Wie auch bei Haddock muss bei mehren Dokumentationskommentaren nur der erste als solcher gekennzeichnet sein. Der Aufbau der Dokumentationskommentare erfolgt dann ähnlich zu Javadoc: Der erste Satz wird als kurze Dokumentation interpretiert und zusammen mit den folgenden Sätzen auch für die ausführliche Dokumentation genutzt. Danach kommt ein Abschnitt, in dem die Parameter und Rückgabewerte einer Funktion oder die Konstruktoren und Record-Felder eines Datentyps dokumentiert werden können.

Auch die Dokumentation des Moduls ist möglich. Diese muss vor dem Modulkopf erfolgen und umfasst zuerst eine Beschreibung des Moduls und dann mehrere Tags zur Angabe des Autors, Version und anderem.

Listing 2.3: Beispielkommentare für CurryDoc

```
--- Description of the module.
---
--- @author     Kai Prott
--- @version    1.0
--- @category   general
module Main where

--- Documentation for 'main'
--  This will also be included
main = print "Hello World"

--- Documentation for the 'List' datatype
--- @cons Nil  Doc for the 'Nil'  constructor
--- @cons Cons Doc for the 'Cons' constructor
data List a = Nil
            | Cons a (List a)
  deriving (Eq, Show)

--- Method documentation
--- @param a first Parameter
--- @param s second Parameter
--- @return result
aMethod :: a → String → String
aMethod a s = s
```

Die Formatierung der Ausgabe kann durch HTML-Tags beeinflusst werden. Markdown ist ebenso nutzbar und sollte aufgrund der Lesbarkeit gegenüber HTML-Tags bevorzugt werden.

Wünschenswert wäre aufgrund der Ähnlichkeiten von Haskell und Curry eine ähnliche Dokumentationssyntax wie Haddock. Zudem fehlt in der aktuellen Version von CurryDoc die Unterstützung für Typklassen. Diese ist bei der jetzigen Implementierung nicht einfach nachrüstbar, da die bisherige Implementierung einen einfachen Ansatz verfolgt, welcher nur für die Dokumentation von Top-Level Deklarationen nutzbar ist. Bei Typklassen sollten aber auch die Funktionen der Klasse dokumentiert werden können. Diese befinden sich jedoch innerhalb der Typklassendeklaration und somit nicht mehr auf Top-Level. Aus diesen Gründen ist eine grundlegende Überarbeitung von CurryDoc nötig.

3. Grundlagen

Die Implementierung von CurryDoc baut auf einigen bereits vorhandenen Grundlagen auf, die ich im folgenden kurz vorstellen möchte.

3.1. Curry

Curry ist eine funktional-logische Programmiersprache, deren Syntax an *Haskell* erinnert. Ein Curry-Programm ist in Module unterteilt und besteht aus einer Menge von Datentypen, Funktionen, Typklassen und Instanzen für diese, Präzendenzen für Operatoren sowie *default*-Angaben.

Datentypen umfassen sowohl Typsynonyme für andere Datentypen als auch benutzerdefinierte Datentypen. Letztere werden dabei über beliebig viele Konstruktoren definiert. Die Argumenttypen der Konstruktoren müssen angegeben werden. Auch polymorphe Typen sind möglich und für algebraische Datentypen können bestimmte Typklasseninstanzen automatisch erzeugt werden.

Beispiel 3.1.1
In diesem Beispiel zeigen wir die bereits vordefinierten Typen `String` *und* `Maybe`. `String` *ist nur ein Typsynonym für eine Liste von* `Char`*s.* `Maybe` *ist polymorph über einem beliebigen Typen* `a` *und besitzt die beiden Konstruktoren* `Nothing` *sowie* `Just`*. Durch das Schlüsselwort* `deriving` *wird dem Compiler mitgeteilt, dass er Instanzen für die angegebenen Typklassen erzeugen soll.*

```
type String = [Char]

data Maybe a = Nothing
             | Just a
  deriving (Eq, Ord, Show, Read)
```

Funktionen werden über *Pattern-Matching* auf den Konstruktoren der Argumenttypen definiert. Eine Funktion kann dabei mehrere solcher Regeln umfassen. Da Curry einige Eigenschaften aus der Logikprogrammierung übernimmt, können Funktionen auch nichtdeterministisch sein. Zudem darf der Programmierer eine Typsignatur für Funktionen angeben. Diese wird, falls angegeben, vom Compiler auf Übereinstimmung mit dem aus

den Regeldefinitionen inferierten Typen geprüft. Eine Typsignatur umfasst dabei die Argumenttypen und den Ergebnistyp der Funktion, sowie (falls nötig) einen Kontext für vorhandene Typvariablen.

Beispiel 3.1.2
Im Folgenden definieren wir eine Funktion `fromJust`*, welche eine Argument vom Typ* `Maybe a` *bekommt und den Inhalt extrahiert.*

```
fromJust :: Maybe a → a
fromJust (Just x) = x
fromJust Nothing  = error "No value to extract from Maybe"
```

Typklassen erweitern die Sprache um Ad-hoc Polymorphismus und erlauben somit das Überladen von Funktionen und Operatoren für verschiedene Typen. Sie ähneln damit eher Schnittstellen aus objektorientierten Programmiersprachen und nicht Klassen. Die Definition einer Typklasse umfasst eine Menge von Funktionsnamen und deren Typen. Für diese Typklassen kann der Programmierer dann Instanzen für Datentypen angeben.

Beispiel 3.1.3
Eine Typklasse zum Vergleichen von Werten ist in Curry wie folgt definiert.

```
class Eq a where
  (==), (/=) :: a → a → Bool

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

Es folgt eine Instanz dieser Klasse für Wahrheitswerte.

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

Wie man sieht, ist es ausreichend nur eine Definition für `(==)` *anzugeben, da in der Typklasse die Ungleichheit als Negation der Gleichheit definiert wurde.*

Sowohl Datentypen als auch Funktionen können als extern gekennzeichnet werden. Für diese muss dann der Compiler die Implementierungsdetails bereitstellen.

Mithilfe von *Fixity*-Deklarationen kann der Programmierer die Bindungsstärke und Assoziativität eines Operators bestimmen. Die Bindungsstärke kann zwischen Null und Neun betragen. Über die Schlüsselworte `infixl`, `infixr` oder `infix` wird die Assoziativität festgelegt.

Beispiel 3.1.4
Im folgenden definieren wir den Operator für die Gleichheit mit Bindungsstärke vier und neutraler Assoziativität.

