ulm university universität
uulm

Das Grammateion

Monadische Implementierung einer virtuellen Maschine zur Ausführung einer polyparadigmatischen Programmiersprache

Sascha Rechenberger

Universität Ulm

Fakultät für
Ingenieurwissenschaften
und Informatik

Institut für
Programmiermethodik
und Compilerbau

Januar 2015

Bachelorarbeit im
Studiengang Informatik

Abstract

Diese Arbeit beschäftigt sich mit der Konzeption und Implementierung einer virtuellen Maschine, die der Ausführung von Programmier- beziehungsweise Skriptsprachen dienen soll, die Sprachelemente sowohl des *imperativen* als auch des *funktionalen* und *logischen* Programmierparadigmas aufweisen. Dabei werden zuerst geeignete Datentypen definiert, gefolgt von der Herleitung der Maschinensprache, die sich an einfachen, in ihrem Paradigma reinen, imperativen, funktionalen und logischen Sprachen orientiert. Anhand dieser Sprache wird dann eine Maschinenarchitektur abgeleitet und als erweiterte Zustandsmonade umgesetzt. Als ausführbares Resultat der Implementierungsarbeit wird dann eine erste Programmiersprache vorgestellt, die, als erweiterte externe Repräsentation der Maschinensprache, einen ersten Eindruck der Funktionsweise und des Potenzials der Maschine bieten soll.

Die Implementierung selbst erfolgte dabei gänzlich in *Haskell* und ist unter der *GNU Public License Version 3* auf *GitHub* unter `https://github.com/SRechenberger/grammata` veröffentlicht.

Gutachter: Prof. Dr. Helmuth Partsch

Betreuer: Alexander Breckel

Fassung 18. Mai 2015

Satz: PDF-LaTeX 2$_\varepsilon$

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Blickt man sich in der heutigen Welt der Programmiersprachen um, so fällt auf, dass *imperative* Sprachen wie *Java* langsam um die Möglichkeiten *funktionaler* Sprachen erweitert werden, um so Vorteile wie die *Seiteneffektfreiheit* oder die der *Lazy-Evaluation* nutzen zu können; auf der anderen Seite werden Sprachen wie *Curry* [3] entwickelt, die das *funktionale* Paradigma um das *logische* erweitert. All dies macht klar, dass es immer populärer wird, die Vorteile verschiedener Sprachparadigmen zu nutzen. Für diese Arbeit war dennoch ein anderes Konzept inspirierend: die teilweise funktionale Sprache *F#* und die imperative *C#*, wie auch auf anderer Seite *Scala* und *Java* lassen sich auf eine interessante, wenngleich simple Weise kombinieren: die in einer Sprache implementierten Funktionalitäten lassen sich in der jeweils anderen einfach aufrufen und nutzen; möglich ist dies, da sie auf die jeweils gleiche *Zwischensprache*, bei *Scala* und *Java* der *JavaVM Bytecode*, bei *F#* und *C#* der Code der *Common Language Infrastructure* beziehungsweise deren konkreter Implementierungen wie, zum Beispiel, *.NET* oder *Mono*, übersetzt werden [12].

Diese Idee soll in dieser Arbeit auf die *Interpretation* von Skriptsprachen angewandt werden: Ziel soll es sein, eine virtuelle Maschine zu entwickeln, die in der Lage ist eine Sprache, oder ein Sprachensystem zu *interpretieren*, das das *imperative*, *funktionale* sowie *logische* Paradigma miteinander vereint; die entwickelte Maschine erhebt dabei aber keinen Anspruch, eine bestmögliche zu sein; auch soll Effizienz keine große Rolle bei der Implementierung spielen; Ziel ist es lediglich, einen *funktionierenden* und *zur Verbesserung des selben tauglichen* Ansatz zu finden.

1.2 Implementierung in Haskell

Um so eine Maschine zu entwickeln sollen zunächst einige theoretische Überlegungen getätigt werden; diesen entspringt dann, nebst ihrer Maschinensprache, ein Modell der zu entwickelnden Maschine. Eine Programmiersprache, die zur Ausprogrammierung formaler Definitionen

besonders gut geeignet scheint, ist dabei die funktionale Sprache *Haskell*; eine formale Definition eines Zustandsautomaten, welcher ebenfalls eine abstrakte Maschine ist, ist mit ihr leicht in ein eine Zustandsmonade nutzendes Programm zu übertragen; andersherum stellt ein derartig entworfenes Haskell Programm ebenfalls eine Art formaler Definition des Verhaltens der Maschine dar, was das Reimplementieren und Optimieren des Ansatzes erleichtert. Aus diesem Grunde ist die Implementierung der zu entwickelnden Maschine letztendlich in Haskell erfolgt.

1.3 Zu dieser Ausarbeitung

Der Hauptteil dieser Ausarbeitung befasst sich in drei Kapiteln mit der Umsetzung des gesteckten Ziels; in Kapitel 2 wird sich dabei zuerst mit der Definition der Maschine selbst, *Grammateion* genannt, befasst; dieses Kapitel ist dabei zu lesen, als Rekonstruktion der Gedanken *vor und während der Implementierungsarbeit*; so folgt oft auf eine *Frage- oder Problemstellung* eine Reihe von *Erwägungen* beendet mit einem *Entschluss*, wie das Problem zu lösen oder die Frage zu beantworten sei. Nehme also die geneigte Leserin und der geneigte Leser an, Sie oder Er stünde selbst vor dieser Aufgabe und stellte sich all jene aufkommenden Fragen, erwöge selbst was erwogen und beschlösse was beschlossen wurde. Zwischen Kapitel 2 und 3 ist sich dann die Implementierungsphase selbst zu denken, deren ausführbares Resultat die Sprache *Grammata* ist, die in Kapitel 3 anhand einiger Beispiele erläutert wird. Viele Worte zur Software selbst zu verlieren bietet sich dabei in dieser Ausarbeitung nicht an, da der Fokus während Implementierung auf der Machbarkeit einer solchen überhaupt und weniger auf der Art und Weise wie diese geschieht lag.

2 Das Grammateion

Ziel dieser Arbeit ist es, eine virtuelle Maschine zur Ausführung einer Programmiersprache zu entwickeln, die das imperative, funktionale und logische Programmierparadigma miteinander vereint; dafür sind einige Überlegungen notwendig: *Welche Datentypen verwaltet die Maschine? Welcher Form sind die Instruktionen, der Maschinencode?* Und *welche Semantik hat die diese?* Da die Maschine letztendlich in *Haskell* implementiert werden soll, und als solche natürlich einen Zustand hat, wird es nötig sein, diesen Zustand zu verwalten; dies geschehe mit Hilfe einer Zustandsmonade, die zudem Fehler, die während der Ausführung auftreten, handhaben können muss; so treten weitere Fragen zur Implementierung selbst auf: *Wie ist die Monade zu gestalten? Wie ihr gehaltener Zustand?*

In diesem Kapitel sollen diese Fragen und Probleme, sowie weitere, während dessen auftretende, gelöst werden, sodass letztendlich ein brauchbarer Ansatz zur Implementierung einer virtuellen Maschine der gesuchten Art entwickelt wurde.

2.1 Daten

Die erste zu stellende Frage ist die der verwalteten Datentypen; da die Algorithmenimplementierung mehr oder weniger Ziel einer jeden Programmiersprache ist, lässt sich einsehen, dass Datentypen für *Wahrheitswerte* sowie *Ganz- und Gleitkommazahlen* sinnvoll sind. Betrachtet man verschiedene Programmiersprachen findet sich oft auch ein *Einheits-* oder *Nulltyp*, welcher entweder tatsächlich *nichts* repräsentiert, wie *nil* in *Lua* [7], *auf nichts verweist*, wie *null* in *Java* [6] oder wie der Wert *()* in *Haskell* [9], der, da er der einzige Wert seines Typs *()* ist, *keinen Informationsgehalt* hat; deshalb soll auch ein solcher Wert zu den von der Maschine verwalteten Datentypen gehören.

Es ließen sich noch weitere *einfache* Datentypen wie *Character* oder allerlei Ausführungen der numerischen Typen aufzählen, doch das spätere Einfügen dieser in die Implementierung sollte bei gut gemachter solcher keinen größeren Aufwand erfordern, zumal die bisher gefundenen für das gesetzte Ziel hinreichend sind; so lassen sie sich wie folgt zusammenfassen:

2.1.1 Einfache Datentypen

Nullwert

Der Nullwert habe nur einen Wert, nämlich *Null*; er repräsentiere dabei weder wie *null* in Java einen Nullzeiger, oder habe wie *void* keine Werte, noch sei er wie *()* in Haskell ein Typ mit nur einem Wert, sondern sei wie *nil* in Lua, der als Wert jedes Typen *keinen Wert* repräsentiert: die Nullifizierung jedes Typen.

Wahrheitswerte

Wahrheitswerte seien die Bool'schen Werte *True* und *False*. Alternativ wären gewiss auch nicht Bool'sche Wahrheitswerte, wie die ternärer Logiken interessant, vor allem zusammen mit dem später durch das logische Paradigma gebrachten Nichtdeterminismus; dies zu untersuchen sei aber nicht Teil dieser Arbeit, weshalb bei binärer Logik verblieben werde.

Ganzzahlen

Da die Implementierung in Haskell erfolgen wird, kann der Datentyp *Integer* [9] verwendet werden; dieser kennt weder untere noch obere Grenze, wodurch dessen Werte nicht überlaufen können; dies wiederum garantiert Korrektheit auch bei großen Zahlenwerten. Die Effizienzeinbußen, die die Verwendung dieses Typen mit sich bringt, werden in Kauf genommen. So decke der Ganzzahlentyp, unter Verwendung des Typen *Integer*, *potienziell alle* ganzen Zahlen ab.

Gleitkommazahlen

Gleitkommazahlen seien die Werte des Haskell Typen *Double*; diese sollten in Präzision und Reichweite an den IEEE Standard für Gleitkommazahlen doppelter Präzision wenigstens heranreichen [15] und sind somit geeignet für das Ausführen von Allzweckprogrammiersprachen.

Eine Alternative wäre, statt Gleitkommazahlen Brüche, nämlich den Typ *Rational* [15], zu verwenden; dieser stellt seine Werte als Paar zweier *Integer* Werte dar, was ihn wiederum dazu befähigt, *potenziell alle* rationalen Zahlen darzustellen; Nachteil dessen wäre aber, dass, werden von der auszuführenden Sprache Gleitkommazahlen verwendet, diese oft in Brüche und zurück konvertiert werden müssten.

2.1.2 Komplexe Daten

Nachdem einfache, die Bedürfnisse einer Programmiersprache stillende Datentypen gefunden wurden, stellt sich jetzt die Frage nach *komplexeren Datentypen* wie *Records*, *Arrays*, *Listen* und dergleichen. Da die resultierende Maschine endlich drei verschiedene Sprachparadigmata zusammen ausführen sollte, empfiehlt es sich, eine Möglichkeit der Modellierung komplexer Daten zu finden, mit der in allen umgegangen werden kann. Dazu soll betrachtet werden, wie die imperative Skriptsprache *Lua* komplexe Daten modelliert, da das dort verwendete Konzept der *assoziativen Arrays* scheint zielführend zu sein; zudem sollen, um ein hinreichend allgemeines Konzept zu finden, das alle etwaigen komplexen Datentypen zu emulieren vermag, auch *Haskell*, als funktionale und Sprache der Implementierung, sowie *Prolog*, als verbreitetste logische Programmiersprache, ob der Möglichkeiten, komplexe Daten zu modellieren, untersucht werden.

Tabellen in Lua

Die einzige Möglichkeit in der Skriptsprache Lua komplexe Daten zu modellieren sind *Tabellen*; der Tabellentyp *table* implementiert dabei assoziative Arrays, also Arrays, die nicht nur über Zahlen, sondern über alle Werte der Sprache indiziert werden können [7]. Betrachtet man das folgende Stück Luacode, sieht man, wie auch ein String als Arrayindex fungieren kann:

```
1  -- Array mit Zahlen als Indizes.
2  array1 = {}
3  array1[0] = 0
4  array1[1] = 1
5
6  -- Array mit Strings als Indizes.
7  array2 = {}
8  array2["i0"] = 0
9  array2["i1"] = 1
10
11 -- Folgende Schreibweise ist aequivalent zu der bei array2 verwendeten.
12 array3 = {}
13 array3.i0 = 0
14 array3.i1 = 1
```

Somit ist ein Array, im Sinne eines Arrays in *C* oder *Java*, in Lua auch immer eine Art Record, dessen Felder gleicher Größe und deren Namen immer Integer sind. Dieser Gedanke ist sehr praktisch, da so die Modellierung eines Arrays auf die eines Records reduziert werden kann

und über diese Konstrukte wie Bäume, Listen, Tupel und somit auch Graphen im allgemeinen modellierbar sind.

