technische universität
dortmund

Bachelorarbeit

Simulation von Berechnungsmodellen innerhalb des Softwaresyntheseframeworks CLS

Marco Pennekamp

August 2017

Gutachter:
Prof. Dr. Jakob Rehof
M. Sc. Andrej Dudenhefner

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 14 für Software Engineering
http://ls14-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Softwaresyntheseframework CLS (Combinatory Logic Synthesizer) ermöglicht die Synthese von Programmen aus Komponenten, die in einer typisierten *kombinatorischen Logik* definiert werden. Es gibt *kombinatorische Ausdrücke*, die aus Komponenten zusammengesetzt werden können. Die Spezifikation eines zu synthetisierenden Programms liegt dabei als Typ vor, den der generierte kombinatorische Ausdruck erfüllen muss, d.h. *inhabitiert*. Das *Inhabitationsproblem* ist das zentrale Problem, welches von CLS praktisch gelöst wird. Formal schreiben wir $\Gamma \vdash e : \tau$, um auszudrücken, dass ein kombinatorischer Ausdruck e, gebildet aus den Komponenten in Γ, den Typ τ inhabitiert. Γ heißt im Folgenden *Repository*.

Das verwendete Typsystem ist ausdrucksstark genug, um innerhalb dessen Berechnungen durchzuführen. Dabei betrachtet man die Typen der Komponenten als Regeln für die Berechnung und die Suche nach einem Inhabitanten als die Berechnung selbst (siehe [7]).

Diese Eigenschaft erlaubt es, aus der theoretischen Informatik bekannte Berechnungsmodelle wie Automaten und Grammatiken in CLS zu simulieren. Die Eingabe der Simulation ist ein Typ, der die Eingabe des Modells kodiert. Bei der Simulation wird ein kombinatorischer Ausdruck gesucht, der diesen Eingabetyp inhabitiert. Findet sich ein solcher Ausdruck, so wird die Eingabe auch vom Modell akzeptiert. Beispielsweise kodiert der Eingabetyp bei der Simulation von DFAs das Eingabewort, das vom DFA erkannt werden soll.

Wir betrachten in dieser Bachelorarbeit ein Berechnungsmodell $\mathcal{M}$ anhand eines *allgemeinen Simulationsansatzes*. Für eine konkrete Instanz $\mathcal{A}$ von $\mathcal{M}$ soll ein Repository $\Gamma_{\mathcal{A}}$ erzeugt werden, das als Simulationsprogramm für $\mathcal{A}$ betrachtet werden kann. Sei $\text{rep}(w)$ der Typ, der eine Eingabe w kodiert. Für jede Eingabe w soll dann gelten:

$$\exists e(\Gamma_{\mathcal{A}} \vdash e : \text{rep}(w)) \iff \mathcal{A} \text{ akzeptiert } w \tag{1.1}$$

Der Vorteil der Simulation besteht darin, dass Softwareteile, die in einem der simulierbaren Modelle darstellbar sind, direkt in CLS umsetzbar sind. Das reine Akzeptanzverhalten der Modelle ist nützlich, um Programme auszusortieren, die einem Modell nicht

entsprechen. Wird eine Eingabe nicht akzeptiert – gibt es also keinen Ausdruck zu dem Eingabetypen – kann kein Programm gefunden werden. So werden nur Programme erzeugt, bei denen die Eingabe vom Modell akzeptiert wird. Wir bezeichnen diese Anwendung als das *Testen von Anforderungen*.

Auch der generierte kombinatorische Ausdruck ist von Nutzen. Er kann als Programm betrachtet werden, welches durch die Instanz des Berechnungsmodells zusammen mit dem Eingabetyp spezifiziert wird. Bei Automaten entspricht der Ausdruck beispielsweise den Transitionen, die für die Akzeptanz des Eingabeworts durchlaufen werden. Damit ist der Ausdruck eine Art Ausführungsplan. So kann mithilfe eines DFA, der die Bewegung einer Spielfigur modelliert, ein Ausdruck zu einer gewünschten Konfiguration generiert werden, z.B. zu einer Strecke mit Hindernissen. Der Ausdruck wäre dann ein Bewegungsplan für die Spielfigur. Diese Anwendung nennen wir *Codegenerierung*.

Zusätzlich stellt die Arbeit ein Entwurfsmuster für die Umsetzung von Berechnungsmodellen in CLS dar. Einerseits werden für die in der Arbeit betrachteten Modelle Verfahren vorgestellt, die eine Instanz eines solchen Modells in CLS umsetzen; Andererseits ist es auch denkbar, sich bei hier nicht behandelten Modellen an den vorgestellten Verfahren zu orientieren.

Die folgenden Berechnungsmodelle werden in der Bachelorarbeit behandelt:

- Deterministische endliche Automaten (DFAs)
- Nichtdeterministische endliche Automaten mit Epsilon-Transitionen (ε-NFAs)
- Reguläre Baumgrammatiken

Dabei sollen für jedes Berechnungsmodell folgende Ziele erreicht werden:

- Die **Grundlagen des Berechnungsmodells** werden geeignet dargelegt.
- Ein **Konstruktionsverfahren** für Repositories wird gemäß des allgemeinen Simulationsansatzes formuliert. Das Konstruktionsverfahren soll aus einer Instanz des Berechnungsmodells ein geeignetes Repository generieren.
- Die **Korrektheit und Vollständigkeit** des Konstruktionsverfahrens werden gemäß der Äquivalenz in (1.1) bewiesen.
- Das Konstruktionsverfahren wird auf konkrete **Beispiele** angewandt.

Man mag sich fragen, warum in dieser Arbeit Baumgrammatiken anstelle von kontextfreien Grammatiken betrachtet werden. Der Vorteil von Baumgrammatiken ist, dass Eingaben und damit Konfigurationen für die Simulation mithilfe von Bäumen strukturierter angegeben werden können als mit Zeichenketten.

Ein Nebenziel ist die *praktische* Umsetzung von Beispielen. Dabei sollen die in den Beispielen konstruierten Repositories mithilfe einer Implementation von CLS names cls-scala in der Programmiersprache Scala implementiert und an Beispieleingaben ausgeführt

werden. Damit lassen sich einerseits die Beispiele exemplarisch überprüfen und die Laufzeit feststellen, andererseits schlagen wir damit auch die Brücke von der theoretischen Arbeit zur praktischen Anwendung.

In **Kapitel 2** stellen wir die Grundlagen von CLS vor, mitsamt der kombinatorischen Logik und dem Inhabitationsproblem. In **Kapitel 3**, **4** und **5** werden jeweils die Berechnungsmodelle DFA, ε-NFA und Baumgrammatik auf Basis der oben genannten Ziele betrachtet. In **Kapitel 6** widmen wir uns der praktischen Simulation der Beispiele. In **Kapitel 7** werden die Ergebnisse der Arbeit evaluiert. Zuletzt schließen wir die Bachelorarbeit in **Kapitel 8** mit einem Fazit ab.

Kapitel 2

Grundlagen der Softwaresynthese

In diesem Kapitel werden die Grundlagen der Softwaresynthese mit kombinatorischer Logik und Intersektionstypen auf Basis von [7] dargelegt. Wir werden nach einem kurzen Überblick über CLS die notwendigen Formalismen einführen. Danach werden wir uns mit dem Inhabitationsproblem und dessen Entscheidbarkeit beschäftigen.

2.1 Einführung

Das in dieser Bachelorarbeit verwendete Softwaresyntheseframework **CLS** (Combinatory Logic Synthesizer) basiert auf einer *kombinatorischen Logik*, mit der typisierte Softwarekomponenten zu Programmen zusammengesetzt werden können. Insbesondere bietet CLS die Möglichkeit, *Intersektionstypen* zur semantischen Spezifikation von Softwarekomponenten zu nutzen, um synthetisierte Programme den gewünschten Anforderungen anpassen zu können. Da der Ansatz stark komponentenorientiert ist, wird diese Form der Softwaresynthese auch *Kompositionssynthese* genannt.

In [3] wird CLS benutzt, um verschiedene Variationen eines Solitaire Spiels auf Basis von modularen Komponenten zu generieren. Diese Komponenten stellen jeweils Features dar, die die Funktionsweise des Spiels verändern. Zur Generierung einer Variation werden nur gewünschte Features ausgewählt, die dann von CLS geeignet synthetisiert werden.

2.2 Kombinatorische Logik mit Intersektionstypen

2.2.1 Definition. Kombinatoren sind Funktionen, die applikativ zu **Ausdrücken** zusammengesetzt werden können. Solche Ausdrücke sind durch die folgende Grammatik gegeben, wobei IDs (bestehend aus Groß- und Kleinbuchstaben) für Kombinatorsymbole stehen:

$$\texttt{e} \rightarrow \texttt{id} \mid (\texttt{e}\ \texttt{e})$$

Die Operation (`e e`) nennen wir **Applikation**.

Konvention. Eine Applikation $((\texttt{K}\ e_1)\ e_2)$ kann $\texttt{K}\ e_1\ e_2$ oder $\texttt{K}(e_1, e_2)$ geschrieben werden.

Die *kombinatorische Logik* enthält als Operation nur die Applikation. Zusätzlich sind Kombinatoren und Ausdrücke *typisiert*. Wir definieren deshalb nun die Form der möglichen Typen.

2.2.2 Definition. Die **Typausdrücke** der kombinatorischen Logik lassen sich durch folgende Grammatik beschreiben:

$$\begin{aligned} \texttt{t} &\rightarrow \texttt{ctr} \mid \texttt{var} \mid \omega \mid \texttt{t} \rightarrow \texttt{t} \mid \texttt{t} \cap \texttt{t} \mid (\texttt{t}\ \{,\texttt{t}\}^+) \mid (\texttt{t}) \\ \texttt{ctr} &\rightarrow \texttt{id} \mid \texttt{id}(\texttt{t}\ \{,\texttt{t}\}^*) \\ \texttt{var} &\rightarrow \alpha \mid \beta \mid ... \end{aligned}$$

Die einzelnen Bestandteile haben folgende Bedeutungen, wobei τ_1, τ_2, τ_3 für beliebige Typausdrücke stehen:

- **Typkonstruktoren** `ctr` erzeugen neue Typen, wobei sie selbst beliebig viele Typen als Argumente annehmen können. Beispielsweise lässt sich ein Listentyp als $\texttt{List}(\tau_1)$ darstellen, wobei `List` der einstellige Typkonstruktor ist und τ_1 der Typ der Listenelemente. **Typkonstanten** werden als 0-stellige Typkonstruktoren modelliert.
- **Typvariablen** `var` können durch Typausdrücke ersetzt werden. Das macht es möglich, polymorphe Typen für Kombinatoren zu definieren. Beispielsweise könnte man einen `sort`-Kombinator definieren, der Listen mit beliebigen Elementen sortieren kann:

  $$\texttt{sort} : (\alpha \rightarrow \alpha \rightarrow \texttt{Bool}) \rightarrow \texttt{List}(\alpha) \rightarrow \texttt{List}(\alpha)$$

  Zu beachten ist hier, dass die Input- und die Output-Liste den gleichen Elementtyp haben, der an Stelle der α-Variable stehen würde. Ebenfalls muss die Vergleichsfunktion, die als erster Parameter übergeben wird, den genannten Elementtyp vergleichen können.
- Die **spezielle Typkonstante** ω ist ein Typ, für den im Kontext der *Subtyping-Relation* für alle Typen τ gilt, dass $\tau \leq \omega$ ist.
- **Funktionstypen** $\tau_1 \rightarrow \tau_2$ sind Typen von Funktionen, die ein Argument vom Typ τ_1 erwarten und einen Wert vom Typ τ_2 zurückgeben. Funktionen $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ mit mehr als einem Argument haben den Typ $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$, wir erlauben aber ersteres als abkürzende Notation.
- **Intersektionstypen** $\tau_1 \cap \tau_2$ beschreiben Ausdrücke, die sowohl den Typ τ_1 als auch den Typ τ_2 inhabitieren. So kann ein Typ $\texttt{List}(\texttt{Int}) \cap \texttt{Sorted}$ nur von *sortierten* Integer-Listen inhabitiert werden. Intersektionstypen können abkürzend $\tau_1 \cap \tau_2 \cap \tau_3$ geschrieben werden. Sie spielen eine besondere Rolle bei der *semantischen Spezifikation* in CLS.

- **Tupeltypen** $(\mathtt{t}_1, ..., \mathtt{t}_n)$ beschreiben n-stellige Tupel. Die Grammatik impliziert, dass $n \geq 2$ ist.

Notation. Hat ein Ausdruck e den Typ τ, so schreiben wir $e : \tau$.

2.2.3 Definition. Ein **Repository** ist eine endliche Menge Γ von Softwarekomponenten $\mathtt{F}_0, ..., \mathtt{F}_n$ mit den Typen $\tau_0, ..., \tau_n$. Wir schreiben $\Gamma = \{\mathtt{F}_0 : \tau_0, ..., \mathtt{F}_n : \tau_n\}$. Jede Komponente wird als *typisierter Kombinator* verstanden. Das Repository stellt somit die Kombinatoren bereit, aus denen kombinatorische Ausdrücke gebildet werden können.

2.2.4 Beispiel. Wir betrachten das Repository Γ_T, welches ein vereinfachter Ausschnitt aus einem Beispiel aus [7] ist:

$$
\begin{aligned}
\Gamma_T = \{ & \\
& \mathtt{track} : () \to ((\mathtt{R}, \mathtt{R}) \cap \mathit{Cartesian}, \mathtt{R} \cap \mathit{Time}) \cap \mathit{Pos} \\
& \mathtt{coord} : ((\mathtt{R}, \mathtt{R}) \cap \alpha, \mathtt{R}) \cap \mathit{Pos} \to (\mathtt{R}, \mathtt{R}) \cap \alpha \\
& \mathtt{cc2pl} : (\mathtt{R}, \mathtt{R}) \cap \mathit{Cartesian} \to (\mathtt{R}, \mathtt{R}) \cap \mathit{Polar} \\
\} &
\end{aligned}
$$

In dem Beispiel geht es um einen Location-Tracking-Service, bei dem unter anderem Koordinaten in verschiedenen Typen vorliegen können. Der Kombinator `track` erzeugt Werte eines Typs, der die Koordinaten und Zeit einer Position speichert. Die Intersektionstypen werden hier benutzt, um den *nativen* Typen wie $(\mathtt{R}, \mathtt{R})$ zusätzliche *semantische* Eigenschaften zu geben. So enthält $(\mathtt{R}, \mathtt{R}) \cap \mathit{Cartesian}$ die Information, dass *kartesische* Koordinaten vorliegen. Der Kombinator `coord` kann Koordinaten aus einer Position extrahieren. Dabei ist interessant, wie α als Typvariable benutzt wird: Die Extraktion der Koordinaten kann erstens unabhängig von dem semantischen Typ der Koordinaten stattfinden, da α nicht eingeschränkt ist. Zweitens wird der semantische Typ erhalten, indem das Ergebnis der Funktion $(\mathtt{R}, \mathtt{R}) \cap \alpha$ wieder durch α spezifiziert wird. Der Kombinator `cc2pl` wandelt kartesische in polare Koordinaten um, was im Typsystem anhand der semantischen Typen reflektiert wird.

Die folgende Definition von *Inhabitation* bezieht das Repository in die Bildung von kombinatorischen Ausdrücken mit ein.

2.2.5 Definition. Das kombinatorische **typing judgement** $\Gamma \vdash e : \tau$ sagt aus, dass durch Applikation von Kombinatoren in Γ ein kombinatorischer Ausdruck e gebildet werden kann, der den Typ τ hat. Wir sagen, dass der Ausdruck e den Typ τ **inhabitiert**.

2.2.6 Definition. Weiterhin gibt es **Regeln**, mit denen obige typing judgements konstruiert werden können. Diese sind in Abbildung 2.1 definiert und haben folgende Bedeutung:

$$\frac{}{\Gamma, X : \tau \vdash X : S(\tau)} \text{ (var)} \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \text{ ($\to$ E)}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash e : \tau_1 \cap \tau_2} \text{ ($\cap$I)} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \text{ ($\leq$)}$$

Abbildung 2.1: Regeln der kombinatorischen Logik mit Intersektionstypen (siehe [7], Figure 3).

- Die Regel (var) ermöglicht die Substitution von Typvariablen, indem die Funktion S Typvariablen innerhalb des Typen τ durch Typausdrücke ohne Variablen ersetzt.
- Mit ($\to$ E) wird die Applikation typisiert. Bei einer Applikation $(e\ e')$ des Kombinators $e : \tau \to \tau'$ muss das Argument e' den Typ τ inhabitieren, da der Funktionstyp von e solch einen Typ verlangt. Gleichzeitig wissen wir, dass $(e\ e')$ den Typ τ' inhabitiert.
- Intersektionstypen können mit ($\cap$I) konstruiert werden. Inhabitiert ein Ausdruck e sowohl den Typ τ_1 als auch den Typ τ_2, so inhabitiert e außerdem den Typ $\tau_1 \cap \tau_2$.
- Durch die Regel ($\leq$) wird die in [7] näher definierte *Subtyping-Relation* $\leq$ in die kombinatorische Logik eingebunden.

Notation. In dieser Arbeit gelten folgende Konventionen bezüglich der Notation von Ausdrücken und Typen im Rahmen der kombinatorischen Logik. Diese Konventionen finden sich auch in Anhang A.

- Freie Variablen (z.B. Teilausdrücke e) und Hilfsfunktionen (z.B. rep aus Definition 5.2.1) werden in der normalen mathematischen Schrift gesetzt: e, w, list, rep, usw.
- Konkrete Wörter werden unterstrichen, z.b. <u>abba</u>.
- Namen von Typen und Kombinatoren werden dagegen in einer Typewriter-Schrift gesetzt. In dem kombinatorischen Ausdruck `f a` sind sowohl `f` als auch `a` Kombinatornamen. In dem Typausdruck `Word`(list(w)) ist `Word` als Typkonstruktor zu lesen, list als Hilfsfunktion und w als freie Variable.
- Gibt es eine Variable a, zu der ein Typkonstruktor gehört, kann dieser Typkonstruktor als `a` bezeichnet werden. Davon wird beispielsweise in Definition 3.2.1 Gebrauch gemacht, wo zu einem beliebigen Zeichen $a_i \in \Sigma$ ein Typkonstruktor $\mathtt{a_i}$ verwendet wird.
- Freie Variablen, die anstelle von konkreten Typen stehen, werden immer als τ bezeichnet, beispielsweise τ, τ_1, τ_2, usw.
- Typvariablen werden immer als α, β, γ bezeichnet. Diese Zeichen kommen nie als freie Variablen vor. Gültige Typvariablen sind z.B. α_1, α_2 und β_1.

- Das Zeichen ε steht immer für das leere Wort, während ϵ der Typkonstante entspricht, die das leere Wort repräsentiert.

2.3 Das Inhabitationsproblem

Der vorherige Teil beschreibt, wie wir Repositories von typisierten Kombinatoren spezifizieren können. Nun kommen wir zur eigentlichen Synthese: Im Kontext eines Repositories suchen wir einen kombinatorischen Ausdruck, der einen bestimmten Typ hat. Dieser Typ ist die Spezifikation, die der synthetisierte Ausdruck erfüllen soll – und somit die Spezifikation des gesuchten Programms im Kontext der Softwaresynthese. Die Suche nach einem solchen Ausdruck kann als *Inhabitationsproblem* formalisiert werden.

2.3.1 Definition (Inhabitationsproblem). Gegeben ein Repository Γ und ein Typ τ, gibt es einen kombinatorischen Ausdruck e, sodass $\Gamma \vdash e : \tau$ gilt?

Zu beachten ist, dass das hier definierte Inhabitationsproblem bzgl. eines Repositories Γ *relativiert* ist. Das heißt, dass der Inhabitationsalgorithmus die Inhabitationsfrage für ein beliebiges Repository lösen kann. Das steht im Gegensatz zu Algorithmen, die nur ein fixiertes Repository kennen. Diese Eigenschaft hat große Auswirkungen auf die Entscheidbarkeit des Problems, auf die wir im nächsten Teil zu sprechen kommen.

2.3.2 Bemerkung. Wir betrachten im theoretischen Teil dieser Arbeit nur semantische Typen und keine nativen Typen, was im Gegensatz zu dem in [7] vorgestellten stratifizierten Typsystem steht. Der Grund ist, dass es bei der reinen Simulation keine bestehenden Komponenten gibt, deren Typen man in einem nativen Typsystem ausdrücken muss, weshalb rein semantische Typen ausreichend sind.

2.4 Entscheidbarkeit und Komplexität

In [7] werden für verschiedene kombinatorische Logiken ihre Inhabitationsfragen hinsichtlich der Entscheidbarkeit und Komplexität miteinander verglichen. Das Inhabitationsproblem für die *fixierte* Basis **S, K** in einer kombinatorischen Logik *ohne* Intersektionstypen ist **entscheidbar**. Relativiert man die Basis aber, ist das Inhabitationsproblem für kombinatorische Logiken sowohl mit als auch ohne Intersektionstypen **unentscheidbar**.

Um die Frage nach der relativierten Inhabitation praktisch trotzdem noch entscheiden zu können, führt [7] eine kombinatorische Logik ein, die in der Tiefe k der substituierten Typausdrücke beschränkt ist (siehe auch [2]). Mit dieser Logik ist das Inhabitationsproblem für alle k entscheidbar. In CLS wird ein Inhabitationsalgorithmus auf Basis dieser Logik implementiert. Wir wollen hier nicht weiter ins Detail gehen, weil lediglich die Komplexität der Inhabitation für diese k-beschränkten Logiken in der Arbeit von Interesse ist.

Für die k-beschränkte kombinatorische Logik mit Intersektionstypen ist das Inhabitationsproblem $(k + 2)$-EXPTIME-vollständig. Eine potentiell exponentielle Laufzeit ist für die Simulation von Berechnungsmodellen natürlich kritisch zu betrachten, da dadurch die praktische Relevanz der Arbeit in Frage gestellt wird. Eine theoretische Analyse der Komplexität der jeweiligen Inhabitationsanfragen für spezifische Berechnungsmodelle ist nicht vorgesehen, allerdings wird in Kapitel 6 eine Betrachtung der Laufzeiten anhand von Beispielen durchgeführt. In diesem Kontext ist es natürlich wichtig, die exponentielle Komplexität des Inhabitationsproblems zu kennen. Wir werden in Kapitel 6 und auch bei der Evaluation sehen, wie sehr sich die Komplexität des Inhabitationsalgorithmus in der *praktischen* Laufzeit niederschlägt.

2.5 Ausdrucksstärke des Typsystems

Eine Besonderheit der verschiedenen Typsysteme der oben genannten kombinatorischen Logiken ist deren *Ausdrucksstärke*. In [7] wird dazu gesagt: „[We] might be able to view simple types[1] as a Turing-complete logic programming language based on the inhabitation relation.“ Dabei wird das Repository Γ als logisches Programm und die Suche nach einem Inhabitanten vom Typ τ als Ausführung des Programms mit der Eingabe τ betrachtet.

In [7] wird ein 2-Zähler Automat – welcher als Turing-vollständig gilt – als Repository modelliert. Die Umsetzbarkeit dieses Berechnungsmodells weist darauf hin, dass auch die in dieser Bachelorarbeit betrachteten Modelle in dieser „logischen Programmiersprache“ umgesetzt werden können.

Da die kombinatorische Logik mit Intersektionstypen als Erweiterung der simplen Typen zu betrachten ist, sind die obigen Ergebnisse auch auf diese Logik übertragbar.

[1] Unter „simple types“ versteht man die hier definierte kombinatorische Logik ohne Intersektionstypen und ohne die Subtyping-Relation.

Kapitel 3

Simulation von DFAs

3.1 Deterministische Endliche Automaten

Wir beginnen die Arbeit mit dem Berechnungsmodell DFA. Dazu definieren wir zunächst einige Grundlagen, bevor wir im nächsten Abschnitt das Konstruktionsverfahren vorstellen.

3.1.1 Definition. Ein **deterministischer endlicher Automat** (DFA, siehe [4], Kapitel 2.2.1) ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$ [4] mit

- Einer endlichen **Zustandsmenge** Q.
- Einer endlichen Menge von Symbolen Σ, dem **Alphabet**.
- Einer **Transitionsfunktion** $\delta : Q \times \Sigma \to Q$.
- Einem **Anfangszustand** $q_0 \in Q$.
- Einer Menge $F \subseteq Q$ der **akzeptierenden Zustände**.

3.1.2 Definition. Die **erweiterte Transitionsfunktion** $\hat{\delta} : Q \times \Sigma^* \to Q$ für einen DFA $(Q, \Sigma, \delta, q_0, F)$ ist induktiv folgendermaßen definiert (siehe [5], Kapitel 1.5). In der Definition gilt $w \in \Sigma^*$, $\sigma \in \Sigma$ und $q \in Q$:

$$\hat{\delta}(q, \varepsilon) = q \tag{3.1}$$

$$\hat{\delta}(q, \sigma w) = \hat{\delta}(\delta(q, \sigma), w) \tag{3.2}$$

Die erweiterte Transitionsfunktion beschreibt die Ausführung eines DFA in Abhängigkeit von einem Eingabewort. Da der Automat **deterministisch** ist, gibt es zu jedem Wort $w \in \Sigma^*$ genau einen Endzustand $q = \hat{\delta}(q_0, w)$. Ist dieser Zustand ein akzeptierender Zustand, so wird w akzeptiert.

3.1.3 Definition. Ein DFA $(Q, \Sigma, \delta, q_0, F)$ **akzeptiert** ein Wort $w \in \Sigma^*$ genau dann, wenn $\hat{\delta}(q_0, w) \in F$ ist.[1]

[1] Die Definition von Akzeptanz in [4], Kapitel 2.2.5 nimmt einen Umweg über die Sprache eines DFA. Da der Begriff der Sprache hier nicht weiter von Bedeutung ist, wurde auf diese Ausführlichkeit verzichtet.

Die Akzeptanz ist zentral für die Simulation von DFAs: Genau bei Wörtern, die vom DFA akzeptiert werden, soll CLS einen kombinatorischen Ausdruck finden, der die Ausführung des DFA beschreibt.

3.2 Konstruktionsverfahren für DFAs

Wir wollen einen DFA so als Repository modellieren, dass jede Transition des DFA als Kombinator modelliert wird. Für jedes Wort, das der DFA akzeptiert, und nur für solche Wörter, soll es einen kombinatorischen Ausdruck geben, der der Ausführung des DFA entspricht. Zur Überführung eines DFA in ein geeignetes Repository definieren wir in diesem Abschnitt ein Konstruktionsverfahren. Wir werden aber zunächst definieren, wie wir Wörter als Typ repräsentieren.

3.2.1 Wortrepräsentation

Damit wir CLS nach einer Berechnung für ein Wort fragen können, müssen wir das Wort erst als Typ kodieren, zum Beispiel als Liste. Eine einfache Form von Listen ergibt sich, wenn wir jedem Buchstaben $a \in \Sigma$ einen Typkonstruktor $\mathtt{a}(\tau)$ zuweisen, wobei τ die Repräsentation des restlichen Wortes ist.

3.2.1 Definition (Wortrepräsentation). Sei Σ ein Alphabet. Wir definieren folgende Typkonstruktoren:

- Für jedes Symbol $a_i \in \Sigma$ gibt es einen einstelligen Typkonstruktor $\mathtt{a_i}$.
- Für das leere Wort ε gibt es eine Typkonstante ϵ.

Mit diesen Typkonstruktoren lässt sich das Wort $\underline{\text{ab}}$ als $\mathtt{a}(\mathtt{b}(\epsilon))$ repräsentieren. Abschließend definieren wir induktiv die Hilfsfunktion list für ein Wort $w \in \Sigma^*$:

- $\text{list}(\varepsilon) = \epsilon$
- $\text{list}(a_i x) = \mathtt{a_i}(\text{list}(x))$ für $a_i \in \Sigma$ und $x \in \Sigma^*$

Der Typ für ein Wort $w \in \Sigma^*$ ist dann $\mathtt{Word}(\text{list}(w))$.

3.2.2 Beispiel. Das Wort $\underline{\text{abba}}$ mit $\Sigma = \{\text{a}, \text{b}\}$ lässt sich folgendermaßen kodieren:

$$\mathtt{Word}(\text{list}(\underline{\text{abba}})) = \mathtt{Word}(\mathtt{a}(\mathtt{b}(\mathtt{b}(\mathtt{a}(\epsilon)))))$$

3.2.2 Konstruktionsverfahren

3.2.3 Definition. Zu einem DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ definieren wir folgende Typkonstruktoren:

- Zu jedem $q \in Q$ gibt es eine Typkonstante $\mathtt{q}$.
- Es gibt einen einstelligen Typkonstruktor $\mathtt{St}$.

- Es gibt Typkonstruktoren für die Wortrepräsentation gemäß Definition 3.2.1.

3.2.4 Definition (Konstruktionsverfahren). Zu einem DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ erzeugen wir das Repository $\Gamma_\mathcal{A}$ wie folgt:

$$\Gamma_F = \{\texttt{Fin[q]} : \texttt{St(q)} \to \texttt{Word}(\epsilon) \mid q \in F\}$$
$$\Gamma_\delta = \{\texttt{D[q,a,p]} : (\texttt{St(p)} \to \texttt{Word}(\alpha)) \to (\texttt{St(q)} \to \texttt{Word(a}(\alpha))) \mid (q, a, p) \in \delta\}$$
$$\Gamma_s = \{\texttt{Run} : (\texttt{St(q}_0) \to \texttt{Word}(\alpha)) \to \texttt{Word}(\alpha)\}$$
$$\Gamma_\mathcal{A} = \Gamma_F \cup \Gamma_\delta \cup \Gamma_s$$

Der Kombinator `Run` ist notwendig, um sicherzustellen, dass der Automat in seinem Startzustand beginnt. Im Allgemeinen sind Ausführungen des Automaten Funktionen vom Typ $\texttt{St(q)} \to \texttt{Word}(\text{list}(w))$, was bedeuten soll, dass der Zustand q das Wort w akzeptiert. `Run` überführt diese zustandsabhängige Berechnung in ein Wort, solange die Berechnung beim Startzustand q_0 beginnt.

Die Transitionskombinatoren in Γ_δ sind im Kontext zu der in [7] erwähnten *zielorientierten* Berechnung zu verstehen. Ein Ziel $\texttt{St(q)} \to \texttt{Word(a}(\alpha))$ kann erreicht werden, indem ein Transitionskombinator $\texttt{D[q,a,p]}$ auf einen Ausdruck $e : \texttt{St(p)} \to \texttt{Word}(\alpha)$ angewandt wird. Dieses e stellt ein weiteres Ziel dar, welches entweder durch weitere Applikation von Transitionskombinatoren erreicht werden kann oder durch Applikation von `Fin[q]` als Basisfall der Rekursion.

3.2.5 Definition. Die **Inhabitationsfrage** lässt sich für einen DFA $\mathcal{A}$ folgendermaßen stellen: Gegeben ein Repository $\Gamma_\mathcal{A}$, das mit dem Verfahren aus Definition 3.2.4 erzeugt wurde, und ein Wort $w \in \Sigma^*$, gibt es einen kombinatorischen Ausdruck e, der $\Gamma_\mathcal{A} \vdash e : \texttt{Word}(\text{list}(w))$ erfüllt?

Zu beachten ist, dass CLS keinen Ausdruck finden wird, wenn das Wort vom Automaten nicht akzeptiert wird. Es können also nur Berechnungen zu Wörtern gefunden werden, die der Automat auch akzeptiert.

3.2.3 Korrektheit und Vollständigkeit

Wir zeigen die Korrektheit und Vollständigkeit des Konstruktionsverfahrens, indem wir den in (1.1) definierten allgemeinen Simulationsansatz auf DFAs zuschneiden und beweisen.

3.2.6 Satz. *Für einen DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ und alle $w \in \Sigma^*$ gilt:*

$$\exists e(\Gamma_\mathcal{A} \vdash e : \texttt{Word}(\text{list}(w))) \iff \hat{\delta}(q_0, w) \in F \tag{3.3}$$

Die Akzeptanz des Automaten wird hierbei wie in Definition 3.1.3 mithilfe der erweiterten Transitionsfunktion präzisiert.

Um Satz 3.2.6 zu beweisen, müssen wir zuerst eine etwas allgemeinere Aussage zeigen, die sich bei der Akzeptanz der Wörter nicht auf den Startzustand q_0 beschränkt. Wir teilen diese Aussage auf zwei Lemmata auf, die im Folgenden definiert und bewiesen werden.

Die Beweise sind Induktionen über der Länge n der Wörter w. Im Beweis von Lemma 3.2.7 folgern wir aus der Form der kombinatorischen Ausdrücke, die zu einem Eingabetypen $\mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))$ synthetisiert werden, dass der Automat das Wort w (mit Simulationsbeginn im Zustand q) akzeptiert. Im Beweis von Lemma 3.2.8 konstruieren wir auf Basis der Annahme, dass der Automat das Wort w (mit Simulationsbeginn im Zustand q) akzeptiert, einen Ausdruck, der den gesuchten Typen hat, und zeigen somit seine Existenz. Dabei kommen jeweils die Induktionsvoraussetzungen zum Tragen, sodass in einem Induktionsschritt nur das erste Zeichen eines Wortes betrachtet werden muss.