```
infix 4 ==
```

Default-Deklarationen sind ein Hinweis an den Compiler, welcher Typ bei Mehrdeutigkeiten bevorzugt verwendet werden soll. Solche Deklarationen sind für diese Arbeit nicht von Bedeutung und nur aufgrund der Vollständigkeit aufgeführt.

Eine detailliertere Betrachtung der Syntax und Semantik von Curry ist für diese Arbeit unerheblich.

3.2. Curry-Frontend

Ein Compiler besteht im Allgemeinen aus zwei Teilen. Im vorderen Teil (dem *Frontend*) wird der Quellcode auf syntaktische sowie semantische Korrektheit analysiert und in eine von Hardware und Zielcode unabhängige Zwischensprache (*Intermediate Representation - IR*) übersetzt. Im hinteren Teil (dem *Backend*) wird aus der Zwischensprache das Zielprogramm für eine spezielle Zielarchitektur und Zielsprache erzeugt. Diese Aufteilung sorgt dafür, dass im Optimalfall für eine Übersetzung von n Quellsprachen auf m Ziele nur $m + n$ (siehe Abbildung 3.1) anstatt $m \cdot n$ (siehe Abbildung 3.2) vollständige Übersetzer benötigt werden.

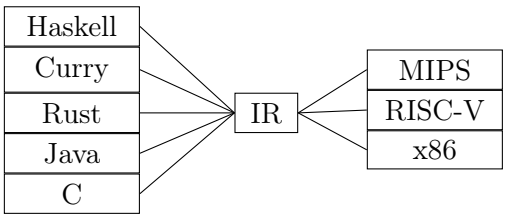


Abbildung 3.1.: Benötigte Übersetzer mit Zwischensprache

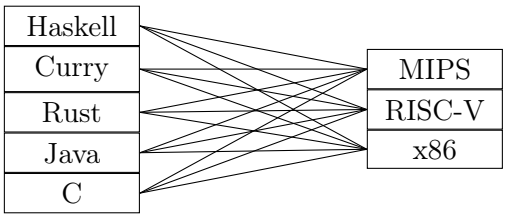


Abbildung 3.2.: Benötigte Übersetzer ohne Zwischensprache

Als *Curry-Frontend* wird das gemeinsame Frontend einiger Curry-Compiler bezeichnet. Seine Hauptaufgabe ist die Generierung von Code im FlatCurry Format, welcher von verschiedenen Backends in andere Sprachen übersetzt wird. Zudem ist das Curry-Frontend in der Lage, verschiedene andere Repräsentationen des Quellcodes für eine Reihe von Tools zu erzeugen. Relevant für diese Arbeit sind davon AbstractCurry (siehe Abschnitt 3.3), die Tokenfolge (siehe Abschnitt 3.2.1) sowie eine HTML-Darstellung des Quellcodes mit Syntaxhighlighting und Querverweisen auf verschiedene Bezeichner.

Der Kompiliervorgang im Frontend ist in mehrere Phasen unterteilt. Diese Phasen lassen sich in drei Abschnitte einteilen: Erkennung, Prüfung und Transformation [Tee16, 53]. Der gesamte Kompilierverlauf ist mit den englischen Namen der Phasen in Abbildung 3.3 dargestellt. Nur die in der Abbildung rot umrandeten Phasen sind für diese Arbeit wichtig. Sie werden im weiteren Verlauf kurz beschrieben. Details zu allen Phasen sind in [Tee16] nachzulesen.

3.2.1. Erkennung

Als Ergebnis der Erkennung wird eine Darstellung des Quellcodes als *abstrakter Syntaxbaum* (Abstract Syntax Tree → AST) generiert. Dabei lassen sich zwei Phasen unterscheiden:

Lexer Die lexikographische Analyse ist die erste Phase der Codeerkennung. Hier wird der Quellcode vom *Lexer* in zusammengehörige Einheiten (*Tokens*) zerlegt. Die Ausgabe ist eine Folge dieser Token. Wahlweise sind die Kommentare aus dem Quellcode auch als Token in der Folge enthalten oder sie werden einfach überlesen.

Parser Der Parser konstruiert mithilfe der gegebenen Grammatik von Curry den abstrakten Syntaxbaum aus der Tokenfolge des Lexers. Diese Phase wird deshalb meistens als syntaktische Analyse bezeichnet. Der AST umfasst bisher nur bei ausgewählten syntaktischen Elementen eine Angabe der ursprünglichen Anfangsposition im Quellcode.

3.2.2. Prüfungen

Die Prüfungen stellen die syntaktische und semantische Korrektheit des Programms sicher und führen gegebenenfalls Umordnungen von Elementen im AST durch. Ansonsten soll der Originalzustand des AST bis zu den Transformationen weitestgehend erhalten bleiben.

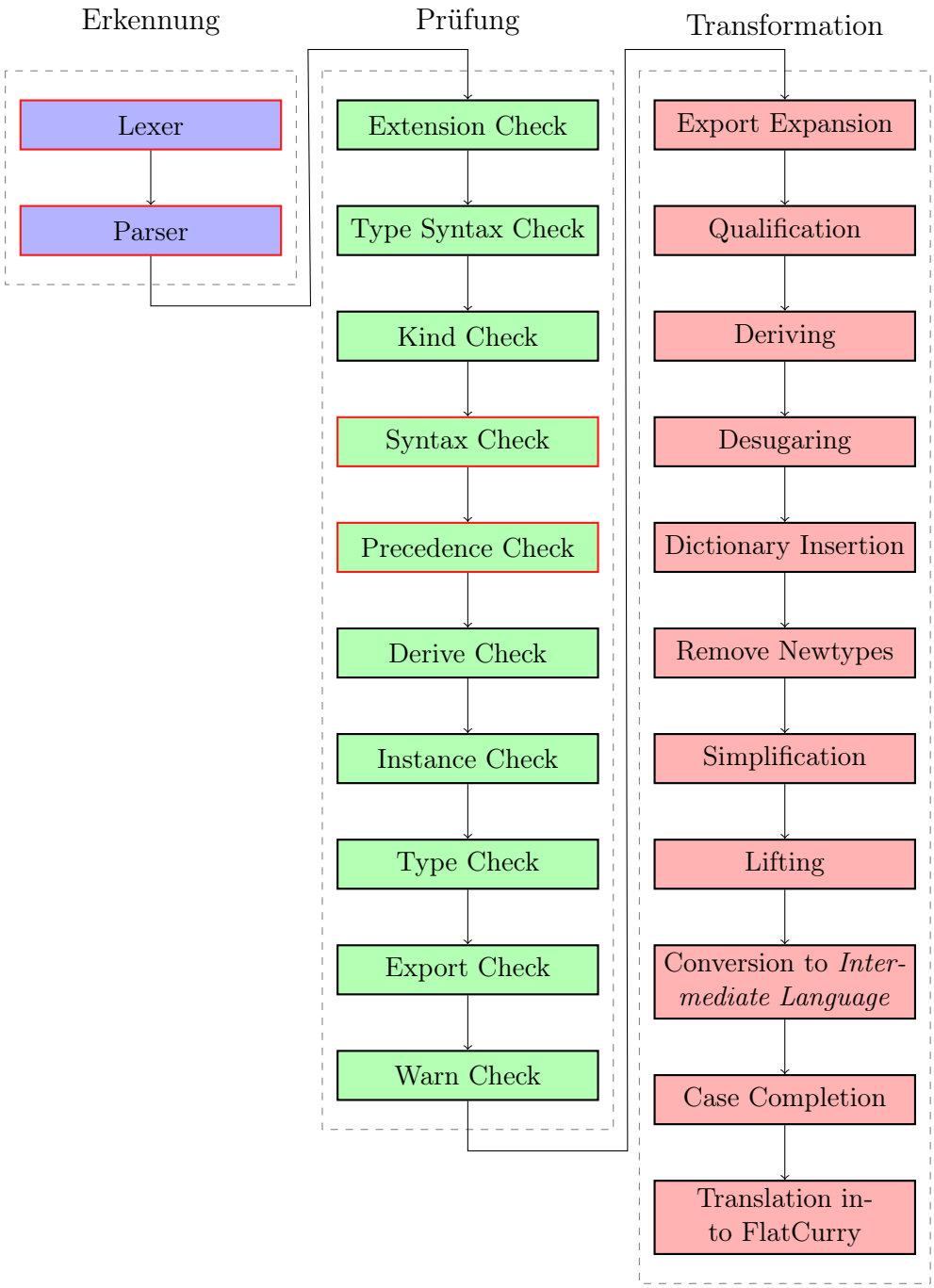


Abbildung 3.3.: Kompilierverlauf im Frontend - aktualisierte Version des Diagramms aus [Tee16, 73]

Syntax Check In dieser Phase werden nullstellige Konstruktoren und Variablen unterschieden. Dies kann nicht im Parser erfolgen, da die Grammatik für Variablen und nullstellige Konstruktoren gleich ist. Zudem wird sichergestellt, dass alle Variablen und Konstruktoren definiert und nicht mehrdeutig sind. Zuletzt werden noch hintereinander stehende Regeln derselben Funktion in eine Funktionsdeklaration im AST zusammengefasst.

Precedence Check Diese Prüfung stellt zunächst sicher, dass es keine doppelten oder fehlerhaften *Fixity*-Deklarationen gibt. Danach werden diese in einer Datenstruktur gespeichert und mit denen der importierten Module ergänzt. Beim *Precedence Check* werden zudem Infix-Operatoren im abstrakten Syntaxbaum basierend auf ihrer Bindungsstärke und Assoziativität neu angeordnet. Dies ist nötig, da dem Parser die gegebenenfalls im Quellcode vorhandenen, benutzerdefinierten Präzedenzen von Operatoren beim Lesen des Quellcodes nicht bekannt sind.

3.2.3. Transformationen

Der Transformationsprozess verändert das Programm schrittweise und erzeugt schlussendlich Code im (Typed-)FlatCurry Format, welcher als Eingabe für den *KICS2* (Curry-Compiler nach Haskell) und den *PAKCS* (Curry-Compiler nach Prolog) dient. Für die meisten anderen Zielformate des Curry-Frontends muss diese Transformation nicht durchgeführt werden.

3.3. AbstractCurry

AbstractCurry ist eines der möglichen Zielformate des Curry-Frontends. AbstractCurry hat eine möglichst detailreiche abstrakte Darstellung des Quellmoduls zum Ziel. AbstractCurry umfasst:

- Den Modulnamen
- Die Namen aller importierten Module
- Die eventuell vorhandene *default*-Deklaration
- Alle Typklassen und Instanzen des Moduls
- Alle Datentypen des Moduls mit Ausnahme von extern definierten
- Alle Funktionen des Moduls
- Alle im Modul definierten Operatorpräzedenzen.

Bei jedem exportierbaren Bezeichner steht zudem die Information, ob dieser exportiert wurde oder nicht. Informationen zu der ursprünglichen Position einer Deklaration im Quellcode sind in AbstractCurry nicht vorhanden.

Die Funktionen im Modul sind alle vollständig getypt und alle vorkommenden Bezeichner sind vollständig mit dem Namen ihres Ursprungsmoduls qualifiziert.

3.4. FlatCurry

Im Unterschied zu *AbstractCurry* hat *FlatCurry* nicht eine möglichst detailreiche Darstellung zum Ziel. Stattdessen wurde für eine möglichst einfache Darstellung einiges an „syntaktischem Zucker“ entfernt. FlatCurry umfasst nicht mehr die Typklassen und Instanzen, da diese mithilfe einer Wörterbuch-Transformation umgesetzt wurden [WB89].

Die Funktionen wurden mithilfe der Transformationen aus dem Frontend (siehe Abschnitt 3.2) vereinfacht. Bei den Datentypen wurden zudem mit *newtype* definierte Datentypen in einfache Datentypen umgewandelt. Externe Datentypen sind in FlatCurry noch vorhanden, jedoch nur als einfache Datentypen ohne Konstruktoren. Datentypen, die schon im Quellcode ohne Konstruktoren definiert wurden, bekommen zur Unterscheidung einen generierten Platzhalterkonstruktor. Das Vorhandensein der externen Datentypen ist auch der einzige Grund, warum FlatCurry für diese Arbeit relevant ist.

FlatCurry ist sowohl mit ungetypten Ausdrücken, als auch als Variante mit vollständig annotierten Typen an jedem Ausdruck verfügbar. In beiden Formaten sind alle Bezeichner vollständig mit ihrem Modulnamen qualifiziert.

3.5. Curry-Pakete

Über den *Curry Package Manager (CPM)* lassen sich Pakete für verschiedene Anwendungszwecke zu dem eigenen Projekt hinzufügen und es können auch eigene Pakete erstellt werden [HO17]. Der Paketmanager nutzt außerdem die vorhandene Version von CurryDoc, um eine Dokumentation für die Pakete zu erstellen.

Für AbstractCurry und FlatCurry stehen Pakete bereit, die den Umgang mit diesen Formaten in Curry erlauben. Alle anderen für CurryDoc benötigten Pakete werden im weiteren Verlauf vorgestellt.

3.5.1. Curry Analysis

Für Curry existiert das *Curry Analysis Server System* (CASS), ein Tool zur automatischen Analyse bestimmter Programmeigenschaften. Dazu gehören unter anderem Analysen auf nichtdeterministisches Verhalten, Terminierung und ob Funktionen partiell definiert sind. Diese Eigenschaften können für den Benutzer einer Bibliothek eine Rolle spielen und sollten somit dokumentiert werden.

3.5.2. Markdown-Paket

Curry bietet ein Paket zur Verarbeitung von Markdown in Zeichenketten. Als Zielformat stehen dabei HTML, LaTeX und PDF zur Auswahl. Markdown ist eine Auszeichnungssprache (Sprache zur Formatierung von Texten), die auch unverarbeitet lesbar sein soll. In Abbildung 3.4 befinden sich einige Beispiele. Eine vollständige Beschreibung der unterstützten Syntax ist unter `https://www.informatik.uni-kiel.de/~pakcs/markdown_syntax.html` zu finden.

`Ausgangstext`	`Zielform`
Text kann *kursiv* und **fett** gedruckt werden	Text kann *kursiv* und **fett** gedruckt werden
Text kann einen Link enthalten <https://www.uni-kiel.de/de/> oder (Uni-Kiel)[https://www.uni-kiel.de/de/]	Text kann einen Link enthalten https://www.uni-kiel.de/de/ oder Uni-Kiel
Text kann Code enthalten ‘True /= False‘	Text kann Code enthalten `True /= False`
Aufzählungen sind möglich - Farbe: + Gelb + Rot + Blau - Form: + Kreis + Quadrat Bei Nummerierungen ist die Zahl egal 1. Erstes Element 5. Zweites Element	Aufzählungen sind möglich • Farbe: ◦ Gelb ◦ Rot ◦ Blau • Form: ◦ Kreis ◦ Quadrat Bei Nummerierungen ist die Zahl egal 1. Erstes Element 2. Zweites Element
Und vieles mehr...	Und vieles mehr...

Abbildung 3.4.: Ausschnitte der unterstützten Markdown-Syntax

3.5.3. HTML-Paket

Curry bietet ein Paket zur Erstellung von statischen und dynamischen HTML-Seiten. Die Funktionalität umfasst auch eine Unterstützung für CGI (*Common Gateway Interface*). Für CurryDoc wird dies aber nicht verwendet.

Über verschiedene Funktionen können HTML-Elemente und Strings zu neuen HTML-Elementen zusammengefasst und mit Attributen versehen werden. Auf diese Weise kann die Struktur des resultierenden HTML-Dokuments aufgebaut werden. Dies ist für die automatische Generierung von Dokumentationsseiten nützlich, da ein hoher Grad an Komposition ermöglicht wird.

In [Han01] befinden sich mehr Informationen zu den Ideen hinter dem Paket.

4. Implementierung

Um für CurryDoc eine zu Haddock ähnliche Syntax umzusetzen, müssen die Dokumentationskommentare den Sprachelementen aus dem Quellcode zugeordnet werden. Dazu reichern wir den abstrakten Syntaxbaum des Curry-Frontends mit genauen Positionsinformationen an. Dieser AST kann dann zusammen mit den Kommentaren und anderen Informationen zu einer abstrakten Dokumentation verarbeitet werden. Aus dieser kann schließlich als Ausgabeformat HTML oder LaTeX erzeugt werden. Die Erzeugung von CDOC ist nicht Teil dieser Arbeit.

Die Generierung der Dokumentation mit CurryDoc lässt sich in zwei Abschnitte und zehn Phasen zerlegen. Diese sind in Abbildung 4.1 dargestellt.

1. **Curry-Frontend:** Im ersten Abschnitt wird mithilfe des Curry-Frontend aus dem Quellcode AbstractCurry, FlatCurry, der AST und die Tokenfolge der Kommentare erzeugt. Die dazu vorgenommenen Änderungen am Code des Frontends sind in Abschnitt 4.1 beschrieben.

2. **CurryDoc:** Der zweite Abschnitt umfasst alle Phasen von CurryDoc. Hier wird aus den vom Frontend erzeugten Daten schrittweise die Dokumentation erstellt. Die einzelnen Phasen werden in Abschnitt 4.2 genauer beschreiben.

   a) **Kommentarzuordnung:** Zuerst wird jeder Dokumentationskommentar dem Modulkopf oder einer Deklaration bzw. einem Teil einer Deklaration zugeordnet. Dabei werden auch die Überschriften für die Dokumentationsabschnitte in der Exportliste zugeordnet und die dadurch definierte Exportstruktur erzeugt.

   b) **Ergänzung mit AbstractCurry:** Danach erweitern wir die Datenstruktur der Zuordnungen um Informationen aus AbstractCurry. In diesem Schritt werden auch unkommentierte Deklarationen aufgesammelt, da diese zur Vereinfachung im vorherigen Schritt ignoriert werden.

   c) **Anreicherung mit Analyseergebnissen:** Die nun vollständigen Deklarationen werden als nächstes um Analyseinformationen ergänzt. Dazu zählen eventuelle benutzerdefinierte Präzedenzen, ob eine Deklaration als extern gekennzeichnet ist, im Code vorhandene Spezifikationen und Eigenschaften sowie einige der in Unterabschnitt 3.5.1 erwähnten Analysen.

d) **Analyse der Modulkommentare:** Parallel werden die dem Modulkopf zugeordneten Kommentare ausgewertet, um Informationen wie die Namen der Autoren oder eine Beschreibung des Moduls zu bekommen.

e) **Modul Re-Export Inlining:** Ebenso werden parallel alle Reexporte anderer Module in der Exportstruktur durch alle importierten Bezeichner des Moduls ersetzt. Dies wird nicht gemacht, wenn das Modul vollständig importiert wurde.

f) **Einfügen der Deklarationen in die Exportstruktur:** Als nächstes werden die Deklarationen in die durch den vorherigen Schritt erweiterte Exportstruktur eingefügt. Für diese Phase wird auch die Dokumentation von potentiell allen importierten Modulen benötigt.

g) **Zusammenführen der Informationen:** Dann werden alle Informationen in einem abstrakten Datentyp zusammengefasst und in einer Datei gespeichert.

h) **Generierung von HTML/LaTeX:** Zuletzt erfolgt die Generierung der Ausgabe im HTML oder LaTeX Format. In diesem Schritt wird auch gegebenenfalls in den Kommentaren vorhandenes Markdown verarbeitet.

Eine detaillierte Beschreibung der umgesetzten Dokumentationssyntax befindet sich in Anhang B.

4.1. Änderungen am Curry-Frontend

Die Hauptänderung am Curry-Frontend ist die Erweiterung der Positionsangaben im abstrakten Syntaxbaum und die Ausgabe des AST in eine Datei. Des Weiteren fügen wir die Möglichkeit hinzu, eine Liste aller Quellcode-Kommentare zu erzeugen. Zusätzlich nutzen wir das Frontend zur Generierung von FlatCurry und AbstractCurry.

4.1.1. Hinzufügen von Span-Informationen zum AST

Bisher beinhaltet der vom Parser konstruierte Abstrakte Syntaxbaum nur wenige Positionsinformationen. Nur der Anfang von Deklarationen, Bezeichnern und Regeln von Funktionen oder Case-Ausdrücken wird gespeichert. Zudem existiert im Frontend eine Typklasse `HasPosition` für einen einfachen Zugriff auf Positionen, welche jedoch nicht genutzt wird. Auch ein Datentyp `Span` existiert bereits, welcher aber nur im Lexer verwendet wird. Diese Informationen reichen für eine Zuordnung von Kommentaren zu syntaktischen Einheiten nicht aus.

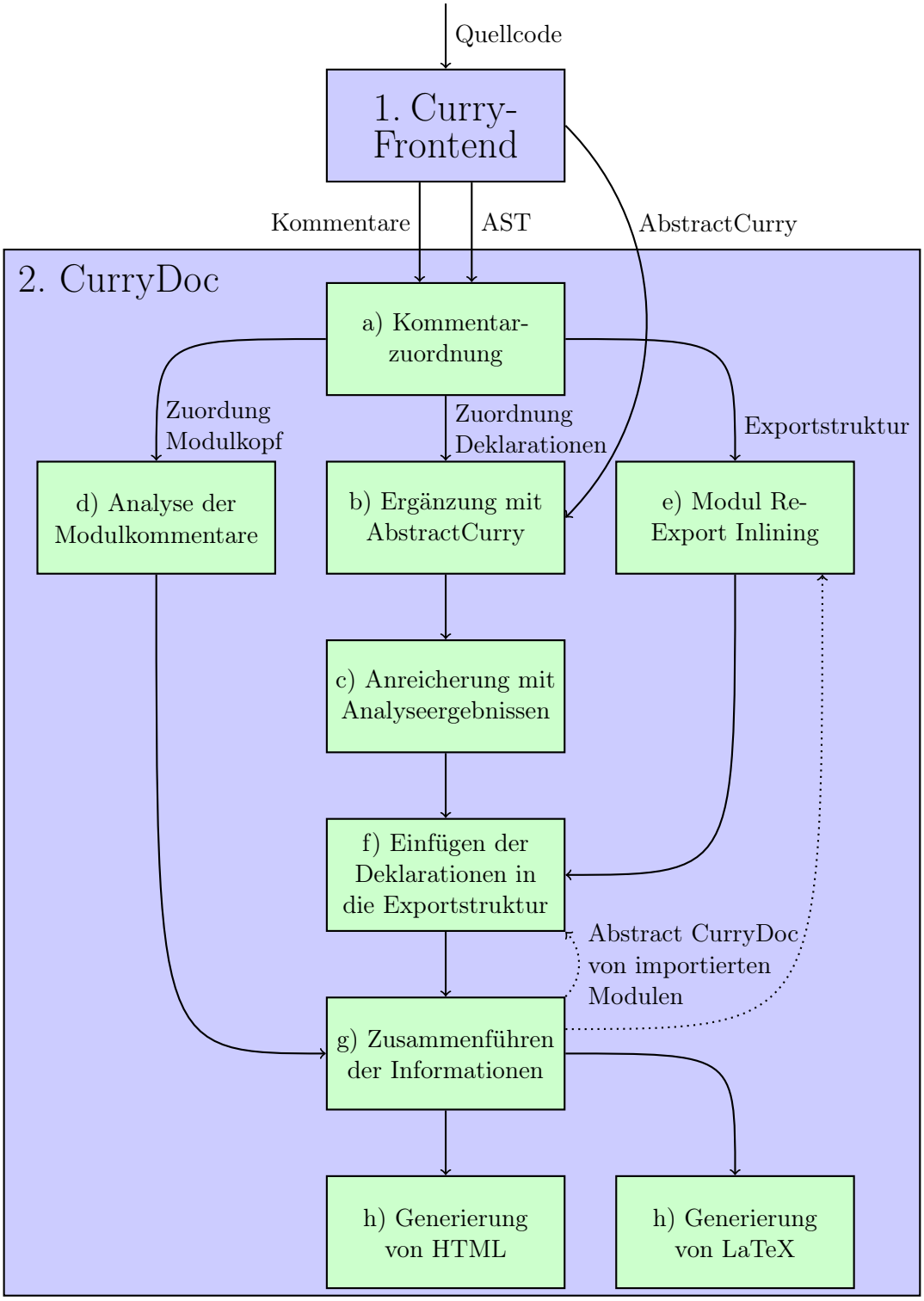


Abbildung 4.1.: Überblick über die Phasen von CurryDoc

Wir erweitern die Positionsangaben im AST deshalb zu *Span-Informationen*, welche den Bereich (engl. Span) eines Elements im Quellcode beschreiben. Die Span-Informationen sind aus den „Haskell Source Extensions“ übernommen, einer Haskell-Bibliothek zur Verarbeitung von Haskell-Quellcode. Die Idee ist, dass die Informationen ausreichend genau sind, um den Quellcode aus dem abstrakten Syntaxbaum exakt rekonstruieren zu können. Diese Genauigkeit ist für CurryDoc nicht nötig, jedoch ist es für andere Werkzeuge (z.B. einen *Style Checker*) nützlich, so genaue Informationen zu haben. Wir definieren deshalb folgenden Datentypen:

```
data SpanInfo = NoSpanInfo
              | SpanInfo
                { srcSpan       :: Span
                , srcInfoPoints :: [Span]
                }
```

`Span` und `Position` sind bereits definiert als:

```
data Span = NoSpan
          | Span
            { file  :: FilePath
            , start :: Position
            , end   :: Position
            }

data Position = NoPos
              | Position
                { file   :: FilePath
                , line   :: Int
                , column :: Int
                }
```

Ein SpanInfo-Eintrag beinhaltet in `srcSpan` den gesamten Bereich einer Deklaration und in `srcInfoPoints` eine beliebige Anzahl an zusätzlichen Bereichsangaben. Letztere werden zum Speichern des Span von Schlüsselwörtern und reservierten Symbolen genutzt. In der Datenstruktur fehlt die Unterstützung für Code mit explizitem Layouting über geschweifte Klammern und Semikolons. Diese Information geht beim Einlesen des Quellcodes im Frontend auch weiterhin verloren. Beispiel 4.1.1 zeigt, wie so eine Span-Information aussehen kann.

Den FilePath auch in dem Span-Datentypen zu speichern ist nicht wirklich nötig und könnte entfernt werden. Ebenso braucht nicht jede Position einen eigenen FilePath, da bisher der Code eines Moduls nicht aus mehreren Dateien zusammengesetzt sein kann. Da sich zur Zeit alle Span und Positionen aus diesem Grund denselben FilePath teilen, ist der zusätzliche Speicherverbrauch dank *sharing* unerheblich. Nur beim Schreiben des Datentypen in eine Datei wird zur Reduzierung der Dateigröße auf den FilePath beider Datentypen verzichtet.

Beispiel 4.1.1 (Span-Information einer Typklasse)
Für eine Typklasse der Form

```
class Name a where
  function :: a → Int
```

könnte die SpanInfo folgendermaßen aussehen (vereinfachte Darstellung):

```
SpanInfo {
   srcSpan       =  Span (Position 1  1) (Position 2 22)
                    -- ^ gesamte Deklaration
   srcInfoPoints = [Span (Position 1  1) (Position 1  5)
                    -- ^ "class" Schlüsselwort
                    Span (Position 1 14) (Position 1 18)]
                    -- ^ "where" Schlüsselwort
}
```

Die Funktionen der Klasse besitzen selbst je eine eigene SpanInfo.

Die Anzahl der Einträge in `srcInfoPoints` ist offensichtlich abhängig von dem zugehörigen Knoten im AST. Welcher Eintrag in der Liste zu welchem Quellcode-Element gehört, ist auch abhängig von der genauen Form des Knotens. Eine Übersicht über die vorhandenen Einträge und ihre Reihenfolge für jeden Knoten im AST befindet sich in Anhang D.

Beispiel 4.1.2 (Unterschiedliche srcInfoPoints einer Typsignatur)
Eine Typsignatur der Form

```
function :: a → Int
```

besitzt in `SrcInfoPoints` *nur den Eintrag für* `::` [1]*. Eine Typsignatur der Form*

```
functionA, functionB :: a → Int
```

hat jedoch als ersten Eintrag den Span des Kommas und danach den vom `::`*.*

Um den Zugriff und das Verändern der SpanInfos von Entitäten zu vereinfachen, definieren wir uns zusätzlich eine Typklasse `HasSpanInfo`. Aufbauend auf dieser Typklasse erstellen wir dann auch einige anderen Funktionen, um zum Beispiel nur die Anfangsposition aus einem Datentypen mit Span-Info zu extrahieren. Man sieht also, dass jeder Datentyp mit einer SpanInfo auch die Kriterien für die Typklasse `HasPosition` erfüllt. Eine Instanz dafür lässt sich auf triviale Weise für beliebige Datentypen mithilfe der Funktionen aus `HasSpanInfo` implementieren. Wir definieren deshalb `HasSpanInfo` wie folgt.

[1] Der → gehört zum Typausdruck in der Typsignatur und nicht direkt zur Typsignatur

```
class HasPosition a ⇒ HasSpanInfo a where
  getSpanInfo :: a → SpanInfo
  setSpanInfo :: SpanInfo → a → a

class HasPosition a where
  getPosition :: a → Position
  setPosition :: Position → a → a
```

Nun ergänzen wir den Datentyp von jedem Knoten des abstrakten Syntaxbaums um die neue Span-Information und erstellen je eine Instanz der erwähnten Typklassen für jeden Datentypen des AST. Für CurryDoc müssten nicht alle Knoten des abstrakten Syntaxbaums erweitert werden, jedoch können diese Informationen ebenfalls für weitere Tools wie *Style Checker* oder die Rekonstruktion des ursprünglichen Quellcodes genutzt werden.

4.1.2. Anpassung des Parsers

Nach der Anpassung der Datenstruktur des AST muss auch der Parser erweitert werden. Dieser arbeitet mithilfe so genannter Parserkombinatoren, welche eine sehr deklarative und lesbare Schreibweise von Parsern ermöglichen. Beispielsweise wird der folgende Kombinator häufig genutzt:

