Infrastruktur zur Verwaltung von Analyseinformation für Curry-Pakete

Dennis Thomsen

Masterarbeit
Dezember 2024

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

Betreut durch
Prof. Dr. Michael Hanus
M.Sc. Kai Prott

Erklärung zur selbstständigen Erstellung

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass ich die Arbeit in keinem anderen Prüfungsverfahren eingereicht habe.

Weiterhin versichere ich, dass die eingereichte schriftliche Fassung der Arbeit der auf dem elektronischen Speichermedium gespeicherten Fassung und der digitalen Fassung entspricht.

Ort, Datum	Unterschrift

Zusammenfassung

Über die Zeit wurden für die Programmiersprache Curry diverse Anwendungen entwickelt, die entweder nützliche Informationen über Curry Programme generieren oder solche Informationen verwenden. Diese Informationen sind aber in unterschiedlichen Formaten, die von Anwendung zu Anwendung unterschiedlich sein können, und diese Informationen dann zu verwenden ist schwierig. Wünschenswert wäre daher eine Möglichkeit zu haben diese Informationen zwischen Anwendungen teilen zu können. Dies ist das Ziel dieser Arbeit. Es wird eine Anwendung namens CurryInfo entwickelt, welche die Informationen von anderen Anwendungen sammelt und anderen Anwendungen zur Verfügung stellt. Durch Anpassung bestehender Anwendungen kann man die Informationen einfacher verwenden.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Curry . . . 3
2.2 JSON . . . 10
2.3 CPM . . . 15
2.4 CASS . . . 17
2.5 curry-calltypes . . . 19
2.6 curry-interface . . . 21

3 CurryInfo - Aufgabe **22**
3.1 Motivation . . . 22
3.2 Ziel . . . 24

4 CurryInfo - Architektur und Aufbau **27**
4.1 Idee . . . 27
4.2 Architektur . . . 28
4.3 Typen . . . 28
4.4 Commands . . . 31
4.5 Checkout . . . 33
4.6 Analysis . . . 34
4.7 Interface . . . 35
4.8 Generator . . . 37
4.9 Printer . . . 40
4.10 Reader und Writer . . . 41
4.11 Configuration . . . 42

5 CurryInfo - Anwendung **46**
5.1 Terminal-Modus . . . 46
5.2 Servermodus . . . 49

6 CurryInfo - Anwendung in anderen Anwendungen **52**

7 Fazit **54**

Literatur **55**

A Bedienungsanleitung **56**

B Hinzufügen neuer Anfragen **59**

C JSON Schema Spezifikationen **62**
C.1 Paket . . . 62
C.2 Version . . . 62
C.3 Modul . . . 63

C.4 Typ 65
C.5 Typklasse 66
C.6 Operation 67

1 Einleitung

Mittlerweile gibt es mehrere Anwendungen, die nützliche Informationen für Curry Programme bestimmen. Allen voran existiert CASS, eine Analyseanwendung mit einer erweiterbaren Menge an Analysen. So kann man zum Beispiel bestimmen, ob eine Operation deterministisch ist oder ob sie immer terminiert. Solche Informationen können nützlich für das Arbeiten mit Curry sein.

Weiterhin existiert *curry-interface*, welches das Interface eines Curry Programms parsen kann und so Zugriff auf die Bestandteile bietet. So kann man zum Beispiel die Liste der Methoden einer Typklasse oder den Typ einer Operation erfahren.

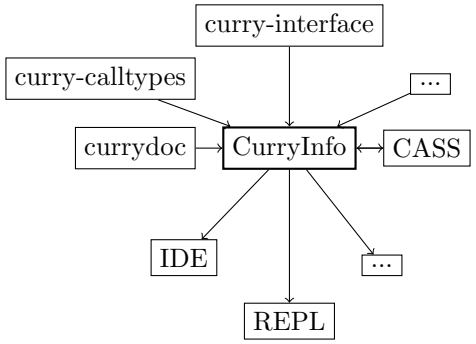
Das sind nur einige Beispiele an Anwendungen und ihren Informationen, die nützlich für die Programmierung sein können. Man kann aber bereits mit diesen erkennen, dass andere Anwendungen mit diesen verbessert und erweitert werden könnten. Zum Beispiel in einer IDE können Analyseergebnisse von CASS nützlich beim Programmieren sein.

Zum jetzigen Zeitpunkt fehlt aber noch eine Möglichkeit diese Information einfach zwischen Anwendungen zu teilen. Anwendungen haben meist eigene Regeln für das Format ihrer Ausgaben. Diese Formate haben auch das Risiko in Zukunft geändert zu werden. Dies würde zu Schwierigkeiten führen, wenn mehrere Anwendungen auf die Einhaltung dieser Ausgaben angewiesen wären.

Daher ist ein Ziel dieser Arbeit diese Möglichkeit zu erstellen. Dies wird in Form einer neuen Anwendung gemacht, die Informationen von anderen Anwendungen sammeln kann und anderen Anwendungen Zugriff auf diese gibt.

Hierbei ist aber auch die Erweiterbarkeit dieser neuen Anwendung wichtig. Da in Zukunft neue Anwendungen entwickelt werden könnten, die man ebenfalls mit dieser verbinden will, so muss dies möglichst einfach gestaltet werden.

Die grundlegende Idee für die neue Anwendung ist diese wie eine Art Cache für andere Anwendungen zu verwenden. Die Informationen werden dort abgespeichert und die Anwendung bietet eine Schnittstelle an um diese Informationen auszulesen. Um nicht die bestehenden Anwendungen mehr als nötig zu verändern, soll diese neue Anwendung auch die Generierung automatisch ausführen können. So müssen nur Anfragen gestellt werden.



In dieser Grafik kann man diese zentrale Rolle der neuen Anwendung se-

hen, die CurryInfo genannt wird. Es gibt Anwendungen wie *curry-calltypes* und *curry-interface*, deren Ergebnisse von CurryInfo gespeichert werden. Auf der anderen Seite sind Anwendungen wie IDEs und REPLs, die wiederum diese Informationen auslesen und verwenden. Es können in Zukunft mehr Anwendungen folgen, die sich ebenfalls mit CurryInfo verbinden könnten.

CASS ist eine Anwendung, die sich in beide Richtungen mit CurryInfo verbinden kann. CASS kann nicht nur Analyseergebnisse in CurryInfo speicher, sondern es kann auch solche Ergebnisse wieder für Analysen verwenden.

Die Arbeit ist wie folgt strukturiert. Zunächst werden die Grundlagen vorgestellt und erklärt, welche zum Verständnis des Rest der Arbeit nötig sind. Danach wird die Motivation und das Ziel der Arbeit erklärt. Darauffolgend wird die Architektur und der Aufbau des Programms vorgestellt und erklärt. Die Anwendung und ihre zwei Modi werden danach vorgestellt. Zuletzt wird beschrieben, wie CASS geändert wurde um CurryInfo zu verwenden, und ein Fazit sowie eine Aussicht auf zukünftige Arbeiten werden vorgestellt.

2 Grundlagen

2.1 Curry

Curry[2] ist eine sogenannte logisch-funktionale Sprache. Das ist eine Kombination von zwei deklarativen Arten der Programmierung, nämlich der logischen Programmierung und der funktionalen Programmierung.

Bei logischer Programmierung definiert man eine Menge von logischen Regeln, welche eine Wissensbasis bildet. Dieser kann man Anfragen stellen und durch logische Schlussfolgerungen und Unifikation werden Antworten für diese Anfragen berechnet. Ein bekanntes Beispiel für solch eine Sprache ist Prolog.

Funktionale Programmierung ist sehr auf Funktionen fokussiert, so dass Funktionen sogar *first-class citizens* sind. Das heißt, dass Funktionen als Werte behandelt werden und entsprechend auch als Ein- und Ausgabe von anderen Funktionen verwendet werden können. Das wohl bekannteste Beispiel für diese Art der Programmierung ist Haskell, die rein-funktionale Sprache. Die Sprache ist rein-funktional und nicht nur funktional, da sie bis auf ausgewählte Ausnahmen keine Nebeneffekte hat. Funktionen berechnen also immer die gleichen Ergebnisse, solange sie auch die gleichen Argumente erhalten.

Curry kombiniert Eigenschaften beider Arten der Programmierung. Damit formt es eine logisch-funktionale Programmiersprache. Statt Funktionen hat Curry Operationen, denn logisch-funktionale Programmierung ermöglicht es nicht-deterministisch zu programmieren. Operationen können daher beliebig viele Ergebnisse haben, während Funktionen immer exakt ein Ergebnis haben. Das bedeutet vor allem, dass Curry im Gegensatz zu Haskell fähig ist mit fehlenden Ergebnissen zu rechnen, während in Haskell dies zu einem Absturz des Programms führen könnte.

Da die Grammatik von Curry fast identisch zu der von Haskell ist und Operationen im Wesentlichen eine Erweiterung von Funktionen sind, lassen sie sich mit der gleichen Syntax definieren. Die Typangabe ist optional, solange der Compiler diesen durch Typinferenz selbst bestimmen kann. Zur Übersicht werden aber in allen Beispielen in dieser Arbeit die Typangaben explizit angegeben. Zur eigentlichen Definition einer Operation siehe man sich das folgende Beispiel an.

```
foo :: Bool -> Int
foo True  = 0 ? 1
foo False = 2
```

Wie in Haskell kann man mittels *Pattern Matching* verschiedene Fälle einer Operation definieren. In diesem Fall wird anhand des Arguments entweder eine 2, wenn das Argument `False` ist, oder der Ausdruck `0 ? 1`, wenn das Argument `True` ist, zurückgegeben. Der Operator `?` ist eine Möglichkeit Nichtdeterminismus in Curry zu verwenden. Dieser wählt nicht-deterministisch eine der zwei Ausdrücke aus und gibt diesen zurück.

Wenn das Argument also `True` ist, dann ist das Ergebnis entweder 0 oder 1. Curry ist in der Lage damit umzugehen und rechnet mit beiden Werten weiter.

Welcher davon zuerst verwendet wird, ist aber nicht vorgegeben. Würde man so etwas in Haskell umsetzen wollen, so müsste man den Typ der Funktion anpassen und zum Beispiel eine Liste von Werten zurückgeben.

Curry übernimmt Unifikation und freie Variablen als Konzepte von der logischen Programmierung, was einem ermöglicht wie in Prolog Operationen zu definieren, die Werte generieren, oder auch Operationen invers aufzurufen, in dem man das Ergebnis vorgibt, aber die Argumente unbestimmt hält.

Da die Syntax der zwei Sprachen sich so stark ähneln, ist man in der Lage viele Programme einer Sprache ohne Änderungen als ein Programm der anderen Sprache zu verwenden. Dies kann aber zu Komplikationen führen, denn die Semantiken der Sprachen unterscheiden sich stellenweise drastisch. Besonders das Pattern Matching, welches bereits zu sehen war, funktioniert in Curry anders als in Haskell. Hierzu siehe man sich das folgende Beispiel an.

```
foo :: Bool -> String
foo True = "Is True"
foo _    = "Is False"

bar :: Bool -> String
bar b = case b of
    True -> "Is True"
    _ -> "Is False"
```

In Haskell würden `foo` und `bar` immer die gleichen Ergebnisse haben. Wenn das Argument `True` ist, dann ist das Ergebnis `"Is True"`. Im anderen Fall mit dem Argument `False` ist das Ergebnis `"Is False"`. In Curry ist dies aber nicht der Fall.

Die Muster im Pattern Matching werden nicht-deterministisch abgearbeitet. Während in Haskell die Muster der Reihe nach durchgegangen werden, ist die Reihenfolge in Curry unbestimmt. Stattdessen werden immer alle passenden Muster verwendet. Im Falle von `foo` sind das beide Muster, da die Wildcard `_` auf jedes Argument passt. In Curry hätte `foo` mit dem Argument `True` also beide Ergebnisse `"Is True"` und `"Is False"`. Um die gleiche Ausgabe wie in Haskell zu bekommen, müsste man den Code etwa wie folgt anpassen.

```
foo :: Bool -> String
foo True  = "Is True"
foo False = "Is False"
```

Nun kann es aber Fälle geben, in denen man einen Fall im Pattern Matching angeben will, welcher nur die nicht genannten Fälle abfängt. Dies entspreche dem Pattern Matching von Haskell. Das ist auch möglich in Curry, in dem man den `case`-Ausdruck verwendet. Dieser ermöglicht es einem Pattern Matching auch in Ausdrücken zu verwenden, statt diese nur auf Definitionsebene von Funktionen und Operationen angeben zu können. Das Pattern Matching von `case` entspricht dann dem von Haskell, es wird also nur das erste passende Pattern gewählt. Für das Beispiel könnte es so aussehen, was exakt dem Code von `bar` entspricht.

```
foo :: Bool -> String
foo b = case b of
  True -> "Is True"
  _    -> "Is False"
```

Curry Programme sind bis auf manche semantische Unterschiede exakt so aufgebaut wie Haskell Programme. Sie bestehen aus Modulen, welche Definitionen enthalten. So kann man auch eigene Typen in Curry definieren. Wie in Haskell verwendet man hier einen `data` Ausdruck. Hierbei zählt man eine Liste von Datenkonstruktoren auf, die wie Operationen Argumente nehmen und damit die verschiedenen Formen an Daten darstellen, die der Typ annehmen kann. Bei den Argumenten handelt es sich hier aber um Typparameter. Diese stehen für Typen statt für Daten. Ein formaler Aufbau einer solchen Definition sowie Beispiele folgen.

$$data\ T\ t_1\ \cdots\ t_n\ = C_1\ t_{11}\ \cdots\ t_{1k_1}\ \mid\ \cdots\ \mid\ C_m\ t_{m1}\ \cdots\ t_{mk_m}$$

```
data Peano = O | F Peano
```

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

Wie man sehen kann, muss ein neuer Typ nicht zwangsläufig Typparameter haben. Die Datenkonstruktoren müssen auch nicht alle vorhandenen Typparameter verwenden. In diesem Beispiel ist eine mögliche Darstellung von Peano Zahlen und `Maybe` als ein spezieller Typ, der auch in der Standardbibliothek von Curry existiert. Vor allem `Maybe` wird viel im Code dieser Arbeit verwendet, weswegen dieser genauer beschrieben wird.

In dieser Form muss man genau wissen, an welcher Stelle im Konstruktor das gewünschte Datum ist. Wenn die Felder unterschiedliche Typen haben, so ist dies noch einfach. Wenn Felder aber den gleichen Typ haben, so kann es zu Schwierigkeiten führen. Dafür folgt ein Beispiel.

```
data Name = Name String String
```

Auf diese Art könnte man den Namen einer Person darstellen. Es ist aber nicht ersichtlich, welcher der beiden `Strings` der Vorname und welcher der Nachname ist. Für solche Fälle gibt es eine alternative Form, mit der man Typen in Curry definieren kann. Für dieses Beispiel könnte das wie folgt aussehen.

```
data Name = Name {forename :: String, surname :: String}
```

In dieser Form haben die Felder eigene Namen, mit denen man sie adressieren kann. In Wirklichkeit sind beide Typen exakt gleich. Die Namen der Felder generieren aber automatisch Operationen, mit welchen man die Werte der Felder erhält. Die Typen dieser Operationen sind wie folgt.

```
forename :: Name -> String
surname :: Name -> String
```

Dadurch ist es auch nicht möglich zwei Typen in einem Modul zu haben, welche die gleichen Namen für Felder verwenden. Denn es würden zwei Varianten dieser Operationen generiert werden, die aber unterschiedliche Typen hätten. Dies wird nicht vom Typsystem zugelassen.

Alternativ gibt es noch zwei andere Möglichkeiten eigene Typen zu definieren. Mit `type` kann man einen Typalias definieren. Wie bei `data` gibt man den Typnamen gefolgt von einer optionalen Aufzählung von Typvariablen links vom Gleichheitszeichen. Rechts vom Gleichheitszeichen gibt man einen Typausdruck an, der auch die angegebenen Typvariablen verwenden kann. Ein Beispiel für solch eine Typdefinition und für eine Anwendung dieses Typs folgt hier.

```
type Filename = String

foo :: String -> Filename
foo s = s ++ ".data"

bar :: Filename -> String
bar f = takeWhile (/= '.') f
```

Typvariablen wurden hier weggelassen. `Filename` ist in diesem Codeausschnitt ein Alias für `String`. Wie man sehen kann, obwohl diese zwei Typen eigentlich unterschiedliche Typen sein sollten, können beide für jeweils den anderen verwendet werden. Ein Typalias ist nur ein alternativer Name für einen Typausdruck, sowohl zur Kompilierzeit als auch zur Laufzeit wird dieser aber durch den originalen Typausdruck ersetzt. In diesem Beispiel wird `Filename` also wie `String` vom Compiler behandelt.

Die andere Möglichkeit einen Typ zu definieren ist mit `newtype`. Dieser funktioniert ähnlich wie `data` mit der Einschränkung, dass auf der rechten Seite des Gleichheitszeichens nur ein Konstruktor angegeben werden kann und dass dieser nur einen Typausdruck als Bestandteil hat. Das vorherige Beispiel kann wie folgt angepasst werden.

```
newtype Filename = Filename String

foo :: String -> Filename
foo s = Filename (s ++ ".data")

bar :: Filename -> String
bar (Filename f) = takeWhile (/= '.') f
```

Die Verwendung dieses Typs ist leicht komplizierter. Da der Typ explizit nicht `String` ist, muss erst durch Pattern Matching auf das Feld zugegriffen werden beziehungsweise das Datum durch Anwendung des Konstruktors erzeugt werden, damit das Programm kompilierbar ist.

