Bachelorarbeit

KiTES
Eine Experimentierumgebung für Termersetzungssysteme

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl Theoretische Informatik

angefertigt von: **Sebastian Schäfer**
betreuender Hochschullehrer: Prof. Dr. Thomas Wilke

Kiel, 20. September 2010

Name, Vorname:	Schäfer, Sebastian
Immatrikulations-Nr:	888594
Studiengang:	2-Fach-Bachelor
betreuender Hochschullehrer:	Prof. Dr. Thomas Wilke
Institut:	Institut für Informatik
Arbeitsgruppe:	Theoretische Informatik
Beginn am:	20. Juli 2010
Einzureichen bis:	20. September 2010

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

.....
Sebastian Schäfer

Inhaltsverzeichnis

1. Einführung **1**

2. Terminologie und Modellierung von Termersetzungssystemen **3**
2.1. Signatur 3
2.2. Terme 4
2.2.1. Grundterme 4
2.2.2. Terme 4
2.2.3. Σ-Γ-Programmterme 5
2.3. Regeln 5
2.4. Regelsystem 6
2.5. Belegungen und Anpassungen 6
2.6. Anwendung von Regeln 7
2.7. Unifizierbarkeit 7

3. Interpretation **9**
3.1. Interpretation als Termersetzungssystem 9
3.1.1. Variablenmengen 10
3.1.2. Instanz 10
3.1.3. Reduktionen 11
3.2. Interpretation als nicht-deterministisches Programm 11
3.2.1. Σ-Γ-Programmterm 11
3.2.2. Reduktionen 11
3.3. Interpretation als Programm 12
3.3.1. Auswertungsstrategien 13

4. Sprachbeschreibung **17**
4.1. Beschreibung in EBNF 17
4.2. Beschreibung mit Railroad-Diagrammen 19

5. Repräsentation von TES im Interpreter **21**
5.1. Repräsentation eines Regelsatzes 22
5.2. Repräsentation einer Instanz 22
5.3. Gerichtete azyklische Graphen 24

6. Umwandlung in Standardform & kodierte Standardform **25**
6.1. Standardform 25

6.2. Kodierte Standardform 26

7. Umsetzung **29**
7.1. Paket `kites.exceptions` 29
7.2. Paket `kites.logic` 30
7.2.1. `CheckTRS` 30
7.2.2. `Codification` 30
7.2.3. `Decomposition` 31
7.2.4. `ProgramRewrite` 32
7.3. Paket `kites.parser` 32
7.4. Paket `kites.model` 33
7.4.1. `ASTNode` 33
7.4.2. `Constant` 33
7.4.3. `Function` 33
7.4.4. `Variable` 33
7.4.5. `Rule` 33
7.4.6. `TRSFile` 34
7.5. Paket `kites.visual` 34
7.5.1. `MainWindow` 34
7.5.2. `InterpreterWindow` 34
7.5.3. `CodificationWindow` 34
7.5.4. `StepRewrite` 35
7.5.5. `NodeContainer` 35
7.5.6. `NodeBox` 35
7.5.7. `NodeLabel` 35
7.6. Lebensweg eines TES 36

A. Mitgelieferte Regeln **39**
A.1. `append.trs` 39
A.2. `boolean.trs` 39
A.3. `comparison.trs` 40
A.4. `formula.trs` 40
A.5. `if-then-else.trs` 41
A.6. `interpretierer.trs` 41
A.7. `ndet-int.trs` 42
A.8. `maximum.trs` 43

B. Benutzungshinweise **45**

Abbildungsverzeichnis

3.1. Instanz als Beispiel für Auswertungsstrategien 13
3.2. LO-Auswertungsstrategie 14
3.3. LI-Auswertungsstrategie 15
3.4. RO-Auswertungsstrategie 16
3.5. RI-Auswertungsstrategie 16

4.1. Railroad-Diagramm für einzeilige Kommentare 19
4.2. Railroad-Diagramm für mehrzeilige Kommentare 19
4.3. Railroad-Diagramm für Funktionen 19
4.4. Railroad-Diagramm für Includes 20
4.5. Railroad-Diagramm für Identifikatoren 20
4.6. Railroad-Diagramm für Instanzen 20
4.7. Railroad-Diagramm für Regelsysteme 20
4.8. Railroad-Diagramm für Regeln 20
4.9. Railroad-Diagramm für Variablen 20

5.1. Baum nach Parsen von (5.1) und (5.2) 22
5.2. Baum nach Parsen von (5.3) 23
5.3. Kompletter Baum des Regelsatzes aus (5.1) und (5.2) 23

6.1. Term in Listendarstellung mit *cons* 27
6.2. Regelsystem in Listendarstellung mit *cons* 27

B.1. Hauptfenster mit eingegebenem Regelsystem 45
B.2. Interpretationsfenster ohne eingegebene Instanz 46
B.3. Interpretationsfenster mit Auswertung als Programm 47
B.4. Interpretationsfenster mit Auswertung als Termersetzungssystem . . 47
B.5. Ausgabe in kodierter Standardform 48

1. Einführung

Zur Einführung in die Themenbereiche „Berechenbarkeit“ und „Komplexität“ wird in der theoretischen Informatik zunächst ein Berechnungsmodell eingeführt. Häufig wird hier das Modell der Turing-Maschine verwandt, die das originäre Beispiel eines Turing-vollständigen Berechnungsmodells darstellt. Es existieren darüber hinaus aber auch andere Berechnungsmodelle, deren Turing-Vollständigkeit bewiesen wurde, zum Beispiel das Modell der Termersetzungssysteme.

Dieses Modell wird im Rahmen der Vorlesung „Theoretische Grundlagen der Informatik“ verwendet, um den Studenten eine Einführung in die oben genannten Bereiche zu geben. Da das Verstehen und Interpretieren von Termersetzungssystemen auf Papier vergleichsweise aufwendig und fehlerträchtig ist und zumindest zu Beginn auch nicht sofort einsichtig ist, entstand der Wunsch, diese Interpretationen mithilfe eines Computerprogrammes durchzuführen. Eine Umgebung, die Termersetzungssysteme auf verschiedene Arten interpretieren kann entstand im Rahmen dieser Arbeit.

Sie ermöglicht es, Termersetzungssysteme als Termersetzungssysteme (das heißt ohne, dass weitere Bedingungen erfüllt werden müssen), als nicht-deterministische Programme und als Programme zu interpretieren. Was diese Modi genau beinhalten, wird in Kapitel 3 genauer ausgeführt.

Zunächst wird jedoch noch eine kurze Einführung in die Theorie der Termersetzungssysteme gegeben, um eine Grundlage für die darauf folgenden Kapitel zu schaffen. Danach wird mit den Interpretationsmodi fortgefahren. Dieser formellastige Teil ist dann mit Kapitel 4 abgeschlossen, in dem die Grammatik zur Erkennung der vom Benutzer eingegebenen Termersetzungssysteme erläutert wird. Nachdem dann die Sprache zwischen Mensch und Maschine klar ist, wird im darauffolgenden Kapitel die interne Repräsentation der Daten in einer Baumstruktur erläutert. Neben der Basisfunktion der Interpretation von Termersetzungssystemen kann KiTES sie darüber hinaus auch noch in ihre kodierte Standardform transformieren, die es ermöglicht, sie mit einem als TES geschriebenen Interpreters zu interpretieren. Diese Transformation wird in Kapitel 6 erklärt. Zum Abschluss wird schließlich noch der Programmaufbau erklärt und ein beispielhafter Weg eines TES durch den Interpreter aufgezeichnet.

Im Anhang sind noch die bei KiTES mitgelieferten Standard-Regeln, darunter auch

der oben genannte Interpreter, sowie eine Benutzeranleitung zu finden, die den Einstieg in KiTES erleichtern soll.

2. Terminologie und Modellierung von Termersetzungssystemen

Einen kompletten Abriss über die Theorie von Termersetzungssystemen zu geben, würde den Rahmen dieser Arbeit bei weitem sprengen. Der Leser sei hier an entsprechende Lehrbücher verwiesen: [1] [2] [3]. Hier werden nur kurz die für das Verständnis notwendigen Definitionen und Lemmata wiederholt. Beweise werden hier nicht angegeben, sie können den entsprechenden Quellen entnommen werden.

2.1. Signatur

Die Terme, aus denen ein Termersetzungssystem besteht, und auf denen es operiert, bestehen aus Funktions-, Konstanten- und Variablensymbolen. Die Menge dieser Symbole bildet die Signatur eines TES. Im Gegensatz zur Literatur wird auf die strikte Typisierung der Funktionssymbole verzichtet. Da sie in der Vorlesung, für die der Interpreter konzipiert ist, nicht angesprochen wird, ist sie auch im Programmcode nicht umgesetzt und wird daher auch hier nicht betrachtet.

Die vorgenannten Symbole sind endliche Folgen von Zeichen über die in 4 definierten Alphabete [5, S. 25]. Sei also F das Alphabet, das Funktionssymbole beschreibt, K das Alphabet, welches Konstantensymbole beschreibt und V schließlich beschreibt Variablensymbole.

Sei $\Sigma = \bigcup_{i=0}^{\infty} \Sigma_i$ mit $\Sigma_0 \subseteq K^*$ und $\Sigma_{>0} \subseteq F^*$ eine endliche Menge von Funktionssymbolen. Dabei sei Σ_i die Menge der Symbole mit i Argumenten. Daher wird die Menge $\Sigma_0 \subseteq \Sigma$ der null-stelligen Funktionssymbole als Menge der Konstantensymbole bezeichnet. Σ selbst wird als Signatur des Termersetzungssystems bezeichnet [2, S. 34]. Der zu interpretierende Term eines TES, die Instanz ist ein Term über einer Signatur Σ.

2.2. Terme

Aus einer solchen Signatur können nun Terme aus Symbolen aufgebaut werden. Zuerst werden Grundterme definiert und auf Ihnen aufbauend weitere Arten von Termen, die im weiteren Verlauf der Arbeit verwendet werden.