```
sepBy :: Symbol s ⇒ Parser a s b → Parser a s c → Parser a s [b]
```

Mithilfe dieses Kombinators kann man zum Beispiel einen Parser für eine mit Komma getrennte Liste von Zahlen implementieren. Dies würde wie folgt aussehen, wobei wir einen Parser für Kommas und Zahlen als vorhanden voraussetzen:

```
number `sepBy` comma
```

Das Ergebnis dieses Ausdrucks ist ein Parser, der als Ergebnis eine Liste von Zahlen beinhaltet. Das Problem dabei ist, dass die Informationen über die Kommas nicht mehr vorhanden sind. Diese werden für die Span-Informationen aber gebraucht. Deshalb erstellen wir einen ähnlichen Parserkombinator:

```
sepBy :: Symbol s ⇒ Parser a s b → Parser a s c → Parser a s ([b],[Span])
```

Dieser liefert nun einen Parser, bei dem die Spans nicht verloren gehen. Ähnliche Änderungen müssen auch bei einigen anderen Parserkombinatoren vorgenommen werden.

Der Span eines einzelnen Tokens lässt sich bereits ermitteln, der Lexer muss dementsprechend nicht angepasst werden. Doch zur Bestimmung der Endposition eines ganzen Knotens (z.B. einer Deklaration oder eines Infix-Ausdrucks) im abstrakten Syntaxbaum reicht dies meistens nicht aus. Stattdessen kann in diesen Fällen die Endposition über die

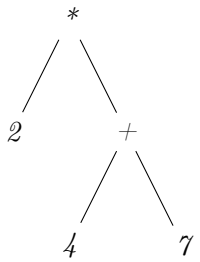
Kindknoten bestimmt werden, denn meistens stimmt die Endposition eines Knotens mit der seines letzten Kindknotens überein. Beispielsweise endet eine Typklassendeklaration am Ende der letzten Funktionsdefinition der Typklasse. Für Klassen ohne Funktionsdefinitionen muss man stattdessen die Endposition des `where` Schlüsselwortes nehmen. Da die genaue Berechnung dieser Eigenschaft für verschiedene Datentypen unterschiedlich sein kann, erweitern wir die Typklasse `HasSpanInfo` um eine Funktion `updateEndPos` mit der Signatur `a` $\rightarrow$ `a`, welche diese Berechnung vornimmt.

4.1.3. Anpassungen der Prüfungen und Transformationen

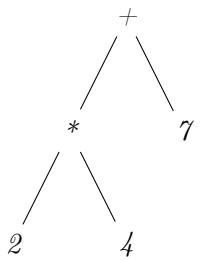
Aufgrund der Änderungen am AST müssen auch alle nachfolgenden Phasen im Curry-Frontend angepasst werden. Dabei muss bei den im *Precedences Check* vorgenommenen Umordnungen an Infix-Ausdrücken darauf geachtet werden, dass die Span-Informationen entsprechend angepasst werden. Dies ist aber kein Problem, da bei Infix-Operationen die Span-Informationen der Operanden bekannt sind.

Im *Syntax-Check* ergibt sich das Problem, dass für *Functional Patterns* (nur über Spracherweiterung nutzbar) bereits neuer Code erzeugt und im AST ersetzt wird. Für diesen Fall ist noch keine Lösung vorhanden. Bisher wird hier nur `NoSpanInfo` für den neuen Code eingetragen. Diese Lösung wird auch in den Transformationsphasen gewählt. Spätestens ab dann sind die Span-Informationen nicht mehr korrekt. In der Zukunft könnte man die in den Prüfungsphasen erzeugten Fehlermeldungen mithilfe der Span-Informationen noch verbessern.

Beispiel 4.1.3 (Beispiel für eine Umordnung von Infix-Ausdrücken)
Der Ausdruck `2 * 4 + 7` wird vom Parser gelesen als hätten alle Operatoren dieselbe Bindungsstärke und wären rechts-assoziativ. Entsprechend ergibt sich der eigentlich falsche Ausdrucksbaum:



Der Span der Multiplikation geht vom Anfang bis zum Ende und der Span der Addition geht von der Vier bis zum Ende. Nach der Umordnung mit den korrekten Präzedenzen ergibt sich als Baum:



Der Anfang der nach oben gezogenen Addition muss auf den ursprünglichen Anfang der Multiplikation geändert werden. Das korrekte Ende der Multiplikation kann über die Funktion `updateEndPos` *bestimmt werden, da die Vier auch eine Span-Information besitzt.*

4.1.4. Weitere Anpassungen am Frontend

Im Frontend müssen zusätzlich noch Funktionen eingebaut werden, damit der AST und die Tokens der Kommentare als Ziel beim Kompilieren erzeugt werden können. Das Format orientiert sich größtenteils an den von Haskell generierten Show-Instanzen. Die Ausgabedatei für den AST wird dabei viel zu groß. Deshalb erweitern wir das Frontend auch um das Ziel *short-ast*, welches vor der Ausgabe des Syntaxbaums bei allen Funktionen die Regeln löscht. Dadurch sinkt die Dateigröße auf akzeptable Ausmaße.

4.2. Änderungen an CurryDoc

Da sich der Implementierungsansatz bei CurryDoc grundlegend verändert hat, kann nur wenig Code aus der vorherigen Version unverändert übernommen werden. Um die Wartbarkeit zu erhöhen, ist das Ziel der ersten Phasen von CurryDoc die Erzeugung einer abstrakten Dokumentation, welche danach nur noch ins entsprechenden Zielformat übersetzt werden muss. Im Folgenden werden die einzelnen Phasen der Dokumentationsgenerierung vorgestellt.

4.2.1. Zuordnung von Kommentaren zu syntaktischen Einheiten

Bisher wurden in CurryDoc Kommentare der dahinter folgenden Deklaration zugeordnet. Diese Zuordnung ermöglichte eine einfache Implementierung, der Code ist aber auf den neuen Ansatz mit Span-Informationen nicht übertragbar.

Die neue Implementierung ordnet zuerst die Kommentare entsprechend ihres Typs (`-- |` oder `-- ^`) der passenden Top-Level Deklaration zu. Aus Kompatibilitätsgründen zur

alten Version wird `---` als `-- |` interpretiert und eine Warnung auf der Konsole ausgegeben. Die Zuordnung der beiden Kommentartypen erfolgt gleichzeitig. Sollten auf den Dokumentationskommentar auch noch „normale“ Kommentare folgen, so werden diese ebenfalls zugeordnet. Im Folgenden bezeichnen wir `-- |` als *Pre*, `-- ^` als *Post* und alle anderen Kommentare als *None*. Eine Zuordnung wird aufgrund des „vertikalen Abstands“ des Kommentars und der Deklaration vorgenommen.

Definition 4.2.1 (Vertikaler Abstand von zwei Span)
Es seinen $s_1 = (Span\ start_1\ end_1)$ und $s_2 = (Span\ start_2\ end_2)$.
Dann ist der vertikale Abstand von s_1 und s_2 wie folgt definiert.

$$vertDist(s_1, s_2) = \begin{cases} rowDist(end_1, start_2) & wenn\ rowDist(end_1, start_2) \geq 0 \\ 0 & wenn\ end_1 \leq end_2 \\ -rowDist(end_2, start_1) & sonst \end{cases}$$

Dabei wird für die Ordnung von Positionen erst die Zeile und dann die Spalte verglichen (lexikographische Ordnung).

Definition 4.2.2 (Zeilenabstand von zwei Positionen)
Es seien $p_1 = (Position\ row_1\ col_1)$ und $p_2 = (Position\ row_2\ col_2)$.
Dann ist der Zeilenabstand wie folgt definiert.

$$rowDist(p_1, p_2) = row_2 - row_1$$

Der zweite Fall bei der Definition des vertikalen Abstands ist dafür, dass 0 als Ergebnis für sich überlappende *Span* herauskommt. Falls eines der Argumente `NoPos` bzw. `NoSpan` ist, so geben wir ebenfalls 0 als Ergebnis zurück. Dies vereinfacht die Implementierung zum Beispiel für Funktionen, die den Abstand zum Span der letzten erfolgreichen Zuordnung benötigen. Denn für den ersten Aufruf existiert ein solcher noch nicht und eine Übergabe als `Maybe`-Wert würde die Anzahl der Regeln der Funktion vergrößern. Entsprechend reicht es, wenn man dieser Funktion `NoSpan` übergeben kann, ohne dass die Berechnung des Abstands fehlschlägt.

Beispiel 4.2.1 (Abstand von zwei aufeinander folgenden Spans)
Angenommen wir wollen den Kommentar im folgenden Code der Deklaration zuordnen.

```
{- | This is a pre-comment
     on multiple lines -}

newtype MyType = ACons Int
  deriving Show
```

Dann sind

$span_{comment}$ = *Span* $start_1\ end_1$ = *Span (Position 1 1) (Position 2 25),*
$span_{newtype}$ = *Span* $start_2\ end_2$ = *Span (Position 3 1) (Position 4 15),*

$rowDist(end_1, start_2) = 1 \geq 0$
und somit $vertDist(span_{comment}, span_{newtype}) = rowDist(end_1, start_2) = 1.$

Beispiel 4.2.2 (Abstand von zwei überlappenden Spans)
Angenommen wir wollen den Kommentar im folgenden Code der Deklaration zuordnen.

