Fachbereich Informatik und Mathematik

Institut für Informatik

Diplomarbeit

Implementierung von Algorithmen für das Wortproblem auf Grundtermen bezüglich endlicher Mengen von Grundtermgleichungen für STG-komprimierte Grundterme

Hana Schäfer

2. April 2012

eingereicht bei

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz / Softwaretechnologie

Erklärung gemäß DPO §11 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Hanau, den 2. April 2012

Hana Schäfer

Danksagung

Besonderer Dank gilt Dr. David Sabel für seine Betreuung und schnelle Hilfestellung, sowie Prof. Dr. Schmidt-Schauß, der diese Arbeit erst ermöglicht hat.

Bedanken möchte ich mich bei allen, die mich während meines Studiums unterstützt und gefördert haben.

Weiterhin bedanke ich mich bei meinem Ehemann Sebastian - einfach für alles.

Zusammenfassung

Grundterme, die rein aus Terminalen bestehen, lassen sich durch Abspeichern in einer Singleton-Tree-Grammar (STG) bestenfalls auf logarithmische Größe komprimieren. Die aktuelle Forschung beschäftigt sich damit, Algorithmen für komprimiert dargestellte Terme zu entwicklen, die direkt auf der Grammatik arbeiten und eine Dekompression vermeiden. So können verbesserter Platzbedarf und asymptotische Laufzeiten erreicht werden.

Diese Arbeit beschäftigt sich mit einem kürzlich veröffentlichten PSPACE Algorithmus, der das Problem der Berechnung einer Normalform eines STG-komprimierten Terms für ein Termersetzungssystem (TRS) löst. Das Verfahren setzt voraus, dass das TRS reduziert ist und beide Seiten aller Regeln selbst wiederum STG-komprimierte Grundterme sind. Ziel dieser Arbeit ist die Untersuchung, die Umsetzung und das Testen des vorgestellten Algorithmus in der funktionalen Programmiersprache Haskell.

Zunächst werden die grundlegende Begriffe wie Terme, Termersetzungssystem und die Grammatik-basierte Kompression mit STGs definiert und erläutert. Auch ein kurzer Überblick über die Programmiersprache Haskell und das verwendete Werkzeug Happy wird gegeben. Im Anschluss wird der PSPACE Algorithmus zur Normalformberechnung vorgestellt und erläutert. Die zu implementierenden Funktionen werden zunächst detailiert konzipiert und deren Umsetzung dann ausführlich erörtert.

Zuerst wird ein Parser in Haskell umgesetzt, der als Klartext eingegebene STGs parst und eine verwendbare Instanz erzeugt. Durch den Parser können Grammatiken und Testfälle leicht zur Laufzeit eingegeben werden.

Das implementierte Termersetzungsverfahrens wird neben der reinen Funktionalität auch auf die Größe der ausgegebenen Grammatik und die Laufzeit des Verfahrens getestet. Die Berechnung der Normalform für einen STG-komprimierten Grundterm kann mit der geschriebenen Funktion in polynomiellen Platz mit exponentieller Laufzeit durchgeführt werden. Die Ergebnisgrammatik ist in ihrer Größe polynomiell zur

Größe der Eingabe und den Termen des TRS. Durch weitere Vereinfachungen, wie dem Entfernen von unnötigen Regeln und Nicht-Terminalen, kann die Grammatik optimiert und verkleinert werden.

Nach Darstellung und Interpretation der erzielten Testergebnisse, wird schließlich ein Fazit gezogen und ein Ausblick auf weitere Forschungsfragen gegeben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Datenkompression ist ein wichtiger Aspekt der angewandten Informatik. Daten verlustfrei zu komprimieren heisst, sie so zu transformieren, dass sie weniger Platz einnehmen ohne den Inhalt zu verändern. Die Größe der Daten wird meistens in physikalischen Parametern wie etwa Anzahl der benötigten Bits angegeben. Sie kann aber auch inhaltlich aufgefasst werden, wie zum Beispiel die Anzahl der Knoten in einem Graphen oder die Anzahl der Regeln einer Grammatik.

Eine Möglichkeit, formale Terme zu komprimieren, ist das Abspeichern mit einer speziellen Grammatik. Die Darstellung der Grammatik mit ihren Produktionsregeln ist also kleiner als das eigentliche abgespeicherte Wort. Mit einer *Singleton-Tree-Grammar* (kurz: STG) können Grundterme sogar um einen exponentiellen Faktor komprimiert werden. Beim Verarbeiten von STG-komprimierten Termen, wie etwa der Termersetzung mit einer endlicher Mengen von Grundtermgleichungen, soll der Platzvorteil beibehalten werden und ein Entpacken möglichst vermieden werden.

In dieser Arbeit wird ein Verfahren in der funktionalen Programmiersprache *Haskell* vorgestellt und implementiert, das Termersetzung mit reduzierten Grundtermersetzungssystemen in STG-komprimierten Grundtermen durchführt, ohne diese zu entpacken. Der Algorithmus hat *PSPACE*-Komplexität und erzeugt optimierte Grammatiken. Um das Wortproblem innerhalb der Ersetzung zu lösen wird der Algorithmus von Plandowski verwendet.

Die Implementierung wird mittels verschiedener Tests überprüft. Um das Testen zu erleichtern, wurde zusätzlich in Haskell ein Parser für STGs implementiert. Mit diesem können Grammatiken in gut lesbarer Form als reiner Text eingelesen und zu Instanzen in Haskell verarbeitet werden.

Überblick über die Arbeit

Im zweiten Kapitel werden die Grundlagen vorgestellt. Dazu zählen Definition von Termersetzungssystem gTRS und von der Singelton-Tree-Grammar STG. Das dritte Kapitel befasst sich mit einem Haskell-Modul zur Grammatik-basierten Kompression, dem Algorithmus von Plandowski und der Vorstellung des Verfahrens zur Termersetzung, auf dem diese Arbeit basiert.

Ein Konzept zur Lösung der in der Aufgabenstellung formulierten Probleme wird in Kapitel vier erarbeitet: STG-Parser und Termersetzung in Haskell. Den Schwerpunkt der Arbeit bildet das fünfte Kapitel. Es befasst sich mit der Umsetzung des Konzeptes und den Details der Implementierung. Diese wird im sechsten Kapitel ausgiebig getestet. Den Abschluß bildet das siebte Kapitel mit einer Bewertung der Ergebnisse der Arbeit und einem Ausblick.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit erörtert. Zunächst werden Terme und Termersetzungssysteme mitsamt einiger wichtiger Notationen erklärt. Dann die im folgenden betrachtete STG Grammatiken zur komprimierten Darstellung von Termen, ihre Eigenschaften sowie das verwendete Grundtermersetzungssystem gTRS definiert. Den Abschluß bildet eine Einführung in das Thema funktionale Programmiersprache Haskell und Parser.

2.1 Terme und Kontexte

2.1.1 Terme und Grundterme

Ein *Term* t ist ein syntaktisch korrektes Wort über einem Alphabet Σ und einer unendlichen Variablenmenge $\mathcal{V}$. Ein Term kann aus Untertermen zusammengesetzt werden und hat daher auch eine Struktur, ist also mehr als ein *flaches* Wort. Die Definitionen entstammen alle dem Skript der Vorlesung *Reduktionssysteme und Termersetzung* [SS10].

Definition 1 (Term) *Sei Σ ein Alphabet, d.h. eine Menge von Funktionssymbolen (Terminalzeichen), wobei jedes $f \in \Sigma$ eine feste Stelligkeit $ar(f) \geq 0$ hat. Sei $\mathcal{V}$ eine Menge von unendlich vielen Variablen und $\Sigma \cap \mathcal{V} = \emptyset$.*

Die Menge der Terme T über Σ und $\mathcal{V}$ (notiert als $\mathrm{Term}(\Sigma, \mathcal{V})$) ist wie folgt definiert:

- *Jedes $a \in \Sigma$ mit $ar(a) = 0$ ist ein Term.*
- *Jedes $v \in \mathcal{V}$ ist ein Term.*

- *Wenn $t_1, \ldots, t_n$ Terme sind und $ar(f) = n$, dann ist $T = f(t_1, \ldots, t_n)$ ein Term. Die Terme $t_1, \ldots, t_n$ sind dann* echte Unterterme *des Terms t. Unterterme von t_i mit $i \in \{1, \ldots, n\}$ sind ebenfalls echte Unterterme von t.*

Die Menge der Variablen in einem Term t liefert die Funktion $FV(t)$.

Der Begriff *Grundterm*[1] bezeichnet einen Term, der komplett aus Terminalen besteht. Ein Term t ist also genau dann ein Grundterm, wenn $FV(t) = \emptyset$ gilt [SSSA11].

Definition 2 (Größe eines Terms) *Die Größe eines Terms entspricht der Anzahl aller Funktionssymbole im Term und ist durch die size-Funktion wie folgt definiert:*

- *für $a \in \Sigma$ mit $ar(a) = 0$: $size(a) = 1$*
- *für $x \in \mathcal{V}$: $size(x) = 1$*
- *für $f \in \Sigma$ mit $ar(f) = n$: $size(f(t_1, \ldots, t_n)) = 1 + \sum_{i=1}^{n} size(t_i)$*

2.1.2 Position

Die Unterterme eines Terms haben eine eindeutige *Position* p innerhalb eines Terms, deren wichtigsten Eigenschaften nach [BN99] wie folgt definiert sind:

Definition 3 (Position) *Sei $t \in Term(\Sigma, \mathcal{V})$.*

- *Die Menge aller Positionen des Terms t ist mit $P = Pos(t)$ bezeichnet.*
- *Der Unterterm des Terms t an der Position $p \in Pos(t)$ wird mit t/p dargestellt.*
- *Die Position p wird durch eine Folge von nicht-negativen Ganzzahlen dargestellt und lässt sich über die Struktur des Terms t eindeutig zuordnen. Eine leere Folge ϵ bezeichnet den kompletten Term.*

Positionen sind für Kontexte und Ersetzungen wichtig.

2.1.3 Kontext

Ein *Kontext* c ist ein Term mit Löchern, die als $[\cdot]$ notiert sind und weder Teil des Alphabets noch der Variablen sind. In ein Loch kann ein Ausdruck (Term oder

[1] engl. *ground term*

Kontext) eingesetzt werden. Setzt man in alle Löcher eines Kontextes einen Term ein, so entsteht ein Term. Ersetzt man die Löcher mit Kontexten, so erhält man wieder einen Kontext. Formaler ist ein Kontext nach [BN99] definiert:

Definition 4 (Kontext) *Sei* $[\cdot] \notin \Sigma \cup \mathcal{V}$ *ein neues Symbol, das wir Loch nennen. Ein Kontext* c *ist dann ein Term* $t \in \mathrm{Term}(\Sigma, \mathcal{V} \cup [\cdot])$ *mit den Löchern* $[\cdot]$*. Jedes Loch hat eine eindeutige Position* p *in einem Kontext* c*.*

2.1.4 Ersetzung und Substitution

Die Ersetzung eines Unterterms ist über die Position definiert:

Definition 5 (Termersetzung) *Seien* s *und* t *Terme und* $p \in \mathrm{Pos}(t)$*. Dann ist* $t[s/p]$ *der Term, der aus* t *entsteht, wenn man den Unterterm an der Position* p *durch* s *ersetzt.*

Eine Substitution ist eine allgemine Form der Ersetzung. Es wird nicht an einer bestimmten Position, sondern an allen Positionen, an denen eine Variable steht, ersetzt.

Definition 6 (Substitution) *Eine Substitution* σ *ist eine Funktion* $\sigma : \mathcal{V} \to Term(\Sigma, \mathcal{V})$ *mit* $\sigma(x) \neq x$ *für endlich viele* $x \in \mathcal{V}$*.*

Ist $DOM(\sigma) = \{x_1, \ldots, x_n\}$ *und* $\forall i \in \{1, \ldots, n\} : \sigma(x_i) = t_i$*, so repräsentieren wir* σ *als* $\{x_1 \to t_1, \ldots, x_n \to t_n\}$*. Die Funktion* σ *setzt man auf den ganzen* $Term(\Sigma, \mathcal{V})$ *fort durch*

$$\sigma(f(t_1, \ldots, t_n)) := f(\sigma(t_1), \ldots, \sigma(t_n)).$$

2.1.5 Beispiel

Beispiel 1 *Seien*

- $\Sigma = \{a, b, c, f\}$
- $\mathcal{V} = \{t, u, v, w, x, y, z\}$
- $ar(f) = 3$*,* $ar(a,\ b,\ c) = 0$
- $p_0 = \epsilon$ *eine Position*
- $p_1 = (2)$ *eine Position*

- $p_2 = (2, 2)$ *eine Position*

dann ist beispielsweise

- $t = f(a, f(a, b, c), c)$ *ein Term,*
- $u = f(a, b, c)$ *ein Term an der Position* p_1 *in* t*,*
- $v = b$ *ein Term an der Position* p_2 *in* t*,*
- $w = u/p_0 = u$
- $x = f(a, [\cdot], c)$ *ist ein Kontext,*
- $y = x[b/p_1] = f(a, b, c)$ *ist ein Term und*
- $z = x[x/p_2] = f(a, f(a, [\cdot], c), c)$ *ist ein Kontext.*

2.2 Termersetzungssysteme

2.2.1 Definition

In einem Termersetzungssystem sind Ersetzungen in Form von Regeln definiert, die auf Terme angewendet werden können und diese sukzessive reduzieren. Ein Termersetzungssystem kurz *TRS*[2] ist nach [SS10] wie folgt definiert:

Definition 7 (Termersetzungssystem) *Für ein Regelsystem gelten folgenden Vereinbarungen:*

1. *Eine* Regel *ist ein Paar* (l, r) *von Termen und* $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$*. Eine Regel schreiben wir als* $l \to r$*.*
2. *Ein* Regelsystem R *ist eine (endliche) Menge von Regeln.*
3. *Ein Regelsystem definiert die Reduktionsrelation* $\to_R$ *auf den Termen* $\mathrm{Term}(\Sigma, \mathcal{V})$ *durch*
   $t_1 \to_R t_2$ *gdw. es gibt* $(l \to r) \in R$*, eine Position* p *in* t_1*, eine Substitution* σ *mit* $\sigma(l) = t_1/p$ *und* $t_2 = t_1[\sigma(r)/p]$.
   ($\mathrm{Term}(\Sigma, \mathcal{V})$*,* $\to_R$*) ist das durch* R *definierte* Termersetzungssystem *(Reduktionssystem).*
4. *Das Regelsystem* R *ist terminierend (konfluent, konvergent), wenn (*$\mathrm{Term}(\Sigma, \mathcal{V})$*,* $\to_R$*) die entsprechende Eigenschaft hat.*

[2] engl. Term-Rewriting-System

2.2.2 Eigenschaften

Im vierten Punkt sind die wesentlichen Eigenschaften eines TRS verwendet, aber an dieser Stelle nicht näher definiert. Die Begriffe sind aber im Kontext der Arbeit wichtig und werden daher im Folgenden erklärt.

Definition 8 (Normalform) *Für ein Termersetzungssystem T, Terme t, u, v mit $u \neq v$ gilt:*

- *t ist* reduzibel*, falls es ein u gibt mit $t \to_T u$, sonst ist t* irreduzibel*.*
- *v ist die* (T-)Normalform *zu t, falls $t \overset{*}{\to}_T v$ und v irreduzibel.*

Definition 9 (Konfluenz eines TRS) *Ein TRS T ist genau dann* konfluent*, wenn gilt: aus $t \overset{*}{\to}_T u$ und $t \overset{*}{\to}_T v$ folgt $\exists z : u \overset{*}{\to}_T z \overset{*}{\leftarrow}_T v$.*

Vereinfacht gesagt heißt *konfluent*, dass die Reihenfolge der Termersetzungen keine Rolle für das Ergebnis spielt.

Satz 1 (Reduziertes TRS) *Ein TRS T ist genau dann* reduziert*, wenn*

- *jede linke Seite l_i irreduzibel für das System $T \setminus \{l_i \to r_i\}$ und*
- *jede rechte Seite r_i eine T-Normalform ist.*

Satz 2 (Konvergenz eines TRS) *Ein TRS ist genau dann* konvergent*, wenn zu jedem Term eine eindeutige Normalform existiert.*

Definition 10 (Terminierendes TRS) *Ein TRS T ist genau dann* terminierend*, wenn es keine unendliche Reduktionssequenz $\overset{*}{\to}_T$ erzeugen kann.*

Definition 11 (Kanonisches TRS) *Ein TRS T ist genau dann* kanonisch*, wenn es* konfluent *und* terminierend *ist.*

Definition 12 (Ground term rewriting system) *Eine Sonderform des TRS ist das* gTRS *(ground term rewriting system) in dem alle Terme Grundterme sind. Auf beiden Seiten aller Regeln kommen keine Variablen vor. Jedes reduzierte gTRS ist auch kanonisch [SSSA11].*

2.2.3 Beispiel

Im folgenden Beispiel wird die Arbeitsweise mit einem TRS verdeutlicht:

Beispiel 2 *Sei T ein konfluentes TRS mit den Ersetzungsregeln R:*

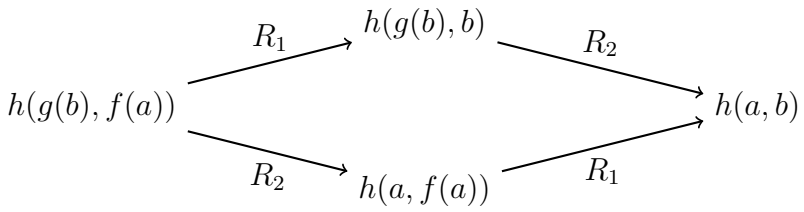
$$R = \left\{ \begin{array}{lcll} f(a) & \rightarrow & b & (\textit{Regel } R_1) \\ g(b) & \rightarrow & a & (\textit{Regel } R_2) \end{array} \right\}.$$

Der Term $t_1 = f(g(f(a)))$ wird durch die Regeln aus T schrittweise ersetzt:

$$t_1 = f(g(f(a))) \xrightarrow{R_1} f(g(b)) \xrightarrow{R_2} f(a) \xrightarrow{R_1} b$$

Auf den Term b ist keine der Regeln aus T mehr anwendbar.

Die Reihenfolge der Anwendung der Regeln um den Term $t_2 = h(g(b), f(a))$ zu ersetzen, ist nicht eindeutig:



2.3 Singleton-Tree-Grammar

2.3.1 Definition

Eine Singleton-Tree-Grammar (kurz: *STG*) ist eine besondere Form der Baumgrammatiken, die vom Typ 3 der Chomsky-Hierarchie sind.