Records in Haskell

Haskell stellt für die Datenstrukturierung, unter anderem, die Recordsyntax zur Verfügung. Folgendes Beispiel zeigt die Definition eines Typen für Paare zweier Werte:

```
1 data Paar a b = Paar
2     { fst :: a
3     , snd :: b
4     }
```

fst und *snd* simulieren dadurch Recordfelder, sind jedoch nicht vom Typ *a* beziehungsweise *b* sondern eigentlich $fst :: Paar\ a\ b \rightarrow a$ und $snd :: Paar\ a\ b \rightarrow b$. Somit lässt sich die Definition auch wie folgt umsetzten:

```
1 data Paar a b = Paar a b
2
3 fst :: Paar a b -> a
4 fst (Paar x _) = x
5
6 snd :: Paar a b -> b
7 snd (Paar _ y) = y
```

Durch dieses Konzept lassen sich Recordtypen mit benannten Feldern zurückführen auf Funktionssymbole, die als Argumente die Feldwerte halten. Zugang zu den Feldern kann dabei über von der Typdefinition unabhängige Funktionen gewährt werden; diese könnten auch als *arithmetische Operationen* betrachtet werden; wäre es dem Sprachdesigner also möglich, diese selbst zu definieren, so könnte er auch leicht auf die selbst definierten komplexen Datentypen zugreifen.

Funktionssymbole in Prolog

Prolog verwendet zur Stukturierung von Daten Funktionssymbole ähnlich denen der Prädikatenlogik [10]; das Beispiel in Abbildung 2.1 zeigt Zugriffsprädikate für ein Paare simulierendes Funktionssymbol. Auffällig ist dabei die Ähnlichkeit zu den in Haskell verwendeten Datenkonstruktoren; der große Unterschied liegt dabei, dass Prolog keine Typen kennt. Das heißt

```
1 fst(paar(X,_),X).
2 snd(paar(_,Y),Y).
```

Abbildung 2.1: *fst* und *snd*

also, dass Typen für die Modellierung komplexer Daten, als Funktionssymbole und deren Argumente, keine Rolle spielen müssen, und sollen sie es doch, so ließe sich das zur Compilezeit realisieren, wie es bei der *FuL* Sprache und der diese ausführenden Maschine gedacht ist [1].

Zusammenfassung

Es gelang, die Arraymodellierung auf die Modellierung eines Recordspezialfalls, in dem alle Felder des Records einen Wert gleichen Typs halten, sowie sukzessive bezeichnet sind, zum Beispiel mit den Namen *i0* bis *i10* für ein Array der Länge zehn, zu reduzieren; des weiteren konnten Recordtypen auf *Funktionssymbole*, wie, zum Beispiel, $f(a_0, a_1, ..., a_n)$, und Zugriffsfunktionen, wie $get_0(f(a_0, a_1, ..., a_n)) = a_0, get_1(f(a_0, a_1, ..., a_n)) = a_1, ...$ reduziert werden, die als arithmetische Operationen angesehen werden können; diese seien dabei vom Compilerautoren und Sprachdesigner selbst definierbar, um ihnen hinreichende Freiheit einzuräumen; wie das geschehen kann, muss im weiteren Verlauf geklärt werden. So scheint es, dass ein geschickter, wenngleich nicht der effizienteste, Weg zur Modellierung komplexer Daten für eine polyparadigmatische Sprachen ausführende virtuelle Maschine Funktionssymbole *ähnlich* denen in Prolog beziehungsweise Haskell sind. Sie erlauben die Modellierung von Arrays und Records, und damit von allen anderen Datenstrukturen, wie deren Modellierung in gängigen Sprachen üblich ist.

Strukturen

Eine Datenstruktur zeichne sich aus durch ein Funktionssymbol, repräsentiert durch einen String, und dessen Argumente, repräsentiert durch eine Folge von Daten, die auch Strukturen selbst sein können müssen. Zusätzlich sei noch ihre Arität Teil der Definition.

2.1.3 Zeigertypen

Ein letzter Gedanke bezüglich der Art der verwalteten Datentypen sollte Zeigern gewidmet sein; um jedoch wissen zu können, welcher Zeigertypen es bedarf, müssen sich erst Gedanken

zur Maschinensprache gemacht werden. Die Frage nach Zeigertypen soll deshalb an anderer Stelle erneut aufgegriffen werden.

2.1.4 Implementierung

Bedenkt man die Idee des Nullwertes, der als Wert jedes Typen diesen nullifiziert, die der Strukturen, die rekursiv wieder Strukturen enthalten können müssen, um Datenstrukturen wie Bäume abbilden zu können, und bedenkt man, dass die Implementierung letztendlich in Haskell geschehen soll, so liegt nahe, dass eine sinnvolle Implementierung der verwalteten Datentypen als Summentyp geschehen kann; dieser sehe nun wie folgt aus:

```
1 data Basic
2     = Null
3     | Boolean Bool
4     | Natural Integer
5     | Real Double
6     | Struct String Int [Basic]
```

Der Konstruktor *Null* repräsentiert *Nullwerte*, *Boolean* Bool'sche Werte, die Haskells nutzend, *Natural* Ganzzahlen, *Real* Gleitkommazahlen und *Struct* Strukturen.

Wie bereits angemerkt, könnten noch Zeigertypen fehlen, die gegebenen Falles später ergänzt werden.

2.1.5 Typisierung

Betrachtet man den Abschnitt 2.1.2, so scheint es sinnvoll, bei der zu entwickelnden Maschinensprache von einer *dynamisch Typisierten* auszugehen; so kann dem Nutzer später maximale Freiheit eingeräumt werden, was die Typisierung der auszuführenden Sprache betrifft.

2.2 Maschinensprache

Die nächste sich zu stellende Frage ist die des zu interpretierenden Codes: der naive Ansatz wäre dabei, drei virtuelle Maschinen, wie die *C-Maschine*, die *MaMa* und die *WiM* als Vorbild zu nehmen, und eine Vereinigungsmenge ihrer Maschineninstruktionen als Maschinensprache der zu implementierenden Maschine zu wählen; der offensichtlichste Nachteil dessen wäre aber, dass ein Compilerautor mit einer recht großen Zahl von Maschineninstruktionen arbeiten

müsste; des weiteren ist der Zweck der drei Maschinensprachen ein ganz anderer, als der der zu entwickelnden Sprache: sie sind Zwischensprachen zur Übersetzung in die Zielsprache der Programmiersprache, zum Beispiel Assemblercode eines beliebigen Mikroprozessors; die hier gesuchte Maschinensprache aber soll direkt ausgeführt werden, weshalb sie nicht so feingranular wie die Zwischensprachen sein muss, allgemein genug jedoch, um gängige und auch ungewöhnlichere Sprachkonstrukte darauf abbilden zu können. Nahe liegt deshalb, abstrakter zu bleiben und sich *einfache* imperative, funktionale und logische Sprachen anzusehen und sich später zu überlegen, wie sich diese sinnvoll vereinen lassen; zudem sind Interpreter für solche einfachen Sprachen in Haskell relativ leicht implementierbar.

So wäre eine einfache *imperative* Sprache die Sprache *While*, die, wenngleich minimal, als Berechenbarkeitsmodell betrachtet äquivalent mächtig den Turingmaschinen ist [2]. Sucht man eine minimale *funktionale* Sprache, so böte sich der *Lambdakalkül* an, der ebenfalls turingmächtig ist [11]. Der *Lambdakalkül*, so wie *While*, sind in ihrer reinen Form aber als Programmiersprachen sehr unpraktisch; sie müssten deshalb erweitert werden. Als einfache logische Sprache bietet [1] eine vereinfachte Variante von Prolog namens *ProL* [1], welche auf einige Konzepte wie Arithmetik, den Cut und Konzepte der Metaprogrammierung, wie zum Beispiel das Prolog Prädikat *assert* verzichtet; auch sie soll ob ihrer Tauglichkeit als Maschinensprache geprüft und gegebenen Falles modifiziert werden.

Werde im Anschluss jede dieser drei Sprachen Schrittweise modifiziert, um drei jeweils einfache, aber als Übersetzungsziel nutzbare, Teilmaschinensprachen zu gewinnen; fürderhin werde diskutiert, wie diese drei Sprachen sinnvoll zu vereinigen sind, um letztendlich eine polyparadigmatische Maschinensprache zu finden.

2.2.1 While

Werde hier mit der Modifikation der Sprache *While* begonnen; ein *While* Programm folgt dabei der folgenden Grammatik [2]:

$$
\begin{aligned}
\langle P \rangle \longrightarrow\ & \mathbf{x_i} := \mathbf{x_j} + \mathbf{c} \\
\mid\ & \mathbf{x_i} := \mathbf{x_j} - \mathbf{c} \\
\mid\ & \langle P \rangle\ ;\ \langle P \rangle \\
\mid\ & \textbf{while } \mathbf{x_i} \neq \mathbf{0} \textbf{ do } \langle P \rangle \textbf{ end}
\end{aligned}
$$

[1] *Pro*log *L*anguage

Dabei sind $\mathbf{x_i}$, mit $i \in \mathbb{N}$, Variablenbezeichner und $\mathbf{c} \in \mathbb{N}$ Konstanten [2]. Die Sprache beschränkt sich also darauf, Bezeichnern Werte zuzuweisen, diese Werte um *Konstanten* zu debeziehungsweise inkrementieren, Anweisungen aufeinander folgen zu lassen und auf Schleifen, die ihren Rumpf wiederholen, solange eine bestimmte Variable ungleich Null ist; dieser können dabei innerhalb des Rumpfes neue Werte zugewiesen werden [2]. Hinreichend ist sie dabei dennoch, um alle Kontrollstrukturen gängiger imperativer Programmiersprachen auf sie zurückzuführen; doch dies kann durchaus aufwändig sein und sollte deshalb vereinfacht werden.

Komplexere arithmetische Ausdrücke

Eine erste mögliche Erweiterung sind *komplexere arithmetische Ausdrücke*, wobei von konkreten Operatoren abstrahiert werden *muss*: dies ist unter anderem eine Konsequenz aus Abschnitt 2.1.2; zudem kann dem Compilerautoren so später die Möglichkeit gegeben werden, eigene arithmetische Operatoren zu definieren, wie er es auch mit Hilfe atomarer Maschineninstruktionen könnte; dies kann erfolgen, indem ein Operator als *Funktion* betrachtet wird, wobei diese auch bezüglich ihrer Stelligkeit frei wählbar sein kann. *Unäre* sowie *binäre* Operatoren sind jetzt bereits abgedeckt: ein Ausdruck kann nun noch entweder eine *Konstante* oder eine *Variable* sein; da imperative Sprachen meist mit Bezeichnern veränderlicher Werte arbeiten, soll dieses schon in *While* umgesetzte Konzept erhalten bleiben. Konstanten wiederum können im Weiteren nicht einfach natürliche Zahlen, sondern sollten Werten des *Basic* Typen sein. Folge ein solcher Ausdruck also dieser Form:

$$\begin{aligned} \langle E \rangle \longrightarrow\ & \mathbf{x} \\ \mid\ & \mathbf{c} \\ \mid\ & \mathbf{op_n}\ \langle E \rangle_1 \ldots \langle E \rangle_n \end{aligned}$$

Hierbei sei $\mathbf{x}$ ein Stringbezeichner und $\mathbf{c}$ vom Typ *Basic*; $\mathbf{op_n}$ bilde seine n Argumente $\langle E \rangle_1 \ldots \langle E \rangle_n$ auf einen Wert des Typen `Basic` ab.

Anweisungen

Mit Hilfe der erweiterten arithmetischen Ausdrücke können auch die Anweisungen der Sprache erweitert werden; die beiden Zuweisungsregeln der Ursprungsgrammatik können deshalb zu

folgender vereinfacht werden:

$$\langle P \rangle \longrightarrow \mathbf{x_i} := \langle E \rangle$$

Weiter empfiehlt sich, *bedingte Anweisungen* einzuführen, um dem Compilerprogrammierer später nicht die Aufgabe, diese mit **while** Schleifen zu simulieren, aufzubürden; werde also die Grammatik der *While* Programme um diese Regel erweitert:

$$\langle P \rangle \longrightarrow \textbf{if } \langle E \rangle \textbf{ then } \langle P \rangle \textbf{ else } \langle P \rangle \textbf{ end}$$

Hierbei muss der Ausdruck $\langle E \rangle$ zu einem *Wahrheitswert* ausgewertet werden können. Endlich können auch die *Schleifenkonstrukte* um komplexere Bedingungen angereichert werden; dies geschehe durch Einführen folgender Regel:

$$\langle P \rangle \longrightarrow \textbf{while } \langle E \rangle \textbf{ do } \langle P \rangle \textbf{ end}$$

Wie bei bedingten Anweisungen muss auch hier die Bedingung $\langle E \rangle$ zu einem Wahrheitswert ausgewertet werden können.