Man kann hier sehen, dass die Verwendung dieses Typs leicht komplizierter geworden ist. Man kann diesen Typ nicht mehr wie vorher einfach wie einen

`String` verwenden, stattdessen muss man wie mit `data` erst durch Pattern Matching auf das Feld zugreifen beziehungsweise den Konstruktor benutzen um das Datum zu erstellen. Es ist schwer zu erkennen, was nun der Unterschied zwischen `data` und `newtype` bis auf die Einschränkungen von `newtype` ist. Tatsächlich wird `newtype` zur Laufzeit durch den verwendeten Typausdruck ersetzt. Zur Laufzeit wird `Filename` also wieder zu `String`.

Nun kann man Operationen definieren, die auch selbstdefinierte Typen als Parameter verwenden. So könnte man eine Operation implementieren, die zwei Peano Zahlen addiert. Dies kann wie folgt umgesetzt werden.

```
add :: Peano -> Peano -> Peano
add O     p2 = p2
add (F p) p2 = F (add p p2)
```

`Maybe` ist von besonderem Interesse, da dieser Typ viel in dieser Arbeit verwendet wird. Dieser besteht aus zwei Konstruktoren `Just` und `Nothing`. `Just` kann einen Wert halten, während `Nothing` einer Konstanten entspricht, die keinen Wert hält. Das führt zu der einer Interpretation von `Maybe` als ein Puffer mit Kapazität 1. Dieser Puffer kann also maximal einen Wert enthalten. Wichtig ist dabei zu beachten, dass `Maybe` ein eigener Typ ist und nicht einfach an Stelle eines anderen verwendet werden kann. Das folgende Beispiel zeigt dies.

```
foo :: Maybe Int -> Int -> Int
foo Nothing  y = y
foo (Just x) y = x + y
```

Um die Zahl im `Just` Konstruktor mit der anderen Zahl zu verrechnen, so muss man diese erst durch Pattern Matching zugänglich machen. Man hätte nicht einfach einen Wert vom Typ `Maybe Int` mit einem Wert vom Typ `Int` addieren können. Das würde der Compiler beziehungsweise das Typsystem nicht zulassen. Dies ist ein wesentlicher Vorteil von `Maybe`, denn eine andere Interpretation ist, dass es für eine möglicherweise fehlschlagende Berechnung steht. Das folgende Beispiel verdeutlicht dies.

```
foo :: Int -> Int -> Maybe Int
foo a b = if b == 0
  then Nothing
  else Just (a / b)
```

In diesem Fall hat man eine Operation, die zwei Zahlen durcheinander teilen will. Selbstverständlich ist der Fall, dass die zweite Zahl 0 sein könnte, kritisch, da dies mathematisch nicht definiert ist. Damit dies aber nicht zu einem Laufzeitfehler führt, wird diese Berechnung im `Maybe` Kontext verschoben. Denn so kann man für diesen Fall mit `Nothing` als Ergebnis den Fehlschlag darstellen. So ist sichergestellt, dass eine Nulldivision nicht zur Laufzeit stattfinden kann. Damit wird eine Fehlerbehandlung erzwungen, denn das Ergebnis muss durch Pattern Matching erst auf die zwei möglichen Ausgaben überprüft werden, bevor man mit der Zahl weiter rechnen kann. Im folgenden Beispiel wird das noch erweitert.

```haskell
foo :: Maybe Int -> Maybe Int -> Maybe Int
foo ma mb = case ma of
  Nothing -> Nothing
  Just a -> case mb of
    Nothing -> Nothing
    Just b -> Just (a + b)
```

In diesem Beispiel nimmt die Operation zwei Berechnungen, die vielleicht fehlschlagen können. Nulldivision ist nur eine Möglichkeit dafür. Im Fall, dass eine der beiden Berechnungen fehlschlägt, kann die restliche Berechnung nicht mehr sinnvoll durchgeführt werden. Deshalb wird die Berechnung dadurch abgebrochen, in dem der Ausdruck zu `Nothing` ausgewertet wird. Schlussendlich sollen die Ergebnisse dieser Berechnungen schlicht addiert werden.

Allerdings wird das Programmieren dadurch auch erschwert. Schon für dieses kleine Beispiel waren mehrere Zeilen an Code notwendig um die Berechnung vollständig zu definieren. Komplexere Berechnungen würden noch mehr Zeilen benötigen. Um dies zu erleichtern gibt es in Haskell und Curry eine syntaktische Hilfe.

```haskell
foo :: Maybe Int -> Maybe Int -> Maybe Int
foo ma mb = do
  a <- ma
  b <- mb
  return (a + b)
```

Dies ist die sogenannte *Do-Notation*. Diese ist allgemein für monadische Berechnungen möglich, zu denen `Maybe` auch zählt. Diese wird hier anhand von `Maybe` erläutert. Man kann diese Notation als *syntaktischen Zucker* bezeichnen, denn sie ist nur eine vereinfachte Schreibweise für eine etwas aufwendigere Schreibweise. Diese würde für dieses Beispiel so aussehen.

```haskell
foo :: Maybe Int -> Maybe Int -> Maybe Int
foo ma mb =
  ma >>= \a ->
    mb >>= \b ->
      return (a + b)
```

Jede Zuweisung einer Variablen, wie man eine Zeile wie `a <- ma` interpretieren könnte, wird zum Aufruf eines speziellen Operators `>>=` mit dem Ausdruck und einer lokalen Funktion, die den Rest der Berechnung enthält. Dies führt zu eine sequentiellen Abarbeitung der Berechnungen. Diese folgt genau der Reihenfolge, wie sie in der *Do-Notation* geschrieben wurde. Der Operator `>>=` für den Typ `Maybe` kann wie folgt implementiert werden.

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
ma >>= f = case ma of
  Nothing -> Nothing
  Just a -> f a
```

Durch Pattern Matching wird die Berechnung darauf geprüft, ob sie fehlgeschlagen ist. Wenn dies der Fall ist, so wird die restliche Berechnung mit der Rückgabe von Nothing abgebrochen. Im anderen Fall wird die übergebene Operation auf das Ergebnis angewendet, die für die restliche Berechnung steht.

Nun ist eine solche Berechnung nicht nur für `Maybe` nützlich, sondern für eine ganze Klasse an Typen. Diese bezeichnet man als Monaden und die Berechnungen kann man entsprechend als monadische Berechnungen bezeichnen. Monadische Berechnungen haben besondere Eigenschaften an sich, die über denen normaler Berechnungen hinaus gehen. So hat `Maybe` eine Fehlerbehaftung, die aus der Laufzeit in die Kompilierzeit verschoben wird. Der Begriff "Klasse" ist hierbei passend, denn sowohl in Haskell als auch in Curry wird dies durch eine Typklasse definiert. Für Monaden könnte es so aussehen.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Mit Typklassen kann man eine Menge von Operationen definieren, die für mehr als einen Typ verwendet werden kann. Wird diese Klasse für Monaden für mehrere Typen implementiert, so kann man die selben Operationen benutzen für unterschiedliche Typen verwenden. Wenn man hier im Beispiel das `m` durch `Maybe` ersetzt, so bekommt man exakt die gezeigte Definition. Nur das `return` ist noch ungeklärt. Dies ist eine Operation, die einen einfachen Wert in den Kontext der Monade zieht, so dass dieser wie eine monadische Berechnung behandelt werden kann. Dies ist für `Maybe` einfach der Konstruktor `Just`. Denn Konstruktoren sind auch Operationen und dieser hat exakt den korrekten Typ und auch die Semantik passt. Der übergebene Wert wird in den `Maybe` Kontext gezogen.

Es gibt noch einen weiteren Typ, der auch monadische Berechnungen repräsentiert, welcher in dieser Arbeit viel verwendet wird. Auch dieser ist von Haskell bekannt, dennoch wird hier einmal genauer auf diesen eingegangen. Es handelt sich hierbei um die `IO` Monade, welche Seiteneffekt-behaftete Berechnungen repräsentiert.

Als reine Sprachen schränken Curry und Haskell Seiteneffekte stark ein. Dies ist ein großer Vorteil, denn so kann ein Großteil des Programms vor Seiteneffekten geschützt werden. Allerdings sind Ein- und Ausgabe von Daten auch Seiteneffekte. Daher ist es nötig Seiteneffekte in irgendeiner Form zu erlauben. Dafür folgt ein simples *Hello World*-Beispiel.

```
foo :: IO ()
foo = do
  putStrLn "Hello World!"
```

Wie der Typ zeigt, gibt die Operation kein Ergebnis zurück. Dies wird in Curry und Haskell durch den *Unit* Typ dargestellt, welcher nur einen konstanten Wert hat. Dennoch, wenn man diese Operation ausführen würde, so würde die Nachricht auf die Standardausgabe ausgegeben. Dies ist ein Seiteneffekt,

denn die Standardausgabe ist nicht Teil der Eingabe. Zumindest ist sie es nicht explizit. Für eine Benutzereingabe gibt es auch ein Beispiel.

```
foo :: IO ()
foo = do
  s <- getLine
  putStrLn s
```

Alles, was diese Operation macht, ist einen String von der Standardeingabe zu lesen und auf die Standardausgabe zu schreiben. Auch dies ist ein Seiteneffekt, denn die Standardeingabe ist auch nicht als Argument übergeben worden. Der `String`, welcher ausgegeben wird, kann auch bei jedem Aufruf anders sein, obwohl das Programm selbst keine Änderungen bei den Argumenten macht.

Um dies trotzdem in die reine Programmierung einzugliedern wurde die `IO` Monade entwickelt. Man kann diese so interpretieren, dass zusätzlich zu den Argumenten, die das Programm explizit übergibt, auch ein spezielles, verstecktes Argument implizit mit übergeben wird. Dieses implizite Argument repräsentiert die Außenwelt wie das System, auf welchem das Programm läuft, aber auch das Netzwerk, mit welchem das System verbunden ist. Darüber hinaus wird diese *Außenwelt* auch als implizite Ausgabe zum expliziten Ergebnis hinzugefügt. Eine konzeptuelle Implementierung von `IO` könnte wie folgt aussehen.

```
data IO a = IO (World -> (a, World))

instance Monad IO where
  return x = IO (\w -> (x, w))

  (IO mf) >>= f = IO (\w -> let (x, w') = mf w
                                IO f2 = f x
                            in f2 w')
```

2.2 JSON

Ein häufig benutztes Datenformat in vielen Programmiersprachen ist JSON[1]. Es steht für *JavaScript Object Notation*, stammt also aus der Welt von JavaScript. Es ist ein extrem nützliches Datenformat, da es nicht nur einfach für Menschen zu lesen ist, sondern es auch für Programme einfach ist solche Daten einzulesen.

Es ist eine Teilsprache von JavaScript und besteht im Grunde nur aus einem Datenausdruck, wie er auch in JavaScript verwendet werden könnte. Um möglichst viele Daten darstellen zu können, enthält JSON wesentliche Datentypen, mit denen man den Ausdruck schreiben kann. Es gibt die folgende Liste an Typen.

- *number*, sowohl Ganzzahlen als auch Gleitkommazahlen
- *boolean*, `true` und `false`

[1] https://www.json.org/json-en.html, letzter Zugriff: 18.11.2024

- *string*, Text
- *array*, Listen von Werten
- *object*, Objekt mit benannten Feldern
- *null*, wie ein Nullpointer

number, *boolean* und *string* sind simple Datentypen. Bei *number* ist zu beachten, dass nicht zwischen Ganzzahlen und Gleitkommazahlen unterschieden wird. Im folgenden Beispiel werden drei Variablen definiert, die alle vom Typ *number* sind. Die erste ist eine Ganzzahl, die zweite eine Gleitkommazahl und die dritte wieder eine Gleitkommazahl, die mittels der anderen beiden berechnet wird.

```
let a = 42
let b = 3.14
let c = b - a // -38.86
```

In Programmiersprachen mit statischen Typsysteme können Listen normalerweise nicht verschiedene Typen enthalten. Eine Liste kann in solchen Sprachen also nur Werte eines Typs enthalten. JavaScript und somit JSON sind aber dynamisch, daher erlauben beide es beliebige Typen in einem Array zu benutzen. Das nachfolgende Beispiel zeigt legitime Arrays.

```
let a = [1, 2, 3.14]
let b = [true, 42, null]
let c = [2, ["hello", "world", 333], false]
```

Wie zu sehen ist, kann ein Array selbst auch einen Array enthalten. Es gibt keinerlei Einschränkung auf den Inhalt eines Arrays. *objects* sind eine weitere Möglichkeit eine Menge von Daten miteinander zu verbinden. Doch während man bei Arrays über einen Index auf die Daten zugreifen kann, kann man bei einem *object* dies durch Feldnamen erreichen. Ein Beispiel hierfür folgt.

```
let o = {
  "a": 42,
  "b": [true, false]
  "c": {
    "foo": 42,
    "bar": true
  }
}
```

Die Reihenfolge der Felder ist nicht wichtig, da man mit den Feldnamen man jedes Feld eindeutig identifizieren kann. Feldnamen sind nichts weiteres als Strings und die Feldwerte sind normale JavaScript beziehungsweise JSON Werte. Dabei ist, wie bereits mit Arrays, es auch möglich andere *objects* als

Werte zu verwenden. Arrays können natürlich auch *objects* enthalten und *objects* können wiederum auch Arrays enthalten.

Mit diesen Typen ist alles vorhanden, was zum Speichern von Daten gebraucht wird. Egal welche Programmiersprache man verwendet, es gibt immer eine Möglichkeit Werte in beide Richtungen umzuwandeln. Es können Eigenschaften verloren gehen wie zum Beispiel die Homogenität von Listen in einer statischen Sprache, doch für den einfachen Umtausch von Daten genügt es.

Dennoch können Probleme in JSON auftreten. Dadurch, dass Ausdrücke viel Freiheit in ihren Formen haben, können dieser der Erwartung beim Programmieren widersprechen. Vielleicht fehlt ein bestimmtes Feld, welches man erwartet hat, oder vielleicht ist der Wert in solch einem Feld nicht vom erwarteten Typ. Möglicherweise erwartete man, dass alle Werte in einem Array den gleichen Typ haben oder dass dieser Arrays eine maximale Länge hat.

Falls man auf solche Eigenschaften angewiesen ist, so kann JSON zunächst nicht weiterhelfen. Man muss beim Programmieren selbst Lösungen dafür erstellen oder externe Lösungen suchen. Eine solche externe Lösung ist JSON Schema[2]. Dies ist ein Beschreibungsformat für JSON Dateien, das die angesprochenen Eigenschaften und mehr überprüfen und verifizieren kann. Ein Beispiel für eine simple JSON Schema Datei folgt hier.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/user.schema.json",
    "title": "User",
    "description": "A user of the web service",
    "type": "object"
}
```

Wie zu sehen ist, ist JSON Schema selbst eine JSON Datei. Man benutzt also JSON selbst um JSON zu erweitern. Das Feld `"$schema"` gibt den verwendeten Standard an und das Feld `"$id"` gibt den Namen der JSON Schema Datei an. `"title"` ist der Name vom Objekt, den man hier definiert, und `"description"` ist eine Beschreibung, worum es sich bei dem Objekt handelt. Zuletzt `"type"` gibt den Typ des Objekts an. In diesem Fall gibt man also vor, dass es explizit ein *object* sein muss und JSON Dateien könnten dann mit entsprechenden Anwendungen verifiziert werden. Es existieren noch weitere Felder, die man im Schema angeben kann.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/user.schema.json",
    "title": "User",
    "description": "A user of the web service",
    "type": "object"
    "properties": {
```

[2]https://json-schema.org/, letzter Zugriff: 18.11.2024

```
        "userID": {
            "description": "The ID of the user.",
            "type": "integer"
        },
        "userName": {
            "description": "The name of the user.",
            "type": "string"
        }
    }
}
```

Für *objects* kann man mit dem Feld `"properties"` die Felder vorgeben. In diesem Fall kann das *object* also zwei Felder haben, `"userID"` und `"userName"`. Die Form dieser Felder kann auch durch ein Schema vorgegeben werden. So ist hier für beide eine Beschreibung und ein Typ angegeben. Besonders `"userID"` ist hierbei hervorzuheben, denn bei diesem wird ein Typ vorgegeben, welcher so nicht in JSON existiert.

JSON Schema erweitert die Menge er möglichen Typen von JSON und erlaubt so in diesem Beispiel zwischen Ganzzahlen und Gleitkommazahlen zu unterschieden. In diesem Fall muss das Feld also eine Ganzzahl enthalten. Es gibt für verschiedene Typen weitere Eigenschaften, die man vorgeben kann. So kann man für Zahlen angeben einen Minimalwert oder Maximalwert vorgeben. Bei Arrays können sogar die Typen der Elemente vorgegeben werden wie im folgenden Beispiel.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/user.schema.json",
    "title": "User",
    "description": "A user of the web service",
    "type": "object",
    "properties": {
        "userID": {
            "description": "The ID of the user.",
            "type": "integer"
        },
        "userName": {
            "description": "The name of the user.",
            "type": "string"
        },
        "friends": {
            "description": "The list of friends of the user.",
            "type": "array",
            "items": {
                "type": "integer"
            }
```

```
        }
    }
}
```

Zuletzt ist noch wichtig, dass Felder in einem *object* auch als notwendig deklariert werden können. In den bisherigen Beispielern würden auch JSON Dateien beim Verifizieren akzeptiert werden, in welchen manche Felder fehlen. Im folgenden werden die Felder als notwendig deklariert.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/user.schema.json",
    "title": "User",
    "description": "A user of the web service",
    "type": "object"
    "properties": {
        "userID": {
            "description": "The ID of the user.",
            "type": "integer"
        },
        "userName": {
            "description": "The name of the user.",
            "type": "string"
        },
        "friends": {
            "description": "The list of friends of the user.",
            "type": "array",
            "items": {
                "type": "integer"
            }
        }
    },
    "required": ["userID", "userName"]
}
```

Im Feld `"required"` gibt man einen Array von Strings an, welche den Feldnamen entsprechen, die man als notwendig deklarieren will. Nun kann eine JSON Datei nur akzeptiert werden, wenn sie auch wirklich diese Felder hat.

Hat man so eine JSON Schema Datei für jede Kategorie von JSON Datei geschrieben, so kann man für diverse Programmiersprachen entwickelte Anwendungen benutzen, die mit diesen Schemen JSON Dateien verifizieren können. So kann zur Laufzeit versichert werden, dass alle gewünschten Eigenschaften erfüllt sind und zusätzliche Überprüfungen nicht nötig sind.

Zum Zeitpunkt dieser Arbeit fehlt eine solche Anwendung für Curry. JSON Schema hat aber auch einen Vorteil über das automatische Verifizieren hinaus. Denn man kann diese Dateien als Spezifikationen für die JSON Dateien verwenden.

Für Curry existiert ebenfalls ein Paket für das Arbeiten mit JSON. Dieses Paket hat den Namen *json*[3]. Zum Einlesen von JSON Ausdrücken existiert die Operation `parseJSON`, die als Ergebnis einen Wert vom Typ `JValue` zurückgibt. Dieser ist wie folgt definiert.