Formal ist eine STG nach [SSSA11] wie folgt definiert:

Definition 13 (Struktur einer STG) *Eine STG ist ein 4-Tupel $G = (\mathcal{TN}, \mathcal{CN}, \Sigma, R)$ mit*

- *den Term-Nichtterminalen $\mathcal{TN}$ mit Stelligkeit 0,*
- *der Kontext-Nichtterminalen $\mathcal{CN}$ mit Stelligkeit 1,*

- *der Signatur der Funktionssymbole* Σ *und*
- *der Regelmenge* R*.*

Die Mengen $\mathcal{TN}$*,* $\mathcal{CN}$ *und* Σ *sind paarweise verschieden und die Menge der Nichtterminale* $\mathcal{N}$ *ist definiert als* $\mathcal{N} = \mathcal{TN} \cup \mathcal{CN}$*. Die Regeln in* R *müssen folgende Form haben:*

- $A ::= f(A_1, \ldots, A_m)$ *mit* $A, A_i \in \mathcal{TN}$ *für* $i = 1, \ldots m$ *und* $f \in \Sigma$ *mit* $ar^3(f) = m$*.*
- $A ::= C_1 A_2$ *mit* $A, A_2 \in \mathcal{TN}$ *und* $C_1 \in \mathcal{CN}$*.*
- $A ::= A'$ *mit* $A, A' \in \mathcal{TN}$*.*
- $C ::= [\cdot]$ *mit* $C \in \mathcal{CN}$*.*
- $C ::= C_1 C_2$ *mit* $C, C_1, C_2 \in \mathcal{CN}$*.*
- $C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_m)$ *mit* $C \in \mathcal{CN}$*,* $A_j \in \mathcal{TN}$ *für* $j = 1, \ldots i-1, i+1, \ldots, m$ *und* $f \in \Sigma$ *mit* $ar(f) = m$*.*

Jedes Nichtterminal $X \in \mathcal{N}$ *steht bei genau einer Regel auf der linken Seite. Der transitive Abschluß* $\xrightarrow{+}_G$ *der Relation* $\rightarrow_G$ *von* $\mathcal{N}$ *ist zyklenfrei mit* $X \rightarrow_G Y$ *genau dann, wenn* $X ::= r$ *eine Regel in* G *und* $Y \in \mathcal{N}$ *in* r *vorkommt.*

Eine STG hat immer nur ein Loch in einem Kontext. Anstatt der Ersetzung eines Lochs mit einem Ausdruck spricht man hier auch vom Einsetzen. Die Operation des Einsetzens in einen Kontext wird wie folgt notiert: $c[a]$ bedeutet, dass in den Kontext c der Ausdruck a an der Stelle des Lochs $[\cdot]$ eingesetzt wird.

2.3.2 Term und Kontext

Um aus einer STG einen Term oder einen Kontext zu erzeugen benötigt man eine Funktion, die den Wert von Nichtterminalen berechnet.

Definition 14 (Term und Kontext einer STG) *Der Term (oder Kontext) eines Nichtterminals* N *aus der STG* G *wird mit der Funktion* $val_G(N)$ *(oder auch nur* $val(N)$*, wenn klar ist, welche Grammatik* G *gemeint ist) berechnet. Der Term oder Kontext über* Σ *wird aus* N *erzeugt durch das wiederholte und vollständige Anwenden der Regeln aus* G*:*

[3] ar() = Stelligkeit()

$$
\begin{array}{ll}
val_G(A) = f(val_G(A_1), \ldots, val_G(A_m)), & \textit{wenn } A ::= f(A_1, \ldots, A_m) \\
val_G(A) = val_G(C_1)[val_G(A_2)], & \textit{wenn } A ::= C_1 A_2 \\
val_G(A) = val_G(A'), & \textit{wenn } A ::= A' \\
val_G(C) = [\cdot], & \textit{wenn } C ::= [\cdot] \\
val_G(C) = val_G(C_1)[val_G(C_2)], & \textit{wenn } C ::= C_1 C_2 \\
val_G(C) = f(t_1, \ldots, t_{i-1}, [\cdot], t_{i+1}, \ldots, t_m), & \textit{wenn} \\
 & C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_m) \\
 & \textit{mit } t_j = val_G(A_j) \\
 & \textit{für } j = 1, \ldots, i-1, i+1, \ldots, m
\end{array}
$$

2.3.3 Eigenschaften

Betrachtet man die Komplexität einer STG, so benötigt man die Eigenschaften *Tiefe* und *Größe*, die wie folgt definiert sind:

Definition 15 (Eigenschaften einer STG) *Die Tiefe eines Nichtterminals N ist die maximale Zahl n von allen Sequenzen $N \to_G N_1 \to_G N_2 \ldots \to_G N_n$. Die Tiefe einer STG G ist definiert als das Maximum der Tiefen aller Nichtterminale der Grammatik G.*

Die Größe $|G|$ einer Grammatik G ist die Summe der Größen von allen rechten Seiten der Regeln, wobei das Loch $[\cdot]$ als 1 gezählt wird.

Eine STG wird häufig benutzt, um einen einzelnen Term zu speichern. Der Vorteil in der Benutzung der STG liegt darin, dass die STG im Vergleich zum Ausgangsterm logarithmische Größe haben kann und daher der Term *komprimiert* wird. Im besten Fall kann also ein Term w in einer STG G mit Größe $|G| = n$ gespeichert werden, während der originale Term die Größe $|w| = 2^n$ hat.

2.3.4 Beispiel

Um den Formalismus einer STG besser zu verstehen, folgt jetzt ein Beispiel:

Beispiel 3 *Gegeben sei eine STG G, die wie folgt definiert ist:*
$G_1 = (\mathcal{TN}, \mathcal{CN}, \Sigma, R)$ *mit*

- $\mathcal{TN} = \{A_1, A_2\}$,

- $\mathcal{CN} = \{C_1, C_2, C_3\}$,
- $\Sigma = \{a, f_2, f_3\}$

und den Produktionsregeln:

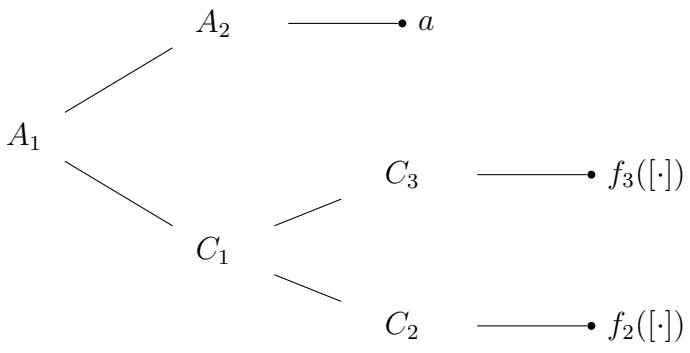
$$R = \left\{ \begin{array}{lcl} A_1 & \rightarrow & C_1[A_2] \\ C_1 & \rightarrow & C_2[C_3] \\ C_2 & \rightarrow & f_2[\cdot] \\ C_3 & \rightarrow & f_3[\cdot] \\ A_2 & \rightarrow & a \end{array} \right\}.$$

Die Terme und Kontexte, die in der STG gespeichert sind, lassen sich nun mit der oben beschriebenen val-Funktion bestimmen - hier für Kontext C_1 und Term A_1:

$$\begin{array}{rcl} val(C_1) \rightarrow & val(C_2)[val(C_3)] \\ & \rightarrow & val(C_2)[f_3([\cdot])] \rightarrow f_2([\cdot])[f_3([\cdot])] \\ & \rightarrow & f_2(f_3([\cdot])) \end{array}$$

$$\begin{array}{rcl} val(A_1) \rightarrow & val(C_1)[val(A_2)] \rightarrow val(C_1)[a] \rightarrow val(C_2)[val(C_3)][a] \\ & \rightarrow & val(C_2)[f_3([\cdot])][a] \rightarrow f_2([\cdot])[f_3([\cdot])][a] \\ & \rightarrow & f_2(f_3([\cdot]))[a] \\ & \rightarrow & f_2(f_3(a)) \end{array}$$

Man kann die Ersetzung, die den Term A_1 berechnet auch mit dem folgendem Baum darstellen:



2.4 Haskell

Haskell ist eine funktionale Programmiersprache. Im Gegensatz zu logischen oder prozeduralen Programmiersprachen besteht ein Haskellprogramm aus einer Menge von Funktionsdefinitionen [SS11a]. Das Ausführen eines Programmes entspricht dem Auswerten einer Funktion mit einer konkreten Parametermenge.

Als pure funktionale Programmiersprache ist Haskell *seiteneffektfrei*, d.h. dass der Wert einer Funktion nur abhängig von den Parametern ist. Die Auswertung einer Funktion erfolgt in Haskell *verzögert*. Dieses Verhalten, das im englischen als *lazy evaluation* bekannt ist, bedeutet, dass der Wert einer Funktion erst dann berechnet wird, wenn er benutzt wird.

2.4.1 Typsystem

Haskell verwendet eine starkes, statisches und polymorphes Typsystem. Dies bedeutet im einzelnen:

- *stark*

  Zu jedem Ausdruck kann der Compiler einen eindeutigen Typ herleiten. Der Compiler akzeptiert nur Programme, die korrekt typisiert sind.

- *statisch*

  Ein Haskell Ausdruck kann auch statisch typisiert sein. Der Typ ist dann zur Compilezeit fest vorgegeben.

- *polymorph*

  Das polymorphe Typsystem von Haskell ist mittels Typvariablen umgesetzt. Dies bedeutet, dass nicht ein konkreter Typ sondern ein abstrakter Typ mit bestimmten Eigenschaften definiert werden kann. Die Typisierung ist genau dann korrekt, wenn die Verwendung der entsprechend getypten Ausdrücke durch die Eigenschaften sichergestellt ist.

In Haskell ist es möglich eigene Datentypen zu definieren, wie in Quellcode 2.1 aus [Sab11] gezeigt. In dem definierten Datentyp **`NonTerminal`** ist der Name **`a`** polymorph entweder vom Typ **`TermNT`** oder **`CtxtNT`** - also entweder ein Term-Nichtterminal oder ein Kontext-Nichtterminal.

Quellcode 2.1: Datentyp **NonTerminal**

```
data NonTerminal a = TermNT a | CtxtNT a
  deriving(Eq,Read)
```

Parameterlose Datentypen können wie ein Aufzählungstyp aus imperativen Programmiersprachen aufgefasst werden.

2.4.2 Sprachkonstrukte

Im folgenden werden zwei wesentliche Elemente der Sprache Haskell vorgestellt, die innerhalb der Arbeit häufig verwendet werden.

Pattern Matching

Durch *Pattern Matching* kann man eine Funktion mit verschiedenen Parametermustern definieren und der Compiler wählt dann den passenden Funktionsaufruf aus. Die Struktur der Muster muss dabei gleich sein, die Anzahl und Typen der Parameter darf sich nicht unterscheiden.

Quellcode 2.2: Pattern Matching

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

In Quellcode 2.2 wird die Summe der Elemente einer Zahlenliste rekursiv berechnet. Eine leere Liste hat die Summe 0, ansonsten berechnet sich der Wert aus dem vordersten Element x plus der Summe der Restliste xs.

Data.Map

Wichtiger Bestandteil der Haskell Distribution GHC [ghc] ist das **Data.Map** Modul. Die assoziative Datenstruktur **Map** ermöglicht es zu je einem Schlüssel einen Wert zu speichern. Der Zugriff erfolgt nicht wie in einer Liste oder Array per Index, sondern mittels Schlüssel, dessen Datentyp festgelegt werden kann. Eine **Map**-Instanz kann aus einer Liste von Tupeln der Form **[(key1, value1), ..., (keyn, valuen)]**

erzeugt und in eine solche Liste umgewandelt werden. Funktionen erlauben das Auslesen, Aktualisieren, Hinzufügen und Löschen von Schlüssel-Wert-Paaren. Auch das Erzeugen einer Liste aller Schlüssel ist direkt möglich. Viele Operationen (darunter **`insert`**, **`delete`**, **`lookup`**, etc.) werden bei einer Map mit n Elementen in Zeit $\mathcal{O}(log\ n)$ durchgeführt.

2.5 Parser

Nach [LSA08] ist ein Parser ein Programm, dass eine Eingabe in Wörter aufteilt, die ein anderes Programm verstehen und verarbeiten kann. Die Eingabe wird dabei syntaktisch auf Basis einer definierten Sprache analysiert. Der Vorgang des *Parsens* besteht aus zwei Teilen:

1. lexikalische Analyse und
2. syntaktische Analyse.

In den meisten Fällen ist die Sprache, die ein Parser analysiert, durch eine kontextfreie Grammatik definiert. Für diese Sprachklasse ist das Wortproblem mit dem Cocke-Younger-Kasami-Algorithmus in kubischer Zeit lösbar, während es für kontextsensitive Sprachen nur in exponentieller Zeit entschieden werden kann [HU00].

Parser sind feste Bestandteile von Compilern und Interpretern, um ein eingegebenes Programm in eine für den Computer weiterverarbeitbare Form umzuwandeln.

Lexikalische Analyse

Die Aufgabe der lexikalischen Analyse ist die Überführung der Eingabe in *Token*. Die Eingabe wird in Worte (Lexeme) aufgeteilt und je nach Typ durch ein entsprechendes Token repräsentiert. Die Worte werden üblicherweise mittels eines regulären Ausdrucks definiert, so dass ein endlicher Automat das passende Token für jedes Wort bestimmen kann.

Syntaktische Analyse

In der syntaktischen Analyse wird überprüft, ob es sich bei einer Tokenfolge um ein gültiges Wort der Sprache handelt. Da die Sprache durch eine kontextfreie Grammatik dargestellt ist, wird ein Kellerautomat genutzt um korrekte Worte zu erkennen.

Kapitel 3

Aktuelle Verfahren

Dieses Kapitel beschreibt aktuelle Verfahren, die in der Arbeit verwendet werden. Zunächst wird ein Cabal-Haskell-Package *grammar-based-compression* (GBC) vorgestellt, auf dem die Implementierung dieser Arbeit basieren soll. Anschließend werden zwei Verfahren näher erörtert, die auf STG-komprimierten Termen die Äquivalenz und Termersetzung lösen.

3.1 Grammar based compression

Von der Professur *Künstliche Intelligenz und Softwaretechnologie* der Goethe-Universität Frankfurt am Main stammt das Cabal-Haskell-Package *grammar-based-compression* [Sab11]. In diesem Package befinden sich verschiedene Algorithmen, die zum grammatikbasierten Komprimieren von Strings und Termen eingesetzt werden können.

Strings werden mit SCFGs[1] komprimiert und repräsentiert. SCFGs sind eingeschränkte kontextfreie Grammatiken, wobei jedes Nichtterminal genau ein Wort erzeugt und die Grammatik in Chomsky-Normalform vorliegen muss.

Für Terme, Bäume und Objekte, die eine Baumstruktur besitzen, wird eine im Grundlagen-Kapitel besprochene STG zum Komprimieren verwendet. Für beide Grammatiken sind verschiedene Algorithmen implementiert, von denen zwei im weiteren Verlauf besonders relevant sind:

- Der Algorithmus von *Plandowski* berechnet die Gleichheit von zwei (Teil-)Ausdrücken und wird im Unterkapitel 3.2 genauer vorgestellt.

[1] singleton context free grammars

- Eine topologische Sortierung, die die Produktionen nach ihrer gegenseitigen Abhängigkeit sortiert. Diese ist stets möglich, da sowohl SCFGs als auch STGs kreisfrei sein müssen.

3.1.1 STG in Haskell

Eine STG ist in Haskell mit einer **`Data.Map`** umgesetzt:

```
type STG sigma nt = Map.Map nt (STGProd sigma (NonTerminal nt))
data NonTerminal a = TermNT a | CtxtNT a
data STGProd sigma nt = AfA nt sigma [nt] | CCC nt nt nt | ...
```

In der Map wird jedem Nichtterminal nt (Schlüssel) genau die Regel r zugewiesen (Wert), die nt auf der linken Seite hat, also von der Form $nt \to \ldots$ ist. So kann effizient (in logarithmischer Zeit) auf die Regeln zugegriffen werden. Durch die auf einer Map implementierten Funktionen **`Map.keys`** und **`Map.elems`** kann direkt eine Liste aller Nichtterminale und aller Regeln erzeugt werden. Mit dem **`NonTerminal`** Datentyp wird für ein Nichtterminal abgespeichert, ob es ein Kontext- oder ein Termnichtterminal ist.

Die **`val`**-Funktion berechnet für ein Nichterminal und eine zugehörige Grammatik den Term, der durch das Nichterminal erzeugt wird. Die Berechnung geschieht analog zu der in Unterkapitel 2.3, Definition 14, definierten Methode.

Quellcode 3.1: Beispiel einer STG

```
tSTG = prodListToSTG
       [ ACA (toTermNT 0) (toCtxtNT 3) (toTermNT 1)
       , ACA (toTermNT 1) (toCtxtNT 3) (toTermNT 2)
       , AfA (toTermNT 2) "a" []
       , CfAC (toCtxtNT 3) "f" [] (toCtxtNT 4) []
       , CEmpty (toCtxtNT 4)
       ]
```

Die STG im Quellcodebeispiel 3.1 hat folgende Terme für jedes Nichtterminal gespeichert:

$$0 \mapsto f(f(a)) \qquad 1 \mapsto f(a) \qquad 2 \mapsto a$$

$$3 \mapsto f([\cdot]) \qquad 4 \mapsto [\cdot]$$

3.1.2 TRS in Haskell

Das Termersetzungssystem ist in Haskell als Tupel umgesetzt:

```
type ReducedTRS sigma nt =
        (STG sigma nt
        ,[(NonTerminal nt,NonTerminal nt)])
```

Die erste Komponente ist die STG G, in der alle Terme des TRS komprimiert gespeichert sind. Die Liste von Nichtterminalpaaren an der zweiten Stelle enthält die Regeln des TRS in Tupelform: das jeweils erste Nichtterminal der STG G soll durch das jeweils zweite ersetzt werden: $[(l_1, r_1), \ldots, (l_n, r_n)]$ mit $l_i, r_i \in \mathcal{N}_G$[2].

Quellcode 3.2: Beispiel eines TRS

```
tSTG1 = prodListToSTG
       [ AfA (toTermNT 1) "f" [toTermNT 2]
       , AfA (toTermNT 2) "a" []
       , AfA (toTermNT 3) "b" []
       ]
tTRS1 = (tSTG1, [(toTermNT 1, toTermNT 3)])
```

Im TRS `tTRS1` von Quellcodebeispiel 3.2 ist folgende Regel in komprimierter Form gespeichert:

$$f(a) \to b.$$

3.2 Äquivalenz von STG-komprimierten Termen

Die Aufgabenstellung dieser Arbeit beinhaltet die Implementierung eines TRS für STG-komprimierte Terme in Haskell. Innerhalb der Termersetzung müssen daher

[2] $\mathcal{N}_G$: Nichtterminale der STG G, siehe Definition 13

STG-komprimierte Terme auf Äquivalenz überprüft werden.

Formal ist das Problem das Wortproblem für eine STG G sowie zwei Nichtterminale A_1 und A_2. Das Wortproblem ist dann genau die Antwort auf die Frage, ob $val_G(A_1) = val_G(A_2)$ gilt.

3.2.1 Äquivalenz über den nichtkomprimierten Term

Der naive Ansatz ist es, den *val*-Wert der Terme zu berechnen und anschließend zu vergleichen. Da die Terme jedoch komprimiert vorliegen, ist dieses Vorgehen nicht ohne Nachteil. Die Größe der nichtkomprimierten Terme kann exponentiell groß werden (siehe Beispiel 3.3). Dies ist insbesondere bei großen Grammatiken problematisch.