2.2.1. Grundterme

Grundterme sind die grundlegendsten Terme und wie folgt definiert [2, S. 38]:

1. Ist $c \in \Sigma_0$, so ist c Grundterm.
2. Ist $f \in \Sigma_n$ und sind $t_0, \ldots, t_{n-1}$ Grundterme, so ist $f(t_0, \ldots, t_{n-1})$ ein Grundterm.

Die Menge aller Grundterme über einer Signatur Σ wird mit $\mathcal{T}(\Sigma)$ bezeichnet [2, S. 38]. Der zu interpretierende Term, die Instanz[1], ist ein solcher Grundterm.

Ein Beispiel für einen Grundterm ist:

$$cons(alpha, cons(beta, cons(suc(suc(zero)), empty))) \tag{2.1}$$

Die zugehörige Signatur Σ ist wie folgt aufgebaut:

$$\Sigma_0 = \{alpha, beta, zero, empty\} \tag{2.2}$$
$$\Sigma_1 = \{suc\} \tag{2.3}$$
$$\Sigma_2 = \{cons\} \tag{2.4}$$
$$\Sigma_{\geq 3} = \emptyset \tag{2.5}$$

2.2.2. Terme

In der bisherigen Definition werden nur Symbole aus Σ benutzt, die Variablensymbole wurden bisher nicht betrachtet. Dies ändert sich nun, indem die Definition der Grundterme wie folgt abgeändert wird [5, S. 26]:

1. Ist $c \in \Sigma_0 \cup X$, so ist c ein Term.

[1] im Folgenden wird ausschließlich die Bezeichnung „Instanz“ benutzt, um den zu interpretierenden Term zu bezeichnen

2. Ist $f \in \Sigma_n$ und sind $t_0, \ldots, t_{n-1}$ Terme, so ist $f(t_0, \ldots, t_{n-1})$ ein Term.

X bezeichnet dabei eine von Σ disjunkte Menge von Variablensymbolen. Die Menge aller derartigen Terme wird mit $\mathcal{T}(\Sigma \cup X)$ bezeichnet.

2.2.3. Σ-Γ-Programmterme

Ein weiterer Typus von Termen sind Σ-Γ-Programmterme. Sei dazu Γ eine von Σ disjunkte Signatur, $f \in \Gamma_n$ und $t_0, \ldots, t_{n-1} \in \Sigma$. Dann ist ein Term der Form $f(t_0, \ldots, t_{n-1})$ ein Σ-Γ-Programmterm. f wird hierbei auch als Rechensymbol bezeichnet.

2.3. Regeln

Nachdem nun der Aufbau der verschiedensten Arten von Termen möglich ist, können aus ihnen zunächst einzelne Regeln und später ganze Regelsysteme aufgebaut werden. Zunächst wird jetzt jedoch eine einzelne Regel definiert.

Eine Regel ist ein Paar (l, r), wobei l ein Term oder Σ-Γ-Programmterm ist und r ein Term. l wird als linke Regelseite, r als rechte Regelseite bezeichnet [5, S. 35]. Die Erfordernis, dass die linke Regelseite ein Σ-Γ-Programmterm ist, ist abhängig vom gewählten Interpretationsmodus. Auf die entsprechenden Anforderungen wird in Kapitel 3 eingegangen.

Alternativ zur Darstellung als Paar kann eine Regel auch in „Pfeilschreibweise“ beschrieben werden: (l, r) wird dann als $l \to r$ geschrieben. Durch diese Schreibweise wird auch die Semantik einer Regel intuitiv klar: Ein Vorkommen der linken Regelseite wird durch die rechte Regelseite ersetzt.

In der Literatur [2, S. 7] werden Regeln auch als eine binäre Relation $\to \in A \times A$ über einer Signatur A gesehen. Da in dieser Arbeit nicht näher auf die Relationseigenschaften [2, S. 8 f.] eingegangen wird, wird hier aus Gründen der Einfachheit und der genaueren Abbildung der Implementation die Definition als Paar benutzt.

2.4. Regelsystem

Werden nun mehrere Regeln $s_0, \ldots, s_{n-1}$ zu einer Folge $r = (s_0, \ldots, s_{n-1})$ zusammengefasst, so wird r als Regelsystem, Regelsatz oder auch einfach Termersetzungssystem bezeichnet. In der Regel ist es nicht sinnvoll, wenn zwei identische Regeln s_i und s_j in einem Regelsystem vorhanden sind. In der vorliegenden Umgebung wird dies nicht aktiv verhindert, vielmehr erfolgt eine Einschränkung der Interpretationsmodi. Dies gilt auch für ein mehrfaches Einbinden von externen Regeln mittels `#include`-Anweisung.

In der Literatur werden Regelsysteme auch oft als Mengen von Regeln definiert. Da das Regelsystem aber als Liste implementiert ist, entspricht eine formale Definition als Folge eher dem tatsächlichen Verhalten. Die unterschiedlichen Definitionen haben weder Auswirkungen auf die einzelnen Berechnungsschritte noch auf das Endergebnis. Im Interprationsmodus als Programm kann entweder keine oder genau eine Regel angewendet werden. In den beiden anderen Modi trifft der Benutzer die Auswahl darüber, welche der potentiell anwendbaren Regeln tatsächlich zur Anwendung kommt.

2.5. Belegungen und Anpassungen

Bei der Interpretation eines TES werden Regeln auf Grundterme angewendet und Teilterme, auf die eine Regel „passt“, durch die entsprechende rechte Regelseite ersetzt. Um eine solche Ersetzung beschreiben zu können, werden zunächst noch Belegungen von Variablen und Anpassungen betrachtet, bevor die Substitution eingeführt werden kann.

Zunächst wird die Menge aller in einem Term $t = f(t_0, \ldots, t_{n-1})$ vorkommenden Variablen, $var(t)$, induktiv definiert :

$$\forall x \in X : var(x) = \{x\} \tag{2.6}$$

$$\forall x \in \Sigma_0 : var(x) = \emptyset \tag{2.7}$$

$$\forall x = f(t_0, \ldots, t_{n-1}) \in \Sigma_n, n > 0 : var(x) = var(t_0) \cup \ldots \cup var(t_{n-1}) \tag{2.8}$$

Eine Belegung ist eine Funktion $\sigma : X \nrightarrow \mathcal{T}(\Gamma)$ mit Γ Signatur, die Variablen in einem Term einen Grundterm zuordnet [5, S. 31].

Dies kann nun erweitert werden zur Funktion $\sigma^* : \mathcal{T}(\Sigma \cup X) \to \mathcal{T}(\Sigma \cup \Gamma \cup X)$, definiert durch [2, S. 38]:

$$\forall x \in X : x \in dom(\sigma) \implies \sigma^*(x) = \sigma(x) \tag{2.9}$$

$$\forall x \in X : x \notin dom(\sigma) \implies \sigma^*(x) = x \tag{2.10}$$

$$\forall x \in \Sigma_0 : \sigma^*(x) = x \tag{2.11}$$

$$\forall t = f(t_0, \ldots, t_{n-1}) \in \mathcal{T}(\Sigma \cup X) : \sigma^*(t) = f(\sigma^*(t_0), \ldots, \sigma^*(t_{n-1})) \tag{2.12}$$

Ist t nun ein Term und u ein Grundterm, so passt t auf u, das heißt eine Substitution ist möglich, wenn es eine Belegung σ mit $dom(\sigma) = var(t)$ gibt, so dass $\sigma^*(t) = u$ gilt. Eine solche Belegung wird Anpassung genannt.

2.6. Anwendung von Regeln

Soll nun eine Regel $x = l \to r$ auf einen Grundterm t angewendet werden, muss zunächst überprüft werden, ob eine Anpassung σ^* existiert, so dass $\sigma^*(l) = t$ gilt. Ist dies der Fall, wird eine Substitution durchgeführt, so dass ein neuer Grundterm u mit $u = \sigma^*(r)$ entsteht. Eine solche Substitution wird auch Reduktion, Rechenschritt oder Schritt genannt.

Bei der Interpretation eines TES und seiner Instanz wird die Frage, welche Regel zu welchem Zeitpunkt angewendet wird, nicht völlig arbiträr beantwortet. Dies wird zum nächsten Kapitel führen, den Interpretationsmodi. Doch zunächst muss noch ein weiterer Begriff definiert werden.

2.7. Unifizierbarkeit

Die Unifizierbarkeit von Regeln ist ein für die Interpretation als Programm wichtiger Punkt, da eine automatische Durchführung sämtlicher Reduktionen nur dann möglich ist, wenn in einem Knoten höchstens eine Regel anwendbar ist. Daher wird hier kurz eine Definition für die Unifizierbarkeit von Regeln gegeben [1].

Es werden nun zwei verschiedene Regeln (s_l, s_r) und (t_l, t_r) aus einem Regelsystem betrachtet. Diese Regeln sind unifizierbar, wenn es zwei Belegungen σ und γ mit $dom(\sigma) = var(s_l)$ und $dom(\gamma) = var(t_l)$ gibt, so dass $\sigma^*(s_l) = \gamma^*(t_l)$.

Anders ausgedrückt sind zwei Regeln unifizierbar, wenn es einen Term u gibt, auf den beide Regeln anwendbar sind.

3. Interpretation

Termersetzungssysteme können auf unterschiedliche Weisen interpretiert werden, die verschiedene Anforderungen an die Eigenschaften des TES stellen. Im Folgenden wird nun, aufbauend auf der Modellierung des vorhergehenden Kapitels, auf diese Interpretationsmodi eingegangen und ihre Eigenschaften und Anforderungen an das TES erläutert.

Die hier vorgestellte Programmumgebung kann nach Angabe von Regelsätzen Instanzen dieser Termersetzungssysteme auf verschiedene Arten interpretieren, die jeweils verschiedene Anforderungen an den Aufbau der Termersetzungssysteme haben. Die Interpretationsmodi werden hier in aufsteigender Reihenfolge nach Höhe ihrer Anforderungen behandelt. Da die Anforderungen der Modi aufeinander aufbauend sind, ergibt sich dadurch eine natürliche Ordnung für dieses Kapitel.