Zusammenfassung

Fasst man die vorgenommenen Modifikationen zusammen, erhält man aus der anfangs beschriebenen *While* Sprache die durch folgende Grammatik beschriebene:

$$\begin{aligned}
\langle E \rangle &\longrightarrow \mathbf{x} \\
&\quad | \ \mathbf{c} \\
&\quad | \ \mathbf{op_n} \ \langle E \rangle_1 ... \ \langle E \rangle_n \\
\langle P \rangle &\longrightarrow \langle S \rangle_1 \ ; ... \ ; \ \langle S \rangle_n \\
\langle S \rangle &\longrightarrow \mathbf{x_i} := \langle E \rangle \\
&\quad | \ \textbf{if } \langle E \rangle \textbf{ then } \langle P \rangle \textbf{ else } \langle P \rangle \textbf{ end} \\
&\quad | \ \textbf{while } \langle E \rangle \textbf{ do } \langle P \rangle \textbf{ end}
\end{aligned}$$

Dabei *muss* erlaubt sein, dass eine Anweisungsfolge $\langle P \rangle$ auch leer sein darf, da nur dann eine Anweisung **if** $\langle E \rangle$ **then** $\langle P \rangle$ **end** simuliert werden kann.

2.2.2 Lambdakalkül

Werde jetzt der *Lambdakalkül* gleich der *While* Sprache schrittweise modifiziert; es kann von folgender Grammatik ausgegangen werden [11]:

$$\begin{aligned} \langle L \rangle \longrightarrow\ & \mathbf{c} \\ \mid\ & \lambda\ \mathbf{s}\ .\ \langle L \rangle \\ \mid\ & \langle L \rangle\ \langle L \rangle \\ \mid\ & (\ \langle L \rangle\ ) \end{aligned}$$

Dabei sind **c** *symbolische Variablen* und **s** das im Term der Abstraktion *gebundene Symbol*.

Arithmetische Ausdrücke

Eine erste vorzunehmende Erweiterung sind, wie schon bei *While*, arithmetische Ausdrücke; diese kommen im minimalen Lambdakalkül nicht vor und müssen, wie auch Zahlen, aufwändig codiert werden [2], weshalb eine Erweiterung um diese, will man eine praktikable Sprache, sehr sinnvoll scheint; auch liegt nahe, dass sie ähnlich den in Abschnitt 2.2.1 der erweiterten *While* Sprache sein sollten, weshalb eine Erweiterung der Kalkülsprache die folgende Regel sei:

$$\langle L \rangle \longrightarrow\ \mathbf{op_n}\ \langle L \rangle_1 ... \ \langle L \rangle_n$$

Auch müssen, da die Argumente endlich zu Basiswerten ausgewertet werden können müssen, konstante Basiswerte von Stringbezeichnern unterschieden werden; werde deshalb die Regel der Ursprungsgrammatik, die symbolische Variablen einführt, durch die folgende ersetzt:

$$\begin{aligned} \langle L \rangle \longrightarrow\ & \mathbf{b} \\ \mid\ & \mathbf{s} \end{aligned}$$

In dieser sei **b** eine Konstante des Typs *Basic* und **s** ein Stringbezeichner.

Ausdrücke

Die nächste Frage ist die nach Ausdrucksformen, um die der Kalkül sinnvoller Weise zu erweitern ist; eine erste einzusehende solche wäre die der *bedingten Ausdrücke*, da diese sonst

[2] siehe *Church Numerale* [5]

nur über den Ausdruck $ifthenelse = (\lambda\ c\ t\ h.c\ t\ h)$ und die Wahrheitswerte codierenden Ausdrücke $true = (\lambda\ x\ y.x)$ und $false = (\lambda\ x\ y.y)$ möglich wären. Werde die Sprache also als nächstes um die folgende Regel erweitert:

$$\langle L\rangle \longrightarrow \textbf{if}\ \langle L\rangle\ \textbf{then}\ \langle L\rangle\ \textbf{else}\ \langle L\rangle\ \textbf{end}$$

Dabei muss der Bedingungsausdruck selbstverständlich in einen Wahrheitswert ausgewertet werden können.

Eine andere Ausdrucksform ist die der *verschränkt rekursiven lokalen Ausdrucksdefinitionen*; diese sind im minimalen Lambdakalkül nur mit Hilfe eines *Fixpunktkombinators* [5] aufwändig realisierbar; werde deshalb die folgende Regel eingeführt:

$$\langle L\rangle \longrightarrow \textbf{letrec}\ \mathbf{id_1} := \langle L\rangle\ ;\ldots;\ \mathbf{id_n} := \langle L\rangle\ ;\ \textbf{in}\ \langle L\rangle\ \textbf{end}$$

Hierbei sei den Symbolen $\mathbf{id_i}$, mit $i \in [1;n]$ im nach **in** stehenden Ausdruck, sowie in den Definitionen der anderen Symbole, der ihnen zugewiesene Ausdruck gleich.

Aufrufsemantik

Um einen Ausdruck des Lambdakalküls auszuwerten, können verschiedene Aufrufsemantiken verwendet werden; umgesetzt werde hier die Call-By-Need Semantik, da diese, wenngleich auch Verwaltungsaufwand, einige Vorteile, wie, zum Beispiel, die Möglichkeit unendlicher Datenstrukturen, mit sich bringt; um diese jedoch umzusetzen, bedarf es eines Speichers, *Halde* genannt, an dem die Abschlüsse gespeichert werden können, und Verweisen, *Zeigern*, auf die deponierten Abschlüsse.

Eine Möglichkeit der Implementierung der Call-By-Need Semantik wäre die Assoziation der gebundenen Variablen mit den Adressen der die Argumente haltenden Haldenzellen mittels einer Symboltabelle; dafür sind lediglich an bestimmten Stellen α-Konversionen nötig, um zu verhindern, dass lokale und globale Bezeichner verwechselt werden. Eine andere Möglichkeit ist die direkte Ersetzung der gebundenen Bezeichner mit dem Verweis auf die entsprechende Haldenzelle; dazu ist jedoch die Einführung einer neuen Ausdrucksform in die Sprache nötig; symbolische Verwechslungen können dabei gar nicht erst auftreten, da die Symbole aus dem Ausdruck entfernt werden. Da für die erste Methode eine neue Datenstruktur, die Symboltabelle, gehalten werden muss, für die zweite aber nicht und so die Interpretation der Sprache maschinenunabhängiger verlaufen kann, soll die Ersetzungsmethode gewählt werden.

Dazu *muss* folgende Regel in die Grammatik der Sprache mit aufgenommen werden:

$$\langle L \rangle \longrightarrow \mathbf{p}^*$$

Dabei sei $\mathbf{p}^*$ ein Zeiger auf eine Haldenzelle; werde dazu in Abbildung 2.2 demonstriert, wie die *Zeigerausdrücke* genutzt werden; links ist dabei der auszuwertende Ausdruck, rechts die Halde zu sehen. Ein Funktionsargument wird zuerst auf die Halde kopiert und durch einen auf die entsprechende Haldenzelle verweisenden Zeigerausdruck ersetzt; danach erst wird die β-Reduktion durchgeführt.

Schritt 1:
Auswertungsbeginn

$(\lambda\ f\ a.(f\ a\ c))\ (\lambda\ x.x)\ (\lambda\ o.(o\ o))$

#	1

Schritt 2:
Argumente auf die Halde legen und durch Zeiger ersetzten

$(\lambda\ f\ a.(f\ a\ c))\ 1^*\ 2^*$

#	3
$\lambda\ o.(o\ o)$	2
$(\lambda\ x.x)$	1

Schritt 3:
β-Reduktion

$(1^*\ 2^*\ c)$

#	3
$\lambda\ o.(o\ o)$	2
$(\lambda\ x.x)$	1

Abbildung 2.2: Funktionsapplikation unter Call-By-Need Semantik und Verwendung der Zeigerausdrücke.

Zusammenfassung

Nach allen Modifikationen ergibt sich die folgende Grammatik:

$$
\begin{aligned}
\langle L \rangle \longrightarrow\ & \mathbf{b} \\
\mid\ & \mathbf{s} \\
\mid\ & \lambda\ \mathbf{s}\ .\ \langle L \rangle \\
\mid\ & \mathbf{op_n}\ \langle L \rangle_1 \ldots\ \langle L \rangle_n \\
\mid\ & \langle L \rangle\ \langle L \rangle \\
\mid\ & (\ \langle L \rangle\ ) \\
\mid\ & \mathbf{letrec}\ \mathbf{id_1} := \langle L \rangle\ ; \ldots ;\ \mathbf{id_n} := \langle L \rangle\ ;\ \mathbf{in}\ \langle L \rangle\ \mathbf{end} \\
\mid\ & \mathbf{if}\ \langle L \rangle\ \mathbf{then}\ \langle L \rangle\ \mathbf{else}\ \langle L \rangle\ \mathbf{end} \\
\mid\ & \mathbf{p}^*
\end{aligned}
$$

Wie die *While* Sprache wurde sie um *arithmetische Ausdrücke* sowie um einen *bedingten Ausdruck*, als Gegenstück zur bedingten Anweisung, erweitert; zudem kamen *Zeigerausdrücke* hinzu, die die Call-By-Need Auswertung unterstützen sollen, sowie die Möglichkeit lokaler rekursiver Definitionen. Als letztes werde jetzt die Sprache *ProL* betrachtet, um eine logische Teilmaschinensprache zu finden.

2.2.3 ProL

Man betrachte die in [1] beschriebene und durch folgende Grammtik definierte Sprache *ProL*:

$$
\begin{aligned}
\langle T \rangle \longrightarrow\ & \mathbf{a} \\
\mid\ & \mathbf{X} \\
\mid\ & \_ \\
\mid\ & \mathbf{f}\ (\ \langle T \rangle_1 , \ldots , \langle T \rangle_n\ ) \\
\langle G \rangle \longrightarrow\ & \mathbf{g}\ (\ \langle T \rangle_1 , \ldots , \langle T \rangle_n\ ) \\
\mid\ & \mathbf{X} = \langle T \rangle \\
\langle C \rangle \longrightarrow\ & \mathbf{q}\ (\ \mathbf{X}_1 , \ldots , \mathbf{X}_n\ ) \Leftarrow \langle G \rangle_1 , \ldots , \langle G \rangle_n \\
\langle A \rangle \longrightarrow\ & \Leftarrow \mathbf{G}_1 , \ldots , \mathbf{G}_n \\
\langle P \rangle \longrightarrow\ & \langle C \rangle_1 \ldots\ \langle C \rangle_n\ \langle A \rangle
\end{aligned}
$$

Sie stellt dabei bereits eine sehr kompakte logische Programmiersprache dar, macht allerdings zwei kleine aber entscheidende Einschränkungen, die aufgehoben werden können.

Klauseln

ProL schränkt den Klauselkopf dahingehend ein, dass das Kopfprädikat *nur* Variablen enthalten darf; das schränkt die Sprache zwar im allgemeinen nicht ein, scheint aber dennoch unnötig; deshalb soll die Regel für $\langle C \rangle$ durch die folgende ersetzt werden:

$$\langle C \rangle \longrightarrow \mathbf{q} \; (\; \mathbf{T}_1 \; , \ldots \; , \mathbf{T}_n \;) \Leftarrow \langle G \rangle_1 \; , \ldots \; , \langle G \rangle_n$$

Dadurch wurden die im Kopfprädikat verwendbaren Terme von Variablen allein auf alle Terme erweitert.

Unifikationsziel

Betrachtet man die Regel für $\langle G \rangle$, sieht man, dass das Unifikationsziel ebenfalls insofern eingeschränkt ist, als dass jeweils eine *Variable* mit einem *Term* unifiziert wird; dies lässt sich ebenfalls verallgemeinern durch das ersetzten der entsprechenden Regel mit der folgenden:

$$\langle G \rangle \longrightarrow \langle T \rangle \sim \langle T \rangle$$

Dadurch kann direkt ein *Term* mit einem anderen unifiziert werden.

Anonyme Variablen

Wie *Prolog* kennt auch *ProL* anonyme Variablen; das Ergebnisignorieren kann dabei aber dem Compilerautoren überlassen werden, weshalb sie aus der Termdefinition getilgt werden können.

