Integration von Curry-Programmen in Webseiten durch Übersetzung nach JavaScript

---

Jasper Paul Sikorra

25. August 2017

Christian-Albrechts-Universität zu Kiel

Technische Fakultät
Institut für Informatik
Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion

Masterarbeit

Integration von Curry-Programmen in Webseiten durch Übersetzung nach JavaScript

Jasper Paul Sikorra

Betreut durch:
Prof. Dr. Michael Hanus und Dipl.-Inf. Jan Tikovsky

25. August 2017

Jasper Paul Sikorra

Integration von Curry-Programmen in Webseiten durch Übersetzung nach JavaScript

Masterarbeit, 25. August 2017

Betreut durch: Prof. Dr. Michael Hanus und Dipl.-Inf. Jan Tikovsky

Christian-Albrechts-Universität zu Kiel

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion

Institut für Informatik

Technische Fakultät

D-24098

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegen hat.

Kiel, 25. August 2017

Jasper Paul Sikorra

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

JavaScript ist die Sprache des Web. Mit der Verbreitung des Internets ist JavaScript zu einer der am meisten verwendeten Programmiersprachen aufgestiegen.
In den letzten Jahren gibt es immer mehr Programmiersprachen, die nach JavaScript übersetzt werden können.[1] Zum Beispiel gibt es für die funktionale Programmiersprache Haskell viele verschiedene Ansätze, um eine Ausführung in JavaScript zu ermöglichen.[2]

Warum machen sich so viele Entwickler von Programmiersprachen die Mühe, nach JavaScript zu kompilieren? Zum einen hat JavaScript eine sehr hohe Verbreitung, zum anderen sind viele Entwickler davon überzeugt, dass eine andere Programmiersprache in bestimmten Anwendungsfällen besser geeignet ist als JavaScript.
Ein Grund hierfür ist, dass JavaScript nicht für die Anwendungsfälle entwickelt wurde, für die es heute eingesetzt wird [Sev12]. Ein weiterer Grund ist das höhere Abstraktionslevel, welches durch nicht-imperative Programmiersprachen erreicht werden kann.

Curry ist eine Programmiersprache aus der Familie der deklarativen Programmiersprachen, welche eine solche höherer Abstraktionsebene durch die Verschmelzung von funktionaler und logischer Programmierung erreicht. Während imperative Programme aus Sequenzen von Anweisungen bestehen und damit nur eine höhere Abstraktion in den Strukturen des von Neumann Computers erreichen [Bac78], bestehen Curry-Programme aus Definition von Datentypen und Funktionen auf diesen Datentypen.
Ein auszuwertender Ausdruck in Curry ist ein aus Datenkonstruktoren und Funktionsapplikationen zusammengesetzter Term, welcher durch Termersetzung im Sinne der Funktionsregeln in eine Grundform gebracht wird. Durch die Einbeziehung des logischen Paradigmas erlaubt Curry in diesen Funktionsregeln die Verwendung von nicht-deterministischen Ausdrücken und freien Variablen. Hierdurch lassen sich be-

[1] Eine Liste findet sich zum Beispiel unter `https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js`.
[2] Eine Liste von Ansätzen um Haskell nach JavaScript zu übersetzten findet sich unter `https://wiki.haskell.org/The_JavaScript_Problem`.

stimmte Probleme präziser oder lesbarer ausdrücken als in einer imperativen Sprache.

Im Jahr 2016 wurde ein neuer Compiler für Curry namens Sprite vorgestellt, welcher in eine imperative Low-Level-Programmiersprache übersetzt. Dieser Compiler basiert auf dem Fair-Scheme, bei dem die Darstellung von Currytermen als Graph und die Auswertung mit Hilfe eines Graphersetzungssystems und Pull-Tabbing durchgeführt wird [AJ14].

Für die Übersetzung von Curry nach JavaScript können mehrere Ansätze in Betracht gezogen werden. Für diese Arbeit wurde eine Implementierung nach ähnlichen Konzepten wie Sprite ausgewählt.

1.2 Resultat

Die in dieser Arbeit besprochene Implementierung setzt auf einer Zwischensprache namens ICurry auf, deren Programme durch Transformation von Curry-Programmen erzeugt werden können. Das Ziel der Implementierung war ein möglichst korrekten Übersetzer und eine korrekte Runtime zu erstellen. Vollständigkeit und Effektivität der Implementierung waren untergeordnete Ziele.

Um eine möglichst korrekte Übersetzung zu gewährleisten, wurde zunächst eine Semantik für ICurry spezifiziert. Diese Semantik umfasst den funktionalen und logischen Teil von Curry mit der Ausnahmen von Constraints und Unifikation, sowie IO-Aktionen. Da in ICurry freie Variablen durch Generatorfunktionen ersetzt werden, werden auch keine Regeln für freie Variablen aufgestellt.

Die Semantik wurde dann in Form einer Runtime implementiert. Schließlich wurde ein Übersetzer für ICurry-Programme nach JavaScript erstellt, welcher zu der Runtime kompatible Programme erzeugt.
Der Übersetzer und die Runtime sind funktionsfähig und können ausprobiert werden.[3] Dabei findet der Übersetzungsvorgang auf dem Server statt und die Ausführung der Auswertung des Curry-Terms wird ausschließlich auf dem Client ausgeführt.

Die Implementierung zeigt, dass auch die Konstruktion eines Übersetzers von Curry nach JavaScript nach den Konzepten von Sprite ein gangbarer Weg ist. Gleichzeitig hat sich gezeigt, dass Stärken und Schwächen in der Performance der Zielsprachen außerordentlich wichtig für die Effektivität der Implementierung sind.

[3] Zum Beispiel unter `curry.sikorra.info` (abgerufen am 17. August 2017).

1.3 Struktur

In den folgenden Abschnitten werden zunächst JavaScript (Abschnitt 2.1) und Curry (Abschnitt 2.2) vorgestellt. Es werden dann die verschiedenen Ansätze für die Übersetzung von Curry nach JavaScript diskutiert (Kapitel 3). In Abschnitt 3.5 wird der gewählte Ansatz vorgestellt. Es wird dann die Implementierung besprochen (Kapitel 4), wobei die für ICurry entwickelte Semantik (Abschnitt 4.1) und die Architektur (Abschnitt 4.2) und ihre einzelnen Komponenten (Abschnitt 4.3) vorgestellt werden. Die Umsetzung wird in Kapitel 5 in Bezug auf ihre Vollständigkeit der Curry-Implementierung und ihre Performance evaluiert. In Kapitel 6 werden die Resultate zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

2.1 JavaScript

JavaScript ist eine Programmiersprache und bildet zusammen mit HTML und CSS den Kern der Technologien für die Darstellung von Webseiten im World Wide Web (*WWW*). JavaScript hat mit einer Einbettung in 94.6% der 10 Millionen meistbesuchten Webseiten[1] eine solche Verbreitung, dass ein populärer Internet Browser im Jahr 2013 sogar die Option für die Abschaltung der Programmiersprache entfernte.[2]

JavaScript wird im WWW auf Client-Seite zur Manipulation des Document Object Model (*DOM*), dass heißt der Oberflächenrepräsentation des Browsers, eingesetzt. Weitere typische Anwendungsfälle sind das asynchrone Abfragen von Daten nach dem Laden der Webseite (Ajax), die Validierung von Nutzereingaben, sowie die Analyse von Nutzerverhalten.

Mit der Veröffentlichung von Node.js ist es möglich JavaScript auch außerhalb von Internet-Browsern auszuführen. Heute wird JavaScript zur serverseitigen Programmierung, für die Erstellung von mobilen Anwendungen, in Datenbanken und in vielen weiteren Bereichen eingesetzt.

Node.js eröffnete auch den Weg für ein Paketmanagement-Tool namens *npm* für JavaScript-Pakete. Dies begünstigte die Verbreitung der Sprache. Die Paketsammlung, auf welche npm zugreift, ist inzwischen größer als die des Apache Maven Repository und ist mit über 400.000 Paketen die größte der Welt.[3]

2.1.1 Eigenschaften von JavaScript

Es gibt verschiedene Implementierungen von JavaScript (*Engines*), die alle die Sprachspezifikation der *ECMA International*-Organisation umsetzen. Bekannte Engines sind *V8* aus dem Chrome Browser, welche auch in Node.js verwendet wird,

[1] Siehe dazu auch `https://w3techs.com/technologies/details/cp-javascript/all/all`.
[2] Siehe dazu auch die Mozilla Firefox Release Notes unter `https://www.mozilla.org/en-US/firefox/23.0/releasenotes/`.
[3] Siehe dazu den Artikel unter `https://www.linux.com/news/event/Nodejs/2016/state-union-npm`.

SpiderMonkey, welche in Mozilla Firefox eingesetzt wird, sowie *JavaScriptCore* aus Apple Safari und *Chakra* aus Microsoft Edge. In dieser Arbeit wird, wenn nicht anders gekennzeichnet, von JavaScript nach der *ECMA-262* Spezifikation ausgegangen.[4] Dies entspricht *ECMAScript* in der Version 5.1, welches von allen oben genannten JavaScript-Engines vollständig implementiert ist.[5]

JavaScript ist eine objektorientierte Skriptsprache, die in einem Interpreter ausgeführt wird. Wie viele Skriptsprachen ist JavaScript dynamisch getypt und erlaubt implizite Typumwandlung. Es sind keine Typdeklarationen im JavaScript Quelltext vorgesehen. JavaScript folgt außerdem dem Prinzip des Duck-Typing, dass heißt ein Objekt wird nicht durch seine Zugehörigkeit zu einer Klasse, sondern durch das Vorhandensein von Attributen und Methoden klassifiziert. JavaScript unterstützt Funktionen höherer Ordnung, da Funktionen wie Objekte behandelt werden.

Syntax

Die Syntax von JavaScript ähnelt der von *Java*, verzichtet jedoch vollständig auf Typangaben und Klassen. In Beispiel 2.1 ist eine Funktion für das Aufsummieren

```
function sumArray(array) {
  var sum = 0;
  for (var i = 0; i < array.length; i++) {
    sum = sum + array[i];
  }
  return sum;
}
```

Bsp. 2.1.: Funktion zum Aufsummieren eines Arrays in JavaScript

eines Arrays dargestellt, in welcher typische Konzepte imperativer Sprachen, wie die Mutation von Variablen und Schleifen eingesetzt werden. Weiterhin ist der Zugriff auf eine Objekteigenschaft in der Schleifenbedingungen zu sehen.

2.1.2 Nachteile von JavaScript

Aus den Eigenschaften von JavaScript folgen einige Vor- und Nachteile. JavaScript lässt dem Benutzer viel Freiheit in der Gestaltung seiner Programme und durch Duck-Typing lasst sich ein hoher Grad an Komposition verwirklichen. Es gibt aber auch Schwachstellen in JavaScript, die vor allem in größeren Projekten zu Problemen führen können.

[4] Die ECMA-262 Spezifikation ist unter `http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf` abrufbar.
[5] Eine Übersicht findet sich unter `https://kangax.github.io/compat-table/es5`.

Scoping

Ein *Scope* ist der Geltungsbereich einer Deklaration. In JavaScript gelten Variablen im globalen Scope, es sei denn, sie werden explizit in einem lokalen Scope deklariert.

```
( function () { x = 1; })();
( function () { console.log(x); })();
```

Bsp. 2.2.: Globales Scoping einer Variable in einer Funktionsdefinition in JavaScript

In Beispiel 2.2 wird eine Variable `x` in einer Funktion definiert. Diese Variable ist nun im globalen Scope gültig, dass heißt, wird `x` in einer anderen Funktion verwendet, so ist `x` möglicherweise schon mit einem Wert belegt. Bei der Ausführung des Beispiels ist die Ausgabe auf der Konsole `1`. Um dies zu vermeiden, muss `x` explizit mit dem Schlüsselwort `var` deklariert werden.

In JavaScript werden lokale Scopes weiterhin nur durch Funktionsdefinitionen geöffnet. In Beispiel 2.3 wird eine Variable in einem Block deklariert. Anders als in

```
do { var y = 1; } while (false);
console.log(y);
```

Bsp. 2.3.: Globales Scoping einer deklarierten Variable in einem Block

den meisten Programmiersprachen aus der C-Familie ist die Variable außerhalb des Blocks gültig. Bei Ausführung des Beispiels wird `1` auf der Konsole ausgegeben.

Die ungewollte Definition der Variablen im globalen Scope ist problematisch, da in anderen Programmteilen möglicherweise unvorhergesehene Effekte auftreten können, die schwer nachvollziehbar sind [Ekb12, S. 5].

Typumwandlung

Wird eine Operation auf Werte angewendet, die nicht mit den in der Operation verlangten Typen zusammenpassen, so versucht JavaScript die Werte so umzuwandeln, dass die Operation durchgeführt werden kann. Dies hat Konsequenzen, die nicht immer wünschenswert sind. Zum Beispiel führt dies dazu, dass $=$ nicht transitiv ist, da

```
!(0 == "0" && 0 == "") || "0" == ""
```

zu `false` ausgewertet wird. Es lassen sich auch Ausdrücke wie

```
(function (x) { return x+1; }) > ['JavaScript']
```

auswerten, wobei der Vergleich einer Funktion mit einem Array in diesem Fall `true` ergibt. Genauso unerwartet ist das Ergebnis des Ausdrucks

```
[0,4] + [2,1]
```

dessen Auswertung zum dem String `"0,42,1"` führt.

Diese Typumwandlung kann vor allem dann unerwarteten Ergebnissen führen, wenn einer Funktion Parameter mit dem falschen Typ übergeben werden. In Bei-

```
function calculation(n) {
  return (1 + n) * 100
}

console.log(calculation('5'))
```

Bsp. 2.4.: Implizite Typumwandlung durch arithmetische Operationen

spiel 2.4 wird die Übergabe des Strings `'5'` an die Funktion zu dem Ergebnis `1500` führen, da die Typumwandlung von `+` und `*` unterschiedlich funktioniert. Dies ist in vielen Fällen kein wünschenswertes Ergebnis, gleichzeitig ist das Vorkommen eines solchen Fehler schwer zu erkennen.

Keine Elimination von Endrekursion und keine unveränderbaren Variablen

Obwohl JavaScript anonyme Funktionen und Funktionen höherer Ordnung unterstützt, ist die Anwendungsmöglichkeit funktionaler Konzepte in der Sprache stark eingeschränkt. JavaScript führt keine automatische Transformation von rekursiven Funktionsaufrufen durch und kann dies auch nicht, da die gesamte Kette der Funktionsaufrufe mit allen Argumenten zu jedem Zeitpunkt zugänglich sein muss [Ekb12, S. 6].
Es gibt auch keine Möglichkeit Unveränderlichkeit zu erzwingen, dass heißt Variablen können jederzeit neu belegt werden.

Keine Module

Eine Einteilung eines Programms in Module ist in JavaScript nicht vorgesehen. Dies führt logischerweise zu großen Problemen, da eine Unterteilung von größeren JavaScript-Programmen in kleinere Teile nicht mittels wiederverwendbaren Modulen geschehen kann. Es haben sich verschiedene Systeme zur Modularisierung entwickelt[6], welche teilweise untereinander inkompatible sind und externe Programme zur Kompilierung benötigen.

[6] Ansätze zur Modularisierung in JavaScript sind z. B. *AMD*, *commonjs* und *UMD*.

Aufgrund dieser Nachteile kann es sinnvoll sein, eine andere Programmiersprache zu verwenden und diese nach JavaScript zu übersetzen.

2.1.3 Transpilation

Für JavaScript sind verschiedene Versionen spezifiziert. In dieser Arbeit wird, wie im letzten Abschnitt beschrieben, immer von der Version 5.1 ausgegangen. Es gibt auch neuere Versionen. So werden in der als ECMAScript 6 spezifizierten Version [7] unter anderem unveränderbare Variablen und Variablendeklarationen für lokale Scopes eingeführt. In ECMAScript 6 ist außerdem vorgesehen, dass rekursive Funktionsaufrufe optimiert werden. Dies soll erreicht werden, indem erlaubt wird, Stack Frames wiederzuverwenden. Dieses Feature wurde zum Zeitpunkt dieser Arbeit in keinem großen Browser außer Safari implementiert.

Es gibt noch weitere Versionen von ECMAScript mit neuen Features, welche jedoch noch nicht vollständig spezifiziert wurden. Solche Features können mit Hilfe der Transpilation in eine ältere ECMAScript Version übersetzt werden. Es gibt zum Beispiel ein Proposal für ECMAScript, welches vorsieht die Schlüsselwörter `async` und `await` einzuführen. Diese Schlüsselwörter erlauben ein übersichtlicheres Programmieren mit asynchronen Funktionen. Mit Hilfe des Transpilers *babel* kann dieses Feature in ein äquivalentes ECMAScript 6 oder ECMAScript 5.1 Konstrukt übersetzt werden.

2.1.4 JavaScript als Zielsprache

JavaScript hat als Zielsprache eines Übersetzers Vor- und Nachtteile. Da JavaScript eine High-Level-Programmiersprache ist und dabei Konstrukte mit hoher Abstraktion, wie Funktionen höherer Ordnung, und objektorientierte Programmierung unterstützt, können übersetzte Programme komfortabel ausgedrückt werden.
Diese abstrakten Konstrukte von JavaScript führt allerdings auch dazu, dass eine Optimierung bei der Übersetzung im Sinne der Verwendung von effektiven, speicher- oder prozessornahen Anweisungen nicht oder nur sehr schwer möglich ist. Das heißt, Konstrukte aus der Quellsprache werden immer in „große“ Konstrukte der Zielsprache übersetzt.

JavaScript setzt auf einen Garbage Collector als Speicherverwaltung. Damit muss die Speicherverwaltung bei der Übersetzung nicht explizit generiert werden. Da die Spei-

[7] Die Spezifikation ist unter `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf` zu finden.

cherverwaltung damit automatisiert ist, ist eine detaillierte Optimierung der Speicherallokation und -freigabe allerdings auch nicht möglich.

Die Syntax von JavaScript ist im Vergleich zu Java schlanker, da es keine Klassen, keine Zugriffsmodifikatoren und keine Schlüsselwörter für die Markierung statischer Methoden und Attribute gibt.

Das Tooling um JavaScript ist ausgereift. Es gibt verschiedene Debugger und Werkzeuge zur Analyse und Visualisierung der Performance von Programmen.
Es gibt auch eine Spezifikation für die Struktur eines abstrakten Syntaxbaumes von JavaScript-Programmen und Werkzeuge, welche auf diesen Bäumen arbeiten, um zum Beispiel JavaScript-Code zu generieren oder Programmtransformationen durchzuführen.
Da JavaScript-Quelltext häufig über ein Netzwerk vom Server zum Client geschickt wird, gibt es verschiedene Compiler, welche die Größe des Quelltexts reduzieren.

2.2 Curry

Curry ist eine General-Purpose-Programmiersprache, welche Paradigmen der funktionalen und der logischen Programmierung vereint und viele der Schwachstellen in JavaScript umgeht.

In der funktionalen Programmierung bestehen Programme aus Datentypen und Funktionen auf diesen Datentypen, wobei Funktionen die Eigenschaft besitzen müssen, bei gleicher Eingabe die gleiche Ausgabe zurückzugeben. Weiterhin können Funktionen durch Verkettung verknüpft und an andere Funktionen als Parameter übergeben werden.

In der logischen Programmierung bestehen Programme aus Axiomen und Ableitungsregeln, wobei ein Programm versucht zu einer Abfrage eine Lösungsaussage zu berechnen. Dabei erlaubt logische Programmierung auch den Einsatz ungebundener Variablen, für die eine passende Belegung gesucht wird.

Curry ist statisch getypt und erlaubt keine implizite Typumwandlung.

2.2.1 Syntax und Semantik von Curry

Die Syntax und Semantik von Curry ist im Curry Report ([He16]) definiert. Curry orientiert aus syntaktischer Sicht stark an der funktionalen Programmiersprache *Haskell*. Ein Curry-Programm besteht aus Datentyp-Deklarationen und Funktionsde-

finitionen, wobei in den Funktionsdefinitionen freie Variablen vorkommen können. Auch semantisch gleicht der funktionale Teil von Curry Haskell. So werden Teilausdrücke nur dann ausgewertet, wenn sie benötigt werden.