Quellcode 3.3: STG mit exponentieller Kompression

```
tSTGexp n = prodListToSTG
([ACA (toTermNT 0) (toCtxtNT (2)) (toTermNT 1)
 ,AfA (toTermNT 1) "a" []
 ]
 ++
 [CCC (toCtxtNT i)(toCtxtNT (i+1))(toCtxtNT (i+1)) | i<-[2..n+2]]
 ++
 [CfAC (toCtxtNT (n+3)) "f" [] (toCtxtNT (n+4)) []
 ,CEmpty (toCtxtNT (n+4))
 ])
```

Die von der Funktion `tSTGexp` erzeugte Grammatik enthält $n + 4$ Regeln und hat damit eine Größe von $\mathcal{O}(n)$. Der entkomprimierte Term, der in der Grammatik gespeichert ist, hat die Form $f^{2^n}(a)$. Die Größe des entkomprimierten Terms ist mit $\mathcal{O}(2^n)$ exponentiell.

3.2.2 Äquivalenz mit dem Algorithmus von Plandowski

Der Algorithmus von Plandowski [Pla94] berechnet die Äquivalenz von Termen einer kontextfreien Grammatik ohne den kompletten Term aus der Grammatik zu erzeugen in polynomieller Zeit. Eingabe des Verfahrens ist eine Grammatik mit zwei Startsymbolen, die miteinander verglichen werden.

Im Gegensatz zum naiven Verfahren benutzt Plandowski die Größe der Terme, die durch Nichtterminale erzeugt werden. Unterscheidet sich diese an einer Stelle, so können die erzeugten Terme logischerweise nicht äquivalent zueinander sein. Somit können Teile der Grammatik vorzeitig von der Überprüfung ausgeschlossen werden.

Der eigentliche Algorithmus von Plandowski [Pla94] löst das Wortproblem auf (SCFG)-komprimierten Strings. Dieser Algorithmus kann jedoch auch verwendet werden um das Wortproblem auf STGs (in Polynomialzeit bezüglich der Größe der Grammatik) zu lösen [BLM05]. Anstelle der Terme werden hierfür die Preorder-Durchläufe der Terme verglichen, das heißt die Terme werden in String-Darstellung an den ursprünglichen Plandowski-Algorithmus übergeben.

Da Gleichheit der Preorder-Durchläufe mit der Gleichheit der Terme übereinstimmt, ist dieses Verfahren korrekt. Eine SCFG stellt die Preorder-Durchläufe der Terme einer STG dar. Es ist somit möglich eine STG in Polynomialzeit in eine SCFG umzuwandeln. Daher ist der kombinierte Algorithmus ebenfalls ein Polynomialzeit-Algorithmus.

Plandowski auf STGs in Haskell

Im GBC-Modul ist der Algortihmus von Plandowski nicht für STGs selbst enthalten. STGs werden in Polynomialzeit in eine SCFG umgewandelt, für die der Algorithmus implementiert ist.

3.3 gTRS-Termersetzung in STG-komprimierten Termen

Die Termersetzung in STG-komprimierten Termen ist eine besondere Herausforderung, wenn man das Dekomprimieren aller beteiligten Terme verbietet. Innnerhalb einer Grammatik müssen die Unterterme mit dem zu ersetzenden Term verglichen werden. Dabei kann es vorkommen, dass ein Unterterm (aus der Grammatik heraus) nur ein Kontext ist, in den ein Term eingesetzt werden muss, um einen vergleichbaren Term zu bilden. Der in das Loch eines Kontextes eingesetzte Term ist normalerweise aus einer Regel, die *höher* in der Termstruktur sitzt, als der Kontext. Der Term muss also in der Betrachtung *mitgenommen* werden. Der Fall ist in Beispiel 4 dargestellt.

Beispiel 4 *Gegeben sei eine STG G, die wie folgt definiert ist:*
$G_1 = (\mathcal{TN}, \mathcal{CN}, \Sigma, R)$ *mit*

- $\mathcal{TN} = \{A_1, A_2\}$, $\mathcal{CN} = \{C_1, C_2, C_3\}$, $\Sigma = \{a, f\}$

und den Produktionsregeln:

$$R = \left\{ \begin{array}{lcl} A_1 & \rightarrow & C_1[A_2] \\ C_1 & \rightarrow & C_2[C_3] \\ C_2 & \rightarrow & f[\cdot] \\ C_3 & \rightarrow & f[\cdot] \\ A_2 & \rightarrow & a \end{array} \right\}.$$

$$\begin{array}{rcl} val(A_1) \rightarrow & val(C_1)[val(A_2)] \rightarrow val(C_1)[a] \rightarrow val(C_2)[val(C_3)][a] \\ \rightarrow & val(C_2)[f([\cdot])][a] \rightarrow f([\cdot])[f([\cdot])][a] \\ \rightarrow & f(f([\cdot]))[a] \rightarrow f(f(a)) \end{array}$$

Der Term $f(a)$ wird durch $C_3[A_2]$ erzeugt. Der Kontext C_3 und der eingesetzte Term A_2 stammen aus unterschiedlichen Regeln.

Diese Betrachtung muss an allen in Frage kommenden Positionen durchgeführt werden, bis der Term gefunden wurde oder keine Positionen zum Testen mehr vorhanden sind.

Eine zweite Schwierigkeit ist, dass der einzusetzende Term an der gefundenen Position in die Grammatik integriert wird. Hierbei muss gegebenenfalls eine (oder mehrere) gültige Regeln neu erzeugt werden. Im Beispiel 4 müssen A_2 und C_3 geändert werden. Der Term, der statt $C_3[A_2]$ in den Kontext C_2 eingesetzt wird, muss der ersetzende Term aus dem gTRS sein.

Die Abbildung 3.1 zeigt, wie die Termersetzung für einen Term t aus der Grammatik G mit einem gTRS funktioniert. Die Grammatik G, die den Term t speichert, wird in mehreren Schritten zur Grammatik G_n transformiert. Diese erzeugt direkt die *Normalform* t_n des Terms t in Bezug auf das gTRS.

Ein naives Verfahren würde alle Positionen eines Terms t prüfen. Allerdings hat der Term t unter Umständen exponentiell viele Positionen (bezüglich der Größe der Grammatik) und ein einzelner Termersetzungsschritt würde demnach schon ex-

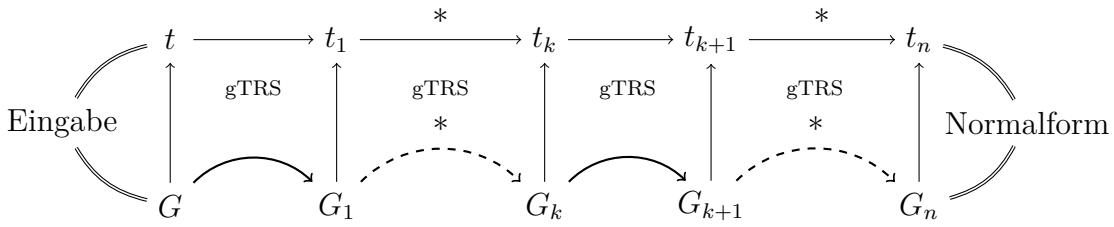


Abbildung 3.1: Termersetzung mit einer Grammatik

ponentielle Laufzeit erfordern. Dieser Ansatz nutzt gemeinsame Verwendung von Nichtterminalen innerhalb der Grammatik nicht aus.

3.3.1 SSSA-PSPACE-Algorithmus

Im Artikel *Congruence closure of compressed terms in polynomial time* [SSSA11] wird ein Algorithmus vorgestellt, der die T-Normalform zu einem gTRS für einen Term in $PSPACE$ berechnet. Der Term selbst und die Terme des gTRS sind mit einer STG komprimiert. Das Verfahren ist algorithmisch aufwändig, arbeitet dafür aber nur auf den komprimierten Termen ohne diese zu dekomprimieren.

Der Algorithmus ist formal wie folgt definiert:

Definition 16 (SSSA-PSPACE-Algorithmus) *Gegeben eine STG G mit Term-Nichtterminal A, das einen Term t generiert, und ein komprimiertes gTRS T mit Regeln $\{l_i \to r_i\}$, wobei l_i, r_i Nichtterminale einer STG G_{TRS} sind.*

Der Algorithmus berechnet G_n mit $val_{G_n}(A) = t_n$, wobei $val_G(A) = t \xrightarrow{*}_T t_n$ eine T-Normalform und irreduzibel in Bezug zu T ist. Der Algorithmus hat PSPACE-Komplexität zur Größe der Grammatiken: $|G| + |G_{TRS}|$.

Das Verfahren

Die Ersetzung eines Terms t hin zur T-Normalform t_n verläuft schrittweise:

$$t \to t_1 \to \ldots \to t_k \xrightarrow{*} t_n.$$

Jedes t_k der Reduktionssequenz wird komprimiert in der STG G_k gespeichert, die polynomiellen Platz benötigt und direkt aus G hergeleitet werden kann:

1. G_k enthält die Regeln von G_{TRS}, von G sowie zusätzliche Regeln. Die Regeln aus G werden durch eine Ersetzung verändert.

2. Wenn ein Term-Nichtterminal A von G den zu ersetzenden Term $val_{G_{TRS}}(l_i)$ erzeugt, wird die rechte Seite der Regel von A durch r_i ersetzt.

3. Wenn für eine Regel des Typs $A ::= C_1 A_2$ der Term $val_{G_{TRS}}(l_i)$ ein echter Unterterm vom Term $val_G(A)$, nicht aber Unterterm von $val_G(A_2)$ ist, wird die rechte Seite der Regel von A durch $C'[r_i]$ ersetzt. Das Kontext-Nichtterminal C' erzeugt den Prefix vor dem zu ersetzenden Term $val_{G_{TRS}}(l_i)$ im Term $val_G(A)$.

Die Erzeugung von Prefixen eines Kontextes führt maximal zu einer polynomiellen Vergrößerung der Grammatik und kann unabhängig voneinander durchgeführt werden. Daher hat G_k höchstens polynomielle Größe in Vergleich zu G.

Für Schritt 3) müssen die entsprechenden in Frage kommenden Positionen effizient bestimmt werden. Es muss für eine solche Position ein Gleichheitstest durchgeführt werden, der die Frage beantwortet, ob $val_G(A) = val_{G_{TRS}}(l_i)$ gilt. Für den Gleichheitstest kann der an STGs angepasste Algorithmus von Plandowski verwendet werden, der polynomielle Laufzeit in Größe der Grammatik benötigt.

Die Erzeugung kann nicht effizient durchgeführt werden, da die gesamte Reduktionssequenz bekannt sein müsste, um zu wissen, an welcher Stelle ersetzt werden muss.

Die Grammatik G_{k+1} kann aus G_k durch einen parallelen Ersetzungsschritt berechnet werden:

- Direkte Ersetzungen eines Term-Nichtterminals können direkt für die Grammatik G durchgeführt werden, sind also schon ersetzt.

- Ersetzungen in der Regel des Typs $A ::= C_1 A_2$ haben zu einer rechten Seite der Form $C'[r_i]$ geführt. Muss in dieser Regeln mit dem Redex r_j ersetzt werden, kann wieder ein Prefix, C'', bestimmt werden. Somit wird die rechte Seite der Regel von A umgeändert zu $C''[r_j]$.

Somit gilt die polynomielle Platzbeschränkung für die gesamte Ersetzung hin zur T-Normalform.

3.4 Parsergenerator “Happy”

Ein Parsergenerator ist ein Programm, das ein Parserprogramm anhand einer Sprachdefinition erzeugt. Ein bekannter Parsergenerator für Haskell ist *Happy* [Mar01]. *Happy* ist in Haskell geschrieben und erzeugt einen Parser in Form eines Haskellmoduls, welches in Haskellprogrammen beliebig eingebunden werden kann. Der Parser selbst basiert auf einem GLR-Parser[3].

Um einen Parser mit Happy zu erzeugen braucht man zunächst eine kontextfreie Grammatik in Backus-Naur-Form (BNF) [GJ10]. Die verwendeten Token müssen definiert sein und ein entsprechender Lexer muß selbst geschrieben werden. Hierbei können z.B. reguläre Ausdrücke eingesetzt werden.

Jeder Regel der Grammatik kann eine Haskellfunktion zugewiesen werden, die mit den zugehörigen Lexemen als Parameter aufgerufen wird. So kann für ein erkanntes Wort entsprechender Code ausgeführt werden. Falls ein Wort nicht zur Sprache der Grammatik gehört, können nähere Informationen ausgegeben werden, an welcher Stelle der Parser die Eingabe nicht mehr verarbeiten konnte. Damit kann der Nutzer leichter Syntaxfehler seiner Eingabe erkennen.

[3]Generalized Left-to-right Rightmost derivation parser

Kapitel 4

Konzeption

Das Ziel dieses Kapitels ist ein Konzept für die Lösung der Probleme der Aufgabenstellung. Es wird besprochen, welche Eingaben der Parser verarbeiten soll und welche Anforderung an die Implementierung der Termersetzung gestellt werden. Außerdem wird erörtert, nach welchen Testkriterien die Implementierung überprüft werden soll.

4.1 Parser

Zunächst soll ein Parser zum Einlesen einer STG geschrieben werden. Die Eingabe in Klartext soll zu einer STG Instanz des GBC-Moduls geparst werden. Es bietet sich an, für die Implementierung den Parsergenerator *Happy* zu benutzen. Als Haskellprogramm kann das Ergebnis (ein Parser in Haskell) direkt genutzt werden.

Im Beispiel 4.1 ist eine STG angegeben, die so oder in ähnlicher Form vom Parsergenerator verarbeitet werden soll.

Beispiel 4.1: Eingaben des Parsers

```
1->*2,3;        % Regeltyp: ACA
*2->*6,*7;      % Regeltyp: CCC
3->4;           % Regeltyp: Lam
4->f(5);        % Regeltyp: AfA
5->a;           % Regeltyp: AfA
*6->.;          % Regeltyp: CEmpty
*7->g(5,*6,5)   % Regeltyp: CfAC
```

Klar ist, dass es eine eindeutige Unterscheidung in der Eingabe zwischen den Kontext-Nichtterminalen CNT und den Term-Nichtterminalen TNT gibt: ein vorangestellter $*$ markiert CNTs. Dieses Kennzeichen erleichtert die Programmierung des Parsergenerators, wie im Abschnitt 5.2.1 noch besprochen wird.

Die Sprache, die ein mit Happy generierter Parser versteht, ist in Form einer annotierten BNF anzugeben. Dazu muss die zur Eingabe passende BNF entwickelt werden und in ein passendes Happy-Programm integriert werden.

4.2 Termersetzung in STG-komprimierten Termen

Hauptteil der Arbeit ist die Implementierung des SSSA-PSPACE-Algorithmus [SS11b] in Haskell auf Basis des GBC-Moduls. Bei der Umsetzung sollen verschiedene Aspekte beachtet werden:

- Polynomieller Platzbedarf
- Optimierte Laufzeit
- Optimierte Ergebnisgrammatik

Die konzeptuelle Auswirkung der einzelnen Punkte wird im folgenden besprochen.

Polynomieller Platzbedarf

Wie in 3.3.1 erörtert, ergibt sich der polynomielle Platzbedarf durch die Ersetzung innerhalb der Grammatik. Das Verfahren vergleicht dabei sukzessive die erzeugten Teilwörter mit dem Term der linken Seite einer Termersetzungsregel. Dafür benutzt er den Algorithmus von Plandowski, von dem bewiesen ist, dass er einen polynomiellen Platzbedarf hat.

Konnte das Nichtterminal in der STG gefunden werden, das den zu ersetzende Term erzeugt, muss die Grammatik umgeschrieben werden. Hier wird durch eine Fallunterscheidung sichergestellt, dass nur der relevante Teil ausgetauscht wird und die Größe nur polynomiell anwächst.

Falls nicht direkt ein Term-Nichtterminal ersetzt werden kann, erzeugt der Algorithmus eine neue Kette von Kontext-Nichtterminalen. Diese generieren den Prefix durch eine neue Menge von Kontext-Nichtterminalen. So wird sichergestellt, dass

doppelt verwendete Kontext-Nichtterminale noch funktionieren. Wird das Kontext-Nichtterminal neben der Erzeugung des Prefix auch noch für die Erzeugung eines anderen Unterterms eingesetzt, würde die Veränderung der Regel auch den anderen Term verändern. Der Ersetzungsschritt ist dadurch ein paralleler Reduktionsschritt, wie im Algorithmus [SSSA11] gefordert.

Optimierte Laufzeit

Die Laufzeit des Verfahrens kann durch Minimierung der Aufrufe von Plandowskis Algorithmus verbessert werden. Ein solcher Aufruf macht nur dann Sinn, wenn die zu vergleichenden Nichtterminale Terme gleicher Länge erzeugen. Berechnet man die Länge der Terme, die jedes Nichtterminal erzeugt, im Voraus und speichert sie in einer Lookup-Map ab, kann der Algorithmus deutlich beschleunigt werden. Je nach Längenvergleich kann der Algorithmus am aktuellen Zweig abgebrochen werden (wenn der zu ersetzende Term länger als der aktuell generierte Term ist) oder nur bei gleichen Wortlängen auf Gleichheit überprüft werden.

Optimierte Ergebnisgrammatik

Die vom Programm erzeugte Grammatik kann aus verschiedenen Gründen “unsauber” beziehungsweise unoptimiert sein. Durch das Ersetzen bleiben zunächst die alten Regeln in der STG enthalten, werden aber eventuell nicht mehr erreicht. Ein Kontext-Nichtterminal, das in keiner Produktionskette eines Term-Nichtterminals vorkommt, kann bedenkenlos entfernt werden. In einem letzten Optimierungsschritt werden solche verzichtbaren CNTs identifiziert und entfernt.

In einer STG wird nur ein einziges CNT benötigt, das ein Loch erzeugt. Falls mehrere solcher Regeln in einer STG existieren, können alle bis auf eine gelöscht und durch das verbleibende ersetzt werden.

4.3 Tests

Nach Abschluss der Implementierung muss das Programm intensiv getestet werden. Hier lassen sich mehrere Testkriterien aufstellen:

1. Funktioniert das Verfahren?

2. Wie schnell läuft das Verfahren mit/ohne Optimierung?
3. Welchen Platz braucht das Verfahren?
4. Wie groß ist die erzeugte und optimierte Grammatik im Vergleich zu einer manuell optimierten?

Um diese Fälle betrachten zu können müssen verschiedene Testfälle konzipiert und anschließend ausgewertet werden.

Kapitel 5

Umsetzung

Details der Implementierung werden in diesem Kapitel erörtert. Zunächst wird der Parser vorgestellt. Die BNF der Eingabe, Grundlage der Implementierung, wird besprochen und die Umsetzung wird exemplarisch gezeigt.

Den zweiten Teil bilden die verschiedenen Schritte der Termersetzung. Dazu zählen Vorverarbeitung, Ersetzung von Term- und Kontext-Nichtterminalen sowie die Optimierung der Ergebnisgrammatik.

Alle hier erwähnten STG-Regeln beziehen sich auf die Definition 13.