Disjunktionen

Eine weitere Einschränkung der *ProL* ist die, dass Ziele in Anfragen oder Regelrümpfen lediglich *konjunktiv* verknüpft werden können; *Prolog* dagegen unterstützt auch die *Disjunktion* mit ';';

Werde dies durch Einführen der folgenden Regeln ermöglicht:

$$\begin{aligned}\langle G\rangle \longrightarrow\ & \langle G\rangle_1 \,,\ldots\, , \langle G\rangle_n \\ |\ & \langle G\rangle_1 \,;\ldots\, ; \langle G\rangle_n \\ |\ & (\ \langle G\rangle\ )\end{aligned}$$

Dabei kann so auch die *leere Klausel*, als Dis- beziehungsweise Konjunktion ohne Ziele dargestellt werden, was das Codieren von *Fakten* erleichtert. Zusätzlich müssen die Produktionsregeln für *Anfragen* und *Regeln* modifiziert werden:

$$\begin{aligned}\langle C\rangle &\longrightarrow \mathbf{q}\ (\ \mathbf{T}_1 \,,\ldots\, , \mathbf{T}_n\ ) \leftarrow \langle G\rangle\ . \\ \langle A\rangle &\longrightarrow\ ?\!-\ \langle G\rangle\end{aligned}$$

Zusammenfassung

Nach diesen Änderungen bleibt folgende Grammatik:

$$\begin{aligned}\langle T\rangle \longrightarrow\ & \mathbf{a} \\ |\ & \mathbf{X} \\ |\ & \mathbf{f}\ (\ \langle T\rangle_1 \,,\ldots\, , \langle T\rangle_n\ ) \\ \langle G\rangle \longrightarrow\ & \mathbf{g}\ (\ \langle T\rangle_1 \,,\ldots\, , \langle T\rangle_n\ ) \\ |\ & \langle T\rangle\ \sim\ \langle T\rangle \\ |\ & \langle G\rangle_1 \,,\ldots\, , \langle G\rangle_n \\ |\ & \langle G\rangle_1 \,;\ldots\, ; \langle G\rangle_n \\ |\ & (\ \langle G\rangle\ ) \\ \langle C\rangle \longrightarrow\ & \mathbf{q}\ (\ \langle T\rangle_1 \,,\ldots\, , \langle T\rangle_n\ ) \leftarrow \langle G\rangle\ . \\ \langle A\rangle \longrightarrow\ & ?\!-\ \langle G\rangle \\ \langle P\rangle \longrightarrow\ & \langle C\rangle_1 \ldots\ \langle C\rangle_n\ \langle A\rangle\end{aligned}$$

Ein weiterer Unterschied zu *ProL* liege jetzt dabei, dass ein *Atom* **a** nicht einfach nur eine *textuelle* Konstante, sondern ein *Wert des Typs Basic* sein kann, welche derartige Konstanten, des *Struktur* Typs wegen, enthält.

2.2.4 Möglichkeiten der Synthesis

Jetzt gilt es, die drei gefunden Sprachen auf sinnvolle Art und Weise miteinander zu verknüpfen und gegebenen Falles nötige Modifikationen daran durchzuführen; da bei den verwalteten Daten bereits eine Übereinkunft gefunden wurde, muss man sich darum nicht mehr sorgen. So werde jetzt versucht, die Kombination der Teilsprachen paarweise durchzuführen, um einen Weg zu finden, sie sinnvoll zu verknüpfen.

Logisch und funktional

Ein geschickter Ansatz zur Verknüpfung des funktionalen und des logischen Paradigmas findet sich in der Sprache *Curry*, die grundsätzlich Haskell sehr ähnlich, allerlei Konzepte der logischen Programmierung zulässt [3]: zum Beispiel können Funktionen in Curry nichtdeterministisches Verhalten aufweisen [4], was folgendes Beispiel bei [4] zeigt:

```
1 element :: [el] -> el
2 element (el : _)   = el
3 element (_  : els) = element els
```

Dabei wird zuerst versucht das *erste* Element der Liste zurückzugeben und als Alternative dann die Funktion rekursiv auf den Listenrest angewandt. Der Basisfall der leeren Liste wird dabei, wie in Prolog, als *unbeweisbar* angenommen. Der Ausdruck `element [1,2,3,4]` kann jetzt die Werte 1 bis 4 annehmen [3], wogegen das GHC, gäbe man den selben Code als Haskellprogramm ein, [4] monierte, dass die beiden Regeln überlappten.

Ein nichtdeterministisches Verhalten wie in *Curry*, bei dem ein Ausdruck mehrere Werte annehmen kann, scheint, der prologähnlichen logischen Teilsprache wegen, sehr wünschenswert für die gesuchte Maschinensprache; es muss aber eine Lösung gefunden werden mit dem Nichtdeterminismus auch im Imperativen umzugehen.

Funktional und imperativ

Eine Möglichkeit funktionale Sprachelemente mit Call-By-Need Semantik in imperative Sprachen einzubinden findet sich recht intuitiv darin, zusätzlich zu arithmetischen auch *funktionale*

[3] getestet mit `http://www-ps.informatik.uni-kiel.de/smap/smap.cgi?new/curry` (15.01.2015)
[4] *The Glorious Glasgow Haskell Compilation System*, Version 7.6.3

Programmausdrücke in eine imperative Sprache einzubinden; dazu bedarf es lediglich der Möglichkeit, Abschlüsse anzulegen, damit spätere, durch Anweisungen verursachte Zustandsänderungen, keinen anderen Wert produzieren, als der Ausdruck zum Anfangszeitpunkt seiner Auswertung hätte haben sollen. Auch wäre es von Vorteil, könnten Zeiger auf Abschlüsse unterversorgter Funktionsanwendungen verweisen und diese Ausdrücke so repräsentieren. So bleibt für einen funktionalen Ausdruck, *im Kontext des Programmzustands zu Auswertungsbeginn*, die *referenzielle Transparenz* [1] gewährleistet.

Will man aber imperative Sprachelemente, genauer *zustandsverändernde Anweisungen*, in funktionale Sprachen einbinden, sprich, ermöglichen, während der Auswertung eines funktionalen Ausdrucks, Variablenwerte zu verändern, wäre die *referenzielle Transparenz nicht* mehr gewährleistet und einer der größten Vorteile funktionaler Sprachen dahin. Dennoch bietet sich eine Möglichkeit: kapselte man imperative Programmteile in *Funktionen*, also Unterprogramme, die nur *ihren lokalen* Zustand manipulieren und einen Wert zurückliefern, könnten solche problemlos in funktionale Programmausdrücke eingebettet werden; auch andersherum und im Hinblick auf die Einbindung logischer in funktionale Programme und umgekehrt ist dies ein vielversprechender Ansatz.

Imperativ und logisch

Auch zum Einbinden der imperativen Sprache in die logische, sowie andersherum, scheint das Kapseln der *fremden* Sprachelemente in *seiteneffektfreie Unterprogramme* eine gute Lösung; gekapselt in eine Funktion könnte ein Aufruf eines logischen Unterprogramms zudem als *Ausdruck nichtdeterministischen Wertes* gelten; dies schlösse auch den Kreis zur ersten Betrachtung über die funktional-logische Kombination.

Was jedoch geschieht in einem Falle, in dem ein nichtdeterministischer Ausdruck, Bedingung einer Kontrollstruktur ist? Eine Möglichkeit wäre, das erste oder *ein* zufälliges der Ergebnisse als Bedingung zu nehmen und alle anderen zu verwerfen. Eine andere Möglichkeit wäre das Ausführen *aller möglichen Programmpfade*, inklusive des Verwerfens eines Ausführungspfades via *Backtracking*, was freilich wenig effizient, aber bezüglich des möglichen Kontrollflusses hochinteressant wäre.

Zusammenfassung

Beendeter Betrachtungen scheint es, dass ein geschickter Weg, die *verschiendenparadigmatischen*, in sich aber *reinen*, Sprachen zu kombinieren, das Kapseln der jeweiligen Sprachen in

aufrufbare *Unterprogramme* ist; wird dabei in einer beliebigen Sprache ein Unterprogramm aufgerufen, muss das aufrufende Unterprogramm nicht wissen, welchen Paradigmas das gerufene Unterprogramm ist, da der Aufruf schlicht einen Ausdruck mit einem Wert repräsentiert.

2.2.5 Umsetzung der Synthesis

Werden jetzt die drei Teilsprachen im Lichte der zu vollziehenden Synthesis betrachtet und dahingehend aufbereitet, als Unterprogramme gekapselt und definiert zu werden. Geht man von einem klassischen Prozeduraufruf der Form $p\,(a_1, ..., a_n)$ aus, das heißt eine Prozedur, genannt p, bekommt n Argumente a_1 bis a_n übergeben, verarbeitet diese und gibt danach einen Wert zurück, gilt es, die drei Sprachen um Verwaltungsinformationen bezüglich formaler Parameter und Prozedurnamen zu erweitern, sowie um Anweisungen, die festlegen, wann welcher Wert zurückgegeben wird; dabei sei davon auszugehen, dass sowohl die Argumente, ausgewertet, als auch der zurückgegebene Wert, des Typen *Basic* sind.

Zuerst sollte jedoch das nichtdeterministische Verhalten und dessen Steuerung und Nutzung im Allgemeinen reflektiert werden.

Nichtdeterminismus

Geht man davon aus, dass der Aufruf eines *logischen* Unterprogramms mehrere Programmabläufe zur Folge haben kann, scheint es sinnvoll, wäre der Zustand der Variablen für jeden zu einem Zeitpunkt möglichen Verlauf zu dessen Beginn der gleiche; sprich: es müssen *Rücksetzpunkte* gehalten werden, die den Programmzustand zum Zeitpunkt der Entscheidung für einen möglichen Programmablauf sichern; dies bringt im weiteren ein Problem mit sich: wird der Programmzustand für jeden Ablauf wieder neu hergestellt, gehen alle Informationen, über das in den vergangenen Abläufen geschehene, verloren; wenngleich dies natürlich im Sinne der Sauberkeit logischer Programme ist, wäre es praktisch, *könnten* Informationen in den nächsten Durchlauf *weitergetragen* und dort bei Bedarf *abgerufen* werden; da die Maschine mit dem Typ *Basic* arbeiten soll und dieser, wie gezeigt wurde, gut dazu geeignet ist, auch komplexe Daten zu repräsentieren, reicht es, dafür einen einzelnen, einen Wert des Typs *Basic* haltenden Speicherplatz zu reservieren; heiße dieser Speicherplatz fortan *persistentes Register*. Um dieses Register zu nutzen werden folgende zwei Operationen definiert:

keep (x) lege den Wert x in das persistente Register ab und habe selbst den Wert x.

remind lese den zuletzt abgelegten Wert aus dem persistenten Register aus; wurde bisher kein solcher abgelegt, so resultiere *Null*.

Um mehr Kontrolle über den Programmfluss zu haben scheint es zudem sinnvoll, Backtracking erzwingen zu können; sprich: den aktuellen Programmablauf abzubrechen und am letzten Rücksetzpunkt zu beginnen; dazu ist eine weitere Operation nötig:

backtrack verwerfe den aktuellen Ausführungspfad und beginne, falls ein solcher vorhanden ist, beim letzten Rücksetzpunkt; ist kein weiterer gesetzt worden, terminiere die Ausführung.

Da *keep* und *backtrack auch* der Charakter einer Prozedur und *remind* der eines Ausdrucks zugesprochen werden kann, können diese Operationen durch das Erweitern der Grammatik der imperativen Teilsprache durch die folgenden Regeln eingefügt werden:

$$\begin{aligned} \langle E \rangle &\longrightarrow \textbf{remind} \\ \langle S \rangle &\longrightarrow \textbf{backtrack} \\ &\mid \textbf{keep}\ \langle E \rangle \end{aligned}$$

Der funktionalen Teilsprache werden die Operationen mit dieser Regel beigefügt:

$$\begin{aligned} \langle L \rangle &\longrightarrow \textbf{remind} \\ &\mid \textbf{backtrack} \\ &\mid \textbf{keep}\ \langle L \rangle \end{aligned}$$

Imperativ

Einem Unterprogramm in gängigen imperativen Sprachen sind als Bezeichner für gewöhnlich zum einen lokale und globale Variablen und zum anderen formale Parameter bekannt; da die globalen Variablen allen Prozeduren zur Verfügung stehen, müssen diese an anderer Stelle verwaltet werden. Werde die Grammatik der gefundene imperativen Sprache mit den folgenden Regeln dahingehend erweitert, dass ein imperatives Unterprogramm $\langle PROC \rangle$ definiert ist durch einen Bezeichner **name**, n Parameter $\mathbf{param_1}$ bis $\mathbf{param_n}$, m lokale Variablen und eine Anweisungsfolge $\langle P \rangle$:

$$\begin{aligned} \langle PROC \rangle &\longrightarrow \textbf{proc name}\ (\ \textbf{param}_1\ ,\ldots\ ,\ \textbf{param}_n\ )\ \textbf{with}\ \langle D \rangle_1\ ;\ldots\ ;\ \langle D \rangle_m\ \textbf{does}\ \langle P \rangle\ \textbf{end} \\ \langle D \rangle &\longrightarrow \textbf{var id}\ [\ :=\ \langle E \rangle] \end{aligned}$$

Noch kann ein solches Programm aber keine Werte zurückliefern; bei der Sprache *While* wurde das fehlen einer *return* Anweisung dadurch kompensiert, dass das Programm eine Variable x_0 als Rückgaberegister nutzt [2]. Orientiert man sich aber an konkreten modernen Programmiersprachen, fügt man eine solche Anweisung hinzu; dies geschehe durch die Erweiterung der Grammatik mit folgender Regel:

$$\langle S \rangle \longrightarrow \textbf{return } \langle E \rangle$$

Ein anderer Aspekt der Unterprogramme ist der, dass sie in imperativen Unterprogrammen statt als Funktion als Prozedur aufgerufen werden können; um aber eine Funktion zu haben, *müsste* ihnen dann gestattet werden, Seiteneffekte zu haben; dies steht aber im Widerstreit zu der bisherigen Annahme, ein Unterprogrammaufruf wäre lediglich ein seiteneffektloser Ausdruck. Wäre die zu entwickelnde Sprache dazu gedacht, von Programmierern direkt genutzt zu werden, sollte diese Annahme beibehalten werden; zu entwickeln ist aber eine *Maschinensprache*, die, wenngleich abstrakter als übliche, den Entwickler einer Programmiersprache möglichst nicht weiter einschränken soll, als eine Maschinensprache jeder anderen virtuellen oder realen Maschine.