Datentypen

```
data Peano = Z | S Peano
```

Bsp. 2.5.: Die Peano-Zahlen

In Beispiel 2.5 werden die Peano-Zahlen als Curry-Datentyp definiert. Der Datentyp hat zwei Konstruktoren `Z` und `S`. Mit Hilfe dieser Konstruktoren können Terme zusammengesetzt werden, so zum Beispiel die dritte Peano-Zahl `S (S Z)`. Alle Ausdrücke in Curry sind stark und statisch getypte. Der Ausdruck `S (S Z)` hat somit den Typ `Peano`. In Curry sind parametrische Typpolymorphismen erlaubt. So können zum Beispiel polymorphe Bäume mit Hilfe eines Typparameters, wie in Beispiel 2.6 gezeigt, implementiert werden.
Durch Kombination kann dadurch ein Baum, welcher Peano-Zahlen enthält, gebil-

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

Bsp. 2.6.: Datentyp polymorpher Bäume

det werden. Ein mögliches Mitglied dieses Typs wäre zum Beispiel

```
Branch (Leaf Z) (Leaf (S Z))
```

Funktionen

Auf den Datentypen können Funktionen definiert werden. In Beispiel 2.7 wird ei-

```
add :: Peano → Peano → Peano
add Z     b = b
add (S a) b = S (add a b)
```

Bsp. 2.7.: Addition auf den Peano-Zahlen

ne Funktion definiert, welche zwei Peano-Zahlen zusammenzählt. Diese Funktion hat zwei Regeln, welche, in Abhängigkeit vom ersten Parameter, in unterschiedlichen Fällen angewendet werden. Diese Unterscheidung wird als Pattern-Matching bezeichnet, da der Konstruktor des Parameters mit einem Pattern verglichen wird. Mit einem Curry-Interpreter kann ein aus den Konstruktoren `Z` und `S` und der Funktion `add` zusammengesetzter Ausdruck ausgewertet werden. Zum Beispiel wird der Ausdruck `add (S Z) (S Z)` zu `S (S Z)` ausgewertet.

In Beispiel 2.8 wird eine Funktion mit Typparametern definiert und dann angewendet, um eine ähnliche Funktion wie in Beispiel 2.1 in Curry zu definieren. In diesem

```
sumArray :: [Int] → Int
sumArray = foldr (+) 0

foldr :: (a → b → b) → b → [a] → b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Bsp. 2.8.: Aufsummieren einer Liste in Curry

Beispiel ist auch zu sehen, dass rekursive Funktionsaufrufe in Curry, anders als in JavaScript, bedenkenlos benutzt werden können. Außerdem ist ersichtlich, dass auch Funktionen mit Typparametern definiert werden können. Die Funktion `foldr` hat einen Typen mit zwei Typparametern. Dies ist möglich, da die Funktion kein Wissen über den Inhalt dieser Parameter besitzen muss.

Lokale Deklarationen

Um übersichtlichere Programme zu ermöglichen und Hilfsfunktionen zu kapseln, sind in Curry lokale Deklarationen möglich. In Beispiel 2.9 wird `let` verwendet, um eine Funktion zu definieren, welche nur in einem lokalen Scope gültig ist.

```
sumArray :: [Int] → Int
sumArray =
  let foldr _ z []     = z
      foldr f z (x:xs) = f x (foldr f z xs)
  in  foldr (+) 0
```

Bsp. 2.9.: Aufsummieren einer Liste in Curry mit `let`

In Beispiel 2.10 wird die Verdopplung von Fibonacci-Zahlen mit Hilfe einer Hilfsfunktion definiert. In lokalen Deklarationen sind dabei sämtliche Variablen verfügbar, welche in der umfassenden Definition eingeführt werden. In diesem Beispiel wird `result` lokal deklariert und muss dadurch nur einmal berechnet werden.

```
doubleFib :: Int → Int
doubleFib a = result + result
  where
    result = fib a

fib n = case n of
  0 → 0
  1 → 1
  n →  fib (n-1) + fib (n-2)
```

Bsp. 2.10.: Definition einer Funktion mit lokaler Deklaration

Nicht-Determinismus und freie Variablen

Im Beispiel 2.10 wurde die Funktion `fib` mit einem case-Ausdruck definiert. Würde statt der Fallunterscheidung Pattern-Matching benutzt, wie in Beispiel 2.11 dargestellt, so würde die Funktion nicht-deterministisch ausgewertet werden und es gäbe Auswertungspfade, welche nicht terminieren. In der letzten Funktionsregel über-

```
ndFib 0 = 0
ndFib 1 = 1
ndFib n = fib (n-1) + fib (n-2)
```

Bsp. 2.11.: Definition der Funktion fib mit überlappenden Pattern

lappt das Pattern des ersten Parameters mit den ersten beiden Funktionsregeln. In Curry bedeutet dies, dass die anzuwendene Regel nicht-deterministisch ausgewählt wird. Im Gegensatz dazu wird bei überlappenden Pattern in case-Ausdrücken eine determinstische Auswertung beibehalten und das erste passende Pattern ausgewählt. Deshalb ist `fib` aus Beispiel 2.10 deterministisch und `ndFib` nicht-deterministisch. In Beispiel 2.12 wird die nicht-deterministische Funktion `coin` definiert. Wird diese

```
coin = Z
coin = S Z
```

Bsp. 2.12.: Definition der nicht-deterministischen Funktion `coin`

Funktion aufgerufen, so werden die beiden Ergebnisse `Z` und `S Z` berechnet. In Bei-

```
double :: Peano → Peano
double n = add n n
```

Bsp. 2.13.: Definition der Funktion `double`

spiel 2.13 wird die zusätzliche Funktion `double` definiert, welche eine Peano-Zahl verdoppelt. Bei der Applikation

```
double coin
```

hängt nun die Menge der Ergebnisse von der Auswertung des Parameters `coin` ab. Wird das Parameter erst ausgewertet, wenn es benötigt wird, so ergibt sich die Menge der möglichen Ergebnisse `{Z, S Z, S (S Z)}`. Wird das Parameter zuerst ausgewertet, so ergeben sich die möglichen Ergebnisse `Z` und `S (S Z)`. Da es sich bei diesem Ergebnis um das weniger überraschende handelt, folgt Curry dieser Semantik, welche als *call-time-choice*-Semantik bezeichnet wird.
Durch den Nicht-Determinismus gibt es verschiedene Auswertungspfade, welche gewählt werden können. Die Auswahl dieser Auswertungspfade hängt von der Suchstrategie ab. Typische Suchstrategien sind die Tiefensuche und die Breitensuche. Bei der Tiefensuche wird bei jeder Auswahl eine Entscheidung getroffen und auf diese Weise einem einzigen Auswertungspfad gefolgt, bis ein Ergebnis oder ein Fehlschag

erzielt wurde und die Auswertun des Auswertungspfades abgeschlossen ist. Ist dieser Zustand erreicht, so wird zur letzten Auswahl zurückgesprungen und die andere Entscheidung ausgewertet. Bei der Breitensuche wird eine Entscheidung getroffen und der Auswertungspfad verfolgt, bis wieder eine Entscheidung getroffen werden muss. An diesem Punkt wird auf den nicht abgeschlossenen Auswertungspfad gewechselt, dessen Auswertung am längsten zurückliegt.

Gleichheitsconstraints

In Beispiel 2.14 wird die Funktion `last` mit Hilfe von freien Variablen und dem Constraint `=:=` definiert. Bei der Auswertung kann die Regel der Funktion `last` nur

```
append []     ys = ys
append (x:xs) ys = x : append xs ys

last xs | append ys [x] =:= xs = x
  where x,ys free
```

Bsp. 2.14.: Definition der Funktion `last` mit freien Variablen

angewendet werden, wenn eine passende Belegung für die beiden freien Variablen `x` und `ys` gefunden wird. Die Auswertung von

```
last [Z, S Z, S (S Z)]
```

ergibt somit `S (S Z)` mit den Belegungen `ys` → `[Z, S Z]` und `x` → `S (S Z)`.

2.2.2 FlatCurry

FlatCurry ist eine Zwischensprache für den Übersetzungsvorgang von Curry in eine Zielsprache. Ein Curry-Programm kann in ein FlatCurry-Programm transformiert werden. Die einzelnen Phasen dieses Prozesses sind in Abbildung 2.1 abgebildet. Zunächst wird das Curry-Programm durch einen Lexer und einen Parser in die Struktur eines abstrakten Syntaxbaumes überführt. Dann wird eine Syntax- und Typüberprüfung vorgenommen und der abstrakte Syntaxbaum mit den Ergebnissen annotiert. Schließlich werden verschiedene Phasen der Normalisierung durchlaufen. Die aufeinander folgenden Phasen werden im folgenden beschrieben.

Qualification

Sämtliche in einem Modul vorkommenden Bezeichner werden mit dem Modulnamen qualifiziert. Wird zum Beispiel eine Funktion `add` in einem Modul `Peano` deklariert, so ergibt sich der Name `Peano.add`.

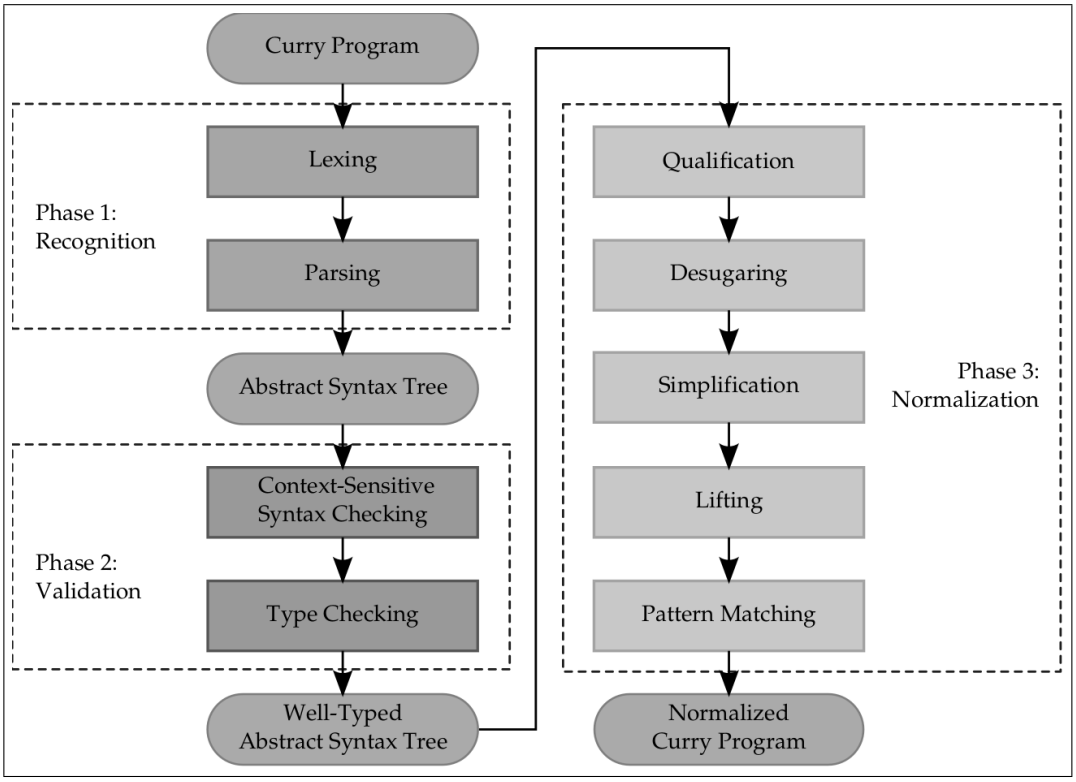


Abb. 2.1.: Die Phasen der Übersetzung von Curry nach FlatCurry. Quelle: [Pee17, S. 52]

Desugaring

Beim *Desugaring* werden Konstrukte aus Curry, welche syntaktischen Zucker darstellen, in andere Konstrukte transformiert. Die wichtigsten Transformationen sind hierbei die Umwandlung von Typdeklarationen mit Zugriffsfunktionen und die Vereinfachung von komplexen Patterns, Ausdrücken und Funktionen[8].

Simplification

In diesem Schritt werden lokale Bindungen in Funktionen mittels *let*-Ausdrücken nach ihrer Abhängigkeit voneinander geordnet. Dann werden lokale Patterns durch Variablenbindungen und Zugriffsfunktionen ersetzt.

Lifting

Beim *Lifting* werden alle lokalen Funktionsdeklarationen zu Funktionen auf Modulebene gehoben. Hierzu werden Variablen, welche auf der rechten Seite der Funktion vorkommen, aber auf der linken Seite nicht gebunden sind, als Parameter der Funktion eingeführt. Bei den Aufrufen der Funktion werden diese Parameter hinzugefügt. Die Funktionen werden dann umbenannt und auf Modulebene gehoben.

[8] Siehe [Pee17, S. 53] für eine detaillierte Beschreibung aller Transformationen.

Pattern Matching

Im letzten Schritt wird das Pattern Matching auf der linken Seite von Funktionsregeln zu *case*-Ausdrücken auf der rechten Seite transformiert. Weiterhin werden die *case*-Ausdrücke auf der rechten Seite so umgewandelt, dass das Pattern nur noch ein Literal oder Konstruktor am Anfang und dann Variablen enthält.

Das resultierende FlatCurry-Programm ist eine semantisch äquivalentes, vereinfachtes Curry Programm.

```
data Peano = Z | S Peano

add :: Peano → Peano → Peano
add v1 v2 = fcase v1 of Z    → v2
                        S v3 → S (add v3 v2)
```

Bsp. 2.15.: Addition auf den Peano-Zahlen in FlatCurry

In Beispiel 2.15 ist die übersetze Version des Curry-Programms aus Beispiel 2.7 dargestellt. Dabei wurde das Pattern Matching in der `add`-Funktion in ein `case`-Ausdruck umgewandelt.

Semantik von FlatCurry

Für die Kernbestandteile von FlatCurry wurde in der Arbeit *Operational Semantics for Declarative Multi-Paradigm Languages*[9] von *Albert, Hanus, Huch, Oliver* und *Vidal* eine Semantik vorgestellt. In dieser Arbeit wird eine Big-Step- und eine Small-Step-Semantik definiert. Die Big-Step-Semantik umfasst dabei die Bedarfsauswertung, das Teilen von auszuwertenden Termen und Nicht-Determinismus, allerdings nicht auf welche Weise der Nicht-Determinismus ausgewertet werden soll. Die Semantik selbst ist damit nicht-deterministisch.
Zu der Big-Step-Semantik wird eine Small-Step-Semantik definiert, welche verschiedene Ergänzungen wie Constraints und externe Funktionen umfasst. Die Small-Step-Semantik wird mit Hilfe eines Heaps definiert und ist deterministisch. Dies wird erreicht, indem für die Auswahlregel wird zusätzlich zum Stack und der Kontrolle eine Menge von Stacks, Kontrollen und Lösungen eingeführt, welche alle nicht-deterministischen Pfade umfasst. Auf dieser Menge können dann Suchstrategien definiert werden.

2.2.3 ICurry

ICurry ist eine Zwischensprache für die Übersetzung von Curry in imperative Sprachen [Kir16, S. 26]. Ein ICurry-Programm wird durch Umwandlung von einem

[9] Im Quellenverzeichnis unter [Alb+05].

FlatCurry-Programm erzeugt. Im folgenden werden die hierbei durchgeführten Transformationen erläutert.

Fallunterscheidungen und lokale Deklarationen

In ICurry kommen case-Ausdrücke in jeder Funktion nur ein einziges Mal vor. Weiterhin kommen Fallunterscheidungen nicht in Ausdrücken vor.
Bei der Transformation wird deshalb jede Fallunterscheidung in FlatCurry in ein neue Funktion in ICurry übersetzt. Die auf der rechten Seite und nicht auf der linken Seite vorkommenden Variablen müssen der Funktion dabei übergeben werden.

Der Grund für diese Transformation liegt darin, dass für die Auswertung eines case-Ausdrucks die Kopfnormalform des gematchten Ausdrucks benötigt wird. Hierfür ist eine Auswertung notwendig, welche beliebig groß sein und neue Funktionsaufrufe und case-Ausdrücke beinhalten kann. Würde diese Auswertung innerhalb der Funktion mit dem case-Ausdruck passieren, könnte es in imperativen Sprachen zu Überläufen des Stacks kommen. Durch die Aufteilung in zwei Funktionen kann die imperative Sprache die Funktion verlassen und einen neuen Schleifendurchlauf beginnen.

Generatoren

In ICurry werden freie Variablen durch Generatorfunktionen ersetzt. Diese Generatorfunktionen generieren sämtliche Werte für einen Typen.
Ein Problem ist hierbei, dass Funktionen mit Typvariablen unterschiedliche Generatorfunktionen je nach ihrer Anwendung benötigen. Um dies zu gewährleisten, wird diesen Funktionen für jede freie Variable ein Generatorparameter hinzugefügt, welches bei der Applikation an eine Generatorfunktion gebunden wird. Die Ersetzung von freien Variablen durch Generatoren ist korrekt und vollständig, solange die Eigenschaften der Konfluenz und Terminierung verletzt werden dürfen [DCLF07].