5.1 Module

Im Rahmen der Diplomarbeit sind folgende Module entstanden:

- **`ParseSTG`**

  Der in Happy implementierte STG-Parser. Die Hauptfunktion **`parseSTG`** liefert auf Eingabe einer STG als String eine Instanz der STG aus dem GBC-Package.

- **`STGHelper`**

  Im Modul **`STGHelper`** sind verschiedene Hilfsfunktionen auf Basis der STG implementiert, die nicht Teil des GBC-Packages sind, aber die Implementierung vereinfacht haben:

  - **`getRule stg x`** gibt die Regel aus der **`stg`** zurück, die das Nichtterminal **`x`** auf der linken Seite hat.

  - **`appendRule stg`** und **`appendRules stg`** fügen der **`stg`** eine bzw. mehrere Regeln hinzu. Die neu hinzugefügten Regeln überschreiben existierende Regeln mit gleichem Nichtterminal auf der linken Seite.
  - Die Funktion **`getLeftSide prod`** gibt das Nichtterminal auf der linken Seite der Regel **`prod`** zurück.
  - **`getConnectedNTs prod`** erzeugt eine Liste mit allen Kontext-Nichtterminalen der Regel **`prod`**, die auf der rechten Seite vorkommen.

- **`ComboSTGTRS`**

  In der Datenstruktur **`ComboSTGTRS`** sind STG und TRS zusammengefasst. Funktionen aus dem Modul erlauben den Zugriff auf die ursprünglichen Grammatiken und Nichtterminale, die gemeinsam gespeichert sind.

- **`STGLenMap`**

  Die Größen der Terme und Kontexte, die aus Nichtterminalen einer Grammatik erzeugt werden, sind in der **`STGLenMap`** gespeichert. Sie werden mittels topologischer Sortierung berechnet.

- **`TermReplacement`**

  Das Modul **`TermReplacement`** beinhaltet die Termersetzung innerhalb einer STG. Es wird unterschieden, wodurch der zu ersetzende Term erzeugt wird: durch ein Term-Nichtterminal oder durch ein Kontext-Nichtterminal.

- **`CleanUp`** und **`Optimize`**

  Eine STG enthält nach der Ersetzung unter Umständen nicht erreichte Kontext-Nichtterminale, die durch die Funktion **`cleanUp`** aus dem gleichnamigen Modul entfernt werden. Die Funktion **`optimize`** entschlankt die Grammatik durch verschiedene einfache Optimierungen.

- **`TestCases`**

  Die Testfälle sind im Modul **`TestCases`** gespeichert.

5.2 Happy

Der erste Schritt in der Erstellung eines Happy-Programms ist das Aufstellen einer BNF, die der gewünschten Eingabesyntax entspricht. Aus dieser können dann To-

kens und der Lexer entsprechend abgeleitet werden, aus denen das Happy-Programm besteht.

5.2.1 BNF

Mit der BNF aus Beispiel 5.1 lässt sich die gewünschte Eingabe, die im Beispiel 4.1 dargestellt ist, erzeugen und parsen.

Beispiel 5.1: BNF der STG

```
<syntax>     ::= <rule> | <rule> ";" <syntax>
<rule>       ::= <nonterm> "->" <parameters>
<nonterm>    ::= <C> | <A>
<parameters> ::= <f> | <f> "(" <Alist> ")" | <C> "," <A> | <A> |
                 <empty> | <C> "," <C> | <f> "(" <ACAlist> ")"

<ACAlist>    ::= <C> | <Alist> "," <C> "," <Alist> |
                 <Alist> "," <C> | <C> "," <Alist>
<Alist>      ::= <A> | <A> "," <Alist> | <epsilon>

<C>          ::= "*" int
<A>          ::= int
<f>          ::= string
<empty>      ::= "."
<epsilon>    ::= "_" | ""
```

Erwähnenswert ist die Unterscheidung zwischen CNTs und TNTs durch das vorangestellte $*$ bei CNTs. Beides sind Nichtterminale der Grammatik und kommen beide auf linker und rechter Seite der Regeln vor. Wichtig ist die Unterscheidung allerdings bei der genauen Auswahl der Regel. Die Regeln $A ::= C_1 A_2$ und $C ::= C_1 C_2$ unterscheiden sich nur durch die Art des Nicht-Terminals und sind sonst strukturell gleich. Ohne angegebene Unterscheidung müsste man bei Unklarheiten erst die Art der anderen Nichtterminale einer Regel bestimmen. Da aber die explizite Kennzeichnung im Rahmen der Aufgabenstellung zulässig ist, wurde auf eine solche Erkennung verzichtet.

Die Regel $C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_m)$ wurde in einer etwas allgemeineren Form umgesetzt. Anstelle des Lochs $[\cdot]$ erlaubt die BNF ein beliebiges Nichtterminal. Dieses Verhalten entspricht dem des GBC-Moduls. Beide Formen unterscheiden sich nur auf den ersten Blick, sie haben aber die gleiche Ausdrucksstärke:

$$
\begin{aligned}
C ::= & f(A_1, \ldots, A_{i-1}, C_X, A_{i+1}, \ldots, A_m) \\
& \Longleftrightarrow \\
C ::= & C_1 C_X \ \wedge \ C1 ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_m).
\end{aligned}
$$

5.2.2 Token und Regeln

Aus der BNF ergeben sich die benötigten Token, die in Tabelle 5.1 aufgeführt sind. Jedem Token der BNF wird ein korrespondierendes Haskell-Token zugewiesen, das in der Ausgabe des Lexers erzeugt wird. Durch das **`$$`** kann auf ein Element der Eingabe zugegriffen werden, das der Lexer dem Token mitanhängt.

BNF-Token	Haskell-Token
cnt	`TokenCNT $$`
tnt	`TokenTNT $$`
fnc	`TokenFNC $$`
'->'	`TokenTO`
';'	`TokenSC`
','	`TokenCO`
'('	`TokenOB`
')'	`TokenCB`
'.'	`TokenDT`

Tabelle 5.1: Tokenliste des STG-Parsers

In dem BNF-Segment kann jede Regel mit einer Haskell-Funktion versehen werden, die bei Anwendung der Regel ausgeführt wird. Mit den Eingabedaten können die entsprechenden STG-Regeln, Terminale und Nicht-Terminale erzeugt werden. Das Vorgehen ist im Quellcode 5.2 exemplarisch für einen Teil der STG dargestellt.

Quellcode 5.2: BNF-Regeln und Haskell-Funktionen

```
Exp  :: {[STGProd String (NonTerminal Integer)]}
Exp  : Rule              {[$1]}    -- Eine Liste mit einer Regel
     | Exp ';' Rule      {$3 : $1} -- Kopf: Regel, Rumpf: Regeln

Rule :: {STGProd String (NonTerminal Integer)}
Rule : C '->' C ',' C    {CCC $1 $3 $5} -- neue Regel CCC

C    :: {NonTerminal Integer}
C    : cnt               {toCtxtNT $1}  -- neues CNT aus Eingabe
```

Startpunkt des Parsers ist das Symbol **`Exp`**, ein Ausdruck (Expression), die in Haskell vom Typ *Liste von STG-Produktionen* ist. Mit dem **`$i`** kann auf den i-ten Teil der geparsten Seite von Haskell aus zugegriffen werden. Dies wird in der *Rule*-Regel deutlich: die Symbole **`'->'`** und **`','`** sind für die eigentliche Verarbeitung irrelevant, weswegen nur das erste, dritte und fünfte Element des Parseergebnisses benutzt wird.

5.2.3 Lexer

Der Lexer hat die Aufgabe, die Eingabe in Token zu zerlegen und wird als Haskell-Funktion implementiert. Die Eingabe ist ein String und kann als Liste des Typs **`Char`** gut in Haskell mittels *Pattern Matching* verarbeitet werden.

Quellcode 5.3: Lexer für ein BNF-Beispiel

```
lexer :: String -> [Token]
lexer []             = []
lexer (' ':cs)       = lexer cs
lexer ('*':cs)       = lexCNT cs
lexer ('-':('>':cs)) = TokenTO : lexer cs
lexer (';':cs)       = TokenSC : lexer cs
lexer (',':cs)       = TokenCO : lexer cs
lexer (_:cs)         = lexer cs

lexCNT cs = TokenCNT (read num) : lexer rest
       where (num,rest) = span isDigit cs
```

Der Quellcode 5.3 kann eine Eingabe verarbeiten, deren Syntax in BNF im Beispiel 5.1 definiert ist. Mittels *Pattern Matching* wird je nach Eingabesymbol das passende Token erzeugt. Im Falle eines CNTs muss noch der Name in Form einer Zahl ausgelesen und zugewiesen werden. Der verwendete Datentyp **`Token`** muss im Haskell-Teil des Happy-Programms auch definiert werden:

```
data Token
     = TokenCNT Integer
     | TokenTO
     | TokenSC
     | TokenCO
```

5.2.4 Ausführung

Der letzte Schritt in der Erstellung des Happy-Programms ist die Implementierung einer Funktion, die Parser und Lexer aufruft. Eine solche Funktion vereinfacht das Arbeiten mit dem generierten Parser, da selten beide Teile getrennt genutzt werden. Die Ausgabe des Parsers kann an dieser Stelle auch direkt vom Typ `[`**`STGProd String (NonTerminal Integer)`**`]` mit der Funktion **`prodListToSTG`** in eine STG-Instanz umgewandelt werden:

```
parse    :: String -> [STGProd String (NonTerminal Integer)]
parse    =  parser . lexer

parseSTG :: String -> STG String Integer
parseSTG =  prodListToSTG . parse
```

Anschließend kann aus dem Programm mit `Happy` ein Parser generiert werden, der in Haskell verwendet werden kann:

Beispiel 5.4: Aufruf von Happy und Parsertest

```
$ happy STGParser.y
$ ghci STGParser.hs
[...]
*STGParser> let txt = "*1->f(*3);2->a;*3->.;4->*1,2"
```

```
*STGParser> parse txt
[4 ::= 1[2],3 ::= [.],2 ::= a,1 ::= f(3)]
*STGParser> parseSTG txt
fromList[(1,1 ::= f(3)),(2,2 ::= a),(3,3 ::= [.]),(4,4 ::= 1[2])]
*STGParser> val (parseSTG txt) 4
f(a)
```

5.3 Termersetzung

Die Implementierung des SSSA-PSPACE-Algorithmus in Haskell besteht aus drei wesentlichen Schritten:

1. Überprüfung der TNT-Regeln
2. Überprüfung der CNT-Regeln
3. Optimierung der Grammatik

Zunächst wird der zu ersetzende Term, die linke Seite einer gTRS-Regel, in der STG gesucht und entsprechend ausgetauscht. Dies wird zuerst für TNT-Regeln, dann für CNT-Regeln durchgeführt. Im dritten Schritt wird die Grammatik optimiert und nicht erreichte CNT-Regeln gelöscht. Die Optimierung sorgt dafür, dass die Grammatik nicht unnötig groß ist und die nächsten Schritte somit schneller durchgeführt werden können.

Diese drei Schritte werden für eine Regel des gTRS so lange wiederholt, bis die Grammatik nicht verändert wird, der zu ersetzende Term also nicht Teil des in der Grammatik gespeicherten Terms ist. Diese Ersetzung wird für alle Regeln des Grundtermersetzungssystemes wiederholt durchgeführt, bis sich die STG nicht mehr ändert.

Am Ende erhält man als Ausgabe eine STG, in der alle Regeln des gTRS angewendet wurden und die keine nicht erreichten CNT-Regeln enthält. Das gesamte Verfahren ist in Abbildung 5.1 dargestellt.

Schritt 1) und 2) sind im Pseudocode 5.5 detailierter beschrieben. Die genaue Vorstellung und Implementierung ist in Abschnitt 5.5 und Abschnitt 5.6 zu finden.

Beispiel 5.5: Pseudocode des Ersetzungsalgorithmus

```
Eingabe: Li -> Ri

TNT-Regeln:
- A ::= f(A1, ..., An)
  |A| == |Li|? Plandowski(A, Li) == True? => neue Regel A ::= Ri
- A ::= C[A']
  |A| == |Li|? Plandowski(A, Li) == True? => neue Regel A ::= Ri
- A ::= A'
  ignorieren: A' wird auch überprüft.


CNT-Regeln:
Einstiegspunkt:
 A ::= C[A']
  |A| > |Li|? Überprüfe Regel C mit Term A'
  - C ::= f(A1, ..., C', ..., An)
    |C'| + |A'| == |Li|?
       Plandowski(C'[A'], Li) == True?  Pos gefunden
                                 False? abbrechen
    |C'| + |A'| > |Li|?
      Überprüfe Regel C' mit Term A'
    |C'| + |A'| < |Li|?
      abbrechen
  - C ::= C1C2
    |C2| + |A'| == |Li|?
       Plandowski(C2[A'], Li) == True?  Pos gefunden
                                 False? abbrechen
    |C2| + |A'| > |Li|?
      Überprüfe Regel C2 mit Term A''
    |C2| + |A'| < |Li|?
      neue Regel A'' ::= C2[A']
      Überprüfe Regel C1 mit Term A''
  - C ::= []
    ignorieren
```

Da ein reduzierte gTRS kanonisch und somit terminierend ist, wird der Fixpunkt in endlicher Zeit erreicht und das Programm terminiert immer. Ein kanonisches TRS ist auch konfluent und somit ist die Reihenfolge der Ersetzungen für den Ablauf des Verfahrens beliebig.

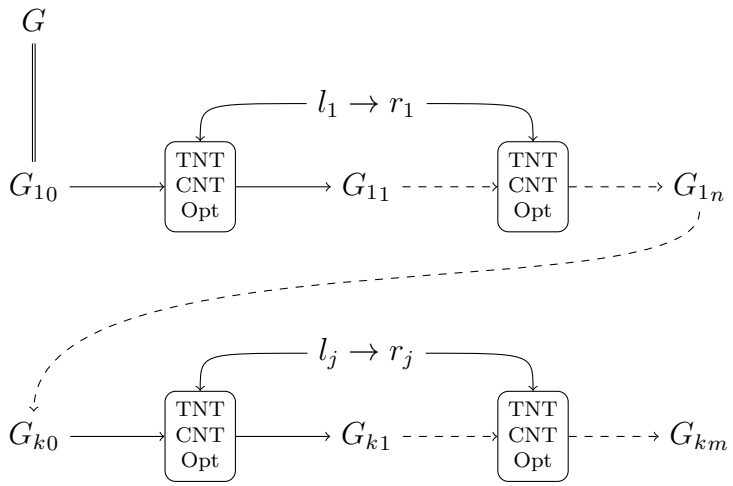


Abbildung 5.1: Termersetzungsalgorithmus im Detail

Im folgenden werden die einzelnen Stufen eines Ersetzungsvorgangs besprochen.

5.4 Vorverarbeitung

Zunächst muß die STG für das weitere Vorgehen vorbereitet werden. Der Algorithmus von Plandowski funktioniert nur inerhalb einer einzelnen Grammatik, daher müssen die Regeln des gTRS mit denen der STG fusioniert werden. Desweiteren kann man die Anzahl der Plandowski-Aufrufe minimieren, in dem man sie nur mit Nichtterminalen ausführt, dessen zugehörige Terme die gleiche Größe haben.

5.4.1 Kombination von Regel und STG

Um den Algorithmus von Plandowski auf Nichtterminale der STG und der linken Seite der gTRS-Regel ausführen zu können, müssen diese zu einer gemeinsamen Grammatik zusammengefügt werden. Allerdings muß man dennoch zwischen Produktionen unterscheiden können, die aus der ursprünglichen STG stammen und solchen, die aus dem gTRS kommen. Zu diesem Zweck wurde die Datenstruktur

ComboSTGTRS erzeugt. Im Quellcode 5.6 ist auch die Funktion zur Erzeugung einer Instanz von **ComboSTGTRS** aus einer **STG** und einem **ReducedTRS** gezeigt.

Quellcode 5.6: Datenstruktur ComboSTGTRS

```
data ComboSTGTRS sigma nt = ComboSTGTRS
        { trs      :: [(NonTerminal nt,NonTerminal nt)]
        , lstNTTRS :: [NonTerminal nt]
        , stg      :: STG sigma nt
        } deriving (Show)

createSTGTRScombo stgIn trsIn =
  let
    n = maxNT stgIn
    (trsNew,changeMap) = renameSTGFrom (n + 1) (fst trsIn)
    stgListTRS = toSTGList (trsNew)
    stgListSTG = toSTGList (stgIn)
    comboSTG = appendRules trsNew stgListSTG
    newTRSreplacer = renameTRSreplacer (snd trsIn) changeMap
  in
    ComboSTGTRS
    { trs = newTRSreplacer
    , lstNTTRS = map getLeftSide stgListTRS
    , stg = comboSTG
    }

getLstNTSTG cmb =
  [nt | nt <- map getLeftSide (toSTGList (stg cmb))
            , notElem nt (lstNTTRS cmb)]
```

In **stg** ist die kombinierte Grammatik gespeichert. Die STG wird original übernommen, damit sie leichter wieder herausgelesen werden kann. Die TRS-Regeln werden hinter den STG-Regeln hinzugefügt. Die Liste mit den Ersetzungen des gTRS muss dementsprechend auch aktualisiert werden.

Durch die Liste aller Nichtterminale des TRS **lstNTTRS** kann auch indirekt auf die Nichtterminale der originalen STG zugegriffen werden. Dies wird in der Funktion **getLstNTSTG** gezeigt. Die Definition der Datenstruktur und Funktionen, die den

Umgang mit ihr erleichtern, sind im Modul **ComboSTGTRS** implementiert.

5.4.2 Größenberechnung der Nichtterminale

Ein Äquivalenztest von zwei Term-Nichtterminalen mit dem Algorithmus von Plandowski kann nur dann erfolgreich sein, wenn die Wörter die gleiche Größe haben. Durch diese Einschränkung können viele Tests von vorneherein verworfen werden. Insbesondere ist der Größenvergleich wichtig, da er auch entscheiden kann, ob noch tiefer in einem Term nach dem zu ersetzenden Term gesucht werden soll.

Der direkte Vergleich der Größe zweier Terme kann insgesamt drei verschiedene Ergebnisse haben. Jedes Ergebnis hat eine relevante Aussage für das Verfahren:

- |Term der STG| = |linke Seite einer gTRS-Regel|

  Ein Vergleich mit Plandowski macht Sinn - fällt er negativ aus, bricht der Algorithmus ab. Ein echter Unterterm wird nur kleiner sein und damit kleiner als der zu ersetzende Term.

- |Term der STG| > |linke Seite einer gTRS-Regel|

  Da der an dieser Stelle erzeugte Term zu groß ist, wird kein Termvergleich durchgeführt. Der Algorithmus bricht aber auch nicht ab, sondern sucht den zu ersetzende Term im Unterterm.

- |Term der STG| < |linke Seite einer gTRS-Regel|

  Der Algorithmus bricht ab und führt keinen Vergleich durch, er würde für den Term der STG und dessen Unterterme negativ ausfallen.

Die Größeninformation ist also wichtig für die Laufzeit des Ersetzungsverfahrens. Sie wird deshalb in einer Datenstruktur abgespeichert, die zusammen mit hilfreichen Funktionen im Modul **STGLengthMap** implementiert ist. Die Typendefinition ist in Quellcode 5.7 dargestellt.