3.2.7 Lemma. *Für einen DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ und alle $w \in \Sigma^*$ sowie $q \in Q$ gilt:*

$$\exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))) \implies \hat{\delta}(q, w) \in F \tag{3.4}$$

Beweis. Wir zeigen die Aussage mithilfe einer Induktion über der Länge n der Wörter w. **Induktionsanfang:** Für $n = 0$ gilt $w = \varepsilon$. Sei $q \in Q$ beliebig. Die einzigen Ausdrücke, die einen Typ $\mathtt{St(q)} \to \mathtt{Word}(\epsilon)$ inhabitieren, sind Ausdrücke $e = \mathtt{Fin[q]}$. Da $\mathtt{Fin[q]}$ nur für $q \in F$ in $\Gamma_{\mathcal{A}}$ vorhanden ist, gilt $q \in F$. Damit gilt natürlich auch:

$$q \in F \implies \hat{\delta}(q, \varepsilon) \in F \implies \hat{\delta}(q, w) \in F$$

Induktionsschritt: Sei $q \in Q$ beliebig. Wir betrachten nun Wörter $w = av$ mit $|v| = n$, $a \in \Sigma$ und $|w| = n + 1$. In der Induktion nehmen wir an, dass (3.4) für alle $|x| \leq n$ gilt.

Da w mindestens die Länge 1 hat, müssen wir nur Kombinatoren der Art $\mathtt{D[q, a, p]}$ mit dem Typ $(\mathtt{St(p)} \to \mathtt{Word}(\alpha)) \to (\mathtt{St(q)} \to \mathtt{Word(a}(\alpha)))$ betrachten. $\mathtt{Fin[q]}$ kann aufgrund der Länge des Wortes nicht die Form von e sein. `Run` hat einen unpassenden Typ. Wir betrachten folgende Ausdrücke:

$$e = \mathtt{D[q, a, p]}\ e' : \mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))$$

Für den Ausdruck e' gilt $\Gamma_{\mathcal{A}} \vdash e' : \mathtt{St(p)} \to \mathtt{Word}(\mathrm{list}(v))$. Mit der Existenz von e' können wir aufgrund der Induktionsvoraussetzung und $|v| \leq n$ folgern, dass $\hat{\delta}(p, v) \in F$ ist. Da der Kombinator $\mathtt{D[q, a, p]}$ in e vorkommt und somit existiert, muss insbesondere $\delta(q, a) = p$ gelten. Mit diesen Eigenschaften schließen wir den Beweis:

$$\begin{aligned} \hat{\delta}(p, v) \in F &\implies \hat{\delta}(\delta(q, a), v) \in F \\ &\implies \hat{\delta}(q, av) \in F \\ &\implies \hat{\delta}(q, w) \in F \end{aligned}$$

□

3.2.8 Lemma. *Für einen DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ und alle $w \in \Sigma^*$ sowie $q \in Q$ gilt:*

$$\hat{\delta}(q, w) \in F \implies \exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))) \tag{3.5}$$

Beweis. Wir zeigen dies ebenfalls mit einer Induktion über der Länge n der Wörter w.
Induktionsanfang: Für $n = 0$ gilt $w = \varepsilon$. Sei $q \in Q$ beliebig. Wir nehmen an, dass $\hat{\delta}(q, \varepsilon) = q \in F$ ist. Aus der Definition von $\Gamma_{\mathcal{A}}$ folgt nun, dass $\mathtt{Fin[q]} : \mathtt{St(q)} \to \mathtt{Word}(\epsilon)$ in $\Gamma_{\mathcal{A}}$ enthalten ist. Mit $e = \mathtt{Fin[q]}$ gilt dann $\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q)} \to \mathtt{Word}(\epsilon)$.
Induktionsschritt: Sei $q \in Q$ beliebig. Wir betrachten ein Wort $w = av$ mit $a \in \Sigma$ sowie $v \in \Sigma^*$ und $|v| = n$. In der Induktion nehmen wir an, dass (3.5) für alle $|x| \leq n$ gilt.

Wir nehmen zunächst an, dass $\hat{\delta}(q, av) = \hat{\delta}(\delta(q, a), v) \in F$ ist. Es sei $p = \delta(q, a)$. Da $(q, a, p) \in \delta$, ist der Kombinator $\mathtt{D[q, a, p]} : (\mathtt{St(p)} \to \mathtt{Word}(\alpha)) \to (\mathtt{St(q)} \to \mathtt{Word(a}(\alpha)))$ in $\Gamma_{\mathcal{A}}$ enthalten. Wir wählen:

$$e = \mathtt{D[q, a, p]}\ e'$$

Der Ausdruck e hat den gewünschten Typ: $\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))$. Wir benötigen aber einen weiteren Ausdruck e' mit $\Gamma_{\mathcal{A}} \vdash e' : \mathtt{St(p)} \to \mathtt{Word}(\mathrm{list}(v))$.

Für den Induktionsbeweis reicht die Existenz eines solchen Ausdrucks. Aufgrund der Induktionsvoraussetzung gilt für v (wobei $|v| \leq n$) und alle $q' \in Q$:

$$\hat{\delta}(q', v) \in F \implies \exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q')} \to \mathtt{Word}(\mathrm{list}(v)))$$

Wir haben aber gerade schon gesehen, dass $\hat{\delta}(p, v) = \hat{\delta}(q, w)$ und damit $\hat{\delta}(p, v) \in F$. Setzen wir $q' = p$ folgt also, dass es einen Ausdruck e' gibt, der den Typ $\mathtt{St(p)} \to \mathtt{Word}(\mathrm{list}(v))$ inhabitiert. □

Mithilfe der zwei Lemmata können wir nun Satz 3.2.6 beweisen.

Beweis (Satz 3.2.6). Sei $w \in \Sigma^*$ beliebig. Wir zeigen zunächst:

$$\exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{Word}(\mathrm{list}(w))) \implies \hat{\delta}(q_0, w) \in F$$

Wir nehmen an, dass es einen Ausdruck e mit $\Gamma_{\mathcal{A}} \vdash e : \mathtt{Word}(\mathrm{list}(w))$ gibt. Ein Ausdruck vom Typ `Word` lässt sich nur über den Kombinator `Run` erzeugen, weshalb e die Form `Run` e' hat. Der Ausdruck e' inhabitiert als Argument von `Run` den Typen $\mathtt{St(q_0)} \to \mathtt{Word}(\mathrm{list}(w))$. Mit der Existenz von e' und Lemma 3.2.7 erhalten wir, dass $\hat{\delta}(q_0, w) \in F$ ist.

Wir müssen noch die andere Richtung zeigen:

$$\hat{\delta}(q_0, w) \in F \implies \exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{Word}(\mathrm{list}(w)))$$

Wir nehmen an, dass $\hat{\delta}(q_0, w) \in F$ ist. Mit Lemma 3.2.8 erhalten wir einen Ausdruck e', der die Aussage $\Gamma_A \vdash e' : \mathtt{St(q_0)} \to \mathtt{Word}(\mathrm{list}(w))$ erfüllt. Der hier gesuchte Ausdruck ergibt

sich mit $e = \texttt{Run}\ e'$. Insbesondere lässt sich `Run` anwenden, da e' von $\texttt{St}(\texttt{q}_0)$ ausgeht. Also existiert ein Ausdruck mit dem gesuchten Typen.

Zusätzlich zeigen wir noch, dass die Typkonstante ω keinen Einfluss auf die Simulation hat. Wir nehmen an, dass wir einen Ausdruck $e : \texttt{Word}(\text{list}(w))$ haben. Für alle Typen τ gilt, dass $\tau \leq \omega$ ist (siehe Definition 2.2.2). Nach der Subtyping-Regel aus Abbildung 2.1 gilt damit für den Ausdruck e auch $e : \texttt{Word}(\omega)$, da $\texttt{Word}(\text{list}(w)) \leq \texttt{Word}(\omega)$ ist. Wir suchen bei der Inhabitation aber nach einem Typen $\texttt{Word}(\text{list}(w))$, weshalb der Typ $\texttt{Word}(\omega)$ zu allgemein ist. Die gleiche Argumentation lässt sich auf die Zwischenschritte der Simulation übertragen, weshalb wir uns auch sicher sein können, dass ω nichts an den obigen Aussagen ändert. Damit hat ω keinen Einfluss auf die Simulation.

Der Satz 3.2.6 ist nun insgesamt bewiesen. □

3.3 Beispiel zur Codegenerierung

Wie in der Einleitung schon erwähnt, kann die Simulation von Berechnungsmodellen verwendet werden, um Programme zu generieren. In diesem Abschnitt behandeln wir ein Beispiel zur Codegenerierung. Wir definieren zunächst einen Automaten zu einer gegebenen Problemstellung, konstruieren anschließend das Repository dazu und generieren einen Ausdruck zu einer Beispielkonfiguration. Zum Schluss kommen wir darauf zu sprechen, wie dieser Ausdruck im Kontext des Beispiels zu interpretieren ist.

3.3.1 Beispiel-DFA

Wir betrachten den DFA aus Abbildung 3.1, den wir in ein Repository umwandeln wollen. Der DFA modelliert eine künstliche Intelligenz (KI) in einem generischen Runner-Spiel, das folgendermaßen aufgebaut ist: Das Spielfeld besteht aus ground (**g**) und barrier (**b**) Feldern. Die Spielfigur befindet sich am Anfang auf dem Boden (**grnd**). Ist das nächste Feld eine Barriere, so muss die Spielfigur springen, um nicht gegen das Hindernis zu laufen. Die Spielfigur befindet sich dann in der Luft (**air1**). Ist die Spielfigur einmal gesprungen, so darf sie höchstens ein weiteres mal springen (**air2**). Danach fällt die Figur für zwei Felder (**fall**). Schafft die Figur es nicht, die Hindernisse zu bewältigen, so stirbt sie (**dead**). Die Wörter des DFA sind Spielfelder, also Sequenzen von ground und barrier Feldern. Wir akzeptieren eine Lösung nur, wenn die Spielfigur den Boden erreicht hat, bevor das Spielfeld endet.
Mit dem in Definition 3.2.4 vorgestellten Verfahren kann man ein Repository erzeugen, mit dem KI-Verhaltenspläne erzeugt werden können, die aus der Konfiguration des Spielfelds berechnet werden.

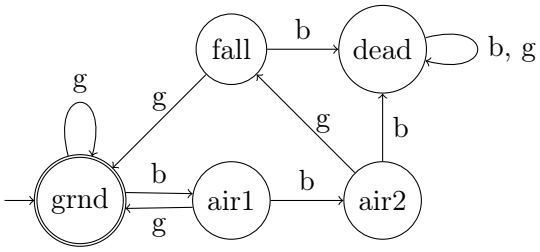

Abbildung 3.1: Der DFA $\mathcal{A}_{KI}$, der die KI einer einfachen Runner-Spielfigur modelliert.

3.3.2 Repository und Inhabitation

Wir wenden nun das Konstruktionsverfahren auf den Beispiel-DFA an. Das konstruierte Repository $\Gamma_{\mathcal{A}_{KI}}$ enthält folgende Kombinatoren, wobei wir Kombinatoren aus Γ_δ der Übersichtlichkeit halber nur exemplarisch aufführen:

$$\begin{aligned}
\texttt{Fin}[\texttt{grnd}] &: \texttt{St}(\texttt{grnd}) \to \texttt{Word}(\epsilon) \\
\texttt{Run} &: (\texttt{St}(\texttt{grnd}) \to \texttt{Word}(\alpha)) \to \texttt{Word}(\alpha) \\
\texttt{D}[\texttt{grnd},\texttt{g},\texttt{grnd}] &: (\texttt{St}(\texttt{grnd}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{grnd}) \to \texttt{Word}(\texttt{g}(\alpha))) \\
\texttt{D}[\texttt{grnd},\texttt{b},\texttt{air1}] &: (\texttt{St}(\texttt{air1}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{grnd}) \to \texttt{Word}(\texttt{b}(\alpha))) \\
&\ldots
\end{aligned}$$

Wir wollen nun eine KI finden, die das Spielfeld $\underline{\text{bbgg}}$ bewältigen kann, d.h. wir wollen einen Ausdruck finden, der den Typ inhabitiert, der das Spielfeld geeignet kodiert. Formal stellen wir die Frage, ob es einen Ausdruck e gibt, sodass $\Gamma_{\mathcal{A}_{KI}} \vdash e : \texttt{Word}(\text{list}(\underline{\text{bbgg}}))$ gilt. Ein erster Schritt führt uns zum Kombinator Run und dem Ausdruck

$$e = \texttt{Run}\ e_1 : \texttt{Word}(\texttt{b}(\texttt{b}(\texttt{g}(\texttt{g}(\epsilon)))))$$

wobei für e_1 gelten muss, dass

$$e_1 : \texttt{St}(\texttt{grnd}) \to \texttt{Word}(\texttt{b}(\texttt{b}(\texttt{g}(\texttt{g}(\epsilon)))))$$

Wir finden

$$e_1 = (\texttt{D}[\texttt{grnd},\texttt{b},\texttt{air1}]\ e_2) : \texttt{St}(\texttt{grnd}) \to \texttt{Word}(\texttt{b}(\texttt{b}(\texttt{g}(\texttt{g}(\epsilon)))))$$

mit

$$e_2 : \texttt{St}(\texttt{air1}) \to \texttt{Word}(\texttt{b}(\texttt{g}(\texttt{g}(\epsilon))))$$

Führen wir diese Auswertung weiter, erreichen wir den Ausdruck

$$e = \texttt{Run}\ (\texttt{D}[\texttt{grnd},\texttt{b},\texttt{air1}]\ (\texttt{D}[\texttt{air1},\texttt{b},\texttt{air2}]\ (\texttt{D}[\texttt{air2},\texttt{g},\texttt{fall}]\ (\texttt{D}[\texttt{fall},\texttt{g},\texttt{grnd}]\ \texttt{Fin}[\texttt{grnd}]))))$$

Insbesondere benötigen wir $\texttt{Fin}[\texttt{grnd}]$, um die Kette von Transitionskombinatoren abzuschließen, da $\texttt{Fin}[\texttt{grnd}]$ der einzige Kombinator ist, der argumentlos eine Funktion vom Typ $\texttt{St}(\texttt{grnd}) \to \texttt{Word}(\epsilon)$ bereitstellt. Damit ist sichergestellt, dass die Berechnung auch in einem akzeptierenden Zustand endet.

3.3.3 Verwendung des Ergebnisses

Der generierte Ausdruck e entspricht dem Pfad durch den DFA, der für das Wort bbgg genommen wird. Insbesondere sind Kombinatoren wie $\texttt{D}[\texttt{grnd}, \texttt{b}, \texttt{air1}]$ konzeptuell an **Aktionen** geknüpft, die der Spielfigur ermöglichen, das Spielfeld zu überwinden. So muss die Spielfigur zum Beispiel bei einer Barriere **springen**, sofern sie denn noch springen darf. Indem man den Kombinatoren Aktionen zuweist, die eine semantische Bedeutung haben, wird der erzeugte Ausdruck zum realisierbaren und verständlichen Programm.

In dem konkreten Ausdruck kommen dann beispielsweise folgende Aktionen vor. Die Kombinatoren sind in der Reihenfolge der DFA-Ausführung geordnet.

- $\texttt{D}[\texttt{grnd}, \texttt{b}, \texttt{air1}]$: **Springen**
- $\texttt{D}[\texttt{air1}, \texttt{b}, \texttt{air2}]$: **Springen** (Einmaliger Sprung in der Luft)
- $\texttt{D}[\texttt{air2}, \texttt{g}, \texttt{fall}]$: **Fallen**
- $\texttt{D}[\texttt{fall}, \texttt{g}, \texttt{grnd}]$: **Fallen**

Die Figur muss also genau zwei mal springen und sich danach zwei mal fallen lassen.

3.3.1 Bemerkung. Der aufmerksame Leser mag sich jetzt fragen, wie der Typ von Kombinatoren wie $\texttt{D}[\texttt{grnd}, \texttt{b}, \texttt{air1}]$ mit dem Konzept von *Aktionen* übereinstimmt. Eine Aktion wirkt auf ein Subjekt. Hier fehlt das Subjekt als Parameter. Eine Lösung ergibt sich daraus, dass man einen semantischen Typ $(\texttt{St}(\texttt{p}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{q}) \to \texttt{Word}(\texttt{a}(\alpha)))$ zum Typen $((\texttt{St}(\texttt{p}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{q}) \to \texttt{Word}(\texttt{a}(\alpha)))) \cap (\texttt{Subject} \to \texttt{Subject})$ erweitert, wobei $(\texttt{Subject} \to \texttt{Subject})$ der native Typ ist. Konkret ist das Ergebnis dann auch eine Funktion, die auf ein Subjekt wirkt und dessen neuen Zustand berechnet – eine Aktion. Dies ist ohne weiteres möglich, da die semantischen Typen an sich nur während der Inhabitation relevant sind.

In Kapitel 6 wird eine native Semantik für den Runner-DFA mit passenden nativen Typen in cls-scala implementiert. Dort werden die obigen Überlegungen praktisch umgesetzt.

Kapitel 4

Simulation von ε-NFAs

4.1 NFAs mit Epsilon-Transitionen

Nichtdeterministische endliche Automaten (NFAs) sind ähnlich definiert wie DFAs und sogar gleichmächtig zu DFAs (siehe [4], Theorem 2.11). Dennoch haben NFAs einen Zweck: Der **Nichtdeterminismus** von NFAs macht es oft einfacher, Automaten zu einer Problemstellung zu erstellen (siehe [4], Einleitung von Kapitel 2.3).

Dieser Nichtdeterminismus wird beim NFA durch die nichtdeterministische Transitionsfunktion erreicht: Im Gegensatz zum DFA, bei dem jede Transition zu genau einem Zustand führt, kann eine Transition beim NFA zu keinem oder mehreren Zuständen führen. Hier betrachten wir eine Erweiterung von NFAs, sogenannte ε-NFAs. Sie erlauben es, Transitionen durchzuführen, die kein Zeichen konsumieren. Dies erleichtert die Erstellung von Automaten weiter (siehe [4], Einleitung von Kapitel 2.5).

4.1.1 Definition. Ein **NFA mit Epsilon-Transitionen** (ε-NFA, siehe [4], Kapitel 2.5.2) ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$ mit

- Einer endlichen **Zustandsmenge** Q, einem **Alphabet** Σ, einem **Startzustand** q_0 und einer Menge an **akzeptierenden Zuständen** $F \subseteq Q$, parallel zu DFAs.
- Einer **Transitionsfunktion** $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$, wobei $\mathcal{P}(Q)$ die Potenzmenge von Q ist. Dabei setzen wir voraus, dass $\varepsilon \notin \Sigma$ ist.

4.1.2 Definition. Der **Epsilon-Abschluss** ECLOSE eines Zustands $q \in Q$ ist die Menge aller Zustände, die von q aus über ε-Transitionen erreichbar sind (siehe [4], Kapitel 2.5.3). Formal ist $\text{ECLOSE}(q)$ die kleinste Menge, die folgende Eigenschaften erfüllt:

$$q \in \text{ECLOSE}(q) \tag{4.1}$$

$$\delta(p, \varepsilon) \in \text{ECLOSE}(q) \text{ für alle } p \in \text{ECLOSE}(q) \tag{4.2}$$

Zudem ist ECLOSE für Mengen $S \subseteq Q$ definiert: $\text{ECLOSE}(S) = \bigcup_{q \in S} \text{ECLOSE}(q)$.

4.1.3 Definition. Die **erweiterte Transitionsfunktion** $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ eines ε-NFA $(Q, \Sigma, \delta, q_0, F)$ ist induktiv folgendermaßen definiert (siehe [5], Kapitel 4.1). In der Definition gilt $w \in \Sigma^*$, $\sigma \in \Sigma$ und $q \in Q$:

$$\hat{\delta}(q, \varepsilon) = \text{ECLOSE}(q) \tag{4.3}$$

$$\hat{\delta}(q, \sigma w) = \bigcup_{p \in P} \hat{\delta}(p, w) \text{ mit } P = \text{ECLOSE}(\bigcup_{q' \in \text{ECLOSE}(q)} \delta(q', \sigma)) \tag{4.4}$$

Die Regel (ETF2) aus [5] können wir entfernen, da sie von (ETF3) abgedeckt wird. Das ist einfach zu sehen, wenn man $w = \varepsilon$ setzt:

$$\begin{aligned}
\hat{\delta}(q, \sigma\varepsilon) &= \bigcup_{p \in P} \hat{\delta}(p, \varepsilon) \text{ mit } P = \text{ECLOSE}(\bigcup_{q' \in \text{ECLOSE}(q)} \delta(q', \sigma)) \\
&= \bigcup_{p \in P} \text{ECLOSE}(p) \\
&\overset{(1)}{=} \bigcup_{p \in P} p \\
&= P, \text{ wobei die Def. von } P \text{ genau (ETF2) entspricht}
\end{aligned}$$

Die Umformung (1) hält, da sich jedes $p \in P$ bereits in einem Epsilon-Abschluss befindet und somit schon alle Zustände in P sind, die über ε-Transitionen von irgendeinem $p \in P$ aus erreicht werden können. Also können wir durch die Anwendung von $\text{ECLOSE}(p)$ keine neuen Zustände finden.

Mit der erweiterten Transitionsfunktion können wir nun die Akzeptanz von ε-NFAs formal definieren. Ein ε-NFA soll ein Wort w genau dann akzeptieren, wenn mindestens einer der Zustände, die mit w erreicht werden können, ein akzeptierender Zustand ist.

4.1.4 Definition. Ein ε-NFA $(Q, \Sigma, \delta, q_0, F)$ **akzeptiert** ein Wort $w \in \Sigma^*$ genau dann, wenn $\hat{\delta}(q_0, w) \cap F \neq \emptyset$ ist.[1]

4.2 Konstruktionsverfahren für ε-NFAs

Das Konstruktionsverfahren soll einen ε-NFA in ein simulationsbereites Repository überführen. Dabei stützen wir uns zunächst auf das Konstruktionsverfahren für DFAs. In der Tat bedarf es für NFAs keiner weiteren Art von Kombinatoren, da Nichtdeterminismus durch CLS unterstützt wird und wir somit lediglich mehrere Kombinatoren für Transitionen bereitstellen müssen, die aus einem Zustand heraus das gleiche Zeichen konsumieren. Für ε-NFAs benötigen wir allerdings noch Kombinatoren, die die ε-Transitionen modellieren, da die `D[q, a, p]` Kombinatoren keine Übergänge unterstützen, die kein Zeichen konsumieren. Für die Kodierung von Wörtern benutzen wir die in Definition 3.2.1 vorgestellten Typen.

[1] Die Definition von Akzeptanz in [4], Kapitel 2.5.4 nimmt wieder einen Umweg über die akzeptierte Sprache.

4.2.1 Konstruktionsverfahren

4.2.1 Definition (Konstruktionsverfahren). Für einen ε-NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ erzeugen wir das Repository $\Gamma_\mathcal{A}$ wie folgt:

$$\Gamma_F = \{\texttt{Fin[q]} : \texttt{St(q)} \to \texttt{Word}(\epsilon) \mid q \in F\}$$
$$\Gamma_\delta = \{\texttt{D[q,a,p]} : (\texttt{St(p)} \to \texttt{Word}(\alpha)) \to (\texttt{St(q)} \to \texttt{Word(a}(\alpha))) \mid (q, a, p) \in \delta \wedge a \in \Sigma\}$$
$$\Gamma_\varepsilon = \{\texttt{E[q,p]} : (\texttt{St(p)} \to \texttt{Word}(\alpha)) \to (\texttt{St(q)} \to \texttt{Word}(\alpha)) \mid (q, \varepsilon, p) \in \delta\}$$
$$\Gamma_s = \{\texttt{Run} : (\texttt{St(q}_0\texttt{)} \to \texttt{Word}(\alpha)) \to \texttt{Word}(\alpha)\}$$
$$\Gamma_\mathcal{A} = \Gamma_F \cup \Gamma_\delta \cup \Gamma_\varepsilon \cup \Gamma_s$$

4.2.2 Definition. Die **Inhabitationsfrage** lässt sich für einen ε-NFA $\mathcal{A}$ folgendermaßen stellen: Gegeben ein Repository $\Gamma_\mathcal{A}$, das mit dem Verfahren aus Definition 4.2.1 erzeugt wurde, und ein Wort $w \in \Sigma^*$, gibt es einen kombinatorischen Ausdruck e, der $\Gamma_\mathcal{A} \vdash e : \texttt{Word}(\text{list}(w))$ erfüllt?

Wie bei DFAs gilt auch wieder, dass kein Ausdruck erzeugt wird, wenn der Automat nicht akzeptiert. Dies kann auch hier benutzt werden, um zu verhindern, dass Programme mit unerwünschten Konfigurationen erzeugt werden.

4.2.2 Korrektheit und Vollständigkeit

Wir zeigen die Korrektheit und Vollständigkeit des Konstruktionsverfahrens, indem wir den in (1.1) definierten allgemeinen Simulationsansatz auf ε-NFAs zuschneiden und beweisen.

4.2.3 Satz. *Für einen ε-NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ und alle $w \in \Sigma^*$ gilt:*

$$\exists e(\Gamma_\mathcal{A} \vdash e : \texttt{Word}(\text{list}(w))) \iff \hat{\delta}(q_0, w) \cap F \neq \emptyset \tag{4.5}$$

Für den Beweis benötigen wir zunächst Lemmata, die uns erlauben, zu anderen Zuständen des Epsilon-Abschlusses zu wechseln. Im Folgenden wird ein ε-NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ angenommen.

4.2.4 Lemma. *Genau wenn $p \in \text{ECLOSE}(q)$ ist, gibt es eine Folge $S = (q, ..., p)$ von Zuständen, die einen Pfad von q nach p repräsentiert, auf dem nur ε-Transitionen verwendet werden. Solch einen Pfad nennen wir **ε-Pfad**. Insgesamt bedeutet das formal:*

$$p \in \text{ECLOSE}(q) \iff \exists S = (q, ..., p).\, P(S) \tag{4.6}$$
$$P((s_1, ..., s_k)) \iff \forall 1 \leq i < k\, (s_{i+1} \in \delta(s_i, \epsilon)) \tag{4.7}$$

Beweis. Wir zeigen $p \in \text{ECLOSE}(q) \implies \exists S = (q, ..., p).\, P(S)$ mit einer Induktion über die Anzahl n der ε-Transitionen, die benötigt werden, um von q nach p zu gelangen. Man beachte, dass für einen Pfad der Länge k gilt, dass $n = k - 1$ ist.

Induktionsanfang: Bei $n = 0$ gibt es nur einen Zustand, der über 0 ε-Transitionen erreichbar ist: q. Der einfache ε-Pfad (q) existiert, für den $k \not< 1$ ist. Damit gilt (4.6).
Induktionsschritt: Wir betrachten Zustände p, die über n ε-Transitionen erreichbar sind. Wir nehmen an, dass für einen Zustand p, der über $n-1$ ε-Transitionen von q aus erreichbar ist, (4.6) gilt. Wir müssen einen Pfad $S = (q, ..., p)$ finden, der die Länge $k + 1$ hat. Sei $s_k \in \text{ECLOSE}(q)$ ein Zustand, der über $n - 1$ ε-Transitionen erreichbar ist und für den $p \in \delta(s_k, \varepsilon)$ gilt. Dieser Zustand muss existieren, da $p \in \text{ECLOSE}(q)$ über n ε-Transitionen erreichbar ist. Hätte p keinen solchen Vorgänger, so wäre p nicht in $\text{ECLOSE}(q)$. Laut Induktionsvoraussetzung gibt es einen Pfad $S' = (q, s_2, ..., s_k)$, der (4.7) erfüllt. Somit können wir auch einen Pfad $S = (q, ..., s_k, p)$ konstruieren, der (4.7) erfüllt, weil $p \in \delta(s_k, \varepsilon)$ ist und S' (4.7) erfüllt.

Die Gegenrichtung ergibt sich aus der Definition von ε-Pfaden: Da wir auf dem ε-Pfad von einem q zu einem p nur ε-Transitionen verwenden dürfen, muss $p \in \text{ECLOSE}(q)$ sein. □

4.2.5 Lemma. *Zu jedem ε-Pfad $S = (s_1, ..., s_k)$ und jedem Ausdruck $e : \mathtt{St}(\mathtt{s_k}) \to \mathtt{Word}(\tau)$ gibt es einen Ausdruck $f : \mathtt{St}(\mathtt{s_1}) \to \mathtt{Word}(\tau)$.*

Beweis. Wir betrachten ε-Pfade $S = (s_1, ..., s_k)$ und Ausdrücke $e : \mathtt{St}(\mathtt{s_k}) \to \mathtt{Word}(\tau)$. Für $s_1 = s_k$ ist $f = e$ trivial. Für $k \geq 2$ gilt: Da S ein ε-Pfad ist, gilt für alle Paare (s_i, s_{i+1}) mit $1 \leq i < k$, dass $s_{i+1} \in \delta(s_i, \varepsilon)$ ist. Damit existiert für alle diese Paare auch ein Kombinator $\mathtt{E}[\mathtt{s_i}, \mathtt{s_{i+1}}] : (\mathtt{St}(\mathtt{s_{i+1}}) \to \mathtt{Word}(\alpha)) \to (\mathtt{St}(\mathtt{s_i}) \to \mathtt{Word}(\alpha))$ in $\Gamma_{\mathcal{A}}$. Wir konstruieren einen Ausdruck $f = \mathtt{E}[\mathtt{s_1}, \mathtt{s_2}]$ ... $\mathtt{E}[\mathtt{s_{k-1}}, \mathtt{s_k}]$ e. Die Typen stimmen überein, da ein Kombinator $\mathtt{E}[\mathtt{s_i}, \mathtt{s_{i+1}}]$ einen Ausdruck vom Typ $\mathtt{St}(\mathtt{s_{i+1}}) \to \mathtt{Word}(\alpha)$ in einen Ausdruck vom Typ $\mathtt{St}(\mathtt{s_i}) \to \mathtt{Word}(\alpha)$ überführt. $\mathtt{E}[\mathtt{s_{k-1}}, \mathtt{s_k}]$ lässt sich somit auf e anwenden, $\mathtt{E}[\mathtt{s_{k-2}}, \mathtt{s_{k-1}}]$ auf $\mathtt{E}[\mathtt{s_{k-1}}, \mathtt{s_k}]$ e, und so weiter. Letztendlich führt die Applikation von $\mathtt{E}[\mathtt{s_1}, \mathtt{s_2}]$ zum gewünschten Typ $\mathtt{St}(\mathtt{s_1}) \to \mathtt{Word}(\tau)$. □

Ähnlich wie beim Beweis zur DFA-Simulation zeigen wir nun eine allgemeinere Aussage, die sich nicht auf Startzustände beschränkt. Diese teilen wir wieder auf zwei Lemmata auf.

Der Aufbau der Beweise orientiert sich an den Beweisen der Lemmata aus Abschnitt 3.2.3. Zusätzlich zum dortigen Vorgehen müssen wir hier die ε-Transitionen des ε-NFA beachten. Im Beweis von Lemma 4.2.6 zeigen wir, wie im Beweis von Lemma 3.2.7, über die Form eines synthetisierten Ausdrucks, dass der Automat das zugehörige Wort (mit Simulationsbeginn im Zustand q) akzeptiert. Dabei beinhaltet die Form der synthetisierten Ausdrücke eine beliebige Anzahl von $\mathtt{E}[\mathtt{q}, \mathtt{p}]$-Kombinatoren, die wir über das Konzept der ε-Pfade behandeln. Im Beweis von Lemma 4.2.7 konstruieren wir, wie im Beweis von Lemma 3.2.8, einen Ausdruck, der den gesuchten Typ hat. Dabei verwenden wir Lemma 4.2.5, um auch bei dem Vorhandensein von ε-Transitionen einen solchen Ausdruck konstruieren zu können.

4.2.6 Lemma. *Für einen ε-NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ und alle $w \in \Sigma^*$ sowie $q \in Q$ gilt:*

$$\exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))) \implies \hat{\delta}(q, w) \cap F \neq \emptyset \tag{4.8}$$

Beweis. Wir zeigen die Aussage mithilfe einer Induktion über der Länge n der Wörter w.
Induktionsanfang: Sei $q \in Q$ beliebig. Für $n = 0$ gilt $w = \varepsilon$. Hier kommen nur Ausdrücke der Form $e = \mathtt{E[q_0, q_1]}$ ... $\mathtt{E[q_{k-1}, q_k]}$ $\mathtt{Fin[q_k]} : \mathtt{St(q_0)} \to \mathtt{Word}(\epsilon)$ für $k \geq 0$ in Frage. Da für die Existenz von $\mathtt{Fin[q_k]}$ in $\Gamma_{\mathcal{A}}$ der Zustand $q_k \in F$ sein muss, gilt $\hat{\delta}(q_k, \epsilon) \cap F \neq \emptyset$. Für jedes $\mathtt{E[q_i, q_{i+1}]}$ aus e gilt aufgrund der Konstruktion von $\Gamma_{\mathcal{A}}$, dass $(q_i, \epsilon, q_{i+1}) \in \delta$ ist. Damit gibt es vom Zustand q_0 bis zum Zustand q_k einen ε-Pfad. Also ist $q_k \in \textsc{eclose}(q_0)$ und somit $\hat{\delta}(q_0, \varepsilon) \cap F \neq \emptyset$.
Induktionsschritt: Sei $q \in Q$ beliebig. Wir betrachten Wörter $w = \sigma v$ mit $\sigma \in \Sigma$, $|v| = n$ und damit $|w| = n + 1$. Wir setzen voraus, dass (4.8) für alle $|x| \leq n$ gilt.