```
newtype MyType =
{- | This comment is overlapping -} ACons Int
  deriving Show
```

Dann sind

$span_{comment}$ = *Span* $start_1$ end_1 = *Span (Position 2 1) (Position 2 35),*
$span_{newtype}$ = *Span* $start_2$ end_2 = *Span (Position 1 1) (Position 3 15),*
$rowDist(end_1, start_2) = -1 \ngeq 0$ *sowie* $end_1 \leq end_2$
und somit $vertDist(span_{comment}, span_{newtype}) = 0.$

Beim Zuordnen steigen wir in den geordneten Listen der Kommentare und Deklarationen gleichzeitig ab. Dabei führen wir zusätzlich die vorherige Deklaration als Argument mit.

- Für *Pre*-Kommentare mit einem Vertikalen Abstand ≥ 0 zur Deklaration erfolgt eine Zuordnung.
- Für *Post*-Kommentare mit einem vertikalen Abstand ≤ 0 zur Deklaration erfolgt nicht unbedingt eine Zuordnung. Es könnte ja auch noch eine Deklaration zwischen dem Kommentar und der betrachteten Deklaration stehen. Deshalb muss zusätzlich sichergestellt werden, dass der Kommentar noch vor der nächsten Deklaration steht. Deshalb erfolgt die Zuordnung eines *Post*-Kommentars zur vorherigen Deklaration, sobald der vertikale Abstand zur betrachteten Deklaration größer als Null ist.
- Für *Post*-Kommentare mit einem Vertikalen Abstand von Null zur Deklaration erfolgt eine Zuordnung.
- Wenn der Abstand für *Pre* oder *Post* Kommentare nicht in eine dieser Kateogrien passt, so wird die Deklaration übersprungen.
- *None* Kommentare werden grundsätzlich übersprungen.

Wenn die Zuordnung für einen Kommentar erfolgt, dann werden in demselben Schritt auch alle nachfolgenden Kommentare desselben Typs und alle *None*-Kommentare bis zu einer Zeile ohne einen dieser Kommentare mitverarbeitet. Die exakte Paarung innerhalb einer Deklaration erfolgt für Typsignaturen, Datentypen und Klassen unterschiedlich. Dabei werden jedoch immer ähnliche Schritte angewendet:

1. Als erstes werden alle Kommentare übersprungen, die nicht zu dem gewünschten Teil der Deklaration gehören. Beispielsweise könnte ein *Post*-Kommentar zwischen

zwei Argumenten einer Konstruktordeklaration stehen. In Haddock führt ein solcher Kommentar zu einer Fehlermeldung, in CurryDoc soll dieser Kommentar nur übersprungen werden.

2. Dann werden mit einer Funktion alle Kommentare (auch *None*-Kommentare) bis zum Start der nächsten (Teil-)Deklaration herausgesucht. Dabei wird als weiteres Argument der Span des zuletzt herausgesuchten Kommentars benötigt, um festzustellen, ob ein *None*-Kommentar nah genug an dem vorherigen Kommentar ist. Da nicht gleichzeitig *Pre*- und *Post*-Kommentare verarbeitet werden, bekommt die Funktion auch ein Prädikat übergeben, mit welchem der korrekte Kommentartyp festgestellt werden kann.

3. Zuletzt werden die passenden Kommentare zusammen mit Informationen zur eindeutigen Identifikation der zugeordneten Deklaration in einem Datentyp abgelegt.

Durch die getrennte Verarbeitung der *Pre*- und *Post*-Kommentare müssen am Ende alle Kommentare für dieselbe Deklaration noch verschmolzen werden. Glücklicherweise stehen die Zuordnungen in der Ergebnisliste in der ursprünglichen Reihenfolge der Kommentare. Deshalb ist dieser Schritt trivial. Nur bei Deklarationen, die innerhalb von Klassendeklarationen stehen, muss eine aufwändige Verschmelzung erfolgen. Dies trifft auch auf andere Deklarationen zu, die nicht auf Top-Level sind. Instanzdeklarationen sind aber bisher nicht dokumentierbar, da die Dokumentation bei der Typklasse erfolgen sollte.

Zuletzt werden noch ein paar Vereinfachungen an den Daten vorgenommen: Typsignaturen und externe Funktionsdeklarationen können sich auf mehrere Bezeichner beziehen. Deshalb wird für diese Deklarationen bei der Zuordnung eine Liste von Bezeichnern gespeichert. Als Vorbereitung für die nächsten Verarbeitungsschritte wird zur Vereinfachung eine Aufteilung in mehrere Zuordnungen mit denselben Kommentaren und jeweils nur einem Bezeichner vorgenommen. Zudem werden mit CurryDoc nicht dokumentierbare Deklarationen aus der Zuordnung entfernt und externe Deklarationen zu einfachen Datentyp oder Funktionsdefinitionen geändert. In dem Analyseschritt (siehe Unterabschnitt 4.2.3) müssen die Informationen über externe Funktionsdeklarationen ohnehin hinzugefügt werden. Externe Datentypen werden bei der Erweiterung um Informationen aus AbstractCurry (siehe Unterabschnitt 4.2.2) unterschieden.

Modulkommentare

Die Kommentare für das Modul sind alle Kommentare die nach dem oben erklärten Verfahren dem Modulkopf (`module A (f1, ..., fn) where`) zugeordnet wurden. Dabei werden alle Kommentare ignoriert, die sich mit dem Modulkopf überlappen. Eine Weiterverarbeitung erfolgt dann in einer späteren Phase (siehe Unterabschnitt 4.2.4).

Dokumentationsreihenfolge und Überschriften

Die Dokumentationsreihenfolge entspricht der Reihenfolge der Deklarationen in der Exportliste. Falls keine existiert, so ordnet Haddock die Deklarationen nach der Reihenfolge im Quellcode. Dies gestaltet sich in CurryDoc als schwierig, da die Reihenfolge der Deklarationen in AbstractCurry nicht enthalten ist. Deshalb werden die Deklarationen in diesem Fall nach Funktions-/Daten- und Klassendeklarationen angeordnet. Um Überschriften für mehrere Deklarationen in der späteren Dokumentation zu erstellen, kann der Benutzer diese im Quellcode mithilfe von Kommentaren der Form `-- *` in der Exportliste angeben. Die Anzahl der `*` gibt dabei das Level der Überschrift (Abschnitt, Unterabschnitt, ...) an. Diese Überschrift zählt für alle nachfolgenden Deklarationen oder bis eine weitere Überschrift mit demselben oder niedrigerem Level (d.h. weniger `*`) folgt. CurryDoc erstellt aus dieser Exportstruktur später automatisch auch eine Navigationsleiste. Diese wird bisher nur in HTML-Dokumentationen eingeblendet und nicht in der LaTeX-Dokumentation. Letztere wird hauptsächlich für die Handbücher der Curry-Compiler genutzt und es ist zur Zeit noch nicht klar, ob diese eine solche Navigation haben sollen.

Da später die Deklarationen zur einfachen Verarbeitung in die Exportstruktur eingefügt werden, nutzen wir einen Datentyp, der polymorph über einer Typvariablen ist und eine Baumstruktur besitzt:

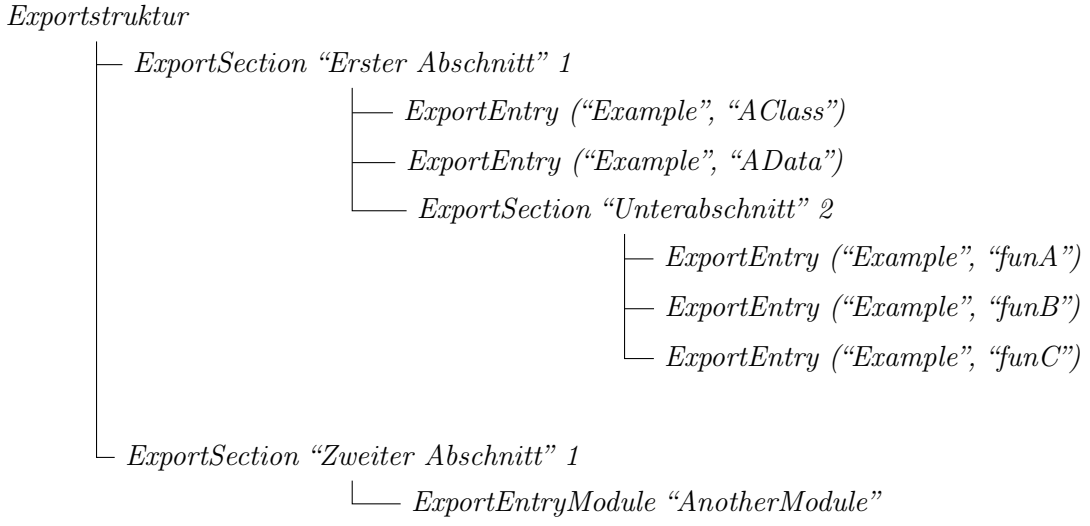
```
data ExportEntry a = ExportEntry a
                   | ExportEntryModule MName
                   | ExportSection Comment Int [ExportEntry a]
```

In dieser Phase wird eine Struktur der Form `[ExportEntry QName]` erzeugt, da an dieser Stelle nur die qualifizierten Namen (`QName`) bekannt sind.

Beispiel 4.2.3 (Exportstruktur eines Moduls)
Wir betrachten den folgenden Modulkopf.

```
module Example (
   -- * Erster Abschnitt
   AClass(..), AData(..),
   -- ** Unterabschnitt
   funA, funB, funC,
   -- * Zweiter Abschnitt
   module AnotherModule)
```

Für dieses Modul wird folgende Exportstruktur angelegt:



Das Ergebnis dieser Phase sind Zuordnungen $Deklaration \mapsto Kommentare$ sowie die Exportstruktur und Modulkommentare. Die Zuordnungen enthalten jedoch nur diejenigen Deklarationen, die mindestens einen Kommentar zugeordnet bekommen haben. [2]

4.2.2. Anreicherung mit Informationen aus AbstractCurry

In der fertigen Dokumentation sollen ausschließlich alle exportierten Funktionen, Datentypen, Konstruktoren, Felder und Klassen auftauchen. Um dies zu erreichen, orientiert sich diese Phase an der Struktur von AbstractCurry, da dort an allen exportierbaren Entitäten steht, ob diese `Public` (also exportiert) oder `Private` (nicht exportiert) sind. Für jede exportiere AbstractCurry-Deklaration versuchen wir dann, eine passende kommentierte Deklaration aus der vorherigen Phase zu finden. Abgesehen von z.B. Importen, oder Deklarationen sind alle Bezeichner im AST vollständig qualifiziert. Da in AbstractCurry jedoch ausnahmslos alle Bezeichner qualifiziert sind, muss beim Vergleich darauf geachtet werden, dass der qualifizierte Teil nur verglichen wird, wenn beide Bezeichner qualifiziert sind. Alternativ könnte man bei der Umwandlung des AST in die internen Datenstrukturen auch die fehlende Qualifizierung vornehmen. Dies wäre aus verschiedenen Gründen jedoch in früheren Stadien der Entwicklung nicht möglich gewesen, weshalb der erste Ansatz gewählt wurde.

Falls eine passende Deklaration unter den kommentierten Deklarationen existiert, so wird diese um später benötigte Informationen angereichert. Wenn keine gefunden wurde, dann

[2] Genau genommen kann es auch sein, dass eine Deklaration ohne Kommentare dort auftaucht, wenn diese bei der groben Zuordnung einen Kommentar bekommen hat, welcher später wieder verworfen wurde.

wird eine unkommentierte Deklaration angelegt und ebenso um die folgenden Informationen ergänzt:

- Bei Funktionen wird nach einer kommentierten Typsignatur gesucht und diese hinzugefügt und der Typ gespeichert.
- Bei Typsynonymen wird der genaue Typ mit Typariablen hinterlegt.
- Bei anderen Datentypen werden Typvariablen und alle bekannten Typklasseninstanzen abgelegt, sowie all nicht exportierten Konstruktoren und eventuell vorhandene Record-Felder aussortiert. Die übrigen Konstruktoren und Felder werden um ihre Typen ergänzt.
- Bei Klassen werden die Namen aller Superklassen gespeichert und die Funktionsdeklarationen in der Klasse werden auf dieselbe Art behandelt wie Top-Level Funktionen (siehe oben).

Externe Datentypen sind, wie bereits erwähnt, in AbstractCurry überhaupt nicht enthalten. Für diese Information wird FlatCurry benutzt, da diese dort Datendeklarationen ohne Konstruktoren sind. Es ist im Quellcode möglich, Datentypen ohne Konstruktoren zu definieren, jedoch werden diese in FlatCurry um einen Platzhalter-Konstruktor ergänzt. Somit ist eine Unterscheidung doch möglich. Zur Vereinfachung der Anreicherung mit Informationen aus AbstractCurry ergänzen wir die Darstellung in AbstractCurry zu Beginn deshalb ebenfalls um Platzhalter-Konstruktoren für alle Datentypen ohne Konstruktoren und fügen die externen Datentypen hinzu. Die neuen Konstruktoren werden als *Private* markiert. Ohne diese Ergänzungen könnten undokumentierte externe Datentypen nicht in der Dokumentation auftauchen.

Das Ergebnis ist eine Liste aller im Modul definierten und exportierten Deklarationen inklusive ihrer Kommentare, Typen, etc.

4.2.3. Anreicherung mit Analyseinformationen

Das Ziel dieser Phase ist es, weitere Informationen zu den Deklarationen zu sammeln. Dazu gehören:

- Nichtdeterministisches Verhalten
  (ob die Auswertung verschiedene Berechnungspfade nutzt)
- Indeterministisches Verhalten
  (ob eine Funktion bei erneuter Ausführung ein anderes Ergebnis erzeugen kann)
- Lösungsvollständigkeit
  (ob alle möglichen Ergebnisse von der Funktion berechnet werden können)

- Partielle Definition
  (ob die Funktion nicht für alle typkorrekten Argumente definiert ist)
- Bindungsstärke und Assoziativität
- Externe Definition
- Spezifikationen und Eigenschaften

Die Bestimmung der ersten vier Eigenschaften erfolgt über *CASS* (Curry Analysis Server System [HR13]) und nimmt unter Umständen viel Zeit in Anspruch. Deshalb kann über den Kommandozeilenparameter *–noanalysis* die Analyse über dieses System abgestellt werden. Die restlichen Eigenschaften werden dann auch weiterhin bestimmt.

Bindungsstärke und Assoziativität Diese Information ist in AbstractCurry enthalten. Auch für Konstruktoren und Felder kann der Programmierer eine Bindungsstärke und Assoziativität angeben, selbst wenn der Konstruktor nicht in Operatorschreibweise definiert wurde. Dabei ist nur wichtig, dass der Konstruktor oder das Feld überhaupt als Operator in Infix-Notation benutzt werden kann. Dies ist nur für zweistellige Funktionen/Konstruktoren oder Record-Felder mit einem Typen der Form $a \to b$ möglich.

Externe Definition Eine als extern definierte Funktion erkennt man in AbstractCurry daran, dass sie eine leere Regelliste enthält. Eine Funktion kann im Quellcode nicht ohne eine einzige Regel definiert werden. Deshalb ist dieses Kriterium ausreichend. Externe Datentypen wurden schon in der vorherigen Phase erkannt.

Spezifikationen und Eigenschaften CurryDoc hatte schon in der vorherigen Version die Möglichkeit, Funktionsspezifikationen mit einem speziellen Namen und für CurryCheck definierte Funktionseigenschaften der zugehörigen Funktion zuzuordnen. Die Spezifikationen und Eigenschaften müssen dazu hinter der eigentlichen Funktion in beliebiger Reihenfolge stehen. Eigenschaften werden anhand der Ergebnistypen `Test.EasyCheck.Prop` bzw. `Test.Prop.Prop` sowie Spezifikationen anhand der Namensendungen `'pre`, `'post` und `'spec` erkannt. Gespeichert und später dargestellt werden die η-expandierten Regeln der Funktionen. [3]

Das Ergebnis dieser Phase ist eine Liste aller im Modul definierten und exportierten Deklarationen mit den oben beschriebenen Informationen.

[3] In Curry kann bei Funktionsregeln das letzte Argument weggelassen werden, wenn es auf der Rechten Regelseite ebenfalls nur rechts außen vorkommt. In diesem Fall spricht man von eine η-Reduktion. Eine η-Expansion ist damit die Erweiterung einer Regel um die „fehlenden“ Argumente.

4.2.4. Analyse der Modulkommentare

Die Kommentare des Modulkopfs können eine Kurzbeschreibung (*Description*), eine Kategorie (*Category*), einen Autor (*Author*) und eine Version (*Version*) beinhalten. In kursiv sind die akzeptierten Feldnamen angegeben. Für keines dieser Felder wird irgendein Format benötigt. Der angegebene Wert wir nicht überprüft und einfach so übernommen. Zusätzlich kann nach diesen Feldern eine lange Beschreibung erfolgen. Die Reihenfolge ist unbedingt einzuhalten, ansonsten werden die Kommentare ab dem fehlerhaften Feld als Teil der langen Beschreibung gewertet. Ein Feld muss dabei die Form *Feldname* : *Wert* haben, wobei die Anzahl der Leerzeichen vor und hinter dem Feldnamen sowie dem Wert egal ist. Der Wert darf auch über mehrere Zeilen gehen, solange diese weiter eingerückt sind als der Feldname. Bisher wird nicht der vollständige Umfang der in Haddock verfügbaren Felder bereitgestellt. Die Menge der erkannten Felder kann jedoch auf einfache Art in CurryDoc erweitert werden. Sollten die Modulkommentare nach den Feldern noch weiteren Text beinhalten, so wird dieser als eine ausführliche Modulbeschreibung in die Dokumentation aufgenommen. Auch hier kann Markdown zur Formatierung der Ausgabe verwendet werden.

Das Ergebnis dieser Phase ist eine Liste der Feldern mit ihren Werten, sowie der ausführliche Beschreibungstext des Moduls

Beispiel 4.2.4 (Ein vollständiger Modulkommentar)