Quellcode 5.7: Definition der STGLengthMap

```
type STGLenMap nt = Map.Map (NonTerminal nt) Integer
```

In einer **Map** wird zu jedem Nichtterminal die Größe des erzeugten Terms gespeichert. Über ein konkretes Nichtterminal als Schlüssel kann man also direkt die Größe auslesen.

Auflösen der Abhängigkeiten von Nichtterminalen mittels Topsort

Die Größenberechnung für jedes Nichtterminal einer STG kann nicht direkt erfolgen. Die Abhängigkeiten von Nichtterminalen untereinander müssen erst vollständig aufgelöst werden. Die Grammatik aus Beispiel 5.8 hat klar erkennbare Abhängigkeiten und eine Sortierung, in der jedes Nichtterminal nur von vor ihm stehedenden abhängt.

Beispiel 5.8: Beispielgrammatik mit Abhängigkeiten

```
1->*2,3;        % Regeltyp: ACA
*2->*6,*7;      % Regeltyp: CCC
3->4;           % Regeltyp: Lam
4->f(5);        % Regeltyp: AfA
5->a;           % Regeltyp: AfA
*6->.;          % Regeltyp: CEmpty
*7->g(5,*6,5)   % Regeltyp: CfAC

Ergebnis von topSort:
[*6, 5, 4, 3, 7, 2, 1]
```

Die Verknüpfungen wurden innerhalb dieser Arbeit mit einer topologischen Sortierung entschlüsselt. Das Ergebnis ist eine Folge von Nichtterminalen, in der das i-te Element nur Elemente vorher referenziert. Das erste Nichtterminal der sortierten Folge erzeugt also entweder ein Terminal oder ein Loch.

Eine STG ist per Definition azyklisch. Dies ist eine Grundvoraussetzung für eine topologische Sortierung. Mit einem Kreis könnte keine eindeutige Reihenfolge definiert werden. Im STG-Modul ist eine topologische Sortierung bereits implementiert, sie hat den Funktionsnamen **`topSort`**.

Berechnung der Größe von sortierten Nichtterminalen

Mit einer topologisch sortierten Nichtterminalliste kann die Größe der erzeugten Terme von Nichtterminalen, mit Hilfe von dynamischer Programmierung, berechnet werden. Die Größe ergibt sich aus der Art der Regel und der Länge der Nichtterminale auf der rechten Seite, wie es in Tabelle 5.2 aufgeführt ist.

Regeltyp	rechte Seite	Länge
AfA	$f()$	1
AfA	$f(a_1, \ldots, a_n)$	$1 + \sum_{i=1}^{n} \lvert a_i \rvert$
ACA	ca	$\lvert c \rvert + \lvert a \rvert$
CEmpty	$[.]$	0
CCC	$c_1 c_2$	$\lvert c_1 \rvert + \lvert c_2 \rvert$
CfAC	$f(a_1, \ldots, a_i, c, a_{i+1}, \ldots, a_n)$	$1 + \lvert c \rvert + \sum_{i=1}^{n} \lvert a_i \rvert$

Tabelle 5.2: Größe des Terms einer STG-Regel

Die Funktion **lenMap** des Moduls **STGLengthMap** speichert die Länge für eine STG-Regel in eine **STGLenMap**. Mittels *Pattern Matching* wird zwischen den verschiedenen Regeltypen unterschieden. Der **lookup** Aufruf, der eigentlich einen **Maybe**-Datentyp zurückliefert, ist an dieser Stelle durch einen eigenen Ausleseaufruf abgekapselt. Der **Maybe**-Datentyp soll einen Auslesefehler abfangen, wenn der auszulesende Schlüssel noch nicht in die **Map** eingetragen wurde. Durch die topologische Sortierung kann dieser Fehler in diesem Anwendungsfall nicht auftreten. In Beispiel 5.9 ist die Längenberechnung für zwei Regeltypen exemplarisch und die Berechnung der gesamten **STGLenMap** aufgeführt.

Die gesamte Berechnung geschieht wie folgt:

1. Sortiere die Produktionsregeln mit **topSort**.
2. Erzeuge leere **Map**.
3. Berechne für die erste Regel der sortierten Liste die Länge und speichere sie in der **Map**. Die Map dem rekursiven Aufruf mit der Restliste der sortierten Produktionsregeln mitgeben. Ist die Liste leer, wird die komplett gefüllte **Map** zurückgegeben.

Schritte 1 und 3 können in Laufzeit $\mathcal{O}(n \cdot log\ n)$ für n = Größe der Grammatik durchgeführt werden. Die leere **Map** wird in konstanter Zeit erzeugt.

Quellcode 5.9: Größenberechnung der STG

```
lenMap :: (Ord nt, Enum nt, Num nt, Show sigma, Ord sigma)
       => STG sigma nt
       -> STGLenMap nt
       -> STGProd sigma (NonTerminal nt)
       -> STGLenMap nt
lenMap stg m (AfA a f [])              = Map.insert a 1 m
lenMap stg m (AfA a f al)              =
    let
        lookup m val = Map.lookup val m
        lst = map (lookup m) al
        len = 1 + sum [getMaybeValue val | val <- lst]
    in
        Map.insert a len m

calcLenMap :: (Ord nt, Enum nt, Num nt, Show sigma, Ord sigma)
           => STG sigma nt
           -> STGLenMap nt
           -> [STGProd sigma (NonTerminal nt)]
           -> STGLenMap nt
calcLenMap stg m [] = m
calcLenMap stg m (x:xs) = calcLenMap stg (lenMap stg m x) xs

createLenMapSTG :: (Ord nt,Enum nt,Num nt,Show sigma,Ord sigma)
                => STG sigma nt
                -> STGLenMap nt
createLenMapSTG stg =
    let
        lstRules = getTopSortedRules stg
        m = Map.empty
    in
        calcLenMap stg m lstRules
```

Aktualisierung der Größen

Es ist leicht zu sehen, dass die Berechnung der **STGLenMap** ein (relativ) zeitintensives Verfahren ist. Leider muss sie nach jedem durchgeführten Ersetzungsschritt komplett neu berechnet werden:

- Ändert man eine Regel, kann dies Auswirkungen auf alle Größen haben. Je mehr Regeln in der topologischen Sortierung hinter dem Nichtterminal auf der linken Seite der geänderten Regel stehen, desto mehr Regeln müssen aktualisiert werden.
- Durch das Ändern der Regel kann aber auch eine vormals korrekte topologische Sortierung inkorrekt werden. Dies ist dann der Fall, wenn Regeln durch die Termersetzung neu hinzugefügt werden oder wenn die Regel teilweise umstrukturiert wurde.
- Nach der Termersetzung wird die Grammatik optimiert. In diesem Schritt werden unter Umständen Regeln und Nichtterminal komplett entfernt. In einem solchen Fall ist die topologische Sortierung auch ungültig.

Es ist zwar vorstellbar, dass ein Algorithmus konstruiert werden kann, der die **STGLenMap** nur aktualisiert und nicht komplett neu aufbaut. Allerdings wird ein solches Verfahren die komplexen Zusammenhänge der Grammatik betrachten müssen und wohl ähnlich aufwändig sein, wie die schlichte Neuberechnung der **STGLenMap**.

Zugriffe aus der Ersetzung

Die **STGLenMap** wird in der Ersetzung als Kriterium gebraucht, ob ein Plandowski-Vergleich sinnvoll ist und ob der Algorithmus in Untertermen weitersuchen soll. Dazu wurden drei Methoden implementiert, die im Quellcode 5.10 aufgeführt sind.

Die Funktionen **isPlandowskiWise***n* führt einen Längenvergleich durch und ist genau dann wahr, wenn das Nichtterminal des Parameters **pNT** einen Term der gleichen Länge erzeugt, wie die Nichtterminale der Parameter **cNT**, beziehungsweise die Summe der Parameter **cNT1** und **cNT2**. Ein Vergleich ist also genau dann sinnvoll, wenn die Funktion *wahr* zurückliefert.

Ob der Algorithmus in Untertermen nach einem Term suchen soll, wird mit der Funktion **shouldIGoFurther** bestimmt. Diese liefert genau dann *Wahr* zurück, wenn das

Nichtterminal des Parameters **pNT** einen Term erzeugt, der kürzer ist, als die Summe der Nichterminal in der Liste **cNTs**.

Quellcode 5.10: Zugriff auf die **STGLenMap** aus der Ersetzung

```
getMapValueX m nt = Data.Maybe.fromMaybe 0 (Map.lookup nt m)

isPlandowskiWise2 m pNT cNT  =
        (getMapValueX m pNT) == (getMapValueX m cNT)

isPlandowskiWise3 m pNT cNT1 cNT2 =
        (getMapValueX m pNT) ==
                ((getMapValueX m cNT1) + (getMapValueX m cNT2))

shouldIGoFurther m pNT cNTs =
        (getMapValueX m pNT) <
                (sum [getMapValueX m nt | nt <- cNTs])
```

Im praktischen Einsatz werden beide Methoden miteinander kombiniert: wenn der Plandowski nicht durchgeführt werden soll, wird überprüft, ob es noch Sinn macht, in Untertermen nach einem Term zu suchen.

5.5 Ersetzung von Term-Nichtterminalen

Eine TNT-Regel hat auf der linken Seite ein Term-Nichtterminal. Es gibt in einer STG genau drei verschiedene Arten von TNT-Regeln:

- $A ::= f(A_1, \ldots, A_m)$ mit $A, A_i \in \mathcal{TN}$ für $i = 1, \ldots m$, in Haskell **AfA**
- $A ::= C_1 A_2$ mit $A, A_2 \in \mathcal{TN}$ und $C_1 \in \mathcal{CN}$, in Haskell **ACA**
- $A ::= A'$ mit $A, A' \in \mathcal{TN}$, in Haskell **Lam**

Die Unterscheidung zwischen TNT-Regeln und CNT-Regeln vereinfacht die eigentliche Ersetzung. Wird der zu ersetzende Term von einer TNT-Regel erzeugt, kann die Regel direkt zu einer Regel der Form $A ::= A'$ umgeändert werden. Aus dem originalen Nichtterminal wird direkt der ersetzende Term erzeugt:

Beispiel 5 *Seien $A \in \mathcal{TN}$, die Produktionsregel $P_1 : A \to \ldots$ aus einer STG und einer gTRS-Regel $R_1 : l \to r$.*

Gilt $val(A) = val(l)$*, so kann die Produktionsregel* P_1 *geändert werden zu* $A \to r$*.*

Desweiteren ist der Vergleich mittels Plandowski für ein Term-Nichtterminal direkt ausführbar. Für ein Kontext-Nichtterminal müssen noch verschiedene Vorbereitungen durchgeführt werden, bevor getestet werden kann. Dies wird im Unterkapitel 5.6 detailiert besprochen.

Implementierung der Ersetzung

Im Haskell-Programm werden alle TNT-Regeln der Reihe nach mit Plandowskis Algorithmus mit der linken Seite der gTRS-Regel verglichen. Sind die erzeugten Wörter gleich, so wird die alte Regel aus der Grammatik gelöscht und eine neue Regel in der entsprechenden Form erzeugt. Folgende Funktionen zur Termersetzung von Term-Nichtterminalen sind im Modul **`TermReplacement`** implementiert:

1. **`checkTNTs`** filtert aus einer Liste aller TNTs diejenigen heraus, die den zu ersetzenden Term erzeugen.

```
checkTNTs cmb r m =
  let
    candidates = filter
      (isPlandowskiWise2 m r)
      (getTermNonTerminals (stg cmb))
    tnts = map removeNTMarker candidates
    listToCheck = createTNTchecklist
      (tnts)
      (removeNTMarker r)
  in
    [TermNT (fst (pair !! 0)) | pair <- listToCheck
      ,(plandowski (stg cmb) pair) == True]
```

2. **`createReplaceList`** generiert rekursiv eine Liste mit Ersetzungspaaren der Form $(oldNT, newNT)$.

```
createReplaceList _ [] _ = []
createReplaceList cmb (set:xs) lenmap =
  let
    repList = checkTNTs cmb (fst set) lenmap
    tuples = zip repList (cycle [(snd set)])
  in
    tuples ++ (createReplaceList cmb xs lenmap)
```

3. **replaceRules** erzeugt für jedes Ersetzungspaar einer Liste rekursiv eine neue Regel vom Typ **Lam**, in der direkt der einzusetzende Teil generiert wird.

```
replaceRules cmb [] = cmb
replaceRules cmb (x:xs) =
  let
    toReplace = fst x
    replacer = snd x
    newCombo = appendRule (stg cmb) (Lam toReplace replacer)
  in
    replaceRules (cmb {stg = newCombo}) xs
```

In der Funktion **appendRule** wird eine Regel der STG hinzugefügt. Existiert schon eine Regel mit gleichem Nichtterminal auf der linken Seite, so wird die alte Regel überschrieben.

4. **exchangeARules** erstellt zuerst eine Liste von Ersetzungspaaren und führt anschliessend die Ersetzung durch.

```
exchangeARules cmb =
  let
    m = createLenMapSTG (stg cmb)
  in
    replaceRules cmb (createReplaceList cmb (trs cmb) m)
```

5.6 Ersetzung von Kontext-Nichtterminalen

Wird ein zu ersetzender Term nicht komplett durch Term-Nichtterminale erzeugt, ist die Ersetzung ungleich schwerer. Dies beginnt damit, dass in einen Kontext erst ein Term eingesetzt werden muss, bevor er mit dem zu ersetzendem Term verglichen werden kann. Daher muss der Term, der in den Kontext eingesetzt wird, auf der Suche nach der passenden Position bekannt sein.

Eine weitere Schwierigkeit ist der Vorgang des Ersetzens selbst. Hier muss nicht nur ein Term-Nichtterminal umgeändert werden, sondern unter Umständen auch noch neue Kontext-Nichtterminal Regeln hinzugefügt werden. Diese erzeugen das Prefix aus neuen Kontext-Nichtterminalen heraus. Damit der gesamte Vorgang ein paralleler Reduktionsschritt ist, werden keine existierenden CNT-Regeln verändert.

Der gesamte Vorgang ist stark abhängig davon, welche Kontext-Nichtterminal Regeln verwendet werden und betrachtet werden müssen. Der Algorithmus läuft vereinfacht wie folgt ab:

1. Für jede Regel des Typs **`ACA`** überprüfe C und merke das A der rechten Seite, also den in den Kontext einzusetzenden Term.
2. Je nach Typ der Regel, die das C auf der linken Seite hat, muß unterschiedlich überprüft werden. Im Allgemeinen läuft das nach folgendem Prinzip ab:
   - Erzeuge ein neues Term-Nichtterminal A', füge die Regel $A' ::= CA$ zur STG hinzu und teste den zu ersetzenden Term gegen den Term, der durch A' erzeugt wird.
     - Falls der gleiche Term erzeugt wird, wird die C-Regel zu einem Loch und das A wird zu einer Regel des Typs **`Lam`**.
     - Ist der Term nicht gleich, muss die C-Regel durchgegangen werden und darin enthaltene Kontext-Nichtterminale rekursiv betrachtet werden.

   Die unterschiedlichen Regeln werden durch *Pattern Matching* erkannt und behandelt. Die Diskussion der Fälle ist weiter unten im Detail aufgeführt.

Bei der Suche nach dem zu ersetzenden Term muß unter Umständen der gesamte Baum des in der STG gespeicherten Terms durchsucht werden. Dies geschieht mit einem rekursiven Algorithmus, der als Rückgabewert eine Liste mit neuen Regeln hat.

Ist die Liste nicht leer, so wurde der Term gefunden und die vorherige Rekursionsstufe kann eigene neue Regeln der Liste hinzufügen. Dies ist je nach Regeltyp und Position, an der der Term erzeugt wird, notwendig. Eine leere Liste bedeutet umgekehrt, dass der Term nicht in dem untersuchten Zweig gefunden wurde.

Weiterhin wird in jeder Rekursionsstufe ein neues Kontext-Nichtterminal erzeugt, das die nächste Stufe verwenden kann. Wird der Term erzeugt, so muss eine neue Regel hinzugefügt werden, in die ersetzt wird. Allerdings muss diese Regel von der übergeordneten Regel "aufgerufen" werden, weshalb es an dieser Stelle erzeugt werden muss und der nächsten Stufe mitgegeben werden muss.

5.6.1 Startpunkt und Tests

Startpunkt der Ersetzung

Der Startpunkt für die Suche nach dem Term in Kontext-Nichtterminalen ist eine Regel des Typs **ACA**. Nur in diesem Typ wird ein kompletter Term (also kein Kontext) unter Einbeziehung eines Kontextes erzeugt.

Quellcode 5.11: Start der CNT-Ersetzung mit einer **ACA** Regel

```
processACARule stg m (ACA a1 c a2) rule replacer =
  if shouldIGoFurther m rule [a2,c]
  then
    let
      cX = newCtxtNT stg
      lst = processCRule stg m c a2 rule cX
    in
      if   length lst > 0
      then lst ++ [(ACA a1 cX replacer)]
      else []
  else []
checkACAs cmb m (rule, replacer) =
  let
    rulesSTG = getACARulesSTG cmb
  in
    [processACARule (stg cmb) m r rule replacer | r <- rulesSTG]
```

Die Funktion **checkACAs** aus Beispiel 5.11 zeigt den Start der Ersetzung. Im Beispiel wird auch der oben beschriebene Mechanismus deutlich. Es wird das Kontext-Nichtterminal **cX** erzeugt, dass der rekursiven Funktion **processCRule** mitgegeben wird. In dieses wird bei erfolgreicher Suche nach dem Term eine neue Regel geschrieben und in Form der Rückgabeliste zurückgegeben. Ist sie nicht leer wird die Liste um eine neue **ACA** Regel erweitert, die die ursprüngliche Regel ersetzt. In ihr wird der ersetzende Term (**replacer**) des Termersetzungssystems direkt in die neue Regel des Nichtterminals **cX** eingesetzt.

Tests innerhalb des Algorithmus

Im rekursiven Algorithmus **processCRule** muss ein Kontext-Nichtterminal C und ein Term-Nichtterminal A zusammen mit dem zu ersetzenden Term auf Gleichheit getestet werden. Zu diesem Zweck wird ein neues Term-Nichtterminal A' erzeugt und die Regel $A' ::= CA$ der STG hinzugefügt. In A' wird der Term, der in A gespeichert ist, in den Kontext von C zu einem Term eingesetzt. Der Test ist in der Funktion **checkCwithA** in Quellcode 5.12 implementiert.

Quellcode 5.12: Test von Kontext und Term gegen zu ersetzenden Term

```
checkCwithA stg c a rule =
  let
    newA = newTermNT stg
    newP = ACA newA c a
    newG = appendRule stg newP
    testset = [( (removeNTMarker newA), (removeNTMarker rule))]
  in
    plandowski newG testset
```

5.6.2 Rekursive Suche nach dem Term

Je nach Typ der Regel, die auf der linken Seite das zu untersuchende Kontext-Nichtterminal hat, muss unterschiedlich vorgegangen werden. Um zwischen den Typen zu unterscheiden benutzt die Funktion **processCRuleX** *Pattern Matching*. Diese wird von der Funktion **processCRule** aufgerufen, die alle Parameter bis auf das

CNT unverändert weitergibt. Zu dem CNT wird die Regel aus der **stg** rausgesucht, die das CNT auf der linken Seite hat und damit die Funktion **processCRuleX** aufgerufen, wie in Quellcode 5.13 gezeigt ist.