Werde der Aspekt der *Seiteneffektfreiheit der Unterprogrammaufrufe* deshalb an dieser Stelle verworfen; es obliege dem Nutzer, entsprechende Prüfungen durchzuführen.

Dadurch können Unterprogramme auch als Prozeduren aufgerufen werden, was folgende Regel einführt:

$$\langle S \rangle \longrightarrow \textbf{call name } (\langle E \rangle_1 , \dots , \langle E \rangle_n)$$

Des weiteren sollen auch in arithmetischen Ausdrücken Unterprogrammaufrufe möglich sein; das ermögliche diese Regel:

$$\langle E \rangle \longrightarrow \textbf{name } (\langle E \rangle_1 , \dots , \langle E \rangle_n)$$

Funktional

Da auch funktionale Unterprogramme als Prozeduren gerufen werden können, müssen für sie formale Parameter sichtbar sein; werde deshalb die Grammatik der funktionalen Sprache um

die folgende Regel erweitert:

$$\langle LAMBDA \rangle \longrightarrow \textbf{lambda name} \; (\; \textbf{param}_1 \; , \ldots \; , \; \textbf{param}_n \;) \; \textbf{is} \; \langle L \rangle \; \textbf{end}$$

Hierbei bindet ein Unterprogramm $\langle LAMBDA \rangle$ bezeichnet durch einen String **name**, die Parameter $param_1$ bis $param_n$ in seinem Programmausdruck $\langle L \rangle$.

Auch sollen aus funktionalen Unterprogrammen andere Unterprogramme aufgerufen werden können; werde deshalb die folgende Regel in die Grammatik eingefügt:

$$\langle L \rangle \longrightarrow \textbf{name} \; (\; \langle L \rangle_1 \; , \ldots \; , \; \langle L \rangle_n \;)$$

Wendet man fürderhin, im Kontext der Synthesis, seine Gedanken erneut auf die Call-By-Need Semantik und durchdenkt man, wie, zum Beispiel, ein imperatives Unterprogramm ein funktionales aufruft, dieses jedoch seinen Ausdruck nicht zu einem Basiswert auswerten kann und einen funktionalen Programmausdruck zurückgäbe, stellt sich die Frage, wie mit dieser Situation umgegangen werden soll, denn ein imperatives Unterprogramm kann einen funktionalen Ausdruck nicht verarbeiten; es wäre jedoch übereilt, dies als Fehler zu werten und die Ausführung zu terminieren: man denke sich eine Situation, in der ein funktionales Unterprogramm einen Abstraktionsausdruck ergeben soll, ein anderes diesen dann übergeben bekommt und verwendet; zwischen Rück- und Eingabe könnten allerlei Unterprogramme stehen. *Wie könnten also funktionale Ausdrücke auch durch logische und imperative Programme hindurch getragen werden?* Daten werden durch den Datentyp *Basic* repräsentiert, also läge es nahe, diesen um funktionale Ausdrücke zu erweitern; doch wurde mit dem Einführen der *Zeigerausdrücke* bereits eine geschicktere Lösung gefunden. So scheint sinnvoll, kann ein Ausdruck nicht gänzlich ausgewertet werden, muss er auf der Halde abgelegt und der Verweis auf ihn als Funktionsergebnis zurückgeliefert werden; diese müssen aber des Typs *Basic* sein, also kann der weiter oben gestellten Frage nach Zeigertypen zumindest eine Antwort gegeben werden: man entferne die Haldenzeiger aus den funktionalen Ausdrücken und füge sie dem Typen *Basic* mit dem Konstruktor $HeapObj :: Int \rightarrow Basic$ hinzu; so bleiben sie weiterhin als Konstanten des Basiswertes auch für die funktionale Teilsprache zugänglich.

Logisch

Man betrachte die Grammatik der logischen Teilsprache: ein Programm hat immer die Form $R_1 ... R_n A$, wobei R_1 bis R_n die Wissensbasis aus n Klauseln und A eine Anfrage an diese

Basis darstellt; dies bringt einen großen Nachteil mit sich: da Anfrage und Wissensbasis eine Einheit bilden, ist es unmöglich, eine andere Anfrage an eine Wissensbasis zu stellen, als die mit ihr zusammen geschriebene; dies wiederum ist kaum im Sinne der Programmcodewiederverwendbarkeit, weshalb es sinnvoll scheint, Wissensdatenbank und Anfrage als separate Unterprogramme zu definieren, was noch einen Vorteil mit sich bringt: eine Anfrage könnte so an eine Vereinigung mehrerer Wissensdatenbanken gestellt werden. Werde also die folgende Regel der Grammatik der logischen Sprache getilgt:

$$\langle P \rangle \longrightarrow \langle C \rangle_1 ... \langle C \rangle_n \ \langle A \rangle$$

An ihre Stelle trete dafür diese:

$$\langle BASE \rangle \longrightarrow \textbf{base name says } \langle C \rangle_1 ... \langle C \rangle_n \textbf{ end}$$

Sie ermöglicht eine eigenständige Definition einer Wissensbasis durch einen Stringbezeichner **name** und eine Liste von Regeln $\langle C_1 \rangle$ bis $\langle C_n \rangle$.

Da eine Wissensbasis allein nicht ausführbar ist, gilt es jetzt, Anfragen so definieren zu können, dass diese an *eine* oder *mehrere* Wissensdatenbanken gestellt werden können; zudem müssen Anfragen das aufrufbare Element logischer Programme sein, weshalb sie in der Form eines Funktionsaufrufs gestellt werden können müssen. Weiter stellt sich die Frage, was eine aufgerufene Anfrage zurückzugeben hat; es liegen dabei zwei Möglichkeiten nahe: es könnte entweder eine vollständige Liste der Werte aller gebundenen Variablen, oder eine Variable könnte ausgewählt und nur der an sie gebundene Wert zurückgegeben werden; da die erste Option mittels Strukturen realisiert werden kann, werde sich für die zweite entschieden. So soll die Produktionsregel für Anfragen durch die folgende ersetzt werden:

$$\begin{aligned} \langle QUERY \rangle \longrightarrow\ & \textbf{query name } (\ \textbf{param}_1 \text{ ,... , } \textbf{param}_n\ ) \\ & [\ \textbf{asks base}_1 \text{ ,... , } \textbf{base}_m\ ] \\ & \textbf{for X} \\ & ?- \mathbf{G}_1 \text{ ,... , } \mathbf{G}_n \\ & \textbf{end} \end{aligned}$$

Diese ermöglicht, eine mit dem Namen **name** benannte Anfrage an m beim Namen $\mathbf{base_1}$ bis $\mathbf{base_m}$ genannte Wissensdatenbanken abzusetzen, wobei bei der Ausführung die Symbole $\mathbf{param_1}$ bis $\mathbf{param_n}$ die dem Aufruf der Anfrage übergebenen Argumentwerte annehmen;

anschließend wird der an die Variable **X** gebundene Wert zurückgegeben. Dabei sei die **asks**-Klausel optional: das heißt eine Anfrage kann an *keine* Wissensbasis gestellt, also als *eigenständiges* Unterprogramm gerufen werden. Eine mögliche eigenständige Anfrage wäre die in Abbildung 2.3 beschriebene; sie bände jeweils 1, 2 und 3 an die Variable X, womit ihr Aufruf ein Ausdruck mit drei möglichen Werten wäre.

```
query foo() for X
    ?- X ~ 1; X ~ 2; X ~ 3
end
```

Abbildung 2.3: Eine Anfrage als eigenständiges Unterprogramm

Globaler Definitionsbereich

Endlich definiere man noch den *globalen Definitionsbereich*, der *globale Variablen* und *Unterprogramme* zu einem Programm zusammenfasst. Betrachtet man die imperativen Unterprogramme, so ist dort Variablen- und Programmdefinition voneinander getrennt; so werde dieses Schema, der Einfachheit wegen, beibehalten. Die folgende Grammatik definiert ein Programm des Namens **name**, mit n globalen Variablendefinitionen $\langle D \rangle_1$ bis $\langle D \rangle_n$ und m Unterprogrammen $\langle UPROG \rangle_1$ bis $\langle UPROG \rangle_m$:

$$
\begin{aligned}
\langle PROG \rangle &\longrightarrow \textbf{program name with } \langle D \rangle_1 \text{ ;... ; } \langle D \rangle_n \textbf{ begin } \langle UPROG \rangle_1 \text{... } \langle UPROG \rangle_m \textbf{ end} \\
\langle UPROG \rangle &\longrightarrow \langle PROC \rangle \\
&\quad | \ \langle LAMDBA \rangle \\
&\quad | \ \langle QUERY \rangle \\
&\quad | \ \langle BASE \rangle
\end{aligned}
$$

Eines der Unterprogramme muss dabei den Namen *main* tragen, um später einen Einstiegspunkt für die Ausführung zu haben.

2.2.6 Implementierung

Letztendlich stellt sich die Frage der Repräsentation der Maschinenprogramme in der Implementierung; dabei liegt nahe, dass abstrakte Syntaxbäume, implementiert als algebraische Datentypen, Programme repräsentieren können; es ist jedoch eines zu beachten: da von konkreten Operatoren abstrahiert und lediglich benutzerdefinierte Funktionen diese stellen, können in

diesen Funktionen eventuell Prüfungen der Operanden vorgenommen werden, denen diese eventuell nicht standhalten; da man in solchen Fällen gern eine Fehlermeldung ausgeben möchte und die Berechnung ohnehin im Kontext einer Zustandsmonade stattfindet, empfiehlt es sich, die Methode *fail* der Monadenklasse zu nutzen [14]; dafür aber muss das Ergebnis einer Operatorenanwendung, wie auch das von Prozeduraufrufen, ebenfalls im Monadenkontext stehen; da im übrigen von Operatoren beliebiger Stelligkeit ausgegangen wird, ergibt sich, dass diese im Syntaxbaum als Funktionen des Typs `[Basic]` $\rightarrow$ `m Basic` gehalten werden können.

Konkret lässt sich dabei der Syntaxbaum implementieren, wie in den Abbildungen B.1, B.2, B.3 und B.4 in Anhang B auf Seite 43 beschrieben; dieser weicht dabei etwas von den gefundenen Grammatiken ab, enthält aber die selben Konstrukte.

2.3 Die Maschine

Final gilt es jetzt, eine *Zustandsmonade*, *das Grammateion*, zu entwickeln, die eine die hergeleitete Maschinensprache ausführende Maschine simuliert; beginnend mit der Monade selbst, werde mit der Definition ihres Zustandes fortgefahren und darauf mit der Darstellung des auszuführenden Programms.