```
data JValue = JTrue
            | JFalse
            | JNull
            | JString String
            | JNumber Float
            | JArray [JValue]
            | JObject [(String, JValue)]
```

Mit diesem Typ können alle JSON Ausdrücke dargestellt werden und man kann sie innerhalb eines Curry Programms verarbeiten. Es gibt außerdem das Modul `Pretty` um aus einem JSON Ausdruck einen `String` zu erzeugen.

Die in dieser Arbeit entwickelte Anwendung verwendet JSON zum Speichern von Informationen. Doch werden diese nicht nur gespeichert, sondern sie werden auch gelesen. Um dies möglichst einfach durchführen zu können gibt es ein weiteres Modul im Paket *json*, welches das Lesen und Schreiben von Werten über den Typ [JValue] hinaus ermöglicht.

Durch die Typklasse `ConvertJSON` ist es möglich, Curry Terme zu JSON Ausdrücken umzuwandeln und umgekehrt aus JSON Ausdrücken Curry Terme zu erhalten. Die Typklasse ist wie folgt definiert.

```
class ConvertJSON a where
  toJSON :: a -> JValue
  fromJSON :: JValue -> Maybe a

  toJSONList :: [a] -> JValue
  fromJSONList :: JValue -> Maybe [a]
```

Beim Umwandeln von `JValue` zu einem anderen Typ kann es leicht zu unpassenden Eingaben kommen, weshalb das Ergebnis im `Maybe` Kontext ist. Zum Beispiel kann man aus einem JSON String nicht so einfach eine Liste in Curry generieren.

2.3 CPM

Moderne Programmiersprachen benutzen spezielle Programme um das Teilen von Programmbibliotheken zu erleichtern. Diese haben verschiedene Namen, im Falle von Curry gibt es CPM (Curry Package Manager)[7]. Curry Code wird hierbei in Pakete unterteilt, die mit Versionen bestückt werden und so erlauben Abhängigkeiten anzugeben.

Ein Paket ist eine Menge von Curry Modulen, die mit einer JSON Datei mit Verwaltungsinformationen erweitert ist. Diese Datei hat immer den Namen

[3] https://cpm.curry-lang.org/pkgs/json-3.0.0.html, letzter Zugriff: 02.12.2024

package.json. Die Module sind ganz normale Curry Module, sie haben keinerlei Zusatz. Für den Aufbau von *package.json* kann man sich die Datei für das Paket *json* als Beispiel anschauen.

```
{
    "name": "json",
    "version": "3.0.0",
    "author": "Jonas Oberschweiber <jonas@oberschweiber.com>",
    "maintainer": "Michael Hanus <mh@informatik.uni-kiel.de>",
    "synopsis": "A JSON library for Curry",
    "category": [ "Data", "Web" ],
    "license": "BSD-3-Clause",
    "licenseFile": "LICENSE",
    "dependencies": {
        "base"     : ">= 3.0.0, < 4.0.0",
        "det-parse": ">= 3.0.0, < 4.0.0",
        "wl-pprint": ">= 3.0.0, < 4.0.0"
    },
    "exportedModules": ["JSON.Data", "JSON.Parser", "JSON.Pretty"],
    "testsuite": [
        { "src-dir": "src",
          "modules": ["JSON.Parser"]
        },
        { "src-dir": "examples",
          "modules": [ "SimpleTest" ]
        },
        { "src-dir": "test",
          "modules": [ "TestConvert" ]
        }
    ],
    "source": {
        "git": "https://github.com/curry-packages/json.git",
        "tag": "$version"
    }
}
```

Da es in erster Linie Verwaltungsinformationen enthält, sind manche der Felder selbsterklärend. Dazu gehören zum Beispiel die Felder `"name"`, `"author"` und `"maintainer"`. Stattdessen werden die Felder erklärt, die für diese Arbeit wichtig sind.

Abhängigkeiten trägt man im Feld `"dependencies"` ein. Statt einen Array mit den entsprechenden Abhängigkeiten anzulegen, wird jede Abhängigkeit zu einem eigenen Feld in einem *object*. Als Wert trägt jedes Feld einen String, welcher logische Bedingungen an das genannte Paket stellt. Bedingungen können auch komplexer sein, doch meist sind sie so simpel wie im Beispiel. Das Komma kann man als logisches *Und* verstehen. Im Beispiel müssen die Versionen also mindestens `"3.0.0"` und kleiner als `"4.0.0"` sein.

Versionen folgen hier dem *Semantic Versioning Standard*[4]. Diese teilt eine Version in drei Nummern auf. Die erste Nummer ist die *Major Version* und gibt an, dass Änderungen an der API vorgenommen wurden. Dies kann bedeuten, dass Operationen umbenannt wurden, sich Argumente von Operationen geändert haben oder auch dass ganze Operationen oder Module entfernt wurden. Die zweite Nummer ist die *Minor Version* und gibt an, dass die vorgenommenen Änderungen die API unverändert lassen. Darunter würde zum Beispiel *Refactoring* fallen, die am Interface nichts ändert, sonder nur die Implementierungsdetails verändert. Die Verwendung des Pakets bleibt unverändert. Die letzte Nummer ist die *Patch Version* und gibt an, dass nur kleine Änderungen vorgenommen wurden. Meist handelt es sich hierbei um Fehlerausbesserungen.

Ein weiteres wichtiges Feld in der JSON Datei ist `"exportedModules"`. Hier gibt man an, welche Module von Außen sichtbar sein sollen. Das Paket kann also mehr Module enthalten, die aber nur vom Paket selbst verwendet werden sollen. Besonders gilt es hier zu beachten, dass dieses Feld auch ausgelassen werden kann. In diesem Fall gelten alle Module im Paket als sichtbar.

Eine wichtige Funktion von CPM ist noch hervorzuheben. Die `checkout` Funktion erlaubt es einem, den Inhalt eines Pakets an eine andere Stelle zu kopieren. So kann man dieses zum Beispiel bearbeiten. In dieser Arbeit wird es benutzt um Kopien der Pakete an eine zentrale Stelle des Systems anzulegen.

2.4 CASS

CASS[8] ist eine Anwendung, welche für Curry entwickelt wurde um Analysen für Curry Programme durchzuführen. Es gibt zahlreiche Analysen und die Anwendung wurde so konzipiert, dass sich leicht weitere Analysen in der Zukunft einfügen lassen. Manche der Analysen sind nur Approximationen, da diese im Allgemeinen nicht exakt berechenbar sind.

Zur Veranschaulichung wird ein einfacher Curry Code hier angelegt und einige der Analysen werden auf diesen ausgeführt.

```
module Example where

foo :: Int -> Int -> Int
foo x y = x

bar :: Bool
bar = False ? True
```

Die Analysen, welche als Beispiel ausgewählt wurden, sind *Demand*, *Deterministic* und *Terminating*. *Demand* bestimmt, welche Argumente einer Operation nötig für die Berechnung sind. *Deterministic* bestimmt, ob die Operation deterministisch oder nicht-deterministisch ist. Zuletzt *Terminating* bestimmt, ob die Operation garantiert immer ein Ergebnis berechnet.

[4] https://semver.org/, letzter Zugriff: 30.11.2024

```
$ cass Demand Example
bar : no demanded arguments
foo : demanded arguments: 1

$ cass Deterministic Example
bar : non-deterministic
foo : deterministic

$ cass Terminating Example
bar : terminating
foo : terminating
```

Standardmäßig ist die Ausgabe für Menschen lesbar, doch es gibt weitere Möglichkeiten für die Ausgabe. Für die automatische Verarbeitung der Ausgabe sind vor allem *CurryTerm* und *JSONTerm* nützlich. *CurryTerm* gibt die Ergebnisse in Form der eigentlichen Curry Terme aus. *JSONTerm* gibt ebenfalls die Curry Terme aus, diese werden aber im JSON Format ausgegeben. Es gibt auch die Option einfach *JSON* zu benutzen, dies entspricht aber etwas der normalen Ausgabe im JSON Format. Für die Analyse *Demand* sehen die Ausgabe wie folgt aus.

```
$ cass -f CurryTerm Demand Example
[(("Example","bar"),"[]"),(("Example","foo"),"[1]")]

$ cass -f JSONTerm Demand Example
[ {
  "module": "Example",
  "name": "bar",
  "result": "[]"
}
, {
  "module": "Example",
  "name": "foo",
  "result": "[1]"
} ]

$ cass -f JSON Demand Example
[ {
  "module": "Example",
  "name": "bar",
  "result": "no demanded arguments"
}
, {
  "module": "Example",
  "name": "foo",
  "result": "demanded arguments: 1"
} ]
```

Als Curry Term ist das Ergebnis der *Demand* Analyse nur eine Liste. Die Ergebnistypen können sich aber zwischen Analysen unterscheiden. CASS wurde so konzipiert, dass Analysen mit verschiedenen Typen als Ausgabe implementiert werden können.

Die Analysen laufen inkrementell. Inkrementell bedeutet, dass nur die importierten Module analysiert werden, die wirklich benötigt werden. Wenn also das zu analysierende Modul nur ein Modul aus einem Paket importiert, wird auch nur dieses aus dem Paket analysiert. Dies zeigt das folgende Beispiel.

```
module Foo where

import JSON.Data
import JSON.Parser

foo :: String -> Maybe JValue
foo = parseJSON
```

Das Paket *json* hat mehrere Module, aber in diesem Beispiel werden nur die Module `JSON.Data` und `JSON.Parser` importiert. Wenn nun das Module `Foo` analysiert wird, werden auch diese beiden Module analysiert. Da aber nur diese zwei importiert werden, werden auch nur diese mit analysiert.

2.5 curry-calltypes

Allgemein beim Programmieren existieren zwei grundlegende Arten von Laufzeitfehlern. Eine kann als extern bezeichnet werden. Zu solchen Fehlern zählt das Fehlen von Dateien, die man lesen will, oder das Fehlen von Rechten um zum Beispiel eine Datei anzulegen. Möglicherweise laufen auch andere Funktionalitäten im Hintergrund, die aus irgendeinem Grund fehlschlagen und abstürzen. Diese Fehler können aber nicht statisch überprüft werden, da sie erst zur Laufzeit auftreten können und vom System abhängen.

Die andere Art von Laufzeitfehlern können als intern bezeichnet werden. Im Falle von Haskell und Curry würden partiell definierte Funktionen, die also nicht für alle möglichen Eingaben eine Definition haben. Die wohl bekanntesten Beispiele hierfür sind die Funktionen *head* und *tail*.

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

Dies sind Möglichkeiten die beiden Funktionen zu definieren. Für beide fehlt der Fall der leeren Liste. Der Grund dafür ist, dass logisch kein Ergebnis für eine leere Liste existiert. Eine leere Liste hat weder ein Kopfelement noch eine Restliste. Falls diese Funktionen auf leere Listen angewendet werden, dann wird in Haskell ein Fehler geworfen und das Programm stürzt möglicherweise ab.

In Curry würde kein Ergebnis ausgegeben werden, womit Curry aber arbeiten kann.ein Fehler geworfen, der unbehandelt das Programm zum Absturz führt.

In solche einer Situation kann man versuchen die Funktion statisch zu analysieren und so zu bestimmen, ob es an irgendeiner Stelle mit Argumenten aufgerufen wird, für welche die Funktion nicht definiert ist. Im Allgemeinen ist dies nicht möglich, aber eine Approximation kann berechnet werden.

curry-calltypes[3] ist eine Anwendung, welche für Curry entwickelt wurde und solche Approximationen berechnet. Um dies zu ermöglichen bestimmt es für jede Operation einen abstrakten *call type*. Dieser repräsentiert im wesentlichen die Menge von Eingabewerten, für welche die Operation definiert ist. Da die Menge auch unendlich sein könnte, wird diese auf die Konstruktoren beschränkt. Für arithmetische Ausdrücke verwendet die Anwendung Z3 um diese zu überprüfen.

```
module Example1 where

foo :: Bool -> Bool
foo True = True

bar :: Maybe () -> ()
bar (Just x) = x
```

In diesem Beispiel sind beide Operationen nur partiell definiert. Ruft man `foo` mit `False` auf, so wird kein Ergebnis berechnet. Das Gleiche passiert, wenn man `bar` mit `Nothing` aufruft. Führt man die Analyse von *curry-calltypes* aus, so erhält man das folgende Ergebnis.

```
$ curry-calltypes Example1
MODULE 'Example1' VERIFIED W.R.T. NON-TRIVIAL ABSTRACT CALL TYPES:
bar: {Just}
foo: {True}
```

Man kann sehen, dass für `foo True` und für `bar Just` als abstrakte Typen herauskommen. Dies bedeutet, dass sie für solche Werte definiert sind und kein interner Laufzeitfehler mit diesen auftreten kann. Ändert man den Code, dass die Operationen nicht mehr partiell-definiert sind, dann ändert sich auch das Ergebnis von *curry-calltypes*.