Da w mindestens die Länge 1 hat, müssen wir nur Ausdrücke betrachten, die Wörter der Länge 1 erzeugen können. Damit fällt $\mathtt{Fin[q]}$ raus. `Run` muss ebenfalls nicht betrachtet werden, da der Kombinator nicht den gewünschten Typ erzeugen kann. Es werden nun Ausdrücke der Form $e = \mathtt{E[q_0, q_1]}$ ... $\mathtt{E[q_{k-1}, q_k]}$ $\mathtt{D[q_k, \sigma, p]}$ $e' : \mathtt{St(q_0)} \to \mathtt{Word}(\mathrm{list}(w))$ für $k \geq 0$ betrachtet.

Für den Ausdruck e' gilt $\Gamma_{\mathcal{A}} \vdash e' : \mathtt{St(p)} \to \mathtt{Word}(\mathrm{list}(v))$. Aufgrund der Induktionsvoraussetzung und e' können wir folgern, dass $\hat{\delta}(p, v) \cap F \neq \emptyset$ ist. Wegen der Existenz von $\mathtt{D[q_k, \sigma, p]}$ gilt $p \in \delta(q_k, \sigma)$. Damit gilt:

$$\begin{aligned} \hat{\delta}(p, v) \cap F \neq \emptyset &\implies (\bigcup_{p \in P} \hat{\delta}(p, v)) \cap F \neq \emptyset \text{ mit } P = \textsc{eclose}(\bigcup_{q'_k \in \textsc{eclose}(q_k)} \delta(q'_k, \sigma)) \\ &\implies \hat{\delta}(q_k, \sigma v) \cap F \neq \emptyset \end{aligned}$$

Wir haben nun, dass $\hat{\delta}(q_k, w) \cap F \neq \emptyset$ ist. Wir wollen diese Eigenschaft aber für q_0 zeigen. Parallel zum Beweis im Induktionsanfang gibt es einen ϵ-Pfad von q_0 nach q_k. Damit gilt $q_k \in \textsc{eclose}(q_0)$ und schließlich $\hat{\delta}(q_0, w) \cap F \neq \emptyset$. □

4.2.7 Lemma. *Für einen ε-NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ und alle $w \in \Sigma^*$ sowie $q \in Q$ gilt:*

$$\hat{\delta}(q, w) \cap F \neq \emptyset \implies \exists e(\Gamma_{\mathcal{A}} \vdash e : \mathtt{St(q)} \to \mathtt{Word}(\mathrm{list}(w))) \tag{4.9}$$

Beweis. Wir zeigen dies ebenfalls mit einer Induktion über der Länge n der Wörter w.
Induktionsanfang: Für $n = 0$ gilt $w = \varepsilon$. Sei $q \in Q$ beliebig. Wir nehmen an, dass $\hat{\delta}(q, \varepsilon) \cap F \neq \emptyset$ ist. Da $\hat{\delta}(q, \varepsilon) = \textsc{eclose}(q)$ ist, gibt es ein $q' \in \textsc{eclose}(q)$ mit $q' \in F$ und damit auch einen Kombinator $\mathtt{Fin[q']} : \mathtt{St(q')} \to \mathtt{Word}(\epsilon)$. Wir suchen allerdings einen Ausdruck $e : \mathtt{St(q)} \to \mathtt{Word}(\epsilon)$. Da $q' \in \textsc{eclose}(q)$ ist, können wir Lemma 4.2.4 verwenden, um einen ε-Pfad $S = (q, ..., q')$ zu erhalten. Aus S und $\mathtt{Fin[q']}$ erhalten wir mit Lemma 4.2.5, dass ein Ausdruck $e : \mathtt{St(q)} \to \mathtt{Word}(\epsilon)$ existiert.

Induktionsschritt: Sei $q \in Q$ beliebig. Wir betrachten Wörter $w = \sigma v$ der Länge $n+1$ mit $\sigma \in \Sigma$ und $v \in \Sigma^*$ mit $|v| = n$. Wir setzen voraus, dass (4.9) für alle $|x| \leq n$ gilt.
Wir können zunächst annehmen:

$$\hat{\delta}(q, \sigma v) \cap F \neq \emptyset \tag{4.10}$$

Wir führen uns (4.4) erneut vor Augen, mit $w = v$:

$$\hat{\delta}(q, \sigma v) = \bigcup_{p \in P} \hat{\delta}(p, v) \text{ mit } P = \textsc{eclose}(\bigcup_{q' \in \textsc{eclose}(q)} \delta(q', \sigma)) \tag{4.11}$$

Sei P wie in (4.11). Wir wählen ein $p \in P$ so, dass $\hat{\delta}(p, v) \cap F \neq \emptyset$ ist. Dies geht, da wegen (4.10) und der Gleichheit in (4.11) mindestens ein Zustand aus mindestens einem der $\hat{\delta}(p, v)$ akzeptierend sein muss. Wir verlangen außerdem, dass $p \in \delta(q', \sigma)$ für mindestens ein $q' \in \textsc{eclose}(q)$ gilt. Da der Epsilon-Abschluss eines Zustands diesen Zustand selbst beinhaltet, gibt es so ein p auch in P.

Es gibt also eine Transition von q' zu p. Also gibt es auch den Kombinator $\texttt{D}[\texttt{q}', \sigma, \texttt{p}]$ in Γ_δ. Wir wählen $e_1 = \texttt{D}[\texttt{q}', \sigma, \texttt{p}]\ e_2$ mit $e_2 : \texttt{St}(\texttt{p}) \to \texttt{Word}(\text{list}(v))$. Der Typ von e_1 ist $\texttt{St}(\texttt{q}') \to \texttt{Word}(\text{list}(w))$, wir suchen aber einen Ausdruck vom Typ $\texttt{St}(\texttt{q}) \to \texttt{Word}(\text{list}(w))$. Da $q' \in \textsc{eclose}(q)$ ist, gibt es mit Lemma 4.2.4 einen ε-Pfad $S = (q, ..., q')$. Aus Lemma 4.2.5 erhalten wir mit S und e_1 den gesuchten Ausdruck $e : \texttt{St}(\texttt{q}) \to \texttt{Word}(\text{list}(w))$.

Es bleibt zu zeigen, dass es ein e_2 mit dem gesuchten Typen gibt. Nach der Induktionsvoraussetzung gilt für v und alle $s \in Q$ wegen $|v| \leq n$:

$$\hat{\delta}(s, v) \cap F \neq \emptyset \implies \exists e(\Gamma_\mathcal{A} \vdash e : \texttt{St}(\texttt{s}) \to \texttt{Word}(\text{list}(v)))$$

Da $\hat{\delta}(p, v) \cap F \neq \emptyset$ ist, gibt es einen Ausdruck e_2, der den Typen $\texttt{St}(\texttt{p}) \to \texttt{Word}(\text{list}(v))$ inhabitiert. □

Mit den soeben bewiesenen Lemmata können wir nun Satz 4.2.3 beweisen. Der Beweis erfolgt parallel zu dem in Kapitel 3 geführten Beweis zu Satz 3.2.6, weshalb auf ausführliche Erklärungen verzichtet wird.

Beweis (Satz 4.2.3). Sei $w \in \Sigma^*$ beliebig. Wir zeigen zuerst folgende Implikation:

$$\exists e(\Gamma_\mathcal{A} \vdash e : \texttt{Word}(\text{list}(w))) \implies \hat{\delta}(q_0, w) \cap F \neq \emptyset$$

Es existiert ein $e : \texttt{Word}(\text{list}(w))$. Damit muss der Ausdruck die Form $e = \texttt{Run}\ e'$ haben, wobei $e' : \texttt{St}(\texttt{q0}) \to \texttt{Word}(\text{list}(w))$ gilt. Mit Lemma 4.2.6 ist schließlich $\hat{\delta}(q_0, w) \cap F \neq \emptyset$.

Wir zeigen nun die Gegenrichtung:

$$\hat{\delta}(q_0, w) \cap F \neq \emptyset \implies \exists e(\Gamma_\mathcal{A} \vdash e : \texttt{Word}(\text{list}(w)))$$

Wir können annehmen, dass $\hat{\delta}(q_0, w) \cap F \neq \emptyset$ ist. Mit Lemma 4.2.7 sichern wir die Existenz eines Ausdrucks $e' : \texttt{St(q0)} \to \texttt{Word}(\text{list}(w))$. Wir wählen $e = \texttt{Run}\ e'$ und schließen den Beweis der Implikation.

Wie im Beweis zu Satz 3.2.6 hat ω keinen Einfluss auf die Simulation. Satz 4.2.3 ist nun insgesamt bewiesen. □

4.3 Beispiel zum Testen von Anforderungen

Wie in der Einleitung dieser Arbeit angesprochen eignet sich die Simulation von Berechnungsmodellen dazu, zusätzliche Anforderungen an generierte Programme zu stellen. Konkret im Kontext der ε-NFAs wird das Programm nur dann generiert, wenn die Anforderung in Form eines Wortes vom Automaten akzeptiert wird. Der Grund dafür ist, dass wegen Satz 4.2.3 nur dann ein kombinatorischer Ausdruck gefunden werden kann.

In Anlehnung an [3] wollen wir in diesem Beispiel Variationen eines Spiels synthetisieren. Anstatt eines Solitaire betrachten wir aber ein fiktives Rollenspiel (RPG), bei dem der Spieler je nach Variation des Spiels verschiedene Fähigkeiten verwenden kann. Wir beschränken uns dabei auf zwei *aktive* Fähigkeiten, die aus einer Menge von Fähigkeiten ausgewählt werden.

Die mit dem ε-NFA zu modellierende Anforderung ist, dass der Charakter nur Fähigkeiten verwendet, die die Wörter „Fire“ und/oder „Ice“ enthalten. Wir wollen also Variationen des Spiels synthetisieren, in denen der Charakter nur Feuer- und Eiszauber verwenden kann.

Wir gehen nun wie folgt vor: Wir präsentieren zunächst das Repository Γ_F, welches den relevanten Teil des Spiels modelliert. Daraufhin wird der ε-NFA $\mathcal{A}$ definiert, der die oben genannte Anforderung testet. Wir vereinen Γ_F mit $\Gamma_\mathcal{A}$ und passen die Kombinatoren der Fähigkeiten so an, dass die Akzeptanz des Automaten Voraussetzung dafür ist, dass der Kombinator verwendet werden kann. Schließlich betrachten wir eine erfolgreiche Inhabitation. Während der Behandlung des Beispiels wird sich zudem zeigen, dass Kombinatoren von Fähigkeiten, die die Anforderung nicht erfüllen, nicht sinnvoll verwendet werden können.

4.3.1 Definition des Fähigkeiten-Repository

Das Repository Γ_F ist wie folgt definiert:

$$
\begin{aligned}
\Gamma_F = \{ & \\
& \texttt{Fireball} : \texttt{Skill} \\
& \texttt{WallOfIce} : \texttt{Skill} \\
& \texttt{IceLance} : \texttt{Skill}
\end{aligned}
$$

$$\texttt{Poison} : \texttt{Skill}$$
$$\texttt{DeepCut} : \texttt{Skill}$$
$$\texttt{SkillSet} : \texttt{Skill} \to \texttt{Skill} \to \texttt{SkillSet}$$
$$\}$$

Der Typ `Skill` ist als Typ für Fähigkeiten zu verstehen und wird hier aufgrund der mangelnden Relevanz nicht weiter definiert. Wir haben 5 Fähigkeiten: *Fireball*, *Wall of Ice*, *Ice Lance*, *Poison* und *Deep Cut*. `SkillSet` vereint zwei Fähigkeiten zu einer Menge von aktiven Fähigkeiten. Die Menge aller Fähigkeiten kann einfach dadurch erweitert werden, dass neue Kombinatoren hinzugefügt werden, die den Typ `Skill` haben.[2]

Es kann vorkommen, dass `SkillSet` auf zwei gleiche `Skill`-Werte angewandt wird, beispielsweise `SkillSet Fireball Fireball`. Dies zu verhindern ist nicht trivial und in diesem Kontext eher nebensächlich interessant, weshalb wir das Problem aufgrund der Komplexität des Beispiels unberührt lassen.

4.3.2 Teilwort-Automat und Repository $\Gamma_{\mathcal{A}}$

Der ε-NFA $\mathcal{A}$ in Abbildung 4.1 akzeptiert genau dann ein Wort, wenn in dem Wort das Teilwort Fire oder das Teilwort Ice vorkommt. Für $\mathcal{A}$ gilt $\Sigma = \{\mathrm{A}, ..., \mathrm{Z}, \mathrm{a}, ..., \mathrm{z}, \text{␣}\}$, wobei ␣ für Leerzeichen steht.

Die Erkennung von Teilwörtern in Strings lässt sich gut sowohl mit einem NFA (siehe [4], Kapitel 2.4.3) als auch mit einem ε-NFA (siehe [4], Beispiel 2.17) lösen. In den zitierten Beispielen werden allerdings Automaten betrachtet, die nur Wörter akzeptieren, die in den gesuchten Teilwörtern *enden*. Wir erlauben auch Wörter, die nach dem gesuchten Teilwort weitere beliebige Zeichen haben, indem die Endzustände die restlichen Zeichen konsumieren.

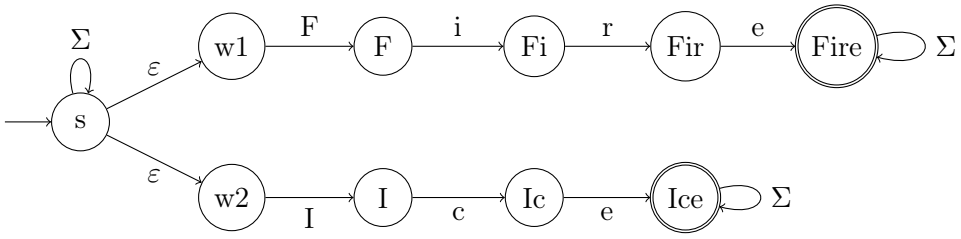

Abbildung 4.1: Der ε-NFA $\mathcal{A}$, der die Teilwörter Fire und Ice erkennt.

[2] Am Rande sei bemerkt, dass mit der Art, wie CLS synthetisiert, auch Fähigkeiten definiert werden können, die sich aus kleineren Komponenten zusammensetzen lassen; Ohne, dass etwas an den anderen Kombinatoren geändert werden muss. So könnte man z.B. eine Fähigkeit `Cut` : `DamageType` → `Skill` definieren, die einen variablen Schadenstyp (Feuer, Wasser, etc.) hat.

Wir konstruieren das Repository $\Gamma_{\mathcal{A}}$ nach dem Verfahren aus 4.2.1. Wir listen zuerst die Transitionen auf, die die Teilwörter erkennen:

$$
\begin{aligned}
\Gamma_1 = \{ & \\
& \texttt{Run} : (\texttt{St}(\texttt{s}) \to \texttt{Word}(\alpha)) \to \texttt{Word}(\alpha) \\
& \texttt{E}[\texttt{s},\texttt{w1}] : (\texttt{St}(\texttt{w1}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{s}) \to \texttt{Word}(\alpha)) \\
& \texttt{D}[\texttt{w1},\texttt{F},\texttt{F}] : (\texttt{St}(\texttt{F}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{w1}) \to \texttt{Word}(\texttt{F}(\alpha))) \\
& \texttt{D}[\texttt{F},\texttt{i},\texttt{Fi}] : (\texttt{St}(\texttt{Fi}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{F}) \to \texttt{Word}(\texttt{i}(\alpha))) \\
& \texttt{D}[\texttt{Fi},\texttt{r},\texttt{Fir}] : (\texttt{St}(\texttt{Fir}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{Fi}) \to \texttt{Word}(\texttt{r}(\alpha))) \\
& \texttt{D}[\texttt{Fir},\texttt{e},\texttt{Fire}] : (\texttt{St}(\texttt{Fire}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{Fir}) \to \texttt{Word}(\texttt{e}(\alpha))) \\
& \texttt{Fin}[\texttt{Fire}] : \texttt{St}(\texttt{Fire}) \to \texttt{Word}(\epsilon) \\
& \texttt{E}[\texttt{s},\texttt{w2}] : (\texttt{St}(\texttt{w2}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{s}) \to \texttt{Word}(\alpha)) \\
& \texttt{D}[\texttt{w2},\texttt{I},\texttt{I}] : (\texttt{St}(\texttt{I}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{w2}) \to \texttt{Word}(\texttt{I}(\alpha))) \\
& \texttt{D}[\texttt{I},\texttt{c},\texttt{Ic}] : (\texttt{St}(\texttt{Ic}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{I}) \to \texttt{Word}(\texttt{c}(\alpha))) \\
& \texttt{D}[\texttt{Ic},\texttt{e},\texttt{Ice}] : (\texttt{St}(\texttt{Ice}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{Ic}) \to \texttt{Word}(\texttt{e}(\alpha))) \\
& \texttt{Fin}[\texttt{Ice}] : \texttt{St}(\texttt{Ice}) \to \texttt{Word}(\epsilon) \\
\} &
\end{aligned}
$$

Zudem gibt es für jedes Zeichen eine Transition vom Startzustand s zu sich selbst:

$$\Gamma_2 = \{\texttt{D}[\texttt{s},\texttt{a},\texttt{s}] : (\texttt{St}(\texttt{s}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{s}) \to \texttt{Word}(\texttt{a}(\alpha))) \mid a \in \Sigma\}$$

Schließlich haben die Endzustände ebenfalls für alle Zeichen Transitionen zu sich selbst:

$$\Gamma_3 = \{\texttt{D}[\texttt{q},\texttt{a},\texttt{q}] : (\texttt{St}(\texttt{q}) \to \texttt{Word}(\alpha)) \to (\texttt{St}(\texttt{q}) \to \texttt{Word}(\texttt{a}(\alpha))) \mid a \in \Sigma, q \in F\}$$

Insgesamt ergibt sich das Repository für den Automaten als $\Gamma_{\mathcal{A}} = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$.

4.3.3 Verschmelzung von Γ_F und $\Gamma_{\mathcal{A}}$

Wir passen Γ_F so an, dass ein Fähigkeiten-Kombinator zum Typ $\texttt{Word}(\text{list}(w)) \to \texttt{Skill}$ erweitert wird, wobei w dem Namen der Fähigkeit entspricht. Damit stellen wir sicher, dass der Automat akzeptieren muss, bevor wir einen Ausdruck vom Typ `Skill` erhalten können. Denn akzeptiert der Automat nicht, so gibt es nach Satz 4.2.3 keinen Ausdruck zum jeweiligen Worttypen und damit kein Argument, welches in die Funktion eingesetzt werden kann.

$$
\begin{aligned}
\Gamma'_F = \{ & \\
& \texttt{Fireball} : \texttt{Word}(\text{list}(\underline{\text{Fireball}})) \to \texttt{Skill}
\end{aligned}
$$

$$\begin{aligned}
\texttt{WallOfIce} &: \texttt{Word}(\text{list}(\underline{\text{Wall_of_Ice}})) \to \texttt{Skill} \\
\texttt{IceLance} &: \texttt{Word}(\text{list}(\underline{\text{Ice_Lance}})) \to \texttt{Skill} \\
\texttt{Poison} &: \texttt{Word}(\text{list}(\underline{\text{Poison}})) \to \texttt{Skill} \\
\texttt{DeepCut} &: \texttt{Word}(\text{list}(\underline{\text{Deep_Cut}})) \to \texttt{Skill} \\
\texttt{SkillSet} &: \texttt{Skill} \to \texttt{Skill} \to \texttt{SkillSet}
\end{aligned}$$

$\}$

Γ'_F enthält die Schnittstelle zum Automaten, aber noch nicht den Automaten selbst. Mit der folgenden Verschmelzung erhalten wir das gewünschte Repository:

$$\Gamma_G = \Gamma'_F \cup \Gamma_{\mathcal{A}}$$

4.3.4 Inhabitation in Γ_G

Angenommen, wir suchen einen Ausdruck s mit $\Gamma_G \vdash s : \texttt{Skill}$, der eine Fähigkeit darstellt. Der einzige Weg in Γ_G, um an einen Ausdruck vom Typ `Skill` zu gelangen, ist über Funktionen $\texttt{Word}(\text{list}(w)) \to \texttt{Skill}$. Wegen Satz 4.2.3 können wir direkt sagen, *welche* der Kombinatoren in Frage kommen: Der Automat $\mathcal{A}$ muss das jeweilige Wort akzeptieren. Da $\mathcal{A}$ nur Wörter akzeptiert, die die Teilwörter Fire oder Ice enthalten, lässt sich nur zu folgenden Kombinatoren ein `Word`-Ausdruck finden: `Fireball`, `WallOfIce` und `IceLance`.

Wir suchen zunächst Ausdrücke e_1 und e_2, die den Typ $\texttt{Word}(\text{list}(\underline{\text{Fireball}}))$ bzw. $\texttt{Word}(\text{list}(\underline{\text{Wall\_of\_Ice}}))$ inhabitieren. Dabei verfolgen wir die Ausführung des Automaten für das jeweilige Wort. Wir haben:

$$\begin{aligned}
e_1 = \texttt{Run}\ &\texttt{E[s, w1]}\ \texttt{D[w1, F, F]}\ \texttt{D[F, i, Fi]}\ \texttt{D[Fi, r, Fir]}\ \texttt{D[Fir, e, Fire]} \\
&\texttt{D[Fire, b, Fire]}\ \texttt{D[Fire, a, Fire]}\ \texttt{D[Fire, l, Fire]}\ \texttt{D[Fire, l, Fire]} \\
&\texttt{Fin[Fire]}
\end{aligned}$$

Zuerst wird das Teilwort Fire erkannt. Danach werden die restlichen Zeichen (ball) konsumiert. Die Erkennung von Wall_of_Ice verläuft ähnlich, mit dem Unterschied, dass zuerst einige Zeichen gelesen werden müssen, bevor das Teilwort Ice erkannt werden kann.

$$\begin{aligned}
e_2 = \texttt{Run}\ &\texttt{D[s, W, s]}\ \texttt{D[s, a, s]}\ \texttt{D[s, l, s]}\ \texttt{D[s, l, s]} \\
&\texttt{D[s, _, s]}\ \texttt{D[s, o, s]}\ \texttt{D[s, f, s]}\ \texttt{D[s, _, s]} \\
&\texttt{E[s, w2]}\ \texttt{D[w2, I, I]}\ \texttt{D[I, c, Ic]}\ \texttt{D[Ic, e, Ice]} \\
&\texttt{Fin[Ice]}
\end{aligned}$$

Letztlich erhalten wir zwei Ausdrücke vom Typ `Skill`:

$$\begin{aligned}
s_1 &= \texttt{Fireball}\ e_1 \\
s_2 &= \texttt{WallOfIce}\ e_2
\end{aligned}$$

Wir wollen an dieser Stelle noch einmal betonen, dass nur Ausdrücke vom Typ `Skill` gefunden werden können, wenn die Fähigkeit die Anforderung erfüllt, die der Automat überprüft. Wir haben oben zusätzlich gesehen, wie ein Ausdruck für eine Fähigkeit aussieht, die die Anforderung erfüllt.

4.3.5 Vorteile und Nachteile des Verfahrens

Wir wollen nun einige Vorteile und Nachteile des Verfahrens aus diesem Beispiel diskutieren.

Ein Vorteil ist, dass das hier vorgestellte Verfahren auf beliebige Automaten anwendbar ist. Man braucht insbesondere nur das Repository $\Gamma_{\mathcal{A}}$ verändern, um die Anforderung anzupassen. Die Schnittstelle zwischen den restlichen Kombinatoren und $\Gamma_{\mathcal{A}}$ bleibt dabei gleich.

Es ist auch möglich, mehrere Automaten zu benutzen, um mehrere Anforderungen umzusetzen. Möchte man dies tun, muss man aber die Typen und die Kombinatoren von einigen Automaten umbenennen, damit sich die Namen nicht überlappen.

In [3] (dem Solitaire) wurden die Kombinatoren zu den gewünschten Features außerhalb der Inhabitation ausgewählt, indem nur die gewünschten Kombinatoren in das Repository aufgenommen wurden. Das ist natürlich ein sehr flexibler Ansatz, der aber auch verlangt, dass das Repository bei jeder unterschiedlichen Feature-Kombination neu zusammengebaut wird. Im Unterschied zu dem Ansatz von Heineman wird in diesem Beispiel die Auswahl der “Features“ während der Inhabitation getroffen, durch die Simulation eines Automaten. Damit lässt sich die Anforderung lokalisiert auf das Repository des Automaten betrachten und verändern, ohne den Rest des Repositories verändern zu müssen.

Ein großer Nachteil dieses Verfahrens ist das Vorkommen von überflüssigen kombinatorischen Ausdrücken, die die Ausführung des Automaten repräsentieren und den `Word`-Typen inhabitieren. In diesem Anwendungsfall sind diese Ausdrücke nur für die Simulation relevant, vergrößern aber das Ergebnis der Inhabitation.

Ein weiterer Nachteil ist, dass Eigenschaften der Bedeutungen hinter den Kombinatoren explizit in den Typ geliftet werden müssen. So musste in diesem Beispiel der Name der Fähigkeit in den Typ des Fähigkeiten-Kombinators geschrieben werden.

Schließlich besteht auch bei der Performance ein Nachteil. Die praktische Simulation dieses Beispiels (Kapitel 6) hat 2369 Sekunden gedauert. Ansätze, die den ε-NFA direkt simulieren oder die Namen als Strings vergleichen, wären damit offensichtlich schneller.

4.3.6 Fazit

Wir haben in diesem Beispiel gesehen, wie wir einen ε-NFA simulieren können, um Anforderungen an Teilprogramme zu stellen. Wir haben die Anforderung in einem ε-NFA kodiert und ein Repository dazu konstruiert. Wir haben das Repository des Anwendungs-

falls geeignet angepasst und mit dem Repository des Automaten verschmolzen, woraufhin wir eine Inhabitation in diesem zusammenfassenden Repository gesehen haben. Letztlich haben wir Vor- und Nachteile des Ansatzes diskutiert.

Kapitel 5

Simulation von Baumgrammatiken

5.1 Reguläre Baumgrammatiken

Baumgrammatiken sind ein Formalismus, mit dem man Bäume beschreiben kann, die aus einem bestimmten Alphabet zusammengesetzt sind. Ähnlich wie ein DFA eine Menge von Wörtern beschreibt, beschreibt eine Baumgrammatik eine Menge von Bäumen. Die Bäume bestehen dabei aus Symbolen eines Rangalphabets, d.h. aus Symbolen, die eine bestimmte Anzahl an Teilbäumen als Kinder haben. Die hier betrachteten *regulären* Baumgrammatiken haben die Besonderheit, dass auf der linken Seite einer Produktion nur Nichtterminale stehen können, die außerdem die Stelligkeit 0 haben müssen.

Reguläre Baumgrammatiken sollten nicht mit den regulären Wortgrammatiken verwechselt werden, da das Wort „regulär“ eine andere Bedeutung hat. In der Tat sind reguläre Baumgrammatiken eher mit kontextfreien Wortgrammatiken zu vergleichen (siehe [1], Theorem 2.4.3).

Im Vergleich zu kontextfreien Wortgrammatiken haben reguläre Baumgrammatiken den Vorteil, dass Bäume Information strukturierter darstellen können als Zeichenketten. Für die in dieser Arbeit betrachteten Anwendungsfälle wie Codegenerierung oder das Testen von Anforderungen bietet es sich eher an, die nötige Information strukturiert in einem Baum darzustellen.

Wir definieren zunächst den Begriff des Rangalphabets und der Terme, bevor wir zu der Definition von regulären Baumgrammatiken kommen.

5.1.1 Definition. Ein **Rangalphabet** (siehe [1], Seite 15) ist ein Paar $(\mathcal{F}, \text{Arity})$, wobei $\mathcal{F}$ die endliche Menge der **Symbole** des Alphabets ist und $\text{Arity} : \mathcal{F} \to \mathbb{N}_0$ die **Stelligkeit** der Symbole aus $\mathcal{F}$ festlegt. Die Menge aller Symbole mit Stelligkeit k nennen wir $\mathcal{F}_k$. Symbole $f \in \mathcal{F}_0$ heißen **Konstanten**. $\mathcal{F}$ muss mindestens eine Konstante enthalten.

Notation. Ein Symbol $f \in \mathcal{F}_1$ kürzen wir mit $f()$ ab, ein Symbol $g \in \mathcal{F}_2$ mit $g(,)$, ein Symbol $h \in \mathcal{F}_3$ mit $h(,,)$, und so weiter.

5.1.2 Definition. Die Menge der **Terme** $T(\mathcal{F})$ (siehe [1], Seite 15)[1] über dem Rangalphabet[2] $\mathcal{F}$ ist die kleinste Menge, die folgende Eigenschaften erfüllt:

$$\mathcal{F}_0 \subseteq T(\mathcal{F}) \tag{5.1}$$

$$f(t_1, ..., t_k) \in T(\mathcal{F}) \text{ wenn } f \in \mathcal{F}_k \text{ für } k \geq 1 \text{ und } t_1, ..., t_k \in T(\mathcal{F}) \tag{5.2}$$

Terme sind die Grundstruktur, mit der wir Bäume darstellen können. Streng genommen sind Terme keine Bäume, welche in [1] anders definiert sind. Allerdings kann man jeden Term in einen Baum überführen und umgekehrt. Diese Isomorphie erlaubt es uns, die für uns im Rahmen von CLS eher nützliche Termstruktur zu betrachten und trotzdem über Bäume zu reden.

5.1.3 Bemerkung. Kombinatorische Ausdrücke werden in der Literatur auch als „Terme“ bezeichnet, wie z.B. in [7]. Wir reservieren das Wort „Term“ **ausschließlich** für die Bezeichnung von Termen einer Baumgrammatik, **nicht** als Synonym für kombinatorische Ausdrücke.

5.1.4 Definition. Die **Höhe** eines Terms $t \in T(\mathcal{F})$, height(t), ist folgendermaßen induktiv definiert (siehe [1], Seite 16):

$$\begin{aligned} \text{height}(f) &= 1 \text{ für } f \in \mathcal{F}_0 \\ \text{height}(f(t_1, ..., t_k)) &= 1 + \max(\{\text{height}(t_i) \mid 1 \leq i \leq k\}) \text{ für } f \in \mathcal{F}_k \text{ mit } k \geq 1 \end{aligned}$$

5.1.5 Definition. Eine **reguläre Baumgrammatik** (siehe [1], Kapitel 2.1.1) ist ein 4-Tupel $(S, N, \mathcal{F}, R)$ mit:

- Einem **Startsymbol** $S \in N$.
- Einer Menge N von **Nichtterminalen**.
- Einer Menge $\mathcal{F}$ von **Terminalen**.
- Einer Menge $R \subseteq N \times T(N \cup \mathcal{F})$ von **Produktionen** der Art $A \to \phi$.

Jedes Symbol aus $N \cup \mathcal{F}$ muss eine endliche Stelligkeit haben. Für S gilt $Arity(S) = 0$. Außerdem müssen N und $\mathcal{F}$ disjunkt sein.

Notation. Parallel zur üblichen Notation von kontextfreien Wortgrammatiken können wir eine reguläre Baumgrammatik spezifizieren, indem wir die Produktionen auflisten. Die Symbole A_i aller Produktionen $A_i \to \phi_i$ gelten dabei als Nichtterminale. Alle Symbole, die in mindestens einem ϕ_i vorkommen, aber kein Nichtterminal sind, gelten als Terminale. Das Nichtterminal der ersten Produktion gilt als Startsymbol S.

[1] Wir haben die Variablen aus der Definition entfernt, da diese für unsere Zwecke nicht nützlich sind.

[2] Wir setzen die Funktion Arity voraus, ohne sie explizit aufzuführen.

5.1.6 Beispiel. Wir betrachten die Produktionen der folgenden regulären Baumgrammatik G_{List} (siehe [1], Beispiel 2.1.1), die Listen von nicht-negativen Zahlen beschreibt:

$$List \to nil$$
$$List \to cons(Nat, List)$$
$$Nat \to 0$$
$$Nat \to s(Nat)$$

Nach der oben genannten Notationskonvention ergibt sich eine Grammatik mit $S = List$, $N = \{List, Nat\}$ und $\mathcal{F} = \{0, nil, s(), cons(,)\}$. Weiterhin ist $cons(s(0), nil)$ ein Term aus $T(\mathcal{F} \cup N)$.

Mit den bisherigen Definitionen können wir die Akzeptanz des Berechnungsmodells Baumgrammatik definieren. Wir sagen, dass eine reguläre Baumgrammatik genau dann einen Term aus $T(\mathcal{F})$ akzeptiert, wenn der Term mithilfe der Produktionen der Grammatik aus dem Startsymbol gebildet werden kann. Um diese Idee zu formalisieren, müssen wir zunächst den Begriff der Ableitung definieren.