```
{-| Description: An example for module comments
    Category   : Example
    Author     : Kai-Oliver Prott
                 and another Author
    Version    : September 2018

    This text is supposed to be a **detailed** Description of this Module.
    Of course it can be formatted using markdown.
-}
```

4.2.5. Inline-Expansion der reexportierten Module

In Curry kann man alle mit einem Modul importierten Deklarationen wieder exportieren, indem man in der Exportliste das Modul angibt. Für vollständig reexportierte Module soll in der Dokumentation nur ein Verweis eingefügt werden. Aber wenn nicht alle Deklarationen des Moduls importiert wurden und damit auch nicht alle wieder exportiert werden sollen, dann ist ein einfacher Verweis für die Nutzer der Dokumentation potentiell irreführend. Deshalb soll sich CurryDoc in diesen Fällen so verhalten, als wären die importierten Deklarationen im momentanen Modul definiert worden. Der *User-Guide* von Haddock spezifiziert dasselbe Verhalten, jedoch steht dort auch, dass momentan immer nur eine Referenz auf das reexportierte Modul erzeugt wird. Um das umzusetzen, muss

der Modulexport in der Exportstruktur durch Einträge für alle damit exportierten Deklarationen ersetzt werden. Gleichzeitig werden eventuell vergebende Alias-Bezeichner für die exportierten Module aufgelöst.

Beispiel 4.2.5 (Transformation der Exportstruktur bei vollständigem Import)
Angenommen wir haben als Modul:

```
module Bar (elem, notElem, lookup, print) where
(...)
```

welches teilweise reexportiert wird von:

```
module Foo (Eq(..), Ord(..), Show(..), module Bar) where

import Bar hiding (print, lookup)
(...)
```

Die Exportstruktur von `Foo` *wird von dieser Phase dann so verändert, als wäre die Exportliste folgendermaßen gewesen:*

```
module Foo (Eq(..), Ord(..), Show(..), elem, notElem) where

import Bar hiding (print, lookup)
(...)
```

Beispiel 4.2.6 (Transformation der Exportstruktur bei vollständigem Import)
Wenn wir das Modul `Foo` *stattdessen ändern zu:*

```
module Foo (Eq(..), Ord(..), Show(..), module Bar) where

import Bar
(...)
```

wird die Exportstruktur von `Foo` *von dieser Phase nicht verändert, da* `Bar` *vollständig importiert wurde.*

Um diese Expansion vorzunehmen, müssen natürlich die exportierten Deklarationen der importierten Module bekannt sein. Deshalb lesen wir von diesen Modulen die abstrakte CurryDoc-Darstellung ein, die auch im nächsten Abschnitt (Unterabschnitt 4.2.6) gebraucht und später (Unterabschnitt 4.2.7) näher erläutert wird. Falls der Export für das aktuelle Modul ist, so wird dieser durch Einträge für alle Deklarationen des Moduls ersetzt. Wenn die *Prelude* [4] nicht explizit importiert wurde und als Export auftaucht, dann wird nur eine Referenz erzeugt, da der implizite Import der Prelude immer vollständig erfolgt. Sollte eine Deklaration in der Exportliste zusätzlich zum Export ihres

[4] Die *Prelude* ist ein Modul der Standardbibliothek, welches von jedem anderen Modul implizit importiert wird. Sie enthält einige grundlegende Deklarationen für die Arbeit mit Curry.

Moduls auftauchen, dann gibt es in der fertigen Dokumentation zwei Einträge für die Deklaration. Dieses Verhalten zeigt auch Haddock.

Das Einlesen aller importierten Module erscheint auf den ersten Blick keine gute Idee zu sein, da in der Regel nur ein Bruchteil der importierten Module wieder exportiert wird. Aufgrund der *Lazy*-Auswertung von Curry-Programmen ist dieses Problem jedoch vernachlässigbar.

Das Ergebnis dieser Phase ist eine angepasste Exportstruktur, in der Modulexporte falls nötig durch die Liste der damit exportierten Deklarationen ersetzt wurden.

4.2.6. Einfügen der Deklarationen in die Exportstruktur

Da nun alle Informationen zu den Deklarationen vorliegen, müssen diese noch entsprechend angeordnet werden. Dazu wird die Liste mit der Exportstruktur rekursiv durchlaufen und für alle `ExportEntry QName` die passende CurryDoc-Deklaration herausgesucht. Da manche der qualifizierten Bezeichner nicht auf eine Deklaration des aktuellen Moduls verweisen, werden zusätzlich eine Liste mit Zuordnungen *Modulname* $\mapsto$ AbstractCurryDoc sowie der Name des aktuellen Moduls benötigt. Das Einfügen erfolgt dann einfach durch Nachschlagen in der Liste der Deklarationen des aktuellen Moduls oder für importierte Deklarationen durch Nachschlagen in der abstrakten CurryDoc Repräsentation. Bei `ExportSections`, also Gruppierungen mit Überschrift, erfolgt ein rekursiver Abstieg in die Kindknoten. Für `ExportModule`-Einträge, also Exporte eines gesamten Moduls, muss nichts getan werden.

Das Ergebnis dieser Phase ist die Anordnung der exportierten Deklarationen eines Moduls in der Exportstruktur.

4.2.7. Zusammenführen der Informationen

Hier werden alle Informationen über das Modul zusammengetragen und zu folgender Datenstruktur zusammengefasst:

```
data CurryDoc = CurryDoc
                  MName                       -- Modulname
                  ModuleHeader                -- Verarbeitete Modulkommentare
                  [ExportEntry CurryDocDecl]  -- Exportierte Deklarationen
                  [MName]                     -- Importierte Module
```

Diese abstrakte Repräsentation der Dokumentation wird dann in eine Datei geschrieben, damit sie bei der Generierung für andere Module eingelesen werden kann. Da die Verarbeitung der Kommentare mit Markdown erst bei der Generierung der Zielformate

erfolgt, wird die `--nomarkdown` Option auch für importierte Module korrekt berücksichtigt. Auf die `--noanalysis` Option trifft dies leider nicht zu. Eine Möglichkeit wäre es, im abstrakten CurryDoc zu speichern, ob eine ausführliche Analyse erfolgt ist. Darauf wird aber bisher verzichtet.

Das Ergebnis dieser Phase ist die abstrakte Dokumentation des Moduls, die alle benötigten Informationen für die Generierung der Zielformate beinhaltet.

4.2.8. Generierung der Zielformate

Zur Erzeugung der Zielformate wird die Exportstruktur rekursiv durchlaufen und die Deklarationen werden hintereinander dargestellt. Wenn Markdown nicht über den Kommandozeilenparameter `--nomarkdown` deaktiviert wurde, werden die Kommentare mit dem *markdown*-Paket von Curry verarbeitet. Zudem werden für in einfachen Anführungsstrichen (') stehende Bezeichner Hyperlinks zur entsprechenden Stelle in der HTML-Dokumentation eingefügt. Die LaTeX-Dokumentation formatiert diese Bezeichner bisher nur als Code, die Verlinkung ist jedoch auch dort angedacht. Die Darstellung in beiden Formaten orientiert sich bisher weiterhin grob an der Darstellung in der vorherigen CurryDoc-Version.

Zudem können von CurryDoc auch Indexseiten für alle Konstruktoren, Funktionen und Typklassen eines Moduls erstellt werden. Auch eine Übersichtsseite für mehrere Module, die je einen Link auf die Dokumentation jedes Moduls enthält, ist möglich. Dies wird vor allem für Pakete genutzt. Für die vom Compiler mitgelieferte Standardbibliothek kann auch eine solche Seite erstellt werden.

In der LaTeX-Dokumentation werden einige Informationen nicht dargestellt. Dies trifft unter anderem auf alle Analyseinformationen und Dokumentation für Parameter von Funktionen zu. Dies war auch schon in vorherigen Versionen der Fall.

4.2.9. Weitere Anpassungen an CurryDoc

Am Hauptmodul von CurryDoc, welches das Einlesen der Kommandozeilenparameter und die Steuerung der Dokumentationsgenerierung übernimmt, musste nicht viel geändert werden. Die wohl größte Änderung ist, dass in den vorherigen Versionen die Dokumentation für die Abhängigkeiten eines Moduls erst nach der Erzeugung für das Hauptmodul erstellt wurden. Diese Reihenfolge wurde in der neuen Version umgedreht, da die abstrakte Dokumentation der importierten Module benötigt wird. Bei der Generierung der LaTeX-Dokumentation wurde bisher auf die Generierung für importierte Module verzichtet. Dies ist aus denselben Gründen nicht mehr möglich, da auch dort die importierten Module benötigt werden. Falls nur für das Hauptmodul die Dokumentationsdatei erstellt

werden soll, kann die Erzeugung der HTML/LaTeX-Dateien für importierte Module mit dem neuen Kommandozeilenparameter `--norecursive` unterdrückt werden. Dabei ist zu beachten, dass die abstrakte Dokumentation trotzdem erstellt wird, außer diese ist bereits vorhanden und auf dem neuesten Stand.

Bei der rekursiven Erzeugung für importierte Module kann man zudem Zeit einsparen, wenn man sich merkt für welche Module bereits die Dokumentation erzeugt wurde. Andernfalls kann es zum Beispiel passieren, dass immer wieder die Dokumentation für Module erstellt wird, die von mehreren Modulen importiert werden. Die Dokumentation würde in diesen Fällen meistens nicht wirklich neu erstellt werden, da sie bereits vorhanden und aktuell genug ist. Es hat sich aber trotzdem gezeigt, dass das Vergleichen der Zeitstempel der Dateien viel mehr Zeit benötigt als der gewählte Ansatz.

Zudem wurde die Unterstützung für HTML-Tags in CurryDoc-Kommentaren entfernt, da über diese auch beliebiger Code in die von CurryDoc erzeugte Webseite eingebettet werden konnte. Aufgrund dieser Änderung mussten auch am Markdown-Paket zwei Anpassungen vorgenommen werden. Bisher wurden mit Markdown formatierte Hyperlinks nur als solche akzeptiert, wenn sie mit „http“ beginnen. Andere Hyperlinks konnten nur über HTML-Tags eingefügt werden. Diese Beschränkung von Markdown wurde entfernt. Des Weiteren fehlte die Unterstützung für Inline-Quelltext, der Backticks (‘) als Zeichen enthält, da nur Text innerhalb von zwei Backticks als Code interpretiert wurde. Die Syntax wurde dahingehend erweitert, dass man Inline-Code nun mit beliebig vielen Backticks einleiten kann und dann dieselbe Anzahl zum Schließen benötigt. Dies ermöglicht die Nutzung von ‘ als Code, zum Beispiel wenn in Curry eine Funktion in Infix-Schreibweise genutzt wird, da diese dann selber zwischen Backticks stehen muss.

Des Weiteren wurde CurryDoc, wie bereits erwähnt, um den Parameter `--noanalysis` erweitert, um die Dokumentationserzeugung etwas zu beschleunigen.

Als weitere Beschleunigung der Dokumentationsgenerierung ist zudem geplant, dass die Dokumentation direkt aus der in einer Datei gespeicherten abstrakten CurryDoc Repräsentation erzeugt wird, falls die Datei aktuell genug ist. Dies würde vor allem bei der Erzeugung von mehreren Dokumentationsformaten helfen, da bisher für jedes Zielformat alle Phasen von CurryDoc durchlaufen werden.

5. Abschlussbetrachtungen

In diesem Kapitel werden die erzielten Ergebnisse noch einmal zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

5.1. Fazit

Im Rahmen dieser Arbeit wurde CurryDoc um die Unterstützung für Typklassen erweitert und gleichzeitig die Syntax für die Dokumentation an Haddock angeglichen. Des Weiteren wurden nun auch die Unterstützung für Exporte von Entitäten aus anderen Modulen hinzugefügt und kleinere Anpassungen am Markdown vorgenommen.

Die für den neuen Implementierungsansatz von CurryDoc notwendige Implementierung von allgemeinen Span-Informationen im Curry-Frontend ermöglicht auch die (Weiter-) Entwicklung anderer Werkzeuge. Dabei wurde darauf geachtet, dass die Struktur des Parsers sich nicht verändert, jedoch hat trotzdem die Übersichtlichkeit des Codes etwas gelitten. Um dem entgegenzuwirken, wurden die bestehenden Parserkombinatoren erweitert.

Die Anpassungen an CurryDoc ermöglichen ein einfaches Hinzufügen von neuen Zielformaten und die Aktualisierung bereits vorhandener. Dies wurde vor allem durch die Aufteilung in mehrere Phasen sowie in ein Frontend und mehrere Backends erreicht.

5.2. Weiterführende Arbeiten

5.2.1. Span Informationen

Auf Basis der Span-Informationen sind mehrere Arbeiten denkbar:

- Für Curry existiert bereits ein Style-Checker, welcher jedoch einen anderen Ansatz verfolgt, um die exakten Positionen von Einheiten im Quellcode zu rekonstruieren [Rah16]. Dieser könnte auf Basis der Span-Informationen angepasst und um weitere Überprüfungen ergänzt werden.

- Eine exakte Rekonstruktion des Quellcodes aus dem AST ließe sich nun auch umsetzen. Dies könnte zur allgemeinen Fehlersuche im Compiler und zur Überprüfung der Korrektheit der im Parser konstruierten Span-Informationen nützlich sein.

- Die Positionsangaben der vom Curry-Frontend erzeugten Warnungen und Fehlermeldungen sind häufig sehr ungenau. Durch die genaueren Positionsinformationen könnten diese verbessert werden. Vorstellbar wäre es auch, den fehlerhaften Teil des Quellcodes in der Konsole unterstrichen auszugeben, um bei der Fehlersuche zu helfen. Dies macht zum Beispiel der GHC (Glasgow-Haskell-Compiler).

5.2.2. CurryDoc

Manches der in Haddock vorhandenen Funktionalität wird in CurryDoc nicht umgesetzt. Dazu zählt zum Beispiel die Unterstürzung für Dokumentationstext, der nicht zu einer speziellen Deklaration gehört. Zudem sollte die generierte LaTeX-Dokumentation um einige Funktionen (Referenzen für Bezeichner, etc.) erweitert werden.

Auch schon in der vorherigen Version kann es vorkommen, dass ein in einem Kommentar vorhandener Verweis auf einen Bezeichner nicht eindeutig ist. Da sich zum Beispiel Typ- und Wertekonstruktoren nicht denselben Namensraum teilen, existieren manchmal Konstruktoren, die sich ihren Namen mit einem Typkonstruktor teilen. Dasselbe Problem kann auch mit Funktionen oder Typklassen auftreten und ist auch in Haddock nicht vollständig gelöst [1]. Ein ähnliches Problem ergibt sich auch für die Referenzen auf Teile des Quellcodes. Zur Behebung könnte man die Referenzen mit einem Präfix versehen (z.B. `t:Tree` falls der Typ Tree gemeint ist und `c:Tree` für den Konstruktor). Dafür muss dann auch die vom Curry-Frontend generierte HTML-Darstellung des Quellcodes entsprechend verändert werden. Damit der Programmierer nicht immer diesen Präfix an die Bezeichner in Kommentaren schreiben muss, sollte CurryDoc bei fehlendem Präfix versuchen, die gemeinte Art selber zu bestimmen. Bei Mehrdeutigkeiten könnte dann eine Warnung ausgegeben werden.

Die neu hinzugekommene Unterstützung für Reexportierte Module beachtet nicht, ob der Import eines Datentyps oder einer Typklasse über die explizite Angabe der importierten Konstruktoren oder Typklassenfunktionen eingeschränkt wurde. Ein Ansatz zur Lösung dieses Problems ist bereits implementiert, jedoch funktioniert dieser noch nicht.

Eine vollständige Auflistung der bekannten Fehler und Einschränkungen befindet sich in Anhang C.

[1] siehe https://github.com/haskell/haddock/issues/667.

6. Literaturverzeichnis

[Han01] M. Hanus. High-level server side web scripting in curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[HO17] M. Hanus and J. Oberschweiber. Cpm: A declarative package manager with semantic versioning. In *Pre-Proceedings of the Conference on Declarative Programming (Declare 2017)*, pages 231–241. Technical Report 499, University of Würzburg, 2017.

[HR13] M. Hanus and F. Reck. A generic analysis server system for functional logic programs. In *Proc. of the 13th International Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS 2013)*, pages 1–16, 2013.

[Rah16] Katharina Rahf. Überprüfung von Stilrichtlinien für deklarative Programme. Master's thesis, Christian-Albrechts-Universität zu Kiel, 2016.

[Tee16] Finn Teegen. Erweiterung von Curry um Typklassen und Typkonstruktorklassen. Master's thesis, Christian-Albrechts-Universität zu Kiel, 2016.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

A. Quellcode und Beispiele

Der Quellcode von CurryDoc ist im öffentlichen Git-Repository unter
https://git.ps.informatik.uni-kiel.de/curry-packages/currydoc
in dem Branch „`currydoc2.0`“ zu finden. Dort befinden sich auch zwei Module mit Beispielen für jede dokumentierbare Deklaration und den Modulkopf.

Die Änderungen am Curry-Frontend teilen sich auf die beiden Pakete
curry-base (https://git.ps.informatik.uni-kiel.de/curry/curry-base) sowie
curry-frontend (https://git.ps.informatik.uni-kiel.de/curry/curry-frontend)
auf und sind jeweils im Branch „`SpanInfo`“ zu finden.

In dem Branch „`code_markdown`“ unter
https://git.ps.informatik.uni-kiel.de/curry-packages/markdown
befinden sich die Änderungen am Markdown-Paket.

B. Neue CurryDoc-Syntax