Der Abschnitt eines einzelnen Modus ist so aufgebaut, dass zunächst Anforderungen an das Regelsystem erläutert werden, dann gegebenenfalls Anforderungen an den zu interpretierenden Term, die Instanz. Zum Abschluss wird noch auf die mögliche Auswertungsreihenfolge eingegangen, also wie entschieden wird, welche Substitution oder Reduktion in einem Schritt angewendet werden kann. Die Auswertungsreihenfolge spielt insbesondere bei der Interpretation als Programm (Abschnitt 3.3) eine wichtige Rolle.

Nach den drei Modi wird noch auf die eigentliche Substitution eingegangen. Im vorhergehenden theoretischen Kapitel wurde dies nur sehr kurz am Rande in einer Formel erwähnt. Hier soll sich dies ändern und erklärt werden, wie die Substitution in KiTES durchgeführt wird.

3.1. Interpretation als Termersetzungssystem

Die Interpretation als Termersetzungssystem stellt die geringsten Anforderungen an das zu interpretierende TES:

3.1.1. Variablenmengen

Bestehe ein Regelsystem aus Regeln der Form $l_i \rightarrow r_i$ mit $0 \leq i < n$ mit $n =$ Anzahl der Regeln im Regelsystem. Seien weiterhin alle $l_i, r_i \subseteq \mathcal{T}(\Sigma \cup X)$, also Terme. Bezeichne schließlich $var(t)$ mit $t \subseteq \mathcal{T}(\Sigma \cup X)$ die Menge der in t vorkommenden Variablen.

So muss ein zu interpretierendes Termersetzungssystem für alle i folgende Bedingung erfüllen [5, S.35]:

$$var(r_i) \subseteq var(l_i) \tag{3.1}$$

Die Notwendigkeit von 3.1 ist offensichtlich: Auf der rechten Seite dürfen nur Variablen verwendet werden, die auch links belegt worden sind. Zu beachten ist, dass damit auch eine Regel $l_i \subseteq X$ möglich ist[1]. Eine solche Regel ist im Allgemeinen allerdings nicht sinnvoll, da sie auf jeden beliebigen Term anwendbar ist und mit jeder beliebigen anderen Regel unifizierbar ist. Weiterhin wird die Häufigkeit des Auftretens einer Variablen $x \in X$ auf einer linken Regelseite l_i nicht beschränkt. x kann mehrfach in l_i auftreten und ist dabei natürlich mit demselben Wert zu belegen. Ein Term, in dem keine Variable mehr als ein Mal vorhanden ist, wird linear genannt [5, S. 26], eine Regel, bei der dies für die linke Regelseite gilt, wird links-linear genannt.

Ohne die Forderung der Linkslinearität lässt sich beispielsweise mit einer Regel

$$equal(X, X) \rightarrow true \tag{3.2}$$

die Gleichheit zweier Terme feststellen.

3.1.2. Instanz

Für Instanzen gelten bei der Interpretation als Termersetzungssystem keinerlei Einschränkungen.

[1] In der Definition in [5, S. 35] ist dies zwar ausgeschlossen, diese Einschränkung soll aber hier nicht gelten

3.1.3. Reduktionen

Bei der Interpretation als Termersetzungssystem gibt es keine Einschränkungen bezüglich der durchführbaren Reduktionen. Der Interpreter prüft für alle Regeln alle Stellen in der Instanz, ob die jeweilige Reduktion anwendbar ist und stellt dem Benutzer alle anwendbaren Regeln zur Auswahl zur Verfügung.

3.2. Interpretation als nicht-deterministisches Programm

Soll ein TES als nicht-deterministisches Programm interpretiert werden, so muss es die Anforderungen an die Interpretation als TES (3.1) sowie die Folgenden erfüllen.

3.2.1. Σ-Γ-Programmterm

Alle linken Regelseiten l_i im Termersetzungssystem müssen, zusätzlich zu den Anforderungen an die Interpretation als Termersetzungssystem, Σ-Γ-Programmterme sein, wie in 2.2.3 beschrieben.

Dies bedeutet, dass alle linken Regelseiten immer mit einem Rechensymbol beginnen müssen und auch nur ein einziges Rechensymbol beinhalten. Diese Einschränkung gilt aber nicht für die zugehörigen Instanzen. Andernfalls wären Instanzen wie die folgende nicht zulässig, die aber durchaus sinnvoll sein können:

$$append(cons(\alpha, empty), append(append(cons(\beta, empty), cons(\delta, cons(\gamma, empty))), cons(\epsilon, empty))) \quad (3.3)$$

3.2.2. Reduktionen

Im Unterschied zur Interpretation als TES sind die Reduktionsmöglichkeiten bei nicht-deterministischen Programmsystemen deutlich beschränkt. Es sind nicht mehr alle, wie in 3.1.3 beschriebenen, Reduktionen erlaubt, sondern nur noch die am weitesten außen liegenden. „Am weitesten außen“ bezeichnet hierbei die im Baum am nächsten zur Wurzel gelegenen Knoten. Wurde ein Knoten gefunden, bei dem eine Reduktion möglich ist, so werden seine Kindelemente nicht mehr überprüft. Die möglichen Reduktionen werden wieder dem Benutzer zur Auswahl übergeben.

3.3. Interpretation als Programm

Die Interpretation als Programm stellt die höchsten Anforderungen an ein TES. So müssen alle Anforderungen für die Interpretation als reines TES, als nicht-deterministisches Programm sowie noch eine weitere Eigenschaft erfüllt sein: Die linken Regelseiten des TES dürfen paarweise nicht unifizierbar sein. Dies ist die Umkehrung des Begriffes der Unifizierbarkeit aus 2.7. Formal ausgedrückt bedeutet dies[2]:

Sei Ω die Menge aller linken Regelseiten eines Termersetzungssystems, σ eine Substitution. Gilt

$$\forall s, t \in \Omega : s \neq t \rightarrow \neg(\exists \sigma : s\sigma = t\sigma) \tag{3.4}$$

so sind alle Regeln des TES paarweise nicht unifizierbar.

Eine weitere, geforderte Eigenschaft ist die Links-Linearität der Regeln. Termersetzungssysteme, die diese beiden Anforderungen zusätzlich zu den bisherigen erfüllt, wird auch orthogonales TES genannt [6, S. 88]. Diese beiden Eigenschaften zusammen bewirken die Konfluenz des TES [3, S. 54], eine wichtige Eigenschaft von Termersetzungssystemen. Sollte eines der beiden Kriterien nicht erfüllt sein, so kann das TES nicht-konfluent sein. Im Falle der Nicht-Unifizierbarkeit kann allerdings auch eine schwächere Bedingung gelten und die wichtigsten Erkenntnisse sind noch immer gültig. Diese schwächere Bedingung ist die, dass nur *trivial critical pairs* existieren dürfen. Dies ist bei zwei Regeln (s_l, s_r), (t_l, t_r) gegeben, für die gilt [6, S. 89]:

$$s_l = t_l \tag{3.5}$$
$$s_r = t_r \tag{3.6}$$

KiTES nutzt dennoch die stärkere Eigenschaft, dass überhaupt keine *critical pairs* existieren dürfen, da dies in der Vorlesung, für die KiTES genutzt werden soll, ebenfalls so gefordert ist.

Termersetzungssysteme, die die Anforderungen an die Interpretation als Programm erfüllen, werden auch orthogonale Termersetzungssysteme genannt [6, S. 88]. Zwar wird für diese nach [6] noch gefordert, dass die

[2] Es gibt mehrere Möglichkeiten diese Eigenschaft auszudrücken. Für eine Übersicht siehe [3, S. 75].

$$
\begin{aligned}
cons(append(append(&cons(\alpha,\ empty),\\
&cons(\beta,\ empty)),\\
&cons(\delta,\ empty)),\\
append(&cons(\gamma,\ empty)),\\
&append(cons(\epsilon,\ empty),\\
&cons(\zeta,\ empty)))
\end{aligned}
$$

Abbildung 3.1.: Instanz als Beispiel für Auswertungsstrategien

Zu beachten ist, dass auch hier nicht explizit ausgeschlossen ist, dass eine linke Regelseite ausschließlich aus einem Variablensymbol besteht. Eine solche verstößt aber offensichtlich gegen die Anforderung der Nicht-Unifizierbarkeit und verhindert damit die Interpretation als Programm.

3.3.1. Auswertungsstrategien

Bei der Interpretation muss eine feste Auswertungsstrategie gewählt werden. Der Benutzer hat hier keine Auswahlmöglichkeit, welche Reduktion in welchem Schritt angewendet werden soll. Aufgrund der Nicht-Unifizierbarkeit der Regeln ist bei jedem Knoten prinzipiell entweder keine oder genau eine Reduktion möglich. Die Suchstrategie nach der der Baum nach möglichen Reduktionen durchsucht wird, bestimmt einen ersten Knoten, bei dem eine Regel anwendbar ist. Diese Reduktion muss dann durchgeführt werden. Sollte bei keinem Knoten mehr eine Reduktion möglich sein, ist die Auswertung abgeschlossen.

Es gibt eine Reihe verschiedener Strategien, von denen hier nur über eine kleine Auswahl gesprochen wird. Eine Übersicht über diese und andere Strategien nebst ihren Eigenschaften ist in [6, S. 129 ff.] und [8] zu finden.

Die feste Auswertungsstrategie und das Fehlen von Auswahlmöglichkeiten ermöglicht damit ein automatisches Auswerten der Instanz bis zu ihrem Endzustand, was bei den beiden anderen Interpretationsmodi nicht möglich ist. Im Nachfolgenden werden die unterschiedlichen Auswertungsstrategien erklärt. Als Beispiel wird das aus Kapitel 5 bekannte Regelsystem mit der Instanz aus Abbildung 3.1 verwendet. Es werden immer Bäume dieser Instanz gezeigt, bei denen der Knoten, an dem die Reduktion stattfindet, fett gedruckt ist.