```
data Peano = Z | S Peano

genPeano = Z ? S genPeano

add v1 v2 = (h0_add $! v1) v2

h0_add v1 v2 = case v1 of Z    → v2
                         S v3 → S (add v3 v2)
```

Bsp. 2.16.: Addition auf den Peano-Zahlen in ICurry

In Beispiel 2.16 ist das von FlatCurry nach ICurry übersetzte Programm aus Beispiel 2.15 in Curry-nahmen Pseudocode dargestellt. Es wurde eine Generatorfunktion für den Datentyp `Peano` eingeführt. Außerdem wurde die `add`-Funktion in zwei Funk-

tionen aufgeteilt, wobei die erste Funktion dafür zuständig ist, die Auswertung des ersten Arguments zur Kopfnormalform zu erzwingen.

Abstrakter Syntax Baum von ICurry Programmen

In Abbildung 2.2 auf Seite 20 ist die Struktur des Abstrakten Syntax Baumes (AST) der ICurry-Programme dargestellt. Die Wurzel eines ICurry-Programms in der AST-Darstellung ist immer mit einem Modulnamen beschriftet und kann mehrere Datentyp- und Funktionsdefinitionen enthalten.
Im Gegensatz zu der Curry-Grammatik werden keine Export- oder Import-Deklarationen benötigt, da sämtliche Identifikatoren nach der Transformation nach ICurry vollständig qualifiziert sind und für die auf das Modul beschränkten Funktionen eine eigene Repräsentation gewählt wurde.
Ein Datentyp hat einen Namen und viele Konstruktoren, welche wiederum einen generierten Namen, eine Stelligkeit und einen Originalnamen besitzen. Der generierte Name kann vom Originalnamen abweichen und wird zur Identifikation benötigt [Kir16, S. 30]. Der Originalname wird für die Ausgabe von Termen benötigt, damit das Ergebnis in sinnvoller Form ausgegeben werden kann.
Bei einer Funktionsdefinitionen kann es sich entweder um eine exportiere oder eine private Funktion handeln. Private Funktion sind Funktionen, welche außerhalb des Moduls nicht sichtbar sind. Die Repräsentationen von privaten und exportieren Funktionen beinhalten Eigenschaften über die Stelligkeit und die Regel, welche die Funktion definiert. Eine exportierte Funktion hält zusätzlich Informationen über ihren Typen. Dieser Typ ist für die richtige Anwendung von Generatoren wichtig [Kir16, S. 35].
Für die Funktionsregeln gibt es drei verschiedene Typen: extern definierte Regeln, einfache Regeln und Case-Regeln.
Eine extern definierte Regel hält Informationen über den Namen der externen Funktion. Eine externe Funktion wird nicht in Curry definiert, sondern in der Zielsprache.
Eine einfache Regel enthält nur die Parameter, lokale Definitionen und einen Ausdruck. Lokale Definitionen halten Informationen über die gebundene Variable und den Ausdruck, der an die Variable gebunden wird.
Ein Ausdruck kann im einfachsten Fall entweder eine Variable oder ein Wert von einem primitiver Typ sein. Ein Ausdruck kann auch ein Funktionsaufruf, ein Konstruktoraufruf, eine Auswahl oder ein Generator sein. Wichtig ist, dass in einem Ausdruck kein case-Ausdruck vorkommen kann.
Der einzige Fall, in welchem ein case-Ausdruck vorkommen kann, ist eine Case-Regel. Diese besteht aus einem Funktionsnamen, einer Liste von Parameters und lokalen Definitionen, einer Liste von zu übergebenden Variablen, eine Variable, welche eine Kopfnormalform benötigt, möglicherweise einen primitiven Wert und einer Liste von Branches. Wie schon in Abschnitt 2.2.3 erklärt, wird jede Funktion mit einer Fallunterscheidung in FlatCurry in zwei Funktionen übersetzt. In der Repräsentati-

on der case-Regel ist der Funktionsname der Name der zweiten Funktion, welche die Fallunterscheidung beinhaltet. Die erste Funktion ist dafür zuständig, dass die richtige Kopfnormalform berechnet wird. Die Liste von zu übergebenden Variablen ist die Liste der Variablen, welche der zweiten Funktion übergeben werden müssen. Der optionale primitive Wert ist nur für die Typidentifikation notwendig, falls das Matching auf einem primitiven Wert stattfindet, damit hier andere Implementierung für das Matching möglich sind [Kir16, S. 31].
Die Branches einer Case-Regel bestehen aus einem Pattern und Parametern oder aus einem primitiven Wert. Wird mit einem Pattern gematcht, so enthält dieser den qualifizierten Identifikator des zu matchenden Kopf des Ausdrucks und eine Menge von Parametern. Diese Parameter sind die neu zu bindenden Variablen des Pattern.

IProg	::=	*ModName* {*IData*} {*IFunc*}	
IData	::=	*DataName* {*ICons*}	(Datentyp)
ICons	::=	*CombName Arity OriName*	(Wertkonstruktor)
IFunc	::=	*CombName Arity TypeExpr IRule*	(exportierte Funktion)
	|	*CombName Arity IRule*	(private Funktion)
IRule	::=	*Arity ExternName*	(extern definierte Regel)
	|	{*IParam*} {*ILocal*} *IExpr*	(einfache Regel)
	|	*CombName* {*IParam*} {*ILocal*} {*VarIndex*} *VarIndex* [*Literal*] {*IBranch*}	(Case-Regel)
IBranch	::=	*IPattern* {*ILocal*} *IExpr*	(Branch eines Cases)
ILocal	::=	*VarIndex IExpr*	(lokale Definition)
IExpr	::=	*VarIndex*	(Variable)
	|	*Literal*	(Literal)
	|	*String*	(String)
	|	{*IExpr*}	(Liste)
	|	*gen VarIndex*	(Generator einer Typvariable)
	|	*noGen String*	(nicht-existenter Generator)
	|	*fCall QName* {*IExpr*}	(Funktionsaufruf)
	|	*cCall QName* {*IExpr*}	(Konstruktoraufruf)
	|	*fPart Arity QName* {*IExpr*}	(partieller Funktionsaufruf)
	|	*cPart Arity QName* {*IExpr*}	(partieller Konstruktoraufruf)
	|	*or* {*IExpr*}	(Disjunktion)
	|	*free* {*IExpr*}	(Disjunktion einer freien Variable)
IPattern	::=	*QName* {*IParam*}	(Konstruktor-Match)
	|	*Literal*	(Literal-Match)
IParam	::=	*VarIndex*	
	|	_	

Abb. 2.2.: Die Struktur von ICurry. Quelle: [Kir16, S. 29f]

2.2.4 Basic Scheme

Das Basic Scheme ist eine Übersetzungsschema für logisch funktionale Programmiersprachen und wurde von *Antoy* und *Peters* entwickelt [AP12]. Letztlich beschreibt das Basic Scheme eine Auswertungsstrategie nicht-deterministischer Operationen in einem Graphersetzungssystem, welches die Eigenschaften besitzt, eingeschränkt überlappend zu sein. Die durch die Transformation von Curry nach FlatCurry enstehenden Programme fallen in die Klasse dieser Graphersetzungssysteme [AP12]. Das Basic Scheme verzögert bei der Graphersetzung die Auswertung von Auswahlen, bis diese getroffen werden können, ohne Backtracking verwenden zu müssen oder Teile des Graphen über dem Knoten kopieren zu müssen. Diese Strategie wird als Pull-Tabbing bezeichnet und führt zu einer korrekten Auswertungsstrategie.

Auf Basis des Basic Schemes wurde ein Fair Scheme definiert, welches auch die Eigenschaft der Vollständigkeit besitzt. Diese Eigenschaft bedeutet, dass bei nicht-deterministischen Berechnungen alle Ergebnisse terminierender Auwertungen gefunden werden können [AJ14]. Diese Eigenschaft wird erreicht, indem in einem festen Interval zwischen den nicht-deterministischen Berechnungspfaden gewechselt wird.

3 Ansätze

Für die Übersetzung von Curry nach JavaScript können verschiedene Ansätze in Betracht gezogen werden. Grundsätzlich ist es sinnvoll, existierende Übersetzer zu benutzen, falls diese die Anforderungen erfüllen.
Da es verschiedene Übersetzer von Haskell nach JavaScript und einen ausgereiften Übersetzer von Curry nach Haskell gibt, ist dies der erste Ansatz, der beschrieben wird. Ein weiterer Ansatz eröffnet sich durch *emscripten*, einem Übersetzer der Zwischensprache *LLVM* nach JavaScript. Da es mit *Sprite* einen Übersetzer von Curry nach LLVM gibt, wird auch diese Möglichkeit analysiert. Als letzter Ansatz wird die direkte Übersetzung von Curry nach JavaScript betrachtet.
Sämtliche Ansätze werden in Bezug auf die im nächsten Abschnitt gesetzten Anforderungen bewertet. Warum der Ansatz gewählt wurde, der im Rahmen dieser Arbeit implementiert wurde, wird am Ende dieses Kapitels beschrieben.

3.1 Anforderungen

Die wichtigste Anforderung an die Ansätze für die Übersetzung von Curry nach JavaScript ist eine vollständige Übersetzung von Curry-Programmen zu ermöglichen. Dabei sollte mindestens die Ausführung der Auswertung vollständig in JavaScript ablaufen.
Eine weitere Anforderung ist, dass einzelne Curry-Modul in einzelne JavaScript-Dateien übersetzt werden und getrennt voneinander ausgeliefert werden können. Die Übersetzung der Curry-Module muss dabei möglich sein, ohne den auszuwertenden Ausdruck zu kennen.
Da ein Einsatz im Web möglich sein sollen, sind dessen Einschränkungen zu beachten. Die übersetzten Module und die Runtime zur Ausführung sollten keine zu große Dateigröße haben.
Die Effizienz der übersetzen Module und der Runtime ist ein ein nachrangiges Kriterium, kann aber bei ähnlich starken Ansätzen in Betracht gezogen werden.

3.2 Curry nach Haskell nach JavaScript

3.2.1 Curry nach Haskell

Für die Kompilierung von Curry nach Haskell existiert ein ausgereifter Übersetzer namens *KiCS2*[1]. Da die Semantik des funktionalen, deterministischen Teils von Curry und Haskell sich überschneiden, muss dieser Teil in KiCS2 nicht durch abstrakte Maschinen implementiert werden [Bra+11, S. 1] und kann wiederverwendet werden.
Der nicht-deterministische Teil von Curry wird in KiCS2 durch Datentypen repräsentiert. Eine nicht-deterministische Auswahl wird durch einen zusätzlichen Konstruktor abgebildet. Zum Beispiel wird dem Datentypen `Bool` bei der Übersetzung nach Haskell ein zusätzlicher Konstruktor `Choice` hinzugefügt, wie in Beispiel 3.1 zu sehen ist. Wird nun der Wert eines solchen Datentyp benötigt und fängt dieser

```
type ID = Integer
data Bool = False | True | Choice ID Bool Bool
```

Bsp. 3.1.: Definition des Curry-Datentyps `Bool` nach der Übersetzung mit KiCS2

Wert mit dem Konstruktor `Choice` an, so muss zunächst der Nicht-Determinismus aufgelöst werden.

KiCS2 ist komplett in Curry und Haskell geschrieben, was auch eine Übersetzung von KiCS2 nach JavaScript durch einen Übersetzer von Haskell nach JavaScript ermöglichen würde. KiCS2 benötigt neben einem Haskell-Übersetzer auch das Paketmanagement-Tool Cabal. Außerdem benötigen mit KiCS2 übersetzte Programme zwei Extensions des Glasgow Haskell Compilers (GHC), nämlich *MagicHash* und *ScopedTypeVariables*. KiCS2 selbst benötigt für die Ausführung auch einige GHC-Extensions.

3.2.2 Haskell nach JavaScript

Es gibt verschiedene Ansätze Haskell nach JavaScript zu übersetzen. Im folgenden werden nur die Ansätze betrachtet, die einen Umfang von Haskell übersetzen, welche eine Ausführung von mit KiCS2 kompilierten Curry-Programmen ermöglicht. Als Anforderungen wird die korrekte und vollständigen Übersetzung von Haskell-Programmen gefordert. Außerdem wird die Möglichkeit, Cabal auszuführen, gefordert. Als weitere Anforderung ist die Möglichkeit, die beiden im letzten Abschnitt

[1] Der Übersetzer ist unter `http://www-ps.informatik.uni-kiel.de/kics2` zu finden.

erwähnten GHC-Extensions zu verwenden, gesetzt. Außerdem sollte der Übersetzer möglich kleine und effiziente Programme erzeugen.

UHC

UHC ist ein Haskell Übersetzer, welcher auch JavaScript als Zielsprache unterstützt. UHC implementiert Haskell98. Viele Haskell Extensions funktionieren nur unter Haskell2010 und können deshalb in UHC nicht benutzt werden [Ekb15, S. 20]. Da viele Cabal-Pakete und auch KiCS2 GHC-Extensions benutzen, verfehlt UHC die Anforderungen. Außerdem ist UHC um etwa um eine Größenordnung langsamer als GHC [Ekb12, S. 13] und generiert JavaScript-Programm, die sehr groß sind [Ekb12, S. 13], was im Web hinderlich ist.

GHCJS

GHCJS ist ein Übersetzer von Haskell nach JavaScript, der auf GHC basiert und darauf abzielt die Haskell-Semantik möglichst vollständig umzusetzen. GHCJS versucht unter anderem die Semantik des Threading aus dem GHC in JavaScript zu imitieren. GHCJS bietet ein eigenes Paket-Verzeichnis um Cabal zu unterstützen.
Der Übersetzer hat verschiedene Nachteile, die ihn für dieses Projekt ungeeignet machen. Seine Funktionalität und der Übersetzungsprozess sind schlecht dokumentiert und auch der Build-Prozess von GHCJS selbst ist schlecht dokumentiert [Ekb12, S. 12].
Der von GHCJS generierte JavaScript-Quelltext ist sehr groß [Ekb12, S. 11], unübersichtlich und in verschiedene Dateien geteilt.
Auch der Export von Haskell-Funktion nach JavaScript ist zum Zeitpunkt dieser Arbeit noch nicht ausgereift.[2] Die Autoren empfehlen für diesen Anwendungsfall eine andere Sprache zu benutzen.[3]
Im Vergleich zum im nächsten Abschnitt beschriebenen Übersetzer *Haste* ist der generierte Code etwa 6 bis 10 mal größer und um den Faktor 2 langsamer [Ekb15, S. 45ff].

Haste

Haste ist ein Haskell nach JavaScript Übersetzer, der GHC zur Kompilierung bis zu der Zwischenrepräsentation der Sprache für die Spineless-Tagless-G-Machine (STG) benutzt und danach eine eigenen Übersetzer von STG nach JavaScript bereitstellt. Dabei unterstützt Haste die vollständige Haskell-Semantik ist aber nicht vollständig kompatibel zu GHC. Unterstützt werden viele Extensions, eine wichtige Ausnahme bildet Template Haskell. Haste bietet auch eine Cabal-Integration. Bei der Über-

[2] Siehe dazu die Diskussion unter `https://github.com/ghcjs/ghcjs/issues/194`.
[3] Dies wird im Repository unter `https://github.com/ghcjs/ghcjs/blob/master/doc/foreign-function-interface.md#calling-haskell-from-javascript` beschrieben.

setzung nach JavaScript werden endrekursive Aufrufe durch Trampolining transformiert.

Die von Haste generierten Programme sind klein und schnell. Haste bietet weiterhin ein Foreign-Function-Interface, welches den Aufruf von externen JavaScript-Funktionen in dem übersetzten Haskell-Programm ermöglicht, und auch den Aufruf der übersetzten Haskell-Funktionen in JavaScript zulässt. Diese Interface basiert auf dem Foreign-Function-Interface von GHC. Diese Interface lässt allerdings keinen Export von Funktionen mit Typparametern zu. Aus diese Grund verfehlt Haste die Anforderungen.

GHC und emscripten

Der Haskell-Übersetzer GHC bietet die Möglichkeit nach *LLVM* zu übersetzen. LLVM ist ein virtueller Maschinenbefehlssatz und dient als Zwischensprache zwischen diversen Frontends und Backends. Für LLVM gibt es einen Übersetzer nach WebAssembly, welches mit *asm.js* kompatible ist, und eine performante Ausführen von maschinennahen Instruktionen im Browser ermöglicht[Zak11]. Dieser Übersetzer heißt *emscripten*. Ein Problem bei der Übersetzung von mit GHC übersetzten Programmen mit emscripten ist, dass die GHC-Runtime auch übersetzt werden muss. Hierfür gibt es momentan keine Lösung und selbst die GHCJS-Entwickler lehnen diesen Ansatz momentan ab.

3.3 Curry nach LLVM nach WebAssembly

Sprite ist ein nativer Curry-Übersetzer, der 2016 von *Antoy* und *Jost* vorgestellt wurde [AJ16]. Sprite transformiert dabei ein Curry-Programm in ein Graphersetzungssystem. Ein Curry-Term bildet dabei einen Graphen und die Semantik von Curry bildet zusammen mit den Funktionsdefinitionen die Transformations- und Ersetzungsregeln auf dem Graphen.
Die Zielsprache von Sprite ist LLVM. Eine Übersetzung des von Sprite erzeugten LLVM-Codes mit emscripten gestaltet sich als sehr schwierig, da die LLVM-Version, welche Sprite benutzt, eine andere ist, als die, welche emscripten übersetzt. Weiterhin benutzt Sprite auch LLVM-Sprachkonstrukte, die emscripten nicht übersetzen kann. Es ist daher nicht möglich das von Sprite generierte LLVM nach WebAssembly zu übersetzen.
Ein weiteres Problem ist, dass Sprite selbst noch nicht ausgereift ist. Allerdings sind die Konzepte von Sprite für die direkte Übersetzung von Curry nach JavaScript sehr interessant.

3.4 Direkte Übersetzung von Curry nach JavaScript

3.4.1 Existierender Übersetzer eines Curry-Subsets nach JavaScript

Ein Übersetzer, welche einige Konstrukte der Curry-Programmiersprache nach JavaScript übersetzen kann, existiert bereits. Die Motivation ist, die Constraints hinter Formularen effektiver definierten und überprüfen zu können. Dazu werden diese Constraints in Curry definiert und dann nach JavaScript übersetzt. Der Übersetzer operiert dabei unter den Einschränkungen, dass Funktionen total definiert sein müssen, Nicht-Determinismus ausgeschlossen ist und auch unendliche Datenstrukturen nicht erlaubt sind [Han07]. Dies ermöglicht die Auswertung von Curry-Programmen ohne die Bedarfsauswertung. Aufgrund dieser Einschränkungen erfüllt der Übersetzer nicht die Anforderungen.

3.4.2 Übersetzung nach dem Konzept von Cam

Von *Kirchmayr* wurde im Jahr 2017 ein Übersetzer namens *Cam* entwickelt, der ähnliche Eigenschaften, wie der Übersetzer Sprite besitzt [Kir16], implementiert. Cam übersetzt von Curry nach Java und basiert auch auf einem Graphersetzungssystem. Dabei wird neben der Repräsentation von Curry-Programmen als Graphen auch das Pull-Tabbing und das Ersetzen von freien Variablen durch Generatoren eingesetzt.
Da Java eine imperative Sprache mit ähnlicher Syntax wie JavaScript ist, können einige Konzepte sehr gut übernommen werden. Für Cam wurde die Zwischensprache ICurry implementiert, um eine einfache Übersetzung in die imperative Sprache Java zu ermöglichen. Diese Zwischensprache könnte bei diesem Ansatz wiederverwendet werden.

3.5 Gewählter Ansatz

Für diese Arbeit wurde die direkte Übersetzung von Curry nach JavaScript gewählt. Dabei erfolgt die Übersetzung ähnlich wie bei dem Übersetzer Cam, welcher von Curry nach Java übersetzt. Es wird zunächst mit dem Curry Frontend nach ICurry übersetzt und daraus ein JavaScript-Programm generiert. Die Programmauswertung wird auf einer Graphstruktur ausgeführt und die Auswertung von Nicht-Determinismus wird durch Pull-Tabbing verzögert. Die Implementierung folgt damit auch den Konzepten des Basic Scheme.

4 Implementierung

4.1 Semantik von ICurry-Programmen

ICurry ist eine Zwischensprache, die aus einer Transformation von FlatCurry entsteht. Da aus ICurry ein JavaScript-Programm generiert werden soll, muss zunächst die Semantik von ICurry festgelegt werden.[1] Diese Semantik orientiert sich an der Semantik aus der Arbeit [Alb+05] und dem in Abschnitt 2.2.4 beschriebenen Basic Scheme aus der Arbeit [AP12].

Schreibweise

Bei jeder in der nachfolgenden Semantik definierten Regeln handelt es sich immer um zwei Zeilen. Die erste Zeile repräsentiert den Zustand, auf den die Regel angewendet wird, die zweite Zeile repräsentiert den Zustand nach der Anwendung. Passt der aktuelle Zustand auf den Zustand, welcher in der ersten Zeile angegeben wird, so ist die Regel anwendbar. Ein Zustand ist dabei ein Tripel (G, c, Q).

G repräsentiert den Graphen, dessen für die Regel relevanter Teil in der Spalte „Teilgraph“ abgebildet ist. G ist ein Tupel (K, E) von Knoten K und Kanten $E \subseteq K \times K$. Knoten werden als $ref : s$ dargestellt, dabei ist ref aus der Menge der Referenzen R, welche einen Zugriff von außerhalb des Graphen erlauben, und s aus der Menge der Beschriftungen S.
Bei den Beschriftungen S kann es sich um Konstruktorsymbole, Funktionssymbole, das Auswahlsymbol $?$ oder das Identitätssymbol $\hookrightarrow$ handeln.
Eine Beschriftung mit einem Funktionssymbol kann zusätzlich den Index i eines Kindes beinhalten, dies wird mit hnf_i dargestellt, falls es relevant ist. Eine Beschriftung für ein Funktionssymbol enthält immer die Stelligkeit der Funktion, was in den Abbildungen aus Gründen der Übersichtlichkeit nicht dargestellt wird. Die Stelligkeit ergibt sich in dieser Darstellung daher durch die Anwendung von $arity$ auf die Beschriftung.
Eine Beschriftung mit einem Auswahlsymbol $?$ enthält zusätzlich immer einen Identifier id aus der unendlichen Menge IDs.

[1] Im Folgenden werden die einzelnen Regeln beschrieben, eine Übersicht über alle Regeln ist im Anhang ab S. 75 aufgeführt.

c ist der momentane Werte in der Kontrolle, welche in der Spalte „Kontrolle“ dargestellt wird. Es kann sich dabei um eine Referenz, die Symbole $ oder €, eine Projektion, einen Fehlschlag oder einen Ausdruck handeln.
Das Symbol $ wird genutzt, um die Terminierung der Auswertung zu signalisieren. Referenzen werden in den Abbildungen immer ref genannt, um sie von Ausdrücken abzugrenzen.

Q ist eine Liste und wird in der Spalte „Queue“ abgebildet. $\{\}$ repräsentiert die leere Liste. Mit dem Operator ; können Elemente vor die Liste gehängt werden. Mit dem Operator $+\!\!+$ können zwei Listen dieses Typs zusammengehängt werden.
In der Liste Q können sich Tupel (s, f) befinden.
Dabei ist s eine Liste von Referenzen R. Die Listendarstellung der Listen s unterscheidet sich aus Gründen der Lesbarkeit von der Listendarstellung für Q, es wird [ ] für die leere Liste und : für die Anhängen am Anfang verwendet.
f ist eine Abbildung auf der Relation $IDs \times \{0, 1, 2\}$ und hat die Form $g[x_0 \to n_0] \ldots [x_m \to n_m]$. Für $g[\ldots] \ldots [\ldots][x \to n]$ gilt $y = x \Rightarrow g[\ldots] \ldots [\ldots][x \to n](y) = n$ und $y \neq x \Rightarrow g[\ldots] \ldots [\ldots][x \to n](y) = g[\ldots] \ldots [\ldots](y)$. Weiterhin gilt $g(y) = 0$ für alle $y \in IDs$.
$newid()$ wählt einen Wert id, für den im momentanen Zustand gilt $f(id) = 0$.

Als Schreibweise wird außerdem $\overline{x_n}$ und $\overline{x_{n\ldots m}}$ verwendet. Der erste Ausdruck bezeichnet $x_0, \ldots, x_n$, der zweite Ausdruck bezeichnet $x_n, \ldots, x_m$.

Die Regeln der Semantik

Mit Hilfe dieser Definitionen kann nun die Semantik definiert werden. Zunächst wird festgelegt, dass die Auswertung des Nicht-Determinismus durch den Einsatz einer Queue erfolgt. Für jeden Auswertungspfad gibt es in dieser Queue einen Stack. In

Regel	Teilgraph	Kontrolle	Queue
1		€ ref_0	$(ref_0 : S, F); Q$ $(S, F); Q$
2		€ $	$\{\}$ $\{\}$
3		fail €	$S; Q$ Q

Abb. 4.1.: Die Semantik der Queue

Abbildung 4.1 sind die Grundregeln dieser Queue definiert. In Regel 3 liegt *fail* in der Kontrolle. Dies bedeutet, die nicht-determinstische Berechnung mit dem aktuellen Stack ist fehlgeschlagen. Dieser Stack liegt ganz vorne in der Queue (repräsentiert durch *S*). Es wird nun ein Symbol in die Kontrolle gelegt, welches darauf hinweist,

dass mit dem nächsten Stack in der Queue fortgefahren werden soll (€). Weiterhin wird der aktuelle Stack S von der Queue genommen.

Die Anwendung der nächsten Regel hängt dann davon ab, ob die Queue leer ist oder nicht. Ist die Queue leer, so wird *$* als Symbol für die Terminierung in die Kontrolle gelegt (Regel 2). Die Auswertung ist fertig. Ist die Queue nicht leer, so wird das erste Element des ersten Stacks in die Kontrolle gelegt (Regel 1). Dieses Element ist eine Referenz auf einen Knoten im Graphen (hier ref_0).

Die Regeln 1, 2 und 3 haben gemeinsam, dass der Graph keine Relevanz für die Anwendung hat. Die Anwendung dieser Regeln ist allein abhängig von der Kontrolle und dem Zustand der Queue. Neben *fail* gibt es noch weitere Regeln, welche die Auswertung eines Stacks zunächst abschließen. Hierbei handelt es sich um Regeln, bei denen entweder ein Term in Kopfnormalform in der Kontrolle referenziert wird, oder eine nicht-deterministische Auswahl notwendig ist.

Regel	Teilgraph	Kontrolle	Queue
4	ref_0: c ref_0: c	ref_0 €	$([\,], F); Q$ Q
5	ref_0: f $ref_1 \quad \ldots \quad ref_n$ ref_0: f $ref_1 \quad \ldots \quad ref_n$	ref_0 €	$([\,], F); Q$ Q

Bedingungen für Regel 5: $n <$ arity(f)

Abb. 4.2.: Die Regeln für Kopfnormalformen

In Abbildung 4.2 werden die Regeln für die Kopfnormalformen definiert. In Regel 4 liegt ref_0 in der Kontrolle. ref_0 zeigt auf einen Knoten im Graphen, der mit dem Konstruktor c beschriftet ist. Weiterhin ist der Stack, der oben auf der Queue liegt, leer. Da der Stack die Position im auszuwertenden Term repräsentiert, bedeutet ein leer Stack, dass in der Kontrolle der Kopf des Terms liegt. Daraus kann geschlossen werden, dass am Kopf des Terms ein Konstruktor steht, was bedeutet, dass der Term sich in Kopfnormalform befindet. Die Auswertung dieses Stacks ist somit abgeschlossen. Der Stack wird von der Queue entfernt und das Symbol für den Abschluss der Auswertung des Stacks in die Kontrolle gelegt.

Regel 5 ähnelt im Resultat der Regel 4. In Regel 5 liegt auch eine Referenz ref_0 in der Kontrolle, die jedoch auf einen Funktionsknoten zeigt, der mit einer Funktion f beschriftet ist. Da der oberste Stack auf der Queue leer ist, steht die Beschriftung

des Knoten ref_0 am Kopf des Terms. Gilt nun die Bedingung, dass die Anzahl der Kinder von ref_0 echt kleiner ist als die Stelligkeit der Funktion f, so ist der Term in Kopfnormalform, da es sich um eine partielle Funktionsanwendung handelt. In diesem Fall wird der Stack aus der Queue entfernt und das Symbol für den Abschluss der Auswertung des Stacks in die Kontrolle gelegt.

Der letzte Fall, in welchem das $\in$ Symbol für den Abschluss der Auswertung des Stacks in die Kontrolle gelegt wird, wird bei den Regeln für den Nicht-Determinismus besprochen.

Regel	Teilgraph	Kontrolle	Queue
6	ref_0: c	ref_0	$(ref_x : S, F); Q$
	ref_0: c	ref_x	$(S, F); Q$

Abb. 4.3.: Die Regel zur Abarbeitung des Stacks bei Konstruktor in der Kontrolle

In Abbildung 4.3 ist die zweite Regel für die Abarbeitung eines Konstruktors definiert. Liegt eine Referenz auf einen Knoten, der mit einem Konstruktor beschriftet ist, in der Kontrolle und ist der Stack nicht leer, so ist der Teilterm in Kopfnormalform. Es kann also der im Pfad vorhergehende Knoten weiter ausgewertet werden. Dementsprechend wird der Knoten ref_x vom Stack in die Kontrolle gelegt.

Der Abstieg in den Graphen, der die Anwendung der Regel 6 sinnvoll macht, erfolgt bei der Funktionsapplikation, deren Regeln in Abbildung 4.4 zu sehen sind. In Regel

Regel	Teilgraph	Kontrolle	Queue
7	ref_0: f_{hnf_i} ref_1 ... ref_i: g ... ref_n	ref_0	$(S, F); Q$
	ref_0: f_{hnf_i} ref_1 ... ref_i: g ... ref_n	ref_i	$(ref_0 : S, F); Q$
8	ref_0: f ref_1 ... ref_i ... ref_n	ref_0	$(S, F); Q$
	ref_0: f ref_1 ... ref_i ... ref_n	$\sigma(exp)$	$(ref_0 : S, F); Q$

Bedingungen für Regel 7: ref_i nicht in Kopfnormalform

Regel 8: $f(\overline{par_n}) = exp \in$ Programm
$\sigma = \{\overline{par_n \rightarrow ref_n}\}$
ref_1 bis ref_n nicht mit $\hookrightarrow$ beschriftet

Abb. 4.4.: Die Regeln für die Funktionsapplikation ohne $\hookrightarrow$ als Kinderknoten

7 ist ref_0 mit einer Funktion beschriftet, die die Kopfnormalform des Kindes ref_i benötigt. Da nach den Bedingungen ref_i nicht in Kopfnormalform ist, muss ref_i zunächst in Kopfnormalform gebracht werden. Hierzu wird ref_0 auf den aktuellen Stack gelegt und ref_i in die Kontrolle gelegt.

In Regel 8 benötigt f keine Kopfnormalform mehr. Es wird daher eine Ersetzung der in der Kontrolle liegende Referenz durch einen Ausdruck vorgenommen. Dabei wird ref_0 auf den Stack gelegt, damit diese Referenz nach der Auswertung des Ausdrucks auf dessen Resultat zeigen kann. Die Ersetzung von ref_0 durch einen Ausdruck erfolgt dabei mit Hilfe der Funktionsdefinition von f. Hierzu wird im Rumpf von f die Ersetzung der in der Funktionsdefinition gebunden Variablen durch die passenden Kinder von ref_0 durchgeführt. Dann wird dieser Rumpf in die Kontrolle gelegt.

Bei dieser Funktionsapplikation sind Identitätsknoten ausgenommen, welche nur als Weiterleitung dienen. Diese Knoten werden später, zusammen mit den Regeln für die Funktionsapplikation, in denen Identitätsknoten vorkommen, eingeführt.

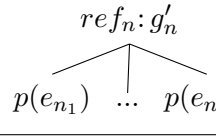
Regel	Teilgraph	Kontrolle	Queue
9	ref_0: f … … … ref_0: g' ref_1 … ref_n	$g(\overline{ref_n})$ ref_0	$(ref_0 : S, F); Q$ $(S, F); Q$

Bedingungen für Regel 9: $g \in Constructors \cup Functions \cup \{?\}$
$g =? \Rightarrow g' = g_{newid()}$
$g \neq? \Rightarrow g' = g$

Abb. 4.5.: Die Regel zur Auswertung eines Funktionsaufrufsausdruck

In Abbildung 4.5 ist die Regel aufgeführt, welche angewendet wird, wenn in der Kontrolle ein Ausdruck, welcher mit einem Funktionssymbol, einem Konstruktorsymbol oder einer nicht-deterministischen Auswahl beginnt, steht. Beginnt ein Ausdruck in der Kontrolle auf diese Weise, folgen im Ausdruck dann immer Referenzen auf Knoten. Diese Referenzen stellen die Argumente dar.

Wird die Regel angewendet, so wird die oberste Referenz vom Stack geholt und in die Kontrolle gelegt. Der Knoten, auf den diese Referenz zeigt, wird mit dem Funktions-, Konstruktions- oder Auswahlsymbol beschriftet und die Kinder durch die im Ausdruck dem Symbol folgenden Referenzen ersetzt. Weiterhin wird, falls es sich um ein Auswahlsymbol handelt, die Beschriftung des Knoten zusätzlich mit einem neuen, einzigartigen Identität versehen. Dies ist notwendig, da es sich um eine gänzlich neue Auswahl handelt.

Regel	Teilgraph	Kontrolle	Queue
10		$let\ \{\overline{x_n = g_n(\overline{e_{n_m}})}\}\ in\ e$ $p(e)$	$(S, F); Q$ $(S, F); Q$

Bedingungen für Regel 10: $p = \{\overline{x_n \to ref_n}\}$, $\overline{ref_n}$ neue Referenzen
$e_{k_j} \in Referenzen \cup \{\overline{x_n}\}$
$g_k =? \Rightarrow g'_k = g_{k_{newid()}}$
$g_k \notin \{?\} \cup Referenzen \Rightarrow g'_k = g_k$
$\overline{g_n} \in Constructors \cup Functions \cup \{?\}$

Sonderfall für Regel 10: Wenn $g_k = ref_x$, dann wird ein Knoten $ref_k :\hookrightarrow$ mit Kind ref_x erstellt

Abb. 4.6.: Die Regel zur Auswertung von *case* und *let*

In Abbildung 4.6 ist die Auswertung von *let*-Ausdrücken definiert. Dabei werden für die in dem let-Ausdruck neu definierten Variablen $\overline{x_n}$ neue Referenzen $\overline{ref_n}$ eingeführt. Für diese Referenzen $\overline{ref_n}$ werden jeweils neue Knoten erstellt. Jeder Knoten ref_i wird dann mit dem Kopf des Ausdrucks beschriftet, der an die Variable x_i gebunden wurde. Im restlichen Teil des Ausdrucks werden die lokalen Variablen durch Referenzen ersetzt, welche dann von dem Knoten als Kinder referenziert werden.

Die Argumente werden mit Hilfe der Abbildung von neue eingeführten Variablen auf Referenzen durch Referenzen ersetzt und als Kinder des Knotens eingesetzt. Wie bei Regel 9 werden neue Identitäten für Auswahlsymbole eingeführt.

Wichtig ist hier die Sonderbedingung, falls ein Ausdruck einfach eine Referenz ist. Dies wird im weiteren als Projektion bezeichnet. In diesem Fall muss ein Identitätsknoten mit der Referenz als Kind eingeführt werden und die neue Referenz auf diesen Knoten zeigen. Diese Identitätsknoten sind notwendig. Ein Beispiel kann dies verdeutlichen. In Beispiel 4.1 ist ein Curry-Programm abgebildet, welches ei-