B.1. Modulkopf

Alle Dokumentationskommentare, die zur Beschreibung des Moduls genutzt werden sollen, müssen entweder als *Pre*-Kommentare vollständig vor dem Modulkopf oder als *Post*-Kommentare vollständig danach erfolgen (aber noch vor der ersten Deklaration).

Zu Beginn der zugeordneten Kommentare können die folgenden Felder dokumentiert werden: Description, Category, Author, Version. Alle Felder sind optional, die Reihenfolge muss jedoch eingehalten werden. Das Format ist dabei "*Feldname : Feldwert*", wobei der Feldwert über mehrere Zeilen gehen kann, so lange diese weiter eingerückt sind als der Feldname.

Alle Kommentare nach dem letzten Feld werden als langer Beschreibungstext in die Dokumentation aufgenommen.

B.2. Exportliste

Innerhalb der Exportliste können Abschnittsüberschriften mit Kommentaren der Form `-- *` eingefügt werden. Die Anzahl der `*` legt dabei die Art der Überschrift(Abschnitt, Unterabschnitt, ... fest. Die Abschnitte dürfen erst nach der öffnenden Klammer der Exportliste erfolgen und ein Abschnitt gilt für alle in der Liste nachfolgenden Bezeichner bis zu einem Abschnitt mit weniger oder gleicher Anzahl an `*`.

B.3. Deklarationen

Nur die aufgeführten Deklarationen können dokumentiert werden. Kommentare für andere Deklarationen werden nicht berücksichtigt. Für Instanzdeklarationen existiert bereits die Zuordnung analog zu Klassen, die Kommentare werden jedoch nicht in der Dokumentation angezeigt.

data-Deklarationen und newtype-Deklarationen

Kommentare für die gesamte Datentypdeklaration müssen entweder vor dem `data`-Schlüsselwort oder nach der vollständigen `deriving` Klausel stehen.

Kommentare für Konstruktoren können vor dem Namen des Kostruktors oder nach allen Argumenten des Konstruktors erfolgen. Bei Konstruktoren in Operatorschreibweise müssen die Kommentare vor dem ersten Argument oder nach dem letzten Argument erfolgen. Bei Records werden Kommentare innerhalb der geschweiften Klammern einem Feld zugeordnet, wenn sie vor dem Feldnamen oder nach dem Feldtypen erfolgen.

Bei externen Datentypen müssen die Kommentare vor dem `external`-Schlüsselwort oder nach der vollständigen Deklaration stehen.

Typsynonyme

Die Kommentare müssen vor dem `type`-Schlüsselwort oder nach dem kompletten Typsynonym erfolgen.

Funktionsdeklarationen und Typsignaturen

Kommentare für die gesamte Funktion müssen entweder vor der Typsignatur, vor der ersten Regel der Funktion oder nach der letzten Regel der Funktion stehen. Dabei werden Kommentare vor der Typsignatur in der Dokumentation grundsätzlich vor den Kommentaren der Funktion angezeigt, auch wenn die Typsignatur hinter der Funktion steht.

Argumente können innerhalb der Typsignatur dokumentiert werden. Dazu müssen sie entweder vor dem Argumenttypen oder dahinter stehen. Die Kommentare dürfen auch vor dem `->` oder `::` stehen und werden trotzdem dem nächsten Argument zugeordnet.

Für externe Funktionen dürfen die Kommentare unverändert in der Typsignatur, vor den Bezeichnern der `external`-Deklaration oder danach stehen.

Klassendeklarationen

Die allgemeinen Kommentare zu einer Klasse müssen entweder vor dem *class* oder nach dem *where*-Schlüsselwort, aber vor der ersten Deklaration in der Klasse, erfolgen.

Die Deklarationen innerhalb der Klasse werden wie Deklarationen auf Toplevel behandelt.

C. Bekannte Einschränkungen

- Die Span-Informationen für *FunctionalPatterns* sind nach dem Syntax-Check inkonsistent. Hier muss noch eine Lösung gefunden werden.
- Die Span-Informationen unterstützen kein explizites Layout. Dies ist aber in der Regel nicht relevant.
- Das AST-Zielformat des Frontends ist für viele Module zu groß, um vom *PAKCS* eingelesen zu werden. Eine effizientere Implementierung der automatisch erstellten *Read*-Instanzen für Datentypen im Frontend ist angedacht und würde hier auch helfen. Alternativ sollte über eine individuell erstellte Read-Instanz für den AST nachgedacht werden.
- Die Generierung des *CDOC*-Formats wird nicht unterstützt. *CDOC* wird von dem Tool *curr(y)gle* genutzt, welches Zeitgleich zu CurryDoc für Typklassen angepasst wird. Ob und in welcher Form *CDOC* weiterhin benötigt wird ist noch unklar.
- In Kommentaren referenzierte Bezeichner sind unter Umständen nicht eindeutig. Dies trifft auch auf den vom Curry-Frontend generierten HTML-Code für ein Modul zu. Eine Möglichkeit wäre es, dass der Benutzer über spezielle Syntax beim Referenzieren von Bezeichnern angibt, ob er einen Typ, Konstruktor, Typklasse, Feld, oder Funktion meint.
- Beim Reexportieren von Modulen wird nicht beachtet, ob der Import eines Datentyps oder einer Typklasse über die explizite Angabe der importierten Konstruktoren oder Typklassenfunktionen eingeschränkt wurde.
- Wenn nur die Funktion einer Typklasse exportiert wird, nicht jedoch die Typklasse selber, dann taucht die Funktion nicht in der Dokumentation auf.
- Die Dokumentationsreihenfolge bei fehlender Exportliste weicht von Haddock ab.
- Für existenziell quantifizierte Typen wird auch weiterhin der Kontext und die quantifizierten Typvariablen nicht angezeigt. Dies sollte einfach hinzuzufügen sein.
- Wenn CurryDoc mit dem PAKCS kompiliert wird, dann kann es bei großen Paketen vorkommen, dass die Generierung der Dokumentation aufgrund zu vieler geöffneter Dateien fehlschlägt.

- Die Einrückung der Felder von Modulkommentaren wird nur über die Anzahl der Leerzeichen bestimmt. Dies kann unerwartete Konsequenzen haben, z.B. wenn die Felder in derselben Zeile wie der Kommentar begonnen werden.

D. Übersicht über die srcInfoPoints

Hier ist eine Auflistung der möglichen srcInfoPoints für jeden Datentypen im AST. Wenn ein Schlüsselwort oder eine Zeichenfolge aus dem Quellcode gemeint ist, so sind diese *kursiv* geschrieben.

`module`	1. *module* (falls Modulkopf vorhanden) 2. *where* (falls Modulkopf vorhanden)
`LanguagePragma`	1. *{-# LANGUAGE* 2. Alle Kommas zwischen den Erweiterungsnamen 3. *#-}*
`OptionsPragma`	1. *{-# OPTIONS* 2. *#-}*
`Exporting`	1. *(* 2. Alle Kommas zwischen den Bezeichnern 3. *)*
`Export`	
`ExportTypeWith`	1. *(* 2. Alle Kommas zwischen den Bezeichnern 3. *)*
`ExportTypeAll`	1. *(* 2. *..* 3. *)*
`ExportModule`	1. *module*
`ImportDecl`	1. *import* 2. *qualified* (falls vorhanden) 3. *as* (falls vorhanden)

`Importing`	1. *(* 2. Alle Kommas zwischen den Bezeichnern 3. *)*
`Hiding`	1. *hiding* 2. *(* 3. Alle Kommas zwischen den Bezeichnern 4. *)*
`Import`	
`ImportTypeWith`	1. *(* 2. Alle Kommas zwischen den Bezeichnern 3. *)*
`ImportTypeAll`	1. *(* 2. *..* 3. *)*
`InfixDecl`	1. *infix*, *infixl* oder *infixr* 2. Bindungsstärke (falls vorhanden) 3. Alle Kommas zwischen den Bezeichnern
`DataDecl`	1. *data* 2. = (falls Konstruktoren vorhanden) 3. Alle | zwischen den Konstruktoren 4. *deriving* (falls vorhanden) 5. *(* (falls vorhanden) 6. Alle Kommas zwischen den Typklassenbezeichnern 7. *)* (falls vorhanden)
`ExternalDataDecl`	1. *external* 2. *data*
`NewtypeDecl`	1. *newtype* 2. = 3. *deriving* (falls vorhanden) 4. *(* (falls vorhanden) 5. Alle Kommas zwischen den Typklassenbezeichnern 6. *)* (falls vorhanden)

`TypeDecl`	1. *type* 2. =
`TypeSig`	1. Alle Kommas zwischen den Funktionsbezeichnern 2. *::*
`FunctionDecl`	
`ExternalDecl`	1. Alle Kommas zwischen den Funktionsbezeichnern 2. *external*
`PatternDecl`	
`FreeDecl`	1. Alle Kommas zwischen den Variablenbezeichnern 2. *free*
`DefaultDecl`	1. *default* 2. *(* 3. Alle Kommas zwischen den Typen 4. *)*
`ClassDecl`	1. *class* 2. *(* (Falls im Kontext vorhanden) 3. Alle Kommas zwischen den Typenklassenbezeichnern im Kontext 4. *)* (Falls im Kontext vorhanden) 5. => (Falls Kontext vorhanden) 6. *where*
`InstanceDecl`	1. *instance* 2. *(* (Falls im Kontext vorhanden) 3. Alle Kommas zwischen den Typenklassenbezeichnern im Kontext 4. *)* (Falls im Kontext vorhanden) 5. => (Falls Kontext vorhanden) 6. *where*

`ConstrDecl`	1. *forall* (Falls vorhanden) 2. . (Falls vorhanden) 3. *(* (Falls im Kontext vorhanden) 4. Alle Kommas zwischen den Typenklassenbezeichnern im Kontext 5. *)* (Falls im Kontext vorhanden) 6. => (Falls Kontext vorhanden)
`ConOpDecl`	1. *forall* (Falls vorhanden) 2. . (Falls vorhanden) 3. *(* (Falls im Kontext vorhanden) 4. Alle Kommas zwischen den Typenklassenbezeichnern im Kontext 5. *)* (Falls im Kontext vorhanden) 6. => (Falls Kontext vorhanden)
`RecordDecl`	1. *forall* (Falls vorhanden) 2. . (Falls vorhanden) 3. *(* (Falls im Kontext vorhanden) 4. Alle Kommas zwischen den Typenklassenbezeichnern im Kontext 5. *)* (Falls im Kontext vorhanden) 6. => (Falls Kontext vorhanden) 7. *{* 8. Alle Kommas zwischen den Feldern 9. *}*
`NewConstrDecl`	
`NewRecordDecl`	1. *{* 2. *::* 3. *}*
`FieldDecl`	1. Alle Kommas zwischen den Feldnamen 2. *::*

`ConstructorType`	Bei Unit 1. *(* 2. *)* ODER bei Funktionstypkonstruktor 1. *(* 2. =>* 3. *)* ODER bei Listtypkonstruktor 1. *[* 2. *]*
`ApplyType`	
`VariableType`	
`TupleType`	1. *(* 2. Alle Kommas zwischen den Typen 3. *)*
`ListType`	1. *[* 2. *]*
`ArrowType`	1. ->
`ParenType`	1. *(* 2. *)*
`ForallType`	Nicht möglich
`QualTypeExpr`	1. *(* (Falls im Kontext vorhanden) 2. Alle Kommas zwischen den Typenklassenbezeichnern im Kontext 3. *)* (Falls im Kontext vorhanden) 4. => (Falls Kontext vorhanden)
`Constraint`	1. *(* (Falls Komplexer Typ mit Klammern) 2. *)* (Falls Komplexer Typ mit Klammern)
`Equation`	
`FunLhs`	

`OpLhs`	1. ‘ (Falls vorhanden) 2. ‘ (Falls vorhanden)
`SimpleRhs`	1. = 2. *where* (Falls vorhanden)
`GuardedRhs`	1. *where* (Falls vorhanden)
`LiteralPattern`	
`NegativePattern`	1. Das Literal
`VariablePattern`	
`ConstructorPattern`	Falls List-Cons oder Tuple-Cons 1. *(* 2. *)*
`InfixPattern`	
`InfixPattern`	1. ‘ (Falls vorhanden) 2. ‘ (Falls vorhanden)
`ParenPattern`	1. *(* 2. *)*
`RecordPattern`	1. *{* 2. Alle Kommas zwischen den Feldern 3. *}*
`RecordPattern`	1. *{* 2. Alle Kommas zwischen den Feldern 3. *}*
`TuplePattern`	1. *{* 2. Alle Kommas zwischen den Werten 3. *}*
`ListPattern`	1. *[* 2. Alle Kommas zwischen den Werten 3. *]*

`AsPattern`	1. *@*
`LazyPattern`	1. *~*
`FunctionPattern`	siehe Anhang C
`InfixFuncPattern`	siehe Anhang C
`Literal`	
`Variable`	
`Constructor`	Bei Unit 1. *(* 2. *)* ODER bei Tupelkonstruktor 1. *(* 2. Alle Kommas 3. *)*
`Paren`	1. *(* 2. *)*
`Typed`	1. *::*
`Record`	1. *{* 2. Alle Kommas zwischen den Feldern 3. *}*
`RecordUpdate`	1. *{* 2. Alle Kommas zwischen den Feldern 3. *}*
`Tuple`	1. *(* 2. Alle Kommas zwischen den Werten 3. *)*
`List`	1. *[* 2. Alle Kommas zwischen den Werten 3. *]*

`ListCompr`	1. *[* 2. *|* 3. Alle Kommas zwischen den Statements 4. *]*
`EnumFrom`	1. *[* 2. *..* 3. *]*
`EnumFromTo`	1. *[* 2. *..* 3. *]*
`EnumFromThen`	1. *[* 2. *,* 3. *..* 4. *]*
`EnumFromThenTo`	1. *[* 2. *,* 3. *..* 4. *]*
`UnaryMinus`	1. -
`Apply`	
`InfixApply`	
`LeftSection`	1. *(* 2. *)*
`RightSection`	1. *(* 2. *)*
`Lambda`	1. \ 2. ->
`Let`	1. *let* 2. *in*

`Do`	1. *do*
`IfThenElse`	1. *if* 2. *then* 3. *else*
`Case`	1. *case* 2. *of*
`StmtExpr`	
`StmtDecl`	1. *let*
`StmtBind`	1. <-
`Alt`	
`Field`	1. =
`Goal`	1. *where* (falls vorhanden)
`ModuleIdent`	1. Der Span des gesamten Modulbezeichners
`Ident`	Falls geklammerter Operator: 1. *(* 2. Operatorname 3. *)* Falls Funktion in Infix-Schreibweise: 1. ‘ 2. Funktionsname 3. ‘ Sonst: 1. Funktions-/Operatorname
`qIdent`	1. Der gesamte Bezeichner