```
module Example2 where

foo :: Bool -> Bool
foo True  = True
foo False = False

bar :: Maybe () -> ()
bar (Just x) = x
bar Nothing  = ()
```

```
$ curry-calltypes Example2
MODULE 'Example2' VERIFIED
```

2.6 curry-interface

Wenn ein Curry Programm kompiliert wird, werden neben dem lauffähigen Programm diverse weitere Dateien erzeugt. Darunter sind auch *icurry* Dateien, die das Interface des Programms repräsentieren. Zum Interface gehören Typdefinitionen, Definitionen von Operationen und Definitionen von Typklassen.

Um diese *icurry* Dateien zu verarbeiten wurde *curry-interface*[5] entwickelt, welches fähig ist diese Dateien zu parsen und in einen Curry Typ umzuwandeln. Mithilfe dieses Typs kann man dann Informationen aus dem Interface suchen.

Das ist hilfreich, da der Compiler von Curry in Haskell geschrieben ist. Daher kann man diesen nicht direkt in einem Curry Programm verwenden. Dadurch wird das Verarbeiten von Informationen im Curry Programm schwierig. Mit *curry-interface* ist es aber möglich diverse Informationen zu erlangen.

Zu diesen Informationen zählen zum Beispiel die Signaturen. Das sind die Typen von Operationen und auf diese einen leichten Zugriff zu haben ist sehr nützlich. Man kann auch auf die Typklassen einschließlich ihrer Methoden zugreifen.

[5] https://cpm.curry-lang.org/pkgs/curry-interface.html, letzter Zugriff: 02.12.2024

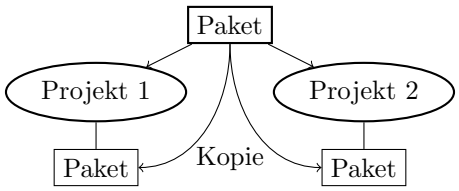
3 CurryInfo - Aufgabe

3.1 Motivation

CASS ist als Analyseanwendung sehr nützlich, doch hat es trotzdem Probleme. Hierzu muss man sich anschauen, wie CASS intern Module bei einer Analyse verarbeitet und was beim Einbinden von Paketen in einem Projekt passiert.

Wenn man ein Paket im Projekt als Abhängigkeit angegeben hat, dann kann man dieses mit `cypm install` im Projektordner für das Projekt installieren. Falls das Paket noch nicht auf dem System vorliegt, wird es automatisch heruntergeladen. In jedem Fall wird eine Kopie des Codes, der im Paket zu finden ist, in einem lokalen versteckten Ordner im Projekt abgespeichert. Dies ist der Code, der beim Kompilieren eingebunden wird.

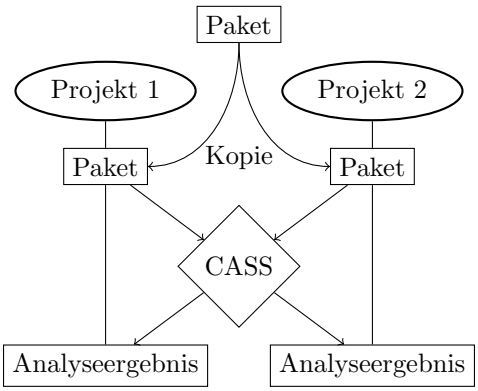
Hat man zwei verschiedene Projekte, die vom selben Paket abhängen, so haben beide Projekte nur eine Kopie des Pakets bei sich liegen. Die folgende Grafik verdeutlicht diesen Fall.



Für eine Analyse kopiert CASS die Verzeichnisstruktur, in welchem das zu analysierende Modul liegt. So können die Ergebnisse, die zentral abgespeichert werden, eindeutig den richtigen Modulen zugeordnet werden. Dies ist zum Beispiel wichtig, wenn unterschiedliche Projekte den gleichen Namen für Module verwenden.

Sollte bei einer Analyse das Ergebnis für ein anderes Modul benötigt werden, so schaut CASS zu erst nach, ob ein Ergebnis bereits vorliegt. Hierzu schaut es in die Kopie des versteckten Ordners nach, welcher für die Analyse angelegt wurde. Wenn das Ergebnis bereits vorliegt, so wird dieses einfach übernommen. Ansonsten muss CASS das Module ebenfalls analysieren.

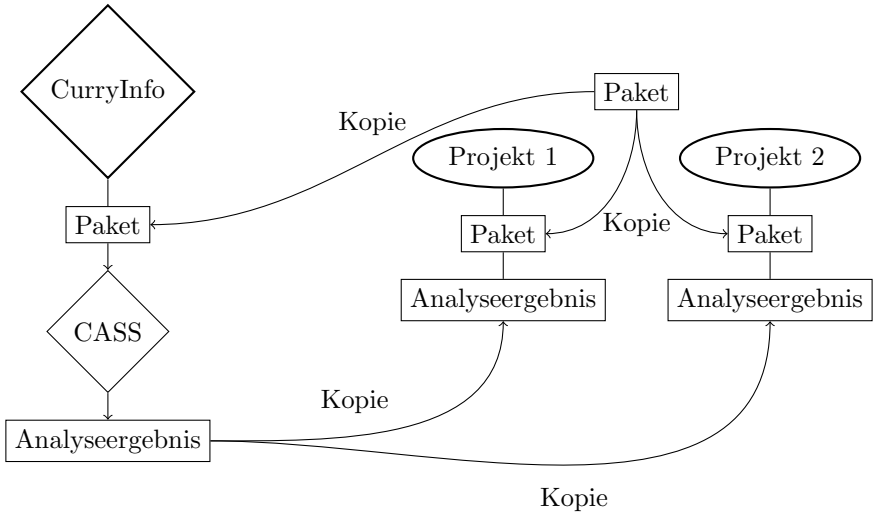
Dies funktioniert aber nur innerhalb eines Projekts. Schließlich sind die Module eines anderen Projekts in einem anderen versteckten Ordner. Im Beispiel mit zwei Projekten heißt das, dass CASS während der Analyse vom ersten Projekt nicht auf Ergebnisse vom zweiten Projekt zugreifen kann. Der Code vom Paket, der in beiden Projekten vorliegt, muss also für beide Projekte jeweils analysiert werden. Die folgende Grafik zeigt dies.



Eigentlich ist es überflüssig den selben Code mehrfach zu analysieren. Allerdings hat CASS keine Möglichkeit zu bestimmen, ob der Code eines Pakets bereits in einem anderen Projekt analysiert wurde.

An dieser Stelle kann CurryInfo helfen. Als Verwaltungssystem für Informationen von Curry Programmen kann es Informationen wie ein Cache an einem zentralen Ort auf dem System speichern. Mit einem passenden Interface können andere Anwendungen wie CASS Informationen anfragen und verarbeiten.

Statt lokale Kopien von Paketen zu analysieren, könnten die Analyseergebnisse für das Paket selbst erstellt werden. Wenn während einer Analyse CASS diese Ergebnisse benötigt, dann kann es CurryInfo anfragen und erhält diese. Die überflüssige Analysen sind so vermieden. Die folgende Grafik zeigt den groben Ablauf.



3.2 Ziel

Pakete sind die grundlegende Einteilung für Curry Programme. Wenn Informationen zu einem Paket vorliegen, so müssen diese nicht neu generiert werden. Pakete enthalten alles, was man über ein Curry Programm wissen muss.

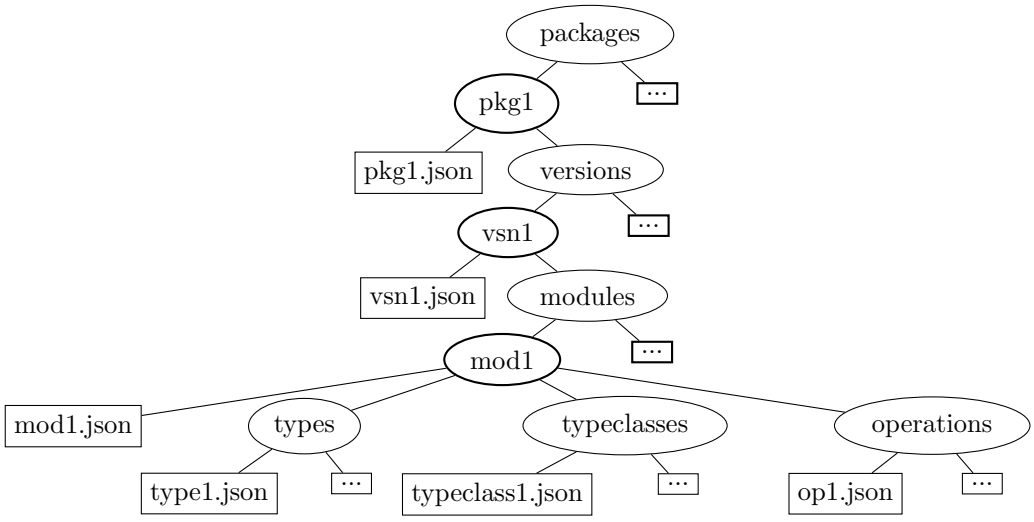
Pakete werden noch weiter unterteilt. Dies wird durch eine hierarchische Struktur umgesetzt. Jedes Paket hat eine Liste von Versionen, die Unterschiede untereinander haben. Daher ist nicht garantiert, dass generierte Informationen für unterschiedliche Versionen gleich bleiben.

Module sind die nächste Aufteilung. Pakete können aus mehreren Modulen bestehen und CASS hat die Analyse *UnsafeModule*, welche statt Operationen ein Modul analysiert. Diese Information kann nützlich sein, daher sollte sie zuweisbar sein.

Jedes Modul wiederum besteht aus diversen Bestandteilen. Zunächst die bereits genannten Operationen, für welche Analysen mit CASS bereits vorgestellt wurden. Aber auch Typen und Typklassen werden hier aufgeführt, da auch für diese Informationen vorhanden sein könnten.

Die Informationen werden in JSON Dateien gespeichert. Für jede Entität existiert eine eigene JSON Datei und die Informationen sind Felder. Die Struktur der JSON Dateien wird durch JSON Schema definiert. Diese sind in Appendix C zu finden.

Die Hierarchie wird durch eine Verzeichnisstruktur umgesetzt. Als Wurzel gibt es ein Verzeichnis für die Pakete. Jedes Paket hat ein Verzeichnis für Versionen, die wiederum jeweils ein Verzeichnis für Module haben. Module haben ebenfalls Verzeichnisse für Typen, Typklassen und Operationen. Diese Verzeichnisstruktur sieht also wie folgt aus.



Pakete sind die grundlegende Einteilung von Curry Programmen. Denn in der Regel wird man Informationen zu einem bestimmten Paket anfragen statt zu

einer Kombination aus Paket und Version. Zum Zeitpunkt dieser Arbeit haben Pakete als Informationen nur ihre eigenen Namen und Listen ihrer Versionen.

Versionen haben insbesondere Verwaltungsinformationen. Man könnte diese auch einem Paket direkt zuordnen wie Kategorien. Doch ist es nicht auszuschließen, dass sich diese über die Zeit in neueren Versionen verändern könnten. Die Dokumentation kann sich definitiv über die Zeit verändern, zum Beispiel wenn neue Module hinzugefügt, die in der Dokumentation beschrieben werden. Ansonsten sind auch eine Liste der Module und der Abhängigkeiten da.

Auch Module können eine Dokumentation in Form eines Dokumentationskommentar enthalten. Auf dieses einfachen Zugriff zu haben ist von Vorteil. Wie bisher sind auch hier Listen der nächsten Entitäten, die dem Modul zugeordnet sind. Dies sind Typen, Typklassen und Operationen. Außerdem kann ein Modul auf Sicherheit von CASS analysiert werden.

Zu Typen sind bisher keine Analysen implementiert. Dennoch gibt es Informationen, die nützlich sein können. So kann zum Beispiel die Liste der Konstruktoren von Nutzen sein. Ansonsten kann man auch einen Codeausschnitt abspeichern, welcher die Definition des Typs zeigt. Sollte es nötig sein einen derart detaillierten Einblick zu brauchen, kann man so einfach auf diesen zugreifen.

Typklassen gibt es auch keine Analysen. Auch hier kann man die Definition abspeichern für das Einsehen der Implementierungsdetails. Statt einer Liste von Konstruktoren gibt es hier eine Liste von Methoden, die für die Typklasse vorhanden sind.

Operationen sind die Entitäten mit den meisten Informationen. Ein Großteil der Informationen sind Analysergebnisse. Es wurde eine Auswahl an Analysen für diese Arbeit implementiert, die in Zukunft noch erweitert werden kann. Weiterhin werden auch Informationen über die Verwendung von Operationen gespeichert. Die Signatur ist nützlich um die Operation typkorrekt verwenden zu können. Die Präzedenz und die Assoziativität ist nützlich für den Fall, dass man die Operation infix verwenden kann.

Da bereits diverse Anwendungen die genannten Informationen generieren, muss diese Funktionalität nicht noch einmal umgesetzt werden. Stattdessen werden diese Anwendungen in dieser Arbeit in CurryInfo eingebunden.

CASS hat eigentlich ein Paket, welches den Code der Analysen enthält[6]. Da dies in Curry selbst geschrieben ist, könnte man es als Abhängigkeit von CurryInfo verwenden. Allerdings wäre CASS eine große Abhängigkeit. CASS ist auch nur nötig für die Generierung von Analyseergebnisse. Möglicherweise will man beim Verwenden von CurryInfo diese nicht anfragen. Dann wäre CASS eine überflüssige Abhängigkeit.

Deshalb wird CASS nicht als Abhängigkeit angegeben. Stattdessen wird es wie eine normale Anwendung auf der Kommandozeile von CurryInfo aufgerufen und die Ausgabe wird mit einem Parser verarbeitet.

Für *curry-calltypes* wird ebenfalls nicht als Abhängigkeit aufgeführt. Die Begründung ist ähnlich wie für CASS.

[6] https://cpm.curry-lang.org/pkgs/cass-analysis-4.0.0.html, letzter Zugriff: 03.12.2024

curry-interface ist aber als Abhängigkeit aufgeführt. Das Paket hat nicht nur die Anwendung, die ein Interface für Curry Code generieren kann. Das Paket enthält auch den Code zum Parsen des Interfaces und den Typ des Interfaces. Dieser wird benötigt um Informationen aus dem Interface zu extrahieren.

4 CurryInfo - Architektur und Aufbau

4.1 Idee

Die Hauptaufgabe von CurryInfo ist das Verwalten von Informationen von Entitäten. Dazu gehört das zentrale Abspeichern der Informationen und der einfache Zugriff auf diese für andere Anwendungen. Andere Anwendungen brauchen aber nicht alle Informationen, die zu einer Entität vorliegt. CASS zum Beispiel würde sich nur für Analyseergebnisse interessieren und weniger für Signaturen. Eine IDE braucht vermutlich die Kategorien einer Version nicht.

Anwendungen sollen also die Möglichkeit haben eine Menge von Informationen zu nennen, die sie haben wollen. Dies wird durch Anfragen umgesetzt. Eine Anwendung würde in diesem Fall die Entität nennen, für welche sie Informationen haben will, gefolgt von den benötigten Informationen.

Statt die Anfragen als eine Menge zu betrachten kann man sie auch als Liste ansehen. Damit ist eine sequentielle Abarbeitung dieser naheliegend. CurryInfo würde also die Anfragen in der Reihenfolge abarbeiten, in welcher es diese erhalten hat.

Da es erstrebenswert ist die Antwort der Anfragen möglichst schnell zu beantworten, werden unnötige Berechnungen vermieden. Dies bedeutet in diesem Fall, dass nur die angefragten Informationen bearbeitet werden sollen.

Möglicherweise liegen die angefragten Informationen noch nicht im Cache vor. Statt in diesem Fall die Anfrage fehlschlagen zu lassen, kann CurryInfo die Information automatisch generieren.

Zuletzt wird sichergestellt, dass keine überflüssigen Berechnungen ausgeführt werden. Wenn eine Information bereits im Cache vorliegt, so muss diese nicht nochmal generiert werden. Stattdessen sucht CurryInfo die angefragte Information erst im Cache und nur wenn dieser nicht vorhanden ist, dann wird sie generiert. Dies führt zum folgenden Ablauf.

- Angefragte Entität wird angegeben
- Anfragen werden angegeben
- Angefragte Informationen werden erst im Cache gesucht
- Wenn eine Information gefunden wurde, dann wird diese als Ergebnis verwendet
- Wenn eine Information nicht gefunden wurde, dann wird sie automatisch generiert
- Generierte Informationen werden im Cache gespeichert

In manchen Fällen ist dieser Ablauf nicht optimal. Wenn zum Beispiel CASS Operationen analysiert, dann werden immer alle Operationen eines Moduls analysiert. In normalen Ablauf würde aber nur das Ergebnis für eine Operation im Cache gespeichert werden, obwohl die Ergebnisse für andere auch vorliegen.

Deshalb existiert in den Fällen, wenn automatisch für mehr als eine Entität Informationen generiert werden, die Ergänzung, dass alle Informationen abgespeichert werden.

Außerdem kann es sein, dass man für mehrere Entitäten Informationen anfragen will. Zum Beispiel könnte eine Anwendung die Signaturen aller Operationen eines Moduls anfragen. Im normalen Ablauf wäre dies nicht möglich, da nur eine Entität zur Zeit bearbeitet wird. Deshalb wird es auch die Möglichkeit geben Anfragen für alle Entitäten eines Moduls zu stellen.

4.2 Architektur

CurryInfo besteht aus diversen Modulen mit ihren eigenen Aufgaben. Die Wichtigsten werden hier vorgestellt und genauer erklärt. Die folgende Liste zeigt die Module an, welche im Weiteren beschrieben werden.

- Commands: Operationen um externe Anwendungen aufzurufen
- Checkout: Kopieren von Paketen in einen lokalen Ordner für weitere Verarbeitungen
- Analysis: Analysen mit CASS und *curry-calltypes* ausführen
- Interface: Interface-Dateien erzeugen und Informationen extrahieren
- Generator: Informationen generieren
- Printer: Ausgabe von Informationen anfertigen
- Reader: Lesen vom Cache
- Writer: Schreiben vom Cache
- Configuration: Einstellung von Anfragen

4.3 Typen

Dem gesamten Programm sind einige Typen unterlegt, die durch das gesamte Programm hindurch verwendet werden. Daher wird auf diese näher eingegangen. Zunächst existieren eigene Typen für die Arten von Entitäten. Diese sind wie folgt definiert.

```
data CurryPackage
    = CurryPackage Package

data CurryVersion
    = CurryVersion Package Version

data CurryModule
    = CurryModule Package Version Module
```

```
data CurryType
    = CurryType Package Version Module Type

data CurryTypeclass
    = CurryTypeclass Package Version Module Typeclass

data CurryOperation
    = CurryOperation Package Version Module Operation
```

Für jede Entität existiert ein eigener Typ. Dadurch ist es möglich Operationen auf bestimmte Arten von Entitäten zu beschränken. So kann das Typsystem sicherstellen, dass alles korrekt ist. Die Typen der Felder sind nur Typaliase für `String`. Beispiele für diese Typen folgen hier.

```
CurryPackage "base"
CurryVersion "json" "3.0.0"
CurryModule "directory" "3.0.0" "System.Directory"
CurryType "json" "3.0.0" "JSON.Data" "JValue"
CurryTypeclass "base" "3.2.0" "Prelude" "Monad"
CurryOperation "directory" "3.0.0" "System.Directory"
    "doesFileExist"
```

Neben der Eingabe ist auch für die Ausgabe ein eigener Typ definiert. Hierbei sind es die verschiedenen Ausgabeformate, die jeweils einen Fall dieses Typ darstellen. Dieser ist wie folgt definiert.

```
data Output
    = OutputText String
    | OutputJSON JValue
    | OutputTerm [(String, String]
    | OutputError String
```

Falls etwas fehlschlägt, gibt es den speziellen Fall `OutputError`. Zum Einstellen des Ausgabeformats gibt es noch einen weiteren Typ, welcher konstante Konstruktoren für die drei Formate hat.

```
data OutFormat = OutText | OutJSON | OutTerm
```

Der letzte grundlegende Typ ist `Options`. Dieser wird verwendet um die Optionen zu verarbeiten, die beim Aufruf von CurryInfo angegeben werden können. Dieser ist wie folgt definiert.

```
data Options = Options
    { optVerb           :: Int
    , optHelp           :: Bool
    , optForce          :: Int
```

```
  , optPackage        :: Maybe String
  , optVersion        :: Maybe String
  , optModule         :: Maybe String
  , optType           :: Maybe String
  , optTypeclass      :: Maybe String
  , optOperation      :: Maybe String
  , optOutput         :: OutFormat
  , optClean          :: Bool
  , optShowAll        :: Bool
  , optServer         :: Bool
  , optPort           :: Maybe Int
  , optAllTypes       :: Bool
  , optAllTypeclasses :: Bool
  , optAllOperations  :: Bool
  }
```

Die einzelnen Optionen werden später genauer erklärt. Dieser Typ tritt aber wegen dem Feld `optVerb` in vielen Operationen vor. Dieser stellt die Verbosität der Anwendung ein und bestimmt so, welche Nachrichten zusätzlich zum Ergebnis ausgegeben werden.

Es gibt verschiedene Arten von Statusnachrichten, die ausgegeben werden können. Es gibt verschiedene Operationen, mit denen diese Nachrichten ausgegeben werden könne. Diese sind `printStatusMessage`, `printDetailMessage` und `printDebugMessage`.

Die meisten Informationen haben Typen, die in anderen Paketen definiert sind. Für eine Art von Information wird hier aber ein eigener Typ definiert. Manche Informationen sind Textausschnitte von Dateien.

Statt den tatsächlichen Textausschnitt im Cache zu speichern, wird eine Referenz auf diesen gespeichert. Diese wird durch den Typ `Reference` wie folgt definiert.