5.1.7 Definition. Sei $G = (S, N, \mathcal{F}, R)$ eine reguläre Baumgrammatik. Die **Ableitungsrelation** $\to_G$ (siehe [1], Seite 52) ist eine zweistellige Relation zwischen Termen aus $T(\mathcal{F} \cup N)$. Es gilt $s \to_G t$ genau dann, wenn es eine Produktion $A \to \phi$ in R gibt und ein Vorkommen von A in s, das in t durch ϕ ersetzt wurde. Der Term t darf sich nur an dieser Stelle von s unterscheiden.[3] Kann man einen Term t in einem oder mehr Schritten aus einem Term s ableiten, so schreiben wir $s \overset{+}{\to}_G t$. Für die Ableitung in beliebig vielen Schritten schreiben wir $s \overset{*}{\to}_G t$.

Notation. Wir können die Angabe der Grammatik bei der Ableitung weglassen, wenn diese aus dem Kontext ersichtlich ist. Wir schreiben dann $\to$ anstatt $\to_G$.

5.1.8 Beispiel. Ein Beispiel zu der oben definierten Grammatik G_{List} ist in [1], Example 2.1.2 zu finden. Wir haben folgende Ableitung:

$$List \to cons(Nat, List) \to cons(s(Nat), List) \to cons(s(Nat), nil) \to cons(s(0), nil)$$

5.1.9 Definition. Eine reguläre Baumgrammatik $G = (S, N, \mathcal{F}, R)$ **akzeptiert** einen Term $t \in T(\mathcal{F})$ genau dann, wenn $S \overset{+}{\to}_G t$ gilt.[4]

Zu beachten ist hier, dass wir nur Terme aus $T(\mathcal{F})$ akzeptieren, also nur solche, die ausschließlich aus Terminalen bestehen.

[3] Die Ableitungsrelation wird in [1] mithilfe von Kontexten definiert, die die Ableitung etwas weiter formalisieren. Die Definition über Kontexte stimmt allerdings semantisch mit unserer Definition überein.

[4] Zu beachten ist, dass die Akzeptanz in [1] nicht explizit definiert wird. Wir definieren diese im Rahmen der Simulation genau parallel zu den anderen Berechnungsmodellen über die vom Modell erzeugte Sprache. Die Sprache an sich wird hier wie gehabt nicht definiert, da die Definition in dieser Arbeit – außerhalb der Definition der Akzeptanz – nicht benötigt wird.

5.2 Konstruktionsverfahren für reguläre Baumgrammatiken

Auch für reguläre Baumgrammatiken definieren wir ein Konstruktionsverfahren, welches ein Repository erzeugt, das eine konkrete Baumgrammatik in CLS simuliert. Dazu müssen wir aber zunächst definieren, wie wir Terme als Typen kodieren wollen.

5.2.1 Repräsentation von Termen

5.2.1 Definition. Sei $G = (S, N, \mathcal{F}, R)$ eine reguläre Baumgrammatik. Wir definieren folgende Typkonstruktoren zu G:

- Für jedes Nichtterminal $n \in N$ gibt es eine Typkonstante `n`.
- Für jedes Terminal $f \in \mathcal{F}$ gibt es einen Typkonstruktor `f`, dessen Stelligkeit der Stelligkeit Arity(f) entspricht.
- Außerdem gibt es einen Typkonstruktor `Term(t)`, der einen Typen `t` als Term ausweist (parallel zu `Word`).

Wir benutzen die oben definierten Typkonstruktoren und Typkonstanten, um einen Term aus $T(\mathcal{F} \cup N)$ als Typ darzustellen. Dazu definieren wir induktiv eine Funktion rep, die einen Term in den dazugehörigen Typen umwandelt:

$$\begin{aligned} \text{rep}(n) &= \texttt{n} \text{ für } n \in N \\ \text{rep}(f) &= \texttt{f} \text{ für } f \in \mathcal{F}_0 \\ \text{rep}(f(t_1, ..., t_k)) &= \texttt{f}(\text{rep}(t_1), ..., \text{rep}(t_k)) \text{ für } k \geq 1, f \in \mathcal{F}_k \end{aligned}$$

Der Typ zu einem Term t ist dann `Term`(rep(t)).

5.2.2 Beispiel. Wir betrachten die Grammatik G_{List} aus Beispiel 5.1.6. In der folgenden Tabelle wurden die Terme auf der linken Seite mithilfe von Definition 5.2.1 in Typen überführt:

$cons(s(0), nil)$	`Term(cons(s(0),nil))`
$cons(s(Nat), List)$	`Term(cons(s(Nat),List))`
$cons(0, cons(s(Nat), nil))$	`Term(cons(0,cons(s(Nat),nil)))`

Wie man sehen kann, stimmt die Repräsentation von Termen im Typsystem intuitiv mit den ursprünglichen Termen überein.

5.2.3 Bemerkung. Die mehrstelligen Typkonstruktoren, die für die Repräsentation von Terminalen ab einer Stelligkeit von 2 verwendet werden, sind theoretisch komplizierter als einstellige Typkonstruktoren und Typkonstanten, werden in CLS aber bereits (experimentell) unterstützt.[5] Aufgrund der Unterstützung in CLS und dem Vorteil, dass eine

[5] Quelle: Persönliche Kommunikation mit Andrej Dudenhefner.

Repräsentation von Termen in einer intuitiven Darstellung weitaus nutzbarer und auch theoretisch besser zu betrachten ist als eine beispielsweise listenähnliche Kodierung der Terme, benutzen wir mehrstellige Typkonstruktoren trotz der möglicherweise damit behafteten Schwierigkeiten.

5.2.2 Heranführung anhand eines Beispiels

Das Konstruktionsverfahren für reguläre Baumgrammatiken ist komplizierter als die Konstruktionsverfahren für DFAs und ε-NFAs. Deshalb wollen wir den Leser zunächst anhand eines Beispiels an die Konstruktionsidee heranführen. Wir werden sehen, dass die Simulation von regulären Baumgrammatiken in CLS nicht weniger intuitiv als die Simulation von DFAs und ε-NFAs ist. Wir definieren ein Repository $\Gamma_{G_{List}}$ für die reguläre Baumgrammatik G_{List} aus Beispiel 5.1.6 wie folgt:

$$\texttt{List}_1 : \texttt{List} \to \texttt{Term}(\texttt{nil})$$
$$\texttt{List}_2 : (\texttt{Nat} \to \texttt{Term}(\alpha_1)) \to (\texttt{List} \to \texttt{Term}(\alpha_2)) \to (\texttt{List} \to \texttt{Term}(\texttt{cons}(\alpha_1, \alpha_2)))$$
$$\texttt{Nat}_1 : \texttt{Nat} \to \texttt{Term}(\texttt{0})$$
$$\texttt{Nat}_2 : (\texttt{Nat} \to \texttt{Term}(\alpha_1)) \to (\texttt{Nat} \to \texttt{Term}(\texttt{s}(\alpha_1)))$$
$$\texttt{Start} : (\texttt{List} \to \texttt{Term}(\alpha)) \to \texttt{Term}(\alpha)$$

Wir wollen folgende Besonderheiten anmerken:

- Produktionskombinatoren lassen sich nicht so einfach bezeichnen wie Transitionskombinatoren von DFAs oder ε-NFAs. Für eine Produktion $A \to \phi$ könnte man den Kombinator eindeutig als $\texttt{A}[\phi]$ bezeichnen, allerdings führt das möglicherweise zu sehr langen und sehr unhandlichen Namen. Deshalb nummerieren wir die Kombinatoren stattdessen. Gibt der Kontext, in dem eine konkrete Grammatik definiert ist, mehr Information für die Bezeichnung der Kombinatoren her, kann man diese natürlich umbenennen.
- Kombinatoren zu Produktionen $A \to f$ mit $f \in \mathcal{F}_0$ bilden parallel zu den `Fin`-Kombinatoren der DFAs und ε-NFAs das Ende einer Berechnung.
- Der Kombinator `Start` sorgt dafür, dass gültige Terme nur vom Startsymbol aus generiert werden können. Dies ist parallel zu dem Kombinator `Run` aus den Abschnitten zu DFAs und ε-NFAs zu sehen.
- Parallel zu der schrittweisen Berechnung von Wörtern bei der Simulation von DFAs und ε-NFAs setzen wir mit Kombinatoren wie $\texttt{List}_2$ schrittweise Terme aus Teiltermen zusammen. Der Unterschied zu DFAs und ε-NFAs ist, dass ein Produktionskombinator mehr als einen einzigen Teilterm verarbeiten kann, wie z.B. $\texttt{List}_2$, der zwei Teilterme benötigt: Einen Term, der eine natürliche Zahl repräsentiert, $\texttt{Nat} \to \texttt{Term}(\alpha_1)$, und den Rest der Liste als Term, $\texttt{List} \to \texttt{Term}(\alpha_2)$.

- In einer Produktion $A \to \phi$ kann ϕ in der Regel aus einem beliebig langen Term bestehen. Das fällt bereits bei dem Kombinator $\texttt{List}_2$ auf. Wir könnten uns aber auch weitere Produktionen ausdenken, die weitaus komplizierter sind.

Die letzten beiden Anmerkungen weisen darauf hin, dass wir folgende Hilfsfunktionen benötigen, um das Konstruktionsverfahren ausreichend formal definieren zu können:

- Da eine Produktion beliebig viele Nichtterminale auf der rechten Seite beinhalten kann, und wir dem Kombinator im Sinne einer Funktion zu jedem Nichtterminal genau einen weiteren Parameter geben müssen (wie z.B. zwei Parameter bei $\texttt{List}_2$), benötigen wir eine Funktion nt, die alle Nichtterminale in der richtigen Reihenfolge aus einem Term extrahiert.
- Da in Produktionen beliebig lange Terme auf der rechten Seite stehen können, in denen die Nichtterminale während der Konstruktion des Repositories durch Typvariablen ersetzt werden müssen, benötigen wir eine Funktion, die diese Nichtterminale durch die dazugehörigen Typvariablen ersetzt.

Mit diesen Erkenntnissen widmen wir uns nun der Definition der Hilfsfunktionen und des Konstruktionsverfahrens.

5.2.3 Konstruktionsverfahren

Wir definieren zunächst die Hilfsfunktionen nt und rep_α. Im Folgenden nehmen wir eine reguläre Baumgrammatik $G = (S, N, \mathcal{F}, R)$ an.

5.2.4 Definition. Wir definieren die Funktion $\text{nt} : T(\mathcal{F} \cup N) \to [N]$ wie folgt. Dabei ist $[\tau]$ der Typ von Listen mit Elementen vom Typ τ. $++$ ist der Konkatenationsoperator. Wir definieren die Konkatenation über 0 Listen als die leere Liste $[\,]$.

$$\text{nt}(n) = [n] \text{ für } n \in N \tag{5.3}$$

$$\text{nt}(f(t_1, ..., t_k)) = \text{nt}(t_1) ++ ... ++ \text{nt}(t_k) \text{ für } k \geq 0, f \in \mathcal{F}_k \tag{5.4}$$

Insbesondere sei angemerkt, dass die Verwendung von Listen die Reihenfolge der Nichtterminale erhält. Diese Eigenschaft benötigen wir für die Definition des Konstruktionsverfahrens.

5.2.5 Beispiel. Die Funktion nt soll die Nichtterminale aus einem Term extrahieren. Wir betrachten dazu die rechte Seite der Produktion $\mathit{List} \to \mathit{cons}(\mathit{Nat}, \mathit{List})$ aus Beispiel 5.1.6:

$$\text{nt}(\mathit{cons}(\mathit{Nat}, \mathit{List})) = [\mathit{Nat}] ++ [\mathit{List}] = [\mathit{Nat}, \mathit{List}]$$

5.2.6 Definition. Wir definieren die Funktion rep_α zu einem Term $t \in T(\mathcal{F} \cup N)$ wie folgt: Sei $\tau = \text{rep}(t)$. Sei $\text{nt}(t) = [n_1, ..., n_k]$ für ein $k \geq 0$. Ein Typ τ' ergibt sich daraus, dass wir die Typkonstanten $\mathtt{n_1}, ..., \mathtt{n_k}$ in τ, die zu den Nichtterminalen $n_1, ..., n_k$ gehören,

von links nach rechts durch die Typvariablen $\alpha_1, ..., \alpha_k$ ersetzen. Dabei muss jedes α_i an die genaue Stelle von $\mathtt{n_i}$ gesetzt werden und nicht etwa an eine andere Stelle $\mathtt{n_j}$, bei der möglicherweise $n_i = n_j$ gilt. Es ist dann $\text{rep}_\alpha(t) = \tau'$.

5.2.7 Beispiel. Wir betrachten wieder den Term aus Beispiel 5.1.6. Die Anwendung von rep_α sieht folgendermaßen aus:

$$\text{rep}_\alpha(cons(Nat, List)) = \mathtt{cons}(\alpha_1, \alpha_2)$$

Es gibt auch Terme, die ein Nichtterminal mehrmals enthalten. Wir betrachten als Beispiel den Term $cons(Nat, cons(Nat, List))$, in dem Nat zwei mal vorkommt. Es gilt:

$$\text{rep}_\alpha(cons(Nat, cons(Nat, List))) = \mathtt{cons}(\alpha_1, \mathtt{cons}(\alpha_2, \alpha_3))$$

Die Nummerierung der α_i ergibt sich aus der Ersetzungsweise der Nichtterminale, die oben definiert wurde. Man könnte auch sagen, dass die Nichtterminale von links nach rechts durch Variablen α_i ersetzt werden, wobei i bei 1 beginnt und bei jeder Ersetzung inkrementiert wird.

5.2.8 Definition. Für eine reguläre Baumgrammatik $G = (S, N, \mathcal{F}, R)$ erzeugen wir das Repository Γ_G wie folgt. Für jede Produktion $A \to \phi \in R$ enthält Γ_G einen Kombinator $\mathtt{A_{id}}$. Es sei $\text{nt}(\phi) = [n_1, ..., n_k]$. Dann gilt:

$$\begin{aligned}\mathtt{A_{id}} : (\text{rep}(n_1) \to \mathtt{Term}(\alpha_1)) \to ... \to (\text{rep}(n_k) \to \mathtt{Term}(\alpha_k))\\ \to (\text{rep}(A) \to \mathtt{Term}(\text{rep}_\alpha(\phi)))\end{aligned}$$

Die Variable `id` wählen wir so, dass jeder Kombinator einzigartig benannt wird. Es bietet sich dort z.B. eine sequentielle Nummerierung mit natürlichen Zahlen an.

Neben den Kombinatoren für Produktionen gibt es einen weiteren Kombinator `Start` in Γ_G, der sicherstellt, dass die Simulation beim Startsymbol S beginnt:

$$\mathtt{Start} : (\text{rep}(S) \to \mathtt{Term}(\alpha)) \to \mathtt{Term}(\alpha)$$

5.2.9 Beispiel. Wir wollen für die Produktion $List \to cons(Nat, List)$ aus Beispiel 5.1.6 einen Kombinator $\mathtt{List_2}$ definieren. Sei $\phi = cons(Nat, List)$. In den obigen Beispielen haben wir bereits folgende Eigenschaften gesehen:

$$\begin{aligned}\text{nt}(\phi) &= [Nat, List]\\ \text{rep}_\alpha(\phi) &= \mathtt{cons}(\alpha_1, \alpha_2)\end{aligned}$$

Nach Definition 5.2.8 haben wir $n_1 = Nat$ und $n_2 = List$. Insgesamt ergibt sich folgender Kombinator:

$$\mathtt{List_2} : (\mathtt{Nat} \to \mathtt{Term}(\alpha_1)) \to (\mathtt{List} \to \mathtt{Term}(\alpha_2)) \to (\mathtt{List} \to \mathtt{Term}(\mathtt{cons}(\alpha_1, \alpha_2)))$$

Wenden wir das Konstruktionsverfahren auf die gesamte Grammatik G_{List} an, erhalten wir das in Abschnitt 5.2.2 konstruierte Repository $\Gamma_{G_{List}}$.

5.2.10 Definition. Die **Inhabitationsfrage** lässt sich für eine reguläre Baumgrammatik G folgendermaßen stellen: Gegeben ein Repository Γ_G, das mit dem Verfahren aus Definition 5.2.8 erzeugt wurde, und einen Term $t \in T(\mathcal{F})$, gibt es einen kombinatorischen Ausdruck e, der $\Gamma_G \vdash e : \mathtt{Term}(\mathrm{rep}(t))$ erfüllt?

Zu beachten ist, dass wir die Inhabitationsfrage nur für Terme $t \in T(\mathcal{F})$ stellen, also für Terme ohne Nichtterminale. Dies ist konsistent mit der Akzeptanz von regulären Baumgrammatiken aus Definition 5.1.9.

5.2.4 Korrektheit und Vollständigkeit

Wir schneiden wieder den Simulationsansatz aus (1.1) auf das Berechnungsmodell zu und beweisen den daraus entstehenden Satz.

5.2.11 Satz. *Für eine reguläre Baumgrammatik $G = (S, N, \mathcal{F}, R)$ und alle $t \in T(\mathcal{F})$ gilt:*

$$\exists e(\Gamma_G \vdash e : \mathtt{Term}(\mathrm{rep}(t))) \iff S \xrightarrow{+}_G t \tag{5.5}$$

Wir beobachten zunächst, dass es Produktionen $n \to n'$ geben kann, die ein Nichtterminal n in ein anderes Nichtterminal n' überführen. Bei der Ableitung eines Terms könnte es sein, dass mehrere solcher Produktionen angewandt werden. Das folgende Lemma hilft uns, im Beweis damit umzugehen.

5.2.12 Lemma. *Seien $G = (S, N, \mathcal{F}, R)$ eine reguläre Baumgrammatik und $n, n' \in N$ zwei beliebige Nichtterminale. Gilt $n \xrightarrow{*}_G n'$ und gibt es einen Ausdruck $e' : \mathtt{n}' \to \mathtt{Term}(\tau)$, gibt es einen kombinatorischen Ausdruck $e : \mathtt{n} \to \mathtt{Term}(\tau)$.*

Beweis. Gilt $n \xrightarrow{*}_G n'$, so gibt es folgende Produktionen in R, mit $s \geq 1$, $n_1 = n$ und $n_s = n'$: $n_1 \to n_2,\ n_2 \to n_3, ...,\ n_{s-1} \to n_s$. Damit gibt es folgende Kombinatoren in Γ_G:

$$\begin{aligned}
\mathtt{n_1} &: (\mathtt{n_2} \to \mathtt{Term}(\alpha)) \to (\mathtt{n_1} \to \mathtt{Term}(\alpha)) \\
\mathtt{n_2} &: (\mathtt{n_3} \to \mathtt{Term}(\alpha)) \to (\mathtt{n_2} \to \mathtt{Term}(\alpha)) \\
&\vdots \\
\mathtt{n_{s-1}} &: (\mathtt{n_s} \to \mathtt{Term}(\alpha)) \to (\mathtt{n_{s-1}} \to \mathtt{Term}(\alpha))
\end{aligned}$$

Wir wählen $e = \mathtt{n_1}\ (\mathtt{n_2}\ (...\ (\mathtt{n_{s-1}}\ e')))$. Es ist leicht zu sehen, dass die Kombinatoren $\mathtt{n_1}, \mathtt{n_2}..., \mathtt{n_{s-1}}$ den Ausdruck $e' : \mathtt{n}' \to \mathtt{Term}(\tau)$ in einen Ausdruck vom Typ $\mathtt{n} \to \mathtt{Term}(\tau)$ umwandelt. Damit hat e den gesuchten Typ. □

Wir benötigen nun eine zu Satz 5.2.11 ähnliche Aussage, die sich bei der Ableitung nicht auf das Startsymbol beschränkt. Diese Aussage teilen wir wieder auf zwei Lemmata auf.

Die Beweise der Lemmata sind Induktionen über der Höhe der Terme $t \in T(\mathcal{F})$. Wir betrachten in jedem Induktionsschritt lediglich die Ableitung eines einzigen Terminals, da

zwei (verschachtelte) Terminale die Höhe des Terms um 2 erhöhen würden. Die Teilterme werden dann über die Induktionsvoraussetzung behandelt.

Wie wir schon gesehen haben, müssen wir außerdem beachten, dass es Produktionen gibt, die Nichtterminale in andere Nichtterminale überführen. Diese ähneln den ε-Transitionen aus Kapitel 4 und werden hier auch ähnlich behandelt.

Ähnlich zu den Beweisen von Satz 3.2.6 und 4.2.3 zeigen wir die gewünschten Aussagen wieder mit der Konstruktion eines kombinatorischen Ausdrucks (Lemma 5.2.13) bzw. mit der Konstruktion einer Ableitung (Lemma 5.2.14).

5.2.13 Lemma. *Für eine reguläre Baumgrammatik $G = (S, N, \mathcal{F}, R)$ und alle $t \in T(\mathcal{F})$ sowie $n \in N$ gilt:*

$$n \xrightarrow{+}_G t \implies \exists e(\Gamma_G \vdash e : \mathtt{n} \to \mathtt{Term}(\mathrm{rep}(t))) \tag{5.6}$$

Beweis. Wir zeigen die Aussage mit einer Induktion über der Höhe k der Terme $t \in T(\mathcal{F})$.
Induktionsanfang: Für $k = 1$ hat der Term t die Höhe 1 und muss damit eine Konstante sein. Es gilt also $t = f$ mit $f \in \mathcal{F}_0$. Sei $n \in N$ beliebig. Wir nehmen an, dass $n \xrightarrow{+}_G t$ gilt.

Vor der Ableitung des Terminals f kann ein einzelnes Nichtterminal über Produktionen der Form $n_1 \to n_2$ mit $n_1, n_2 \in N$ beliebig oft in ein anderes Nichtterminal überführt werden, weshalb wir eine Ableitung $n \xrightarrow{*}_G n'$ mit $n' \xrightarrow{+}_G f$ annehmen.

Da f eine Konstante ist und t aus n ableitbar ist, gibt es eine Produktion $n' \to f \in R$. Wir beobachten, dass $\mathrm{nt}(f) = [\,]$ ist, da f keine Nichtterminale enthält. Ebenfalls gilt $\mathrm{rep}_\alpha(f) = \mathtt{f}$. Damit gibt es einen Kombinator $\mathtt{n}'_{\mathtt{id}} : \mathtt{n}' \to \mathtt{Term}(\mathtt{f})$ in Γ_G. Mit $e' = \mathtt{n}'_{\mathtt{id}}$ und $n \xrightarrow{*}_G n'$ erhalten wir über Lemma 5.2.12 den gesuchten Ausdruck $e : \mathtt{n} \to \mathtt{Term}(\mathtt{f})$.
Induktionsschritt: Sei $n \in N$ beliebig. Wir betrachten Terme $t = f(t_1, ..., t_m)$ der Höhe $k + 1$ mit $f \in \mathcal{F}_m$. Für alle $1 \leq i \leq m$ gilt per Definition von height, dass $\mathrm{height}(t_i) \leq k$ ist. Wir beachten, dass $m \geq 1$ ist, da Konstanten nur als Blätter eines Terms vorkommen können und somit nur die Höhe 1 haben können. Wir setzen voraus, dass (5.6) für alle Terme $u \in T(\mathcal{F})$ mit $\mathrm{height}(u) \leq k$ gilt.

Wir nehmen an, dass $n \xrightarrow{+}_G t$ gilt. Wie im Induktionsanfang nehmen wir eine Ableitung $n \xrightarrow{*}_G n'$ mit $n' \xrightarrow{+}_G t$ an. Da $f \in \mathcal{F}_m$ ist und t aus n ableitbar ist, gibt es eine Produktion $n' \to f(u_1, ..., u_m) \in R$ mit $u_1, ..., u_m \in T(\mathcal{F} \cup N)$. Der Grund für diese Existenz ist, dass es im Laufe der Ableitung genau eine Produktion geben muss, die das Terminal f ableitet. Da bei regulären Baumgrammatiken nur Nichtterminale ersetzt werden können, kann das abgeleitete Terminal f in der weiteren Ableitung nicht mehr verändert werden.

Im Folgenden suchen wir einen Ausdruck $e' : \mathtt{n}' \to \mathtt{Term}(\mathrm{rep}(t))$. Sei $\phi = f(u_1, ..., u_m)$ und $\mathrm{nt}(\phi) = [n_1, ..., n_l]$ die Nichtterminale des Terms ϕ. Wir unterscheiden zwei Fälle:

Fall 1: Für $l = 0$ gibt es keine Nichtterminale, über die wir die Induktionsvoraussetzung anwenden können. Der Term e' lässt sich aber mit nur einem Kombinator konstruieren, da für Terme ohne Nichtterminale $\mathrm{rep}_\alpha(\phi) = \mathrm{rep}(t)$ ist. Da $l = 0$ ist und

die Produktion $n' \to \phi$ existiert, existiert ein Kombinator $\mathtt{n}'_{\mathtt{id}} : \mathtt{n}' \to \mathtt{Term}(\mathrm{rep}_\alpha(\phi))$ in Γ_G. Wegen $\mathrm{rep}_\alpha(\phi) = \mathrm{rep}(t)$ hat $e' = \mathtt{n}'_{\mathtt{id}}$ den gesuchten Typ $\mathtt{n}' \to \mathtt{Term}(\mathrm{rep}(t))$.
Fall 2: Sei $l \geq 1$. Da die Produktion $n' \to \phi$ existiert, existiert der folgende Kombinator in Γ_G:

$$\begin{aligned}\mathtt{n}'_{\mathtt{id}} : (\mathtt{n}_1 \to \mathtt{Term}(\alpha_1)) \to ... \to (\mathtt{n}_l \to \mathtt{Term}(\alpha_l)) \\ \to (\mathtt{n}' \to \mathtt{Term}(\mathrm{rep}_\alpha(\phi)))\end{aligned}$$

Wir betrachten nun jedes Nichtterminal n_j mit $1 \leq j \leq l$. Wir beobachten, dass n_j als Teilterm zu genau einem Term $u \in \{u_1, ..., u_m\}$ gehört, da n_j mit der Funktion nt aus einem der Terme $u_1, ..., u_m$ extrahiert wurde. Da $n \xrightarrow{+}_G t$ gilt und damit $n' \xrightarrow{+}_G t$, gilt insbesondere auch $u \xrightarrow{+}_G t_u$ für ein $t_u \in \{t_1, ..., t_m\}$. Wäre das nicht der Fall, so könnten wir den Teilterm u nicht weiter ableiten, was im Widerspruch zu $n' \xrightarrow{+}_G t$ steht. Da wir t_u aus u ableiten können und n_j ein Teilterm von u ist, gibt es einen Teilterm t_{n_j} von t_u, für den $n_j \xrightarrow{+}_G t_{n_j}$ gilt. Da $\mathrm{height}(t_u) \leq k$ ist, ist auch $\mathrm{height}(t_{n_j}) \leq k$. Mit der Induktionsvoraussetzung und dem Ergebnis, dass $n_j \xrightarrow{+}_G t_{n_j}$ gilt, erhalten wir einen Ausdruck $e_j : \mathtt{n_j} \to \mathtt{Term}(\mathrm{rep}(t_{n_j}))$.

Wir wählen $e' = \mathtt{n}'_{\mathtt{id}}\ e_1\ ...\ e_l$. Die Typen der Argumente stimmen offensichtlich mit der Definition von $\mathtt{n}'_{\mathtt{id}}$ überein. Als Belegung von α_j haben wir damit $\mathrm{rep}(t_{n_j})$. Da t_{n_j} der von n_j abgeleitete Term ist und $\mathrm{rep}_\alpha(\phi)$ im Typ $\mathrm{rep}(\phi)$ gerade die $\mathtt{n_j}$ mit den Typvariablen α_j ersetzt, erhalten wir beim Einsetzen der Belegungen aller α_j in den Typ $\mathrm{rep}_\alpha(\phi)$ den Typen $\mathrm{rep}(t)$. Damit hat e' den gesuchten Typ $\mathtt{n}' \to \mathtt{Term}(\mathrm{rep}(t))$.

Wir haben nun einen Ausdruck $e' : \mathtt{n}' \to \mathtt{Term}(\mathrm{rep}(t))$. Mit e' und $n \xrightarrow{*}_G n'$ erhalten wir über Lemma 5.2.12 den gesuchten Ausdruck $e : \mathtt{n} \to \mathtt{Term}(\mathrm{rep}(t))$. □

5.2.14 Lemma. *Für eine reguläre Baumgrammatik $G = (S, N, \mathcal{F}, R)$ und alle $t \in T(\mathcal{F})$ sowie $n \in N$ gilt:*

$$\exists e(\Gamma_G \vdash e : \mathtt{n} \to \mathtt{Term}(\mathrm{rep}(t))) \implies n \xrightarrow{+}_G t \tag{5.7}$$

Beweis. Wir zeigen die Aussage mit einer Induktion über der Höhe k der Terme $t \in T(\mathcal{F})$.
Induktionsanfang: Für $k = 1$ hat der Term t die Höhe 1 und muss somit eine Konstante sein. Term t hat also die Form $t = f$ mit $f \in \mathcal{F}_0$. Sei $n \in N$ beliebig. Wir nehmen an, dass es einen Ausdruck $e : \mathtt{n} \to \mathtt{Term}(\mathtt{f})$ gibt.

Der Ausdruck e muss die folgende Form haben, wobei die Kombinatoren $\mathtt{n_1}$ bis $\mathtt{n_{s-1}}$ aus dem Beweis zu Lemma 5.2.12 stammen und $\mathtt{n}'_{\mathtt{id}}$ den Typ $\mathtt{n}' \to \mathtt{Term}(\mathtt{f})$ hat:

$$e = \mathtt{n_1}\ (\mathtt{n_2}\ (...\ (\mathtt{n_{s-1}}\ \mathtt{n}'_{\mathtt{id}})))$$

Der Grund für diese Form ist, dass es Kombinatoren wie $\mathtt{n_1}$ und $\mathtt{n_2}$ gibt, die jeweils ein Nichtterminal in ein anderes Nichtterminal überführen. Diese können immer unabhängig

von der Höhe eines Terms angewandt werden, da sie keine Terminale erzeugen und somit die Höhe des Terms nicht ändern.[6]

Wegen der Existenz von $\mathtt{n_1}, ..., \mathtt{n_{s-1}}$ gibt es Produktionen $n \to n_2, n_2 \to n_3, ..., n_{s-1} \to n'$ in R. Mit $n \to n_2 \to n_3 \to ... \to n_{s-1} \to n'$ erhalten wir eine Ableitung $n \overset{*}{\to}_G n'$. Außerdem gibt es wegen der Existenz von $\mathtt{n'_{id}}$ eine Produktion $n' \to f$ in R. Damit gilt insgesamt $n \overset{+}{\to}_G f$.

Induktionsschritt: Wir betrachten Terme $t = f(t_1, ..., t_m)$ der Höhe $k+1$ mit $f \in \mathcal{F}_m$. Es gilt für alle $1 \leq i \leq m$, dass $\text{height}(t_i) \leq k$ ist und dass $m \geq 1$ ist. Wir setzen voraus, dass (5.7) für alle Terme $u \in T(\mathcal{F})$ mit $\text{height}(u) \leq k$ gilt.

Wir nehmen an, dass es einen Ausdruck $e : \mathtt{n} \to \mathtt{Term}(\text{rep}(t))$ gibt. Parallel zu den Überlegungen im Induktionsanfang muss der Ausdruck e die folgende Form haben, wobei e' den Typ $\mathtt{n'} \to \mathtt{Term}(\text{rep}(t))$ hat:

$$e = \mathtt{n_1}\ (\mathtt{n_2}\ (...\ (\mathtt{n_{s-1}}\ e')))$$

Für e' unterscheiden wir zwei Fälle:

Fall 1: Der Term e' hat die Form $e' = \mathtt{n'_{id}}$ mit $\mathtt{n'_{id}} : \mathtt{n'} \to \mathtt{Term}(\text{rep}(t))$ in Γ_G. Da $\mathtt{n'_{id}}$ keine Funktionen $\mathtt{n_i} \to \mathtt{Term}(\alpha_\mathtt{i})$ als Argument nimmt und in der zugrundeliegenden Produktion $n' \to \phi$ damit keine Nichtterminale in ϕ vorkommen, gilt parallel zu Fall 1 aus dem Beweis von Lemma 5.2.13, dass $\text{rep}_\alpha(\phi) = \text{rep}(t)$ ist. Damit gibt es eine Produktion $n' \to t$ in R und somit gilt auch $n' \overset{+}{\to}_G t$.

Fall 2: Der Term e' hat die Form $e' = \mathtt{n'_{id}}\ e_1\ ...\ e_l$ mit $l \geq 1$, wobei wir eine zugrundeliegende Produktion $n' \to \phi$ annehmen. Für $\mathtt{n'_{id}}$ gilt:

$$\begin{aligned}\mathtt{n'_{id}} : (\mathtt{n_1} \to \mathtt{Term}(\alpha_1)) \to ... \to (\mathtt{n}_l \to \mathtt{Term}(\alpha_l))\\ \to (\mathtt{n'} \to \mathtt{Term}(\text{rep}_\alpha(\phi)))\end{aligned}$$

Da e' existiert, existieren auch die Teilausdrücke $e_1, ..., e_l$. Der Term e' hat den Typ $\mathtt{n'} \to \mathtt{Term}(\text{rep}(t))$. Da dieser Typ mit dem Typ $\mathtt{n'} \to \mathtt{Term}(\text{rep}_\alpha(\phi))$ aus der Definition von $\mathtt{n'_{id}}$ nach Substitution der Typvariablen α_j gleich sein muss – sonst hätte e' nicht den richtigen Typen – gilt $e_j : \mathtt{n}_j \to \mathtt{Term}(\text{rep}(t_{n_j}))$ für alle $1 \leq j \leq l$ und Teilterme $t_{n_1}, ..., t_{n_l}$. Parallel zu Fall 2 aus dem Beweis von Lemma 5.2.13 muss jeder dieser Teilterme in einem der Terme $t_1, ..., t_m$ vorkommen. Damit gilt $\text{height}(t_{n_j}) \leq k$. Die Typen $\text{rep}(t_{n_j})$ entsprechen dabei jeweils der Belegung der Typvariablen α_j.