```
main = let a = hard
       in (id a) + (id a)
```

Bsp. 4.1.: Ein Programm mit Projektionen und teuren Berechnungen

ne speicher- und laufzeitintensive Operation `hard` ausführt. Es ist dementsprechend sinnvoll, dass die Berechnung nur einmal ausgeführt wird. In Abbildung 4.7 wird der Ausdruck aus Beispiel 4.1 in der Spalte *Start* als Graph dargestellt. Wird kein Identitätsknoten genutzt, so ergibt sich bei der Auswertung *Resultat 1*. In diesem Fall wird die Funktion `hard` zweimal ausgewertet. Dies bedeutet, dass der durch den Auflösung der Funktion entstehende Graph auch zweimal existiert und es keine gemeinsame Benutzung dieses Graphen gibt. Im Knoten ref_2 kann ref_1 nicht als

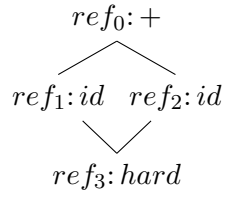
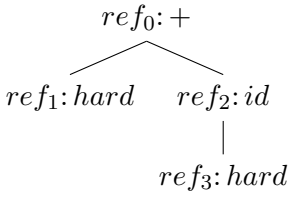
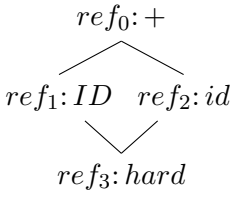
Start	Resultat 1	Resultat 2
		

Abb. 4.7.: Eine Graphauswertung mit Projektionen und teurer Berechnung

Kind gesetzt werden, da ref_3 seine Eltern nicht kennt. Auch ref_0 kann nicht direkt auf ref_3 zeigen, da andere Knoten noch auf ref_1 zeigen könnten. Außerdem müsste ref_0 hierfür vom Stack geholt werden, was außerhalb der Funktion passieren müsste.
Bei *Resultat 2* wird der Knoten ref_1 mit *ID* markiert und damit nur als Indirektion auf Knoten ref_3 verwendet. Die Auswertung von *hard* kann somit wiederverwendet werden. Der Nachtteil ist eine kompliziertere Semantik und dass ein solche Indirektion aufgelöst werden muss, wenn der Knoten verwendet werden soll.

Neben der Regel 10 gibt es noch einen Fall, wo eine Projektion auftreten kann, nämlich dass ein Ausdruck in der Kontrolle eine Referenz ist. Dies kann nach einer Funktionsapplikation und der Überführung des Ausdrucks in einen Graphen passieren. In

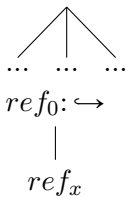
Regel	Teilgraph	Kontrolle	Queue
11		proj(ref_x) ref_0	$(ref_0 : S, F); Q$ $(S, F); Q$

Abb. 4.8.: Regel für die Projektion bei Auswertung eines Ausdrucks

Abbildung 4.8 wird die Auswertung in diesem Fall definiert. Dabei wird der Knoten ref_0, welcher oben auf dem Stack liegt, in die Kontrolle gelegt. Weiterhin wird der Knoten mit dem Identitätssymbol ↪ beschriftet und der projizierte Knoten als Kind angefügt.

Die Regeln zur Dereferenzierung von Identitätsknoten sind in Abbildung 4.9 definiert. Liegt ein Identitätsknoten in der Kontrolle, so wird sein Kindknoten in die Kontrolle gelegt und fortgefahren. Liegt ein Funktionsknoten in der Kontrolle und hat dieser Funktionsknoten einen Identitätsknoten als Kind, so wird dieser Identitätsknoten durch sein Kind ersetzt. Dabei bleibt allerdings der Identitätsknoten erhalten, da andere Knoten auf den Identitätsknoten zeigen könnten.

Regel	Teilgraph	Kontrolle	Queue
12	ref_0: $\hookrightarrow$ | ref_x	ref_0	$(S, F); Q$
	ref_0: $\hookrightarrow$ | ref_x	ref_x	$(S, F); Q$
13	ref_0: f ref_1 ... ref_i: $\hookrightarrow$ ... ref_n | ref_{i_0}	ref_0	$(S, F); Q$
	ref_0: f $\quad$ ref_i: $\hookrightarrow$ ref_1 ... ref_{i_0} ... ref_n $\quad$ ref_{i_0}	ref_0	$(S, F); Q$

Abb. 4.9.: Die Regel für die Dereferenzierung von Identitätsknoten

Regel	Teilgraph	Kontrolle	Queue
14	ref_0: c ref_1 ... ref_n	$case\ ref_0\ of\ \{\overline{pat_m \rightarrow exp_m}\}$	$(S, F); Q$
	ref_0: c ref_1 ... ref_n	$\sigma(exp_j)$	$(S, F); Q$
Bedingungen für Regel 14: $pat_j = c(\overline{par_n})$ $\sigma = \{\overline{par_n \rightarrow ref_n}\}$			

Abb. 4.10.: Die Regel für die Auswertung von *case*-Ausdrücken

In Ausdrücken kann auch ein *case* vorkommen, wobei ein *case* in ICurry maximal einmal pro Ausdruck vorkommen kann.[2] Die Regel für einen case-Ausdruck ist in Abbildung 4.10 abgebildet. Dabei wird ein Pattern mit einem durch ref_0 referenzierten Knoten geprüft. Ein Pattern ist dabei auf einen Konstruktorsymbol und Variablen beschränkt[3]. Bei dem Knoten kann es sich nur um einen Konstruktorknoten handeln, da das Programm sonst nicht frei von Typfehlern wäre oder die Berechnung der Parameter, die eine Kopfnormalform brauchen, in der ICurry-Übersetzung fehlerhaft wäre. Ist ein Pattern mit dem richtigen Konstruktor vorhanden, so wird der zum Pattern passende Ausdruck in die Kontrolle gelegt. In dem Ausdruck werden vorher die in dem Pattern eingeführten Variablen durch die zugehörigen Kinder des Knotens ref_0 ersetzt.

[2] Siehe Abschnitt 2.2.3.
[3] Siehe Abschnitt 2.2.2 unter Pattern Matching.

Der deterministische Teil der Semantik ist mit den Regeln 4 bis 14 definiert, es fehlt noch die nicht-deterministische Auswahl. In Abbildung 4.11 ist die erste Regel

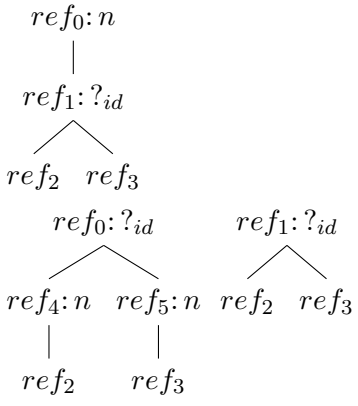
Regel	Teilgraph	Kontrolle	Queue
15		ref_1 ref_0	$(ref_0 : S, F); Q$ $(S, F); Q$

Abb. 4.11.: Die Regel für den Pull-Tab-Schritt

für die nicht-deterministische Auswahl dargestellt. Steht eine nicht-deterministische Auswahl nicht am Anfang des Stacks, so werden Pull-Tab-Schritte ausgeführt, bis der Stack und der Knoten sich somit ganz oben im Graphen befindet. Pull-Tabbing besitzt dabei die Eigenschaften, keine vorschnelle, nicht-deterministische Entscheidung zu treffen, welche zu einer Unvollständigkeit der Auswertung führt, und gleichzeitig nicht unnötig große Teile des Graphen zu kopieren [Ant11]. Die Korrektheit von Pull-Tabbing wurde von *Antoy* im Jahr 2011 bewiesen [Ant11].
In Regel 15 liegt eine Referenz ref_1 auf einen Knoten in der Kontrolle. Dieser Knoten ist mit einer Auswahl beschriftet. Weiterhin ist der Stack nicht leer. Die Referenz, die ganz oben auf dem Stack liegt, ist ref_0. Daraus lässt sich schließen, dass ref_1 ein Kind von ref_0 ist. Nun werden zwei neue Knoten mit den Referenzen ref_4 und ref_5 eingeführt. Diese Knoten werden mit der Beschriftung von ref_n versehen, bilden in dieser Hinsicht also eine Kopie. Jeder dieser Knoten erhält weiterhin eine Auswahl als Kind. An die Referenz ref_0 wird ein neuer Auswahlknoten geschrieben, welcher ref_4 und ref_5 als Auswahl erhält. Der ref_1 bleibt weiterhin erhalten, da andere Knoten noch auf ref_1 zeigen könnten. In die Kontrolle wird der neue Auswahlknoten ref_0 gelegt.

Ist der Stack leer und eine Referenz auf einen Auswahlknoten liegt in der Kontrolle und wurde für die Identität des Auswahlknotens noch keine Entscheidung getroffen, so gibt es zwei Regeln, die je nach Auswertungsstrategie angewendet werden können. In Abbildung 4.12 ist die Regel für die Breitensuche dargestellt. In der Kontrolle liegt ein Auswahlknoten und der Stack ist leer. Weiterhin gilt, dass noch keine Entscheidung für die Identität id getroffen wurde. Um beide Möglichkeiten nicht-deterministisch im Sinne der Breitensuche zu berechnen, wird zunächst die eine Auswahl berechnet, bis erneut eine Auswahl getroffen werden muss. Diese Auswahl wird dann aufgeschoben, und zunächst die andere Möglichkeit berechnet, bis eine

Regel	Teilgraph	Kontrolle	Queue
16a	ref_0: $?_{id}$ ref_1 ref_2	ref_0	$([\,], F); Q$
	ref_0: $?_{id}$ ref_1 ref_2	€	$Q +\!\!+ \{([ref_1], F[id \to 1]\} +\!\!+ \{([ref_2], F[id \to 2]\}$
Bedingungen für Regel 16a: $F(id) = 0$			

Abb. 4.12.: Die Regel für die Breitensuche

Auswahl getroffen werden muss. Dies wird in der Regel 16a erreicht, indem zwei neue Stacks angelegt werden. Auf dem einen Stack liegt die erste Möglichkeit, auf dem anderen Stack die zweite Möglichkeit. Zu jedem Stack werden die bisher getroffenen Auswahlen mitgeführt und die in diesem Schritt getroffene Auswahl hinzugefügt. Dies Stacks und die Speicher der Auswahlen werden dann hinten auf die Queue gelegt.

Regel	Teilgraph	Kontrolle	Queue
16b	ref_0: $?_{id}$ ref_1 ref_2	ref_0	$([\,], F); Q$
	ref_0: $?_{id}$ ref_1 ref_2	€	$([ref_1], F[id \to 1]); ([ref_2], F[id \to 2]); Q$
Bedingungen für Regel 16b: $F(id) = 0$			

Abb. 4.13.: Die Regel für die Tiefensuche

In Abbildung 4.13 ist die Regel 16b für die Tiefensuche dargestellt. Sie gleicht der Regel 16a, allerdings werden die Stacks und die Speicher der Auswahlen vorne auf die Queue gelegt. Dies führ dazu, der vordere Stack erst vollständig ausgewertet wird, bevor die Auswertung des hinteren Stacks beginnt. Es handelt sich somit um eine Auswertung im Sinne der Tiefensuche.

Sowohl die Breitensuche als auch die Tiefensuche ziehen das linke Kind des Auswahlknotens vor. Terminiert die Auswertung dieses Kindes nicht, so terminiert das Programm nicht. Die nicht-deterministische Berechnung nach dieser Semantik ist damit nicht vollständig. Eine Vollständigkeit könnte erreicht werden, indem nach jeder Regelanwendung nach der eine Referenz in der Kontrolle liegt, diese oben auf den momentanen Stack gelegt wird, der Stack ans Ende der Queue gelegt wird und € in die Kontrolle gelegt wird.

Zuletzt wird in Abbildung 4.14 die Auswertung einer Auswahl, welche schon getroffen wurde, dargestellt. Im Speicher F ist die Identität der Auswahl id bereits belegt.

Regel	Teilgraph	Kontrolle	Queue
17	ref_0: $?_{id}$ ref_1 ref_2	ref_0	$([\,], F); Q$
	ref_0: $?_{id}$ ref_1 ref_2	ref_i	$([\,], F); Q$
Bedingungen für Regel 17: $F(id) = i \wedge i \in \{1, 2\}$			

Abb. 4.14.: Die Regel für getroffene Auswahlen

Es wird der entsprechende Kindknoten des in der Kontrolle referenzierten Knotens in die Kontrolle gelegt.

Die Runtime kann nun nach dieser Semantik implementiert werden. Gleichzeitig muss der Übersetzer das Curry-Programm so überführen, dass die Einschränkungen durch die Semantik erfüllt werden und ein korrektes Programm entsteht.

4.2 Architekturübersicht

In Abbildung 4.15 ist die Architektur des idealen Übersetzers dargestellt. Diese Architektur wurde in der Implementierung für diese Arbeit nicht umgesetzt. Stattdessen wurde eine einfachere Variante implementiert. In der idealen Variante werden vor-

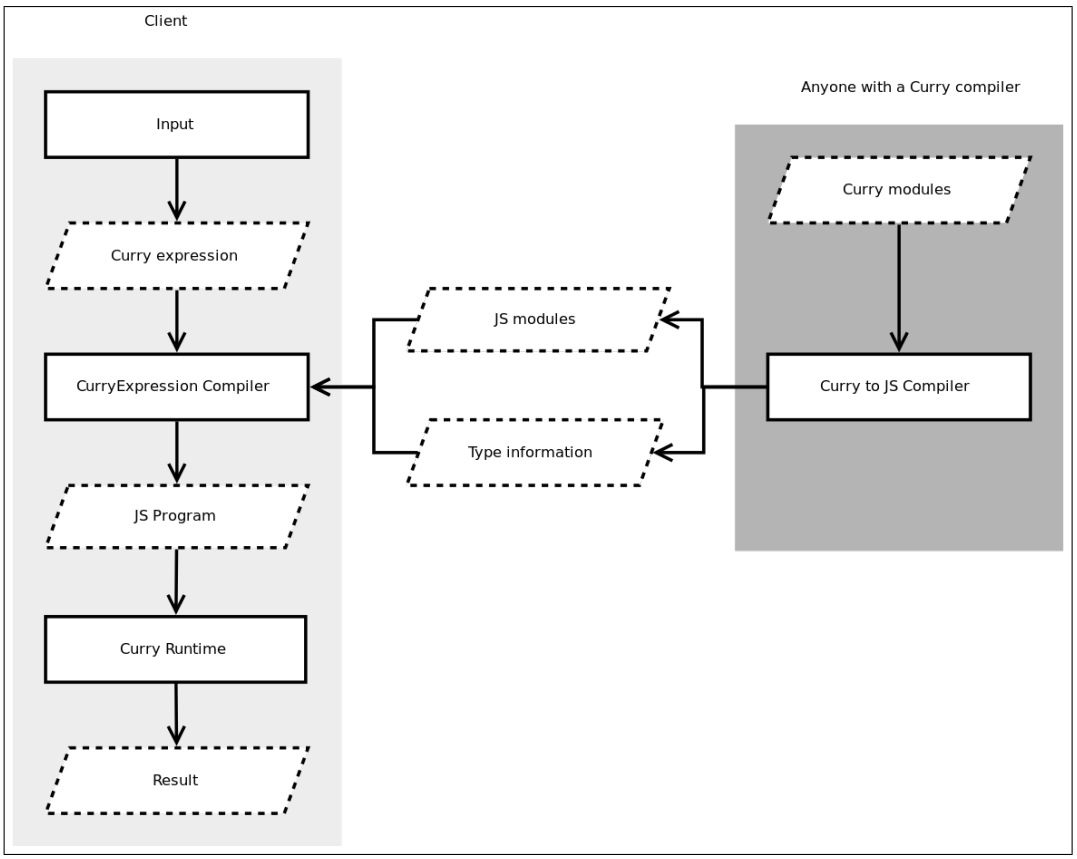


Abb. 4.15.: Die Architektur des idealen Übersetzers

handene Curry-Module mit einem in Curry geschriebenen Übersetzer in JavaScript-Module übersetzt. Zusätzlich werden vom Übersetzer zu den Curry-Modulen Typinformationen bereitgestellt. Möchte ein JavaScript-Client nun einen Curry-Term übersetzen, so ruft der Client einen in JavaScript ausführbaren Übersetzer auf, welcher den Curry-Term zusammen mit den übersetzten Curry-Modulen und den Typinformationen nach JavaScript übersetzt. Das übersetzte JavaScript wird dann mit einer in JavaScript ausführbaren Curry-Runtime ausgeführt. Die Übersetzung der Curry-Module nach JavaScript muss nur ein einziges Mal ausgeführt werden und kann dann beliebig vielen Clients zur Verfügung gestellt werden.

In Abbildung 4.16 ist die tatsächlich implementierte Architektur abgebildet. Dabei wird der Curry-Ausdruck an den Server gesendet und dort nach JavaScript übersetzt. Das übersetzte Programm wird dann an den Client geschickt, dieser kann das Programm mit seiner in JavaScript ausführbaren Runtime evaluieren. Dabei ist die

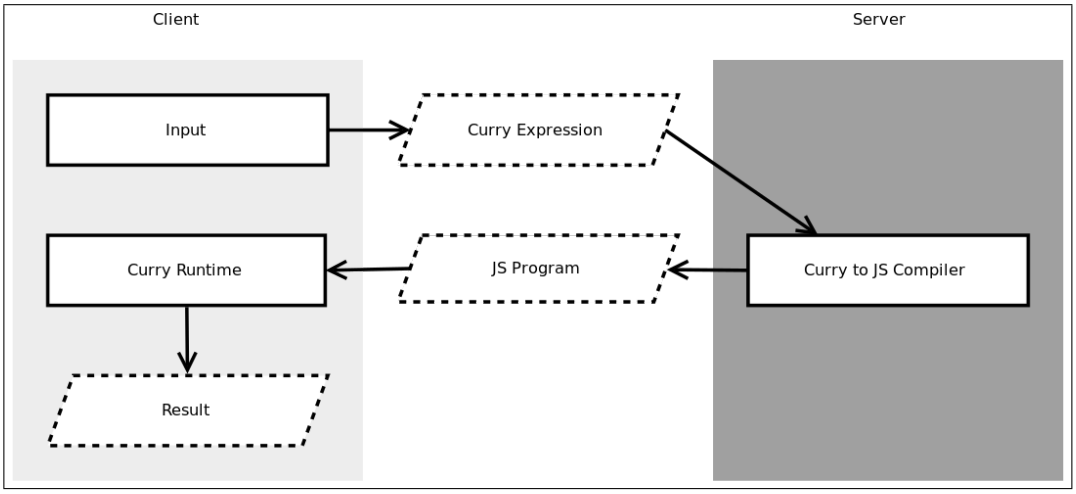


Abb. 4.16.: Die Architektur des implementierten Übersetzers

Curry-Runtime eine Implementierung der Semantik inklusive des Graphen. Im folgenden werden die einzelnen Komponenten beschrieben.

4.3 Komponenten

4.3.1 Runtime

Die Runtime dient dazu die Datenstruktur für den Graphen bereitzustellen, sowie die korrekte Auswertung des Graphen vorzunehmen.

Graphstruktur

Für die Repräsentation der ICurry-Ausdrücken wird ein gerichteter Graph benötigt. Dieser besteht aus Knoten und gerichteten Kanten. Der Quellknoten muss seine Zielknoten kennen, der Zielknoten muss jedoch keinerlei Informationen über Quellknoten haben. Dies ergibt sich aus der Semantik, liegt nämlich ein Knoten in der Kontrolle, so werden niemals die zum Knoten gerichteten Kanten benötigt. Eine Ausnahme bildet der aktuelle Auswertungspfad zum Knoten, dieser wird als Kontext mitgeführt und als Stack repräsentiert. Die Graphstruktur ist kein Baum, da Zyklen vorkommen können.

Die Knoten des Graphen wechseln aufgrund der Ersetzung die Beschriftung und die Kanten auf Zielknoten. Die Referenz auf einen Knoten muss jedoch konsistent bleiben. In der Implementierung des des Übersetzers Cam wird dies erreicht, indem sämtliche Attribute aus dem Knoten gelöscht werden und die Attribute danach neu gesetzt werden [Kir16, S. 39]. Dies führt dazu, dass keine eigene Knotenklasse für die Knotentypen verwendet wird, sondern eine einzige Klasse benutzt wird, die ein Typattribut besitzt. Sprite benutzt ein ähnlichen Ansatz und implementiert Knoten als Heap-Objekte, die Ersetzt werden können [AJ16, S. 5]. Dabei muss jedes Heap-Objekt jeden Knotentypen speichern können. Dies wird gewährleistet, indem in Knotentypen Pointer auf Zusatzinformationen gespeichert werden können.

In der Implementierung für diese Arbeit wurde ein anderer Ansatz gewählt. Ein Knoten besteht immer aus einem Container, welcher als JavaScript-Objekt referenziert werden kann und sich nicht ändert, und einem Inhalt, der die wechselnden Attribute des Knoten bereithält. In der momentanen Implementierung wird dieser Inhalt als JavaScript-Objekt bei jeder Änderung der Beschriftung neu instantiiert. Dieser Ansatz wurde gewählt, um eine sauberere und einfachere Implementierung zu ermöglichen. Allerdings werden hierdurch häufig neue Objekte alloziert und andere Objekte vom Garbage Collector eingesammelt.

Die Runtime kennt fünf Knotentypen. Diese sind *Konstruktorknoten*, *Funktionsknoten*, *Auswahlknoten*, *Fail-Knoten* und *Projektionsknoten*. Alle Knoten halten Informationen über ihren Typ und ihre ausgehenden Kanten. Konstruktorknoten halten zu-

sätzlich noch eine Referenz auf den Konstruktor. Funktionsknoten speichern eine Referenz auf die Funktion und ob ein Kind vor der Auswertung der Funktion eine Kopfnormalform haben muss. Auswahlknoten speichern zusätzlich die Choice-Identität. Neben diesen Attributen wurden verschiedene Hilfsfunktionen implementiert, die die Arbeit mit den Knoten erleichtern.