2.3.1 Die Monade

Zustandsbehaftete Berechnungen lassen sich in *Haskell* am elegantesten eine *Zustandsmonade* verwendend durchführen; so werde das Grammateion als solche zuerst in ihrer einfachsten Form definiert, und notfalls erweitert; so ergibt sich die folgende Implementierung:

```
1 data Grammateion state a = Grammateion
2     { runGrammteion :: state -> (a, state) }
```

Sie kapselt dabei Funktionen, welche einen *Ausgangszustand* auf einen aus diesem Zustand gewonnen Wert sowie einen *Folgezustand* abbilden; diese simple Definition birgt jedoch einen Nachteil: bedenkt man, dass während der Ausführung einer Programmiersprache allerlei Fehler auftreten können, seien es unbekannte Bezeichner oder Typfehler, die die verwendete dynamische Typisierung der *Maschinensprache* erst zur Laufzeit ausgäbe, so scheint es sinnvoll, bei solchen Fehlern die Berechnung terminieren und eine Fehlermeldung zurückgeben zu können; bei dieser einfachen Zustandsmonadenimplementierung ist dies aber nur mit der vordefinierten Funktion *error* [15] möglich, die aber die Programmausführung selbst beendet; dies wiederum

ist kaum von Vorteil, da genannte Fehler eventuell aufgefangen und behandelt werden können. Eine Alternative fände sich im Typ *Either*, einem Summentyp mit zwei Konstruktoren, der im Fehlerfall Fehlermeldungen und im Normalfall das Berechnungsergebnis halten kann. So werde die Monade wie folgt erweitert:

```
1 data Grammateion state a = Grammateion
2     { runGrammteion :: state -> Either String (a, state) }
```

Ein weiteres Problem stellt sich, betrachtet man die Interaktion mit dem Nutzer späterer Programme: da die Maschinensprache nichtdeterministisches Verhalten zeigt, sprich, es mehrere Programmverläufe gibt, ergeben sich auch mehrere Endergebnisse. Dieses Problem ließ sich jedoch, wie bei *Prolog*, durch die schlichte Frage lösen, ob der Nutzer mit der gelieferten Lösung denn zufrieden sei. Dazu jedoch ist es nötig, eine Tastatureingabe zu tätigen; da Ein- und Ausgabe, welche bisher noch außer Acht gelassen wurden, ohnehin ein nette Programmiersprachenfunktion darstellen, bedarf es der Verwendung der *IO* Monade; werde diese also ebenfalls in die Typdefinition eingeführt:

```
1 data Grammateion state a = Grammateion
2     { runGrammteion :: state -> IO (Either String (a, state)) }
```

Final bedarf es noch eines Gedankens: *an welcher Stelle hält die Monade das auszuführende Programm, also den zu interpretierenden Syntaxbaum?* Ihn in den Zustand selbst einzubetten wäre möglich, aber ungeschickt, da er während der Laufzeit *nicht* veränderbar sein sollte. Eine Monade, die eine *nur lesbare* Information trägt, wäre die *Readermonade* [17]; werde also auch das Grammateion um eine *Readerfunktion* erweitert:

```
1 data Grammateion dict state a = Grammateion
2     { runGrammteion :: dict -> state -> IO (Either String (a, state)) }
```

Jetzt gilt es lediglich, wie in Abbildung 2.4, nebst der Monadenklasse *Monad*, einige Typklassen für die Monade zu implementieren: *Functor* und *Applicative* zur optimalen Nutzung der Monade selbst; *Alternative* zur Fehlerbehandlung; *MonadState*, *MoandReader* und *MonadIO* zur optimalen Nutzung aller Erweiterungen.

Um später mehr Übersichtlichkeit im Code zu haben, empfiehlt es sich ein Typsynonym einzuführen, das die definierte Monade samt beider noch zu definierender Typen, *State* für den Zustand und *Dict* das Programm, abkürzt:

```
data Grammateion dict state a = Grammateion
    { runGrammteion :: dict -> state -> IO (Either String (a, state)) }

instance Monad (Grammateion d s) where ...
instance Functor (Grammateion d s)  where ...
instance Applicative (Grammateion d s) where ...
instance Alternative (Grammateion d s) where ...
instance MonadState s (Grammateion d s) where ...
instance MonadReader d (Grammateion d s) where ...
instance MonadIO (Grammateion d s) where ...
```

Abbildung 2.4: Implementierte Typklassen

```
type G = Grammateion Dict State
```

2.3.2 Der Zustand

Als nächstes ist der *Maschinenzustand* zu definieren; so war während der Herleitung der Maschinensprache von *vier zu speichernden Objektarten* die Rede: von lokalen und globalen *symbolische Variablen* [5], *Abschlüssen* [6], *Rücksetzpunkten* [7] und einem *persistenten Register* [8]; werde also als nächstes für jede dieser vier Objektarten eine Möglichkeit der Unterbringung im Maschinenzustand ermittelt.

Symbolische Variablen

Eine einfache Möglichkeit symbolische Variablen zu verwalten ist eine Symboltabelle [9] die wie folgt implementiert werden könnte:

```
type Ident = String
type Frame = [(Ident,Basic)]
```

Auf deren Werte könnte mit *lookup* [18] lesend und mit Hilfe einer *map* verwendenden Funktion schreibend zugegriffen werden. Durchsucht man die Haskellbibliotheken aber etwas weiter

[5] siehe Abschnitt 2.2.1
[6] siehe Abschnitt 2.2.2
[7] siehe Abschnitt 2.2.5
[8] siehe Abschnitt 2.2.5
[9] werde eine solche *Rahmen* oder englisch *Frame* genannt, nach den *Kellerrahmen* in [1]

findet sich in [19] der Typ *Map*, der auf solcherlei Strukturen spezialisiert ist. So werde auch die Typdefinition der Rahmen wie folgt geändert:

```
1 type Frame = Map Ident Basic
```

Dies ist jedoch noch nicht genug, bedenkt man, dass *globale* und *lokale* Variablen zu verwalten sind; für globale Variablen reichte ein solcher einfacher Rahmen aus; um jedoch bei Unterprogrammaufrufen die alten lokalen Variablen nicht zu verlieren, scheint, wie bei allen in [1] definierten Maschinen, eine kellerartige Speicherung der Rahmen sinnvoll; also werde jetzt, nach genannten Kriterien, ein Typ definiert, der als Speicher für *globale* und *lokale* symbolische Variablen fungieren soll:

```
1 data Stack = Stack
2     { global :: Frame
3     , locals :: [Frame] }
```

Abschlüsse

Der nächste zu entwerfende Speicher ist jener, der zur Lazy-Evaluation der funktionalen Ausdrücke Abschlüsse, beziehungsweise deren ausgewertete Versionen, verwaltet. Dieser Speicher muss, wie in Abschnitt 2.2.2 definiert, Zeiger, in Form von Integerwerten, auf Haldenobjekte abbilden. Ein Haldenobjekt ist dabei entweder ein *leerer Abschluss*, wie er bei der Auswertung von *letrec* Ausdrücken nötig ist [1], ein *ausgewerteter Abschluss* oder ein *frischer Abschluss*; derartiges lässt sich leicht als Summentyp definieren, was folgend getan werde:

```
1 data HeapObj
2     = Empty
3     | Basic (CoreLambda G)
4     | Closure (CoreLambda G)
```

Der Name des Konstruktors *Basic* ist dabei dem des entsprechenden Haldenobjects der *MaMa* in [1] entlehnt, doch er vereine die dort definierten Objekte *FunVal* und *Basic* insofern, als dass er für einen Ausdruck stehe, der nicht weiter vereinfacht werden kann; dazu zählen auch unterversorgte Lambdaabstraktionen. Werde die Halde, als eine Art Symboltabelle, auch mit Hilfe des *Map* Typen implementiert:

```
1 type Pointer = Int
```

```
2 data Heap = Heap
3     { next :: Int
4     , heap :: Map Pointer HeapObj }
```

Das Feld *next* halte dabei die Information, welche die nächste freie Haldenzelle ist; wird eine Haldenzelle alloziert, so erhöhe sich *next* um 1.

Rücksetzpunkte

Rücksetzpunkte müssen, zu einem Zeitpunkt, zu dem es mehrere alternative Programmverläufe gibt, den Zustand, sowie diese Verläufe sichern; es genügt dabei, den *nächsten* möglichen Verlauf zu sichern, wenn dieser vor dessen eigentlichem Beginn seinen Nachfolger sichert. Geht man davon aus, dass ein Programmverlauf damit endet, dass dem Nutzer ein mögliches Ergebnis seines Programms präsentiert und wie in beispielsweise *SWI-Prolog* auf dessen Reaktion gewartet wird, also das Ausgeben des Ergebnisses und das Reagieren auf einen Tastendruck mit Backtracking oder Termination, so scheint es sinnvoll, wäre der Typ einer solchen Aktion *G ()*; sie kann dabei ähnlich einer Prozedur mit dem Rückgabetyp *void* in *C* oder ähnlichen Sprachen betrachtet werden.

Bezüglich des zu sichernden Zustands stellt sich die Frage, welche Elemente dessen zu sichern sind; außen vor bleiben dabei gewiss das die Rücksetzpunkte haltende Zustandselement sowie das persistente Register. Recht sicher scheint, dass die symbolischen Variablen gesichert werden müssen, da diese essenziell den Programmfluss steuern. So verbleibt die Halde: gegen das Sichern dieser spricht, dass ein persistent gespeicherter Haldenzeiger nach dem Backtracking eventuell ungültig wäre; weiter spricht dafür, die Halde nicht zu sichern, dass eine allozierte Zelle für einen Programmverlauf ohne Zeiger auf diese unerreichbar ist und somit den Programmfluss nicht beeinflusst. Es ergibt sich also, dass es genügt, die symbolischen Variablen zu sichern; sei ein *Rücksetzpunkt* also wie folgt definiert:

```
1 type BacktrackPoint = (Stack, G ())
```

Der Speicher der Rücksetzpunkte, die *Spur* [10], kann dabei als einfacher Keller, beziehungsweise als Liste, welche praktisch einen Keller repräsentiert, implementiert werden. Geschehe dies also mit folgendem:

```
1 data Trail = Trail { btps :: [Backtrackpoint] }
```

[10] vergleiche [1]

Das persistente Register

Um das persistente Register zu implementieren, wird dem Zustand lediglich ein Feld für einen Basiswert zugeordnet.

Der Zustandstyp

Final ergibt sich aus alledem der Typ *State*, welcher wie folgt implementiert werde:

```
1 data State = State
2     { stack      :: Stack
3     , heap       :: Heap
4     , trail      :: Trail
5     , persistent :: Basic }
```

2.3.3 Das Programm

Das *Programm* selbst, ist bereits definiert als der abstrakte Syntaxbaum; für den Typen *Dict* bedarf es jedoch lediglich des zweiten Teils, also der *Unterprogramme*, da die globalen Variablen schon in der Symboltabelle untergebracht sind. So ergibt sich folgende Definition:

```
1 data Dict = Dict [(Ident,Subprogram G)]
```

2.3.4 Resümee

Die definierte Monadenimplementation *Grammateion* bietet jetzt alle nötigen Funktionen, um die definierte Maschinensprache ausführen zu können.

2.4 Gedanken zur Implementierung

Bevor es an die Implementierung selbst geht, sollten sich dazu noch Gedanken gemacht werden; allen voran bezüglich der Rückgabesemantik von Unterprogrammaufrufen.

2.4.1 Rückgabesemantik

Es wurde angenommen, ein Funktionsaufruf könne mehrere Werte annehmen, wie dies in *Curry* der Fall ist; es stellt sich jedoch die Frage: *wie sind mehrere Funktionsergebnisse an das aufrufende Unterprogramm zurückzugeben?* Eine naheliegende Möglichkeit wäre, die Ergebnisse als *Liste* zurückzugeben und den folgenden Programmverlauf, abstrahiert vom Rückgabewert, mit Hilfe der Funktion *mapM* beziehungsweise *forM* auf jeden Rückgabewert anzuwenden. Betrachte man dazu das Maschinensprachenprogramm in Abbildung C.1 [11]; der Aufruf *inf()* kann dabei alle Werte von 0 an annehmen und ein Aufruf von *pair(x)* die Werte aller Paare (x, i) mit $i \in \mathbb{N}_0$; das Programm sollte dem Nutzer folglich alle Paare (x, y) mit $x, y \in \mathbb{N}_0$ anbieten.

Werde das Programm jetzt, um die Interpretationsstrategie mit *forM* zu testen, in das in Abbildung C.2 gezeigte Programm umgesetzt. Wird es kompiliert und ausgeführt, so erhält man wider erwarten *keine* Anfrage, ob ein Ergebnis gefällt oder nicht. Sieht man sich den Code genauer, so kann man finden, dass *pair* alle seine unendlich vielen Ergebnisse zusammensammelt und dann erst an *main* weiterreicht, was, der unendlichen Liste *inf* wegen, nie geschehen wird.

Es gibt jedoch eine Möglichkeit, zu einem Ergebnis zu kommen, arbeitet man mit unendlich vielen Werten. Man kann, anstatt den Wert des Funktionsaufrufs zurückzuerwarten, eine Funktion übergeben, die auf das Ergebnis des Aufrufs angewandt werden soll. Diese Technik nennt sich *Continuation Passing* [8] und soll in Abbildung C.3 auf das Programm angewandt werden. Ein Aufruf der Funktion *inf* erwartet so noch eine Funktion, die einen *einzelnen* Integerwert erwartet und eine Aktion ausführt die als *Rücksprungadresse* fungiert; diese Funktion wird dabei auf *jeden* der Listenwerte angewandt; in Abbildung C.4 wird gezeigt, dass, wird *Continuation Passing* genutzt, *ask*, auch bei unendlichen Ergebnissen, garantiert aufgerufen wird, da der Ausdruck in diesem Programm äquivalent ist zum folgenden:

```
1 sequence_ [ask (a,r) | a <- [0..], r <- [0..]]
```

Ein weiterer Vorteil dessen ist, dass die *Rücksprungfunktion* implementiert werden kann, als erhielte sie nur einen einzigen Wert; dies ist sehr nah an der gedachten Semantik der Maschinensprache, weshalb diese Technik bei der Implementierung des Interpreters Verwendung finden soll.