```
data Reference = Reference String Int Int
```

Dieser Typ hat drei Bestandteile. Der `String` ist der Pfad zur referierten Datei. Der erste `Int` ist die Zeile, in der referierte Textausschnitt beginnt. Hierbei wird von 0 an gezählt. Der zweite `Int` ist die Nummer der ersten Zeile, die nicht mehr zum Textausschnitt gehört.

Man kann es sich wie einen Ausschnitt einer Liste vorstellen. Der erste Index gehört zum ersten Element des Ausschnitts und der zweite Index gehört zum ersten Element, welches nicht mehr zum Ausschnitt gehört. Der erste Index ist also inklusive und der zweite Index ist exklusive. Dafür kann man sich das folgende Beispiel ansehen.

```
Reference path 20 23
```

Der Pfad der Datei ist irrelevant, daher wird dieser hier ausgelassen. Der Ausschnitt beginnt in der Zeile mit der Nummer 20. Da der zweite Index 23 ist,

enthält die angegebene Referenz die Zeilen 20, 21 und 22. Zeile 23 gehört nicht mehr zu diesem Ausschnitt.

4.4 Commands

Das erste Module, welches vorgestellt wird, ist `Commands`. Wie bereits erklärt, gibt es einige Anwendungen, die grundsätzlich im Curry Code eingebunden werden könnten, aber hier nicht gemacht wurde. Stattdessen müssen diese Anwendungen extern ausgeführt werden. Dafür sind folgende Schritte nötig.

- Aufruf der Anwendung
- Übergabe von Argumenten an die Anwendung
- Lesen der Ausgabe der Anwendung

Dies ist mit dem Paket *io-extra*[7] möglich. Dieses enthält Operationen für das Aufrufen von externen Kommandos und dem Auslesen ihrer Ausgabe. Im Modul `Commands` wird die Operation `evalCmd` verwendet, deren Signatur wie folgt aussieht.

```
evalCmd :: String -> [String] -> String
-> IO (Int, String, String)
```

Das erste Argument ist das Kommando, welches man aufrufen will. Das zweite Argument, also die Liste von `Strings`, ist die Liste von Argumenten, die dem Aufruf beigefügt werden. Das dritte und letzte Argument ist die Eingabe über die Standardeingabe, die dem aufgerufenen Kommando übergeben wird. Dieses kann nötig sein, wenn die Anwendung Eingaben während dem Programmablauf benötigt. In dieser Arbeit wird `evalCmd` in einer anderen Operation aufgerufen, die den Ablauf ergänzt.

```
1  runCmd :: Options -> (String, IO (Int, String, String))
2  -> IO (Int, String, String)
3  runCmd opts (cmd, action) = do
4      printDetailMessage opts $
5          "Running command '" ++ cmd ++ "'..."
6      (exitCode, output, err) <- action
7      case exitCode of
8                  127 -> do
9                      printDetailMessage opts
10                         "Command could not be found."
11                 126 -> do
12                     printDetailMessage opts
13                         "Command was not an executable."
14                 0 -> do
```

[7] https://cpm.curry-lang.org/pkgs/io-extra.html, letzter Zugriff: 02.12.2024

```
15                      printDetailMessage opts
16                          "Command finished successfully."
17                  _ -> do
18                      printDetailMessage opts $
19                          "Command failed with exit code '"
20                          ++ show exitCode ++ "'."
21
22      printDebugMessage opts
23          "Command finished with output:"
24      printDebugMessage opts output
25
26      printDebugMessage opts
27          "Command finished with error output:"
28      printDebugMessage opts err
29
30      return (exitCode, output, err)
```

Das zweite Argument ist ein Tupel aus einem `String` und einem `IO` Wert, der dem Ergebnis von `evalCmd` entspricht. Der `String` ist der Name vom Kommando und wird nur in den Statusnachrichten benutzt. Der zweite Wert ist der eigentliche Aufruf von `evalCmd`.

In Zeile 6 wird dieser ausgeführt und in Zeile 7 wird Pattern Matching auf den *exit code* angewendet. Denn es können verschiedene Probleme beim Aufruf auftreten. Hier werden 3 Fälle explizit angegeben. Der Code 127 bedeutet, dass das Kommando nicht gefunden werden konnte. Zum Beispiel könnte es auf einem Linux System nicht auf dem Pfad sein. Der Code 126 bedeutet, dass das Kommando zwar gefunden wurde, aber kein ausführbares Programm ist. Wieder auf Linux könnte es Rechte fehlen, das Programm wirklich auszuführen. Der Code 0 ist das gewünschte Ergebnis, denn mit diesem Code ist alles korrekt abgelaufen. Der Rest wird mit der *Wildcard* in Zeile 17 abgefangen.

Mittels `runCmd` kann man allgemein Kommandos ausführen und entsprechend der Situation Nachrichten ausgeben lassen. Im Rest des Moduls sind dann noch explizite Operationen für die verschiedenen Anwendungen beziehungsweise für bestimmte Funktionen der Anwendungen definiert. Hierzu ein Beispiel.

```
cmdCASS :: String -> String -> Module
-> (String, IO (Int, String, String))
cmdCASS path analysis m =
  let
    x@(cmd:args) = if analysis == "FailFree"
      then ["cypm", "exec", "curry-calltypes", "--format=json", m]
      else ["cypm", "exec", "cass", "-f", "JSONTerm", analysis, m]
    action = do
        current <- getCurrentDirectory
        setCurrentDirectory path
        (exitCode, output, err) <- evalCmd cmd args ""
```

```
        setCurrentDirectory current
        return (exitCode, output, err)
  in (unwords x, action)
```

Mit dieser Operation lässt sich CASS für eine bestimmte Analyse aufrufen. Das erste Argument ist der Pfad, in welchem das zu analysierende Modul liegt, und das zweite Argument ist die Analyse. Der Aufruf von *curry-calltypes* ist ebenfalls in dieser Operation und wird verwendet, wenn die Analyse `"FailFree"` ist.

4.5 Checkout

Dieses Modul ermöglicht das Nutzen der *checkout* Funktion von CPM. Für diverse Anwendungen und Operationen, zum Beispiel bei CASS, benötigt man den Code von Paketen. Mit *checkout* kann man eine bestimmte Version eines Pakets an einen angegebenen Pfad kopieren. Um dies in Curry durchführen zu können gibt es die Operation `checkoutIfMissing`.

```
1  checkoutIfMissing :: Options -> Package -> Version
2  -> IO (Maybe String)
3  checkoutIfMissing opts pkg vsn = do
4      printDetailMessage opts $
5          "Computing checkout path for package '" ++ pkg ++
6          "' with version '" ++ vsn ++ "'..."
7      path <- getCheckoutPath pkg vsn
8      printDetailMessage opts $
9          "Checkout path: " ++ path
10     printDetailMessage opts
11         "Determining if checkout is necessary..."
12     b <- doesDirectoryExist path
13     case b of
14         True -> do
15             printDetailMessage opts
16                 "Directory already exists. Checkout unnecessary."
17             return $ Just path
18         False -> do
19             printDetailMessage opts
20                 "Directory does not exist. Checkout necessary."
21             printDetailMessage opts $
22                 "Creating checkout..."
23             runCmd opts $ cmdCheckout path pkg vsn
24             printDetailMessage opts
25                 "Checkout created."
26             return $ Just path
```

Zunächst bestimmt die Operation, ob ein *checkout* nötig ist, und führt diesen entsprechend aus. Die Operation `getCheckoutPath` gibt den Pfad zurück, an

welchem der *checkout* ausgeführt werden soll. Der Rest überprüft, ob dieser Pfad bereits existiert und führt den *checkout* aus, wenn dieser Pfad nicht existiert.

4.6 Analysis

Dieses Modul hat alle Operationen, die man zum Ausführen von Analysen mit CASS und *curry-calltypes* benötigt. Wenn die Analyse durchgeführt wurde, dann kann das Ergebnis nach der angefragten Entität durchsucht werden. Das Ergebnis kann dann zurückgegeben werden.

Da zum Beispiel CASS Ergebnisse für mehrere Entitäten ausgibt, können diese hier im Cache gespeichert werden. Dies ist generell für alle Analysen möglich, daher ist eine generische Operation für Analysen implementiert. Diese ist wie folgt definiert.

```
1  analyseWithCASS :: (ErrorMessage a, Path a) =>
2  Options -> Package -> Version -> Module -> String -> String
3  -> String -> (String -> a) -> IO (Maybe String)
4  analyseWithCASS opts pkg vsn m name analysis field constructor
5      = do
6          mpath <- checkoutIfMissing opts pkg vsn
7          case mpath of
8              Nothing -> do
9                  return Nothing
10             Just path -> do
11                 (_, output, _) <- runCmd opts
12                                       (cmdCASS path analysis m)
13                 case parseJSON output of
14                     Just (JArray jvs) -> do
15                         case getJsonResults jvs of
16                             Nothing -> do
17                                 return Nothing
18                             Just results -> do
19                                 mapM addInformation results
20
21                                 case lookup name results of
22                                     Nothing -> do
23                                         return Nothing
24                                     Just result -> do
25                                         return (Just result)
26                     _ -> do
27                         return Nothing
```

Um die Menge an Code zu reduzieren wurden die Ausgaben von Nachrichten nur in der Ausarbeitung weggelassen. Damit man CASS beziehungsweise *curry-calltypes* ausführen kann, benötigt man ein kompilierbares Programm, welches man analysiert. Zunächst wird der Pfad des zu analysierenden Moduls in Zeile 6 mittels `Checkout` bestimmt.

Wenn dieser Pfad gefunden wurde, wird in Zeile 13 die Ausgabe geparst. Die Operation `getJSONResults` in Zeile 15 ist eine lokal definierte Operation, welche die Ergebnisse aus dem JSON Ausdruck extrahiert. Das Ergebnis dieser Operation ist vom Typ `Maybe [(String, String)]`. Der erste `String` ist der Name einer analysierten Entität, zum Beispiel eine Operation, und der zweite `String` ist das Ergebnis der Analyse.

Die Operation `addInformation` wird in Zeile 19 auf alle Listenelemente angewendet. `addInformation` ist ebenfalls eine lokale Operation. Diese speichert das Ergebnis für die Entität im Cache. Als letztes wird im `case`-Ausdruck in Zeile 21 nach der angefragten Entität gesucht.

Mit der generischen Operation definiert, lassen sich für alle integrierten Analysen eine eigene Operation anlegen. Zwei Beispiele hierfür sehen so aus.

```
analyseDeterministicWithCASS ::
Options -> Package -> Version -> Module -> Operation
-> IO (Maybe String)
analyseDeterministicWithCASS opts pkg vsn m o = do
    analyseWithCASS opts pkg vsn m o
        "Deterministic" "cass-deterministic"
        (CurryOperation pkg vsn m)

analyseTerminatingWithCASS ::
Options -> Package -> Version -> Module -> Operation
-> IO (Maybe String)
analyseTerminatingWithCASS opts pkg vsn m o = do
    analyseWithCASS opts pkg vsn m o
        "Terminating" "cass-terminating"
        (CurryOperation pkg vsn m)
```

4.7 Interface

Das Modul `Interface` hat Operationen um *icurry* Dateien zu lesen, deren Inhalt zu parsen und daraus die gewünschten Informationen zu extrahieren. Zunächst wird die Operation `readInterface` gezeigt, welche die *icurry* Datei des angefragten Moduls ausliest und parst.

```
1  readInterface :: Options -> Package -> Version -> Module
2  -> IO (Maybe Interface)
3  readInterface opts pkg vsn m = do
4      mpath <- checkoutIfMissing opts pkg vsn
5      case mpath of
6          Just path -> do
7              icurry <- icurryPath pkg vsn m
8              b <- doesFileExist icurry
9              case b of
10                 True -> do
```

```
11                      result <- readCurryInterfaceFile icurry
12                      return $ Just result
13                  False -> do
14                      runCmd opts (cmdCYPMInstall path)
15                      runCmd opts (cmdCurryLoad path m)
16
17                      result <- readCurryInterfaceFile icurry
18                      return $ Just result
19          Nothing -> do
20              return Nothing
```

Für das Interface ist es ebenfalls nötig einen *checkout* mit CPM durchzuführen. Entsprechend gibt es auch hier ein Aufruf von `checkoutIfMissing` in Zeile 4. Mit der Operation `icurryPath` in Zeile 7 wird der spezifische Pfad zur *icurry* Datei bestimmt.

Wenn diese bereits existiert, welches dem `True` Fall in den Zeilen 10 bis 12 entspricht, dann wird diese mit `readCurryInterfaceFile` gelesen und geparst. Sollte die Datei noch fehlen, welches dem `False` Fall in den Zeilen 13 bis 18 entspricht, dann wird vorher mit zwei externen Aufrufen diese Datei erzeugt.

Mit `cmdCYPMInstall` wird aus dem Modul, welches mit dem *checkout* kopiert wurde, durch einen Aufruf von CYPM ein kompilierbares Programm. Der darauffolgende Aufruf mit der Operation `cmdCurryLoad` startet die REPL und lädt das genannte Modul. Dadurch wird automatisch auch die *icurry* Datei erzeugt. Die REPL wird daraufhin direkt wieder beendet. Danach entspricht es dem `True` Fall, da jetzt die Datei existiert.

Das Ergebnis ist vom Typ `Interface`, welches in *curry-interface* definiert wird. Dieser enthält alle Informationen, die im Interface zu finden sind. Was aber noch fehlt sind Operationen um bestimmte Informationen zu extrahieren. Solche werden in dieser Arbeit definiert.

Ein Beispiel hierfür ist das Extrahieren von Typklassendeklarationen. Dieses sieht wie folgt aus.

```
getClassDecl :: String -> [IDecl] -> Maybe IDecl
getClassDecl c = find (\decl -> Just c == getClassName decl)

getClassName :: IDecl -> Maybe String
getClassName decl = case decl of
    IClassDecl _ name _ _ _ _ _ -> Just $ idName $ qidIdent name
    _                          -> Nothing
```

`getClassName` bestimmt den Namen eines Typs aus seiner Deklaration und `getClassDecl` nutzt diese um die passende Deklaration zu finden. Der Type `IDecl` ist ein weiterer Typ, der in *curry-interface* definiert wurde. Das Interface hat eine Liste von Deklarationen, die diesen Typ haben. Durch Anwendung von `getClassDecl` auf diese Liste findet man die entsprechende Deklaration und kann diese weiter verarbeiten. So kann man zum Beispiel aus der gefunden Deklaration die Liste der Methoden bestimmen.

```
getClassMethods :: IDecl -> Maybe [Method]
getClassMethods decl = case decl of
    IClassDecl _ _ _ _ _ methods _  ->
        Just $ map (pPrint . ppMethodDecl defaultOptions) methods
    _                               ->
        Nothing
```

Der Ausdruck `pPrint . ppMethodDecl defaultOptions` wandelt eine Methodendeklaration, die wieder einen eigenen Typ in *curry-interface* hat, zu einem String. Der Type `Method` ist nur ein Typalias für `String`.

4.8 Generator

In diesem Modul sind Operationen für jede Anfrage um die entsprechende Information zu generieren. Dieses ist abhängig von all den vorher beschriebenen Modulen um diese Generierung durchführen zu können. Ein generischer Typ wird benutzt um die Struktur der Operationen vorzugeben.

```
type Generator a b = Options -> a -> IO (Maybe b)
```

Die Typvariable `a` steht für den Eingabetyp und die Typvariable `b` steht für den Typ der Information. Eine generierende Operation nimmt also die Optionen, die vorrangig für Statusnachrichten benötigt wird, und die Entität, zu der die Anfrage gestellt werden. Da bei der Generierung möglicherweise Seiteneffekte auftreten müssen, ist das Ergebnis im `IO` Kontext, und da während der Generierung Fehler auftreten könnten, ist das Ergebnis auch im `Maybe` Kontext. Der Ergebnistyp ist generisch, da die Informationen unterschiedliche Formen annehmen können.

Im weiteren werden einige ausgewählte Beispiele vorgeführt und so gezeigt, wie der Code der anderen Module für die Generierung benötigt wurde. Das erste Beispiel ist ein einfaches.

```
gPackageName :: Generator CurryPackage String
gPackageName opts (CurryPackage pkg) = do
    printDetailMessage opts $
        "Generating name for package '" ++ pkg ++ "'..."
    printDebugMessage opts $
        "Name is: " ++ pkg
    let res = pkg
    printDebugMessage opts $
        "Result is: " ++ res
    printDetailMessage opts
        "Generating finished successfully."
    return $ Just res
```

Diese Operation generiert den Namen des übergebenen Pakets. Dies ist trivial, weshalb der Großteil der Operation Statusnachrichten erzeugt. Als nächstes

wird gezeigt, wie ein Analyseergebnis generiert wird. Dies wird durch eine generische Operation erzielt, die wie folgt definiert ist.

```
1  generateOperationAnalysisWithCASS ::
2  Show b => String -> (Options -> Package -> Version -> Module
3  -> Operation -> IO (Maybe b)) -> Generator CurryOperation b
4  generateOperationAnalysisWithCASS
5      desc analysis opts (CurryOperation pkg vsn m o) = do
6          printDetailMessage opts $
7              "Generating " ++ desc ++
8              " analysis of operation '" ++ o ++
9              "' of module '" ++ m ++
10             "' of version '" ++ vsn ++
11             "' of package '" ++ pkg ++ "'..."
12         mres <- analysis opts pkg vsn m o
13         processAnalysisResult opts mres
```

Das erste Argument ist eine Beschreibung der Analyse. Im einfachsten Fall ist dies der Name der Analyse. Das zweite Argument ist der Aufruf der Analyse, welche in Zeile 12 durchgeführt wird. Die Operation `processAnalysisResult` ist eine simple Operation, die je nachdem, ob ein Ergebnis berechnet werden konnte, passende Nachrichten ausgibt. Das Ergebnis wird ansonsten unverändert zurückgegeben. Ein Beispiel, wie diese generische Operation verwendet werden kann, ist im folgenden.