```
class FunctionNode extends Node {
  constructor(fun, children = [], childThatNeedsHNF = null) {
    super(children)
    this.childThatNeedsHNF = childThatNeedsHNF
    this.fun = fun
  }
}
```

Bsp. 4.2.: Der Konstruktor des Funktionsknoten

In Beispiel 4.2 ist der Konstruktor des Funktionsknoten zu sehen. Dabei ist zu erkennen, dass ein Funktionsknoten zusätzlich zu den Kindern auch Referenzen auf die Kinder und der Index des Kindes, welches eine Kopfnormalform benötigt, gesetzt werden. Mit diesen Informationen kann ein Funktionsknoten ausgewertet werden.

Die Schleife für die Graphersetzung

Die Semantik auf der im letzten Abschnitt beschriebenen Graphstruktur ist mit Hilfe einer Schleife implementiert. Die Schleife ruft immer wieder eine *step*-Funktion auf, bis diese `false` zurückgibt. In der *step*-Funktion findet ein Ersetzungsschritt im Sinne der Semantik statt. Dabei hängt der getätigte Ersetzungsschritt von verschiedenen Bedingungen ab, die sich aus der Semantik ergeben.

```
  if (currentNode.getType() === IdentityNode) {
    nodeStack.push({ node: currentNode.getChild(0), choices })
    return kontinue
  }
```

Bsp. 4.3.: Die Implementierung der Regel 17

In Beispiel 4.3 ist die Implementierung der Regel 17, der Dereferenzierung von Identitätsknoten, angegeben. Anders als in der Semantik wird das Kind des Identitätsknoten nicht in die Kontrolle sondern auf den Stack gelegt. Dies begründet sich darin, dass am Anfang der *step*-Funktion immer das erste Element vom Stack und somit dann erst in die Kontrolle geholt wird.

In Beispiel 4.4 ist die Implementierung der Regeln 1 und 2 dargestellt. Da sich kein Element auf dem Stack befindet, welches in der Kontrolle gelegt werden kann, ist die Anwendung von Regel 1 oder 2 notwendig. Ist auch die Queue leer, so ist die Regel 2 erreicht und der Interpreter fertig. Ansonsten wird der nächste Pfad aus der Queue geholt.