[11] Anhang C, Seite 45

2.4.2 Abstraktion von der konkreten Zustandsmonade

Ein weiterer Gedanke sei der Architektur der Software selbst gewidmet: da es von großem Vorteil ist, Programmteile möglichst unabhängig voneinander implementieren zu können, kann sich einem die Frage stellen, ob es nicht eventuell möglich ist, die Interpretation unabhängig von einer konkreten Monadenimplementierung zu entwickeln: tatsächlich bietet Haskell die Möglichkeit, über die Definition von *Typklassen* Funktionen bereitzustellen, die, setzt man voraus, dass die verwendeten Typen diese implementieren, unabhängig von der jeweiligen Implementierung verwendet werden können; gäbe es jetzt eine oder mehrere solcher Typklassen, die die Monadenklasse erweiternd Funktionen für die Interpretation der Maschinensprache bereitstellten, so gewänne man daraus drei Vorteile: erstens könnte man, unabhängig von der konkreten Implementierung der Maschine, die Interpretation optimieren; zweitens könnte man die Maschinenimplementation unabhängig von der Interpretation verbessern; es müsste lediglich sichergestellt werden, dass die neue Implementierung der Klasse für die Interpretationsfunktionen das gleiche Verhalten zeigt; zudem stellten sie eine Art Anleitung dar, wie eine Monade zur Interpretation der Maschinensprache zu implementieren wäre. All dies spricht dafür, die Software derart zu strukturieren. Dabei mag auffallen, dass die einzelnen Unterprogrammarten unterschiedliche Zugriffsfunktionen benötigen, was ein weiteres Aufspalten der Typklassen ermöglicht, deren Implementierung die der Klasse voraussetzt, die die gemeinsam genutzten Funktionen stellt.

Werden also vier Typklassen definiert:

```
1 class (Monad m, Alternative m) => CoreGeneral m where ...
2 class CoreGeneral m => CoreImperative m where ...
3 class CoreGeneral m => CoreFunctional m where ...
4 class CoreGeneral m => CoreLogical m where ...
```

Sollen diese Klassen während der Implementierungsarbeit mit Methoden versehen werden, die der Interpretation als Schnittstelle zu der sie ausführenden Monade dienen; diese sei hier das *Grammateion*, mag aber jede sein, die dessen Verhalten simulieren kann.

2.5 Resümee

Mit diesen wichtigsten Charakterisierungen der zu implementierenden virtuellen Maschine, sowie der Orientierung der *Interpretation* an den Übersetzungs- beziehungsweise Interpretati-

onsverfahren, wie sie in [1] und [10] beschrieben wurden, sollte es möglich sein, diese tatsächlich zu implementieren.

3 Grammata

Letztendlich ist es gelungen, eine virtuelle Maschine zur Ausführung einer Maschinensprache, die als Zielsprache einer polyparadigmatischen Programmiersprache dienen kann, zu entwerfen; da auch bereits wichtige Kernpunkte zu Implementierung geklärt wurden, ist es ebenfalls gelungen die Maschine samt interner Darstellung ihrer Maschinensprache zu implementieren. Des weiteren galt es darauf, eine erste Sprache für das *Grammateion* zu entwickeln; dabei bot sich eine externe Darstellung der Maschinensprache selbst an, die letztendlich in der Sprache *Grammata*, als leicht erweiterte Form der Maschinensprache, implementiert wurde; diese soll in diesem Kapitel an Beispielen erläutert werden, die die Möglichkeiten betonen sollen, die diese Implementierung bietet.

3.1 Syntax

Im wesentlichen orientiert sich die Syntax an der Grammatik, die in Abschnitt 2.2 zur Maschinensprache ermittelt wurde; es gibt jedoch einen Unterschied bezüglich der Schreibweise symbolischer Bezeichner: um diese im Funktionalen und Imperativen von Strukturen unterscheiden zu können, beginnen sie als Wert immer mit einem *'$'*. Des weiteren wurden alle gängigen arithmetischen Operatoren implementiert, inklusive der Boole'schen und der Vergleichsoperatoren, sowie der unären Operatoren *'.'*, um auf den Listenkopf, und *'%'*, um auf den Listenrest, zugreifen zu können, und des binären Operators *':'*, um ein Element an das vordere Ende einer Liste anzuhängen. Die direkte Syntax zur Konstruktion einer Liste ist dabei *[]* für die leere Liste und $[e_1,...,e_n]$ für eine Liste mit n Elementen; diese Listen werden dabei mit den beiden Strukturen *cons* und *nil* konstruiert.

3.2 Noch nicht implementierte Funktionen

Da die Sprache vorerst nur zu Demonstrationszwecken implementiert wurde, ermangelt sie noch einiger Funktionen: zu diesen gehört, zum Beispiel, das *Pattern Matching*, das aber mit Hilfe

logischer Unterprogramme simuliert werden kann, wie im Beispielprogramm in Abbildung D.1 gezeigt wird; ein andere, relativ leicht implementierbare Funktion wären *Arrays*, wobei ein Array der Länge n durch eine Struktur $arr\,(e_1, ...e_n)$ dargestellt und der *Lesezugriff* auf ein Element als binäre arithmetische Operation implementiert werden könnte; der direkte *Schreibzugriff*, wie in C oder ähnlichen Sprachen, müsste dabei wohl simuliert werden durch eine Anweisung $arr := write(\$i, \$x, \$arr)$, wobei i der zu beschreibende Index, x der neue Wert dieser Stelle, arr das Array, und $write$ als ternärer arithmetischer Operator zu implementieren wäre.

Eine weitere noch hinzuzufügende Funktion wäre eine Art *Postprozessor* für die Ausgabe, die in der mit dieser Arbeit abgegebenen Version lediglich die interne Darstellung der Daten ausgibt; dies ist aber der Demonstration selbst sehr zuträglich.

3.3 Beispielprogramme

Es folgen nun einige Beispielprogramme zur Erläuterung der Sprache *Grammata*: zum einen ein einfaches Beispielprogramm, das wenig praktischen Sinn hat, jedoch die Funktion des persistenten Registers demonstrieren sollen, gefolgt von einer Implementation des *Quicksort* Algorithmus, welcher funktionale und imperative Unterprogramme nutzt; begonnen werde jedoch mit der Implementierung von Zugriffsfunktionen auf Strukturen via logischer Anfragen. Der Code der Programme befindet sich dabei in Anhang D ab Seite 49.

3.3.1 Zugriffsoperationen mit Anfragen

Für den Zugriff auf die Werte anderer Strukturen als die vordefinierten Listen kann man in Grammata Zugriffsfunktionen mit Hilfe von Anfragen realisieren; in Abbildung D.1 wird so die Funktionen *fst*, zum Zugriff auf das erste Element eines Paares, durch Unifikation implementiert; zudem realisiert die Anfrage *setFst* die *Manipulation* des ersten Wertes eines Paares. Die Anfrage *tf* sorgt dabei lediglich dafür, dass jeder der beiden möglichen Anweisungszweige ausgeführt wird.

3.3.2 Stammbaum

Sei gegeben ein Stammbaum, repräsentiert als Wissensbasis. Gesucht sei eine *Liste aller Enkelkinder* einer Person. Hierfür kann gut die Funktion des persistenten Registers genutzt werden, wie

es im Programm *Stammbaum* in Abbildung D.2 geschieht. Die Anfrage *enkel* sucht ein Enkelkind Peters und bindet dessen Namen jeweils an *X*, beziehungsweise an *false*, gibt es kein weiteres Suchergebnis; wäre sie der Einsteigspunkt, früge das Programm, wie in einem Prologsystem bei jedem Ergebnis ab, ob weiter gesucht werden soll; die Hauptprozedur jedoch sammelt ein jedes Ergebnis in einer Liste im *persistenten Register* und erzwingt das Backtracking, solange, bis es keine Ergebnisse mehr gibt; darauf wird der Wert im persistenten Register zurückgegeben.

3.3.3 Quicksort

Der *Quicksortalgorithmus* kann in Grammata wie in Abbildung D.3 implementiert werden, wobei die *Listenkonkatenation* mit Hilfe der Prozedur *concat* iterativ implementiert wurde. Das Unterprogramm *filter* nutzt dabei die Möglichkeit, funktionale Ausdrücke als Argumente übergeben bekommen zu können. Zu beachten ist dabei aber der Unterschied zwischen Unterprogrammaufrufen, die ihre Argumente als Tupel übergeben bekommen, und *Funktionsapplikationen*, die der Syntax des Lambdakalküls und vieler funktionaler Programmiersprachen folgen.

4 Konklusion

4.1 Zur Implementierungsarbeit

Die im Zuge dieser Arbeit implementierte Software umfasst *Sprachdefinition* und *Parser* der Sprache *Grammata*, den *Übersetzer* für diese in die interne Darstellung der Maschinensprache, sowie, selbstverständlich, die virtuelle Maschine selbst; sie wird dabei in eigenem Interesse weiterentwickelt und verbessert, lässt diesbezüglich aber noch einigen Spielraum, da gerade Effizienz bei der Entwicklung bisher kaum eine Rolle spielte; daher ist eine jede Interessierte und ein jeder Interessierter herzlich dazu eingeladen an diesem Entwicklungsprozess teilzuhaben! Die Software hat zu diesem Punkt einen Umfang von rund 2000 bereinigter Zeilen *Haskell* Quellcodes [1], ist unter der *GNU Public License Version 3* veröffentlicht und verfügbar auf *GitHub* unter `https://github.com/SRechenberger/grammata` .

4.2 Fazit

Mit der Implementierung und Ausführung der Sprache *Grammata* wurde das Ziel dieser Arbeit erreicht: es existiert eine virtuelle Maschine, *das Grammateion*, und mit ihr ein *Implementierungsansatz* für andere ähnliche oder gleichartige Maschinen, deren Ziel es ist, Programmiersprachen auszuführen, die nebst rein *logischer*, *funktionaler* oder *imperativer* Programmierung, auch Mischformen dieser, sogar mit zusätzlichen Funktionen, wie sie die reinen Paradigmata oft nicht zuließen, darstellen. Wie bereits genannt, bedarf es der Implementierung noch an einiger Optimierung bezüglich ihrer Effizienz; darüber hinaus wurde sie so implementiert, dass es möglich ist, weitere Unterprogrammarten hinzuzufügen, sowie die ausführende Monade selbst relativ leicht gänzlich neu zu implementieren. Eine Steigerung der Effizienz ließe sich so zum Beispiel dadurch erreichen, anstatt einer einfachen Zustandsmonade, eine Maschine zu entwickeln, die vermehrt auf die *schmutzigen* Funktionen der *IO* Monade zugreift; zeigt eine solche Implementierung an seiner Schnittstelle zum Interpreter das gleiche Verhalten wie die jetzige,

[1] Ermittelt unter Verwendung von *David A. Wheelers SLOCCount*

so läuft die Interpretation der Sprache völlig unbeeinflusst davon. Ein wahrlich schwierigerer Schritt wäre der Versuch, die Maschinensprache, zum Beispiel, via *LLVM* [13] in Code realer Maschinen zu übersetzen, womit der Sprung vom Interpreter zum Compiler gelungen wäre; eine Herausforderung dabei könnten jedoch die frei definierbaren Operatoren sein.

Zum Schluss bleibt zu sagen, dass, wenngleich gewiss nicht alle Probleme der Synthesis der drei verschiedenen Programmierparadigmata gefunden und gelöst wurden, allein schon der funktionierenden polyparadigmatischen Programmiersprache *Grammata* wegen, der gefundene Ansatz vielversprechend scheint; so gilt es jetzt, sein Potenzial auszuloten und noch weiter auszubauen.

A Inhalt der CD

Auf der CD befindet sich das vollständige Resultat der geleisteten Implementierungsarbeit inklusive aller zu Installation der Software via Cabal nötigen Dateien. Dazu gehören die Beispielprogramme im Ordner *examples* und der Quellcode im Ordner *src*; darin befindet sich zum einen das Hauptmodul *Main*, zum anderen der Ordner und das Modul *Grammata*; dies enthält den Parser und die Sprachdefinition der Sprache Grammata im Ordner *Language*, den Übersetzer von Grammata in die Maschinensprache im Ordner *Interpreter* und das Grammateion, samt all seiner Elemente, im Ordner *Machine*. Zudem befindet sich auch eine digitale Version dieser Ausarbeitung darauf.