```
gOperationCASSDeterministic :: Generator CurryOperation String
gOperationCASSDeterministic =
    generateOperationAnalysisWithCASS "deterministic"
        analyseDeterministicWithCASS
```

Wenn eine generierende Operation das Interface eines Moduls benötigt, so gibt es auch hierfür eine generische Operation. Diese ist wie folgt definiert.

```
1  generateFromInterface :: Show b => Package -> Version -> Module
2  -> String -> (Interface -> Maybe b) -> Options -> IO (Maybe b)
3  generateFromInterface pkg vsn m desc selector opts = do
4      minterface <- readInterface opts pkg vsn m
5      case minterface of
6          Nothing -> do
7              printDetailMessage opts "Failed to read interface."
8              printDetailMessage opts "Generating failed."
9              return Nothing
10         Just interface -> do
11             printDetailMessage opts $
12                 "Reading " ++ desc ++ " from interface..."
13             case selector interface of
14                 Nothing -> do
```

```
15                      printDetailMessage opts
16                          "Failed to find information in interface."
17                      printDetailMessage opts
18                          "Generating failed."
19                      return Nothing
20                  Just res -> do
21                      printDebugMessage opts $
22                          "Result: " ++ show res
23                      printDetailMessage opts
24                          "Generating finished successfully."
25                      return $ Just res
```

Das Argument `desc` ist wie bei CASS Analysen eine Beschreibung der angefragten Information. Im einfachsten Fall ist das der Name der Information. Das Argument `selector` hat den Typ `Interface -> Maybe b`. Dies ist die Operation, welche die eigentliche Information aus dem Interface extrahiert.

In Zeile 4 wird das Interface mit der Operation `readInterface` gelesen. Wenn dies erfolgreich ist, wird in Zeile 13 das Argument `selector` verwendet um die angefragte Information zu bekommen. Der Rest der Operation ist nur zum Ausgeben von Statusnachrichten und dem Abfangen von Fehlschlägen. Die Information wird unverändert als Ergebnis zurückgegeben.

Als ein Beispiel kann man sich eine Operation anschauen, welche die Liste der Operationen eines Moduls bestimmt. Diese sieht wie folgt aus.

```
gModuleOperations :: Generator CurryModule [String]
gModuleOperations opts (CurryModule pkg vsn m) = do
        printDetailMessage opts $
            "Generating exported operations for module '" ++ m ++
            "' of version '" ++ vsn ++
            "' of package '" ++ pkg ++ "'..."
        generateFromInterface pkg vsn m "operations"
            operationsSelector opts
    where
        operationsSelector interface = Just (
            catMaybes $
            map getOperationName $
            getOperations $
            getDeclarations interface)
```

In der lokalen Operation `operationsSelector` werden diverse Operationen nacheinander aufgerufen. Zunächst wird die Liste der Deklarationen mit `getDeclarations` ausgewählt und von dieser werden die Deklarationen für Operationen mit `getOperations` ausgewählt. Von diesen sind nur die Namen von Interesse, die mit `getOperationName` ausgewählt werden.

Da das Ergebnis von `getOperationName` im `Maybe` Kontext ist, ist das Ergebnis der Liste vom Typ `[Maybe String]`. Mit der Operation `catMaybes` wird diese Liste zum Typ `[String]`.

4.9 Printer

Mit den bisher vorgestellten Modulen ist das Generieren der Informationen möglich. Diese Informationen müssen als nächstes in eine sinnvolle Form für die Ausgabe umgewandelt werden. Für die meisten Informationen ist so eine Umwandlung nicht nötig, da die Typen wie `Bool` und `Int` schon aussagekräftig genug sind. Manche Informationen können aber eine reduzierte Form im Cache haben. Für diese Informationen ist eine solche Umwandlung sinnvoll. Die Operationen, die diese Umwandlung durchführen, sind vom folgenden Typ.

```
type Printer b = Options -> b -> IO String
```

Die Typvariable `b` steht wieder für den Typ der Information. Die Information wird also verwendet um einen `String` zu erzeugen, der in der Ausgabe von CurryInfo benutzt wird. Das Ergebnis ist im `IO` Kontext, weil für manche Informationen ein Zugriff auf die Außenwelt nötig ist.

Ein Typ, für welche diese Operation nötig ist, ist `Reference`. `Reference` enthält nur den Pfad zur Datei und Zeilenangaben. Der eigentliche Textausschnitt ist nicht im Cache gespeichert.

Wenn solch ein Textausschnitt angefragt wird, ist eine Ausgabe des Pfads und Zeilenangaben nicht sinnvoll. Zum einen ist dies nicht wirklich die angefragte Information, da eigentlich der Textausschnitt erwartet wird. Zum anderen können auch Komplikationen mit dem Pfad existieren.

Während es zwar möglich, dass CurryInfo auf der selben Maschine läuft, auf welcher die Anfragen gestellt wird, muss dies nicht immer zutreffen. Es ist auch möglich die Anwendung auf einer anderen Maschine laufen zu lassen. Dann hat man womöglich keinen Zugriff auf den ausgegebenen Pfad.

Deswegen gibt es die folgende Operation, die allgemein für Werte des Typs `Reference` verwendet werden kann um den entsprechenden Textausschnitt zu erhalten. Damit kann dieser als Ergebnis ausgegeben werden. Definiert ist die Operation wie folgt.

```
printFromReference :: Options -> Reference -> IO String
printFromReference opts (Reference path start end) = do
    b <- doesFileExist path
    case b of
        False -> do
            printDebugMessage opts $ "File '" ++ path ++
                "' does not exist."
            return "FAILED DUE TO FILE NOT EXISTING"
        True -> do
            printDebugMessage opts $ "Reading from file '" ++
                path ++ "'..."
            content <- readFile path
            let ls = lines content
            return (unlines (slice start end ls))
```

4.10 Reader und Writer

Die Module `Reader` und `Writer` sind eng verbunden, da sie auf die gleichen Dateien zugreifen. Wie die Namen bereits implizieren, ist der Code von `Reader` zuständig für das Lesen und der Code von `Writer` zuständig für das Schreiben.

Die Informationen, die generiert werden, werden von CurryInfo in JSON Dateien gespeichert. Entsprechend liest die Operation `readInfomation` im Modul `Reader` diese Dateien. Sie ist wie folgt definiert.

```
readInformation :: Path a => Options -> a
-> IO (Maybe [(String, JValue)])
readInformation opts obj = do
  printDebugMessage opts "Detemining path to json file..."
  path <- getJSONPath obj
  printDebugMessage opts $ "Path to json file: " ++ path
  b <- doesFileExist path
  case b of
    False -> do
      printDebugMessage opts
        "json file does not exist.\nReading failed."
      return Nothing
    True -> do
      printDebugMessage opts "json file exists."
      jtext <- readFile path
      printDebugMessage opts $
        "Read json file.\n" ++ jtext
      case parseJSON jtext of
        Just (JObject fields) -> do
          printDebugMessage opts
            "Parsing json file succeeded."
          return $ Just fields
        Nothing -> do
          printDebugMessage opts
            "Parsing failed."
          return Nothing
        _ -> do
          printDebugMessage opts
            "Parsing succeeded, but structure was not expected."
          return Nothing
```

Die Typvariable `a` steht wider für den Eingabetyp der angefragten Entität. Die `JValues` sind die Einträge der angefragten Informationen in der JSON Datei.

Zunächst wird der Pfad der JSON Datei bestimmt. Dann wird versucht diese zu lesen. Wenn die Datei existiert, wird sie gelesen und geparst. All die JSON Dateien enthalten `objects`. Entsprechend muss nur für den Fall ein richtiges Ergebnis bestimmt werden. Sollte die JSON Datei einen anderen Typ enthalten

oder sollte etwas beim Einlesen fehlgeschlagen sein, so schlägt die Operation mit `Nothing` fehl.

Im Erfolgsfall wird die Liste der Felder als Ergebnis zurückgegeben. Diese muss entsprechend noch nach der gewünschten Information durchsucht werden.

Das Modul `Writer` enthält die Operation `writeInformation` und den Operator `(<+>)`. `writeInformation` ist das Gegenstück von `readInformation` und überschreibt die JSON Dateien mit neuen Informationen. Der Operator ist zum Hinzufügen von neuen Informationen zu den alten, die vorher gelesen wurden. Diese beiden sind wie folgt definiert.

```
(<+>) :: [(String, a)] -> [(String, a)] -> [(String, a)]
info1 <+> info2 =
    nubBy (\(k1, _) (k2, _) -> k1 == k2) (info1 ++ info2)

writeInformation :: Path a => a -> [(String, JValue)] -> IO ()
writeInformation obj fields = do
    let jtext = (ppJSON . JObject) fields
    path <- getJSONPath obj
    writeFile path jtext
```

Der Operator `(<+>)` konkateniert die Liste mit den alten Information und die Liste mit den neuen Informationen zusammen. Die Ergebnisliste wird dann mit der Operation `nubBy` von Duplikaten befreit, die durch den Feldnamen bestimmt werden. Hier wird ausgenutzt, dass `nubBy` den ersten Wert von Duplikaten behält. Wenn die Liste mit den neuen Informationen als erstes Argument und die Liste mit den alten Informationen als zweites Argument benutzt werden, so werden die neuen Informationen priorisiert und die alten werden überschrieben.

Die Operation `writeInformation` nimmt wieder den Eingabetyp um den Pfad zur JSON Datei zu bestimmen. Die Liste der Felder wird dann in ein JSON `object` umgewandelt und als Text in die Datei geschrieben.

4.11 Configuration

Das Module `Configuration` ist das Bindeglied zwischen den anderen Modulen, vor allem den Modulen `Reader`, `Writer`, `Printer` und `Generator`.

Ein Ziel für CurryInfo ist die Möglichkeit einfach neue Anfragen hinzuzufügen. Dies wird in diesem Modul umgesetzt. Für jede Art von Entität gibt es eine eigene Konfiguration, die vorgibt, welche Anfragen zur Verfügung stehen und wie diese bearbeitet werden. Hierzu folgt als Beispiel die Konfiguration für Pakete.

```
packageConfiguration :: Configuration CurryPackage
packageConfiguration =
    [ registerRequest
        "package"
        "\t\t\tThe name of the package"
        gPackageName
```

```
        pPackageName
    , registerRequest
        "versions"
        "\t\t\tThe versions available of the package"
        gPackageVersions
        pPackageVersions
    ]
```

Die Operation `registerRequest` nimmt die gegebenen Argumente und aus diesen formt sie die Anfrage. Die ersten zwei Argumente sind simpel. Das erste ist der Name der Anfrage und das zweite ist eine Beschreibung. Der Name ist das Argument, welches man der Anwendung gibt um diese Information anzufragen. Die Beschreibung wird im Hilfetext angezeigt und soll die Information erklären.

Die restlichen zwei Argumente sind Operationen, welche die Logik der Anfrage vorgibt. Die erste Operation ist jene, welche die Information generiert. Im Normalfall sollte diese Operation im Modul `Generator` definiert sein. Die andere Operation ist jene, welche für die Information einen `String` für die Ausgabe erzeugt. Im Normalfall sollte diese Operation im Modul `Printer` definiert sein.

Aus diesen vier Bestandteilen erzeugt die Operation `registerRequest` die Anfrage. Dies wird durch einen eigenen Typ erreicht, welcher diese Operationen zu neuen kombiniert. Dieser Typ ist wie folgt definiert.

```
data RegisteredRequest a = RegisteredRequest
    { request ::
        String
    , description ::
        String
    , extraction ::
        (Options -> [(String, JValue)]
            -> IO (Maybe (JValue, String)))
    , generation ::
        (Options -> a -> IO (Maybe (JValue, String)))
    }
```

`request` und `description` sind einfach nur der Name der Anfrage und ihre Beschreibung. Die anderen beiden Felder sind die Operationen, welche die angefragte Information aus der JSON Datei extrahiert oder die Information generiert. `registerRequest` ist wie folgt implementiert.

```
registerRequest
:: ConvertJSON b =>
String -> String -> Generator a b -> Printer b
-> RegisteredRequest a
registerRequest req desc generator printer =
        RegisteredRequest req desc
            createExtraction createGeneration
```

Die Erzeugung der Felder `extraction` und `generation` wird durch lokale Operationen umgesetzt. `createExtraction` ist wie folgt definiert.

```
1  createExtraction opts infos = do
2      printDebugMessage opts $
3          "Looking for information for request '" ++ req ++ "'..."
4      case lookup req infos of
5          Nothing -> do
6              printDebugMessage opts "Information not found."
7              return Nothing
8          Just jv -> do
9              printDebugMessage opts "Information found."
10             printDebugMessage opts "Reading information..."
11             case fromJSON jv of
12                 Nothing -> do
13                     printDebugMessage opts "Reading failed."
14                     return Nothing
15                 Just info -> do
16                     printDebugMessage opts "Reading succeeded."
17                     printDebugMessage opts "Creating output..."
18                     output <- printer opts info
19                     printDebugMessage opts $
20                         "Finished with (" ++ ppJSON jv ++
21                         ", " ++ output ++ ")."
22                     return $ Just (jv, output)
```

`infos` sind die Felder, die aus der JSON Datei gelesen werden. Mit `lookup` in Zeile 4 wird die Information der Anfrage in dieser Liste gesucht. Wenn dies fehlschlägt, dann endet die Operation mit der Rückgabe von `Nothing`. Wenn die Information gefunden wurde, dann wird sie mit `fromJSON` in Zeile 11 geparst.

Das Ergebnis wird dann mittels `printer` in Zeile 18 aus der Information erzeugt. Diese wird zusammen mit dem JSON Wert zurückgegeben. Der JSON Wert wird für das erneute Schreiben der JSON Datei benötigt.

```
1  createGeneration opts obj = do
2      printDebugMessage opts $
3          "Generating information for request '" ++ req ++ "'..."
4      res <- generator opts obj
5      case res of
6          Nothing -> do
7              printDebugMessage opts "Generating failed."
8              return Nothing
9          Just info -> do
10             printDebugMessage opts "Generating succeeded."
11             let jv = toJSON info
12             printDebugMessage opts "Creating output..."
13             output <- printer opts info
```

```
14              printDebugMessage opts $
15                  "Finished with (" ++ ppJSON jv ++
16                  ", " ++ output ++ ")."
17              return $ Just (jv, output)
```

Für `generation` ist der finale Teil wie bei `extraction` definiert. Statt aus einer Liste von JSON Feldern die Information zu suchen, wird diese direkt generiert. Dies passiert in Zeile 4.

Im Erfolgsfall muss aus der Information der JSON Wert erzeugen werden. Dies passiert mit `toJSON` in Zeile 11. Der Rückgabewert enthält auch hier den `String` der Ausgabe und dem JSON Wert.

Durch diese Implementierung ist das Hinzufügen neuer Anfragen einfach. Es muss nur eine Operation, welche die Information generiert, und eine Operation, welche die Information zu einem `String` für die Ausgabe umwandelt, angeben. Der Typ der Information wird dann durch das statische Typsystem bestimmt.

5 CurryInfo - Anwendung

In diesem Kapitel wird erklärt wie CurryInfo zu benutzen ist. Grundsätzlich gibt es zwei Modi wie diese Anwendung benutzt werden kann. Im Terminal-Modus verwendet man es wie eine normale Kommandozeilenanwendung ähnlich wie CASS. Man gibt die Anfragen direkt als Argumente im Aufruf und erhält das Ergebnis auf der Kommandozeile als Ausgabe. Im Servermodus läuft die Anwendung dauerhaft bis sie abgeschaltet wird. Solange sie läuft kann man über einen bestimmten Port Anfragen in Form von Nachrichten senden. Über den selben Port werden die Ergebnisse als Nachricht zurückgesendet. Anders als im Terminal-Modus können so mehrere Anfragen nacheinander gestellt werden, bevor die Anwendung beendet wird.

5.1 Terminal-Modus

Im Terminal-Modus läuft CurryInfo als eine Kommandozeilenanwendung. Man ruft diese mit Optionen und Argumenten auf und erhält eine Antwort. Es gibt hier aber keine Interaktion während das Programm läuft, alle Eingaben müssen beim Aufruf erfolgen. Das Kommando sieht wie folgt aus.

```
curry-info [options] <requests>
```

Man gibt die Entität, für welche man Informationen erfahren möchte, in den Optionen an. Diese kann man wie folgt angeben.

- Paket: `-p <pkg>` oder `--package=<pkg>`
- Version: `-x <vsn>` oder `--version=<vsn>`
- Modul: `-m <mod>` oder `--module=<mod>`
- Typ: `-t <type>` oder `--type=<type>`
- Typklasse: `-c <class>` oder `--typeclass=<class>`
- Operation: `-o <op>` oder `--operation=<op>`

Die Anwendung achtet darauf, dass diese Optionen nur mit sinnvollen Werten eingestellt werden. Für eine Entität muss die gesamte Hierarchie angegeben werden. Wenn man also für ein Modul Informationen anfragen will, so muss man das Paket, die Version und das Modul angeben. Wenn man zum Beispiel gleichzeitig einen Typ und eine Operation angibt, dann wird CurryInfo keine Anfragen bearbeiten. Denn diese Angabe ergibt keine Entität. Einige Beispiele für korrekte Verwendungen ohne ihre Ausgabe folgen hier.