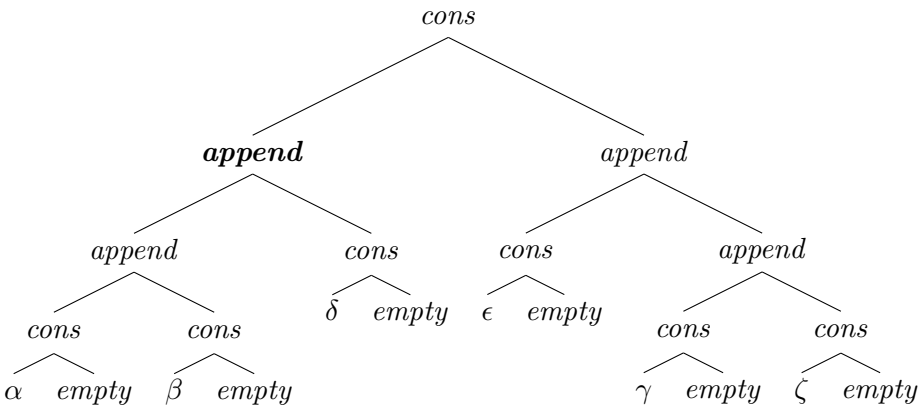

Abbildung 3.2.: LO-Auswertungsstrategie

Leftmost-outermost

Die LO-Strategie ist die, für den Einsatzzweck des Interpreters als unterstützendes Medium bei der Vorlesung „Theoretische Grundlagen der Informatik“, relevanteste Auswertungsstrategie, da sie dort als einzige Strategie verwendet wird. Ihre Wichtigkeit ist in der Tatsache begründet, dass diese Strategie *(hyper-)(head-) normalizing* ist, dies gilt allerdings nicht für schwach orthogonale Programmsysteme [8, S. 513].

Bei dieser Strategie wird zuerst für den Wurzelknoten überprüft, ob eine Reduktion möglich ist. Ist dies der Fall wird die Suche beendet und die Reduktion hier durchgeführt. Andernfalls werden die Kindelemente des Knotens von links nach rechts untersucht, die wiederum zuerst überprüfen, ob eine Reduktion möglich ist und andernfalls wiederum ihre Kinder von links nach rechts untersuchen, etc. Die LO-Strategie sucht also erst in der Breite, dann in der Tiefe nach möglichen Reduktionen.

Für die Beispielinstanz ergibt die LO-Strategie den Baum in Abbildung 3.2.

Sofern alle Regeln eines Programmsystems links-normal sind, das heißt, dass in dem die linke Regelseite repräsentierenden Term alle Konstanten- und Funktionssymbole links von den Variablensymbolen stehen, wird mit dieser Strategie die Normalform der Instanz erreicht [6, S. 133]. Die Normalform eines Termes ist ein Term, der aus diesem durch Reduktionen entsteht, in dem keine weiteren Reduktionen mehr durchführbar sind, egal mit welcher Auswertungsstrategie. Ist das System nicht links-normal, so kann keine allgemeingültige Aussage über das Normalisierungsverhalten der LO-Strategie getroffen werden [6, S. 136].

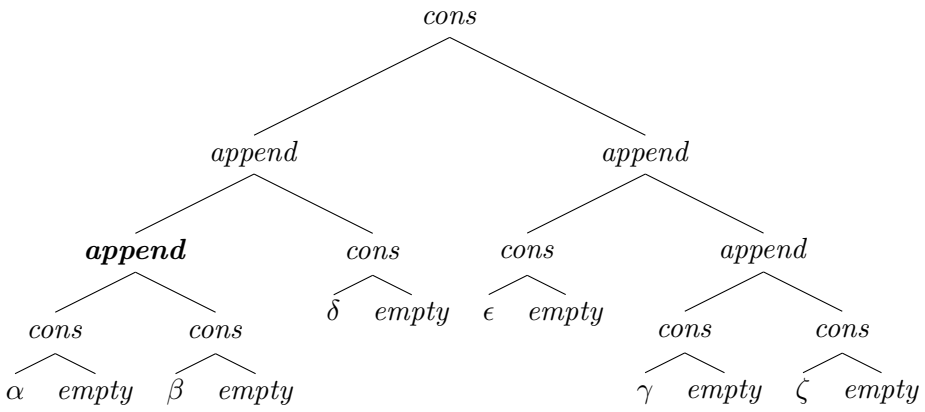

Abbildung 3.3.: LI-Auswertungsstrategie

Leftmost-innermost

Die LI-Strategie ist der LO-Strategie ähnlich, mit dem Unterschied, dass bei einem Knoten zunächst überprüft wird, ob auf das am weitesten links liegende Kind eine Reduktion angewendet werden kann. Danach wird überprüft, ob auf den Knoten selbst eine Regel angewendet werden kann. Sollte auch dies nicht der Fall sein, werden alle restlichen Kindsknoten, also bis auf den am weitesten links liegenden überprüft. Die LI-Strategie sucht also im Gegensatz zur LO-Strategie zuerst in die Tiefe und danach in die Breite.

Siehe dazu den Baum in Abbildung 3.3.

Im Gegensatz zur LO-Strategie ist die LI-Strategie nicht normalisierend, führt also nicht zur Normalform. Im Gegenteil kann sogar gezeigt werden, dass die LI-Strategie sowohl für nicht-links-normale als auch links-normale Systeme eine unaufhörliche Strategie ist und es vermeidet zu einer Normalform zu gelangen, sofern dies nicht unausweichlich ist.

Rightmost-outermost

Die RO-Auswertungsstrategie ist das symmetrische Gegenstück zur LO-Zerlegung. Die Kinder werden statt von links nach rechts von rechts nach links überprüft. Siehe dazu den Baum in Abbildung 3.4.

Aufgrund der Symmetrie zur LO-Auswertung besitzt sie ebenfalls deren normalisierende Eigenschaft, mit dem Unterschied, dass hier alle Konstanten- und Funktionssymbole rechts von den Variablensymbolen stehen müssen.

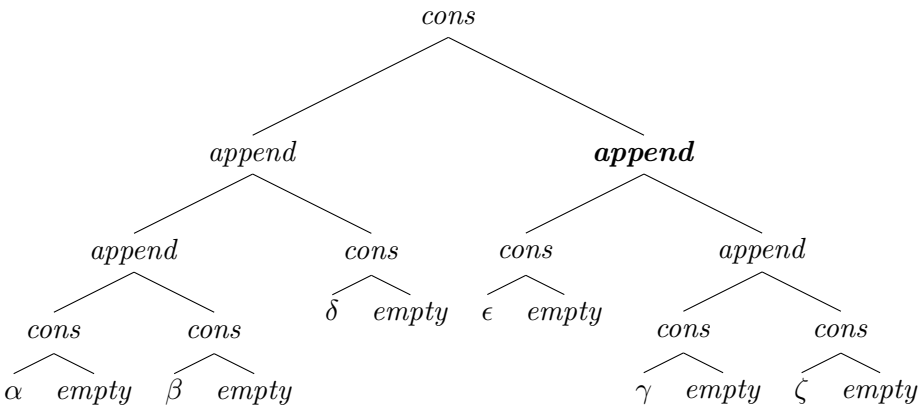

Abbildung 3.4.: RO-Auswertungsstrategie

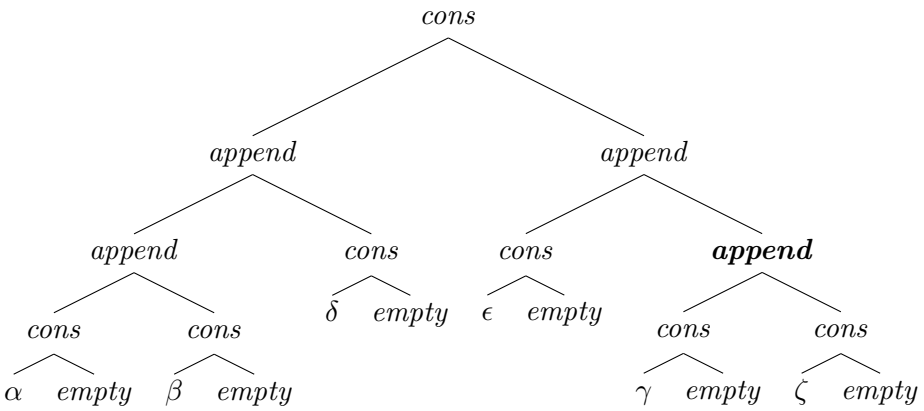

Abbildung 3.5.: RI-Auswertungsstrategie

Rightmost-innermost

Die RI-Auswertung wiederum ist analog zur RO-Auswertung das symmetrische Gegenstück zur LI-Auswertung, die Kindelemente werden von rechts nach links durchlaufen. Wie der LI-Strategie fehlt der RI-Strategie die normalisierende Eigenschaft, stattdessen ist sie eine unaufhörliche Strategie.

Siehe dazu den Baum in Abbildung 3.5.

4. Sprachbeschreibung

Nachdem der theoretische Aufbau von Termersetzungssystemen nun klar ist, kann eine Grammatik zur Erkennung von Regelsystemen und ihrer Instanzen entwickelt werden. Diese Grammatik wird automatisch durch den Parser-Generator ANTLR [7] in einen LL*-Parser in Java umgewandelt.

4.1. Beschreibung in EBNF

```
rulelist = { (rule | INCLUDE) }, "#instance", instance ;

instance = (function | constant) ;

INCLUDE = '#include' {WS} FILENAME ;

rule = (var | constant  | function)
       RARROW
       (var | constant | function) ;

function = IDENT LPAR (var | constant | function)
           { (var | constant | function) } RPAR ;

constant = IDENT ;

var = VAR ;

IDENT = 'a'..'z' { ('a'..'z' | 'A'..'Z' | '0'..'9') } ;

VAR = 'A'..'Z' { ('a'..'z' | 'A'..'Z' | '0'..'9') } ;

FILENAME = ('a'..'z' | 'A'..'Z' | '0'..'9' | '.' | '-' | '_' | '/'
           | '\' | '~' | '+' | '*')
           { ( 'a'..'z' | 'A'..'Z' | '0'..'9' | '.' | '-' | '_' | '/'
             | '\' | '~' | '+' | '*') } ;

WS = (' ' | ',' | '\n' | '\r' | '\t' | '\f')
     { (' ' | ',' | '\n' | '\r' | '\t' | '\f') } ;

LPAR = '(' ;
```

```
RPAR = ')' ;

RARROW = '-->' ;
```

Die unterschiedliche Schreibweise der Bezeichner spiegelt die ANTLR-Syntax wider. Dort wird zwischen Lexer-Regeln und Parser-Regeln unterschieden, wobei Lexer-Regeln durch Großbuchstaben gekennzeichnet werden, Parser-Regeln durch Kleinbuchstaben. Dies bietet Vorteile, da so zum Beispiel Whitespace direkt im Lexer ausgeblendet werden kann und nicht vom Parser interpretiert werden muss. Ebenso findet die Einbindung von Include-Dateien auf der Lexer-Ebene statt. Für den Parser entsteht so der Eindruck, dass die eingebundenen Regeln direkt im ursprünglichen Quelltext stehen. Sie werden damit an korrekter Position in derselben `rulelist`-Regel erkannt. Im Unterschied zu der EBNF-Darstellung enthält die eigentliche ANTLR-Grammatik `TRS.g` auch noch zusätzliche Funktionen zum Erzeugen der programminternen Repräsentation.