```
if (nodeStack.length === 0) {
      if (workQueue.length === 0) {
        return !kontinue
      }
      nodeStack.push(workQueue.pop())
      return kontinue
}
```

Bsp. 4.4.: Die Implementierung der Regeln 1 und 2

Für die Auswertung von Auswahlknoten wurde eine eigene Funktion implementiert. Diese trifft eine Auswahl, falls der Stack leer ist. Ansonsten führt sie einen Pull-Tab-Schritt durch.

Die Queue und die Speicherung der Auswahl

Für die Implementierung der Queue, auf der die verschiedenen Stacks für die Auswertungspfade abgelegt sind, wird ein einfaches Array verwendet. Zu jedem Stack werden die bisher in dem Auswertungspfad getroffenen Auswahlen gespeichert. Als Speicher wird ein `Map`-Objekt benutzt, welches eine Schlüssel-Wert-Zuordnung erlaubt. Die Identitäten des Auswahlknoten, welche bei der Erstellung automatisch als `Symbols` erzeugt werden, bilden dabei die Schlüssel. Als Werte werden die Indizes des Kindes, welches ausgewählt wurde, eingetragen.

Runtime Callbacks

Die Runtime wurde auf eine Weise implementiert, die es erlaubt nach jedem Auswertungsschritt eine Funktion ausführen zu lassen oder die Auswertung zu pausieren. Zusätzlich wurden zwei Module für die Visualisierung des Graphen programmiert. Das erste Modul gibt den momentane Graphen als Curry-Term aus. Das zweite Modul gibt die tatsächliche Struktur des Graphen visuell wieder, dabei erscheint der Graph wie in Abbildung 4.17 dargestellt. Bei der Ausgabe wird der Knoten, welcher momentan zur Kopfnormalform ausgewertet wird, markiert.

4.3.2 Übersetzer

Der Übersetzer ist dafür zuständig die im Curry-Programm definierten Datentypen und Funktionen in ein JavaScript-Programm zu übersetzen, welches mit Hilfe der Runtime ausgewertet werden kann.

In Abbildung 4.18 sind die Schritte abgebildet, welche eingesetzt wurden, um dieses Ziel zu erreichen. Zunächst wird Curry nach FlatCurry mit Hilfe des Curry-Frontends übersetzt. FlatCurry ist in Abschnitt 2.2.2 beschrieben. Zusätzlich wird das FlatCurry-

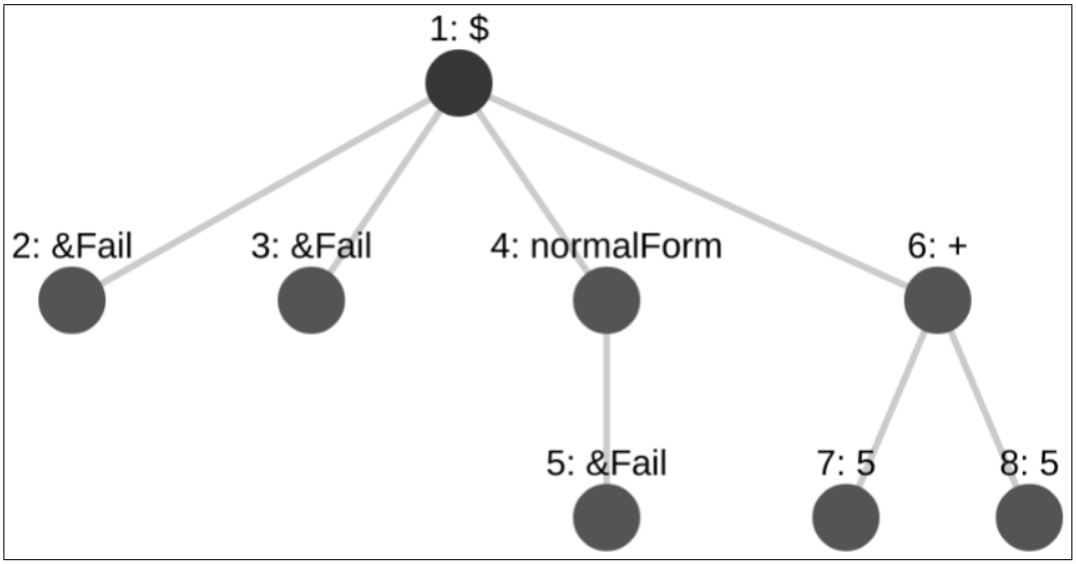


Abb. 4.17.: Die Darstellung des Graphen mit Hilfe des Visualisierungs-Moduls

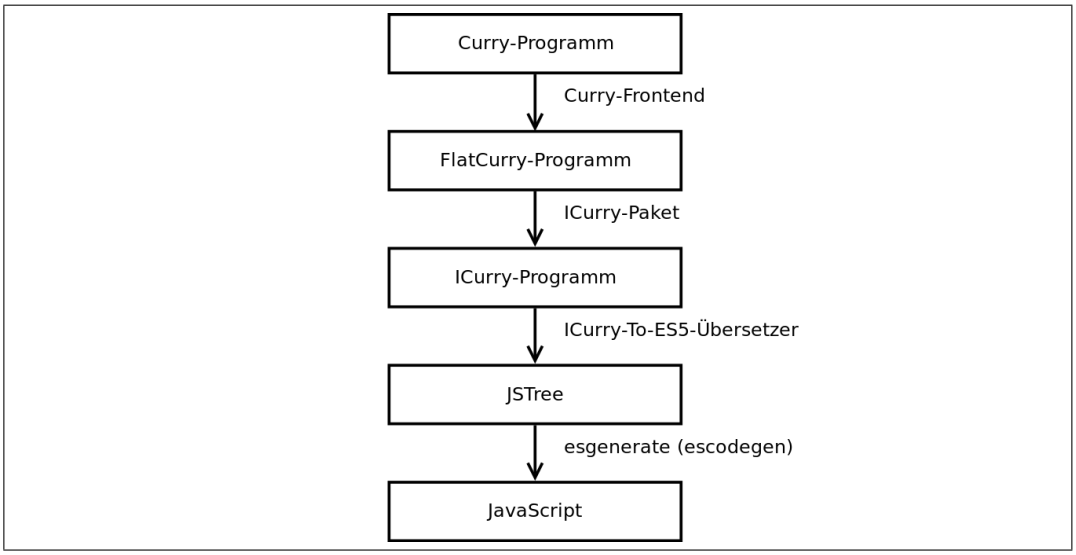


Abb. 4.18.: Die Übersetzer-Pipeline

Programm mit Typen annotiert. Dies geschieht in der Implementierung mit Hilfe der existierenden Module für Annotated-FlatCurry. Bei der Annotierung werden zunächst alle bekannten Konstruktoren und Funktionen mit neuen Typvariablen versehen. Dann werden durch Typinferenz Typgleichungen gebildet, die schließlich durch Substitution aufgelöst werden. Diese Typinformationen sind für die Übersetzung nach ICurry notwendig.

Die Übersetzung von FlatCurry mit Annotationen nach ICurry ist der nächste Schritt. Hierzu wird das *ICurry*-Paket, welches von Kirchmayr für den Übersetzer von Curry nach Java implementiert wurde, verwendet. ICurry ist in Abschnitt 2.2.3 beschrieben.

Nach diesem Schritt wird das ICurry-Programm in einen Abstrakten-Syntax-Baum, welcher ein JavaScript-Programm repräsentiert übersetzt. Im folgenden Abschnitt wird dieser Abstrakte-Syntax-Baum und der Übersetzungsprozess von ICurry in diesen Baum erläutert.

Die Übersetzung von ICurry in die JavaScript-Repräsentation

Um von einer Sprache in eine andere Sprache zu übersetzen, ist es sinnvoll, eine Repräsentation von Programmen in beiden Sprachen zu besitzen und somit von der konkreten Syntax zu abstrahieren. ICurry wird nur in Form einer solchen Repräsentation verwendet und diese existiert bereits.

Für JavaScript wurde eine neue Repräsentation in Curry implementiert, die für JavaScript nach der ECMAScript 5.1 Spezifikation vollständig ist. Diese Repräsentation baut auf der *estree*-Spezifikation auf.[4] Diese Spezifikation definiert einen Abstrakten-Syntax-Baum für ECMAScript 5 als JavaScript-Objekt. Die Spezifikation wurde am Anfang von der API des *SpiderMonkey*-Parsers abgeleitet. SpiderMonkey ist die JavaScript-Engine von Mozilla Firefox. Diese API gilt als die *lingua franca* für Werkzeuge, die sich mit dem JavaScript-Quelltext beschäftigen. Auf die Repräsentation nach der estree-Spezifikation setzen daher auch viele ausgereifte Programme aus, die zum Beispiel die Generierung von validem JavaScript-Quelltext ermöglichen.
Für diese Repräsentation von JavaScript in Curry wurde ein Hilfsmodul geschrieben, welches die Komposition des ASTs vereinfacht. Für die Generierung des Abstrakten Syntax Baums von JavaScript aus ICurry wurden fast ausschließlich diese Funktionen benutzt.

Die Übersetzung von ICurry nach JavaScript erfolgt dabei durch einen rekursiven Abstieg durch die Repräsentation von ICurry.

Dabei wird ein Curry-Modul in JavaScript immer als eine anonyme Funktion, welche einen Kontext erhält, dargestellt. In dieser Funktion wird als Attribut des Kontextes ein Objekt deklariert, in welchem sämtliche Funktionen und Konstruktoren als Attribute gesetzt werden. Schließlich wird diese anonyme Funktion auf `this` appliziert. Lädt man das Modul im Browser, so wird diese Applikation ausgeführt. Da `this` in diesem Fall der globale Kontext ist, wird also das Objekt, welches die Modul-Mitglieder enthält, als Attribut des globalen Kontextes gesetzt. Die Funktionen und Konstruktoren sind nur über dieses Objekt erreichbar. In node kann durch den Aufruf mit `require` ein ähnliches Ergebnis erreicht werden.

[4] Die Spezifikation ist unter `https://github.com/estree/estree` zu finden.

In Beispiel 2.7 wurde die Addition auf den Peano-Zahlen in Curry definiert. Diese Programm wird in Abbilding 4.5 als `Main`-Modul definiert.

```
module Main where
data Peano = Z | S Peano

add :: Peano → Peano → Peano
add Z     b = b
add (S a) b = S (add a b)
```

Bsp. 4.5.: Addition auf den Peano-Zahlen als `Main`-Modul

Bei der Übersetzung wird das Modul in einer anonymen JavaScript-Funktion gekapselt werden. Dies ist in Beispiel 4.6 dargestellt.

```
(function (context) {
  context.Main = {
      ...
  };
}(this));
```

Bsp. 4.6.: Von ICurry nach JavaScript übersetzter Modulbezeichner

Die Konstruktoren, welche in dem Curry-Programm definiert werden, werden als Attribute in dem Modul-Objekt gesetzt. Für jeden definierten Konstruktor wird dabei ein *Symbol* definiert. Symbole sind in JavaScript einzigartige Werte. Für unser Beispiel ist dies in Beispiel 4.7 dargestellt. Zu jedem Datentyp wurde außerdem

```
context.Main = {
  c_Z: Symbol('c_Z'),
  c_S: Symbol('c_S'),
   ...
};
```

Bsp. 4.7.: Von ICurry nach JavaScript übersetzte Konstruktoren

von ICurry eine Generatorfunktion eingeführt. Diese Generatorfunktionen konstruieren mit Hilfe von Auswahlen alle möglichen Werte des Datentyps. In Beispiel 4.8 ist die Generatorfunktion für den `Peano`-Datentyp dargestellt. Dabei konstruiert `new HurryRuntime.V` ein neues Objekt, welches von einem Knoten referenziert werden kann. In diesem Fall handelt es sich um eine Auswahl. Als Argumente werden dann die Kinder dieser Auswahl übergeben. `HurryRuntime.R.c(Main.c_Z, [])` konstruiert einen Konstruktorknoten mit der Beschriftung `Z` und ohne Kinder. Das zweite Argument der Auswahl konstruiert entsprechend einen Konstruktorknoten mit der Beschriftung `S`, der als Kind wieder die Generatorfunktion erhält.

Es bleibt noch die Generierung der Funktionsregeln. Da es in ICurry eine Funktionsregel pro Funktion gibt, wird für jede Funktionsregel ein Attribut in dem Modul-Objekt

```
context.Main = {
  g_Peano: function () {
    return new HurryRuntime.V(
      HurryRuntime.R.c(Main.c_Z, []),
      HurryRuntime.R.c(Main.c_S,
                       [HurryRuntime.R.f(Main.g_Peano, [])])
    );
  },
  ...
};
```

Bsp. 4.8.: Von ICurry nach JavaScript übersetzte Generatorfunktion

gesetzt. Der Wert dieses Attributs ist eine anonyme JavaScript-Funktion. Diese Funktion bekommt als Parameter durch die Runtime die Kinder des Knotens in der Kontrolle. In Beispiel 4.9 ist der erste Teil der übersetzten `add`-Funktion zu sehen. Der

```
context.Main = {
  f_add: function (child1, child2) {
    return new HurryRuntime.F(Main.h0_add, [
        child1,
        child2
    ], 0);
  },
  ...
};
```

Bsp. 4.9.: Von ICurry nach JavaScript übersetzte `add`-Funktion

einzige Zweck dieser Funktion ist es, denn Knoten so zu verändern, dass durch die Runtime als nächstes das erste Argument in Kopfnormalform gebracht wird. Die eigentliche Auswertung passiert in der Funktion `h0_add` welche als neue Beschriftung zurückgegeben wird.

```
h0_add: function (child1, child2) {
  switch (child1.getConstructor()) {
    case Main.c_Z:
      return new HurryRuntime.I(child2);
    case Main.c_S:
      var child3 = child1.getChild(0);
      return new HurryRuntime.C(Main.c_S,
        [HurryRuntime.R.f(Main.f_add, [child3, child2])]);
    default:
      return new HurryRuntime.E();
    }
}
```

Bsp. 4.10.: Von ICurry nach JavaScript übersetzte `add`-Hilfsfunktion

In Beispiel 4.10 ist diese Hilfsfunktion abgebildet. Wie in der Funktionsdefinition wird ein Pattern-Matching über den Konstruktor des ersten Arguments durchführt. Wird der Konstruktor `Z` gematcht, so würde eine Referenz zurückgegeben werden. In diesem Fall handelt es sich also um eine Projektion und es muss ein Identitätsknoten eingesetzt werden. Dieser wird mit `new HurryRuntime.I(child2)` konstruiert. In dem Fall, dass der Konstruktor `S` am Anfang des ersten Arguments steht, wird zunächst das Argument des Konstruktors als `child3` gesetzt. Dann wird ein Graph, der dem Ausdruck auf der rechten Seite der `add`-Funktion entspricht, konstruiert. Sollte das Pattern-Matching unvollständig sein, so wird am Ende ein Fallback-Fall angefügt, welcher als neues Objekt einen Fail-Knoten zurückgibt.

Mit Hilfe der `main`-Funktion kann nun die Auswertung des Programms durch die Runtime durchgeführt werden.

Im folgenden wird das Programm, welches die Übersetzung durchführt, formal beschrieben. Da es sich um ein funktionales Programm handelt, kann die Übersetzung als mathematische Funktion angegeben werden. Die Übersetzung ist dabei eine Funktion translate von der Menge der ICurry-Strukturen, welche in Abschnitt 2.2.3 eingeführt wurden, in die Menge der JavaScript-ASTs. Um die Ausführung zu vereinfachen, wird im folgenden eine Funktion betrachtet, deren Zielmenge JavaScript ist. Diese hat die Signatur

$$\text{translate} : \text{IProgs} \rightarrow \text{JavaScriptProgs}$$

mit der Menge aller ICurry-Programme IProgs und der Menge aller JavaScript-Programme JavaScriptProgs. Aus Gründen der Lesbarkeit wird auch die Umbennennung der Identifier weggelassen. Außerdem wird es erlaubt, mittels dem Ausdruck `${ ... }` mathematische Funktionen in das JavaScript Ergebnis einzubetten. Dies ist so zu verstehen, dass der Ausdruck `${f(x)}` durch das Ergebnis von $f(x)$ ersetzt wird.

In Beispiel 4.11 ist die Funktion dargestellt, welche die anonyme Funktion für die Moduldefinition in JavaScript erzeugt. Wie oben beschrieben erlaubt dies die Kapselung der Modulattribute. Die Modulattribute werden in einem Objekt mit dem Modulnamen generiert. In der Beispiel 4.12 ist die Übersetzung der Datentypen dargestellt. Dabei spielen die Namen der Datentypen keine Rolle mehr. Für jeden Konstruktor wird ein Attribut angelegt, welches als Wert ein einzigartiges Symbol mit dem Konstruktornamen erhält.
Da die `<`-Funktion auf der Definitionsreihenfolge der Konstruktoren aufbaut, ist diese Lösung problematisch. Es wäre besser, ganzzahlige Nummern zu vergeben, was

translate(modname, {$\overline{\text{datatype}_n}$}, {$\overline{\text{func}_m}$}) =
```
(function (context) {
  context.${modname} = {
    ${translateDatatype(datatype_0)}
    ...
    ${translateDatatype(datatype_n)}

    ${translateFunction(modname, func_0)}
    ...
    ${translateFunction(modname, func_m)}
  };
}(this));
```

Bsp. 4.11.: Die translate-Funktion

translateDatatype(dataname, {$\overline{(\text{combname}_n, \text{arity}_n, \text{oriname}_n)}$} =
```
  ${combname_0}: Symbol(${combname_0})
  ...
  ${combname_n}: Symbol(${combname_n})
```

Bsp. 4.12.: Die translateDatatype-Funktion

allerdings zu einem größeren Umbau der Implementierung führen würde, da für die Ausgabe der Konstruktornamen dann auch der Modulname im Knoten gespeichert werden müsste und das Modul ein Mapping der Konstruktornummern auf Konstruktornamen bereithalten müsste.

translateFunction(modulename, (combname, arity, typeexpr, irule)) =
  translateRule(modulename, combname, irule)

translateFunction(modulename, (combname, arity, irule)) =
  translateRule(modulename, combname, irule)

Bsp. 4.13.: Die translateFunction-Funktion

In Beispiel 4.13 wird die Übersetzung der IFunc aus dem ICurry-Repräsentation definiert. Da es in JavaScript keine Zugriffsmodifikatoren gibt, wird nicht zwischen den exportierten und auf das Modul beschränkten Funktionen unterschieden.

In Beispiel 4.14 ist die Übersetzung der einzelnen IRule-Typen dargestellt. Für externe Regeln werden keine Attribute in dem Modulobjekt erzeugt. Diese Attribute müssen in der Programmdatei des externen Moduls dem Modulobjekt hinzugefügt werden.
Eine einfache Regel aus ICurry wird als eine Funktion in JavaScript übersetzt. Die Funktion bekommt die in der Repräsentation definierten Argumente, definiert im

translateRule(modulename, combname, (arity, externname))) = *empty*

translateRule(modulename, combname, ($\{\overline{\text{iparam}_n}\}$, $\{\overline{\text{ilocal}_m}\}$, iexpr)) =

```
${combname}: function (${iparam_0, ..., iparam_n}) {
  ${translateLocals({ilocal_m})}
  return ${translateReturnExpr(iexpr)}
}
```

translateRule(modulename, combname, (newname, $\{\overline{\text{iparam}_n}\}$, $\{\overline{\text{ilocal}_m}\}$, $\{\overline{\text{newparam}_p}\}$, hnfparam, ?literal, $\{\overline{\text{ibranch}_q}\}$)) =

```
${combname}: function (${iparam_0, ..., iparam_n}) {
  ${translateLocals({ilocal_m})}
  return new FunctionNode(${modulname}.${newname},
                          [${newparam_0, ..., newparam_p}],
                          ${hnfparam})
}

${newname}: function (${newparam_0, ..., newparam_p}) {
  switch(${hnfparam}.getConstructor()) {
    ${translateBranch(hnfparam, ibranch_0)
    ...
    ${translateBranch(hnfparam, ibranch_q)
    default:
      return new NodeReference(new FailNode())
  }
}
```

Bsp. 4.14.: Die translateRule-Funktion

Funktionsrumpf zunächst die lokalen Deklarationen und gibt dann einen neuen Inhalt für einen Knoten zurück. Nach der Auswertung wird dieser Inhalt von dem Knoten referenziert werden, welcher vorher mit dieser Funktion beschriftet war.
Als dritte Möglichkeit kann es sich um eine IRule mit einem Case handeln. In diesem Fall werden zwei Funktionen generiert. In der ersten Funktion werden die lokalen Deklarationen definiert und schließlich ein neuer Funktionsknoteninhalt erstellt, welcher mit der zweiten Funktion beschriftet ist. Dem Funktionsknoteninhalt werden die Parameter für die zweite Funktion als Kinder angehängt und das Kind, welches im case-Ausdruck vorkommt und somit in eine Kopfnormalform gebracht werden muss, wird bei der Konstruktion ebenfalls übergeben.
Die zweite Funktion erhält die Parameter für die zweite Funktion und führt im Rumpf ein `switch`-Statement über den Konstruktor des Kindes aus, über das gematcht wird. Die einzelnen Branches werden mittels der in Beispiel 4.14 auf Seite 51 dargestellten Funktion übersetzt. Ein Branch besteht aus einem qualifizierten Konstruktornamen, einer Liste von in dem Pattern gebundenen Variablen, lokalen Deklarationen und einem Ausdruck. Bei der Übersetzung wird der qualifizierte Konstruktorname als Referenz verwendet, um auf das einzigartige Symbol zuzugreifen

und dieses mit dem Symbol zu vergleichen, über welches gematcht wird. Liegt eine Gleichheit vor, so werden zunächst neue Variablen für die Bindung im Pattern eingeführt. Dabei sind die Pattern-Variablen genau die Kinder des Konstruktorknoten über den gematcht wird.
Im Branch werden die übersetzten lokalen Deklarationen und der übersetzte Ausdruck eingefügt. Die Übersetzung der lokalen Deklarationen ist in Beispiel 4.16 dargestellt. Auffällig ist dabei, dass zunächst alle Variablen mit leeren Objekten instantiiert werden und erst danach mittels `Object.assign` die passenden Ausdrücke gesetzt werden. Der Grund hierfür liegt darin, dass die lokalen Deklarationen sich gegenseitig referenzieren können. So ist zum Beispiel

```
main = let { a = b; b = 1:a } in head a
```

ein valides Programm.

---

translateBranch(hnfparam, (patternname, $\{\overline{\text{patparam}_n}\}$, $\{\overline{\text{ilocal}_m}\}$, iexpr)) =

```
  case ${patternname}:
    var ${patparam_0} = ${hnfparam}.getChild(0)
    ...
    var ${patparam_n} = ${hnfparam}.getChild(n)
    ${translateLocals({ilocal_m})}
    return ${translateReturnExpr(iexpr)}
```

Bsp. 4.15.: Die translateBranch-Funktion

In der Funktion translateAssign wird die unterschiedliche Übersetzung von normalen Ausdrücken und Projektionen deutlich. Bei einer Projektion wird ein Identitätsknoten erstellt.

---

translateLocals($\{\overline{(\text{param}_n, \text{iexpr}_n)}\}$) =