Die Induktionsvoraussetzung (5.7) lässt sich nun auf die Terme $t_{n_1}, ..., t_{n_l}$ anwenden. Aufgrund der Existenz von $e_1, ..., e_l$ erhalten wir, dass es Ableitungen $n_1 \overset{+}{\to}_G t_{n_1}, ..., n_l \overset{+}{\to}_G t_{n_l}$ gibt. Wir bezeichnen $s(\phi)$ als die eindeutige Ersetzung der Nichtterminale n_j in ϕ durch die abgeleiteten Terme t_{n_j}. Aufgrund der Konstruktion von Γ_G gilt dann $s(\phi) = t$, weitgehend analog zur Ersetzung der Typvariablen in

[6] Dies ist analog zu der möglichen Ableitung $n \overset{*}{\to}_G n'$, die wir Beweis zu Lemma 5.2.13 gesehen haben.

Fall 2 aus dem Beweis von Lemma 5.2.13. Damit gibt es eine Ableitung $\phi \xrightarrow{+}_G t$. Da es aufgrund des Kombinators $\mathtt{n}'_{\mathtt{id}}$ eine Produktion $n' \to \phi$ gibt, gilt $n' \to_G \phi \xrightarrow{+}_G t$ und somit $n' \xrightarrow{+}_G t$.

Die Betrachtung von e' zeigt, dass $n' \xrightarrow{+}_G t$ gilt. Es bleibt zu zeigen, dass es eine Ableitung $n \xrightarrow{+}_G n'$ gibt. Parallel zum Ergebnis aus dem Induktionsanfang erhalten wir über die Existenz von $\mathtt{n}_1, ..., \mathtt{n}_{\mathtt{s}-1}$ eine Ableitung $n \xrightarrow{*}_G n'$. Insgesamt gilt dann $n \xrightarrow{+}_G t$. □

Die Beweise der Lemmata sind damit abgeschlossen. Wir können sie nun verwenden, um Satz 5.2.11 zu zeigen. Der Beweis wird parallel zu den Beweisen von Satz 3.2.6 und 4.2.3 geführt, weshalb auf ausführliche Erklärungen verzichtet wird.

Beweis (Satz 5.2.11). Sei $t \in T(\mathcal{F})$ beliebig. Wir zeigen zuerst folgende Implikation:

$$S \xrightarrow{+}_G t \implies \exists e(\Gamma_G \vdash e : \mathtt{Term}(\mathrm{rep}(t)))$$

Wir können annehmen, dass $S \xrightarrow{+}_G t$ gilt. Mit Lemma 5.2.13 sichern wir die Existenz eines Ausdrucks $e' : \mathtt{S} \to \mathtt{Term}(\mathrm{rep}(t))$. Wir wählen $e = \mathtt{Start}\ e'$ und schließen den Beweis der Implikation.

Wir zeigen nun die Gegenrichtung:

$$\exists e(\Gamma_G \vdash e : \mathtt{Term}(\mathrm{rep}(t))) \implies S \xrightarrow{+}_G t$$

Es existiert ein $e : \mathtt{Term}(\mathrm{rep}(t))$. Damit muss der Ausdruck die Form $e = \mathtt{Start}\ e'$ haben, wobei $e' : \mathtt{S} \to \mathtt{Term}(\mathrm{rep}(t))$ gilt. Mit Lemma 5.2.14 gilt schließlich $S \xrightarrow{+}_G t$.

Wie im Beweis zu Satz 3.2.6 hat ω keinen Einfluss auf die Simulation. Satz 5.2.11 ist nun insgesamt bewiesen. □

5.3 Simulation von G_{List}

In diesem Abschnitt wollen wir das Beispiel aus Abschnitt 5.2.2 mit einer beispielhaften Inhabitation in $\Gamma_{G_{List}}$ abschließen. Zur Erinnerung sei hier noch einmal das Repository $\Gamma_{G_{List}}$ aufgeführt:

$$\mathtt{List}_1 : \mathtt{List} \to \mathtt{Term}(\mathtt{nil})$$

$$\mathtt{List}_2 : (\mathtt{Nat} \to \mathtt{Term}(\alpha_1)) \to (\mathtt{List} \to \mathtt{Term}(\alpha_2)) \to (\mathtt{List} \to \mathtt{Term}(\mathtt{cons}(\alpha_1, \alpha_2)))$$

$$\mathtt{Nat}_1 : \mathtt{Nat} \to \mathtt{Term}(\mathtt{0})$$

$$\mathtt{Nat}_2 : (\mathtt{Nat} \to \mathtt{Term}(\alpha_1)) \to (\mathtt{Nat} \to \mathtt{Term}(\mathtt{s}(\alpha_1)))$$

$$\mathtt{Start} : (\mathtt{List} \to \mathtt{Term}(\alpha)) \to \mathtt{Term}(\alpha)$$

Angenommen, wir wollten die Simulation mit dem folgenden Term t als Eingabe anstoßen:

$$t = cons(s(s(0)), cons(0, cons(s(0), nil)))$$

Wir suchen also einen Ausdruck e, sodass gilt:

$$\Gamma_{G_{List}} \vdash e : \texttt{Term}(\texttt{cons}(\texttt{s}(\texttt{s}(\texttt{0})), \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{s}(\texttt{0}), \texttt{nil}))))$$

Natürlich muss $e = \texttt{Start}\ e_1$ sein, damit wir einen Ausdruck vom Typ `Term` haben, mit $e_1 : \texttt{List} \to \texttt{Term}(\texttt{cons}(\tau_1, \tau_2))$, $\tau_1 = \texttt{s}(\texttt{s}(\texttt{0}))$ und $\tau_2 = \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{s}(\texttt{0}), \texttt{nil}))$. Wir wählen also $e_1 = \texttt{List}_2\ e_{11}\ e_{12}$ mit $e_{11} : \texttt{Nat} \to \texttt{Term}(\tau_1)$ und $e_{12} : \texttt{List} \to \texttt{Term}(\tau_2)$.

An diesem Punkt können wir anmerken, dass die Inhabitation bei Baumgrammatiken ähnlich abläuft wie schon bei DFAs oder ε-NFAs. Der wesentliche Unterschied ist, dass wir nun zwei Teilausdrücke e_{11} und e_{12} finden müssen. Bei den vorherigen Simulationen musste in jedem Schritt nur immer ein Ausdruck gefunden werden. Das ist natürlich auf den Unterschied zwischen (linearen) Wortstrukturen und verzweigten Baumstrukturen zurückzuführen.

Wir setzen $e_{11} = \texttt{Nat}_2\ \texttt{Nat}_2\ \texttt{Nat}_1 : \texttt{Nat} \to \texttt{Term}(\texttt{s}(\texttt{s}(\texttt{0})))$. Es ist leicht zu sehen, dass e_{11} den genannten Typen inhabitiert. Wir setzen weiterhin $e_{12} = \texttt{List}_2\ \texttt{Nat}_1\ e_{122}$ mit dem Ausdruck $e_{122} : \texttt{List} \to \texttt{Term}(\tau_{21})$ und $\tau_{21} = \texttt{cons}(\texttt{s}(\texttt{0}), \texttt{nil})$. Abschließend ist es ebenfalls leicht zu sehen, dass $e_{122} = \texttt{List}_2\ (\texttt{Nat}_2\ \texttt{Nat}_1)\ \texttt{List}_1$ sein sollte.

Insgesamt haben wir nun:

$$e = \texttt{Start}\ (\texttt{List}_2\ (\texttt{Nat}_2\ \texttt{Nat}_2\ \texttt{Nat}_1)\ (\texttt{List}_2\ \texttt{Nat}_1\ (\texttt{List}_2\ (\texttt{Nat}_2\ \texttt{Nat}_1)\ \texttt{List}_1)))$$

5.4 Testen von Anforderungen mit Baumgrammatiken

Wir wollen das Beispiel aus Abschnitt 4.3 erweitern und dabei Baumgrammatiken nutzen, weil Terme wegen ihrer Struktur gut geeignet sind, um verschiedene Eigenschaften von Fähigkeiten zu kodieren.

Wie in Abschnitt 4.3 sollen nach wie vor Variationen des fiktiven RPGs synthetisiert werden, in denen der Charakter zwei Fähigkeiten verwenden darf. Die unterschiedlichen Fähigkeiten machen die unterschiedlichen Variationen des Spiels aus. Wir wollen Anforderungen stellen, die mehrere Eigenschaften der Fähigkeiten gleichzeitig in Betracht ziehen. Wir definieren deshalb zunächst die möglichen Eigenschaften, die eine Fähigkeit haben kann.

5.4.1 Eigenschaften von Fähigkeiten

Wir überlegen uns folgende drei Eigenschaften, die wir überprüfen wollen:

1. Wir können Fähigkeiten einem bestimmten Archetyp zuordnen. Wir wählen dazu die drei klassischen Archetypen *Warrior*, *Mage* und *Rogue*.
2. Jede Fähigkeit hat beliebig viele Schadenstypen, die die Art des Schadens festlegen, den die Fähigkeit verrichtet. Es gibt folgende fünf Schadenstypen: *Fire*, *Ice*, *Physical*,

Poison und *Bleed*. Zudem verrichtet eine Fähigkeit für jeden Schadenstyp separat Schadenspunkte. Wir stufen den Schaden, den eine Fähigkeit zu einem gegebenen Schadenstyp verrichtet, auf einer Skala von 1 bis 5 ein, wobei 5 sehr hohen Schaden anzeigt.[7]

3. Eine Fähigkeit kann Kosten haben, die beim Nutzen der Fähigkeit anfallen. Wir unterscheiden Kostentypen *Health*, *Mana* und *Stamina*. Die Höhe der Kosten wird wieder auf einer Skala von 1 bis 5 bewertet, wobei Zauber mit der Kostenwertung 5 sehr teure Zauber sind. Eine Fähigkeit kann nur einen der drei Kostentypen haben.

Zur Beschreibung dieser Eigenschaften in Form eines Terms definieren wir zunächst die Baumgrammatik G_F wie folgt. Wir verwenden | Symbole, wie auch üblich bei CFGs, um Alternativen kompakt darzustellen.

$$
\begin{aligned}
\mathit{SkillInfo} &\to \mathit{skinfo}(\mathit{Archetype}, \mathit{DamageList}, \mathit{Cost}) \\
\mathit{Archetype} &\to \mathit{warrior} \mid \mathit{mage} \mid \mathit{rogue} \\
\mathit{DamageList} &\to \mathit{dlnil} \\
\mathit{DamageList} &\to \mathit{dlcons}(\mathit{DamageType}, \mathit{DamageValue}, \mathit{DamageList}) \\
\mathit{DamageType} &\to \mathit{fire} \mid \mathit{ice} \mid \mathit{physical} \mid \mathit{poison} \mid \mathit{bleed} \\
\mathit{DamageValue} &\to 1 \mid 2 \mid 3 \mid 4 \mid 5 \\
\mathit{Cost} &\to \mathit{cost}(\mathit{CostType}, \mathit{CostValue}) \\
\mathit{CostType} &\to \mathit{health} \mid \mathit{mana} \mid \mathit{stamina} \\
\mathit{CostValue} &\to 1 \mid 2 \mid 3 \mid 4 \mid 5
\end{aligned}
$$

Diese Grammatik beschreibt die oben genannten Eigenschaften vollständig als Terme. Wir können G_F als Basis nehmen, um Grammatiken zu formulieren, die gewisse Anforderungen testen. Wir können Produktionen entfernen, um die Wahl von bestimmten Fähigkeiten zu verhindern. Beispielsweise könnten wir die Produktion $\mathit{Archetype} \to \mathit{warrior}$ entfernen, um Fähigkeiten, die zum Archetyp *warrior* gehören, nicht in die Spielvariationen mit einzubeziehen. Weiterhin können wir die Produktionen von *DamageList* anpassen, um z.B. zu verlangen, dass eine Fähigkeit mindestens einen Schadenstyp hat. Diese und andere Anforderungen wollen wir im nächsten Abschnitt beispielhaft in einer neuen Grammatik kodieren.

[7] Dieser Wert muss nicht unbedingt dem tatsächlichen Schadenswert entsprechen, den eine Fähigkeit verrichtet. Es geht hier nur um eine Bewertung des Schadens, sodass im Verlauf der Synthese beispielsweise Fähigkeiten nicht genommen werden, die sehr viel Schaden machen. Dieses Prinzip lässt sich auch auf andere Anwendungen übertragen: Grobe Informationen sind leichter zu modellieren als detaillierte, reichen aber möglicherweise für den konkreten Anwendungsfall aus.

5.4.2 Anforderungen und Repository

Wie in Abschnitt 4.3 wollen wir einige Anforderungen überprüfen, die für einen Fähigkeiten-Kombinator erfüllt sein müssen, damit wir diesen in der Synthese verwenden können. Dazu definieren wir zunächst die Anforderungen in Form einer Baumgrammatik und konstruieren anschließend das zur Baumgrammatik gehörende Repository nach Definition 5.2.8.

Wir stellen folgende Anforderungen:

1. Es sollen lediglich die Archetypen *Mage* und *Rogue* erlaubt werden.
2. Die Fähigkeit soll mindestens einen Schadenstyp haben.
3. Es sollen nur die Schadenstypen *Fire*, *Ice* und *Physical* erlaubt sein.
4. Die Kostentypen sollen sich auf *Mana* und *Stamina* beschränken.
5. Die Kostenwertung soll höchstens 4 und mindestens 2 sein.

Diese Anforderungen setzen wir wie folgt um:

1. Wir entfernen die Produktion $Archetype \to warrior$.
2. Wir verändern die Produktionen des Nichtterminals *DamageList* so, dass mindestens ein *dlcons*-Symbol erzeugt wird. Dazu führen wir ein neues Nichtterminal *FullDamageList* ein, das keine Produktion zum Erzeugen einer leeren Liste hat und anstelle von *DamageList* in der Produktion von *SkillInfo* steht.
3. Wir entfernen die Produktionen $DamageType \to poison$ und $DamageType \to bleed$.
4. Wir entfernen die Produktion $CostType \to health$.
5. Wir entfernen die Produktionen $CostValue \to 1$ und $CostValue \to 5$.

Es ergibt sich nun folgende reguläre Baumgrammatik G_A:

$$
\begin{aligned}
SkillInfo &\to skinfo(Archetype, FullDamageList, Cost)\\
Archetype &\to mage \mid rogue\\
FullDamageList &\to dlcons(DamageType, DamageValue, DamageList)\\
DamageList &\to dlnil\\
DamageList &\to dlcons(DamageType, DamageValue, DamageList)\\
DamageType &\to fire \mid ice \mid physical\\
DamageValue &\to 1 \mid 2 \mid 3 \mid 4 \mid 5\\
Cost &\to cost(CostType, CostValue)\\
CostType &\to mana \mid stamina\\
CostValue &\to 2 \mid 3 \mid 4
\end{aligned}
$$

Mit Definition 5.2.8 können wir G_A in ein Repository Γ_{G_A} überführen. Wir erinnern uns, dass *SkillInfo* per Konvention das Startsymbol ist. Wir haben folgende Kombinatoren in Γ_{G_A}, wobei einige Kombinatoren wegen der Übersichtlichkeit nicht aufgelistet sind:

$$
\begin{aligned}
\texttt{Start} &: (\texttt{SkillInfo} \to \texttt{Term}(\alpha)) \to \texttt{Term}(\alpha) \\
\texttt{SkillInfo}_1 &: (\texttt{Archetype} \to \texttt{Term}(\alpha_1)) \to (\texttt{FullDamageList} \to \texttt{Term}(\alpha_2)) \\
&\quad \to (\texttt{Cost} \to \texttt{Term}(\alpha_3)) \to (\texttt{SkillInfo} \to \texttt{Term}(\texttt{skinfo}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{Archetype}_1 &: \texttt{Archetype} \to \texttt{Term}(\texttt{mage}) \\
\texttt{Archetype}_2 &: \texttt{Archetype} \to \texttt{Term}(\texttt{rogue}) \\
\texttt{FullDamageList}_1 &: (\texttt{DamageType} \to \texttt{Term}(\alpha_1)) \to (\texttt{DamageValue} \to \texttt{Term}(\alpha_2)) \\
&\quad \to (\texttt{DamageList} \to \texttt{Term}(\alpha_3)) \\
&\quad \to (\texttt{FullDamageList} \to \texttt{Term}(\texttt{dlcons}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{DamageList}_1 &: \texttt{DamageList} \to \texttt{Term}(\texttt{dlnil}) \\
\texttt{DamageList}_2 &: (\texttt{DamageType} \to \texttt{Term}(\alpha_1)) \to (\texttt{DamageValue} \to \texttt{Term}(\alpha_2)) \\
&\quad \to (\texttt{DamageList} \to \texttt{Term}(\alpha_3)) \\
&\quad \to (\texttt{DamageList} \to \texttt{Term}(\texttt{dlcons}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{DamageType}_1 &: \texttt{DamageType} \to \texttt{Term}(\texttt{fire}) \\
&\quad \dots \\
\texttt{DamageValue}_1 &: \texttt{DamageValue} \to \texttt{Term}(1) \\
&\quad \dots \\
\texttt{Cost}_1 &: (\texttt{CostType} \to \texttt{Term}(\alpha_1)) \to (\texttt{CostValue} \to \texttt{Term}(\alpha_2)) \\
&\quad \to (\texttt{Cost} \to \texttt{Term}(\texttt{cost}(\alpha_1, \alpha_2))) \\
\texttt{CostType}_1 &: \texttt{CostType} \to \texttt{Term}(\texttt{mana}) \\
\texttt{CostType}_2 &: \texttt{CostType} \to \texttt{Term}(\texttt{stamina}) \\
\texttt{CostValue}_1 &: \texttt{CostValue} \to \texttt{Term}(2) \\
&\quad \dots
\end{aligned}
$$

Die Kombinatoren für die Fähigkeiten sind aufsteigend nummeriert, geordnet nach dem Vorkommen der zugehörigen Produktion in der Grammatik G_A. Bevor wir im Folgenden eine Simulation der Baumgrammatik betrachten können, müssen wir zunächst das Fähigkeiten-Repository für das Testen von Anforderungen mit der Grammatik G_A anpassen. Anschließend müssen wir das angepasste Repository mit Γ_{G_A} verschmelzen. Diese Schritte sind parallel zu dem Beispiel aus dem vorherigen Kapitel zu sehen.

5.4.3 Verschmelzung von Γ_F und Γ_{G_A}

Wir erweitern zunächst das Repository Γ_F aus Abschnitt 4.3 so, dass die Simulation der Baumgrammatik angestoßen wird. Wir haben folgende Kombinatoren in dem entstehenden Repository $\Gamma_{F'}$:

$$\begin{aligned}
\texttt{Fireball} &: \texttt{Term}(\tau_1) \to \texttt{Skill} \\
\texttt{WallOfIce} &: \texttt{Term}(\tau_2) \to \texttt{Skill} \\
\texttt{IceLance} &: \texttt{Term}(\tau_3) \to \texttt{Skill} \\
\texttt{Poison} &: \texttt{Term}(\tau_4) \to \texttt{Skill} \\
\texttt{DeepCut} &: \texttt{Term}(\tau_5) \to \texttt{Skill} \\
\texttt{SkillSet} &: \texttt{Skill} \to \texttt{Skill} \to \texttt{SkillSet}
\end{aligned}$$

Wobei für τ_1 bis τ_5 folgendes gilt:

$$\begin{aligned}
\tau_1 &= \texttt{skinfo}(\texttt{mage}, \texttt{dlcons}(\texttt{fire}, 2, \texttt{dlnil}), \texttt{cost}(\texttt{mana}, 2)) \\
\tau_2 &= \texttt{skinfo}(\texttt{mage}, \texttt{dlnil}, \texttt{cost}(\texttt{mana}, 4)) \\
\tau_3 &= \texttt{skinfo}(\texttt{mage}, \texttt{dlcons}(\texttt{ice}, 3, \texttt{dlcons}(\texttt{physical}, 2, \texttt{dlnil})), \texttt{cost}(\texttt{mana}, 3)) \\
\tau_4 &= \texttt{skinfo}(\texttt{rogue}, \texttt{dlcons}(\texttt{poison}, 3, \texttt{dlnil}), \texttt{cost}(\texttt{stamina}, 2)) \\
\tau_5 &= \texttt{skinfo}(\texttt{warrior}, \tau_{51}, \texttt{cost}(\texttt{stamina}, 3)) \\
\tau_{51} &= \texttt{dlcons}(\texttt{physical}, 2, \texttt{dlcons}(\texttt{bleed}, 4, \texttt{dlnil}))
\end{aligned}$$

Die `Term`-Typen der einzelnen Fähigkeiten entsprechen natürlich jeweils dem Eigenschaften-Term, der von der Grammatik G_F erzeugt werden könnte. Die Eigenschaften sind auf die individuellen Fähigkeiten zugeschnitten. Beispielsweise gehört die Fähigkeit *Ice Lance* zu dem Archetyp *Mage*. Sie macht auf zwei Arten Schaden: Einerseits *Physical* und andererseits *Ice*, wobei jedem Schadenstyp unterschiedliche Schadenswerte zugeordnet sind. *Ice Lance* kostet durchschnittlich viel *Mana* (3 auf der Skala).

Da wir die Anforderungen überprüfen wollen, die in der Baumgrammatik G_A eingebaut sind, müssen wir nur noch $\Gamma_{F'}$ und Γ_{G_A} verschmelzen:

$$\Gamma_V = \Gamma_{F'} \cup \Gamma_{G_A}$$

Dadurch, dass Terme, die die Anforderungen nicht erfüllen, nicht über die Grammatik G_A abgeleitet werden können, findet sich nach Satz 5.2.11 auch kein kombinatorischer Ausdruck, der die Typrepräsentation des Terms inhabitiert. Damit kann ein Fähigkeiten-Kombinator, dessen zugehörige Fähigkeit nicht die Anforderungen erfüllt, nicht angewandt werden. Somit wird die Fähigkeit nicht in eine Variation des Spiels aufgenommen. Dieses Verhalten ist natürlich parallel zu dem Beispiel aus Abschnitt 4.3 zu betrachten.

5.4.4 Inhabitation in Γ_V

Wir stellen die Frage, welche kombinatorischen Ausdrücke s den Typ `Skill` inhabitieren, also für welche Ausdrücke $\Gamma_V \vdash s : \texttt{Skill}$ gilt. Da es nur Kombinatoren des Typs $\texttt{Term}(\tau) \to \texttt{Skill}$ gibt, müssen wir insbesondere einen kombinatorischen Ausdruck finden, der den jeweiligen `Term`-Typ inhabitiert. Wie oben angesprochen können wir mit Satz 5.2.11 diese Frage über die Ableitung in der Baumgrammatik G_A beantworten. Für die folgenden Kombinatoren lässt sich ein solcher Ausdruck finden: `Fireball` und `IceLance`. `WallOfIce` kann nicht verwendet werden, weil die Fähigkeit keinen Schadenstyp hat und unsere Anforderung mindestens einen verlangt. `Poison` kann nicht verwendet werden, weil der Schadenstyp *Poison* nicht erlaubt ist. `DeepCut` kann nicht verwendet werden, weil der *Warrior* Archetyp nicht erlaubt ist.

Wir betrachten zunächst exemplarisch den kombinatorischen Ausdruck $e_1 : \texttt{Term}(\tau_3)$, der bei der Anwendung von `IceLance` synthetisiert wird:

$$
\begin{aligned}
e_1 = \texttt{Start SkillInfo}_1\ \texttt{Archetype}_1 \\
&(\texttt{FullDamageList}_1\ \texttt{DamageType}_2\ \texttt{DamageValue}_3 \\
&\quad (\texttt{DamageList}_2\ \texttt{DamageType}_3\ \texttt{DamageValue}_2\ \texttt{DamageList}_1)) \\
&(\texttt{Cost}_1\ \texttt{CostType}_1\ \texttt{CostValue}_2)
\end{aligned}
$$

Der Ausdruck e_1 kann parallel zu folgender Ableitungskette gesehen werden, die bei einer Ableitung des zu τ_3 gehörigen Terms entstehen würde:

$$
\begin{aligned}
\mathit{SkillInfo} &\to \mathit{skinfo}(\mathit{Archetype}, \mathit{FullDamageList}, \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{mage}, \mathit{FullDamageList}, \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{mage}, \mathit{dlcons}(\mathit{DamageType}, \mathit{DamageValue}, \mathit{DamageList}), \mathit{Cost}) \\
&\xrightarrow{+} \mathit{skinfo}(\mathit{mage}, \mathit{dlcons}(\mathit{ice}, 3, \mathit{DamageList}), \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{mage}, \mathit{dlcons}(\mathit{ice}, 3, \mathit{dlcons}(\mathit{DamageType}, \mathit{DamageValue}, \mathit{DamageList})), \mathit{Cost}) \\
&\xrightarrow{+} \mathit{skinfo}(\mathit{mage}, \mathit{dlcons}(\mathit{ice}, 3, \mathit{dlcons}(\mathit{physical}, 2, \mathit{dlnil})), \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{mage}, \mathit{dlcons}(\mathit{ice}, 3, \mathit{dlcons}(\mathit{physical}, 2, \mathit{dlnil})), \mathit{cost}(\mathit{CostType}, \mathit{CostValue})) \\
&\xrightarrow{+} \mathit{skinfo}(\mathit{mage}, \mathit{dlcons}(\mathit{ice}, 3, \mathit{dlcons}(\mathit{physical}, 2, \mathit{dlnil})), \mathit{cost}(\mathit{mana}, 3))
\end{aligned}
$$

Wir sehen, dass die in der Ableitungskette benutzten Nichtterminale den Kombinatoren entsprechen, die im Ausdruck e_1 genutzt werden. Da der Ausdruck wie bei DFAs und ε-NFAs die Ausführung der Simulation widerspiegelt, und diese Ausführung gerade das Finden einer geeigneten Ableitung ist, ist dieser Zusammenhang nicht überraschend.

Den gesuchten Ausdruck $s_1 : \texttt{Skill}$ kann man schließlich als $s_1 = \texttt{IceLance}\ e_1$ definieren.

Wir wollen nun zuletzt betrachten, warum der Kombinator `Poison` nicht sinnvoll verwendet werden kann. Wir betrachten dazu den Versuch, den zu τ_4 gehörigen Term abzuleiten:

$$\begin{aligned}
\mathit{SkillInfo} &\to \mathit{skinfo}(\mathit{Archetype}, \mathit{FullDamageList}, \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{rogue}, \mathit{FullDamageList}, \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{rogue}, \mathit{dlcons}(\mathit{DamageType}, \mathit{DamageValue}, \mathit{DamageList}), \mathit{Cost}) \\
&\xrightarrow{+} \mathit{skinfo}(\mathit{rogue}, \mathit{dlcons}(\mathit{DamageType}, 3, \mathit{dlnil}), \mathit{Cost}) \\
&\to \mathit{skinfo}(\mathit{rogue}, \mathit{dlcons}(\mathit{DamageType}, 3, \mathit{dlnil}), \mathit{cost}(\mathit{CostType}, \mathit{CostValue}) \\
&\xrightarrow{+} \mathit{skinfo}(\mathit{rogue}, \mathit{dlcons}(\mathit{DamageType}, 3, \mathit{dlnil}), \mathit{cost}(\mathit{stamina}, 2)
\end{aligned}$$

An diesem Punkt kommen wir nicht weiter, da wir *DamageType* nicht zu *poison* ableiten können. Damit wird der zu τ_4 gehörige Term nicht akzeptiert und es gibt auch keinen Ausdruck vom Typ `Term`(τ_4).

5.4.5 Fazit

In diesem Beispiel haben wir gesehen, wie wir mit Baumgrammatiken *komplexe* Eigenschaften modellieren können, die beim Testen von Anforderungen in Betracht gezogen werden sollen. Dazu haben wir eine Baumgrammatik erstellt, die diese Eigenschaften modelliert, und das zugehörige Repository konstruiert. Wir haben das konstruierte Repository ins zuvor definierte Fähigkeiten-Repository eingebettet und abschließend eine beispielhafte Simulation geführt. Für eine Diskussion der Vor- und Nachteile dieses Verfahrens sei auf Abschnitt 4.3.5 verwiesen.

Dieses Beispiel lässt sich als Erweiterung zu dem Beispiel aus Abschnitt 4.3 sehen. Es wäre sehr unschön, diese komplexen Eigenschaften mit einem ε-NFA überprüfen zu müssen (man muss sich nur fragen, wie man die obige Baumgrammatik geeignet als NFA modelliert). Das Modell Baumgrammatik eignet sich dagegen sehr gut, die nötigen Informationen strukturiert darzustellen.

Kapitel 6

Praktische Simulation

In diesem Kapitel wollen wir uns abschließend der *praktischen Simulation* widmen, was heißt, dass wir die Beispiele aus den vorangehenden Kapiteln implementieren und simulieren wollen. Wir benutzen dazu cls-scala, eine Umsetzung von CLS in der Programmiersprache Scala. Wir widmen uns zunächst kurz den Grundlagen von cls-scala und danach den Beispielen.

6.1 Grundlagen von cls-scala

In cls-scala[1] lassen sich Kombinatoren als annotierte Objekte in einem Trait definieren, der ein Repository darstellen soll. Eine Instanz dieses Traits[2] kann dann als Argument der Klasse `ReflectedRepository` verwendet werden, die die Schnittstelle zum Inhabitationsalgorithmus bildet. `ReflectedRepository` erwartet weiterhin ein `Kinding`, welches für die im Repository vorkommenden Typvariablen aufzählt, für welche Typen diese Typvariablen jeweils einstehen können. Diese Typen dürfen dabei insbesondere selbst keine Typvariablen enthalten.

Widmen wir uns zunächst den Kombinatordefinitionen. In Listing 6.1 ist eine Definition vom Kombinator $\texttt{Nat}_2 : (\texttt{Int} \cap (\texttt{Nat} \to \texttt{Term}(\alpha_1))) \to (\texttt{Int} \cap (\texttt{Nat} \to \texttt{Term}(\texttt{s}(\alpha_1))))$ aus Abschnitt 5.2.2 zu sehen. Der Typ $\texttt{Int} \to \texttt{Int}$ ist dabei der native Typ des Kombinators, der über den Typ von `apply` festgelegt wird. $(\texttt{Nat} \to \texttt{Term}(\alpha_1)) \to (\texttt{Nat} \to \texttt{Term}(\texttt{s}(\alpha_1)))$ ist der semantische Typ, der über die Konstante `semanticType` festgelegt wird. Während die nativen Typen Scala-Typen sind, werden die semantischen Typen als Werte vom Typ `Type` konstruiert. Werte der Form `'A` entsprechen dabei Typkonstruktoren `A` einer beliebigen Stelligkeit. Der Operator `=>:` konstruiert einen Funktionstypen. Weiterhin gibt es den Operator `:&:` für Intersektionstypen.

[1] Zu cls-scala gibt es zu dieser Zeit keine offizielle Quelle, allerdings wurde cls-scala schon in [8] verwendet. Der Leser sei also auch auf diese Arbeit verwiesen.

[2] Genau genommen kann man Traits nicht instanziieren. In diesem Fall wird ein neues Objekt einer anonymen Klasse erstellt, die den Trait erweitert (siehe [6], 6.10)

Der Kombinator wird als `object` definiert, d.h., dass ein einzelnes Objekt einer neuen Klasse erzeugt wird (siehe [6], 5.4). Die Annotation `@combinator` weist das Objekt als Kombinator aus.

Listing 6.1: Kombinatordefinition

```
1 trait ListRepository {
2   @combinator object Nat2 {
3     def apply(n: Int): Int = n + 1
4     val semanticType = ('Nat =>: 'Term(alpha1)) =>:
5                        ('Nat =>: 'Term('s(alpha1)))
6   }
7   // ...
8 }
```

Listing 6.2 zeigt die Nutzung des `ListRepository`-Traits. Dabei wird die Inhabitation eines semantischen Typs `termType` über eine Instanz von `ReflectedRepository` angestoßen. Der Wert von `termType` entspricht einem Term $cons(0, nil)$.[3] Die Funktion `enumerate` ist keine Funktion von cls-scala, sondern eine Funktion, die in Listing B.1 zusätzlich definiert wird, um alle einzigartigen Teiltypen eines Typs aufzuzählen. Für `'cons('zero, 'nil)` wären das beispielsweise die Typen `'zero`, `'nil` und `'cons('zero, 'nil)`. Da `parts` dann genau die Typen beinhaltet, die eine Typvariable im Laufe der Simulation annehmen kann, ergibt sich das `Kinding` für jede Typvariable, in diesem Fall `alpha1` und `alpha2`. Bei der Inhabitation mit `gamma.inhabit` wird schließlich der native Zieltyp als Typargument übergeben – hier `List[Int]`, da wir Listen von Zahlen inhabitieren wollen. Der semantische Typ, hier `'Term(termType)`, wird als Argument übergeben.