```
curry-info -p json versions
curry-info --package=directory -x 3.0.0 -m System.Directory
    types operations
curry-info -p base -x 3.2.0 -m Prelude -t Bool constructors
    definition
```

Im folgenden werden die Optionen genauer vorgestellt. Die einfachste Option ist die Hilfeoption. Diese lässt CurryInfo einen Hilfetext ausgeben, welcher die Anwendung und ihre Optionen erklärt. Die Anwendung wird auch direkt danach beendet, es können also keine zusätzlichen Anfragen bearbeitet werden. Diese Option kann eine von diesen Formen annehmen.

- `-h`
- `-?`
- `--help`

Die nächste Option ist zum Einstellen der Verbosität. Dies betrifft die Statusnachrichten, deren genaue Bedeutung bisher ausgelassen wurde. Mit dieser Option kann man einstellen, welche Statusnachrichten während dem Programmablauf ausgegeben werden. Es ist auch möglich alle Nachrichten auszuschalten. Dafür wählt man eine von vier Stufen aus. Je höher eine Stufe ist, desto mehr Nachrichten werden ausgegeben. Außerdem sind sie inklusiv, eine Stufe enthält also alle Nachrichten, die auch die vorherigen Stufen enthalten. Die Stufen sehen wie folgt aus, wobei Stufe 1 die Standardeinstellung ist.

- 0: Keine Nachrichten, nur das Ergebnis
- 1: Nachrichten, die den aktuellen Status angeben (zum Beispiel, welche Anfrage im Moment bearbeitet wird)
- 2: Nachrichten, die die durchgeführten Aktionen angibt (zum Beispiel einen *checkout*)
- 3: Nachrichten, die alle Details enthalten (zum Beispiel Pfade zu Dateien, Ergebnisse von Zwischenberechnungen und Kommandos, die ausgeführt werden)

Um die Verbosität einzustellen, kann man die Option in ein von zwei Wegen angeben.

- `-v<n>`
- `--verbose=<n>`

Die nächste Option ist die Force Option. Mit dieser kann man einstellen, wie erzwungen die Generierung von Informationen durchgeführt werden soll. Auch diese ist in Stufen unterteilt, wobei eine höhere Stufe die Generierung stärker erzwingt. Die Stufen sehen wie folgt aus, wobei Stufe 1 die Standardeinstellung ist.

- 0: Keine Generierung
- 1: Generierung nur, wenn Information noch nicht im Cache ist

- 2: Immer Generierung, auch wenn Information bereits im Cache ist

Stufe 1 entspricht hier dem Normalfall, der bereits in der Arbeit beschrieben wurde. Die Anwendung versucht erst die angefragte Information im Cache zu finden und nur wenn dies fehlschlagen sollte, wird die Information generiert. Will man diese Generierung komplett unterbinden, so kann man dies mit Stufe 0 erreichen. Stufe 2 wiederum ist der Extremfall, denn hier wird die möglicherweise vorhandene Information ignoriert und im Erfolgsfall der Generierung mit den neuen Ergebnis überschrieben.

Will man die Force Option einstellen, so kann man dies mit einem der zwei folgenden Möglichkeiten machen.

- `-f<n>`
- `--force=<n>`

Für die Ergänzungen zum normalen Ablauf von CurryInfo müssen bestimmte Optionen explizit angegeben werden. Um Informationen für alle Entitäten in einem Modul anzufragen kann man eine der drei folgenden Optionen angeben.

- `--alltypes` für Typen
- `--alltypeclasses` für Typklassen
- `--alloperations` für Operationen

Der Aufruf mit einer dieser Optionen sieht fast genauso aus wie bei einer normalen Anfrage. Statt einen Typ, eine Typklasse oder eine Operation als Entität anzugeben, gibt man das Modul an Die Anfragen selbst müssen zu der Entität der Option passen. Für die Option `--alltypes` müssten die Anfragen also für Typen sein. Die Anfragen werden dann für alle entsprechenden Entitäten im angegebenen Modul abgearbeitet. Dafür folgen zwei Aufrufe als Beispiel.

```
curry-info -p directory -x 3.0.0 -m System.Directory
    -o doesFileExist cass-deterministic

curry-info -p directory -x 3.0.0 -m System.Directory
    --alloperations cass-deterministic
```

Der erste Aufruf zeigt nur das Ergebnis für die Operation `doesFileExist` an, während der zweite Aufruf die Ergebnisse für alle Operationen im Module `System.Directory` anzeigt. Dies beinhaltet auch `doesFileExist`.

Es existiert auch eine Option, die alle im Cache liegenden Informationen einer Entität ausgibt. Diese Option ist `--showall`. Man nennt wie normal die Entität an, aber lässt die Anfragen wegfallen. Das Ergebnis enthält alle Informationen, die aktuell für die Entität im Cache gespeichert sind. Ein entsprechender Aufruf sieht wie folgt aus.

```
curry-info -p directory -x 3.0.0 -m System.Directory
    -o doesFileExist --showall
```

`--showall` kann man auch mit den Optionen wie `--alloperations` kombinieren.

Das Löschen vom Cache ist ebenfalls mit einer Option möglich. Diese ist `--clean`. Es gibt zwei Möglichkeiten diese Option zu verwenden. Man kann eine Entität explizit angeben oder lässt diese aus. Wenn man eine Entität angibt, dann wird alles zu dieser Entität gelöscht. Dies enthält die Informationen im Cache aber auch die Kopien vom Paket, die mit *checkout* angelegt wurden. Mit den folgenden Aufrufen kann man sehen, dass die Informationen gelöscht werden.

```
curry-info -p directory -x 3.0.0 -m System.Directory
    -o doesFileExist signature

curry-info -p directory -x 3.0.0 -m System.Directory
    -o doesFileExist --showall

curry-info -p directory -x 3.0.0 -m System.Directory
    -o doesFileExist --clean

curry-info -p directory -x 3.0.0 -m System.Directory
    -o doesFileExist --showall
```

Beim ersten Aufruf wird die Information *signature* generiert, falls sie noch nicht vorhanden ist. Der zweite Aufruf zeigt, dass die Information wirklich im Cache gespeichert ist. Der dritte Aufruf löscht diesen und der letzte Aufruf zeigt, dass wirklich keine Informationen für diese Entität im Cache sind.

Wenn man diese Option ohne Angabe einer Entität verwendet, dann wird der gesamte Cache gelöscht. Auch hier beinhaltet dies alle Kopien von Paketen.

Die letzte Option ist `--output=<format>`. Mit diesem kann man einstellen, welches Format für die Ausgabe des Ergebnisses verwendet werden soll. Zum aktuellen Zeitpunkt gibt es die folgenden Möglichkeiten.

- Text: Von Menschen lesbarer Text
- JSON: JSON Ausdruck
- CurryTerm: Curry Term vom Typ `[(String, String)]`

5.2 Servermodus

Die andere Möglichkeit CurryInfo zu verwenden ist der Servermodus. In diesem Modus bearbeitet die Anwendung nicht direkt Anfragen, die diesem beim Aufruf gestellt werden. Stattdessen läuft die Anwendung dauerhaft als ein Server. Diesem kann man in Form von Nachrichten Anfragen stellen.

```
curry-info --server
```

Mit der Option `--server` kann man CurryInfo im Servermodus starten. Dadurch wird ein zufälliger, freier Port für die Kommunikation ausgewählt. Alternativ kann man mit der Option `--port=<port>` auch einen bestimmten Port beim Aufruf vorgeben.

Der Port wird beim Aufruf auf der Kommandozeile angegeben. Jedes beliebige Programm, welches über einen Port Nachrichten senden und empfangen kann, kann für die Kommunikation verwendet werden.

Die Nachrichten haben eine feste Vorgabe für ihre Form. Zunächst werden die Nachrichten, die vom Server gesendet werden, erklärt. Es gibt die zwei folgenden Nachrichten.

- `err <msg>` für Fehler
- `ok <n>` für Ergebnisse

Im Fehlerfall wird eine einzelne Nachricht gesendet, die eine Erklärung für den Fehler enthält. Im Erfolgsfall wird stattdessen *ok* und eine Nummer gesendet. Diese gibt an, aus wie vielen Zeilen die eigentliche Nachricht besteht. Diese wird direkt nach dem *ok* und der Nummer gesendet.

Auf der anderen Seite gibt es zahlreiche Nachrichten, die dem Server gesendet werden können. Als erstes werden die Nachrichten gezeigt, die den Server selbst betreffen. Diese sind die folgenden.

- StopServer
- GetCommands
- GetRequests

Die StopServer Nachricht lässt den Server sich abschalten. Dies ist nötig, denn der Server schaltet sich nach der Abarbeitung einer Anfrage nicht ab. Der Server läuft durchgehend weiter. Deshalb muss dieser mit einer Nachricht dazu animiert werden, dass er sich abschaltet.

GetCommands lässt den Server die Liste der möglichen Nachrichten ausgeben, die man dem Server senden kann. Mit GetRequests kann man erfahren, welche Anfragen existieren. Dieser Nachricht kann man eine Art von Entität mitgeben, dann antwortet der Server nur mit den Anfragen für diese Art von Entität. Dies kann wie folgt aussehen.

```
GetRequests
GetRequests Package
```

Für jede Art von Entität gibt es eine eigene Nachricht, mit welcher man Anfragen an den Server senden kann. Das sind die folgenden.

- RequestPackageInformation

- RequestVersionInformation
- RequestModuleInformation
- RequestTypeInformation
- RequestTypeclassInformation
- RequestOperationInformation

Man muss zu diesen Nachrichten Argumente hinzufügen. Das erste Argument ist die Force Stufe für die Anfrage. Statt dies über eine Option festzulegen, muss man im Servermodus bei jeder Anfrage diese explizit angeben. Das zweite Argument ist das Ausgabeformat und auch dieses muss explizit genannt werden. Dies sind jeweils die gleichen Angaben wie bei den Optionen im Terminal-Modus.

Die restlichen Argumente sind die Entität und die Anfragen. Auch hier muss die Entität in hierarchischer Reihenfolge angegeben werden und die Anfragen folgen als letze Argumente. Dafür folgen ein paar Beispiele.

```
RequestPackageInformation 1 Text
    json versions
RequestModuleInformation 0 JSON
    directory 3.0.0 System.Directory types operations
RequestTypeInformation 2 CurryTerm
    base 3.2.0 Prelude Bool constructors definition
```

Im Servermodus ist es auch möglich Anfragen für alle Typen, Typklassen oder Operationen eines Moduls zu stellen. Dies wird hier durch eigene Nachrichten umgesetzt. Diese sind die folgenden.

- RequestAllTypesInformation
- RequestAllTypeclassesInformation
- RequestAllOperationsInformation

Die ersten Argumente sind wieder die Force Stufe und das Ausgabeformat. Als Entität gibt man das Modul an und die Anfragen müssen für die Entitäten passen. Ein Beispiel dafür folgt.

```
RequestAllOperationsInformation 1 JSON
    directory 3.0.0 System.Directory cass-deterministic
```

6 CurryInfo - Anwendung in anderen Anwendungen

Um den Nutzen von CurryInfo zu zeigen, wurde an einer Beispielanwendung Änderungen vorgenommen um dieses mittels CurryInfo zu verbessern. Die Anwendung, die dafür ausgewählt wurde, ist CASS.

Wie bereits beschrieben verwendet CASS einen eigenen Cache, der aber die Verzeichnisstruktur des zu analysierenden Moduls kopiert. Dadurch sind bereits durchgeführte Analysen unabhängig voneinander, wenn die Module sich nicht im selben Verzeichnis befinden.

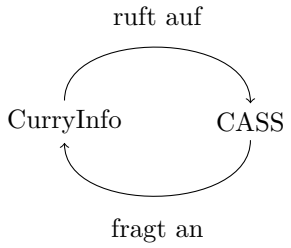
Die Analyse läuft in einem *bottom-up* Prinzip. Bevor ein Modul analysiert werden kann, müssen die importierten Module, vorher analysiert werden. Für diese müssen wiederum auch deren importierten Module vorher analysiert werden.

Wenn Analyseergebnisse bereits im Cache von CASS liegen, so werden diese im Normalfall nicht noch einmal analysiert. Stattdessen werden die Ergebnisse eingelesen und für die weiteren Analysen verwendet. Wenn man CASS irgendwie diese Ergebnisse übergeben kann, wenn sie im Cache noch fehlen, dann können überflüssige Analysen vermieden werden.

Dies kann mit CurryInfo erfolgen. Wenn CASS ein Modul analysieren soll, dann fragt CASS zunächst CurryInfo an, ob Ergebnisse bereits vorliegen. Wenn diese im Cache von CurryInfo vorliegen, so kann CASS die restliche Analyse überspringen und ist fertig.

Dieselbe Operation, die das angefragte Modul analysiert, analysiert auch die importierten Module. Daher werden für diese, wenn Ergebnisse vom zu analysierenden Modul nicht bei CurryInfo vorliegt, auch Anfragen an CurryInfo gestellt. Solange für eines dieser Module die Ergebnisse im Cache von CurryInfo liegen, wird die entsprechende Analyse nicht unnötigerweise wiederholt.

Um eine Endlosschleife zu vermeiden muss CurryInfo die Anfrage mit der Force Stufe 0 abarbeiten. Wenn die Force Stufe 1 oder sogar 2 wäre, dann würde bei fehlendem Analyseergebnis wieder CASS aufgerufen um die Analyse durchzuführen. CASS würde aber wieder CurryInfo anfragen. Dies wird durch die folgende Grafik verdeutlicht.



Eine mögliche Verbesserung wäre eine Option zu hinzuzufügen, mit welcher die Anfrage an CurryInfo verhindert wird. Dann könnte CurryInfo bei fehlen-

der Information CASS mit dieser Option aufrufen und so die Endlosschleife vermeiden.

7 Fazit

In dieser Arbeit wurde eine neue Anwendung CurryInfo entwickelt, welche die Informationen von anderen Anwendungen für die Sprache Curry zusammensammeln kann und über ein simples Interface zugänglich macht. Man kann dies wie ein normales Kommandozeilenprogramm verwenden, bei welchem man die gewünschte Entität und die Anfragen als Argumente übergibt, oder man kann es als Server laufen lassen, mit welchem man über einen Port kommunizieren kann um die Anfragen in Form von Nachrichten zu stellen.

Um zu zeigen, wie CurryInfo anderen Anwendungen helfen kann, wurde auch CASS als Beispiel verändert um CurryInfo zu benutzen. Mit dieser Änderung konnten überflüssige Aufrufe und Analysen vermieden werden.

Solche Änderungen können in Zukunft noch bei diversen weiteren Anwendungen umgesetzt werden. So könnte man existierende REPLs ändern, so dass man auch in diesen auf Informationen zugreifen kann, die von CurryInfo verwaltet werden. Auch IDEs beziehungsweise ein Language Server kann man um ähnliche Funktionen erweitern.

Selbstverständlich kann auch CurryInfo noch erweitert werden. Zum jetzigen Zeitpunkt ist es zum Beispiel nicht möglich Informationen zu Instanzen von Typklassen anzufragen. Die Implementierungen derer Methoden sind aber auch Operationen und können dahr auch zum Beispiel von CASS analysiert werden.

Auch weitere Informationen können noch hinzugefügt werden. Zum einen können weitere Anwendungen entwickelt werden, deren Informationen man auch mit anderen teilen will. Zum anderen existieren auch jetzt bereits weitere Informationen, die man in CurryInfo einbauen könnte. Zum Beispiel wurden nicht alle Analysen, die bei CASS zur Verfügung stehen, eingebaut.

Abseits von CurryInfo und Anwendungen, die Informationen für Curry Programme generieren können, gibt es noch andere Möglichkeiten. In dieser Arbeit wurde JSON Schema nur als Spezifikation verwendet.

Eine große Stärke von JSON Schema ist aber das Verifizieren von JSON Dateien. So eine Möglichkeit existiert aber noch nicht für Curry. Diese könnte entwickelt werden.

Es wäre auch möglich einen automatischen Code-Generator für JSON Schema zu entwickeln. Zum jetzigen Zeitpunkt müssen JSON Ausdrücke mit `JValue` repräsentiert und verarbeitet werden. JSON Schema gibt aber die Struktur einer JSON Datei vor. Mit diesem Schema könnte man ein Interface als Curry Code generieren um so einfacher auf die Felder zugreifen zu können. Ähnliches wird mit ER-Modellen in *ertools*[1] und Spicey[5] gemacht.

Literatur

[1] B. Brassel, M. Hanus und M. Müller. „High-Level Database Programming in Curry“. In: *Proc. of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*. Springer LNCS 4902, 2008, S. 316–332.

[2] M. Hanus. „Functional Logic Programming: From Theory to Curry“. In: *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 2013, S. 123–168.

[3] M. Hanus. „Inferring Non-Failure Conditions for Declarative Programs“. In: *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*. Springer LNCS 14659, 2024, S. 167–187. DOI: 10.1007/978-981-97-2300-3_10.

[4] M. Hanus. „Semantic Versioning Checking in a Declarative Package Manager“. In: *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*. Bd. 58. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 6:1–6:16. DOI: 10.4230/OASIcs.ICLP.2017.6.

[5] M. Hanus und S. Koschnicke. „An ER-based Framework for Declarative Web Programming“. In: *Theory and Practice of Logic Programming* 14.3 (2014), S. 269–291.

[6] M. Hanus, H. Kuchen und J.J. Moreno-Navarro. „Curry: A Truly Functional Logic Language“. In: *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*. 1995, S. 95–107.

[7] M. Hanus und J. Oberschweiber. „CPM: A Declarative Package Manager with Semantic Versioning“. In: *Pre-Proceedings of the Conference on Declarative Programming (Declare 2017)*. Technical Report 499, University of WÃ¼rzburg, 2017, S. 231–241.

[8] M. Hanus und F. Skrlac. „A Modular and Generic Analysis Server System for Functional Logic Programs“. In: *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*. ACM Press, 2014, S. 181–188.

A Bedienungsanleitung

CurryInfo ist eine Anwendung, mit welcher man Informationen über Bestandteile der Curry Sprache anfragen kann. Um diese Informationen zu generieren kann es für manche Anfragen nötig sein andere Anwendungen zu verwenden. CurryInfo übernimmt diese Aufgabe und ruft diese selbst automatisch auf, wenn es nötig ist. Es ist eine Kommandozeilenanwendung und hat den folgenden Aufruf.