Quellcode 5.13: Aufruf der Fallunterscheidung

```
processCRule stg m c a rule cX =
    processCRuleX stg m (getRule stg c) a rule cX

processCRuleX stg m (CfAC c1 f al1 c2 al2) a rule cX = ...
processCRuleX stg m (CCC c c1 c2) a rule cX = ...
processCRuleX stg m _ a rule cx = []
```

In jedem der drei Fälle wird nun überprüft, ob der aktuelle Term eingesetzt in die Kontext-Nichtterminale der rechten Seite mit dem zu ersetzendem Term übereinstimmen.

Regeltyp `CfAC`

Die Regel vom Typ **CfAC** hat auf der rechten Seite nur ein Kontext-Nichtterminal. Dieses muß nur dann überprüft werden, wenn das Wort, dass sich aus Term und Kontext ergibt, die gleiche Länge hat wie der Term, der zu ersetzen ist. Falls die Terme übereinstimmen, muss die Grammatik geändert werden. Andernfalls muss das Kontext-Nichtterminal auf der rechten Seite rekursiv als neuer Startpunkt untersucht werden. Der in den Kontext eingesetzte Term ändert sich nicht. Der gesamte Fall ist in Quellcode 5.14 dargestellt und wird nun im Detail besprochen.

Quellcode 5.14: Suche und Ersetzung in der **CfAC**-Regel

```
processCRuleX stg m (CfAC c1 f al1 c2 al2) a rule cX =
  if isPlandowskiWise3 m rule a c2
  then
    if checkCwithA stg c2 a rule
    then
      let
        r1 = (CfAC cX f al1 c2 al2)
        stg1 = appendRule stg r1
        cX2 = newCtxtNT stg1
```

```
        r2 = (CEmpty cX2)
        r3 = (CfAC cX f al1 cX2 al2)
      in
        [r3, r2]
  else []
else
  if shouldIGoFurther m rule [a,c2]
  then
    let
      r1 = (CEmpty cX)
      stg1 = appendRule stg r1
      m1 = createLenMapSTG stg1
      cX2 = newCtxtNT stg1
      lst = processCRule stg1 m1 c2 a rule cX2
    in
      if   length lst > 0
      then lst ++ [(CfAC cX f al1 cX2 al2)]
      else []
  else []
```

- Die Längen der Terme stimmen überein (**isPlandowskiWise3 = true**) und die Terme sind gleich (**checkCwithA = true**)

  1. Die aktuelle **CfAC**-Regel muß an der Stelle des Kontext-Nichtterminals **c2** ein Loch bekommen. Anstatt das alte CNT **c2** in eine Loch-Regel zu ändern, wird ein neues Kontext-Nichtterminal **cX2** generiert, das in der Regel **r2** ein Loch erzeugt. Die ursprüngliche Regel kann nämlich noch an einer anderen Stelle in der Grammatik verwendet werden und darf daher nicht überschrieben werden. Wie besprochen wird die geänderte **CfAC**-Regel **r3** in das vorher erzeugte CNT **cX** geschrieben. Beide neuen Regeln **r2** und **r3** werden als Ergebnis zurückgegeben.

  2. Der eingesetzte Term muß durch den einzusetzenden Term ersetzt werden. Dies geschieht am Einstiegspunkt der Suche und in den Rekursionsstufen davor.

- Die Längen der Terme stimmen überein (**isPlandowskiWise3 = true**) und die Terme sind ungleich (**checkCwithA = false**).

Die Suche nach dem zu ersetzenden Term kann an dieser Stelle ergebnislos abgebrochen werden: Rückgabewert = **`[]`**.

- Der zu ersetzende Term ist kürzer als der Term, der entsteht, wenn man den aktuellen Term in den Kontext einsetzt (**`isPlandowskiWise3 = false`** und **`shouldIGoFurther = true`**).

  Die Suche muss rekursiv fortgesetzt werden. Falls sich der zu ersetzende Teil im Unterterm befindet, wird eine neue Regel eingefügt, weshalb ein leeres Kontext-Nichtterminal **`cX2`** erzeugt wird. Dieses wird dem rekursiven Aufruf mitgegeben. Kann der Term nicht gefunden werden, ist das Ergebnis der Rekursion eine leere Liste, die weitergegeben wird.

  Konnte der Term gefunden werden, muss wie besprochen die geänderte **`CfAC`**-Regel in das vorher erzeugte CNT **`cX`** geschrieben werden. Die Regel wird an die Ergebnisliste mitangehängt und zurückgegeben.

- Der zu ersetzende Term ist länger als der Term, der entsteht, wenn man den aktuellen Term in den Kontext einsetzt (**`isPlandowskiWise3 = false`** und **`shouldIGoFurther = false`**)

  Die Suche nach dem zu ersetzenden Term kann an dieser Stelle ergebnislos abgebrochen werden: Rückgabewert = **`[]`**.

Regeltyp CCC

Der Hauptunterschied zwischen einer **`CCC`**- und einer **`CfAC`**-Regel ist die Anzahl der Kontext-Nichtterminale auf der rechten Seite. Das grundsätzliche Prinzip der **`CfAC`**-Regel bleibt erhalten, aber es muss geschachtelt in beiden CNTs gesucht werden. Die Abfolge der Suche richtet sich nach der Reihenfolge der Kontexteinsetzung in der Regel, wie in Beispiel 6 verdeutlicht wird.

Beispiel 6 *Seien $A, A_1 \in \mathcal{TN}_G$, $C, C_1, C_2 \in \mathcal{CN}_G$ und die Produktionsregeln*

- $A \to CA_1$,
- $C \to C_1C_2$

aus einer STG G.

Dann berechnet sich der Term, der in A gespeichert ist durch

$$val_G(A) = val_G(C[val_G(A_1)]) = val_G(C_1[val_G(C_2[val_G(A)])]).$$

In das CNT C_1 wird ein Term eingesetzt, der sich aus der Einsetzung von A_1 in C_2 ergibt. In der Funktion ist das so gelöst, dass ein neues Term-Nichtterminal A' erzeugt wird und eine passende Regel $A' \rightarrow C_2 A_1$ der STG hinzugefügt wird. C_1 kann nun zusammen mit dem Term, der in A' gespeichert ist, untersucht werden. In Quellcode 5.15 ist die Funktion **processCRuleX** für eine **CCC**-Regel aufgeführt. Teile, die ähnlich zum **CfAC**-Fall sind, wurden durch Kommentare ersetzt.

Quellcode 5.15: Suche und Ersetzung in der **CCC**-Regel

```
processCRuleX stg m (CCC c c1 c2) a rule cX =
if isPlandowskiWise3 m rule a c2
then
  if  checkCwithA stg c2 a rule
  then
    -- wie oben neue Regel erzeugen und hinzufügen
    return [r2,r3]
  else []
else
  if shouldIGoFurther m rule [a,c2, c1]
  then
    -- wie oben rekursiv in c2 mit a weiter suchen
    lst = processCRule stg1 m c2 a rule cX2
    if length lst > 0
    then    lst ++ [(CCC cX c1 cX2)]
    else
      let
        -- jetzt in c1 mit neuem Term suchen:
        newA1 = newTermNT stg
        stg1 = appendRule stg (ACA newA1 c2 a)
        m1 = createLenMapSTG stg1
        cX2 = newCtxtNT stg1
        lst = processCRule stg1 m c1 newA1 rule cX2
      in
        if   length lst > 0
        then lst ++ [(CCC cX cX2 cX4), (CEmpty cX4)]
        else []
  else []
```

Regeltyp `CEmpty`

Der Fall, dass das betrachtete Kontext-Nichtterminal ein Loch erzeugt, kann nicht vorkommen. In das Nichtterminal wird der aktuelle Term schon auf der Rekursionsstufe davor eingesetzt. Von dem Kontext-Nichtterminal wird selbst nichts dem Kontext hinzugefügt, so dass es nicht weiter betrachtet werden muss. Wurde der zu ersetzende Term bisher nicht gefunden, ändert sich dies in diesem Fall nicht.

5.6.3 Aufruf der Ersetzung

```
exchangeCRule cmb [] = cmb
exchangeCRule cmb (replacePair:xs) =
  let
    m = createLenMapSTG (stg cmb)
    newRules = flatten (checkACAs cmb m replacePair)
    newSTG = appendRules (stg cmb) (newRules)
  in
    exchangeCRule (cmb {stg = newSTG}) xs

exchangeCRules cmb = exchangeCRule cmb (trs cmb)
```

5.7 Optimierung der STG

Die Optimierung besteht aus zwei verschiedenen Schritten, die durch die Funktionen **`cleanUp`** und **`optimize`** implementiert sind. In **`cleanUp`** werden CNT-Regeln entfernt, die nicht von TNT-Regeln aus erreicht werden. Die Funktion **`optimize`** führt grundlegende Vereinfachungen der Regeln der STG durch.

5.7.1 Nicht erreichte CNT-Regeln

Durch das Ersetzen von Kontext-Nichttrerminale kann es passieren, dass komplette Teile des originalen Terms entfernt werden. Die zur Produktion des entfernten Terms notwendigen Regeln befinden sich nach dem geschilderten Ersetzungsverfahren noch

in der Grammatik. Da manche Regeln aber an verschiedenen Stellen in der Erzeugung eines Termes eingesetzt werden, können diese nicht direkt entfernt werden. Erst in einem zusätzlichen Schritt nach der kompletten Ersetzung einer gTRS-Regel können nicht mehr benötigte Regeln sicher identifiziert und entfernt werden.

Das Verfahren zum Auffinden von nicht benötigten CNT-Regeln entspricht einer Tiefensuche und geht wie folgt vor:

1. Bestimme alle TNTs der STG.
2. Für jede Regel des Typs **ACA** (nur in diesem Typ werden CNT-Regeln aus TNT-Regeln erreicht):
   - Bestimme alle TNTs und CNTs, die aus der Regel erreicht werden.
   - Wiederhole das Verfahren rekursiv mit den aktuell gefundenen Nichtterminalen als Eingabe.

Das Verfahren terminiert, da eine STG per Definition azyklisch ist und somit keine Schleifen, die eine Endlosrekursion verursachen können, enthält. Um den Prozess zu beschleunigen, werden bereits gefundene und bearbeitete Nichtterminale durch Aufnehmen in eine Liste markiert und nicht erneut betrachtet.

Der Algorithmus 5.16 aus dem Modul **CleanUp** erzeugt mit dem geschilderten Verfahren eine Liste von Regeln aus der die neue STG konstruiert wird - nicht erreichte Regeln sind nicht in der Liste und somit nicht mehr Teil der STG.

Quellcode 5.16: Entfernen von nicht erreichten CNT-Regeln

```
findUsedRules stg [] found = []
findUsedRules stg (ctxtnt:xs) found =
  if elem ctxtnt found
  then
    findUsedRules stg xs found
  else
    let
      rule = (getRule stg ctxtnt)
      lstCNT = getConnectedNTs rule
    in
      [rule]
      ++ findUsedRules stg (xs ++ lstCNT) (found ++ [ctxtnt])
```

```
cleanUpSTG stg =
  let
    lstTNTprods = [x | x<- (toSTGList stg)
                     , isTermNT (getLeftSide x)]
    lstTNT = map getLeftSide lstTNTprods
    lstCNT = flatten (map getConnectedNTs lstTNTprods)
  in
    prodListToSTG (lstTNTprods ++ findUsedRules stg lstCNT [])

cleanUp cmb = cmb {stg = cleanUpSTG (stg cmb)}
```

Von einem Entfernen von nicht erreichten TNT-Regeln wurde abgesehen, da eine STG keinen definierten "Einstiegspunkt" in Form eines Startsymbols hat. Mit dieser Erweiterung gäbe es auch eine Möglichkeit entsprechende Term-Nichtterminale zu entfernen.

5.7.2 Vereinfachungen einer STG

Im Modul **`Optimize`** sind zwei einfache Vereinfachungen einer STG implementiert.

Vereinen von `CEmpty`-Regeln

Zum einen braucht eine STG nur ein einziges Kontext-Nichtterminal, das ein Loch erzeugt. Falls es mehrere CNTs gibt, die ein Loch generieren, können diese bis auf eins gelöscht werden und durch das verbleibende ersetzt werden. Das Verfahren zum Auffinden ist im Beispiel 5.17 aufgeführt.

Quellcode 5.17: Entfernen von unnötigen **CEmpty**-Regeln

```
unifyCEmptySTG stg =
  let
    rules = (toSTGList stg)
    lstCEmpty = map getLeftSide (filter isCEmptyRule rules)
  in
    if length lstCEmpty > 1
    then
      let
        winner = head lstCEmpty
        loosers = tail lstCEmpty
        pairs = zip loosers (cycle [winner])
      in
        transformSTG rules pairs
    else stg

unifyCEmpty cmb = cmb {stg= (unifyCEmptySTG (stg cmb))}
```

Die Funktion **transformSTG** hat als Parameter neben einer Regelliste eine Liste von Ersetzungspaaren. Das erste Nichtterminal eines solchen Paares wird durch das zweite ersetzt. In der gesamten Regelliste wird nach dem zu ersetzenden Nichtterminal per *Pattern Matching* gesucht und ersetzt.

Entfernen von unnötigen CCC-Regeln

Eine Regel des Typs **CCC** ist genau dann verzichtbar, wenn eines der beide Kontext-Nichtterminale auf der rechten Seite ein Loch erzeugt. Es wird in das Loch des Kontextes wieder ein Loch eingesetzt. In einem solchen Fall können alle Vorkommen des Kontext-Nichtterminals auf der linken Seite durch das Kontext-Nichtterminal der rechten Seite ersetzt werden, dass kein Loch erzeugt.

Der Algorithmus geht wie folgt vor:

1. Erzeuge eine Liste alle Regeln.
2. Filtere die Liste nach **CCC**-Regeln.
3. Transformiere jede Regel zu einem Tupel der Kontext-Nichtterminale der rechten Seite.

4. Generiere eine Liste aller Kontext-Nichtterminale, die ein Loch erzeugen.
5. Entferne alle Kontext-Nichtterminale, die ein Loch erzeugen, aus den Tupeln.
6. Nimm das erste Tupel, das nur noch aus einem Element besteht:
   - Ersetze das Kontext-Nichtterminale, das bei der Regel auf der linken Seite stand durch das verbliebene Kontext-Nichtterminal im Tupel.
7. Rufe die Funktion rekursiv mit der neuen Grammatik auf solange mindestens ein entfernbares Kontext-Nichtterminal gefunden wurde.

Die Funktion ist im Beispiel 5.18 aufgeführt und benutzt die im vorherigen Abschnitt vorgestellte Funktion **`transformSTG`**.

Quellcode 5.18: Entfernen von unnötigen CCC-Regeln

```
optimizeCCCSTG stg =
  let
   rules = (toSTGList stg)
   lstCCC = filter isCCCRule rules
   lstCCC2 = ((map getConnectedNTs lstCCC))
   lstCEmpty = map getLeftSide (filter isCEmptyRule rules)
   checkCEmpty lst = [c | c <- lst, not (elem c lstCEmpty)]
   pairs = zip (map getLeftSide lstCCC) (map checkCEmpty lstCCC2)
   pairs2 = [(cOld, head cNew) | (cOld, cNew) <- pairs
                               , length cNew == 1]
  in
   if      length pairs2 > 0
   then
     let
       fstPair = [pairs2 !! 0]
       toRemove = [getRule stg cOld | (cOld, cNew) <- fstPair]
       rules2 = [rule | rule <- rules, not (elem rule toRemove)]
       stgNew = transformSTG rules2 fstPair
     in
       optimizeCCCSTG stgNew
   else    stg

optimizeCCC cmb = cmb {stg= (optimizeCCCSTG (stg cmb))}
```

Vereinfachungen einer STG

Die besprochenen Funktionen des **Optimize**-Moduls werden durch die Funktion **optimize** aufgerufen:

```
optimize cmb = optimizeCCC (unifyCEmpty cmb)
```

5.8 Ausführung

Das gesamte Programm besteht nun aus der Hintereinanderausführung der einzelnen besprochenen Funktionen: **replace** ↦ **cleanUp** ↦ **optimize**. Diese werden auf eine **ComboSTGTRS** Instanz angewendet, bis sich die STG nicht mehr ändert. Die Hauptfunktion ist in Beispiel 5.19 zu sehen.

Quellcode 5.19: Hauptfunktion der Termersetzung

```
mainSingleStep cmb = optimize (cleanUp (replace cmb))

mainComplete cmb =
    let
        cmbNew = mainSingleStep cmb
    in
        if      (stg cmbNew) /= (stg cmb)
        then    mainComplete cmbNew
        else    cmb

main stg trs =
   extractSTG (mainComplete (createSTGTRScombo stg trs))
```

Die Funktion **extractSTG** liefert alle Regeln der STG zurück und fügt nur die Regel des TRS hinzu, auf die von der STG aus zugegriffen wird. Die Term-Nichtterminale der STG sind in der gesamten Ersetzung nicht verändert worden und das gespeicherte Wort kann an der gleichen Stelle, bzw. mit dem gleichen Term-Nichtterminal, erzeugt werden.

5.9 Komplexitätsbetrachtung

Für alle folgenden Betrachtungen sei n die Größe einer Grammatik nach Definition 15, also die Summe der Größen von allen rechten Seiten der Regeln.

CleanUp

Die Entfernung von nicht erreichten CNT-Regeln wird in linearer Laufzeit, $\mathcal{O}(n)$, durchgeführt. Aufgrund der Liste der durchgesehenen Regeln wird jede Regel nur einmal betrachtet.

Optimize

Das Unifizieren der **`CEmpty`**-Regeln wird in $\mathcal{O}(n)+\mathcal{O}(n\cdot m) = \mathcal{O}(n\cdot m)$ durchgeführt, wobei m die Anzahl der **`CEmpty`**-Regeln ist. Zunächst werden alle Regeln des Typs **`CEmpty`** in einem linearen Durchlauf bestimmt $\mathcal{O}(n)$. Anschließend werden in einem zweiten linearen Durchlauf alle Vorkommen der zu ersetzenden CNTs durch das übrig bleibende Kontext-Nichtterminal ersetzt $\mathcal{O}(n \cdot m)$.

Die **`CCC`**-Regeln werden in quadratischer Zeit $\mathcal{O}(n^2)$ optimiert. In linearer Zeit werden alle optimierbaren **`CCC`**-Regeln bestimmt. Anschließend wird für jede dieser Regeln alle Regeln der Grammatik durchgesehen und die Kontext-Nichtterminale ausgetauscht.

STGLenMap

Die Erstellung der **`STGLenMap`** erfolgt in $\mathcal{O}(n \cdot log\ n) + \mathcal{O}(n \cdot log\ n) = \mathcal{O}(n \cdot log\ n)$ durch ein zweistufiges Verfahren.