Zusätzlich zu den oben angegebenen Produktionen existieren noch zwei weitere, die Kommentare beschreiben[1]. Diese wurden hier nicht explizit angegeben, da sie für die Erkennung des relevanten Teils der Grammatik unerheblich sind und auch während eines Interpreter-Laufs keine Rolle spielen, da sie schon während des Parsings ausgeblendet werden[2].

Das Parsing des Regelsystems eines Termersetzungssystems beginnt immer mit der Produktion `rulelist`, das Parsing einer Instanz mit der Produktion `instance`. Besonders herauszustellen ist auch die Unterscheidung zwischen Konstanten und Funktionen, die rein formal nicht notwendig ist, da Konstanten im Allgemeinen als Funktionen ohne Parameter angesehen werden. In der Grammatik ist die Unterscheidung allerdings notwendig, da sonst auf eine Konstante eine leere Klammerung folgen müsste, was der Vereinbarung der Vorlesung und auch der einschlägigen Literatur widersprechen würde. Diese Unterscheidung wird auch in der programminternen Repräsentation fortgeführt.

Zu beachten ist auch, dass in der gegebenen Grammatik Funktions- und Konstantenbezeichner immer mit Kleinbuchstaben beginnen, wohingegen Variablenbezeichner immer mit einem Großbuchstaben beginnen müssen. Diese Unterscheidung ist wiederum für eine korrekte programminterne Repräsentation notwendig, ansonsten müsste vor dem Parsing bereits eine Signatur vom Benutzer angegeben werden, was den Gebrauch des Interpreters unnötigerweise verkomplizieren und zeitaufwendiger

[1] In der Form von C++-Kommentaren, also `//` und `/* */` für ein- beziehungsweise mehrzeilige Kommentare

[2] Siehe dazu den Quellcode von `TRS.g`

machen würde.

4.2. Beschreibung mit Railroad-Diagrammen

Um auch eine grafische Repräsentation des Parsing-Prozesses zu liefern, die ein einfacheres Verständnis ermöglicht als die formale Darstellung in EBNF, werden hier zusätzlich für die wichtigsten Regeln Railroad-Diagramme abgebildet. Die Diagramme wurden über das ANTLR-Eclipse-Plugin [4] direkt aus der ANTLR-Grammatik heraus erzeugt. Ihre Äquivalenz zur Beschreibung in EBNF-Darstellung ist offensichtlich und lässt sich durch einen einfachen Vergleich der einzelnen Regeln schnell feststellen.

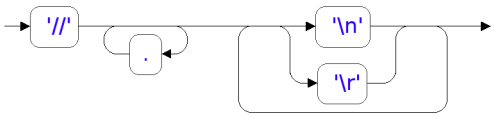

Abbildung 4.1.: Railroad-Diagramm für einzeilige Kommentare

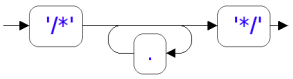

Abbildung 4.2.: Railroad-Diagramm für mehrzeilige Kommentare

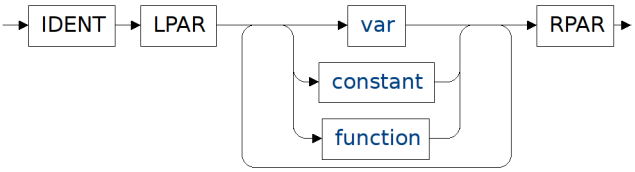

Abbildung 4.3.: Railroad-Diagramm für Funktionen

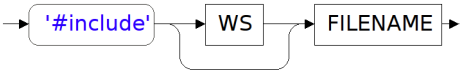

Abbildung 4.4.: Railroad-Diagramm für Includes

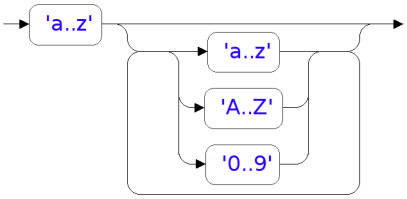

Abbildung 4.5.: Railroad-Diagramm für Identifikatoren

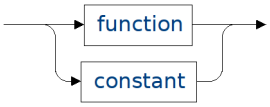

Abbildung 4.6.: Railroad-Diagramm für Instanzen

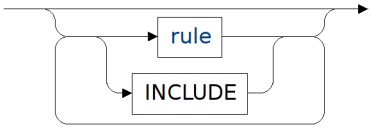

Abbildung 4.7.: Railroad-Diagramm für Regelsysteme

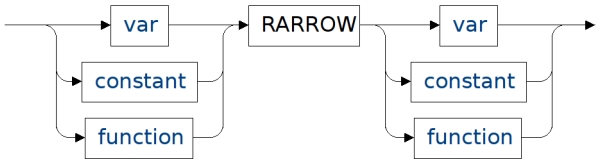

Abbildung 4.8.: Railroad-Diagramm für Regeln

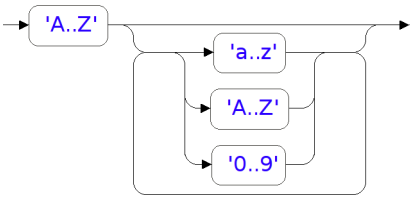

Abbildung 4.9.: Railroad-Diagramm für Variablen

5. Repräsentation von TES im Interpreter

Der Parser wandelt das vom Benutzer angegebene Termersetzungssystem um in eine programminterne Repräsentation der Daten. Hierzu wird das TES in mehrere Bäume zerteilt:

- die `RuleList`: Sie enthält die gesamten Regeln als Kindobjekte vom Typ `Rule`.
- die `Rule`s: Jede Regel im TES wird als ein solches Objekt repräsentiert und enthält jeweils die linke und rechte Regelseite in Form von `ASTNode`s. Ein Objekt diesen Typs besitzt immer genau zwei Kindobjekte[1].
- die `ASTNode`s: Sie sind ein Interface und ermöglichen den generischen Zugriff auf einen Baum, der aus `Function`s, `Constant`s und `Variable`s besteht.
- `Function`s: Sie repräsentieren die in Abschnitt 4.1 beschriebenen Funktionssymbole mit mindestens einem Argument.
- `Constant`s: Sie repräsentieren die beschriebenen Funktionssymbole ohne Argumente, also Konstantensymbole.
- `Variable`s: Sie repräsentieren die nur in Regeln vorkommenden Variablen, die bei der darauffolgenden Interpretation als Platzhalter dienen.

Zur Veranschaulichung der internen Repräsentation wird nun ein Beispiel der Repräsentation eines konkreten, einfachen Termersetzungssystems gegeben. Es wird dazu folgendes TES benutzt:

$$append(cons(X, XS), Y) \rightarrow cons(X, append(XS, Y)) \tag{5.1}$$
$$append(empty, Y) \rightarrow Y \tag{5.2}$$

Es dient der Verkettung zweier mit *cons* konstruierter Listen. Eine mögliche Instanz wäre beispielsweise:

[1] Diese sind über die Methoden `Rule.getLeft()` und `Rule.getRight()` zugänglich.

$$append(cons(\alpha, cons(\beta, empty)), cons(\delta, cons(\gamma, empty))) \tag{5.3}$$

wobei α, β, δ und γ Konstantensymbole darstellen.

5.1. Repräsentation eines Regelsatzes

Beim Parsen von (5.1) und (5.2) wird nun zunächst ein `RuleList`-Objekt erzeugt. Die Erkennung schreitet fort und erzeugt dann ein neues `Rule`-Objekt und fügt dieses als Kind zum `RuleList`-Objekt hinzu. Das `Rule`-Objekt, das (5.1) repräsentiert, erhält nun seinerseits zwei Kindobjekte vom Typ `ASTNode`, nämlich Bäume der linken und rechten Regelseiten. Wie diese Bäume entstehen, wird im nachfolgenden Abschnitt 5.2 dargestellt.

Es entsteht folgende Baumstruktur, wobei Φ und ϕ die linke beziehungsweise rechte Regelseite von (5.1), und Ψ und ψ die jeweiligen Regelseiten von (5.2) repräsentieren:

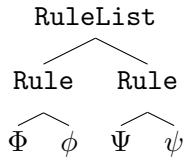

Abbildung 5.1.: Baum nach Parsen von (5.1) und (5.2)

5.2. Repräsentation einer Instanz

Eine Instanz wird ähnlich geparst, mit dem hauptsächlichen Unterschied, dass der Startzustand des zugrundeliegenden Automaten nicht die `rulelist`-Regel, sondern die `instance`-Regel ist. Der einzige Objekttyp, der hier beliebig viele Kindobjekte besitzen kann, ist die `Function`. `Constant`s und `Variable`s, die beiden anderen beim Parsing von Instanzen entstehenden Objekttypen, besitzen beide keine Kindobjekte.

Die Instanz wird, wie in Kapitel 4 beschrieben und oben aufgeführt, in die entsprechenden Objekte umgewandelt und die Argumente einer Funktion dem sie repräsentierenden Objekt als Kindobjekte hinzugefügt. Aus der Beispiel-Instanz (5.3) entsteht der Baum aus Abbildung 5.2.