Listing 6.2: Nutzung des `ListRepository`-Traits und Inhabitation

```
1 val repository = new ListRepository { }
2 val termType   = 'cons('zero, 'nil)
3 val parts      = termType.enumerate
4 val kinding    = Kinding(repository.alpha1).addOptions(parts) merge
5                  Kinding(repository.alpha2).addOptions(parts)
6 val gamma      = ReflectedRepository[ListRepository](inst = repository,
7                                                      kinding = kinding)
8 val results    = gamma.inhabit[List[Int]]('Term(termType))
```

Das Ergebnis der Inhabitation mit cls-scala ist eine Liste von Inhabitanten, die beispielsweise auf der Kommandozeile ausgegeben werden können, aber auch direkt ausgeführt werden können. Dabei werden die `apply`-Funktionen der jeweiligen Kombinatoren aufgerufen.

Mit diesen Grundlagen können wir uns nun den Beispielen widmen. Wir werden uns dort auf die Definition der jeweiligen Repository-Traits und die Ergebnisse der Inhabitation

[3] Es wurde zero anstelle von 0 gewählt, da `'0` kein gültiger Scala-Code ist.

beschränken. Für eine genauere Betrachtung des Quellcodes sei auf den beiliegenden Datenträger verwiesen.

6.2 Praktische Simulation von DFAs und ε-NFAs

In diesem Abschnitt betrachten wir die Beispiele 3.3 und 4.3. Zunächst gehen wir aber kurz auf einen Trait ein, der das Definieren von Kombinatoren für DFAs und ε-NFAs um einiges leichter macht.

Wir betrachten Listing B.3. Die dort definierten Klassen sind Schablonen für Kombinatoren mit der gleichen Struktur, beispielsweise die Klasse `Transition` für Transitionskombinatoren `D`[`q`, `a`, `p`], die von drei Werten abhängig sind: Dem Quellzustand `q`, dem gelesenen Zeichen `a` und dem Zielzustand `p`. Außerdem übergeben wir eine Funktion `f : A => B`, die die native Semantik der Transition darstellt. Der Transitionskombinator `D`[`grnd`, `g`, `grnd`] aus Beispiel 3.3 lässt sich dann folgendermaßen implementieren, wobei wir auf `Action.run` später eingehen:

```
1 @combinator object GrndG extends Transition('grnd, 'g(_), 'grnd)(Action.run)
```

6.2.1 Simulation des Runner-DFAs

Wir beschäftigen uns nun zunächst mit Beispiel 3.3. Listing B.4 zeigt die Implementation von Repository $\Gamma_{\mathcal{A}_{KI}}$ in cls-scala. Wie oben erwähnt werden die Kombinator-Schablonen aus Listing B.3 verwendet. Die in Listing B.5 definierte Semantik baut weitestgehend auf den Anmerkungen aus Abschnitt 3.3.3 auf. Wir definieren ein Subjekt, hier `State` genannt, als eine `Option[PlayerPosition]`. `PlayerPosition` stellt dabei eine zweidimensionale Position der Spielfigur auf dem Spielfeld dar. Das `Option` ist nötig, weil wir eine vernünftige Semantik für Transitionen wie `D`[`fall`, `b`, `dead`] benötigen, bei denen die Spielfigur stirbt oder schon tot ist, und somit keine gültige Position mehr hat.[4] Aktionen (`Action`) sind Funktionen, die einen `State` in einen anderen `State` überführen. Bei der Definition eines jeweiligen Transitionskombinators wird ihm die geeignete Aktion gegeben, wie zum Beispiel `Action.run` für den Transitionskombinator `GrndG`.

Wir schauen uns nun das Ergebnis einer Inhabitation an, wobei wir uns einerseits für den erzeugten Ausdruck und das Ergebnis von dessen Ausführung interessieren, andererseits aber auch für die Laufzeit, insbesondere im Hinblick auf die in Kapitel 2 erwähnte exponentielle Komplexität des Inhabitationsalgorithmus. Als Eingabe wählen wir `'Word('b('b('g('g('epsilon)))))`, welches dem Wort bbgg entspricht, das auch schon in Beispiel 3.3 betrachtet wurde. Wir erhalten folgenden Ausdruck:

[4] Am Rande sei bemerkt, dass ein ε-NFA hier den Vorteil bieten würde, dass man den Tod der Spielfigur nicht explizit modellieren muss, weil man in einem Zustand nicht für jedes Zeichen eine Transition definieren muss.

```
1 Tree(RunRunner,List(
2   Tree(GrndB,List(
3     Tree(Air1B,List(
4       Tree(Air2G,List(
5         Tree(FallG,List(
6           Tree(FinGrnd,List()))))))))))
```

Dies entspricht dem folgenden Scala-Ausdruck, wobei die Funktionsapplikation von jedem Kombinator zum Aufruf der jeweiligen `apply`-Funktion führt (siehe [6], 6.6):

```
1 RunRunner(GrndB(Air1B(Air2G(FallG(FinGrnd())))))
```

Der Scala-Ausdruck entspricht dem Ausdruck e, den wir in Abschnitt 3.3.2 synthetisiert haben. Der mit cls-scala synthetisierte Ausdruck ist also korrekt.

Das Ergebnis der Ausführung des obigen Ausdrucks ist `Some(PlayerPosition(4,0))`. Die x-Koordinate hat den Wert 4, die Spielfigur hat sich also 4 Felder nach rechts bewegt und steht damit am rechten Rand des Spielfeldes bbgg. Die y-Koordinate ist 0, da die Spielfigur am Ende auf dem Boden steht. Sowohl der Ausdruck als auch das Ergebnis der Ausführung entsprechen unseren Erwartungen.

Um die Laufzeit zu testen,[5] haben wir Wörter $w \in (\text{bbgg})^+$ als Eingabe genommen. Die Laufzeit der Inhabitation kann man Tabelle 6.1 entnehmen. In der Laufzeit enthalten ist die Initialisierung von cls-scala, die einige Sekunden dauert, aber als konstanter Wert schnell an Bedeutung verliert. Wir sehen, dass die Laufzeit sehr stark ansteigt. Obwohl ein Wort der Länge 160 nur doppelt so groß ist wie ein Wort der Länge 80, ist die Laufzeit um den Faktor 12 gestiegen. Die von einem DFA erwartete lineare Laufzeit ergibt sich damit nicht. Vielmehr weist dieses Ergebnis auf eine exponentielle Laufzeit hin.

Tabelle 6.1: Laufzeit der Inhabitation in `RunnerRepository`.

Länge	Zeit
20	10s
40	65s
60	248s
80	633s
160	7963s

6.2.2 Inhabitation im Fähigkeiten-Repository

Wir implementieren nun das Repository aus Beispiel 4.3 in cls-scala. In Listing B.6 sind alle Kombinatoren in einem Trait `NfaGameRepository` implementiert. Dabei unterscheiden wir

[5] Der Laptop, auf dem die Tests ausgeführt wurden, hat einen einzelnen i7-Prozessor aus dem Jahre 2013. Das Betriebssystem ist OS X. Einen Mangel an Hauptspeicher gab es nicht.

einerseits die Kombinatoren, deren Anwendung zu `Skill` Objekten führt, und andererseits die Kombinatoren des modellierten ε-NFA. Bei den Letzteren wurden der Übersichtlichkeit halber einige Kombinatoren ausgelassen, die zu den Repositories Γ_2 und Γ_3 gehören. Die `Skill`-Klasse wird in Listing B.7 definiert.

Wir mussten Typkonstanten wie `F` und `I` umbenennen, damit sie sich namentlich nicht mit den Typkonstruktoren von Zeichen wie F und I überlappen. Ist `F` sowohl eine Typkonstante als auch ein Typkonstruktor mit der Stelligkeit 1, wird die Invariante verletzt, dass Typkonstruktoren immer die gleiche Stelligkeit haben müssen.[6]

Die Eingabe zur Inhabitation besteht in diesem Fall nicht mehr aus einem Worttyp, sondern es wird nach einem nativen Typen `List[Skill]` mit dem semantischen Typen `'SkillSet` gefragt. Da wir den ε-NFA innerhalb einer Inhabitationsanfrage mehrmals simulieren (zu jedem Fähigkeiten-Kombinator gibt es ein Wort), müssen wir dem `Kinding` von jeder Variable die Teiltypen aller Worttypen geben. Die einzelnen `Enumeration[Type]` Instanzen werden dazu einfach mit der `union`-Funktion vereinigt. Der Quellcode dazu findet sich in Listing B.8.

Das Ergebnis der Inhabitation besteht aus mehreren Scala-Ausdrücken, die mit der in `NfaGameRepository` festgelegten nativen Semantik zu folgenden einzigartigen Werten ausgewertet werden:

```
1 List(Skill(Fireball), Skill(Fireball))
2 List(Skill(Fireball), Skill(Ice Lance))
3 List(Skill(Fireball), Skill(Wall of Ice))
4 List(Skill(Ice Lance), Skill(Fireball))
5 List(Skill(Ice Lance), Skill(Ice Lance))
6 List(Skill(Ice Lance), Skill(Wall of Ice))
7 List(Skill(Wall of Ice), Skill(Fireball))
8 List(Skill(Wall of Ice), Skill(Ice Lance))
9 List(Skill(Wall of Ice), Skill(Wall of Ice))
```

Wie man leicht sehen kann, sind dies alle Kombinationen, die man bei einer Auswahl von zwei Fähigkeiten aus der limitierten Menge von Fähigkeiten erhalten kann, die Fire oder Ice im Namen haben. Insbesondere konnte zu `Poison` und `DeepCut` wie erwartet kein `SkillSet` inhabitiert werden.

Letztlich sei die Laufzeit zu erwähnen, die leider nicht gut ist. Für die Inhabitation aller Ausdrücke hat cls-scala **2369 Sekunden** benötigt.

6.3 Praktische Simulation von Baumgrammatiken

In diesem Abschnitt beschäftigen wir uns mit der praktischen Simulation der Baumgrammatiken aus den Beispielen 5.2.2 und 5.4. Wir haben hier auch wieder einen Trait, der

[6] Quelle: Persönliche Kommunikation mit Jan Bessai.

Schablonen für Kombinatoren bereitstellt. Dieser Trait wird in Listing B.10 gezeigt. Die enthaltenen Klassen sollten selbsterklärend sein und lassen sich auch leicht anhand des semantischen Typs den jeweiligen Produktionsarten zuordnen. Im Gegensatz zum vorigen Abschnitt können hier nicht alle möglichen Kombinatoren einer Schablone zugeordnet werden, da die Kombinatoren weitaus komplexer sind.

6.3.1 Simulation von G_{List}

Wir simulieren nun die Baumgrammatik G_{List} aus Abschnitt 5.2.2 mit verschiedenen Eingaben. Die Implementation des Repositories $\Gamma_{G_{List}}$ im Trait `ListRepository` findet sich in Listing B.11. $\texttt{List}_1$ und $\texttt{Nat}_1$ werden jeweils mithilfe der `TerminalProduction` Klasse definiert. Zu den Typkonstanten, die zu den Nichtterminalen gehören, wurde `Nt` als Präfix hinzugefügt. Damit sollen semantische Typen wie `'NtList` von unabhängigen nativen Typen wie `'List` abgegrenzt werden. Während der Implementation gab es schon Probleme mit solchen doppeldeutigen Namen.

Als native Typen haben wir für die natürlichen Zahlen, die über die `Nat` Produktionen erzeugt werden, den Typ `Int` gewählt. `Int` unterstützt zwar auch negative Zahlen, allerdings gibt es in Scala keinen vergleichbar standardmäßigen Typ. Die Wahl von `List` als nativen Typ für Listen sollte selbsterklärend sein.

In Tabelle 6.2 sehen wir verschiedene Eingaben (wobei wir das `'Term(...)` aus Übersichtsgründen ausgelassen haben), das berechnete Ergebnis, und die jeweilige Laufzeit des Inhabitationsalgorithmus. Wir sehen, dass bei den Eingaben (3) und (4) nicht bzw. nicht immer die richtigen Ergebnisse berechnet wurden. Bei der Eingabe (4) scheint dies davon abzuhängen, in welcher Reihenfolge die Typen zum `Kinding` hinzugefügt werden. Abgesehen davon steigt die Laufzeit wieder sehr schnell an, was aber im Zuge der bisherigen Ergebnisse zu erwarten war.

Tabelle 6.2: Inhabitationen in `ListRepository`.

Index	Eingabe	Ergebnis	Zeit
1	`'nil`	`List()`	4s
2	`'cons('zero, 'nil)`	`List(0)`	6s
3	`'cons('s('zero), 'nil)`	Kein Ergebnis	7s
4	`'cons('s('s('zero)), 'nil)`	Manchmal `List(2)`	9s
5	`'cons('zero, 'cons('s('zero), 'nil))`	`List(0, 1)`	1376s
6	`'cons('s('zero), 'cons('s('zero), 'nil))`	`List(1, 1)`	428s

6.3.2 Implementation des Fähigkeiten-Repository

Wir wollen nur kurz auf die Implementation des Fähigkeiten-Repository aus Abschnitt 5.4 eingehen. Das zugehörige `RtgGameRepository` findet sich in Listing B.12. Wie bei der Inhabitation im ε-NFA Fähigkeiten-Repository müssen wir wieder Typen zu allen im Repository vorkommenden Termen in das `Kinding` aufnehmen. Die Lösung dazu ist aber genau analog zu der Lösung für das ε-NFA Repository. Abgesehen von dieser Besonderheit und einigen Umbenennungen folgt die Implementation genau dem vorgegebenen Repository.

Wir konnten die Inhabitation leider nicht testen, weil die Ausführung selbst nach einem Tag Laufzeit nicht zum Ende gekommen ist. Somit gibt es auch keine Ausgabe, die hier präsentiert werden kann.

6.4 Synthese von Docker-Konfigurationen

In diesem Abschnitt betrachten wir die Bachelorarbeit *Synthese von Docker-Konfigurationen unter Zuhilfenahme eines Inhabitationsalgorithmus* von Daniel Scholtyssek [8]. Aufgrund des Umfangs der Arbeit werden im Folgenden die Grundlagen von [8] vorausgesetzt.

6.4.1 Einführung

In der Arbeit von Daniel Scholtyssek werden Docker-Konfigurationen auf Basis einer Auswahl an Einstellungen generiert. Das Programm heißt DoSy. Das Kernstück der im Titel genannten Synthese bildet dabei ein Automat ([8], Kapitel 4.3.4), der nichtdeterministisch ein **CompleteWebApp**-Objekt erzeugt und konfiguriert. Dieses wird dann benutzt, um die Docker Compose Konfigurationsdateien zu erzeugen. Besonders zu bemerken ist hier, dass CLS nicht verwendet wird, um die Konfigurationsdateien an sich zu erzeugen, sondern um den Ausdruck zu synthetisieren, der später den Generator der Konfigurationsdateien erzeugt und konfiguriert.

Der genannte Automat kann als nichtdeterministischer Automat mit gelabelten Kanten verstanden werden und wird mithilfe von CLS bei der Synthese simuliert. Auf Basis von Einstellungen, die über semantische Typen im Eingabetyp festgelegt werden können, werden bestimmte Kombinatoren aktiviert oder deaktiviert. Da jeder Kombinator einer gelabelten Kante des Automaten entspricht, können wir auch sagen, dass Kanten je nach der Einstellung aktiviert oder deaktiviert sind. Die synthetisierten Ausdrücke ergeben sich schließlich folgendermaßen: Für jeden Lauf durch den Automaten, der ausschließlich aus aktiven Kanten besteht, wird ein Ausdruck erzeugt. Dabei sei noch zu erwähnen, dass die Transitionen von **WithTemplates** zum Endzustand in der Lösung nicht als Kombinator umgesetzt sind.

Einige Eingaben, die zur Konfiguration der Docker Compose Dateien notwendig sind, werden von der Kommandozeile abgefragt ([8], Kapitel 4.5.4). Es gibt dazu Kombinato-

ren wie z.B. `tomcatPort` mit dem Typen `Int` ∩ *tomcatPort*. Dieser Kombinator soll den Tomcat-Port bereitstellen und setzt dafür eine Kommandozeilenabfrage in Gang. Diese Besonderheit findet sich in der *Implementation* von `tomcatPort` und anderen solchen Kombinatoren.

Wir wollen den Automaten im Folgenden neu umsetzen und zwar in Form einer Baumgrammatik. Der Automat löst ein klassisches Problem der **Codegenerierung**, was auch ein Anwendungsfall für die Simulation von Berechnungsmodellen ist. Wir setzen den Automaten als Baumgrammatik um, da uns dieses Berechnungsmodell eine besondere Strukturiertheit in der Eingabe bietet, die das Modell ε-NFA nicht besitzt.

6.4.2 Modellierung der Baumgrammatik

Zum Überblick stellen wir zunächst folgende Anforderungen an unsere Lösung:

- Wir müssen die Ausdrücke, die das **CompleteWebApp**-Objekt konfigurieren, korrekt synthetisieren.
- Wir müssen die Variationen, die durch das aktiveren bzw. deaktivieren der Kanten des Automaten entstehen, in der Baumgrammatik umsetzen.
- Wir wollen die Werte, die im Laufe der Synthese über die Kommandozeile abgefragt werden, stattdessen als Typen kodieren.

Die Baumgrammatik G_{dosy}, die diese Anforderungen erfüllt, ist in Anhang C.1 zu finden. Die Farben blau und rot sind angelehnt an [8], Kapitel 4.4.1 gewählt: Nichtterminale, die *Methoden* beschreiben, sind blau gefärbt, solche, die *Inputs* beschreiben, rot. Außerdem gibt es violette Nichtterminale, die ein Input matchen, welches für die aktuelle Produktion nicht von Bedeutung ist. Im Folgenden wollen wir erläutern, wie die obigen Anforderungen von G_{dosy} umgesetzt werden.

Für die Codegenerierung mit Baumgrammatiken lässt sich zunächst einmal festhalten, dass der synthetisierte Ausdruck der Anwendung von Nichtterminalen entspricht, die im Laufe der simulierten Ableitung verwendet wurden. Haben wir also einen Ausdruck `A B (C D)`, können wir diesen beispielsweise mit dem aus folgender Grammatik konstruierten Repository und der Eingabe `x(y,z)` synthetisieren:

$$A \to x(B, C) \qquad B \to y$$
$$C \to D \qquad D \to z$$

Dieses Prinzip wird in G_{dosy} beispielsweise im folgenden Auszug verwendet:

$$\begin{aligned} \textit{AssignDependsOn} \to\ & \textit{DoDefaultDatabase} \\ \mid\ & \textit{DoEmptyDatabase} \\ \mid\ & \textit{LoadSQLFile} \end{aligned}$$

$$\textit{DoDefaultDatabase} \rightarrow db(true, \textit{Bool}, \textit{MaybeString}, \textit{WithUsers})$$
$$\textit{DoEmptyDatabase} \rightarrow db(\textit{Bool}, true, \textit{MaybeString}, \textit{WithUsers})$$
$$\textit{LoadSQLFile} \rightarrow db(\textit{Bool}, \textit{Bool}, \textit{String}, \textit{WithUsers})$$

Die Grammatik ist so gewählt, dass ein Teilausdruck `AssignDependsOn(...)` generiert wird, beispielsweise `AssignDependsOn(DoDefaultDatabase(...))` bei der Simulation mit dem Term $db(true, false, none, ...)$ als Eingabe.

Vergleicht man die Baumgrammatik mit dem Automaten, so erkennt man, dass G_{dosy} den Automaten „rückwärts“ umsetzt, da wir bei einem Ausdruck `A (B (C (... empty)))` den Kombinator `A` *zuletzt* auf den vorher konstruierten Wert anwenden. Demnach muss das **CompleteWebApp**-Objekt von „innen nach außen“ konfiguriert werden, woraus sich die Reihenfolge in der Grammatik ergibt. Im Übrigen sei erwähnt, dass die Namensgebung der *DoDefault...* Nichtterminale ungünstig ist, aber konsistent zu der Typumgebung aus [8], Listing D.18 gewählt wurde.

Leiten wir das Nichtterminal *AssignDependsOn* weiter ab, können wir aus drei möglichen Konfigurationsmethoden wählen. Jede dieser Methoden wird wieder als Nichtterminal dargestellt, welches abgeleitet werden kann, wenn die Eingabe aus der Produktion auf der rechten Seite abgeleitet werden kann. Die *With...* Nichtterminale existieren nur, um die Baumgrammatik abzukürzen, und haben keine weitere Bedeutung.

Die Variationen, die durch das Aktivieren von Kanten im Automaten entstehen, werden in G_{dosy} über die Eingabe ermöglicht. Jede Kante, gewählt aus n Kanten von einem Zustand zum Nächsten, entspricht dabei einem Teilterm eines Symbols $f \in \mathcal{F}_{n+1}$. Beispielsweise können wir zum Symbol $db \in \mathcal{F}_4$ den Term $db(t_1, t_2, t_3, c)$ betrachten, mit den folgenden Bedeutungen:

- t_1 ist die Konfiguration zum Nutzen der *Standard Datenbank mit Einträgen* (siehe [8], Kapitel 4.3.2, Datenbank konfigurieren). Da diese Einstellung lediglich einem Schalter entspricht, kann der Wert nur *true* oder *false* sein.
- t_2 konfiguriert parallel zu t_1 das Nutzen der *Leeren Datenbank*.
- t_3 ist die Konfiguration für eine Datenbank, die von der Festplatte importiert wird. Im Gegensatz zu t_1 und t_2 werden hier zusätzliche Inputs benötigt, und zwar der Dateipfad in Form eines *String*-Wertes.
- c ist der Term für weitere Einstellungen. Im Kontext des Automaten wird hier in einen neuen Zustand gewechselt (*With...* Nichtterminale) bzw. die einzige Kante aus dem nächsten Zustand heraus genommen (Blaue Nichtterminale).

Wir ermöglichen mehrere Variationen dadurch, dass mehrere Einstellungen gleichzeitig belegt werden können. Beispielsweise wären sowohl eine leere Datenbank als auch die Standard Datenbank möglich, wenn folgender Term gewählt wird: $db(true, true, none, ...)$. Das Lesen der Datenbank von der Festplatte wird dabei allerdings ausgeschlossen, da *none* kein

String-Wert ist. Die violetten Nichtterminale sorgen dabei dafür, dass Terme von Einstellungen, die für die aktuelle Methode nicht von Bedeutung sind, abgeleitet werden können – egal, ob diese nun gesetzt sind oder nicht.

Wir ermöglichen die Darstellung von Zeichenketten und Zahlen in der Eingabe über die *String* und *Number* Nichtterminale. Da wir solche Eingaben direkt im Term kodieren, ist es nicht mehr nötig, diese über die Kommandozeile abzufragen.

Eine gültige Eingabe könnte folgendermaßen aussehen. Sie entspricht der Anfrage aus [8], Kapitel 4.4, Listing 4.13, wobei die nötigen Inputs in [8] nicht vorkommen und erdacht sind. Außerdem wurde die Konfiguration von *number of replicas* in G_{dosy} nicht umgesetzt, da sie in [8] eigentlich als Erweiterung betrachtet wurde, inkonsistenterweise aber im Repository vorkommt und im Automaten nicht. Wir haben:

$$\begin{aligned}
t &= tp(true, none, t_1)\\
t_1 &= nn(true, none, t_2)\\
t_2 &= wp(false, n_{500}, t_3)\\
n_{500} &= d5(d0(d0(end)))\\
t_3 &= db(true, false, none, t_4)\\
t_4 &= user(true, none, t_5)\\
t_5 &= cont(false, none, t_6, empty)\\
t_6 &= clusterConfig(d2(end), d2(end), d2(end))
\end{aligned}$$

Wir wollen abschließend eine Ableitung als Beispiel führen. Aufgrund der Größe der Baumgrammatik leiten wir den Term t_5 ab dem Nichtterminal *WithContainers* ab:

$$\begin{aligned}
WithContainers &\to CreateContainersWithCluster\\
&\to cont(Bool, MaybeNumber, ClusterConfig, EmptyWebApp)\\
&\overset{+}{\to} cont(false, none, clusterConfig(Number, Number, Number), EmptyWebApp)\\
&\overset{+}{\to} cont(false, none, clusterConfig(d2(end), d2(end), d2(end)), empty)
\end{aligned}$$

Wir sehen, wie die violetten Nichtterminale abgeleitet werden, obwohl *false* bzw. *none* als Teilterme dort stehen. Nur für den dritten Teilterm wird bei der Ableitung von dem Nichtterminal *CreateContainersWithCluster* ein *ClusterConfig* Wert erwartet.

Wir wollen die in diesem Abschnitt definierte Baumgrammatik im Folgenden in ein Repository überführen.

6.4.3 Konstruktion von $\Gamma_{G_{dosy}}$

Das nach dem Konstruktionsverfahren aus Definition 5.2.8 konstruierte Repository $\Gamma_{G_{dosy}}$ ist in Anhang C.2 zu finden. Wir wollen als Beispiel einen Ausdruck zu dem oben definierten

Term t_5 finden. Wir wählen dazu den Eingabetypen `WithContainers` $\rightarrow$ `Term`(rep(`t5`)). Die Inhabitation liefert folgendes Ergebnis:

$$n_2 : \texttt{Number} \rightarrow \texttt{Term}(\texttt{d2}(\texttt{end}))$$
$$n_2 = \texttt{NumberD2}\ \texttt{NumberEnd}$$
$$e_6 : \texttt{ClusterConfig} \rightarrow \texttt{Term}(\texttt{clusterConfig}(\texttt{d2}(\texttt{end}), \texttt{d2}(\texttt{end}), \texttt{d2}(\texttt{end})))$$
$$e_6 = \texttt{ClusterConfig}\ n_2\ n_2\ n_2$$
$$e_5 : \texttt{WithContainers} \rightarrow \texttt{Term}(\texttt{cont}(\texttt{false}, \texttt{none}, \text{rep}(t_6), \texttt{empty}))$$
$$e_5 = \texttt{WithContainers}_3\ (\texttt{CreateContainersWithCluster}\ \texttt{Bool}_2\ \texttt{MaybeNumber}_2\ e_6\ \texttt{EmptyWebApp})$$

Das konstruierte Repository wollen wir nun zum Abschluss in cls-scala implementieren und an einem Beispiel simulieren.

6.4.4 Umsetzung in cls-scala

Die Implementation des Repositories ist auf die folgenden vier Traits aufgeteilt worden:

- `NumberRepository`, Listing C.3
- `StringRepository`, Listing C.4
- `ConfigRepository`, Listing C.5
- `DosyRepository`, Listing C.6

`NumberRepository` und `StringRepository` stellen jeweils Kombinatoren zur Verfügung, mit deren Hilfe Integer- bzw. String-Eingaben dargestellt werden können. Der native Typ `List[Int]` repräsentiert eine Liste von Ziffern (0 - 9). Mit der Hilfsfunktion `toInt` lässt sich ein Wert von `List[Int]` dann in einen Integer umwandeln. `String`-Werte werden direkt als `String` konstruiert.

`ConfigRepository` definiert verschiedene Kombinatoren für komplexere Konfigurationen wie *ClusterConfig*. Dort werden außerdem die Kombinatoren definiert, die zu den violetten Nichtterminalen gehören.

In `DosyRepository` werden die Kombinatoren definiert, über deren Anwendung das **CompleteWebApp**-Objekt konfiguriert wird. Die nativen Typen (abgesehen von den Eingabe-Typen wie `List[Int]`, die zusätzlich hinzukommen) und die Semantik der blauen Kombinatoren (siehe $\Gamma_{G_{dosy}}$) sind von den ursprünglichen DoSy-Kombinatoren aus [8] übernommen worden. Dafür wurde der ursprüngliche DoSy-Quellcode übernommen, insbesondere der Java-Teil. Das Repository wurde neu geschrieben und der Code zum Anstoßen der Inhabitation wurde angepasst. Die Erweiterung *number of replicas* wurde in unserem Fall nicht mit einbezogen, befindet sich aber noch im Java-Teil des übernommenen Quellcodes. Wir rufen in der `apply`-Methode von `AssignTemplates1` eine Methode `doDefault` auf,

die den *number of replicas* Wert auf den Default setzt. Damit umgehen wir eine mögliche Anpassung unserer Baumgrammatik bzw. unseres Repositories.

Wie der Leser möglicherweise schon ahnt, konnten wir eine Inhabitation des Typen, der zum oben definierten Ausdruck t gehört, aufgrund der langen Laufzeit nicht abschließen. Um etwas Perspektive für die Dauer der Inhabitation zu geben, können wir uns den Teilterm t_6 anschauen. Die Inhabitation des dazugehörigen Typen hat (nach Hinzufügen des nötigen `Start`-Kombinators) letztendlich zum erwarteten Ergebnis `ClusterConfig(2,2,2)` geführt. Diese Inhabitation hat aber **2494 Sekunden** gebraucht. Aufgrund dieser Laufzeiten war ein weiterer Test des Dosy-Repositories nicht möglich.

6.5 Fazit

In diesem Kapitel haben wir gesehen, wie wir die in den Beispielen definierten Repositories mithilfe von cls-scala praktisch umsetzen können. Abgesehen von kleineren Besonderheiten wie einer Umbenennung einiger Typkonstruktoren oder Kombinatoren konnten wir die Repositories direkt umsetzen. Dabei mussten wir native Typen und Semantiken spezifizieren, die allerdings weitestgehend offensichtlich waren. Zuletzt haben wir den DoSy-Automaten von Daniel Scholtyssek mithilfe einer Baumgrammatik neu modelliert und das konstruierte Repository implementiert.

Während die Ausgabe der Inhabitationen weitestgehend zufriedenstellend war, können wir das über die Laufzeit nicht behaupten. Schon beim einfachsten Beispiel, der Simulation des Runner-DFA, ist die Laufzeit sehr schnell mit der Länge der Eingabe gestiegen. Wir kommen im nächsten Kapitel, der Evaluation, auf diesen und andere Aspekte zu sprechen.

Kapitel 7

Evaluation

In diesem Kapitel wollen wir theoretische und praktische Aspekte der Ergebnisse dieser Arbeit diskutieren. Wir beginnen dabei mit dem vielleicht offensichtlichsten Problem, der Laufzeit der mit cls-scala implementierten Beispiele.

7.1 Praktische Relevanz

In Abschnitt 2.4 hatten wir gesehen, dass das Inhabitationsproblem für die k-beschränkte kombinatorische Logik mit Intersektionstypen $(k+2)$-EXPTIME-vollständig ist. Dieses Ergebnis hatte schon früh im Verlaufe der Bearbeitung dieser Arbeit zur Identifizierung des allgemeineren *Risikos der mangelnden praktischen Relevanz* geführt. Ein Aspekt der praktischen Relevanz ist die Laufzeit. Eine exponentielle Laufzeit schränkt die praktische Relevanz insofern ein, dass man die hier präsentierten Konstruktionsverfahren nur auf kleine, nahezu triviale Eingaben anwenden kann. In Kapitel 6, der praktischen Simulation, haben wir dann immer wieder eine schnell ansteigende Laufzeit feststellen müssen. Sowohl die Länge der Eingaben hat dabei eine Rolle gespielt, wie z.B. Tabelle 6.1 gezeigt hat, als auch die Art des Berechnungsmodells. Während die Simulation des Runner-DFA für eine Eingabe der Länge 80 ca. 633 Sekunden gedauert hat, hat die Inhabitation eines Termtyps `clusterConfig(d2(end), d2(end), d2(end))` im Dosy-Repository schon ca. 2494 Sekunden gebraucht. Dazu kommt, dass einige Simulationen nicht nur lange gebraucht haben, sondern in zumutbarer Zeit nicht terminiert sind.

Durch diese Probleme ist eine Anwendbarkeit in der Praxis nicht gegeben. Eine Verbesserung dieses Zustands kann aus zweierlei Richtungen erfolgen. Einerseits könnte man die Konstruktionsverfahren so anpassen, dass das erzeugte Repository eine schnellere Inhabitation zulässt. Andererseits könnte man CLS und cls-scala weiter allgemein optimieren, oder sogar speziell für die hier vorgestellten Konstruktionsverfahren. Nützlich wäre außerdem eine allgemeine Analyse der Laufzeit der Inhabitation in den konstruierten Repositories, unabhängig von den Modellinstanzen.

Ein anderer Aspekt der praktischen Relevanz ist die Frage, ob die betrachteten Berechnungsmodelle geeignet sind, um Probleme aus der Praxis zu modellieren. In der Einleitung hatten wir schon die zwei Anwendungsfälle **Codegenerierung** und **Testen von Anforderungen** identifiziert, die wir im Laufe der Arbeit an Beispielen gezeigt haben. Insbesondere das DoSy-Beispiel aus Kapitel 6 hat gezeigt, dass das Berechnungsmodell Baumgrammatik durchaus zur Modellierung von einem nicht-trivialen Sachverhalt geeignet ist. Die beiden fiktiven Beispiele zum Testen von Anforderungen, in denen verschiedene Variationen eines Spiels synthetisiert werden sollten, sind ebenfalls nicht-trivial und auch praktisch von Interesse, da sie ein zu [3] ähnliches Problem lösen. Insgesamt bewerten wir die Eignung, Praxisprobleme zu modellieren, als positiv.