1. Topologische Sortierung der Nichtterminal: $\mathcal{O}(n \cdot log\ n)$.
2. Berechnung der Länge gemäß Reihenfolge der topologischen Sortierung: $\mathcal{O}(n\cdot log\ n)$. Der logarithmische Faktor entsteht durch den **`lookup`** Aufruf einer **`Data.Map`**.

Ersetzung

Für die Ersetzung ist es interessant, nicht nur die Laufzeit zu betrachten. Auch eine Betrachtung der Speicherkomplexität des Verfahrens muss durchgeführt werden. Beides wird zunächst für einen einzelnen Ersetzungsschritt und dann für das komplette Berechnen der Normalform analysiert.

Laufzeit

Die Ersetzung besteht aus der Hintereinanderausführung von TNT- und CNT-Ersetzung. Die Laufzeit eines Termvergleichs mittels Algorithmus von Plandowski erfolgt in polynomieller Zeit: $\mathcal{O}(n^p)$ für $p \in \mathbb{N}$. Die TNT-Regeln müssen jeweils einmal mit der Regel des gTRS verglichen werden, also $\mathcal{O}(n \cdot n^k) = \mathcal{O}(n^p)$.

Aufwändiger ist die Ersetzung, wenn der zu ersetzende Term ein echter Unterterm einer **`ACA`** Regel ist. Die CNT-Ersetzung hat Zugriff auf die **`STGLenMap`** und muss den Termvergleich nur durchführen, wenn beide Terme die gleiche Länge haben.
Allerdings muss dies schlimmstenfalls für alle in Frage kommenden Positionen auf der *Höhe in der Baumdarstellung des Terms* erfolgen, an der ein Term in der Grammatik mit passender Länge erzeugt wird. Da die Grammatik aber den Term komprimiert, werden viele der exponentiell vielen Positionen durch gleiche Subterme beschrieben. Das Verhalten des Algorithmus wird im Beispiel 5.5, der Pseudocode des Ersetzungsalgorithmus, deutlich. Er verzweigt in der Suche nicht, sondern läuft direkt alle Positionen hintereinander durch. Positionen nebeneinander gibt es nur in drei Regeltypen:

- **`AfA`**

  Es befindet sich weder auf der linken noch auf der rechten Seite ein Kontext-Nichtterminal. Die Regel wird in der CNT-Ersetzung nicht betrachtet.

- **`CfAC`**

  Auf der rechten Seite befinden sich eine Term-Nichtterminalliste und ein Kontext-Nichtterminal. Da nur die CNTs betrachtet werden, gibt es keine Verzweigung.

- **`CCC`**

  Durch die Einbeziehung der Größe der erzeugten Terme wird direkt an der richtigen Position geprüft.

Somit gibt es nur polynomiell viele Positionen an denen mit dem Algorithmus von Plandowski überprüft wird:

$$\mathcal{O}(n^{p_1}) \cdot \mathcal{O}(n^{p_2}) = \mathcal{O}(n^p).$$

Der gesamte Algorithmus hat polynomielle Laufzeit für einen einzelnen Schritt. Allerdings sind im schlechtesten Fall exponentiell viele Ersetzungen nötig. Daher hat die Berechnung der Normalform insgesamt exponentielle Laufzeit.

Speicherkomplexität

Die Ersetzung direkt in einer TNT-Regel verändert nur eine Regel, nicht aber die Größe der Grammatik.

Bei einer Ersetzung in einer CNT-Regel werden alle Regeln des Pfades vom Einstiegspunkt **ACA** bis hin zur Regel, in der erfolgreiche Termvergleich stattgefunden hat, kopiert und der Grammatik hinzugefügt. Die Größe wächst im schlimmsten Fall um n an für den Fall, dass alle Regeln kopiert werden müssen. Dies ist insgesamt eine Verdopplung der Größe der Grammatik.

Im folgenden Satz wird die Größe der Ergebnisgrammatik, die die Normalform des Terms speichert, betrachtet.

Satz 3 (Größe der Ergebnisgrammatik) *Die STG G_n, die die T-Normalform bezüglich eines STG-komprimierten gTRS T speichert, hat maximal die Größe $2 \cdot |G|$.*

Begründung:

- *Regeln der Kontext-Nichtterminale werden nicht verändert, höchstens gelöscht.*

  $\Rightarrow$ die Grammatik wird nicht vergrößert.

- *Regeln der Term-Nichtterminale werden verändert und hinzugefügt je nach Regeltyp:*

  1. *Regeln der Form $A ::= f(A_1, \ldots, A_n)$ können modifiziert werden zu Regeln der Form $A ::= R$, wobei R ein Term-Nichtterminal der rechten Seite einer Regel des TRS ist.*

     $\Rightarrow$ die Grammatik wird nicht vergrößert.

2. *Regeln der Form* $A ::= C[A']$ *werden durch die Ersetzung verändert und müssen über alle Schritte bis zur Nomalform analysiert werden. Im schlimmsten Fall wird eine Regeln der Form* $A ::= C[A']$ *in jedem Schritt verändert:*

$$A ::= C[A'] \longrightarrow A ::= C_1[R_1] \longrightarrow A ::= C_2[R_2] \xrightarrow{*} A ::= C_m[R_m].$$

In der Ersetzungssequenz ist R_i *ein Term-Nichtterminal auf der rechten Seite einer Regel des TRS und* $val(C_i)$ *das Prefix von* $val(C_{i-1})$.

Für jeden Schritt gilt:

- $C_i = C_{i-1}$

  $\Rightarrow$ *die Grammatik wird nicht vergrößert.*

- C_i *ist echter Prefix von* C_{i-1}*:*

  $\rightarrow$ C_{i-1} *wird von keiner Regel referenziert und kann gelöscht werden (Aussnahme:* C*).*

  $\Rightarrow$ *insgesamt wird ein neuer Kontext* C_m *hinzugefügt.* C_m *ist ein Prefix von* C *und damit höchstens so groß wie* C*. Es werden also höchstens so viele Produktionen wie zum Erzeugen von* C *zur Grammatik hinzugefügt.*

Da für alle Regeln der Grammatik so argumentiert werden kann und es nur $|G|$ *viele Regeln gibt, folgt die Behauptung.*

Kapitel 6

Tests

Durch verschiedene Testfälle (Kombinationen von STG und gTRS) soll die Implementierung nach den im Konzept vorgestellten Kriterien beurteilt werden können.

6.1 Grundsätzliche Funktion

Die Funktionsfähigkeit des Verfahrens ist sichergestellt, wenn der zu ersetzende Term in allen Regeltypen korrekt gefunden und ersetzt wird. Die Tests müssen daher die Fälle abdecken. In allen folgenden Grammatiken wird der Term $f(a)$ von der entsprechenden Regelart erzeugt. Somit reicht es aus, das folgende Termersetzungssystem mit allen Grammatiken für die Tests zu benutzen:

Quellcode 6.1: gTRS mit der Ersetzung $f(a) \mapsto b$

```
tDAG1 :: STG String Integer
tDAG1 = prodListToSTG
       [ AfA (toTermNT 1) "f" [toTermNT 2]
       , AfA (toTermNT 2) "a" []
       , AfA (toTermNT 3) "b" []
       ]

tTRS1 :: ReducedTRS String Integer
tTRS1 = (tDAG1, [(toTermNT 1, toTermNT 3)])
```

Term in Regel AfA

Durch das TNT 1 wird der Term $f(a)$ komplett durch eine **AfA**-Regel erzeugt.

Quellcode 6.2: $f(a)$ erzeugt durch eine **AfA**-Regel

```
tSTG0 = prodListToSTG
      [ AfA (toTermNT 1) "f" [(toTermNT 2)]
      , AfA (toTermNT 2) "a" []
      ]
```

```
*Test> val tSTG0 1
f(a)
*Test> let tSTG0x = main tSTG0 tTRS1
*Test> val tSTG0x 1
b
```

Term in Regel ACA

Die **ACA**-Regel von TNT 1 erzeugt den Term $f(a)$.

Quellcode 6.3: $f(a)$ erzeugt durch eine **ACA**-Regel

```
tSTG1 = prodListToSTG
       [ ACA (toTermNT 1) (toCtxtNT 3) (toTermNT 2)
       , AfA (toTermNT 2) "a" []
       , CfAC (toCtxtNT 3) "f" [] (toCtxtNT 4) []
       , CEmpty (toCtxtNT 4)
       ]
```

```
*Test> val tSTG1 1
f(a)
*Test> let tSTG1x = main tSTG1 tTRS1
*Test> val tSTG1x 1
b
```

Term in Regel CfAC

Der Term $f(a)$ wird durch das CNT 4 erzeugt, das von der **CfAC**-Regel CNT 3 in den Term eingebunden wird.

Quellcode 6.4: $f(a)$ erzeugt durch eine **CfAC**-Regel

```
tSTG2 = prodListToSTG
        [ ACA (toTermNT 1) (toCtxtNT 3) (toTermNT 2)
        , AfA (toTermNT 2) "a" []
        , CfAC (toCtxtNT 3) "f" [] (toCtxtNT 4) []
        , CfAC (toCtxtNT 4) "f" [] (toCtxtNT 5) []
        , CEmpty (toCtxtNT 5)
        ]
```

```
*Test> val tSTG2 1
f(f(a))
*Test> let tSTG2x = main tSTG2 tTRS1
*Test> val tSTG2x 1
f(b)
```

Term in Regel CCC, rechtes CNT

Das CNT 5 erzeugt den Term $f(a)$, das im rechten CNT der rechten Seite der **CCC**-Regel 4 eingebunden ist.

Quellcode 6.5: $f(a)$ erzeugt durch eine **CCC**-Regel rechts

```
tSTG3r = prodListToSTG
        [ ACA (toTermNT 1) (toCtxtNT 3) (toTermNT 2)
        , AfA (toTermNT 2) "a" []
        , CCC (toCtxtNT 3) (toCtxtNT 4) (toCtxtNT 4)
        , CCC (toCtxtNT 4) (toCtxtNT 5) (toCtxtNT 5)
        , CfAC (toCtxtNT 5) "f" [] (toCtxtNT 6) []
        , CEmpty (toCtxtNT 6)
        ]
```

```
*Test> val tSTG3r 1
f(f(f(f(a))))
*Test> let tSTG3rx = main tSTG3r tTRS1
*Test> val tSTG3rx 1
f(f(f(b)))
```

Term in Regel CCC, linkes CNT

Das CNT 5 erzeugt den Term $f(a)$, das im linken CNT der rechten Seite der **CCC**-Regel 4 eingebunden ist.

Quellcode 6.6: $f(a)$ erzeugt durch eine **CCC**-Regel links

```
tSTG3l = prodListToSTG
       [ ACA (toTermNT 1) (toCtxtNT 3) (toTermNT 2)
       , AfA (toTermNT 2) "a" []
       , CCC (toCtxtNT 3) (toCtxtNT 4) (toCtxtNT 4)
       , CCC (toCtxtNT 4) (toCtxtNT 5) (toCtxtNT 6)
       , CfAC (toCtxtNT 5) "f" [] (toCtxtNT 6) []
       , CEmpty (toCtxtNT 6)
       ]
```

```
*Test> val tSTG3l 1
f(f(a))
*Test> let tSTG3lx = main tSTG3l tTRS1
*Test> val tSTG3lx 1
f(b)
```

Term in Regel `Lam` oder `CEmpty`

Regeln des Typs **`Lam`** und **`CEmpty`** leisten keinen eigenen Beitrag zu einem Term - sie erweitern ihn nicht. Die Regeln müssen daher nicht durch eigene Testfälle überprüft werden. Sie funktionieren genau dann, wenn alle anderen Fälle korrekt verarbeitet werden.

Testergebnis

Alle Tests wurden erfolgreich ausgeführt.

6.2 Geschwindigkeitssteigerung durch Optimierung

Eine Hauptoptimierung ist die Benutzung der **`STGLenMap`**. Durch diese wird die Anzahl der Vergleiche mit Plandowski deutlich eingeschränkt und auch die rekursive Suche nach einem Term rechtzeitig abgebrochen. Um den Vorteil zu messen, wurde der Algorithmus zweimal auf eine Grammatik und ein TRS angwendet. Einmal mit Benutzung der **`STGLenMap`** und einmal ohne.

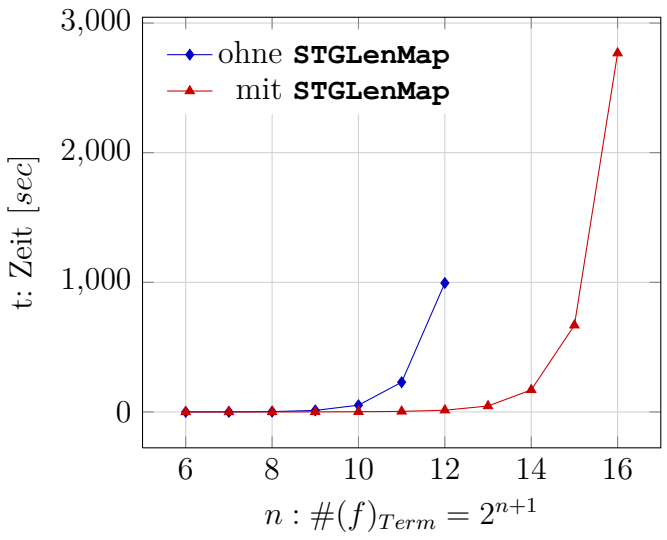
Die Grammatik wird durch die Funktion **`tSTGn`** erzeugt, die im Quellcode 6.7 gezeigt ist. Der Term, der in der Grammatik exponentiell komprimiert gespeichert ist, hat die Form $f(f(\ldots f(a)))$. Die Anzahl der f's ist über den Parameter n der Funktion kontrollierbar: $\#(f) = 2^{n+1}$.

Quellcode 6.7: Erzeugung einer Grammatik mit exponentieller Kompression

```
tSTGn n = prodListToSTG
([ACA (toTermNT 0) (toCtxtNT (2)) (toTermNT 1)
 ,AfA (toTermNT 1) "a" []
 ]
 ++
 [CCC (toCtxtNT i)(toCtxtNT (i+1))(toCtxtNT (i+1)) | i<-[2..n+2]]
 ++
 [CfAC (toCtxtNT (n+3)) "f" [] (toCtxtNT (n+4)) []
 ,CEmpty (toCtxtNT (n+4))
 ])
```

Das gTRS hat nur eine Regel: $f^8(a) \mapsto f^7(a)$. Die Ersetzung kann je nach n häufig angewendet werden. Die Benchmarks messen die Zeit der kompletten Ersetzung mit der **`main`** Funktion. Sie wurden drei mal ausgeführt und der Mittelwert ausgerechnet.

Als Testsystem wurde ein Windows 7 64-Bit Rechner mit Intel E6750 CPU, 4GB RAM und GHC Version 7.0.4 eingesetzt.



n	STGLenMap ohne [sec]	STGLenMap mit [sec]
6	0.1	0.003
7	0.6	0.009
8	2.7	0.2
9	11.9	0.6
10	52.2	1.6
11	229.5	4.4
12	994.7	13.4
13	>1h	45.9
14	...	170
15	...	668.8
16	...	2767.8

Abbildung 6.1: Benchmarks mit und ohne **STGLenMap**

Es ist in Abbildung 6.1 deutlich erkennbar, dass die Beschleunigung sehr effektiv funktioniert. Mit der **STGLenMap** ist die Laufzeit von 2^{15} f's besser als die von 2^{12} f's ohne Optimierung.

6.3 Geschwindigkeit in Abhängigkeit der Größe des TRS

Die Geschwindigkeit der Ersetzung ist auch im Hinblick auf die Größe des Termersetzungssystems interessant. Um diese zu messen wurde das im Quellcode 6.8 angegebene TRS benutzt. Die Größe der STG des TRS ist $|n+5|$ und es beinhaltet die Ersetzung: $f^{n+1} \to f^n$. Das TRS wurde auf die bereits beschriebene STG **tSTGn** aus Quellcode 6.7 angewendet.

Quellcode 6.8: Lineares TRS

```
tSTGnLin n = prodListToSTG
 ([ACA (toTermNT 0) (toCtxtNT 2) (toTermNT 1)
  ,AfA (toTermNT 1) "a" []]
 ++
 [CfAC (toCtxtNT (i)) "f" [] (toCtxtNT (i+1)) [] | i  <-[2..n+2]]
 ++
```

```
 [CEmpty (toCtxtNT (n+3))]
 ++
 [ACA (toTermNT (n+4)) (toCtxtNT 3) (toTermNT 1)])
tTRSn n = (tSTGnLin n , [(toTermNT 0, toTermNT (n + 4))])
```

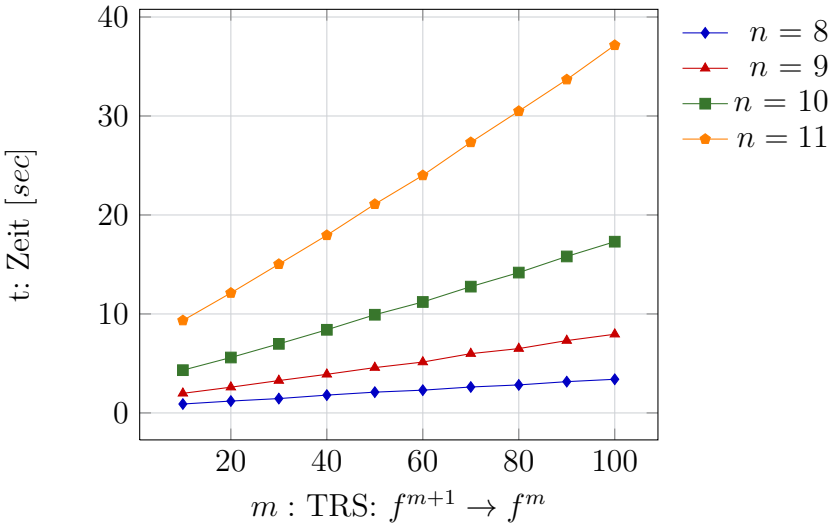


Abbildung 6.2: Benchmarks mit **`tSTGn n`** und **`tTRSn m`**

Zunächst erscheint es widersprüchlich, dass die Laufzeit mit einem größeren TRS ansteigt, wie in Abbildung 6.2 gezeigt ist. Der zu ersetzende Term wird nämlich größer und die Anzahl der ausgeführten Ersetzungen nimmt ab. Das Verhalten kann zwei Gründe haben. Zum einen wird der Algorithmus von Plandowski auf einen größeren Term ausgeführt, der sich an einer späteren Stelle unterscheidet. Eine zweite Erklärung ist in der fortwährenden Berechnung der **`STGLenMap`** zu suchen. Diese wird für die eigentliche STG und die STG des TRS durchgeführt. Je größer eine Grammatik desto länger dauert die Berechnung der **`STGLenMap`**.

Die Messwerte in Abbildung 6.3 zeigen, dass die Dauer der Erzeugung mit Größe einer STG zunimmt.