```
  var ${param_0} = {}
  ...
  var ${param_n} = {}

  Object.assign(${param_0}, ${translateAssign(iexpr_0)})
  ...
  Object.assign(${param_n}, ${translateAssign(iexpr_n)})
```

translateAssign(ref) = `new NodeReference(new IdentityNode(ref))`

translateAssign(exp) = translateExpression(exp)

Bsp. 4.16.: Die translateLocals-Funktion

In Abbildung 4.17 ist schließlich die Übersetzung von Ausdrücken definiert. Die für die Übersetzung von Ausdrücken benutzt Funktion translateReturnExpr ist dabei dafür zuständig, in dem Fall, dass der Ausdruck eine Referenz ist, einen Identitätsknoteninhalt zu erstellen.
Die Funktion translateExpression ist notwendig, um zwischen Ausdrücken, welche

---

translateReturnExpr(ref) = `new IdentityNode(${ref})`
translateReturnExpr(exp) = translateExp(exp)

translateExpression(ref) = `${ref}`
translateExpression(exp) = `new NodeReference(${translateExp(exp)})`

translateExp(param) = `${param}`

translateExp(literal) = `new ConstructorNode(${literal})`

translateExp((gen, genparam)) = `${genparam}`

translateExp(fCall, name, $\{\overline{\text{exprs}_n}\}$) =
```
  new FunctionNode(${name},
                   ${translateExpression(exprs_0)},
                   ...
                   ${translateExpression(exprs_n)})
```

translateExp(cCall, name, $\{\overline{\text{exprs}_n}\}$) =
```
  new ConstructorNode(${name},
                      ${translateExpression(exprs_0)},
                      ...
                      ${translateExpression(exprs_n)})
```

translateExp(or, $\{\overline{\text{exprs}_{n...m,\ n<m}}\}$) =
```
  new ChoiceNode(${translateExpression(exprs_n)},
                 ${translateExpression(or, {exprs_(n+1)...m})})
```
translateExp(or, $\{\overline{\text{exprs}_{n...n}}\}$) =
`${translateExpression(exprs_n)}`

Bsp. 4.17.: Die translateExpression-Funktion

einen Knoten und Ausdrücke welche einen Knoteninhalt zurückgeben, zu unterscheiden.
Schließlich ist die Funktion translateExp für die eigentliche Übersetzung der Ausdrücke zuständig. Enthält die ICurry-Repräsentation des Ausdrucks nur eine Variable, so wird diese einfach eingefügt, da es sich um eine Referenz auf einen Knoten handelt. Ist es ein Literal, dass heißt ein primitiver Wert, so wird ein Konstruktorknoteninhalt erstellt und mit dem Literal beschriftet. Handelt es sich um eine Generatorvariable, so wird, im Gegensatz zu dem Vorgehen des Übersetzer Cam, einfach die Variable zurückgegeben. Generatorvariablen treten auf, wenn eine Funktion ein Typparameter

besitzt. Der Funktion werden zusätzliche Parameter hinzugefügt, über die Generatoren übergeben werden können, wenn bei der Applikation der Typ feststeht. In Cam wird in dem Fall, dass eine solche Generatorvariable in dem Ausdruck vorkommt, der gesamte Graph ab dem von der Variable referenzierten Knoten kopiert und die Referenz auf diesen neuen Graph benutzt [Kir16, S. 48]. Dadurch wird die im Generator getroffen Auswahl nicht mehr geteilt. Hintergrund hinter diesem Verfahren ist die für Cam gewählte Semantik. Dies wird in der Arbeit von *Kirchmayr* an dem Beispiel, welches in Beispiel 4.18 in leicht veränderter Form dargestellt wird, deutlich gemacht [Kir16, S. 28]. In dem Beispiel wurden die Typen angepasst, da sie in der Originalfassung inkorrekt sind. Nach der Idee von *Kirchmayr* würden alle Ergebnisse

```
f :: (a,a)
f = (x, x)
  where x free

test = f :: (Bool, Bool)
```

Bsp. 4.18.: Sharing einer freien Variable

von `test` hier

```
(True, True)
(True, False)
(False, True)
(False, False)
```

lauten [Kir16, S. 28]. Dieser Semantik wird nicht gefolgt. Im Curry Report wird spezifiziert, dass „mehrfache Vorkommen einer Variable immer geteilt werden“[5] [He16, S. 78]. Entsprechend werden in dem Beispiel nur die Ergebnisse

```
(True, True)
(False, False)
```

als korrekt betrachtet. Hieraus ergibt sich, dass Generatorvariablen wie normale Variablen behandelt werden können.

Repräsentationen von Funktionsaufrufen werden durch die einen Ausdruck für die Konstruktion neuer Funktionsknoten übersetzt. Der Funktionsknoten erhält als Beschriftung den Funktionsnamen, in JavaScript kann hier die Referenz auf die Funktion eingesetzt werden, da Funktionen Objekte sind. Die Kinder des Funktionsknoten werden aus den Ausdrücken konstruiert, welche der Funktion übergeben werden.
Die Repräsentation der Konstruktorenapplikation wird analog zu den Funktionsauf-

[5] Originaltext: „[We only note here] that several occurrences of the same variable are always shared [, i.e. ...]“

rufen übersetzt. Einziger Unterschied ist, dass ein Konstruktorknoten statt einem Funktionsknoten benutzt wird.

Die Repräsentation von einem Ausdruck, der mit einer Auswahl beginnt, wird mit Hilfe eines Auswahlknoten übersetzt. Da Auswahlen in ICurry beliebig Möglichkeiten beinhalten können, Auswahlknoten der Runtime jedoch nur zwei Möglichkeiten beinhalten, muss die Auswahl transformiert werden. Besitzt die Auswahl mehr als eine Möglichkeiten, so wird ein Auswahlknoten erstellt, welcher als erstes Kind die erste Möglichkeit hat. Als zweites Kind wird die Übersetzung des Auswahlausdrucks ohne die erste Möglichkeit gesetzt. Besteht irgendwann nur noch eine Möglichkeit, so wird die Auswahl ignoriert und einfach die Möglichkeit übersetzt.

4.3.3 Webservice

Der Webservice bündelt die Runtime und den Übersetzer und stellt ein Webinterface zur Verfügung. Dabei wird auf dem Client die Runtime geladen und ein Eingabefeld zur Verfügung gestellt. Wird das Eingabefeld abgeschickt, so wird der Ausdruck an einen Webserver gesendet. Dieser Webserver ruft ein CGI-Skript auf, welches in Curry geschrieben ist und letztlich den Übersetzer aufruft. Das Resultat der Übersetzer wird dann als JavaScript-Code wieder an den Client gesendet und von diesem ausgeführt. In diesem Moment lädt also der Client den übersetzten Ausdruck. Der Client führt dann mit der Runtime den übersetzten Ausdruck aus.

In den Webservice wurde auch die Bibliothek zur Visualisierung der Auswertung eingebunden. Mit Hilfe des Callbacks der Runtime erlaubt dies, schrittweise die Auswertung eines Terms zu verfolgen.

Da bei dem gesamten Vorgang einige Konfigurationsoptionen gesetzt werden müssen, wurde dem Webservice bei der Implementierung ein Dockerfile beigelegt. Ein Dockerfile ist eine Bauanweisung für ein Docker-Image. Docker ist eine Software, welche das Bauen, Orchestrieren und Ausführen von Containern erlaubt. Ein Container basiert dabei immer auf einem Image, welches nach den Anweisung in einem Dockerfile gebaut wird. Ist ein Image einmal erstellt, so kann es auf beliebigen Systemen, welche Docker zur Verfügung stellen, ausgeführt werden. Für den Webservice wurde das Image auf dem Docker-eigenen Repository veröffentlicht und muss daher nicht mehr gebaut werden. Jeder kann das Image nach dem herunterladen einfach ausführen.

Auf diese Weise ist es möglich, den Webservice auszuführen, ohne den Webserver, Curry-Übersetzer und -Paketmanager, CGI-Handler oder ähnliches installieren zu müssen.

5 Evaluation

5.1 Vollständigkeit der Curry-Implementierung

Als Grundlage für die Übersetzung und Ausführung wurde die in dieser Arbeit spezifizierte Semantik für ICurry implementiert.

Die Semantik für ICurry enthält den funktionalen und logischen Teil von Curry, allerdings ohne Constraints. Das effiziente Lösen von Constraints ist ein wichtiger Bestandteil logischer Programmierung [Alb+05, S. 21], der allerdings im Rahmen dieser Arbeit nicht behandelt werden konnte. Constraints sind auch kein Teil der in dieser Arbeit spezifizierten Semantik und wurden daher nicht bei der Implementierung beachtet. Da keine Constraints benutzt werden können, ist auch keine Implementierung der Unifikation notwendig.

Weiterhin wurden auch keine Input-/Output-Funktionen implementiert, da diese für die grundlegende Funktion des Übersetzers nicht notwendig waren.

Externe Funktionen, dass heißt, Funktionen welche in der Zielsprache implementiert werden, wurden in der Semantik nicht spezifiziert. Sie werden allerdings für die Implementierung der Prelude eingesetzt und es ist grundsätzlich möglich, sie zu benutzen.

Die Prelude ist nicht vollständig implementiert. Neben den Constraints und den Input-/Output-Funktionen, fehlt auch die `show`-Funktion und die Vergleichsfunktionen `<` und alle Funktionen, die auf diese Funktion aufbauen.

Dabei ist die `<`-Funktion ist auf allen Datentypen anwendbar. `a < b == true` gilt, falls der Konstruktor von `a` vor dem Konstruktor von `b` definiert wird. Durch die Verwendung von Zahlen statt Symbolen als Konstruktoridentifikatoren wäre eine Implementierung der `<`-Funktion trivial. Die für die Konstruktoren verwendeten Zahlen müssten pro Konstruktor eines Datentyps einzigartig sein, eine darüber hinausgehende Einzigartigkeit ist nicht notwendig, da die Typen bei der Kompilierung überprüft werden.

Implementiert wurde der Operator zur strikten Auswertung, da die Endergebnisse der Auswertung eines Curry-Terms in Grundnormalform angegeben werden und diese Operation dafür notwendig ist. Außerdem wurden die Boolschen Operationen und die Arithmetik auf den ganzen Zahlen implementiert. Zur Verfügung steht auch der Operator für die nicht-deterministische Auswahl.

Für Listen wurde keine native Implementierung in JavaScript benutzt, sondern eine naive Implementierung in Curry. Das heißt Listen werden nicht als effizientes JavaScript-Array implementiert, sondern als eine Verkettung von Knoten im Graphen. Das gleiche gilt für Strings. Strings wurden nicht als JavaScript-Strings implementiert, sondern als eine Verkettung von Knoten, welche jeweils ein Character enthalten. Dies führt zu einer sehr ineffizienten Implementierung in Bezug auf diese Datentypen und auf eine in der Praxis sehr ineffizienten Curry-Implementierung, da Strings und Listen sehr häufig benutzt werden.

5.2 Benchmarks

Um die Effizient des Übersetzers von Curry nach JavaScript zu evaluieren, werden einige Tests aus der Arbeit [Bra+11] mit dem Übersetzer durchgeführt. Die Tests werden auch mit den ausgereiften Curry-Übersetzern KiC2 und PAKCS ausgeführt.

Das Testsystem verfügt über ein Intel(R) Core(TM) i5-5300U mit einem 64-bit Befehlssatz, zwei Kernen und einer Taktrate von 2.30GHz. Es sind 8 GB RAM installiert. Als Betriebssystem läuft Debian mit der Linux Kernel Version 4.9.0-3-amd64.

KiCS2 ist in der Version 0.5.1 vom 10.04.2017 installiert. PAKCS ist in der Version 1.14.2 vom 24.02.2017 installiert und benutzt SWI-Prolog in der Version 7.2.3.

Die Runtime wird in der Version der Git-Revision `f9edc264` und der Übersetzer in der Version der Git-Revision `1260774c` verwendet. Der JavaScript-Zielcode wird mit Chrome 60.0.3112.90 ausgeführt. In Chrome läuft die V8 JavaScript-Engine in der Version 6.0.286.52. Die Zeit der Ausführung in Chrome wird mit Hilfe der `console.time` und der `console.timeEnd` Funktionen gemessen.

Der Curry nach JavaScript Übersetzer wird im folgenden mit *hurry* bezeichnet. Alle Programme wurden auf allen Übersetzern fünfmal ausgeführt. In den folgenden Tabellen ist der Median dieser Ergebnisse in Sekunden angegeben.

Es ist aus Tabelle 5.1 ersichtlich, dass hurry um etwa das 1000-fache langsamer ist als KiCS2 und etwa um das 20-fache langsamer als PAKCS bei der Ausführung

	ReverseUser	TakPeano (goal0)
KiCS2	0,27	0,22
PAKCS	8,51	73,62
hurry	182,26	out of time

Tab. 5.1.: Benchmark von funktionalen Programmen

von ReverseUser. Bei der Ausführung von TakPeano terminiert Hurry nicht vor dem Ablauf von fünf Minuten. Bei kleineren Werten in TakPeano terminiert Hurry.

Um zu verstehen, warum Hurry bei der Ausführung dieser Programme langsam ist, wurde Performance Monitoring aus den Google Chrome Development Tools verwendet. In Abbildung 5.1 ist der Heap während der Ausführung von TakPeano darge-

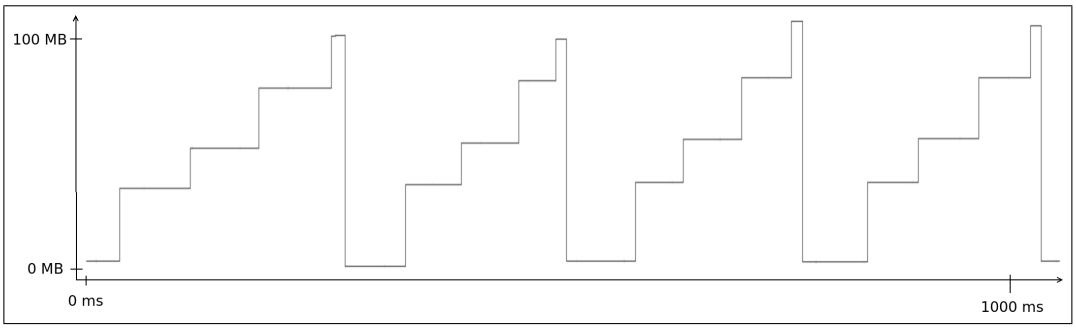


Abb. 5.1.: Der Heap während der Ausführung von TakPeano

stellt. In diesem Fall ist ein Ausschnitt von einer Sekunde dargestellt, dass Pattern wiederholt sich jedoch über die gesamte Zeit der Ausführung. Zu sehen ist, dass der Heap wächst, bis der Garbage Collector einsetzt und überflüssige Objekte entfernt. Es ist offensichtlich, dass die Anzahl der benötigten Objekte auf dem Heap ungefähr konstant bleibt.

Um ein genaueres Profiling zu ermöglichen, wurde TakPeano mit kleineren Zahlen ausgeführt. Hierdurch kann die Zeit pro Aktivität über die gesamte Ausführung betrachtet werden, diese ist in Tabelle 5.2 dargestellt. In Chromes V8 Engine gibt es

Aktivität	Benötigte Zeit	Anteil
JavaScript	1243 ms	69 %
Minor Garbage Collection	488 ms	27 %
Major Garbage Collection	71 ms	4 %

Tab. 5.2.: Anteile der Garbage Collection an der Ausführungsdauer

zwei Arten von Garbage Collection: Minor Garbage Collection und Major Garbage Collection. Wenn ein neues Objekt erstellt wird, so wird dieses zunächst in einem speziellen Bereich alloziert. Dieser Bereich ist relativ klein, normalerweise liegt seine Größe zwischen einem und acht Megabyte. Die Allokation in diesem Bereich ist sehr schnell. Die Minor Garbage Collection setzt ein, wenn dieser Bereich voll ist. Dabei werden nur Objekt in dem Bereich überprüft und eingesammelt. Wenn Objekte zwei Ausführung des Minor Garbage Collectors überstehen, werden sie nicht

mehr durch den Minor Garbage Collector überprüft. Solche Objekte werden dann durch den Major Garbage Collector überprüft und eingesammelt, welcher deutlich seltener ausgeführt wird.

Das Resultat zeigt, dass das Programm etwa ein Drittel der Ausführung mit der Garbage Collection verbringt. Dies ist ein sehr hoher Anteil, der aus der gewählten Architektur des Graphen bei der Implementierung folgt. Da für jeden neuen Knoten zwei Objekte angelegt werden müssen, die durch eine Referenz verknüpft sind, ist die Überprüfung und das Einsammeln dieser Objekte sehr wahrscheinlich der Auslöser für den hohen Anteil des Garbage Collectors an der Ausführungsdauer.

Mit Google Chrome kann auch die Ausführungsdauer der aggregierten Funktionsaufrufe, dass heißt der Zeit die insgesamt bei allen Aufrufen in einer Funktion verbracht wird, ausgezeichnet werden. Dabei lassen sich die Funktionsaufrufe nach der Zeit, die sie ohne Aufruf anderer Funktionen benötigen, sortieren. In Tabelle 5.3 sind die Funktionen mit der größten eigenen, aggregrierten Ausführungsdauer dargestellt. Es wird die gesamte Ausführungsdauer mit allen aus der Funktion erfolgten Aufrufen, die eigene Ausführungsdauer, ohne die aus der Funktion erfolgten Aufrufe, und der Anteil der eigenen Ausführungsdauer an der gesamten Programmausführungsdauer aufgeführt. Die Funktion `makeFunction` verbraucht dabei für die eigene Aus-

Funktion	Ausführungsdauer	Eig. Ausführungsdauer	Eig. Anteil
`makeFunction`	594 ms	338 ms	18,7 %
`FunctionNode`	126 ms	103 ms	7,0 %
`map`	170 ms	86 ms	4,7 %
`h1_leq`	173 ms	78 ms	4,3 %

Tab. 5.3.: Funktionen mit dem größten Anteil an der Ausführungsdauer

führung insgesamt die meiste Zeit. `makeFunction` ist die Funktion, welche genutzt wird um ein neuen Funktionsknoten inklusive Referenz zu erstellen. Dies passiert in TakPeano sehr häufig. Eine Optimierung an dieser Stelle ist nicht nur für die Ausführungsdauer der Funktion, sonder auch, wie oben beschrieben, für die Garbage Collection sinnvoll.

Die Funktion, welche bei dem aggregierten Zeitverbrauch an zweiter Stelle steht, ist die `FunctionNode`-Funktion. Diese Konstruktionsfunktion wird verwendet, um einen neuen Funktionsknoten ohne Referenz zu erstellen.

Als dritte Funktion folgt die `map`-Funktion. Diese wird von verschiedenen Funktionen aufgerufen, allerdings wird fast 100% der durch die `map`-Funktion verbrauchten Zeit durch Aufrufe aus der Funktion `evaluateFunction` in `FunctionNode` verbraucht. In der `evaluateFunction`-Funktion wird `map` eingesetzt, um die Identitäten aufzulösen, wie es in der Regel 13 der Semantik definiert ist (siehe Abbildung 4.9). Das diese

einfache Dereferenzierung fast 9 % der gesamten Programmausführung ausmacht, ist unerwartet.

Als letzte Funktion ist `h1_leq` aufgeführt. Diese Funktion wurde aus der `leq`-Funktion im Curry-Programmcode von TakPeano generiert. Bei der Übersetzung wurde die `leq`-Funktion in vier Funktionen aufgeteilt, da das Pattern von zwei Parametern überprüft werden muss. Dabei ist in `h1_leq` ist dabei die Funktion, welche das zweite Pattern überprüft, nachdem das zweite Parameter ist Kopfnormalform vorliegt. Da diese Überprüfung sehr häufig in dem Programm vorkommt, war eine hohe aggregierte Ausführungszeit hier zu erwarten.

In der Ausführung funktionaler Programme hat der von *Kirchmayr* implementiert Übersetzer von Curry nach Java deutlich bessere Ergebnisse erzielt. Dies hängt sehr wahrscheinlich mit der Wahl der Strategie zur Knotenerstellung und Umwandlung im Graphen zusammen. Eine höhere Effizient könnte daher in diesem Übersetzer erlangt werden, wenn keine Container für die Knoten verwendet werden würde und dafür eine Umwandlung von Knoten eines Typs in einen anderen Typ zugelassen werden würde. Dies wäre im Rahmen der momentanen Umsetzung möglich, eine saubere Implementierung ist nach der momentanen Strategie allerdings einfacher und wurde auch aus diesem Grund gewählt.

Eine weitere Steigerung der Effektivität kann durch eine Optimierung erreicht werden, die auch im Basic Scheme Anwendung findet [AP12, S. 23]. Dabei wird mit der Auswertung fortgefahren, ohne in die `step`-Funktion zurück zu springen, solange der momentan auszuwertende Knoten weiterhin mit einem Funktionssymbol beschriftet ist. Aus den Performance-Aufzeichnung ist nicht zu erkennen, wie groß die Vorteile hieraus wären.

Eine Evaluation der nicht-deterministischen Ausführung wurde nicht vorgenommen, es können aber ähnliche Ergebnisse wie für den funktionalen Teil erwartet werden.

6 Ausblick

In dieser Arbeit wurden die Grundlagen, auf denen die Curry nach JavaScript Übersetzung basiert, erläutert. Im Weiteren wurde die Umsetzung der direkten Übersetzung von Curry-Programmen nach JavaScript beschrieben. Hierfür wurde eine Semantik auf ICurry-Programmen definiert und diese implementiert. Es hat sich herausgestellt, dass die Übersetzung von Curry-Programmen in und die Ausführung mit einer Graphstruktur eine funktionsfähige Lösung ist und als Grundlage für die Übersetzung in verschiedene imperative Sprachen eine solide Basis darstellt.

Es hat sich auch gezeigt, dass die Effektivität der Implementierung stark von den in der Sprache vorhandenen Konstrukten und der gewählten Architektur abhängig ist. Es wird sicher mindestens eine weitere vollständige Überarbeitung des Übersetzers nötig sein, um eine von der Leistung konkurrenzfähige Version zu erhalten. Dabei muss vermutlich vom softwaretechnischen Optimum abgewichen werden, was die Wartbarkeit erschwert kann. Weiterhin sind Kenntnisse über Stärken und Schwächen in der Performance der Zielsprache notwendig.

Als Resultat der Umsetzung existiert nun ein Übersetzer, welcher es erlaubt, Curry (ohne Constraints) nach JavaScript zu übersetzen und im Browser auszuführen. Dies erlaubt die Einbindung in Webseiten, ohne dass der Server explizit IO-Aktionen verbieten oder die Ausführung in eine Sandbox verlagern muss.

Das Idealziel, den gesamten Übersetzer in JavaScript ausführen zu können, wurde nicht erreicht. Da die Übersetzung des gesamten Curry-Frontends nach JavaScript noch nicht möglich ist, und das daraus entstehende Programm vermutlich für Web-Anwendungen zu schwergewichtig wäre.

Als Nebenprodukt ist eine Visualisierung für die Auswertung von Curry-Programmen entstanden, welche es erlaubt, einzelne Schritte genau nachzuvollziehen.

Es gibt verschiedene Felder in denen die Implementierung weiterentwickelt werden kann. Zunächst ist die Vervollständigung der Curry-Prelude ein wichtiger Aspekt, der die konkreten Einsatzmöglichkeiten des Compilers erhöhen würde. Das Aufstellen von Regeln für die Lösung von Constraints und deren Implementierung in der Runtime ist ein weiterer Punkt, der für die Vollständigkeit sorgen würde.

Auch die Effektivität kann mit wenig Anpassungsmaßnahmen vermutlich stark erhöht werden. Eine Implementierung, die dem Idealziel näher kommen würde, wäre auch ein Feld mit viel Potenzial.

Literatur

[AJ14] Sergio Antoy und Andy Jost. „Compiling a Functional Logic Language: The Fair Scheme“. In: *Logic-Based Program Synthesis and Transformation: 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*. Hrsg. von Gopal Gupta und Ricardo Peña. Cham: Springer International Publishing, 2014, S. 202–219 (siehe S. 2, 21).

[AJ16] Sergio Antoy und Andy Jost. „A New Functional-Logic Compiler for Curry: Sprite“. In: *CoRR* abs/1608.04016 (2016) (siehe S. 26, 42).

[Alb+05] E. Albert, M. Hanus, F. Huch, J. Oliver und G. Vidal. „Operational Semantics for Declarative Multi-Paradigm Languages“. In: *Journal of Symbolic Computation* 40.1 (2005), S. 795–829 (siehe S. 16, 29, 57).

[Ant11] Sergio Antoy. „On the Correctness of Pull-Tabbing“. In: *CoRR* abs/1108.0190 (2011) (siehe S. 37).

[AP12] Sergio Antoy und Arthur Peters. „Compiling a Functional Logic Language: The Basic Scheme“. In: *Functional and Logic Programming: 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*. Hrsg. von Tom Schrijvers und Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 17–31 (siehe S. 21, 29, 61).

[Bac78] John Backus. „Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs“. In: *Commun. ACM* 21.8 (1978), S. 613–641 (siehe S. 1).

[Bra+11] Bernd Braßel, Michael Hanus, Björn Peemöller und Fabian Reck. „KiCS2: A New Compiler from Curry to Haskell“. In: *Functional and Constraint Logic Programming: 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings*. Hrsg. von Herbert Kuchen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 1–18 (siehe S. 24, 58).

[DCLF07] Javier de Dios Castro und Francisco J. López-Fraguas. „Extra Variables Can Be Eliminated from Functional Logic Programs“. In: *Electron. Notes Theor. Comput. Sci.* 188 (Juli 2007), S. 3–19 (siehe S. 17).

[Ekb12] Anton Ekblad. „Towards a declarative web“. In: *Master of Science Thesis, University of Gothenburg* (2012) (siehe S. 7, 8, 25).

[Ekb15] Anton Ekblad. „A Distributed Haskell for the Modern Web“. Diss. Chalmers Institute of Technology, 2015 (siehe S. 25).

[Han07] M. Hanus. „Putting Declarative Programming into the Web: Translating Curry to JavaScript“. In: *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'07)*. ACM Press, 2007, S. 155–166 (siehe S. 27).

[He16] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`. 2016 (siehe S. 10, 54).

[Kir16] Bastian Kirchmayr. „Übersetzung von Curry nach Java“. In: *Master of Science Thesis, Christian-Albrechts-Universität zu Kiel* (2016) (siehe S. 16, 18–20, 27, 42, 54).

[Pee17] Björn Peemöller. *Normalization and Partial Evaluation of Functional Logic Programs*. Kiel Computer Science Series 2017/1. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, Kiel University, 2017 (siehe S. 15).

[Sev12] Charles Severance. „JavaScript: Designing a Language in 10 Days“. In: *Computer* 45 (2012), S. 7–8 (siehe S. 1).

[Zak11] Alon Zakai. „Emscripten: An LLVM-to-JavaScript Compiler“. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, S. 301–312 (siehe S. 26).

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele

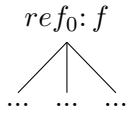
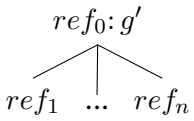
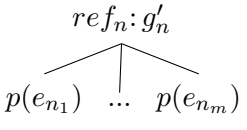
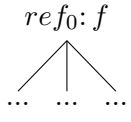
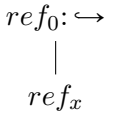
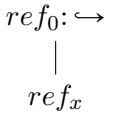
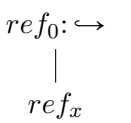
Anhang

Die Semantik von ICurry

A

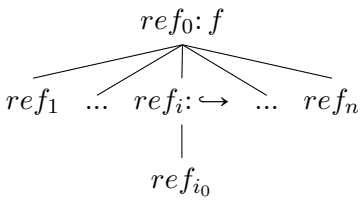
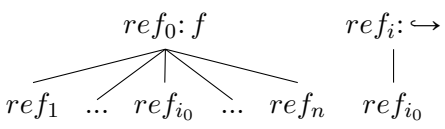
Regel	Teilgraph	Kontrolle	Queue
1		€ ref_0	$(ref_0 : S, F); Q$ $(S, F); Q$
2		€ $	{} {}
3		fail €	$S; Q$ Q
4	ref_0: c ref_0: c	ref_0 €	$([\,], F); Q$ Q
5	ref_0: f ref_1 ... ref_n ref_0: f ref_1 ... ref_n	ref_0 €	$([\,], F); Q$ Q
6	ref_0: c ref_0: c	ref_0 ref_x	$(ref_x : S, F); Q$ $(S, F); Q$
7	ref_0: f_{hnf_i} ref_1 ... ref_i: g ... ref_n ref_0: f_{hnf_i} ref_1 ... ref_i: g ... ref_n	ref_0 ref_i	$(S, F); Q$ $(ref_0 : S, F); Q$
8	ref_0: f ref_1 ... ref_i ... ref_n ref_0: f ref_1 ... ref_i ... ref_n	ref_0 $\sigma(exp)$	$(S, F); Q$ $(ref_0 : S, F); Q$

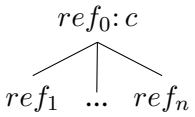
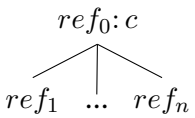
Bedingungen für Regel 5: $n <$ arity(f)
Regel 7: ref_i nicht in Kopfnormalform
Regel 8: $f(\overline{par_n}) = exp \in$ Programm
$\sigma = \{\overline{par_n} \to \overline{ref_n}\}$
ref_1 bis ref_n nicht mit $\hookrightarrow$ beschriftet

Regel	Teilgraph	Kontrolle	Queue
9		$g(\overline{ref_n})$	$(ref_0 : S, F); Q$
		ref_0	$(S, F); Q$
10		$let\ \{\overline{x_n = g_n(\overline{e_{n_m}})}\}\ in\ e$	$(S, F); Q$
		$p(e)$	$(S, F); Q$
11		proj(ref_x)	$(ref_0 : S, F); Q$
		ref_0	$(S, F); Q$
12		ref_0	$(S, F); Q$
		ref_x	$(S, F); Q$

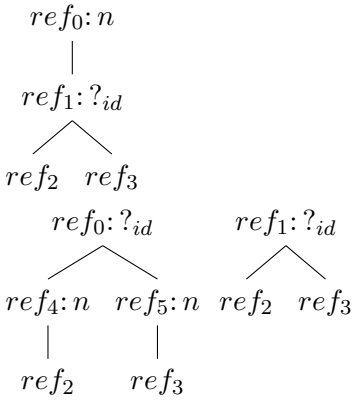
Bedingungen für Regel 9: $g \in Constructors \cup Functions \cup \{?\}$
$g =? \Rightarrow g' = g_{newid()}$
$g \neq? \Rightarrow g' = g$
Regel 10: $p = \{\overline{x_n \rightarrow ref_n}\}$, $\overline{ref_n}$ neue Referenzen
$e_{k_j} \in Referenzen \cup \{\overline{x_n}\}$
$g_k =? \Rightarrow g'_k = g_{k_{newid()}}$
$g_k \notin \{?\} \cup Referenzen \Rightarrow g'_k = g_k$
$\overline{g_n} \in Constructors \cup Functions \cup \{?\}$

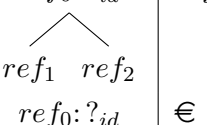
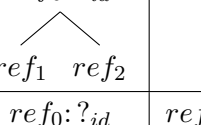
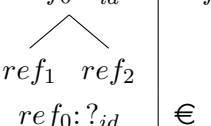
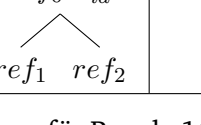
Sonderfall für Regel 10: Wenn $g_k = ref_x$, dann wird ein Knoten $ref_k :\hookrightarrow$ mit Kind ref_x erstellt

13		ref_0	$(S, F); Q$
		ref_0	$(S, F); Q$

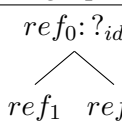
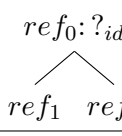
Regel	Teilgraph	Kontrolle	Queue
14		$case\ ref_0\ of\ \{\overline{pat_m \to exp_m}\}$	$(S, F); Q$
		$\sigma(exp_j)$	$(S, F); Q$

Bedingungen für Regel 14: $pat_j = c(\overline{par_n})$
$\sigma = \{\overline{par_n \to ref_n}\}$

Regel	Teilgraph	Kontrolle	Queue
15		ref_1 ref_0	$(ref_0 : S, F); Q$ $(S, F); Q$

Regel	Teilgraph	Kontrolle	Queue
16a		ref_0	$([\,], F); Q$
		∈	$Q +\!\!+ \{([ref_1], F[id \to 1]\} +\!\!+ \{([ref_2], F[id \to 2]\}$
16b		ref_0	$([\,], F); Q$
		∈	$([ref_1], F[id \to 1]); ([ref_2], F[id \to 2]); Q$

Bedingungen für Regel 16a: $F(id) = 0$
Regel 16b: $F(id) = 0$

Regel	Teilgraph	Kontrolle	Queue
17		ref_0	$([\,], F); Q$
		ref_i	$([\,], F); Q$

Bedingungen für Regel 17: $F(id) = i \wedge i \in \{1, 2\}$

B

Dokumentation von Hurry

Übersetzer

Installation

Abhängigkeiten

Voraussetzung für die Kompilierung und Ausführung des Übersetzers ist ein funktionierender Curry-Compiler. Da PAKCS mit SWI-Prolog sich als zu langsam für die Übersetzung der Prelude herausgestellt hat und auch PAKCS mit SICSTUS langsam ist, wird *KiCS2*[1] empfohlen. Weitere Voraussetzungen für die Installation sind das JavaScript-Tool *escodegen* und der *Curry Package Manager* (CPM).

escodegen Für die Ausführung von *escodegen* ist *nodejs* notwendig. Eine Installationsanleitung für nodejs ist unter `https://nodejs.org/de/` zu finden. escodegen kann nach der Installation von nodejs mit