```
curry-info [options] <requests>
```

CurryInfo funktioniert ähnlich wie ein Cache, welche Informationen abspeichert und Zugriff auf diese ermöglicht. Mittels Anfragen kann man so auf diese Informationen zugreifen.

Es gibt zahlreiche Optionen, die hier kurz beschrieben werden. Die erste Option gibt einen Hilfetext aus, welcher eine ähnliche Erklärung für die Anwendung enthält. Außerdem werden für die verschiedenen Bestandteile von Curry, die von nun an als Entitäten bezeichnet werden, die Liste der möglichen Anfragen ausgegeben. Diese haben ebenfalls eine kurze Erklärung für ihre Bedeutung. Die Option kann drei verschiedene Formen annehmen.

- `-h`
- `-?`
- `--help`

Die nächste Option stellt die Verbosität der Anwendung ein. Während der Abarbeitung von Anfragen kann die Anwendung verschiedene Nachrichten über den Ablauf ausgeben. Welche Nachrichten genau ausgegeben werden, kann mit dieser Option eingestellt werden. Die Option sieht wie folgt aus.

- `-v<n>`
- `--verbosity=<n>`

Die verschiedenen Stufen sind inklusive und je höher die Stufe ist, desto mehr Nachrichten werden ausgegeben. Inklusive heißt hier, dass alle Nachrichten von niedrigeren Stufen ebenfalls ausgegeben werden. Die Stufen und ihre Bedeutungen sind wie folgt.

- 0: Keine Nachrichten, nur Ausgabe
- 1: Statusnachrichten (zum Beispiel welche Anfrage bearbeitet wird)
- 2: Nachrichten, welche Aktionen ausgeführt werden (zum Beispiel Aufrufe anderer Anwendungen)
- 3: Alle Details über den Ablauf (ausgeführte Kommandos, Dateipfade, Zwischenergebnisse)

Man kann auch einstellen, wie die Anwendung mit fehlenden und vorhandenen Informationen umgehen soll. Dies wird über die Force Option eingestellt. Diese sieht wie folgt aus.

- `-f<n>`
- `--force=<n>`

Im Standardfall werden fehlende Informationen generiert und wenn sie vorhanden sind, dann werden sie einfach ausgegeben. Man kann dieses Verhalten aber auch ändern mit den folgenden Force Stufen.

- 0: Keine Generierung
- 1: Standardfall, Generierung wenn nötig
- 2: Generierung erzwungen, überschreibt vorhandene Informationen

Um Anfragen stellen zu können muss man eine Entität angeben. Dies passiert auch mittels Optionen, wobei man die gesamte Hierarchie der Entität angeben muss. Diese stellt man mit den folgenden Optionen ein.

- Paket: `-p <pkg>` oder `--package=<pkg>`
- Version: `-x <pkg>` oder `--version=<vsn>`
- Modul: `-m <pkg>` oder `--module=<mod>`
- Typ: `-t <pkg>` oder `--type=<type>`
- Typklasse: `-c <pkg>` oder `--typeclass=<typeclass>`
- Operation: `-o <pkg>` oder `--operation=<op>`

Typen, Typklassen und Operationen gehören zu einem Modul. Ein Modul wiederum gehört zu einer Version, die ebenfalls einem Paket zugeordnet sind. Entsprechend müssen diese für Anfragen genannt werden. Beispiele folgen. Die Reihenfolge der Optionen ist nicht relevant.

```
curry-info -p json -x 3.0.0 -m JSON.Parser
curry-info -p base -x 3.2.0 -m Prelude -o solve
curry-info -p directory
```

CurryInfo hat die Möglichkeit die Ergebnisse in verschiedenen Formaten auszugeben. Dies kann mit der Option `--output=<format>` eingestellt werden. Es existieren die folgenden Möglichkeiten.

- Text: Standardfall, von Menschen lesbar
- JSON: Ergebnis als JSON Ausdruck

- CurryTerm: Ergebnis als Liste von Curry Termen (`[(String, String)]`

Es gibt noch eine Reihe weiterer Optionen, die entweder nur eine Aufgabe erfüllen statt Anfragen zu bearbeiten oder die Abarbeitung von Anfragen verändert. Diese Optionen sind die folgenden.

- `--clean`: Löschen vom Cache. Durch Angabe einer Entität, werden nur die entsprechenden Daten gelöscht.
- `--showall`: Ausgabe aller vorhandenen Information der Entität.
- `--alltypes`: Anfrage werden für alle Typen eines Moduls bearbeitet. Modul muss als Entität angegeben werden.
- `--alltypeclasses` Das Gleiche mit Typklassen.
- `--alloperations` Das Gleiche mit Operationen.

B Hinzufügen neuer Anfragen

Ein Ziel von CurryInfo war es, das Hinzufügen von neuen Anfragen so einfach wie möglich zu gestalten. Dabei sollte es möglichst wenig Einschränkungen geben, welche Formen die Ergebnisse der Anfragen annehmen können. Wie das Hinzufügen funktioniert wird hier mit einem Beispiel Schritt für Schritt erklärt. Das Beispiel ist die Analyse *deterministic* von CASS, die als solche bereits implementiert ist.

Zunächst muss erklärt werden, für was Anfragen gestellt werden können. Die Bestandteile der Curry Sprache wurden in verschiedene Entitäten aufgeteilt, die in einer hierarchischen Beziehung zueinander stehen. Diese werden als Entitäten im weiteren bezeichnet. Es gibt die folgenden Entitäten.

- Package: Curry Pakete
- Version: Version von einem Paket
- Module: Modul einer bestimmten Version eines Pakets
- Type: Typ eines Moduls einer bestimmten Version
- Typeclass: Typklasse eines Moduls einer bestimmten Version
- Operation: Operation eines Moduls einer bestimmten Version

Die Hierarchie entspricht etwa der Reihenfolge dieser Liste. Also ein Modul gehört zu einer bestimmten Version und eine Version gehört zu einem bestimmen Paket. Typen, Typklassen und Operationen stehen natürlich in keiner hierarchischen Beziehung zueinander, aber sie können dem selben Modul zugeordnet sein.

Der Großteil der Änderung liegt im Modul `CurryInfo.Configuration`. In diesem sind Konfigurationen für alle Arten von Entitäten abgelegt, welche die implementierten Anfragen enthalten. Als ersten Schritt muss man also entscheiden, für welche Art von Entität die Anfrage implementiert werden soll. Für dieses Beispiel sind das Operationen. Um die neue Anfrage hinzuzufügen, muss sie als neuer Eintrag mit der Operation `registerRequest` eingetragen werden. Diese hat den folgenden Typ.

```
registerRequest :: ConvertJSON b => String -> String
-> Generator a b -> Printer b -> RegisteredRequest a
```

Zunächst werden einmal die Typvariablen erklärt. Die Typvariable `a` steht für den Typ der Entität. Für das Beispiel ist das also `CurryOperation`. Die Typvariable `b` ist der Typ der Information. Für das Beispiel soll dies einfach `String` sein. Der Typ würder für das Beispiel wie folgt aussehen.

```
registerRequest :: String -> String
-> Generator CurryOperation String -> Printer String
-> RegisteredRequest CurryOperation
```

Die einzige Einschränkung, die für die Typvariable `b` existiert, ist die Typklasse `ConvertJSON`. Es muss eine Instanz für diese Typklasse existieren.

Wenn die Information einen herkömmlichen Typ wie `Int`, `Bool` oder `String` hat, dann existieren solche Instanzen bereits. Auch für Typen wie Listen und `Maybe` existieren entsprechende Instanzen. Für selbstdefinierte Typen kann man eine entsprechende Instanz im Modul `CurryInfo.JConvert` implementieren. Instanzen in diesem Modul sind für die Konfigurationen verfügbar.

Die Operation benötigt also vier Argumente. Die ersten zwei Argumente sind der Name der Anfrage und eine Beschreibung. Der Name muss der Anfrage entsprechen, wie sie CurryInfo als Argument übergeben werden soll. Die Beschreibung wird im Hilfetext angezeigt und soll die Anfrage erklären.

Die letzten zwei Argumente sind Operationen. Das dritte Argument ist die Operation, welche die fehlende Information generiert. Der Typ sieht wie folgt aus.

```
type Generator a b = Options -> a -> IO (Maybe b)
```

Der Typ `Options` wird vor allem verwendet um Statusnachrichten während der Bearbeitung auszugeben. Das Wichtigste ist das Argument vom Typ `a` und das Ergebnis vom Typ `IO (Maybe b)`. Die Typvariablen stehen weiterhin für die gleichen Typen. Für das Beispiel wäre das `CurryOperation` für `a` und `String` für `b`.

Definiert werden diese Operationen im Module `CurryInfo.Generator`. Es existieren auch einige generische Operationen, die für ähnliche Fälle benutzt werden können. Für das Beispiel existiert bereits eine solche Operation, die eine Analyse mit CASS durchführt und das Ergebnis als Information verwendet. Das letzte Argument hat den folgenden Typ.

```
type Printer b = Options -> b -> IO String
```

Diese Operation ist dafür zuständig, die Information in Form eines `Strings` auszugeben. Im Modul `CurryInfo.Printer` sind die bereits implementierten Operationen zu finden. Das Ergebnis muss nicht zwangsläufig der `Show` Instanz entsprechen. Zum Beispiel für Code Ausschnitte gibt es den Typ `Reference`, welcher nicht den Ausschnitt selbst enthält. Aber als Ausgabe wäre dieser nützlicher. Dies ist auch ein Grund dafür, weshalb diese Operation im `IO` Kontext ist, da der Ausschnitt aus der Datei gelesen werden muss.

Wenn man die Operationen implementiert hat, so muss man nur noch die Anfrage registrieren. Für das Beispiel soll die generierende Operation den Namen `gOperationCASSDeterministic` und die ausgebende Operation den Namen `pOperationCASSDeterministic` haben. Der Eintrag in der Konfiguration sieht dann wie folgt aus.

```
registerRequest
    "cass-deterministic"
    "\t\tAnalysis result whether the operation is deterministic"
    gOperationCASSDeterministic
    pOperationCASSDeterministic
```

Zuletzt muss man CurryInfo mit diesen Änderungen neu kompilieren. Danach kann man mit der Hilfeoption überprüfen, dass die hinzugefügte Anfrage angezeigt wird. Sie ist dann auch als Anfrage implementiert und kann verwendet werden.

C JSON Schema Spezifikationen

C.1 Paket

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/package.schema.json",
    "title": "Package",
    "description": "A Curry package.",
    "type": "object",
    "properties": {
        "package": {
            "description": "The name of the package.",
            "type": "string"
        },
        "versions": {
            "description": "The versions of the package that exist.",
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    }
}
```

C.2 Version

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/version.schema.json",
    "title": "Version",
    "description": "The version of a package.",
    "type": "object",
    "properties": {
        "version": {
            "description": "The version number.",
            "type": "string"
        },
        "documentation": {
            "description": "The description of the package of this version.",
            "type": "string"
        },
        "categories": {
            "description": "The categories this version of the package belongs to.",
            "type": "array",
            "items": {
```

```
                "type": "string"
            }
        },
        "modules": {
            "description": "The modules this version contains.",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "dependencies": {
            "description": "The dependencies of this version.",
            "type": "object",
            "properties": {
                "package": {
                    "description": "The package of the dependency.",
                    "type": "string"
                },
                "constraints": {
                    "description": "The constraints of the dependency.",
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                }
            }
        }
    }
}
```

C.3 Modul

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/module.schema.json",
    "title": "Module",
    "description": "A module of a package.",
    "type": "object",
    "properties": {
        "module": {
            "description": "The name of the module.",
            "type": "string"
        },
        "documentation": {
            "description": "The documentation of the module.",
            "type": "object",
```

```
        "properties": {
            "path": {
                "description": "The path of the source file.",
                "type": "string"
            },
            "start": {
                "description": "Index of first line belonging to the sclice.",
                "type": "integer"
            },
            "end": {
                "description": "Index of first line not belonging to the sclice.",
                "type": "integer"
            }
        }
    },
    "sourceCode": {
        "description": "The source code of the module.",
        "type": "object",
        "properties": {
            "path": {
                "description": "The path of the source file.",
                "type": "string"
            },
            "start": {
                "description": "Index of first line belonging to the sclice.",
                "type": "integer"
            },
            "end": {
                "description": "Index of first line not belonging to the sclice.",
                "type": "integer"
            }
        }
    },
    "cass-unsafemodule": {
        "description": "Whether the module is safe or unsafe,
            possibly with a list of modules which results in it being unsafe.",
        "type": "string"
    },
    "typeclasses": {
        "description": "The typeclasses defined in the module.",
        "type": "array",
        "items": {
            "type": "string"
        }
    },
    "types": {
```

```
            "description": "The types defined in the module.",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "operations": {
            "description": "The operations defined in the module.",
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    }
}
```

C.4 Typ

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/type.schema.json",
    "title": "Type",
    "description": "A type defined in a module.",
    "type": "object",
    "properties": {
        "typeName": {
            "description": "The name of the type.",
            "type": "string"
        },
        "documentation": {
            "description": "The documentation of the type.",
            "type": "object",
            "properties": {
                "path": {
                    "description": "The path of the source file.",
                    "type": "string"
                },
                "start": {
                    "description": "Index of first line belonging to the sclice.",
                    "type": "integer"
                },
                "end": {
                    "description": "Index of first line not belonging to the sclice.",
                    "type": "integer"
                }
            }
```

```
        },
        "definition": {
            "description": "The definition of the type.",
            "type": "object",
            "properties": {
                "path": {
                    "description": "The path of the source file.",
                    "type": "string"
                },
                "start": {
                    "description": "Index of first line belonging to the sclice.",
                    "type": "integer"
                },
                "end": {
                    "description": "Index of first line not belonging to the sclice.",
                    "type": "integer"
                }
            }
        },
        "constructors": {
            "description": "The constructors of the type",
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    }
}
```

C.5 Typklasse

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/typeclass.schema.json",
    "title": "Typeclass",
    "description": "A typeclass defined in a module.",
    "type": "object",
    "properties": {
        "typeclass": {
            "description": "The name of the typeclass.",
            "type": "string"
        },
        "documentation": {
            "description": "The documentation of the typeclass.",
            "type": "object",
            "properties": {
```

```
                "path": {
                    "description": "The path of the source file.",
                    "type": "string"
                },
                "start": {
                    "description": "Index of first line belonging to the sclice.",
                    "type": "integer"
                },
                "end": {
                    "description": "Index of first line not belonging to the sclice.",
                    "type": "integer"
                }
            }
        },
        "definition": {
            "description": "The definition of the typeclass.",
            "type": "object",
            "properties": {
                "path": {
                    "description": "The path of the source file.",
                    "type": "string"
                },
                "start": {
                    "description": "Index of first line belonging to the sclice.",
                    "type": "integer"
                },
                "end": {
                    "description": "Index of first line not belonging to the sclice.",
                    "type": "integer"
                }
            }
        },
        "methods": {
            "description": "The methods of the typeclass.",
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    }
}
```

C.6 Operation

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
```

```
"$id": "https://example.com/operation.schema.json",
"title": "Operation",
"description": "An operation defined in a module.",
"type": "object",
"properties": {
    "operation": {
        "description": "The name of the operation.",
        "type": "string"
    },
    "documentation": {
        "description": "The documentation of the operation.",
        "type": "object",
        "properties": {
            "path": {
                "description": "The path of the source file.",
                "type": "string"
            },
            "start": {
                "description": "Index of first line belonging to the sclice.",
                "type": "integer"
            },
            "end": {
                "description": "Index of first line not belonging to the sclice.",
                "type": "integer"
            }
        }
    },
    "definition": {
        "description": "The source code of the typeclass.",
        "type": "object",
        "properties": {
            "path": {
                "description": "The path of the source file.",
                "type": "string"
            },
            "start": {
                "description": "Index of first line belonging to the sclice.",
                "type": "integer"
            },
            "end": {
                "description": "Index of first line not belonging to the sclice.",
                "type": "integer"
            }
        }
    },
    "signature": {
```

```
            "description": "The signature of the operation.",
            "type": "string"
        },
        "infix": {
            "description": "What kind of infix the operation is when used infix.",
            "type": [null, "string"]
        },
        "precedence": {
            "description": "The precedence of the operation when used infix",
            "type": [null, "string"],
            "minimum": 0,
            "maximum": 9
        },
        "cass-deterministic": {
            "description": "Whether the operation is deterministic or non-deterministic.",
            "type": "string"
        },
        "cass-demand": {
            "description": "List of argument indeces
                that are required to compute a result.",
            "type": "string"
        },
        "cass-indeterministic": {
            "description": "Whether the operation is indeterministic.",
            "type": "string"
        },
        "cass-solcomplete": {
            "description": "Whether the operation is solution complete.",
            "type": "string"
        },
        "cass-terminating": {
            "description": "Whether the operation is sure to always terminate.",
            "type": "string"
        },
        "cass-total": {
            "description": "Whether the operation is totally defined.",
            "type": "string"
        },
        "failfree": {
            "description": "Verification result about
                the failing behavior of the operation.",
            "type": "string"
        }
    }
}
```