6.4 Anzahl der Ersetzungen

Für die Messungen wurden die im vorherigen Abschnitt verwendete STG und TRS benutzt. Die Anzahl der Ersetzungen nimmt in etwa linear invers mit der Größe

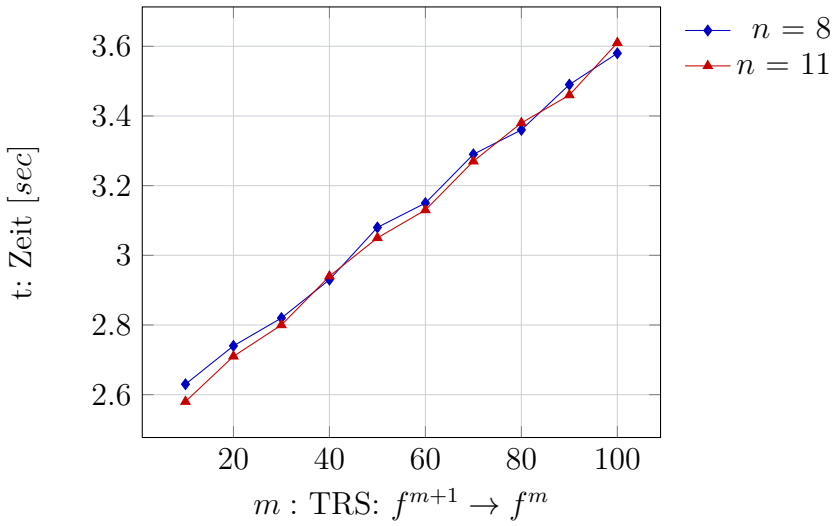


Abbildung 6.3: Benchmarks für die 500-fache Erzeugung der **`STGLenMap`**

des TRS ab. Dieses in Abbildung 6.4 dargetsellte Verhalten ist offensichtlich, da der Term entsprechend der Größe des zu ersetzenden Terms weniger eingsetzt werden kann.

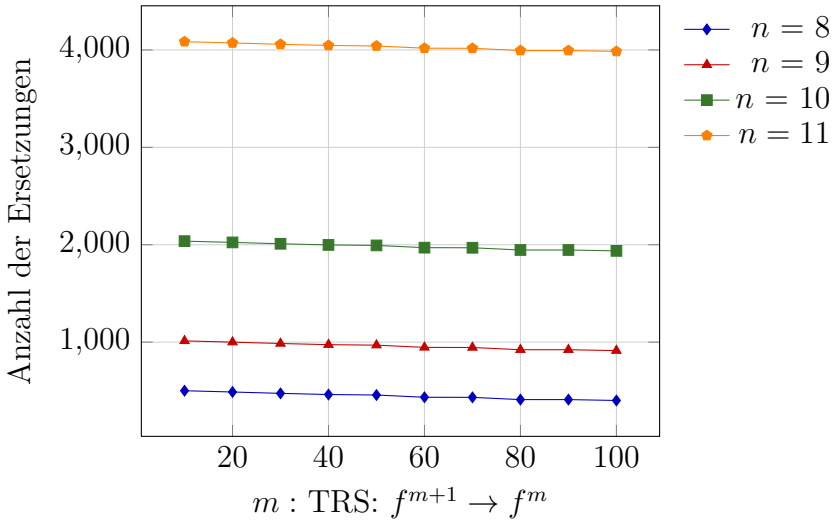


Abbildung 6.4: Benchmarks mit **`tSTGn n`** und **`tTRSn m`**

6.5 Größe der Grammatik

Die Größenentwicklung wurde an Hand von verschiedenen Grammatiken und Termersetzungssystemen getestet.

Allgemein ist festzustellen, dass sich die Regeln der STG nach der Ersetzung wie folgt zusammensetzen:

- Alle TNT-Regeln der STG

  Diese werden nicht entfernt, auch wenn sie nicht gebraucht werden. Es werden nur unerreichte CNT-Regeln gelöscht.

- Alle Regeln des ersetzten Wortes

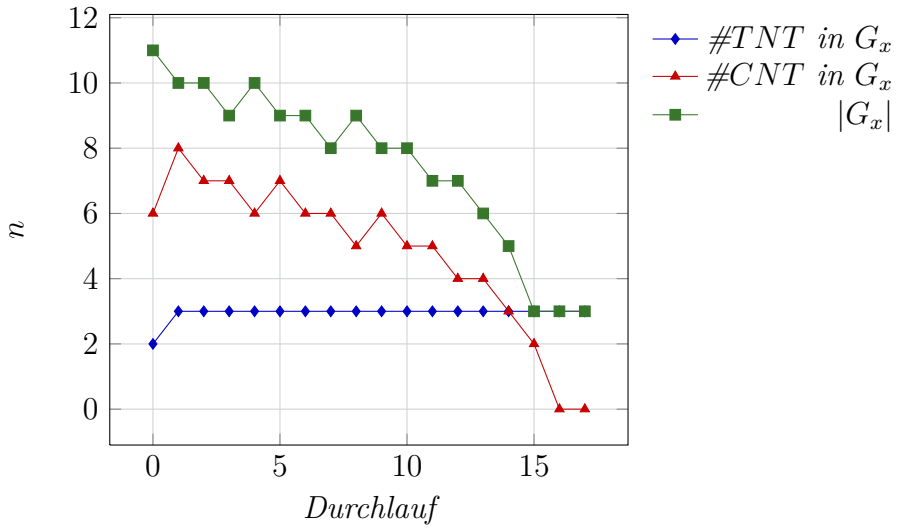
  Diese setzen sich zusammen aus *alten* Regeln der originalen Grammatik und *neuen* Regeln von den rechten Seiten der Ersetzungen des TRS.

- Alle CNT-Regeln des neuen Terms der alten STG

  Der Teil des Terms, der nicht ersetzt und durch CNT-Regeln erzeugt wurde, befindet sich noch im Term, muss also auch in der Grammatik vorkommen.

Beispiel 7 *Vollständige Ersetzung des Wortes:*

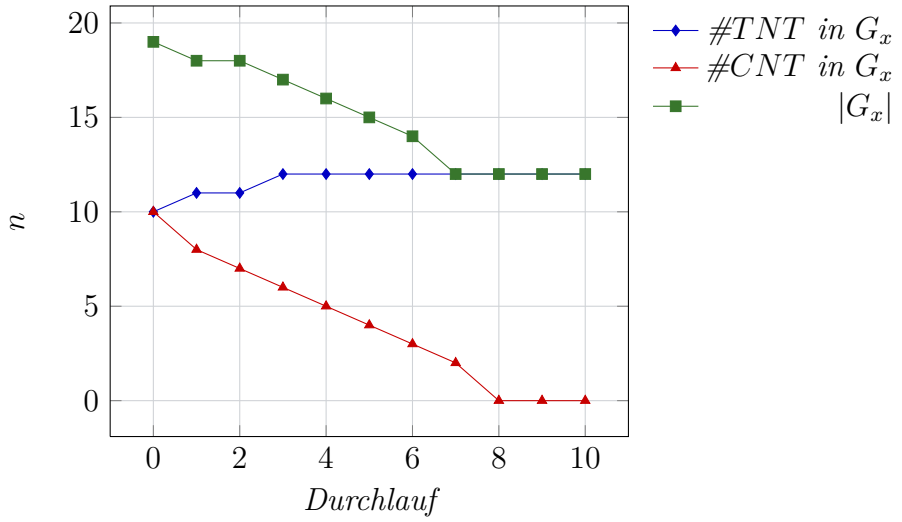
Eingabegrammatik	`0->*2,1; 1->a; *2->*3,*3; *3->*4,*4; *4->*5,*5; *5->*6,*6; *6->f(*7); *7->.`
Eingabewort	$f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(a)))))))))))))))))$
TRS Grammatik	`1->f(2); 2->a`
TRS	$f(a) \mapsto a$
Ausgabegrammatik	`0->9; 1->a; 9-> a`
Ausgabewort	a
|Grammatik|	*8 (STG) + 1 (TRS) → 3*



Die Ergebnisgrammatik enthält 3 Regeln, theoretisch würde eine Regel ausreichen: `0->a`*. Die Regel* `1` *ist aber eine jener originalen TNT-Regeln, die nicht entfernt werden. Regeln* `0` *und* `9` *stammen vom Ersetzungsprinzip, das eine* **`Lam`***-Regel benutzt und so auf das Term-Nichtterminal des ersetzenden Terms verweist.*

Beispiel 8 *Komplexer Boolescher Ausdruck*

Eingabegrammatik	*20 Nichtterminale: 10 TNT, 10 CNT*
Eingabewort	$\neg(\wedge(\neg(\wedge(\neg(\wedge(\neg(\wedge(1,\wedge(1,0))),\wedge(\neg(\wedge(1,\wedge(1,0))),0))),$ $\wedge(\neg(\wedge(\neg(\wedge(1,\wedge(1,0))),\wedge(\neg(\wedge(1,\wedge(1,0))),0))),0))),$ $\wedge(\neg(\wedge(\neg(\wedge(\neg(\wedge(1,\wedge(1,0))),\wedge(\neg(\wedge(1,\wedge(1,0))),0))),$ $\wedge(\neg(\wedge(\neg(\wedge(1,\wedge(1,0))),\wedge(\neg(\wedge(1,\wedge(1,0))),0))),0))),0)))$
TRS Grammatik	*12 Regeln*
TRS	*Wahrheitstabellen für logische Ausdrücke:* $\wedge, \neg, \vee$
Ausgabegrammatik	*12 Regeln, alle TNT-Regeln*
Ausgabewort	*true*
/Grammatik/	*20 (STG) + 12 (TRS)* $\rightarrow$ *12*



Die Beispiele stammen aus dem GBC-Modul **`GBC.STG.Examples`**. *Der Aufruf lautet* **`main exBoolSTG 5 exBoolTRS`**, *das Wort ist im Term-Nichtterminal* `17` *gespeichert.*

Die Ergebnisgrammatik enthält 12 Regeln. Neben den TNT-Regeln der STG sind 2 weitere TNT-Regeln hinzugekommen: `true` *und* `false`. *Sie sind die Regeln der rechten Seiten des TRS (Wahrheitswerte).*

Das Verhalten lässt sich auch für eine größere Grammatik, die mit der gleichen Funktion erzeugt wurde beobachten. Durch `exBoolSTG 23` *wird eine Grammatik mit insgesamt 92 Regeln generiert. Davon sind 46 TNT-Regeln. durch den Aufruf* **`main (exBoolSTG 23) exBoolTRS`** *wird die Grammatik zu einer mit 48 Regeln transformiert. Alle Regeln der Ergebnisgrammatik sind vom Typ TNT. Es wurden also nur*

die Regeln der rechten Seite des TRS hinzugefügt.

Kapitel 7

Bewertung und Ausblick

Im letzten Kapitel werden die Ergebnisse bewertet und mit Blick auf zukünftige Weiterentwicklungen betrachtet.

7.1 Bewertung

Die Aufgabenstellung dieser Arbeit lässt sich in zwei Teile aufspalten: die Entwicklung eines Parsers für STGs und eines Termersetzungsalgorithmus.

Parser

Der erarbeitete Parser erfüllt alle gestellten Anforderungen. Er kann leicht in bestehende Haskell-Programme integriert werden und erzeugt direkt nutzbare Instanzen einer `STG`. Diese kann auch benutzt werden, um Grundterme eines gTRS zu definieren. Die Implementierung wurde mit verschiedenen Beispielen ohne Fehler erfolgreich getestet.

Termersetzung

Die Umsetzung des in [SSSA11] vorgestellten Verfahrens zur Termersetzung mittels eines gTRS in einer STG ist der Schwerpunkt der Arbeit. Die hier konzipierte und umgesetzte Implementierung liefert das gewünschte Ergebnis. Es wurde gezeigt, dass sich die Termersetzung in STG-komprimierten Termen mit einem polynomiell platzbeschränkten Algorithmus durchführen lässt.

Die Ersetzungsfunktion behandelt alle Typen der Produktionsregeln einer STG. Selbst in "verschachtelten" Ersetzungsfällen (Typ: **ccc**, Ersetzung im linken Kontext-Nichtterminal) wird die korrekte Ersetzung durchgeführt.

Das Gesamtverfahren wurde über die eigentliche Aufgabenstellung hinaus noch auf drei Arten optimiert:

- Laufzeitoptimierung durch Minimierung der Äquivalenzvergleiche
- Zusammenfassen von Kontext-Nichtterminal Regeln
- Entfernung von unnötigen Kontext-Nichtterminal Regeln

Die Implementierung wurde umfassend durch Tests evaluiert. Neben der grundsätzlichen Funktionalität wurden Größe der Ergebnisgrammatik und Laufzeit der Ersetzung betrachtet. Die Länge der Ausgabe ist nicht zuletzt durch die Optimierungen positiv zu bewerten. Der Ersetzungsterm wird direkt an der richtigen Stelle integriert und unnötige Teile entfernt. Es muß allerdings erwähnt werden, dass unter Umständen zusätzliche Term-Nichtterminale in der Grammatik bleiben. Dies liegt daran, dass eine STG kein definiertes Startsymbol hat und somit der "gespeicherte" Term nicht eindeutig identifiziert werden kann. Die Laufzeit des Ersetzungsvorgangs konnte durch Vorberechnungen der Längen von erzeugten Termen deutlich verbessert werden.

7.2 Ausblick

Die vorgestellten Methoden lösen die durch die Aufgabenstellung aufgeworfenen Probleme zufriedenstellend. Dennoch lassen sich verschiedene Aspekte ausmachen, an denen die Verfahren weiterentwickelt und verbessert werden können.

Parser

Der Parser verarbeitet korrekte Eingaben anstandslos. Beinhalten diese jedoch syntaktische Fehler, so bricht er ohne weitere Informationen die Ausführung ab. Ein Feedback für den Nutzer würde diesem die Fehlersuche erleichtern. Hierfür bietet der eingesetzte Parsergenerator *Happy* auch entsprechende Möglichkeiten.

Der entworfene Parser akzeptiert nur Zahlen als Nichtterminale. Kontext-Nichtterminale müssen sogar explizit gekennzeichnet werden. Beides ist für eine

grundsätzliche Implementierung akzeptabel, allerdings schränkt es einen Anwender unnötig ein. So ist es denkbar, mit einem zusätzlichem Vorverarbeitungsschritt auch die Benennung von Nichtterminalen mit Buchstaben zu erlauben. Die Art der Nichtterminale kann nach dem zerlegen in einen Tokenstrom eindeutig geklärt werden, so dass hier die explizite Markierung entfallen kann.

Ist die eingegebene STG semantisch nicht korrekt (zyklisch, nicht definierte Nicht-Terminal, etc.), so bekommt dies der Nutzer nicht mitgeteilt. Eine Überprüfung im Anschluss an einen erfolgreichen Parseraufruf könnte diese potentielle Fehlerquelle eliminieren.

Der Parser erzeugt eine STG, die auch Basis eines gTRS sein kann. Die Definition der Ersetzungsregeln, also die Angabe, welches Term-Nichtterminal durch welches ersetzt werden soll, muss jedoch im Quellcode erfolgen. Ein Parser für ein gTRS könnte leicht implementiert werden. Dieser könnte den Parser für STGs verwenden und sich nur um das Parsen der Ersetzungsregeln kümmern.

Termersetzung

Die Verwendung der **`STGLenMap`** hat sich im Gesamten positiv auf die Laufzeit der Termersetzung ausgewirkt. Allerdings ist durch die Benchmarks auch klar geworden, dass diese noch optimiert werden kann. In der aktuellen Implementierung wird die **`STGLenMap`** nach erfolgter Ersetzung komplett neu berechnet. Dies ist sicherlich nicht notwendig. Insbesondere das statische und von der Ersetzung unbetroffene TRS, das mit in der Grammatik abgespeichert werden muss, ändert seine Länge nicht. Aber auch der geänderte Teil könnte unter Umständen nur aktualisiert werden. Dies wurde in der aktuellen Arbeit nicht näher betrachtet.

Eine STG enthält in der Regel mehrere Term-Nichtterminale. An welchem der eigentliche Term gespeichert wird, ist aus der reinen Grammatik nicht ersichtlich. Mit diesem zusätzlichen Wissen könnte die Ergebnisgrammatik noch weiter optimiert bzw. verkürzt werden. In den Tests ist der Fall häufig aufgetreten, dass die Grammatik aus mehreren nicht benötigten Term-Nichtterminalen bestand. Eine ad-hoc Lösung wäre die Untersuchung der Grammatik vor der Ersetzung. So könnte man das Term-Nichtterminal indentifizieren, das den längsten Term erzeugt und dieses als “Startsymbol” auffassen. Testweise liesen sich damit sehr brauchbare Ergebnisse erzielen.

Die Arbeit hat sich explizit nur mit einem gTRS als Ausgangspunkt für die Ersetzungen befasst. Interessant ist allerding auch der Fall, dass die Terme keine Grundterme sind, also ein allgemeines TRS als Basis dient. Der grundsätzliche Ersetzungsalgorithmus sollte ohne große Änderungen weiterverwendet werden können. Nur der Vergleichsalgorithmus von Plandowski müsste geändert oder ersetzt werden. Dies ist somit trotz der geleisteten Arbeit als sehr schwieriges Problem einzustufen.

Literaturverzeichnis

[BLM05] Busatto, Giorgio ; Lohrey, Markus ; Maneth, Sebastian: Efficient Memory Representation of XML Documents. In: *Lecture Notes in Computer Science* 3774/2005 (2005), S. 199–216

[BN99] Baader, Franz ; Nipkow, Tobias: *Term Rewriting and All That.* Cambridge University Press, 1999. – 316 S. – ISBN 0521779200

[ghc] *GHC - Glasgow Haskell Compiler*

[GJ10] Grune, Dick ; Jacobs, Ceriel: *Parsing Techniques: A Practical Guide (Monographs in Computer Science).* Springer, 2010. – 688 S. – ISBN 1441919015

[HU00] Hopcroft, John E. ; Ullman, Jeffrey D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie [Broschiert].* Oldenbourg R. Verlag GmbH; Auflage: 3., korr. A., 2000. – 461 S. – ISBN 3893197443

[LSA08] Lam, Monica S. ; Sethi, Ravi ; Aho, Jeffrey D. Ullman Alfred V.: *Compiler.* Addison Wesley Verlag, 2008. – ISBN 3827370973

[Mar01] Marlow, Simon. *Happy User Guide.* 2001

[Pla94] Plandowski, Wojciech: Testing Equivalence of Morphisms on Context-Free Languages. (1994), September, S. 460–470. ISBN 3–540–58434–X

[Sab11] Sabel, David. *GBC - Grammar Based Compression.* 2011

[SS10] Schmidt-Schauss, Manfred. *Reduktionssysteme und Termersetzung.* 2010

[SS11a] Schmidt-Schauss, Manfred: Einführung in die Funktionale Programmierung. (2011)

[SS11b] Schmidt-Schauss, Manfred: Pattern Matching of Compressed Terms and Contexts and Polynomial Rewriting / Goethe-Universität Frankfurt. Frankfurt, 2011. – Forschungsbericht

[SSSA11] Schmidt-Schauss, M. ; Sabel, David ; Anis, Altug: Congruence closure of compressed terms in polynomial time. In: *Frontiers of Combining Systems*, Springer, 2011, S. 227–242