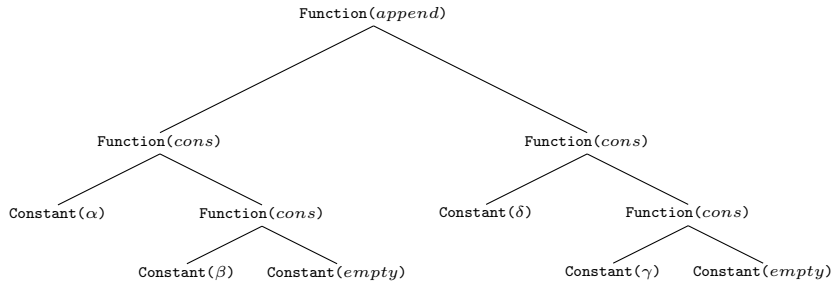

Abbildung 5.2.: Baum nach Parsen von (5.3)

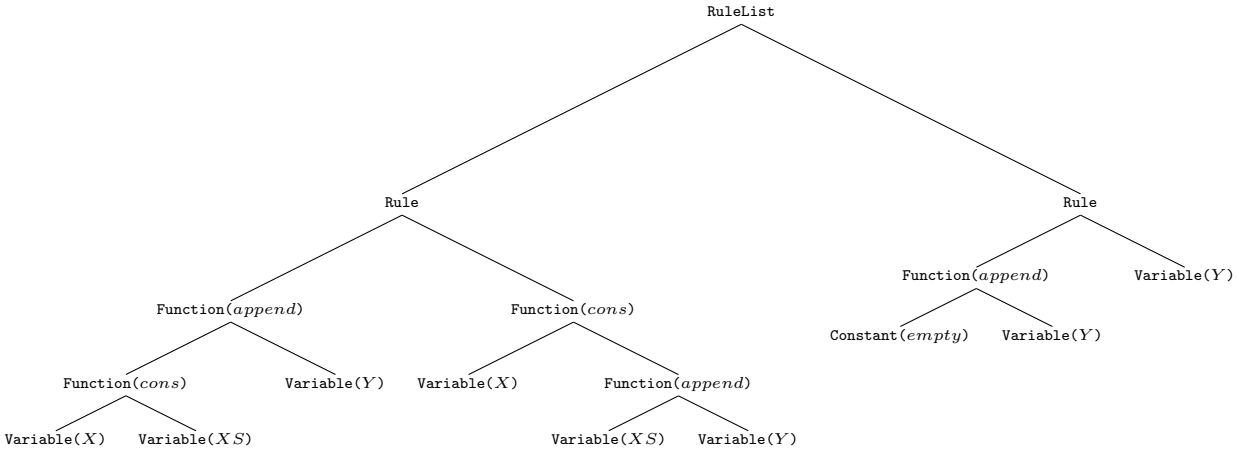

Abbildung 5.3.: Kompletter Baum des Regelsatzes aus (5.1) und (5.2)

Nun kann auch der komplette Baum des Regelsatzes angegeben werden (Abb. 5.3).

Alle vom Interpreter durchgeführten Reduktionen werden auf den so entstandenen Bäumen ausgeführt; auch die Ausgabe der Berechnungsschritte erfolgt mit dieser Baumrepräsentation. Während der Interpretation werden die Bäume allerdings nicht verändert, sondern bei jedem Berechnungsschritt wird ein neuer Baum erstellt. Javas Implementierungen von Listen erschweren Veränderungen bereits gespeicherter Objekte, da nicht immer Referenzen auf ein Objekt, sondern in manchen Fällen auch Kopien der Objekte zurückgeliefert werden. Daher stellte es sich als einfacher heraus, neue Bäume zu erstellen. Da die Termersetzungssysteme, die in der Vorlesung behandelt werden, nicht sehr viele Elemente beinhalten, sind durch dieses Vorgehen auch keine Performance-Probleme zu erwarten. Nichtsdestotrotz wäre eine Neuimplementierung der Substitution in einer späteren Programmversion wünschenswert.

5.3. Gerichtete azyklische Graphen

Auch wenn in der Literatur [3] aus Effizienzgründen zum Einsatz von DAGs[2] geraten wird, in denen Knoten, insbesondere solche, die Variablen repräsentieren, mehrfach im Baum referenziert werden, wurden diese hier nicht verwendet. DAGs bieten Vorteile, beispielsweise bei Substitutionen. Hier kann nach dem ersten Fund einer Variablen und ihrer Ersetzung der Algorithmus abgebrochen werden, da damit auch alle übrigen Referenzen aktualisiert worden sind. Ein anderer Vorteil liegt in der Speicherbedarfsersparnis durch die Mehrfachreferenzierung.

Hier wurde aber auf ihren Einsatz verzichtet. Der Grund für den Verzicht liegt in der intuitiveren Implementierung von normalen Bäumen und der auf ihnen basierenden Algorithmen. Da der Einsatzzweck des Interpreters sowieso auf kleine TES, d. h. mit nur wenigen Regeln, mit vergleichsweise wenigen Reduktionsschritten abzielt, stellt auch die besonders effiziente Speichernutzung kein herausragend wichtiges Implementierungsziel dar.

[2] engl. *directed acyclic graph*

6. Umwandlung in Standardform & kodierte Standardform

Aufgrund der Turing-Vollständigkeit von Termersetzungssystemen ist es möglich, ein TES zu erstellen, das wiederum ein anderes TES mit disjunkter Signatur und eine Instanz dessen als Eingabe erhalten kann, und dann dieses TES interpretiert. Mit anderen Worten: ein Interpreter für TES als TES geschrieben.

In der Vorlesung für die KiTES entwickelt wurde, wird auch tatsächlich ein solcher Interpreter geschrieben. Als Eingabe benötigt dieser das TES und die Signatur in einer bestimmten Form, die als kodierte Standardform bezeichnet wird. Diese Kodierung kann auch von KiTES erzeugt werden.

Zuerst ist dazu zu klären, was genau die Standardform ist, danach noch ihre weitere Kodierung zur kodierten Standardform.

6.1. Standardform

Bei der Umwandlung eines TES in seine Standardform werden die Bezeichner von Funktions-, Konstanten- und Variablensymbolen geändert in „generischere“ Namen. So werden die Bezeichner von Funktions- und Konstantensymbolen in f_{Zahl} umgewandelt, die Bezeichner von Variablensymbolen in x_{Zahl}.

Der Index von f beziehungsweise x ist dabei eine beliebige Zahl, allerdings dürfen keine zwei verschiedenen Symbole mit demselben Index existieren. Für Funktions- und Konstantensymbole gilt dies über alle Regeln des TES hinweg, für Variablensymbole ist dies nicht notwendig, hier kann für jede Regel die Nummerierung erneut beginnen. In der Implementation wird tatsächlich über alle Regeln des TES eine Nummerierung für Variablensymbole verwendet. In aller Regel werden die Symbole einfach aufsteigend durchnummeriert.

Als Beispiel dient folgendes TES:

$$append(cons(X, XS), Y) \rightarrow cons(X, append(XS, Y)) \tag{6.1}$$
$$append(empty, Y) \rightarrow Y \tag{6.2}$$

Durch die Umwandlung in die Standardform entsteht folgendes TES:

$$f_0(f_1(x_0, x_1), x_2) \rightarrow f_1(x_0, f_0(x_1, x_2)) \tag{6.3}$$
$$f_0(f_2, x_0) \rightarrow x_0 \tag{6.4}$$

Dies ist schon eine erste Annäherung an die Darstellung, die als Eingabe für den Interpretierer notwendig ist.

6.2. Kodierte Standardform

Um vom Interpretierer verarbeitet werden zu können, ist diese Form noch nicht ausreichend. Die Indizes werden dazu in eine einfache Zahlendarstellung umgewandelt. Es werden dazu zwei Symbole verwendet: *suc* und *zero*, wobei *suc* ein Argument aufnimmt, *zero* ist ein Konstantensymbol. *zero* bezeichnet die Zahl Null, $suc(x)$ die Zahl $x + 1$.

Später werden f und x durch neue Funktionen fun und var mit je einem Parameter, nämlich dem Index in *suc*-Schreibweise, ersetzt. Hier wird jedoch zunächst aus Gründen der Lesbarkeit die Darstellung mit Indizes verwendet.

Doch diese Umwandlungen reichen noch immer nicht aus. Darüber hinaus müssen auch alle Terme neu als Listen aufgebaut werden. Dies geschieht über die bekannte Listenkonstruktion mit *cons*.

Zunächst wird dazu der Term in eine Listendarstellung umgewandelt, hier werden dazu zunächst die Zeichen [und] verwendet, um den Anfang respektive das Ende einer Liste zu kennzeichnen. Die linke Regelseite der ersten Regel des oben genannten TES wird dann zu folgender Liste:

$$[f_0, [f_1, [x_0], [x_1]], [x_2]]] \tag{6.5}$$

$$
\begin{array}{l}
cons(f_0,\\
\quad cons(cons(f_1,\\
\qquad cons(\ cons(\ x_0,\\
\qquad\qquad\quad empty)\\
\qquad\qquad cons(\ cons(\ x_1,\\
\qquad\qquad\qquad\quad empty),\\
\qquad\qquad\quad empty))),\\
\qquad cons(cons(\ x_2,\\
\qquad\qquad empty),\\
\qquad\ empty)))
\end{array}
$$

Abbildung 6.1.: Term in Listendarstellung mit *cons*

$$
\begin{array}{l}
cons(cons(l_0),\\
\qquad r_0),\\
\quad cons(cons(l_1,\\
\qquad\quad r_1),\\
\quad \ldots\\
\qquad\quad cons(cons(\ \ l_{n-1},\\
\qquad\qquad\quad r_{n-1}),\\
\qquad\qquad empty)
\end{array}
$$

Abbildung 6.2.: Regelsystem in Listendarstellung mit *cons*

Diese Liste wird anschließend in eine Listendarstellung mit *cons* umgewandelt, das Ergebnis ist in Abbildung 6.1 dargestellt.

Nachdem nun klar ist, wie ein einzelner Term in die kodierte Standardform umgewandelt wird, muss noch geklärt werden, wie ein ganzes Regelsystem in die kodierte Standardform umgewandelt werden kann.