7.2 Automatische Übersetzung

Im Laufe der Arbeit hat der Leser möglicherweise gemerkt, dass die Übersetzung vom Modell zum Repository und schließlich zur Implementation in Scala weitgehend mechanisch ist. Aufgrund der Größe einiger Modelle ist diese Übersetzung trotz ihrer Einfachheit sehr zeitaufwendig. Außerdem ist die Implementation um einiges unübersichtlicher als das ursprüngliche Modell, was man beispielsweise bei einem Vergleich der DoSy-Baumgrammatik aus Anhang C.1 mit dem DoSy-Quellcode sieht. Dadurch besteht die Gefahr, dass eine manuelle Übersetzung zu Fehlern führt.

Um die Anwendung in der Praxis komfortabler und weniger fehleranfällig zu machen, sollte die manuelle Implementation durch eine maschinelle Übersetzung der Modelle in Repositories oder direkt Scala-Quellcode ersetzt werden. Dazu müsste ein Compiler entwickelt werden, der die Modelle einliest und aus ihnen Scala-Quellcode generiert.

7.3 Vereinfachte Modelle als Alternativansatz

Wir haben bei der Simulation von Baumgrammatiken aus Kapitel 5 bewusst auf die Normalformen von Baumgrammatiken verzichtet. Die Benutzung einer Normalform hätte den Vorteil gehabt, dass der Beweis zu Satz 5.2.11 wahrscheinlich etwas einfacher gewesen wäre. Eine Normalform hätte aber zwei Nachteile gehabt:

1. Jede Baumgrammatik hätte vor der Anwendung des Konstruktionsverfahrens zunächst in die Normalform gebracht werden müssen. Das erschwert weiter die Übertragung einer Baumgrammatik in die letztendliche Implementation.
2. Während eine Normalform beweistechnisch schöne Eigenschaften besitzt, ist sie für einen Menschen schlechter lesbar und weicht vom ursprünglichen Modell ab. Die Qualität des synthetisierten Codes (im Anwendungsfall der Codegenerierung) hätte dadurch abgenommen und die Implementation der nativen Semantik wäre dadurch erschwert gewesen.

Der letzte Punkt ist im Anwendungsfall des Testens von Anforderungen nicht relevant, weshalb es in diesem Kontext eventuell von Interesse wäre, vereinfachte Modelle als Alternativansatz zu untersuchen, womöglich auch hinsichtlich der Performance der letztendlichen Implementation.

Für die Diskussion, ob man ε-NFAs oder nur NFAs unterstützt, gelten ähnliche Bedingungen (der Beweis zu Satz 4.2.3 wäre ebenfalls einfacher gewesen, hätte man einfache NFAs betrachtet). Wir haben uns aufgrund der zusätzlichen Features von ε-NFAs für diese entschieden. In anderen Kontexten wäre womöglich ein NFA ausreichend und damit vielleicht sinnvoller.

7.4 Nichtdeterminismus und Lösungsauswahl

Wegen dem Nichtdeterminismus von Modellen wie ε-NFAs kann es mehrere Berechnungen geben, über die eine Eingabe akzeptiert wird. Dadurch gibt es auch mehrere unterschiedliche Ausdrücke, die von CLS synthetisiert werden können. Folgende Fragen, die in dieser Arbeit nicht beantwortet werden, stellen sich dabei:

1. Wie kann man Lösungen qualitativ bewerten? Welche allgemeinen Kriterien gibt es für die Lösungsauswahl bei der Simulation von ε-NFAs?
2. Welche Auswirkungen hat der Nichtdeterminismus auf die Laufzeit der Inhabitation?
3. Ist der Nichtdeterminismus sinnvoll für den Anwendungsfall der Codegenerierung?
4. Gibt es ein Repository Γ, das mit dem ε-NFA Konstruktionsverfahren konstruiert wurde, und eine Eingabe w, sodass die Inhabitation von $\text{list}(w)$ unendlich viele Lösungen findet? Kommen solche Repositories in der Praxis vor?

7.5 Zusammenfassung der offenen Probleme und Erweiterungen

Die folgenden offenen Probleme und potentiellen Erweiterungen wurden in diesem Kapitel angesprochen:

1. Die Problematik der schlechten Laufzeiten. Suche nach Optimierungen für die hier vorgestellten Konstruktionsverfahren oder cls-scala.
2. Ein Compiler, der Modellinstanzen in Repositories oder Scala-Quellcode übersetzt.
3. Die Betrachtung von vereinfachten Berechnungsmodellen wie Baumgrammatiken in Normalform oder NFAs ohne ε-Transitionen.
4. Verschiedene Fragen zum Nichtdeterminismus und die dadurch erschwerte Lösungsauswahl.

Kapitel 8

Fazit

In dieser Arbeit haben wir die Simulation von Berechnungsmodellen innerhalb des Softwaresyntheseframeworks CLS anhand eines in der Einleitung vorgestellten *allgemeinen Simulationsansatzes* bearbeitet. Wir sind dazu zunächst auf die Grundlagen der Softwaresynthese mit CLS eingegangen, wo wir die kombinatorische Logik mit Intersektionstypen und das Inhabitationsproblem vorgestellt haben. Wir haben dann die Berechnungsmodelle DFA, ε-NFA und Baumgrammatik jeweils im Rahmen der in der Einleitung dargelegten Ziele betrachtet. Dabei wurden zunächst die Grundlagen des jeweiligen Berechnungsmodells dargelegt. Es wurde jeweils ein Konstruktionsverfahren definiert, welches ein Repository aus einer Modellinstanz erzeugt. Die Simulation einer Modellinstanz mit der Eingabe w entspricht nach dem allgemeinen Simulationsansatz aus (1.1) der Suche nach einem Inhabitanten eines Typs $\text{rep}(w)$ unter Verwendung der Kombinatoren des konstruierten Repositories. Die Funktion rep liftet die Eingabe der Simulation auf die Typebene. Sie wurde für jedes Berechnungsmodell separat definiert. Um die Korrektheit und Vollständigkeit des Konstruktionsverfahrens zu zeigen, wurde jeweils der allgemeine Simulationsansatz auf das konkrete Berechnungsmodell zugeschnitten und bewiesen. Wir haben dabei für jedes Berechnungsmodell einen solchen Beweis erfolgreich geführt.

Schon in der Einleitung wurden die Anwendungsfälle *Codegenerierung* und das *Testen von Anforderungen* identifiziert, die wir im Laufe der Arbeit mit Beispielen untermauert haben. Zur Codegenerierung haben wir das Runner-Beispiel in Kapitel 3 und das DoSy-Beispiel in Kapitel 6 vorgestellt. Die Beispiele zu den Berechnungsmodellen ε-NFA und Baumgrammatik aus Kapitel 4 und 5 haben das Testen von Anforderungen gezeigt.

Mit diesen Ergebnissen sind die Hauptziele, die in der Einleitung aufgelistet worden sind, erfüllt. Das genannte Nebenziel der praktischen Simulation wird durch Kapitel 6 erfüllt. Dort haben wir zunächst cls-scala vorgestellt, bevor wir alle Beispiele aus den vorhergehenden Kapiteln in Scala implementiert haben. Wir haben außerdem ein weiteres Beispiel gegeben, in dem wir einen Automaten aus einer Bachelorarbeit, die sich mit der Synthese von Docker-Konfigurationen beschäftigt hat, in einer Baumgrammatik umgesetzt

und das daraus konstruierte Repository in Scala implementiert haben. Diese Beispiele wurden auch praktisch simuliert, was allerdings aufgrund der langen Laufzeiten insbesondere bei der Simulation der Baumgrammatiken nur teilweise zu guten Ergebnissen geführt hat.

Diese und andere Ergebnisse wurden schließlich in der Evaluation in Kapitel 7 diskutiert. Die erste Diskussion hat sich mit der Laufzeit und dem damit verbundenen, übergeordneten Konzept der praktischen Relevanz beschäftigt. Wir sind zu dem Schluss gekommen, dass die praktische Verwendbarkeit der Ergebnisse aus modelltechnischer Sicht zwar positiv zu bewerten ist, aufgrund der hohen Laufzeit aber momentan keinen Sinn macht. Es wurden Erweiterungen angesprochen, wie die automatische Übersetzung von Modellinstanzen in Scala-Quellcode und die Verwendung von vereinfachten Berechnungsmodellen. Die offene Frage der Lösungsauswahl bei der Simulation von nichtdeterministischen Modellen wurde ebenfalls aufgeworfen.

Die vorliegende Arbeit stellt eine Verbindung von Theorie und Praxis im Themenkomplex der Softwaresynthese dar. Während die Ergebnisse der theoretischen Betrachtung unsere Ansprüche und Erwartungen erfüllt haben, haben wir bei der Anwendung im Praktischen gesehen, dass die Übertragung der Theorie auf die Praxis unverhoffte Ergebnisse mit sich gebracht hat. Wir sind gespannt, welche Fortschritte es in Zukunft auf diesem Gebiet geben wird und wie die verfügbare Theorie in der Praxis umgesetzt und verwendet werden kann.

Anhang A

Notationskonventionen

In dieser Arbeit gelten folgende Konventionen bezüglich der Notation von Ausdrücken und Typen im Rahmen der kombinatorischen Logik:

- Freie Variablen (z.B. Teilausdrücke e) und Hilfsfunktionen (z.B. rep aus Definition 5.2.1) werden in der normalen mathematischen Schrift gesetzt: e, w, list, rep, usw.
- Konkrete Wörter werden unterstrichen, z.b. abba.
- Namen von Typen und Kombinatoren werden dagegen in einer Typewriter-Schrift gesetzt. In dem kombinatorischen Ausdruck `f a` sind sowohl `f` als auch `a` Kombinatornamen. In dem Typausdruck `Word`(list(w)) ist `Word` als Typkonstruktor zu lesen, list als Hilfsfunktion und w als freie Variable.
- Gibt es eine Variable a, zu der ein Typkonstruktor gehört, kann dieser Typkonstruktor als `a` bezeichnet werden. Davon wird beispielsweise in Definition 3.2.1 Gebrauch gemacht, wo zu einem beliebigen Zeichen $a_i \in \Sigma$ ein Typkonstruktor $\mathtt{a_i}$ verwendet wird.
- Freie Variablen, die anstelle von konkreten Typen stehen, werden immer als τ bezeichnet, beispielsweise τ, τ_1, τ_2, usw.
- Typvariablen werden immer als α, β, γ bezeichnet. Diese Zeichen kommen nie als freie Variablen vor. Gültige Typvariablen sind z.B. α_1, α_2 und β_1.
- Das Zeichen ε steht immer für das leere Wort, während ϵ der Typkonstante entspricht, die das leere Wort repräsentiert.

Anhang B

Scala-Quellcode

Listing B.1: Funktion `enumerate`

```
1  implicit class TermTypeEnumerator(termType: Constructor) {
2    def enumerate: Enumeration[Type] = TermTypeEnumerator.enumerate(termType)
3  }
4
5  object TermTypeEnumerator {
6    private def uniqueParts(tt: Type): Set[Type] = tt match {
7      case cst: Constructor if cst.arguments.nonEmpty =>
8        val children = cst.arguments.map(uniqueParts).reduce(_ union _)
9        children + cst
10     case t => Set(t)
11   }
12
13   private def enumerate(tt: Type): Enumeration[Type] = {
14     val set = uniqueParts(tt)
15     set.map(t => Enumeration.singleton(t).asInstanceOf[Enumeration[Type]])
16       .reduce(_ union _)
17   }
18 }
```

Listing B.2: Trait `AutomatonVariables`

```
1  trait AutomatonVariables {
2    val alpha = Variable("alpha")
3  }
```

Listing B.3: Trait `AutomatonCombinators`

```
1  trait AutomatonCombinators extends AutomatonVariables {
2    class Fin[A](q: Constructor)(v: A) {
3      def apply(): A = v
4      val semanticType = 'St(q) =>: 'Word('epsilon)
5    }
```

```
6
7   class Transition[A, B](q: Constructor, a: Type => Constructor,
8       p: Constructor)(f: A => B) {
9     def apply(x: A): B = f(x)
10    val semanticType = ('St(p) =>: 'Word(alpha)) =>:
11                       ('St(q) =>: 'Word(a(alpha)))
12  }
13
14  class EpsilonTransition[A, B](q: Constructor,
15      p: Constructor)(f: A => B) {
16    def apply(x: A): B = f(x)
17    val semanticType = ('St(p) =>: 'Word(alpha)) =>:
18                       ('St(q) =>: 'Word(alpha))
19  }
20
21  class Run[A](q0: Constructor) {
22    def apply(x: A): A = x
23    val semanticType = ('St(q0) =>: 'Word(alpha)) =>: 'Word(alpha)
24  }
25 }
```

Listing B.4: Trait `RunnerRepository`

```
1  trait RunnerRepository extends AutomatonCombinators {
2    @combinator object RunRunner extends Run[State]('grnd)
3    @combinator object FinGrnd extends Fin('grnd)(PlayerPosition.startState)
4
5    import Action._
6    @combinator object GrndG extends Transition('grnd, 'g(_), 'grnd)(run)
7    @combinator object GrndB extends Transition('grnd, 'b(_), 'air1)(jump)
8    @combinator object Air1G extends Transition('air1, 'g(_), 'grnd)(fall)
9    @combinator object Air1B extends Transition('air1, 'b(_), 'air2)(jump)
10   @combinator object Air2G extends Transition('air2, 'g(_), 'fall)(fall)
11   @combinator object Air2B extends Transition('air2, 'b(_), 'dead)(die)
12   @combinator object FallG extends Transition('fall, 'g(_), 'grnd)(fall)
13   @combinator object FallB extends Transition('fall, 'b(_), 'dead)(die)
14   @combinator object DeadG extends Transition('dead, 'g(_), 'dead)(stayDead)
15   @combinator object DeadB extends Transition('dead, 'b(_), 'dead)(stayDead)
16 }
```

Listing B.5: `PlayerPosition`, `State` und `Action`

```
1 case class PlayerPosition(x: Int, y: Int)
2
3 object PlayerPosition {
4   val start = PlayerPosition(0, 0)
5   val startState = Option(start)
6 }
```

```
7
8  type State = Option[PlayerPosition]
9  type Action = State => State
10
11 object Action {
12   val jump: Action = _.map(pos => PlayerPosition(pos.x + 1, pos.y + 1))
13   val fall: Action = _.map(pos => PlayerPosition(pos.x + 1, pos.y - 1))
14   val run: Action = _.map(pos => PlayerPosition(pos.x + 1, pos.y))
15   val die: Action = _ => None
16   val stayDead: Action = p => p
17 }
```

Listing B.6: Trait `NfaGameRepository`

```
1  trait NfaGameRepository extends AutomatonCombinators {
2    private val idu: Unit => Unit = identity
3
4    // Gamma F', Section 4.3.3
5    @combinator object Fireball {
6      def apply(word: Unit): Skill = Skill("Fireball")
7      val semanticType = 'Word('F('i('r('e(
8        'b('a('l('l('epsilon))))))))) =>: 'Skill
9    }
10
11   @combinator object WallOfIce {
12     def apply(word: Unit): Skill = Skill("Wall of Ice")
13     val semanticType = 'Word('W('a('l('l('space('o('f(
14       'space('I('c('e('epsilon)))))))))))) =>: 'Skill
15   }
16
17   @combinator object IceLance {
18     def apply(word: Unit): Skill = Skill("Ice Lance")
19     val semanticType = 'Word('I('c('e('space(
20       'L('a('n('c('e('epsilon))))))))))) =>: 'Skill
21   }
22
23   @combinator object Poison {
24     def apply(word: Unit): Skill = Skill("Poison")
25     val semanticType = 'Word('P('o('i('s('o('n('epsilon))))))) =>: 'Skill
26   }
27
28   @combinator object DeepCut {
29     def apply(word: Unit): Skill = Skill("Deep Cut")
30     val semanticType = 'Word('D('e('e('p('space(
31       'C('u('t('epsilon)))))))))) =>: 'Skill
32   }
33
34   @combinator object SkillSet {
```

```
35      def apply(s1: Skill, s2: Skill): List[Skill] = List(s1, s2)
36      val semanticType = 'Skill =>: 'Skill =>: 'SkillSet
37    }
38
39    // Gamma 1, Section 4.3.2
40    @combinator object RunNfa extends Run[Unit]('start)
41
42    @combinator object EpsS_W1 extends EpsilonTransition('start, 'w1)(idu)
43    @combinator object W1_F extends Transition('w1, 'F(_), 'wF)(idu)
44    @combinator object WF_i extends Transition('wF, 'i(_), 'wFi)(idu)
45    @combinator object WFi_r extends Transition('wFi, 'r(_), 'wFir)(idu)
46    @combinator object WFir_e extends Transition('wFir, 'e(_), 'wFire)(idu)
47    @combinator object FinFire extends Fin('wFire)(())
48
49    @combinator object EpsS_W2 extends EpsilonTransition('start, 'w2)(idu)
50    @combinator object W2_I extends Transition('w2, 'I(_), 'wI)(idu)
51    @combinator object WI_c extends Transition('wI, 'c(_), 'wIc)(idu)
52    @combinator object WIc_e extends Transition('wIc, 'e(_), 'wIce)(idu)
53    @combinator object FinIce extends Fin('wIce)(())
54
55    // Gamma 2
56    @combinator object S_UpperA
57      extends Transition('start, 'A(_), 'start)(idu)
58    // ...
59    @combinator object S_LowerA
60      extends Transition('start, 'a(_), 'start)(idu)
61    // ...
62    @combinator object S_Space
63      extends Transition('start, 'space(_), 'start)(idu)
64
65    // Gamma 3
66    @combinator object WFire_UpperA
67      extends Transition('wFire, 'A(_), 'wFire)(idu)
68    // ...
69    @combinator object WFire_LowerA
70      extends Transition('wFire, 'a(_), 'wFire)(idu)
71    // ...
72    @combinator object WFire_Space
73      extends Transition('wFire, 'space(_), 'wFire)(idu)
74
75    @combinator object WIce_UpperA
76      extends Transition('wIce, 'A(_), 'wIce)(idu)
77    // ...
78    @combinator object WIce_LowerA
79      extends Transition('wIce, 'a(_), 'wIce)(idu)
80    // ...
81    @combinator object WIce_Space
82      extends Transition('wIce, 'space(_), 'wIce)(idu)
```

```
}
```

Listing B.7: Klasse `Skill`

```
case class Skill(name: String)
```

Listing B.8: Enumeration über mehrere Worttypen

```
implicit class TermTypeSeqEnumerator(termTypes: Seq[Constructor]) {
  def enumerate: Enumeration[Type] =
    termTypes.map(_.enumerate).reduce(_ union _)
}

val wordTypes = Seq(
  'F('i('r('e('b('a('l('l('epsilon)))))))),
  'W('a('l('l('space('o('f('space('I('c('e('epsilon))))))))))),
  'I('c('e('space('L('a('n('c('e('epsilon)))))))),
  'P('o('i('s('o('n('epsilon)))))),
  'D('e('e('p('space('C('u('t('epsilon))))))))
)
val kinding = Kinding(repository.alpha).addOptions(wordTypes.enumerate)
```

Listing B.9: Trait `TreeGrammarVariables`

```
trait TreeGrammarVariables {
  val alpha1 = Variable("alpha1")
  val alpha2 = Variable("alpha2")
  val alpha3 = Variable("alpha3")
}
```

Listing B.10: Trait `TreeGrammarCombinators`

```
trait TreeGrammarCombinators extends TreeGrammarVariables {
  class SingleNonterminalProduction[A](leftNonterminal: Type,
      rightNonterminal: Type) {
    def apply(x: A): A = x
    val semanticType = (rightNonterminal =>: 'Term(alpha1)) =>:
                       (leftNonterminal =>: 'Term(alpha1))
  }

  class TerminalProduction[A](nonterminal: Type, terminal: Type, value: A) {
    def apply(): A = value
    val semanticType = nonterminal =>: 'Term(terminal)
  }

  class Start[A](S: Type) {
    def apply(x: A): A = x
    val semanticType = (S =>: 'Term(alpha1)) =>: 'Term(alpha1)
  }
```

```
}
```

Listing B.11: Trait `ListRepository`

```
trait ListRepository extends TreeGrammarVariables
    with TreeGrammarCombinators {
  @combinator object StartList extends Start[List[Int]]('NtList)

  @combinator object List1 extends TerminalProduction[List[Int]]('NtList,
    'nil, List.empty[Int])

  @combinator object List2 {
    def apply(e: Int, l1: List[Int]): List[Int] = e +: l1
    val semanticType = ('NtNat =>: 'Term(alpha1)) =>:
                       ('NtList =>: 'Term(alpha2)) =>:
                       ('NtList =>: 'Term('cons(alpha1, alpha2)))
  }

  @combinator object Nat1 extends TerminalProduction[Int]('NtNat, 'zero, 0)

  @combinator object Nat2 {
    def apply(n: Int): Int = n + 1
    val semanticType = ('NtNat =>: 'Term(alpha1)) =>:
                       ('NtNat =>: 'Term('s(alpha1)))
  }
}
```

Listing B.12: Trait `RtgGameRepository`

```
trait RtgGameRepository extends TreeGrammarCombinators
    with TreeGrammarVariables {
  trait SkillCombinator {
    def apply(skillInfo: SkillInfo): Skill = Skill.fromSkillInfo(skillInfo)
  }

  // Gamma F'
  @combinator object Fireball extends SkillCombinator {
    val semanticType = 'Term('skinfo('mage, 'dlcons('fire, 'n2, 'dlnil),
      'cost('mana, 'n1))) =>: 'Skill
  }

  @combinator object WallOfIce extends SkillCombinator {
    val semanticType = 'Term('skinfo('mage, 'dlnil,
      'cost('mana, 'n4))) =>: 'Skill
  }

  @combinator object IceLance extends SkillCombinator {
    val semanticType = 'Term('skinfo('mage, 'dlcons('ice, 'n3,
```

```
20        'dlcons('physical, 'n2, 'dlnil)), 'cost('mana, 'n3))) =>: 'Skill
21    }
22
23    @combinator object Poison extends SkillCombinator {
24      val semanticType = 'Term('skinfo('rogue, 'dlcons('poison, 'n3, 'dlnil),
25        'cost('stamina, 'n2))) =>: 'Skill
26    }
27
28    @combinator object DeepCut extends SkillCombinator {
29      val semanticType = 'Term('skinfo('warrior, 'dlcons('physical, 'n2,
30        'dlcons('bleed, 'n4, 'dlnil)), 'cost('stamina, 'n3))) =>: 'Skill
31    }
32
33    @combinator object SkillSet {
34      def apply(s1: Skill, s2: Skill): List[Skill] = List(s1, s2)
35      val semanticType = 'Skill =>: 'Skill =>: 'SkillSet
36    }
37
38    // Gamma G_A, Section 5.4.2
39    @combinator object StartSkillInfo
40      extends Start[List[SkillInfo]]('NtSkillInfo)
41
42    @combinator object SkillInfo1 {
43      def apply(archetype: Archetype, damageList: List[Damage],
44        cost: Cost): SkillInfo = SkillInfo(archetype, damageList, cost)
45      val semanticType = ('NtArchetype =>: 'Term(alpha1)) =>:
46        ('NtFullDamageList =>: 'Term(alpha2)) =>: ('NtCost =>: 'Term(alpha3)) =>:
47        ('NtSkillInfo =>: 'Term('skinfo(alpha1, alpha2, alpha3)))
48    }
49
50    @combinator object Archetype1
51      extends TerminalProduction[Archetype]('NtArchetype, 'mage, Mage)
52    @combinator object Archetype2
53      extends TerminalProduction[Archetype]('NtArchetype, 'rogue, Rogue)
54
55    @combinator object FullDamageList1 {
56      def apply(damageType: DamageType, damageValue: Int,
57          damageList: List[Damage]): List[Damage] = {
58        Damage(damageType, damageValue) +: damageList
59      }
60      val semanticType = ('NtDamageType =>: 'Term(alpha1)) =>:
61        ('NtDamageValue =>: 'Term(alpha2)) =>:
62        ('NtDamageList =>: 'Term(alpha3)) =>:
63        ('NtFullDamageList =>: 'Term('dlcons(alpha1, alpha2, alpha3)))
64    }
65
66    @combinator object DamageList1
67      extends TerminalProduction[List[Damage]]('NtDamageList, 'dlnil, List())
```

```
68
69  @combinator object DamageList2 {
70    def apply(damageType: DamageType, damageValue: Int,
71        damageList: List[Damage]): List[Damage] = {
72      Damage(damageType, damageValue) +: damageList
73    }
74    val semanticType = ('NtDamageType =>: 'Term(alpha1)) =>:
75      ('NtDamageValue =>: 'Term(alpha2)) =>:
76      ('NtDamageList =>: 'Term(alpha3)) =>:
77      ('NtDamageList =>: 'Term('dlcons(alpha1, alpha2, alpha3)))
78  }
79
80  @combinator object DamageType1
81    extends TerminalProduction[DamageType]('NtDamageType, 'fire, Fire)
82  @combinator object DamageType2
83    extends TerminalProduction[DamageType]('NtDamageType, 'ice, Ice)
84  @combinator object DamageType3 extends TerminalProduction[DamageType](
85    'NtDamageType, 'physical, Physical)
86
87  @combinator object DamageValue1
88    extends TerminalProduction[Int]('NtDamageValue, 'n1, 1)
89  @combinator object DamageValue2
90    extends TerminalProduction[Int]('NtDamageValue, 'n2, 2)
91  @combinator object DamageValue3
92    extends TerminalProduction[Int]('NtDamageValue, 'n3, 3)
93  @combinator object DamageValue4
94    extends TerminalProduction[Int]('NtDamageValue, 'n4, 4)
95  @combinator object DamageValue5
96    extends TerminalProduction[Int]('NtDamageValue, 'n5, 5)
97
98  @combinator object Cost1 {
99    def apply(costType: CostType, costValue: Int): Cost =
100     Cost(costType, costValue)
101   val semanticType = ('NtCostType =>: 'Term(alpha1)) =>:
102     ('NtCostValue =>: 'Term(alpha2)) =>:
103     ('NtCost =>: 'Term('cost(alpha1, alpha2)))
104 }
105
106 @combinator object CostType1
107   extends TerminalProduction[CostType]('NtCostType, 'mana, Mana)
108 @combinator object CostType2
109   extends TerminalProduction[CostType]('NtCostType, 'stamina, Stamina)
110
111 @combinator object CostValue1
112   extends TerminalProduction[Int]('NtCostValue, 'n2, 2)
113 @combinator object CostValue2
114   extends TerminalProduction[Int]('NtCostValue, 'n3, 3)
115 @combinator object CostValue3
```

```
116     extends TerminalProduction[Int]('NtCostValue, 'n4, 4)
117 }
```

Listing B.13: Typdeklarationen zu `RtgGameRepository`

```
1  sealed trait Archetype
2  case object Warrior extends Archetype
3  case object Mage extends Archetype
4  case object Rogue extends Archetype
5
6  sealed trait DamageType
7  case object Fire extends DamageType
8  case object Ice extends DamageType
9  case object Physical extends DamageType
10 case object Poison extends DamageType
11 case object Bleed extends DamageType
12
13 sealed trait CostType
14 case object Health extends CostType
15 case object Mana extends CostType
16 case object Stamina extends CostType
17
18 case class Damage(damageType: DamageType, damageValue: Int)
19 case class Cost(costType: CostType, costValue: Int)
20 case class SkillInfo(archetype: Archetype, damageList: List[Damage],
21   cost: Cost)
22
23 case class Skill(archetype: Archetype, damageList: List[Damage], cost: Cost)
24
25 object Skill {
26   def fromSkillInfo(skillInfo: SkillInfo) =
27     Skill(skillInfo.archetype, skillInfo.damageList, skillInfo.cost)
28 }
```

Anhang C

DoSy-Baumgrammatik, Repository und Quellcode

C.1 DoSy-Baumgrammatik

$$
\begin{aligned}
\mathit{AssignTemplates} &\rightarrow \mathit{DoDefaultWithNetworkName} \\
&\mid \mathit{SetTomcatPort} \\
\mathit{DoDefaultWithNetworkName} &\rightarrow \mathit{tp}(\mathit{true}, \mathit{MaybeNumber}, \mathit{WithNetworkName}) \\
\mathit{SetTomcatPort} &\rightarrow \mathit{tp}(\mathit{Bool}, \mathit{Number}, \mathit{WithNetworkName}) \\
\mathit{WithNetworkName} &\rightarrow \mathit{DoDefaultWithPort} \\
&\mid \mathit{AssignNetworkName} \\
\mathit{DoDefaultWithPort} &\rightarrow \mathit{nn}(\mathit{true}, \mathit{MaybeString}, \mathit{WithPort}) \\
\mathit{AssignNetworkName} &\rightarrow \mathit{nn}(\mathit{Bool}, \mathit{String}, \mathit{WithPort}) \\
\mathit{WithPort} &\rightarrow \mathit{DoDefaultWithDependsOn} \\
&\mid \mathit{AssignPort} \\
\mathit{DoDefaultWithDependsOn} &\rightarrow \mathit{wp}(\mathit{true}, \mathit{MaybeNumber}, \mathit{AssignDependsOn}) \\
\mathit{AssignPort} &\rightarrow \mathit{wp}(\mathit{Bool}, \mathit{Number}, \mathit{AssignDependsOn}) \\
\mathit{AssignDependsOn} &\rightarrow \mathit{DoDefaultDatabase} \\
&\mid \mathit{DoEmptyDatabase} \\
&\mid \mathit{LoadSQLFile} \\
\mathit{DoDefaultDatabase} &\rightarrow \mathit{db}(\mathit{true}, \mathit{Bool}, \mathit{MaybeString}, \mathit{WithUsers}) \\
\mathit{DoEmptyDatabase} &\rightarrow \mathit{db}(\mathit{Bool}, \mathit{true}, \mathit{MaybeString}, \mathit{WithUsers}) \\
\mathit{LoadSQLFile} &\rightarrow \mathit{db}(\mathit{Bool}, \mathit{Bool}, \mathit{String}, \mathit{WithUsers}) \\
\mathit{WithUsers} &\rightarrow \mathit{DoDefaultWithContainers}
\end{aligned}
$$

$$
\begin{aligned}
&\mid \mathit{SetUsers}\\
\mathit{DoDefaultWithContainers} &\rightarrow \mathit{user}(\mathit{true}, \mathit{MaybeUserConfig}, \mathit{WithContainers})\\
\mathit{SetUsers} &\rightarrow \mathit{user}(\mathit{Bool}, \mathit{UserConfig}, \mathit{WithContainers})\\
\mathit{UserConfig} &\rightarrow \mathit{userConfig}(\mathit{String}, \mathit{String}, \mathit{String}, \mathit{String})\\
\mathit{WithContainers} &\rightarrow \mathit{DoDefaultEmptyWebApp}\\
&\mid \mathit{CreateContainersWithoutCluster}\\
&\mid \mathit{CreateContainersWithCluster}\\
\mathit{DoDefaultEmptyWebApp} &\rightarrow \mathit{cont}(\mathit{true}, \mathit{MaybeNumber}, \mathit{MaybeClusterConfig}, \mathit{EmptyWebApp})\\
\mathit{CreateContainersWithoutCluster} &\rightarrow \mathit{cont}(\mathit{Bool}, \mathit{Number}, \mathit{MaybeClusterConfig}, \mathit{EmptyWebApp})\\
\mathit{CreateContainersWithCluster} &\rightarrow \mathit{cont}(\mathit{Bool}, \mathit{MaybeNumber}, \mathit{ClusterConfig}, \mathit{EmptyWebApp})\\
\mathit{ClusterConfig} &\rightarrow \mathit{clusterConfig}(\mathit{Number}, \mathit{Number}, \mathit{Number})\\
\mathit{EmptyWebApp} &\rightarrow \mathit{empty}\\
\mathit{Bool} &\rightarrow \mathit{true} \mid \mathit{false}\\
\mathit{MaybeNumber} &\rightarrow \mathit{Number} \mid \mathit{none}\\
\mathit{MaybeString} &\rightarrow \mathit{String} \mid \mathit{none}\\
\mathit{MaybeUserConfig} &\rightarrow \mathit{UserConfig} \mid \mathit{none}\\
\mathit{MaybeClusterConfig} &\rightarrow \mathit{ClusterConfig} \mid \mathit{none}\\
\mathit{Number} &\rightarrow \mathit{d0}(\mathit{Number}) \mid \ldots \mid \mathit{d9}(\mathit{Number}) \mid \mathit{end}\\
\mathit{String} &\rightarrow \mathit{chA}(\mathit{String}) \mid \ldots \mid \mathit{chZ}(\mathit{String})\\
&\mid \mathit{cha}(\mathit{String}) \mid \ldots \mid \mathit{chz}(\mathit{String})\\
&\mid \mathit{ch0}(\mathit{String}) \mid \ldots \mid \mathit{ch9}(\mathit{String})\\
&\mid \mathit{chDot}(\mathit{String}) \mid \mathit{chSlash}(\mathit{String})\\
&\mid \mathit{chUnderscore}(\mathit{String}) \mid \mathit{epsilon}
\end{aligned}
$$

Die Teilterme der Einstellungen und Inputs haben dabei folgende Bedeutungen (dies lässt sich auch aus dem DoSy-Automaten ablesen):