B Abstrakter Syntaxbaum

```
type Ident = String

type Program m = ([(Ident,CoreExpression m)], [(Ident, Subprogram m)])

data Subprogram m =
    = Imperative Ident [(Ident, CoreExpression m)] [Ident] [CoreStatement m]
    | Functional Ident (CoreLambda m) [Ident]
    | Logical Ident [Ident] (Maybe Ident) [Ident] [CoreClause]
    | Base Ident [CoreRule]
```

Abbildung B.1: Abstrakter *Maschinenprogramm*syntaxbaum

```
data CoreStatement m
    = Ident := CoreExpression m
    | IIf (CoreExpression m) [CoreStatement m] [CoreStatement m]
    | IWhile (CoreExpression m) [CoreStatement m]
    | IReturn (CoreExpression m)
    | ICall Ident [CoreExpression m]
    | IBackTrack
    | IKeep (CoreExpression m)

data CoreExpression m
    = IVar Ident
    | IVal Basic
    | IOp ([Basic] -> m Basic) [CoreExpression m]
    | IFunc Ident [CoreExpression m]
    | IRemind
```

Abbildung B.2: Abstrakter Syntaxbaum *imperativer Unterprogramme*

```
1  data CoreLambda m =
2        FVar Ident
3      | FConst Basic
4      | FIf (CoreLambda m) (CoreLambda m) (CoreLambda m)
5      | FOp ([Basic] -> m Basic) [CoreLambda m]
6      | FCall Ident [CoreLambda m]
7      | FLet [(Ident, CoreLambda m)] (CoreLambda m)
8      | FApp (CoreLambda m) [CoreLambda m]
9      | FAbs [Ident] (CoreLambda m)
10     | FKeep (CoreLambda m)
11     | FRemind
12     | FBackTrack
```

Abbildung B.3: Abstrakter Syntaxbaum *funktionaler Unterprogramme*

```
1  data CoreTerm
2      = Atom Basic
3      | Var LVar
4      | LFun Ident Int [CoreTerm]
5
6  data CoreGoal
7      = LPred Ident Int [CoreTerm]
8      | CoreTerm :=: CoreTerm
9
10 data CoreClause
11     = LOr [[CoreClause]]
12     | LGoal CoreGoal
13
14 data CoreRule = CoreGoal :- [CoreClause]
```

Abbildung B.4: Abstrakter Syntaxbaum *logischer Unterprogramme*

C Zur Rückgabesemantik

```
1  program Inf
2  begin
3      proc succ(x) does
4          return $x + 1;
5      end
6
7      base nat says
8          nat(0).
9          nat(succ(X)) :- nat(X).
10     end
11
12     query inf() asks nat for X ?-
13         nat(X)
14     end
15
16     proc pair(i) does
17         return p($i,inf());
18     end
19
20     proc main()
21     does
22         return pair(inf());
23     end
24 end
```

Abbildung C.1: Unendliche Anzahl von Ergebnissen

```
1  import System.IO
2  import Control.Monad
3  import System.Exit
4
5  ask :: Show a => a -> IO ()
6  ask x = do
7      putStrLn $ show x ++ "␣?"
8      answer <- getChar
9      putStrLn ""
10     case answer of
11         'y' -> exitSuccess
12         _   -> return ()
13
14 inf :: IO [Integer]
15 inf = return [0..]
16
17 pair :: Int -> IO [(Integer,Integer)]
18 pair i = do
19     rs  <- inf
20     rs' <- forM rs $ \r -> do
21         return (i,r)
22     return rs'
23
24 main :: IO [(Integer,Integer)]
25 main = do
26     hSetBuffering stdin NoBuffering
27     as <- inf
28     forM as $ \a -> do
29         bs <- inf
30         forM bs $ \b -> do
31             ask (a,b)
```

Abbildung C.2: Umsetzungsversuch mit *forM*

```haskell
import System.IO
import Control.Monad
import System.Exit

ask :: Show a => a -> IO ()
ask x = do
    putStrLn $ show x ++ "␣?"
    answer <- getChar
    putStrLn ""
    case answer of
        'y' -> exitSuccess
        _   -> return ()

inf :: (Integer -> IO ()) -> IO ()
inf ret = mapM_ ret [0..]

pair :: Integer -> ((Integer,Integer) -> IO ()) -> IO ()
pair i ret = inf $ \r -> ret (i,r)

main :: IO ()
main = do
    hSetBuffering stdin NoBuffering
    inf $ \a -> pair a ask
```

Abbildung C.3: Umsetzungsversuch mit *Continuation Passing*

```
1  -- Zu Zeigen:
2      main == sequence_ [ask (a,r) | a <- [0..], r <- [0..]]
3
4  -- Beweis:
5  main == inf $ \a -> pair a aks
6  -- inf ret = mapM_ ret [0..]
7       == mapM_ (\a -> pair a ask) [0..]
8
9  -- mapM_ f as == sequence_ (map f as)
10      == sequence_ (map (\a -> pair a ask) [0..])
11
12 -- pair i ret == inf $ \r -> ret (i,r)
13      == sequence_ (map (\a -> inf $ \r -> ask (a,r)) [0..])
14      == sequence_
15         (map (\a -> sequence_ (map (\r -> ask (a,r)) [0..])) [0..])
16
17 -- map f as == [f a | a <- as]
18      == sequence_
19         [sequence_ [ask (a,r) | r <- [0..]] | a <- [0..]]
20
21 -- sequence_ = foldr (>>) (return ())
22      == foldr (>>) (return ())
23         [foldr (>>) (return ()) [ask (a,r) | r <- [0..]] | a <- [0..]]
24
25 -- foldr f e [a,b,...,c] == a `f` b `f` ... `f` c `f` e
26 -- a >> b  == do {a;b}
27      == do (do ask (0,0)
28                ask (0,1)
29                ...
30                return ())
31            (do ask (1,0)
32                ask (1,1)
33                ...
34                return ())
35          return ()
36      == do ask (0,0)
37            ask (0,1)
38            ...
39            ask (1,0)
40            ask (1,1)
41            ...
42            return ()
43      == sequence_ [ask (a,r) | a <- [0..], r <- [0..]]
44      Q.E.D.
```

Abbildung C.4: Continuation Passing garantiert Aufruf von *ask*

D Beispielprogramme

```
1  program Zugriffsfunktionen
2  with
3      var s := pair(a,b);
4  begin
5      query fst(pair) for X ?- pair ~ pair(X,Y) end
6
7      query setFst(pair,val) for X ?-
8          pair ~ pair(Fst,Snd), X ~ pair(val,Snd)
9      end
10
11     query tf() for X ?-
12         X ~ true ; X ~ false
13     end
14
15     proc main()
16     does
17         if tf() then
18             return firstOf(fst($s),$s);
19         else
20             s := setFst($s, 1);
21             return modified(fst($s),$s);
22         end
23     end
24 end
```

Ausgabe:

```
firstOf(a,pair(a,b)) ?
n
modified(1,pair(1,b))
n
OK.
```

Abbildung D.1: Zugriffsoperationen mit Anfragen

```
1  program Stammbaum
2  begin
3      base stammbaum says
4          eltern(marta, hans, fritz).
5          eltern(marta, hans, felix).
6          eltern(marta, hans, anna).
7          eltern(julia, peter, siglinde).
8          eltern(julia, peter, helga).
9          eltern(siglinde, fritz, max).
10         eltern(siglinde, felix, petra).
11         eltern(helga, felix, david).
12         enkel(Gross,Enkel) :-
13             (Gross ~ Oma ; Gross ~ Opa),
14             eltern(Oma,Opa,Elter),
15             (Elter ~ Mutter ; Elter ~ Vater),
16             eltern(Mutter,Vater,Enkel).
17     end
18
19     query enkel() asks stammbaum for X ?-
20         enkel(peter,X) ; X ~ false
21     end
22
23     proc main()
24     with
25         var tmp;
26     does
27         keep [];
28         tmp := enkel();
29         if $tmp != false then
30             keep $tmp:remind;
31             backtrack;
32         else
33             return remind;
34         end
35     end
36 end
```

Ausgabe:

```
cons(david,cons(petra,cons(max,nil))) ?
n
OK.
```

Abbildung D.2: Programm Stammbaum

```
1  program QuickSort
2  begin
3      lambda filter(p,list) is
4          if $list == [] then
5              []
6          else
7              if ($p .$list) then
8                  .$list : filter($p, %$list)
9              else
10                 filter( $p, % $list )
11             end
12         end
13     end
14
15     proc concat(as,bs)
16     with
17         var tmp := [];
18     does
19         while $as != [] do
20             tmp := .$as : $tmp;
21             as := %$as;
22         end
23         while $tmp != [] do
24             bs := .$tmp : $bs;
25             tmp := %$tmp;
26         end
27         return $bs;
28     end
29
30     lambda quicksort(list) is
31         if $list == [] then
32             []
33         else
34             let
35                 $concat := \$as $bs . concat($as,$bs);
36                 $left := quicksort(filter( \$x . $x <= .$list , %$list));
37                 $right := .$list : quicksort(filter( \$x . $x > .$list ,%$list));
38             in
39                 $concat $left $right
40             end
41         end
42     end
43
44     lambda main() is
45         quicksort([6,2,5,22,7,8])
46     end
47 end
```

Ausgabe:

```
cons(2,cons(5,cons(6,cons(7,cons(8,cons(22,nil)))))) ?
y
OK.
```


Abbildung D.3: Programm Quicksort

Abbildungsverzeichnis

Literaturverzeichnis

[1] R. Wilhelm, H. Seidl, *Übersetzerbau - Virtuelle Maschinen*, 1. Auflage, Springer 2007, ISBN 978-3-540-49596-3

[2] U. Schöning *Theoretische Informatik - kurz gefasse* 5. Auflage; Heidelberg, Berlin: Spektrum, Akad. Verl., 2008, ISBN 978-3-8274-1824-1

[3] S. Antoy, M. Hanus *Curry - A Tutorial Introduction* `http://web.cecs.pdx.edu/~antoy/Courses/TPFLP/Tutorial.pdf` (15.01.2015)

[4] W. Jeltsch *A Taste of Curry* `https://jeltsch.wordpress.com/2013/04/27/a-taste-of-curry/` (15.01.2015)

[5] H. Barendregt, E. Barendsen *Introduction to Lambda Calculus* `http://ftp.cs.ru.nl/CompMath.Found/lambda.pdf` (28.1.2015)

[6] C. Ullenboom *Java ist auch eine Insel* `http://openbook.rheinwerk-verlag.de/javainsel/` (28.01.2015) ISBN 978-3-8362-1802-3 Rheinwerk Computing

[7] R. Ierusalimschy, *Programmieren mit Lua*, Aus dem Englischen übersetzt von D. Gherman, ISBN 3-937514-22-8; Englische Originalausgabe: Ṗrogramming in Lua̋(2nd ed.), ISBN 85-903798-2-5

[8] *Making Haskell programs faster and smaller* `http://users.aber.ac.uk/afc/stricthaskell.html#cps` (28.01.2015)

[9] B. O'Sullivan, D. Stewart, J. Goerzen *Real World Haskell* `http://book.realworldhaskell.org/read/` (26.12.2014).

[10] U. Schöning *Logik für Informatiker* 5. Auflage; Heidelberg, Berlin: Spektrum, Akad. Verl., 2000, ISBN 3-8274-1005-3

[11] A. M. Turing *Computability and λ-Definability* The Journal of Symbolic Logic, Volume 2, Number 4, Seiten 153-163, ISSN 00224812, Copyright © 1937 Association for Symbolic Logic

[12] *The Mono Runtime* http://www.mono-project.com/docs/advanced/runtime/ (28.01.2015)

[13] *LLVM* http://llvm.org/ (28.01.2015)

[14] *Hackage.Haskell.org Control.Monad,* http://hackage.haskell.org/package/base-4.7.0.2/docs/Control-Monad.html (02.01.2015)

[15] *Hackage.Haskell.org Prelude,* http://hackage.haskell.org/package/base-4.7.0.2/docs/Prelude.html#t:Double (08.01.2015)

[16] *Hackage.Haskell.org Control.Monad.State.Lazy* https://hackage.haskell.org/package/mtl-2.0.1.0/docs/Control-Monad-State-Lazy.html (20.01.2015)

[17] *Hackage.Haskell.org Control.Monad.Reader* https://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-Reader.html (23.01.2015)

[18] *Hackage.Haskell.org Data.List* http://hackage.haskell.org/package/base-4.7.0.2/docs/Data-List.html

[19] *Hackage.Haskell.org Data.Map* https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html

Name: Sascha Rechenberger Matrikelnummer: 755560

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Sascha Rechenberger