Es wird nun ein Regelsystem der Form $((l_0, r_0), \ldots, (l_{n-1}, r_{n-1}))$ mit n Regeln betrachtet. Die Kodierung der l_i und r_i ist klar, diese werden wie oben ausgeführt als Terme kodiert. Das Zusammenfügen dieser Listen in eine Liste funktioniert wieder über die Darstellung als Liste:

$$[[l_0, r_0], \ldots, [l_{n-1}, r_{n-1}]] \tag{6.6}$$

In Darstellung mit *cons* ergibt dies den in Abbildung 6.2 angegebenen Term.

Ein derartig kodiertes Regelsystem kann nun zusammen mit einer ebenfalls kodierten Instanz an den Interpreter übergeben und dann interpretiert werden. Der Interpreter ist in Abschnitt A.6 abgedruckt.

7. Umsetzung

Zum Abschluss der Arbeit wird nun die konkrete Umsetzung der bisher erläuterten Funktionen der Programmumgebung erklärt. In diesem Kapitel wird jedoch nicht genau auf jede einzelne Methode jeder Klasse eingegangen, dafür sei auf das, auf CD beiliegende, JavaDoc verwiesen, das detaillierte Informationen enthält. Das Ziel dieses Kapitels ist es, das Zusammenspiel der einzelnen Komponenten und ihre grobe Funktionsweise zu erklären.

Dazu wird zunächst die Aufteilung in Java-Packages erläutert, dann eine kurze Inhaltsangabe der Java-Klassen angegeben. Abschließend wird ein beispielhafter „Lebensweg“ eines TES durch die Umgebung nachvollzogen, um den Programmablauf und das Zusammenspiel der einzelnen Klassen und Methoden genauer darzustellen. Doch nun zunächst die Erklärung der Paketstruktur: KiTES besteht aus fünf Java-Package, die nach Funktionsbereichen aufgeteilt sind.

7.1. Paket `kites.exceptions`

Im Paket `kites.exceptions` sind alle Exceptions zusammengefasst, die von KiTES benutzt werden. Sie sind alle implementationslos und erben ihre gesamte Funktionalität von `java.lang.Exception`. Sie dienen der reinen Unterscheidbarkeit von anderen Exceptions.

Die derzeit benutzten Exceptions sind:

- `DecompositionException`: Sie wird ausgelöst, wenn ein Fehler bei der Suche nach einer möglichen Reduktion auftritt.
- `NoChildrenException`: Sie wird ausgelöst, wenn die Kinder eines Knotens abgefragt werden, dieser aber keine hat.
- `NodeException`: Stellt einen allgemeinen Fehler in einem Knoten dar.
- `NoRewritePossibleException`: Wird ausgelöst, wenn keine weitere Reduktion

mehr möglich ist.

- `SyntaxErrorException`: Wird ausgelöst, wenn ein Syntaxfehler im Regelsystem, oder in einem Term gefunden wird oder während der Interpretation auftritt.

- `UnificationException`: Wird ausgelöst, wenn festgestellt wird, dass zwei linke Regelseiten unifizierbar sind.

7.2. Paket `kites.logic`

In diesem Paket sind alle Klassen zu finden, die Veränderungen an Termen vornehmen, also das eigentliche „Gehirn“ des Programms. Hier werden Überprüfungen von TES durchgeführt, die Knoten gesucht, an denen der nächste Reduktionsschritt durchgeführt werden kann und Ähnliches.

7.2.1. `CheckTRS`

Die Methoden der `CheckTRS`-Klasse werden benutzti, um Überprüfungen der Regelsätze und ihrer Instanzen durchzuführen. Diese Tests bestimmen, mit welchen Interpretationsmodi ein TES interpretiert werden kann. Die Überprüfungen werden im Konstruktor der Klasse `InterpreterWindow` im Paket `kites.visual` durchgeführt.

Eine ausführliche Beschreibung der verschiedenen Tests kann dem Kapitel 3 entnommen werden.

7.2.2. `Codification`

Mit den Methoden der `Codification`-Klasse werden TES in ihre kodierte Standardform transformiert. Eine genaue Beschreibung der kodierten Standardform kann Kapitel 6 entnommen werden. Die Methoden dieser Klasse sind entlang der dortigen Beschreibung implementiert.

7.2.3. `Decomposition`

Um einen Interpretationsschritt durchzuführen, muss zuvor ermittelt werden, an welchen Stellen der Instanz Substitutionen gemäß dem aktuellen Interpretationsmodus und, im Falle der Interpretation als Programm, der aktuellen Auswertungsreihenfolge durchgeführt werden. Diese Aufgabe wird von den Methoden dieser Klasse erfüllt. Außerdem enthält sie auch die Methode, die prüft, ob zwei TES-Bäume aufeinander passen und gleichzeitig eine entsprechende Variablenbelegung zurück liefert.

Matching

Um zu überprüfen ob zwei Terme aufeinander passen, wird ein struktureller Vergleich der Bäume durchgeführt. Dies bedeutet, dass überprüft wird, ob jeder Knoten des ersten Baumes mit seinem Gegenstück im zweiten Baum übereinstimmt. Diese Übereinstimmung wird anhand von drei Kriterien überprüft:

1. Sind die Knoten vom gleichen Typ (`Constant` oder `Function`)
2. Tragen die Knoten den gleichen Bezeichner (zum Beispiel *append*)
3. Stimmen die eventuell vorhandenen Kinder der Knoten in derselben Weise überein

Allein durch die dritte Bedingung wird offensichtlich, dass sich dies durch einen einfachen rekursiven Algorithmus implementieren lässt. Die einzige Ausnahme von den Kriterien betrifft Variablen. Bei diesen wird zunächst überprüft, ob sie bereits belegt wurden. Falls dies der Fall ist, wird nun ein Vergleich zwischen der Belegung und dem Teilbaum im zweiten Baum durchgeführt. Sollte dies nicht Fall sein, wird eine neue Belegung in der Form des Teilbaums des zweiten Baums eingetragen.

Schlägt ein Vergleich fehl, steht fest, dass die beiden Terme nicht aufeinander passen.

Auswertungsstrategien

Die Algorithmen der Auswertungsstrategien sind sich alle sehr ähnlich, sie unterscheiden sich hauptsächlich in der Richtung, in der die Bäume durchlaufen werden. Ihnen allen ist gemein, dass der oben genannte Matching-Algorithmus ausgeführt wird. Dabei wird in der für jede Strategie individuellen Reihenfolge für die Knoten,

beziehungsweise Teilbäume, der Instanz überprüft ob eine linke Regelseite auf sie passt. Die Regeln werden dabei in der Reihenfolge durchlaufen, in der sie vom Benutzer angegeben wurden. Dies spielt aber für das Auswertungsergebnis keine Rolle (siehe 2.4).

Bei den Strategien für die Ausführung im Programm-Modus wird nach dem ersten Fund die Suche abgebrochen, definitionsgemäß kann es keine weiteren Treffer geben. Bei der Strategie für die Interpretation als nicht-deterministisches Programm werden noch die verbliebenen Regeln untersucht und dann die Suche abgebrochen. Im TES-Interpretations-Modus wird die Suche erst beendet, wenn alle Knoten untersucht wurden.

7.2.4. `ProgramRewrite`

Die Methoden der `ProgramRewrite`-Klasse führen die tatsächlichen Ersetzungen in Termen durch, nachdem feststeht, an welcher Stelle des Terms die Ersetzung stattfinden soll.

Der Algorithmus für die Substitution ist sehr einfach gehalten: Es ist bereits bekannt, an welcher Stelle die Substitution mit welcher Variablenbelegung durchgeführt werden soll. Der Baum der Instanz wird nun durchlaufen und dabei ein neuer Baum als Kopie der Instanz geschaffen. Wird der zu ersetzende Teilbaum erreicht, so wird dieser allerdings nicht kopiert, sondern an dieser Stelle eine Kopie der rechten Regelseite der anzuwendenden Regel eingesetzt. Dies funktioniert wieder so, dass der Baum der rechten Regelseite Knoten für Knoten durchlaufen wird und eine Kopie jedes Knotens an den neuen Instanz-Baum angehängt wird. Trifft der Algorithmus hier auf eine Variable, so wird diese im neuen Instanzbaum durch ihre Belegung ersetzt.

7.3. Paket `kites.parser`

Das Parser-Paket besteht aus nur einer einzelnen tatsächlichen Quell-Datei: `TRS.g`. Sie enthält die Beschreibung der Grammatik zur Erkennung der Benutzereingaben. Mithilfe von ANTLR können aus ihr zwei weitere Dateien, `TRSLexer.java` und `TRSParser.java`, generiert werden, die einen Automat zum Erkennen der Sprache enthalten und von den anderen Java-Paketen genutzt werden.

Die benutzte Grammatik wird in Kapitel 4 genauer erläutert. Hier erfolgt keine genauere Erläuterung des Java-Sourcecodes, da dieser automatisch generiert wird und die einzig relevante Datei die `TRS.g` ist, deren Syntax schnell verständlich ist.

7.4. Paket kites.model

Im model-Package sind alle Klassen, die die Datenstrukturen definieren, die Knoten, Terme, Regeln und auch ein ganzes Termersetzungssystem repräsentieren. Informationen über die programminterne Repräsentation der Daten sind in Kapitel 5 zu finden.

7.4.1. ASTNode

Die abstrakte Klasse ASTNode stellt die Basis für die Syntaxbäume der Terme dar. Sie bietet eine Schnittstelle unter der alle Funktionen der tatsächlich implementierenden Klassen Constant, Function und Variable vereinigt sind. Objekte dieser Klasse können in einer konkreten Implementation Kindobjekte besitzen, mit denen die Bäume aufgebaut werden.

7.4.2. Constant

Objekte dieser Klasse repräsentieren Funktionssymbole ohne Parameter. Ein Aufruf der Methode getChildren() löst immer eine NoChildrenException aus.

7.4.3. Function

Objekte dieser Klasse repräsentieren Funktionssymbole mit Parametern. Hier wird keine NoChildrenException mehr ausgelöst.

7.4.4. Variable

Objekte dieser Klasse repräsentieren Variablensymbole. Bei einem Aufruf der getChildren-Methode wird immer eine NoChildrenException ausgelöst.

7.4.5. Rule

Regeln werden mit Hilfe dieser Klasse abgebildet. Sie enthält zwei Bäume, die mit Hilfe von ASTNode-Objekten aufgebaut sind, je einen für die linke und rechte Regelseite.