```
npm install -g escodegen
```

installiert werden. Hierfür sind Adminstrationsrechte notwendig. Alternativ kann das Tool auch mit

```
npm install escodegen
```

installiert werden, es muss dann die ausführbare Datei `esgenerate.js` aus dem Verzeichnis `node_modules/bin` nach `esgenerate` verlinkt und im `PATH` auffindbar gemacht werden.

[1] Webseite von KiCS2: `https://www-ps.informatik.uni-kiel.de/kics2/`

CPM Die Installation vom CPM ist im CurryWiki[2] beschrieben. Es gibt auch eine offizielle Anleitung.[3]

Modifiziertes CPM-Repository Da der Übersetzer und seine Abhängigkeiten im offiziellen CPM-Repository nicht vorhanden sind, muss das CPM-Repository geändert werden. Zunächst sollte hierfür das neue Index-Repository[4] in ein passendes Verzeichnis heruntergeladen werden.[5] Beispielhaft wird angenommen, dies sei in das Verzeichnis `/home/user/cpm-index` geschehen. Nun kann in der Konfigurations-Datei `.cpmrc`, welche unter `$HOME/.cpmrc` angelegt werden kann, die Einstellung `REPOSITORY_PATH` gesetzt werden. Für unser Beispiel würde in dieser Datei dann `REPOSITORY_PATH=/home/user/cpm-index` stehen.

Der Übersetzer

Der Übersetzer kann dann mit dem Kommando

```
cpm install hurry-translator
```

installiert werden. Die Executable wird im Verzeichnis der ausführbaren Pakete von CPM erzeugt.[6]

Bedienung

Mit Hilfe des Kommandos

```
hurry-translator --help
```

kann die Hilfe angezeigt werden. Die Übersetzung eines Curry-Moduls `Modul.curry` kann mit

```
hurry-translator Modul.curry
```

ausgeführt werden.

[2] `http://www-ps.informatik.uni-kiel.de/currywiki/tools/cpm`
[3] Unter `http://www-ps.informatik.uni-kiel.de/currywiki/_media/tools/cpm/manual.pdf`
[4] Zu finden unter `https://git.ps.informatik.uni-kiel.de/jsi1/cpm-index`
[5] Zum Beispiel mit `git clone https://git.ps.informatik.uni-kiel.de/jsi1/cpm-index.git`
[6] Normalerweise ist das Verzeichnis `$HOME/.cpm/bin`.

Der Übersetzer übersetzt Curry-Module in passende JavaScript-Dateien. Den JavaScript-Dateien wird die Runtime als Datei beigelegt. Außerdem werden die externen Definition für die Curry-Module den JavaScript-Dateien beigelegt. Diese Dateien können in eine Webseite eingebunden oder in Node.js geladen werden.

Einbindung des erstellten Quelltextes in eine Webseite Im Repository des Übersetzers ist die exemplarische Einbindung in eine Webseite vorgeführt.[7]

Einbindung in Node.js Angenommen es wurde in einem neuen Verzeichnis eine Datei `Test.curry` angelegt. In dieser Datei wird das Modul Test mit einer Funktion `main` definiert. Das Modul kann mit

```
hurry-translator -o build Test.curry
```

übersetzt werden, dabei werden alle notwendigen Dateien im Verzeichnis `build` generiert.

Nun kann mit dem Kommando

```
node
```

Nodejs gestartet werden. In Nodejs kann nun folgendes ausgeführt werden, um die Module und die Runtime zu laden:

```
.load build/hurry-runtime.min.js
.load build/Prelude.curry.js
.load build/Prelude.curry.external.js
.load build/Test.curry.js
```

[7] Siehe `https://git.ps.informatik.uni-kiel.de/jsi1/hurry-translator/tree/master/examples/dist`.

Mit dem JavaScript-Ausdruck,

```
HurryRuntime.Interpreter.run(Test.f_main).then(
  function(results) {
    results.forEach(function(node) {
      s = HurryRuntime.Visualizer.StringVisualizer.toCurryString(node);
      console.log(s);
    })
  }
)
```

der in Nodejs eingegeben werden kann, kann das Programm dann ausgewertet werden.

Quelltext

Im folgenden Text werden kurz die Pakete beschrieben, welche für den Übersetzer erstellt wurden.

ICurry ICurry wurde vollständig von Kirchmayr entwickelt. Für den Übersetzer wurde ICurry in einem CPM-Paket zusammengefasst. Es enthält die Module um annotiertes FlatCurry nach ICurry zu übersetzen. Es ist unter `https://git.ps.informatik.uni-kiel.de/jsi1/icurry` zu finden.

ECMAScript ECMAScript ist ein Curry-Paket, welches einen Datentyp für ECMAScript 5 und ein Modul zur Konvertierung zu ESTree im JSON-Format bereitstellt. Der Quelltext ist unter `https://git.ps.informatik.uni-kiel.de/jsi1/ecmascript` zu finden.

hurry-runtime Die hurry-runtime ist ein NPM-Paket, welches die Runtime inklusive der Visualisierungsmodule enthält. Die hurry-runtime ist in JavaScript geschrieben und benutzt neue ECMAScript Features und muss deshalb transpiliert werden. Das Paket ist unter `https://git.ps.informatik.uni-kiel.de/jsi1/hurry-runtime` zu finden.

hurry-translator Das CPM-Paket hurry-translator enthält den Übersetzer und die kompilierte Runtime. Es ist unter `https://git.ps.informatik.uni-kiel.de/jsi1/hurry-translator` zu finden.

cpm-index Um die Entwicklung mit CPM-Paketen zu ermöglichen wurde ein Fork des Repository `cpm-index`angelegt und dem Index neue Pakete hinzugefügt. Das Repository ist unter `https://git.ps.informatik.uni-kiel.de/jsi1/cpm-index` zu finden.

Webservice

Der Webservice wurde gebaut, um die Client-Server Kommunikation für die Übersetzung und Ausführung zu vereinfachen. Der Quelltext des Webservices befindet sich unter `https://git.ps.informatik.uni-kiel.de/jsi1/hurry-webservice`.

Docker Die einfachste Art den Webservice zu deployen ist in der Nutzung des Docker Containers, welcher auf dem Docker Hub veröffentlicht wurde.[8]

Nach der Docker Installation kann mit dem Kommando

```
docker run -p 80:80 jpsikorra/hurry-webservice
```

der Webservice auf dem Port 80 gestartet werden. Unter `127.0.0.1` kann der Webservice nach dem Start im Browser ausprobiert werden. Weitere Docker-spezifische Optionen können der Docker Anleitung entnommen werden.[9]

CGI Zunächst muss der `hurry-translator` in der Version 0.0.3 und `escodegen` installiert werden. Danach kann das Paket `hurry-webservice` via CPM installiert werden. Hierfür wird wieder das angepasste CPM Index Repository benötigt (siehe S. 80 Abs. Modifiziertes CPM-Repository). Dann muss CGI auf dem Webserver eingerichtet werden. Bei einem Aufruf des hurry-webservice durch den CGI-Handler müssen die passenden Argumente übergeben werden (`HURRY_LIB` und `HURRY_TRANSLATOR_BIN`).

Eine vollständige Installation kann dem Webservice-Beispiel und dem Dockerfile entnommen werden.[10] [11]

[8] `https://hub.docker.com/r/jpsikorra/hurry-webservice/`
[9] `https://docs.docker.com/engine/reference/run/`
[10] Beispiel: `https://git.ps.informatik.uni-kiel.de/jsi1/hurry-webservice/tree/master/example`
[11] Dockerfile: `https://git.ps.informatik.uni-kiel.de/jsi1/hurry-webservice/blob/master/Dockerfile`