- *tp*(default?, tomcatPort, ...)
- *nn*(default?, networkName, ...)
- *wp*(default?, port, ...)
- *db*(default database?, empty database?, sql file path, ...)
- *user*(default?, user config, ...)
- *userConfig*(tomcat username, tomcat password, mysql username, mysql password)
- *cont*(default?, tomcat count, cluster config, ...)
- *clusterConfig*(tomcat count, data node count, sql node count)

C.2 DoSy-Repository

$$
\begin{aligned}
\texttt{Start} &: (\texttt{AssignTemplates} \to \texttt{Term}(\alpha_1)) \to \texttt{Term}(\alpha_1) \\
\texttt{AssignTemplates}_1 &: (\texttt{DoDefaultWithNetworkName} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{AssignTemplates} \to \texttt{Term}(\alpha_1)) \\
\texttt{AssignTemplates}_2 &: (\texttt{SetTomcatPort} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{AssignTemplates} \to \texttt{Term}(\alpha_1)) \\
\texttt{DoDefaultWithNetworkName} &: (\texttt{MaybeNumber} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithNetworkName} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{DoDefaultWithNetworkName} \to \texttt{Term}(\texttt{tp}(\texttt{true}, \alpha_1, \alpha_2))) \\
\texttt{SetTomcatPort} &: (\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{Number} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{WithNetworkName} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{SetTomcatPort} \to \texttt{Term}(\texttt{tp}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{WithNetworkName}_1 &: (\texttt{DoDefaultWithPort} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithNetworkName} \to \texttt{Term}(\alpha_1)) \\
\texttt{WithNetworkName}_2 &: (\texttt{AssignNetworkName} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithNetworkName} \to \texttt{Term}(\alpha_1)) \\
\texttt{DoDefaultWithPort} &: (\texttt{MaybeString} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithPort} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{DoDefaultWithPort} \to \texttt{Term}(\texttt{nn}(\texttt{true}, \alpha_1, \alpha_2))) \\
\texttt{AssignNetworkName} &: (\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{String} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{WithPort} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{AssignNetworkName} \to \texttt{Term}(\texttt{nn}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{WithPort}_1 &: (\texttt{DoDefaultWithDependsOn} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithPort} \to \texttt{Term}(\alpha_1)) \\
\texttt{WithPort}_2 &: (\texttt{AssignPort} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithPort} \to \texttt{Term}(\alpha_1)) \\
\texttt{DoDefaultWithDependsOn} &: (\texttt{MaybeNumber} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{AssignDependsOn} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{DoDefaultWithDependsOn} \to \texttt{Term}(\texttt{wp}(\texttt{true}, \alpha_1, \alpha_2))) \\
\texttt{AssignPort} &: (\texttt{Bool} \to \texttt{Term}(\alpha_1))
\end{aligned}
$$

$$\begin{aligned}
&\to (\texttt{Number} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{AssignDependsOn} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{AssignPort} \to \texttt{Term}(\texttt{wp}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{AssignDependsOn}_1 : &(\texttt{DoDefaultDatabase} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{AssignDependsOn} \to \texttt{Term}(\alpha_1)) \\
\texttt{AssignDependsOn}_2 : &(\texttt{DoEmptyDatabase} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{AssignDependsOn} \to \texttt{Term}(\alpha_1)) \\
\texttt{AssignDependsOn}_3 : &(\texttt{LoadSQLFile} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{AssignDependsOn} \to \texttt{Term}(\alpha_1)) \\
\texttt{DoDefaultDatabase} : &(\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeString} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{WithUsers} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{DoDefaultDatabase} \to \texttt{Term}(\texttt{db}(\texttt{true}, \alpha_1, \alpha_2, \alpha_3))) \\
\texttt{DoEmptyDatabase} : &(\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeString} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{WithUsers} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{DoEmptyDatabase} \to \texttt{Term}(\texttt{db}(\alpha_1, \texttt{true}, \alpha_2, \alpha_3))) \\
\texttt{LoadSQLFile} : &(\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{Bool} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{String} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{WithUsers} \to \texttt{Term}(\alpha_4)) \\
&\to (\texttt{LoadSQLFile} \to \texttt{Term}(\texttt{db}(\alpha_1, \alpha_2, \alpha_3, \alpha_4))) \\
\texttt{WithUsers}_1 : &(\texttt{DoDefaultWithContainers} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithUsers} \to \texttt{Term}(\alpha_1)) \\
\texttt{WithUsers}_2 : &(\texttt{SetUsers} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithUsers} \to \texttt{Term}(\alpha_1)) \\
\texttt{DoDefaultWithContainers} : &(\texttt{MaybeUserConfig} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithContainers} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{DoDefaultWithContainers} \to \texttt{Term}(\texttt{user}(\texttt{true}, \alpha_1, \alpha_2))) \\
\texttt{SetUsers} : &(\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{UserConfig} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{WithContainers} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{SetUsers} \to \texttt{Term}(\texttt{user}(\alpha_1, \alpha_2, \alpha_3)))
\end{aligned}$$

$$
\begin{aligned}
\texttt{UserConfig} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{String} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{String} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{String} \to \texttt{Term}(\alpha_4)) \\
&\to (\texttt{UserConfig} \to \texttt{Term}(\texttt{userConfig}(\alpha_1, \alpha_2, \alpha_3, \alpha_4))) \\
\texttt{WithContainers}_1 &: (\texttt{DoDefaultEmptyWebApp} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithContainers} \to \texttt{Term}(\alpha_1)) \\
\texttt{WithContainers}_2 &: (\texttt{CreateContainersWithoutCluster} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithContainers} \to \texttt{Term}(\alpha_1)) \\
\texttt{WithContainers}_3 &: (\texttt{CreateContainersWithCluster} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{WithContainers} \to \texttt{Term}(\alpha_1)) \\
\texttt{DoDefaultEmptyWebApp} &: (\texttt{MaybeNumber} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeClusterConfig} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{EmptyWebApp} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{DoDefaultEmptyWebApp} \to \texttt{Term}(\texttt{cont}(\texttt{true}, \alpha_1, \alpha_2, \alpha_3))) \\
\texttt{CreateContainersWithoutCluster} &: (\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{Number} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{MaybeClusterConfig} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{EmptyWebApp} \to \texttt{Term}(\alpha_4)) \\
&\to (\texttt{CreateContainersWithoutCluster} \\
&\quad \to \texttt{Term}(\texttt{cont}(\alpha_1, \alpha_2, \alpha_3, \alpha_4))) \\
\texttt{CreateContainersWithCluster} &: (\texttt{Bool} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeNumber} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{ClusterConfig} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{EmptyWebApp} \to \texttt{Term}(\alpha_4)) \\
&\to (\texttt{CreateContainersWithCluster} \\
&\quad \to \texttt{Term}(\texttt{cont}(\alpha_1, \alpha_2, \alpha_3, \alpha_4))) \\
\texttt{ClusterConfig} &: (\texttt{Number} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{Number} \to \texttt{Term}(\alpha_2)) \\
&\to (\texttt{Number} \to \texttt{Term}(\alpha_3)) \\
&\to (\texttt{ClusterConfig} \to \texttt{Term}(\texttt{clusterConfig}(\alpha_1, \alpha_2, \alpha_3))) \\
\texttt{EmptyWebApp} &: \texttt{EmptyWebApp} \to \texttt{Term}(\texttt{empty}) \\
\texttt{Bool}_1 &: \texttt{Bool} \to \texttt{Term}(\texttt{true})
\end{aligned}
$$

$$
\begin{aligned}
\texttt{Bool}_2 &: \texttt{Bool} \to \texttt{Term}(\texttt{false}) \\
\texttt{MaybeNumber}_1 &: (\texttt{Number} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeNumber} \to \texttt{Term}(\alpha_1)) \\
\texttt{MaybeNumber}_2 &: (\texttt{MaybeNumber} \to \texttt{Term}(\texttt{none})) \\
\texttt{MaybeString}_1 &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeString} \to \texttt{Term}(\alpha_1)) \\
\texttt{MaybeString}_2 &: (\texttt{MaybeString} \to \texttt{Term}(\texttt{none})) \\
\texttt{MaybeUserConfig}_1 &: (\texttt{UserConfig} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeUserConfig} \to \texttt{Term}(\alpha_1)) \\
\texttt{MaybeUserConfig}_2 &: (\texttt{MaybeUserConfig} \to \texttt{Term}(\texttt{none})) \\
\texttt{MaybeClusterConfig}_1 &: (\texttt{ClusterConfig} \to \texttt{Term}(\alpha_1)) \\
&\to (\texttt{MaybeClusterConfig} \to \texttt{Term}(\alpha_1)) \\
\texttt{MaybeClusterConfig}_2 &: (\texttt{MaybeClusterConfig} \to \texttt{Term}(\texttt{none})) \\
\texttt{NumberD0} &: (\texttt{Number} \to \texttt{Term}(\alpha_1)) \to (\texttt{Number} \to \texttt{Term}(\texttt{d0}(\alpha_1))) \\
&\vdots \\
\texttt{NumberD9} &: (\texttt{Number} \to \texttt{Term}(\alpha_1)) \to (\texttt{Number} \to \texttt{Term}(\texttt{d9}(\alpha_1))) \\
\texttt{NumberEnd} &: \texttt{Number} \to \texttt{Term}(\texttt{end}) \\
\texttt{StringChA} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{chA}(\alpha_1))) \\
&\vdots \\
\texttt{StringChZ} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{chZ}(\alpha_1))) \\
\texttt{StringCha} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{cha}(\alpha_1))) \\
&\vdots \\
\texttt{StringChz} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{chz}(\alpha_1))) \\
\texttt{StringCh0} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{ch0}(\alpha_1))) \\
&\vdots \\
\texttt{StringCh9} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{ch9}(\alpha_1))) \\
\texttt{StringChDot} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{chDot}(\alpha_1))) \\
\texttt{StringChSlash} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{chSlash}(\alpha_1))) \\
\texttt{StringChUnderscore} &: (\texttt{String} \to \texttt{Term}(\alpha_1)) \to (\texttt{String} \to \texttt{Term}(\texttt{chUnderscore}(\alpha_1))) \\
\texttt{StringEpsilon} &: \texttt{String} \to \texttt{Term}(\texttt{epsilon})
\end{aligned}
$$

C.3 DoSy-Quellcode

Listing C.1: Trait `Variables`

```
trait Variables {
  val alpha1 = Variable("alpha1")
  val alpha2 = Variable("alpha2")
  val alpha3 = Variable("alpha3")
  val alpha4 = Variable("alpha4")
}
```

Listing C.2: Trait `TreeGrammarCombinators`

```
trait TreeGrammarCombinators extends Variables{
  class SingleNonterminalProduction[A](leftNonterminal: Type,
      rightNonterminal: Type) {
    def apply(x: A): A = x
    val semanticType = (rightNonterminal =>: 'Term(alpha1)) =>:
                       (leftNonterminal =>: 'Term(alpha1))
  }

  class TerminalProduction[A](nonterminal: Type, terminal: Type, value: A) {
    def apply(): A = value
    val semanticType = nonterminal =>: 'Term (terminal)
  }

  class Start[A](S: Type) {
    def apply(x: A): A = x
    val semanticType = (S =>: 'Term(alpha1)) =>: 'Term(alpha1)
  }
}
```

Listing C.3: Trait `NumberRepository`

```
trait NumberRepository extends Variables {
  implicit class NumberList(list: List[Int]) {
    def toInt: Int = {
      val highestIndex = list.length - 1
      val (_, intValue) = list.foldLeft((highestIndex, 0)) {
        case ((index, sum), digit) =>
          (index - 1, sum + digit * scala.math.pow(10, index).toInt)
      }
      intValue
    }
  }

  class NumberDigit(digit: Int, constructor: Type => Constructor) {
    def apply(list: List[Int]): List[Int] = digit :: list
    val semanticType = ('Number =>: 'Term(alpha1)) =>:
```

```
16                          ('Number =>: 'Term(constructor(alpha1)))
17    }
18
19    @combinator object NumberD0 extends NumberDigit(0, 'd0(_))
20    @combinator object NumberD1 extends NumberDigit(1, 'd1(_))
21    @combinator object NumberD2 extends NumberDigit(2, 'd2(_))
22    @combinator object NumberD3 extends NumberDigit(3, 'd3(_))
23    @combinator object NumberD4 extends NumberDigit(4, 'd4(_))
24    @combinator object NumberD5 extends NumberDigit(5, 'd5(_))
25    @combinator object NumberD6 extends NumberDigit(6, 'd6(_))
26    @combinator object NumberD7 extends NumberDigit(7, 'd7(_))
27    @combinator object NumberD8 extends NumberDigit(8, 'd8(_))
28    @combinator object NumberD9 extends NumberDigit(9, 'd9(_))
29
30    @combinator object NumberEnd {
31      def apply(): List[Int] = List[Int]()
32      val semanticType = 'Number =>: 'Term('end)
33    }
34  }
```

Listing C.4: Trait `StringRepository`

```
1   trait StringRepository extends Variables {
2     class StringChar(char: String, constructor: Type => Constructor) {
3       def apply(string: String): String = char + string
4       val semanticType = ('String =>: 'Term(alpha1)) =>:
5                          ('String =>: 'Term(constructor(alpha1)))
6     }
7
8     @combinator object StringCharUpperA extends StringChar("A", 'chA(_))
9     @combinator object StringCharUpperB extends StringChar("B", 'chB(_))
10    @combinator object StringCharUpperC extends StringChar("C", 'chC(_))
11    @combinator object StringCharUpperD extends StringChar("D", 'chD(_))
12    @combinator object StringCharUpperE extends StringChar("E", 'chE(_))
13    @combinator object StringCharUpperF extends StringChar("F", 'chF(_))
14    @combinator object StringCharUpperG extends StringChar("G", 'chG(_))
15    @combinator object StringCharUpperH extends StringChar("H", 'chH(_))
16    @combinator object StringCharUpperI extends StringChar("I", 'chI(_))
17    @combinator object StringCharUpperJ extends StringChar("J", 'chJ(_))
18    @combinator object StringCharUpperK extends StringChar("K", 'chK(_))
19    @combinator object StringCharUpperL extends StringChar("L", 'chL(_))
20    @combinator object StringCharUpperM extends StringChar("M", 'chM(_))
21    @combinator object StringCharUpperN extends StringChar("N", 'chN(_))
22    @combinator object StringCharUpperO extends StringChar("O", 'chO(_))
23    @combinator object StringCharUpperP extends StringChar("P", 'chP(_))
24    @combinator object StringCharUpperQ extends StringChar("Q", 'chQ(_))
25    @combinator object StringCharUpperR extends StringChar("R", 'chR(_))
26    @combinator object StringCharUpperS extends StringChar("S", 'chS(_))
```

```
27  @combinator object StringCharUpperT extends StringChar("T", 'chT(_))
28  @combinator object StringCharUpperU extends StringChar("U", 'chU(_))
29  @combinator object StringCharUpperV extends StringChar("V", 'chV(_))
30  @combinator object StringCharUpperW extends StringChar("W", 'chW(_))
31  @combinator object StringCharUpperX extends StringChar("X", 'chX(_))
32  @combinator object StringCharUpperY extends StringChar("Y", 'chY(_))
33  @combinator object StringCharUpperZ extends StringChar("Z", 'chZ(_))
34
35  @combinator object StringCharLowerA extends StringChar("a", 'cha(_))
36  @combinator object StringCharLowerB extends StringChar("b", 'chb(_))
37  @combinator object StringCharLowerC extends StringChar("c", 'chc(_))
38  @combinator object StringCharLowerD extends StringChar("d", 'chd(_))
39  @combinator object StringCharLowerE extends StringChar("e", 'che(_))
40  @combinator object StringCharLowerF extends StringChar("f", 'chf(_))
41  @combinator object StringCharLowerG extends StringChar("g", 'chg(_))
42  @combinator object StringCharLowerH extends StringChar("h", 'chh(_))
43  @combinator object StringCharLowerI extends StringChar("i", 'chi(_))
44  @combinator object StringCharLowerJ extends StringChar("j", 'chj(_))
45  @combinator object StringCharLowerK extends StringChar("k", 'chk(_))
46  @combinator object StringCharLowerL extends StringChar("l", 'chl(_))
47  @combinator object StringCharLowerM extends StringChar("m", 'chm(_))
48  @combinator object StringCharLowerN extends StringChar("n", 'chn(_))
49  @combinator object StringCharLowerO extends StringChar("o", 'cho(_))
50  @combinator object StringCharLowerP extends StringChar("p", 'chp(_))
51  @combinator object StringCharLowerQ extends StringChar("q", 'chq(_))
52  @combinator object StringCharLowerR extends StringChar("r", 'chr(_))
53  @combinator object StringCharLowerS extends StringChar("s", 'chs(_))
54  @combinator object StringCharLowerT extends StringChar("t", 'cht(_))
55  @combinator object StringCharLowerU extends StringChar("u", 'chu(_))
56  @combinator object StringCharLowerV extends StringChar("v", 'chv(_))
57  @combinator object StringCharLowerW extends StringChar("w", 'chw(_))
58  @combinator object StringCharLowerX extends StringChar("x", 'chx(_))
59  @combinator object StringCharLowerY extends StringChar("y", 'chy(_))
60  @combinator object StringCharLowerZ extends StringChar("z", 'chz(_))
61
62  @combinator object StringChar0 extends StringChar("0", 'ch0(_))
63  @combinator object StringChar1 extends StringChar("1", 'ch1(_))
64  @combinator object StringChar2 extends StringChar("2", 'ch2(_))
65  @combinator object StringChar3 extends StringChar("3", 'ch3(_))
66  @combinator object StringChar4 extends StringChar("4", 'ch4(_))
67  @combinator object StringChar5 extends StringChar("5", 'ch5(_))
68  @combinator object StringChar6 extends StringChar("6", 'ch6(_))
69  @combinator object StringChar7 extends StringChar("7", 'ch7(_))
70  @combinator object StringChar8 extends StringChar("8", 'ch8(_))
71  @combinator object StringChar9 extends StringChar("9", 'ch9(_))
72
73  @combinator object StringCharDot extends StringChar(".", 'chDot(_))
74  @combinator object StringCharSlash extends StringChar("/", 'chSlash(_))
```

```
75    @combinator object StringCharUnderscore
76      extends StringChar("_", 'chUnderscore(_))
77
78    @combinator object StringEpsilon {
79      def apply(): String = ""
80      val semanticType = 'String =>: 'Term ('epsilon)
81    }
82  }
```

Listing C.5: Trait `ConfigRepository`

```
1   trait ConfigRepository extends Variables with TreeGrammarCombinators
2       with NumberRepository with StringRepository {
3     @combinator object NewUserConfig {
4       def apply(tomcatUsername: String, tomcatPassword: String,
5                 mysqlUsername: String, mysqlPassword: String): UserConfig = {
6         UserConfig(tomcatUsername, tomcatPassword, mysqlUsername,
7                    mysqlPassword)
8       }
9       val semanticType = ('String =>: 'Term(alpha1)) =>:
10        ('String =>: 'Term(alpha2)) =>: ('String =>: 'Term(alpha3)) =>:
11        ('String =>: 'Term(alpha4)) =>:
12        ('UserConfig =>: 'Term('userConfig(alpha1, alpha2, alpha3, alpha4)))
13    }
14
15    @combinator object NewClusterConfig {
16      def apply(tomcatCount: List[Int], dataNodeCount: List[Int],
17          sqlNodeCount: List[Int]): ClusterConfig = {
18        ClusterConfig(tomcatCount.toInt, dataNodeCount.toInt,
19          sqlNodeCount.toInt)
20      }
21      val semanticType = ('Number =>: 'Term(alpha1)) =>:
22        ('Number =>: 'Term(alpha2)) =>: ('Number =>: 'Term(alpha3)) =>:
23        ('ClusterConfig =>: 'Term('clusterConfig(alpha1, alpha2, alpha3)))
24    }
25
26    @combinator object BoolTrue
27      extends TerminalProduction[Boolean]('Bool, 'true, true)
28    @combinator object BoolFalse
29      extends TerminalProduction[Boolean]('Bool, 'false, false)
30    @combinator object MaybeNumber
31      extends SingleNonterminalProduction[List[Int]]('MaybeNumber, 'Number)
32    @combinator object MaybeNumberNone
33      extends TerminalProduction[List[Int]]('MaybeNumber, 'none, null)
34    @combinator object MaybeString
35      extends SingleNonterminalProduction[String]('MaybeString, 'String)
36    @combinator object MaybeStringNone
37      extends TerminalProduction[String]('MaybeString, 'none, null)
```

```
38    @combinator object MaybeUserConfig
39      extends SingleNonterminalProduction[UserConfig]('MaybeUserConfig,
40        'UserConfig)
41    @combinator object MaybeUserConfigNone
42      extends TerminalProduction[UserConfig]('MaybeUserConfig, 'none, null)
43    @combinator object MaybeClusterConfig
44      extends SingleNonterminalProduction[ClusterConfig]('MaybeClusterConfig,
45        'ClusterConfig)
46    @combinator object MaybeClusterConfigNone
47      extends TerminalProduction[ClusterConfig]('MaybeClusterConfig,
48        'none, null)
49  }
50
51  object ConfigRepository {
52    case class UserConfig(tomcatUsername: String, tomcatPassword: String,
53      mysqlUsername: String, mysqlPassword: String)
54    case class ClusterConfig(tomcatCount: Int, dataNodeCount: Int,
55      sqlNodeCount: Int)
56  }
```

Listing C.6: Trait DosyRepository

```
1   trait DosyRepository extends Variables with ConfigRepository
2       with NumberRepository with StringRepository {
3     @combinator object StartDosy extends Start[WithTemplates](
4       'AssignTemplates)
5
6     @combinator object AssignTemplates1 {
7       def apply(app: WithTomcatPort): WithTemplates = {
8         app.doDefault().assignTemplates()
9       }
10      val semanticType = ('DoDefaultWithNetworkName =>: 'Term(alpha1)) =>:
11        ('AssignTemplates =>: 'Term(alpha1))
12    }
13
14    @combinator object AssignTemplates2 {
15      def apply(app: WithTomcatPort): WithTemplates = AssignTemplates1(app)
16      val semanticType = ('SetTomcatPort =>: 'Term(alpha1)) =>:
17        ('AssignTemplates =>: 'Term(alpha1))
18    }
19
20    @combinator object DoDefaultWithNetworkName {
21      def apply(any: List[Int], app: WithNetworkName): WithTomcatPort =
22        app.doDefault()
23      val semanticType = ('MaybeNumber =>: 'Term(alpha1)) =>:
24        ('WithNetworkName =>: 'Term(alpha2)) =>:
25        ('DoDefaultWithNetworkName =>: 'Term('tp('true, alpha1, alpha2)))
26    }
```

```
27
28  @combinator object SetTomcatPort {
29    def apply(any: Boolean, port: List[Int], app: WithNetworkName
30      ): WithTomcatPort = app.setTomcatPort(port.toInt)
31    val semanticType = ('Bool =>: 'Term(alpha1)) =>:
32      ('Number =>: 'Term(alpha2)) =>:
33      ('WithNetworkName =>: 'Term(alpha3)) =>:
34      ('DoDefaultWithNetworkName =>: 'Term('tp(alpha1, alpha2, alpha3)))
35  }
36
37  @combinator object WithNetworkName1
38    extends SingleNonterminalProduction[WebApp]('WithNetworkName,
39      'DoDefaultWithPort)
40  @combinator object WithNetworkName2
41    extends SingleNonterminalProduction[WebApp]('WithNetworkName,
42      'AssignNetworkName)
43
44  @combinator object DoDefaultWithPort {
45    def apply(any: String, app: WithPort): WithNetworkName = app.doDefault()
46    val semanticType = ('MaybeString =>: 'Term(alpha1)) =>:
47      ('WithPort =>: 'Term(alpha2)) =>:
48      ('DoDefaultWithPort =>: 'Term('nn('true, alpha1, alpha2)))
49  }
50
51  @combinator object AssignNetworkName {
52    def apply(any: Boolean, networkName: String, app: WithPort
53      ): WithNetworkName = app.assignNetworkName(networkName)
54    val semanticType = ('Bool =>: 'Term(alpha1)) =>:
55      ('String =>: 'Term(alpha2)) =>:
56      ('WithPort =>: 'Term(alpha3)) =>:
57      ('DoDefaultWithPort =>: 'Term('nn(alpha1, alpha2, alpha3)))
58  }
59
60  @combinator object WithPort1 extends SingleNonterminalProduction[WebApp](
61    'WithPort, 'DoDefaultWithDependsOn)
62  @combinator object WithPort2 extends SingleNonterminalProduction[WebApp](
63    'WithPort, 'AssignPort)
64
65  @combinator object DoDefaultWithDependsOn {
66    def apply(any: List[Int], app: WithDependsOn): WithPort =
67      app.doDefault()
68    val semanticType = ('MaybeNumber =>: 'Term(alpha1)) =>:
69      ('AssignDependsOn =>: 'Term(alpha2)) =>:
70      ('DoDefaultWithDependsOn =>: 'Term('wp('true, alpha1, alpha2)))
71  }
72
73  @combinator object AssignPort {
74    def apply(any: Boolean, port: List[Int], app: WithDependsOn): WithPort =
```

```
75        app.assignPort(port.toInt)
76      val semanticType = ('Bool =>: 'Term(alpha1)) =>:
77        ('Number =>: 'Term(alpha2)) =>:
78        ('AssignDependsOn =>: 'Term(alpha3)) =>:
79        ('AssignPort =>: 'Term('wp(alpha1, alpha2, alpha3)))
80    }
81
82    @combinator object AssignDependsOn1 {
83      def apply(app: WithDatabase): WithDependsOn = app.assignDependsOn()
84      val semanticType = ('DoDefaultDatabase =>: 'Term(alpha1)) =>:
85        ('AssignDependsOn =>: 'Term(alpha1))
86    }
87
88    @combinator object AssignDependsOn2 {
89      def apply(app: WithDatabase): WithDependsOn = AssignDependsOn1(app)
90      val semanticType = ('DoEmptyDatabase =>: 'Term(alpha1)) =>:
91        ('AssignDependsOn =>: 'Term(alpha1))
92    }
93
94    @combinator object AssignDependsOn3 {
95      def apply(app: WithDatabase): WithDependsOn = AssignDependsOn1(app)
96      val semanticType = ('LoadSQLFile =>: 'Term(alpha1)) =>:
97        ('AssignDependsOn =>: 'Term(alpha1))
98    }
99
100   @combinator object DoDefaultDatabase {
101     def apply(any1: Boolean, any2: String, app: WithUsers): WithDatabase =
102       app.doDefaultDatabase()
103     val semanticType = ('Bool =>: 'Term(alpha1)) =>:
104       ('MaybeString =>: 'Term(alpha2)) =>:
105       ('WithUsers =>: 'Term(alpha3)) =>:
106       ('DoDefaultDatabase =>: 'Term('db('true, alpha1, alpha2, alpha3)))
107   }
108
109   @combinator object DoEmptyDatabase {
110     def apply(any1: Boolean, any2: String, app: WithUsers): WithDatabase =
111       app.doEmptyDatabase()
112     val semanticType = ('Bool =>: 'Term(alpha1)) =>:
113       ('MaybeString =>: 'Term(alpha2)) =>:
114       ('WithUsers =>: 'Term(alpha3)) =>:
115       ('DoEmptyDatabase =>: 'Term('db(alpha1, 'true, alpha2, alpha3)))
116   }
117
118   @combinator object LoadSQLFile {
119     def apply(any1: Boolean, any2: Boolean, filePath: String,
120       app: WithUsers): WithDatabase = app.loadSQLFile(filePath)
121     val semanticType = ('Bool =>: 'Term(alpha1)) =>:
122       ('Bool =>: 'Term(alpha2)) =>:
```

```
123       ('String =>: 'Term(alpha3)) =>:
124       ('WithUsers =>: 'Term(alpha4)) =>:
125       ('LoadSQLFile =>: 'Term('db(alpha1, alpha2, alpha3, alpha4)))
126   }
127
128   @combinator object WithUsers1 extends SingleNonterminalProduction[WebApp](
129     'WithUsers, 'DoDefaultWithContainers)
130   @combinator object WithUsers2 extends SingleNonterminalProduction[WebApp](
131     'WithUsers, 'SetUsers)
132
133   @combinator object DoDefaultWithContainers {
134     def apply(any: UserConfig, app: WithContainers): WithUsers =
135       app.doDefault()
136     val semanticType = ('MaybeUserConfig =>: 'Term(alpha1)) =>:
137       ('WithContainers =>: 'Term(alpha2)) =>:
138       ('DoDefaultWithContainers =>: 'Term('user('true, alpha1, alpha2)))
139   }
140
141   @combinator object SetUsers {
142     def apply(any: Boolean, userConfig: UserConfig,
143         app: WithContainers): WithUsers = {
144       app.setUsersFromCommandLine(userConfig.tomcatUsername,
145         userConfig.tomcatPassword, userConfig.mysqlUsername,
146         userConfig.mysqlPassword)
147     }
148     val semanticType = ('Bool =>: 'Term(alpha1)) =>:
149       ('UserConfig =>: 'Term(alpha2)) =>:
150       ('WithContainers =>: 'Term(alpha3)) =>:
151       ('SetUsers =>: 'Term('user(alpha1, alpha2, alpha3)))
152   }
153
154   @combinator object WithContainers1
155     extends SingleNonterminalProduction[WebApp]('WithContainers,
156       'DoDefaultEmptyWebApp)
157   @combinator object WithContainers2
158     extends SingleNonterminalProduction[WebApp]('WithContainers,
159       'CreateContainersWithoutCluster)
160   @combinator object WithContainers3
161     extends SingleNonterminalProduction[WebApp]('WithContainers,
162       'CreateContainersWithCluster)
163
164   @combinator object DoDefaultEmptyWebApp {
165     def apply(any1: List[Int], any2: ClusterConfig,
166       app: EmptyWebApp): WithContainers = app.doDefault()
167     val semanticType = ('MaybeNumber =>: 'Term(alpha1)) =>:
168       ('MaybeClusterConfig =>: 'Term(alpha2)) =>:
169       ('EmptyWebApp =>: 'Term(alpha3)) =>:
170       ('DoDefaultEmptyWebApp =>:
```

```
171         'Term('cont('true, alpha1, alpha2, alpha3)))
172   }
173
174   @combinator object CreateContainersWithoutCluster {
175     def apply(any1: Boolean, tomcatCount: List[Int], any2: ClusterConfig,
176         app: EmptyWebApp): WithContainers = {
177       app.createContainersWithoutCluster(tomcatCount.toInt)
178     }
179     val semanticType = ('Bool =>: 'Term(alpha1)) =>:
180       ('Number =>: 'Term(alpha2)) =>:
181       ('MaybeClusterConfig =>: 'Term(alpha3)) =>:
182       ('EmptyWebApp =>: 'Term(alpha4)) =>:
183       ('CreateContainersWithoutCluster =>:
184        'Term('cont(alpha1, alpha2, alpha3, alpha4)))
185   }
186
187
188   @combinator object CreateContainersWithCluster {
189     def apply(any1: Boolean, any2: List[Int], clusterConfig: ClusterConfig,
190         app: EmptyWebApp): WithContainers = {
191       app.createContainersWithCluster(clusterConfig.tomcatCount,
192         clusterConfig.dataNodeCount, clusterConfig.tomcatCount)
193     }
194     val semanticType = ('Bool =>: 'Term(alpha1)) =>:
195       ('MaybeNumber =>: 'Term(alpha2)) =>:
196       ('ClusterConfig =>: 'Term(alpha3)) =>:
197       ('EmptyWebApp =>: 'Term(alpha4)) =>:
198       ('CreateContainersWithCluster =>:
199        'Term('cont(alpha1, alpha2, alpha3, alpha4)))
200   }
201
202   @combinator object NewEmptyWebApp {
203     def apply(): EmptyWebApp = new EmptyWebApp()
204     val semanticType = 'EmptyWebApp =>: 'Term('empty)
205   }
206 }
```

Literaturverzeichnis

[1] Comon, H., M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison und M. Tommasi: *Tree Automata Techniques and Applications*. Available on: `http://tata.gforge.inria.fr/`, 2007.

[2] Düdder, Boris, Moritz Martens, Jakob Rehof und Paweł Urzyczyn: *Bounded Combinatory Logic*. In: *Computer Science Logic (CSL'12)*, Band 16 der Reihe *LIPIcs*, Seiten 243–258. Leibniz-Zentrum für Informatik, 2012.

[3] Heineman, George T., Jan Bessai, Boris Düdder und Jakob Rehof: *A Long and Winding Road Towards Modular Synthesis*. In: *ISoLA 2016, 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Part I*, Band 9952 der Reihe *Theoretical Computer Science and General Issues*. Springer International Publishing, 2016.

[4] Hopcroft, John E., Rajeev Motwani und Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2013.

[5] Lawson, Mark V.: *Finite Automata*. Chapman and Hall/CRC, 2003.

[6] Odersky, Martin: *The Scala Language Specification, Version 2.9*. Available on: `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2014.

[7] Rehof, Jakob: *Towards Combinatory Logic Synthesis*. In: *BEAT'13, 1st International Workshop on Behavioural Types*. ACM, 2013.

[8] Scholtyssek, Daniel: *Synthese von Docker-Konfigurationen unter Zuhilfenahme eines Inhabitationsalgorithmus*. Lehrstuhl 14, Fakultät für Informatik, TU Dortmund, 2016.