7.4.6. `TRSFile`

Objekte der Klasse `TRSFile` werden vom Parser aus Dateien eingelesen und enthalten eine Menge von Regeln, die als Liste von `Rule`-Objekten im Objekt gespeichert werden. Zusätzlich kann ein `TRSFile`-Objekt auch noch eine Instanz enthalten, die als einfacher `ASTNode`-Baum abgelegt werden.

7.5. Paket `kites.visual`

Das `visual`-Package enthält die Darstellungsschicht der Applikation. In ihr sind alle Klassen für die verschiedenen Fenster versammelt, als auch der notwendige „Klebstoff“ zwischen den Klassen des `logic`- und `parser`-Pakets in Form der Klasse `StepRewrite`. Auch Wrapper für die `ASTNode`-verwandten Klassen für die Darstellung als `JLabel` sind hier zu finden.

7.5.1. `MainWindow`

Das `MainWindow` ist das Hauptfenster des Programms. Die Klasse enthält außerdem die `main`-Methode, also den Einsprungpunkt des Programms.

7.5.2. `InterpreterWindow`

Im `InterpreterWindow` wird die eigentliche Interpretationsarbeit durchgeführt und die Ergebnisse präsentiert. Die Darstellung der Berechnungsschritte und des Ergebnisses erfolgt hierbei nicht als Text, sondern in Form von Objekten der Klasse `NodeContainer`, erweiterten `JLabel`s.

7.5.3. `CodificationWindow`

Nach der Transformation eines Regelsystems und einer Instanz in ihre kodierte Standardform werden beide in einer Instanz eines `CodificationWindow` angezeigt. Aufgrund der Simplizität des Fensters wird es nicht genauer erläutert.

7.5.4. StepRewrite

Die Klasse StepRewrite tritt in Aktion, wenn die Interpretation eines TES tatsächlich durchgeführt werden soll. In dieser Klasse wird dann die Instanz geparst und die entsprechenden Klassen und Methoden aus dem Paket kites.logic instantiiert und benutzt. Nach der Ausführung eines einzelnen Schrittes werden dann die Bedienelemente der MainWindow-Instanz aktualisiert.

7.5.5. NodeContainer

Ähnlich wie ASTNode ist auch NodeContainer eine abstrakte Klasse, die eine gemeinsame Schnittstelle für die Klassen NodeBox und NodeLabel bereitstellt. Diese beiden Klassen dienen der Darstellung von Termen im InterpreterWindow.

Ihre Verwendung anstatt von einfachem Text ist notwendig, da sie zielgenau klick- und färbbar sein müssen. Es stellte sich heraus, dass dies mit Text beispielsweise in einem JEditorPane nicht oder nur sehr schwierig möglich ist.

7.5.6. NodeBox

Eine NodeBox erweitert die Klasse JPanel. Sie ist mit einem tabellenartigen Layout versehen und ordnet darin ihre Kindelemente, Objekte der Klasse NodeContainer an. Es gibt ein Kopfsymbol, das im linken oberen Feld angezeigt wird, die Kindelemente dieses Knotens werden in der rechten Spalte angezeigt.

Besonders zu beachten ist, dass Kommata und schließende Klammern bereits in den NodeLabel-Objekten stehen. Dies ist erforderlich, da die Darstellung sonst für einen Leser sehr ungewohnt aussieht, da die Kommata, beziehungsweise schließende Klammern rechts nach der Länge des längsten Kindelements ausgerichtet wären.

7.5.7. NodeLabel

Durch die Erweiterung von JLabel entsteht die Klasse NodeLabel, die den Namen eines einzelnen Knoten anzeigt sowie zur Steuerung der Interpretation einfärbbar ist und ein Popup-Menü mit auf diesen Knoten anwendbaren Regeln anzeigen kann.

7.6. Lebensweg eines TES

Nach dem Programmstart muss zunächst ein Regelsatz angegeben werden. Dies kann entweder durch das Öffnen einer bereits vorhandenen Datei oder die manuelle Eingabe einer Liste von Regeln erfolgen.

Nach einem Druck auf den Toolbar-Button „Interpreter starten“ wird das angegebene Termersetzungssystem eingeparst und in ein Objekt der Klasse `TRSFile` übertragen. Dieses Objekt wird einer neuen `InterpreterWindow`-Instanz übergeben. Hier werden nun zunächst die Überprüfungen der Klasse `CheckTRS` durchgeführt, um festzustellen, welche Interpretationsmodi für dieses TES möglich sind. In unserem Beispiel besteht das TES alle Tests, es sind also alle drei Interpretationsmodi möglich. Das `InterpreterWindow` ist nun bereit und wartet auf eine Benutzerinteraktion.

Sofern bei der Eingabe des Termersetzungssystems nicht auch direkt eine Instanz angegeben wurde, so muss nun noch eine Instanz angegeben werden.

Nach der Eingabe einer Instanz ist KiTES zur Interpretation bereit. Nach einem Druck auf Ausführen wird die `run()`-Methode einer Instanz von `StepRewrite` aufgerufen. Beim ersten Aufruf wird von dieser zunächst die angegebene Instanz geparst und in die Baumdarstellung überführt. Nun werden auf diesem Baum alle möglichen Ersetzungen im aktuellen Modus gesucht. Dies geschieht durch die Klasse `Decomposition`. Aus dem Baum und diesem Ergebnis wird nun die Darstellung in Form von `NodeContainer`-Objekten erstellt. Als nächstes wird dieser Baum als Ergebnis des ersten „Berechnungsschrittes“ im Ergebnisbereich ausgegeben.

Abhängig vom Auswertungsmodus unterscheidet sich allerdings die Ausgabe. Läuft der Interpreter im Programm-Modus, so wird der Schritt auf grauem Grund ausgegeben und der Button „Schritt“ hat seine Beschriftung und auch Bedeutung zu „Ersetzung finden“ geändert. Nach einem Druck auf diesen Button wird die Stelle, an der die Ersetzung vorgenommen werden wird, gelb eingefärbt und der Knopf erhält seine ursprünglichen Eigenschaften zurück. Wird er gedrückt, wird nun die Ersetzung über die Methoden der `ProgramRewrite`-Klasse durchgeführt und ein neuer Baum, wie zuvor beschrieben, ausgegeben.

Läuft das Programm in einem anderen Modus, so wird der Baum zunächst ebenfalls auf grauem Grund ausgegeben, allerdings werden die Stellen der möglichen Ersetzung sofort eingefärbt, die Popup-Menüs der jeweiligen `NodeLabel`-Objekte aktiviert und der Schritt-Button deaktiviert. Wird nun das Popup-Menü eines reduzierbaren Knotens geöffnet und eine der passenden Regeln ausgewählt, so wird ein neuer Berechnungsschritt über die `run()`-Methode der `StepRewrite` ausgeführt.

Dies kann so lange vonstatten gehen, bis keine weiteren Reduktionen mehr möglich

sind. Bei der Interpretation im Programm-Modus ist auch die komplette Ausführung eines TES möglich, so lange bis keine Reduktion mehr möglich ist. Dies ist einfach nur die Ausführung der `run()`-Methode in einer Endlos-Schleife, die erst abgebrochen wird, wenn von der `run()`-Methode die `NoRewritePossibleException` geworfen wird.

A. Mitgelieferte Regeln

Die hier angegebenen Regelsätze stellen eine einfache „Standardbibliothek“ häufig verwendeter Funktionen dar und liegen KiTES bei. Sie wurden alle den Vorlesungsmaterialien von Prof. Dr. Thomas Wilkes Vorlesung „Theoretische Grundlagen der Informatik“ entnommen. Sie können über die Funktion „Regelsätze laden“ im Hauptfenster in die aktuelle Arbeitsdatei über eine `#include`-Anweisung eingebunden werden und liegen ebenfalls auf CD bei.

A.1. `append.trs`

append(X, Y) fügt zwei mit *cons* aufgebaute Listen X und Y zu einer Liste zusammen.

```
append(cons(X, XS), Y) --> cons(X, append(XS, Y))
append(empty, Y) --> Y
```

A.2. `boolean.trs`

Boolean.trs enthält die booleschen Operationen *and*, *or* und *not*.

```
and(true, true) --> true
and(true, false) --> false
and(false, X) --> false

or(true, X) --> true
or(false, true) --> true
or(false, false) --> false

not(true) --> false
not(false) --> true
```

A.3. comparison.trs

Es sind zwei Funktionen enthalten: $leq(X, Y)$ liefert das Ergebnis von $X \leq Y$, $equal(X, Y)$ das Ergebnis von $X = Y$.

```
leq(zero, X) --> true
leq(suc(X), zero) --> false
leq(suc(X), suc(Y)) --> leq(X, Y)

equal(zero, zero) --> true
equal(zero, suc(x)) --> false
equal(suc(x), zero) --> false
equal(var(x), var(y)) --> equal(x, y)
```

A.4. formula.trs

Mit diesen Regeln können aussagenlogische Formeln ausgewertet werden. Variablen müssen wie bei der kodierten Standardform mit $var(suc(\dots(zero)\dots))$ angegeben werden. Eine Beispielinstanz für dieses System ist:

$$val(land(var(zero), lor(neg(var(suc(suc(zero)))), var(suc(zero))))) \tag{A.1}$$

Sie liefert das Ergebnis der Formel

$$x_0 \wedge (\neg x_2 \vee x_1) \tag{A.2}$$

```
val(true, Y) --> true
val(false, Y) --> false
val(var(X), Y) --> lookup(var(X), Y)
val(neg(X), Y) --> not(val(X, Y))
val(land(X, Y), Z) --> and(val(X, Z), val(Y, Z))
val(lor(X, Y ), Z) --> or(val(X, Z), val(Y, Z))

lookup(X, cons(cons(Y , Z), XS)) --> if-then-else(equal(X, Y),
     Z, lookup(X, XS))
```

A.5. if-then-else.trs

Mit den Regeln aus `if-then-else.trs` ist eine Verzweigung in einem Programm möglich. Ein Aufruf von $if - then - else(P, X, Y)$ wird zu X reduziert, wenn P wahr ist, andernfalls zu Y.

```
if-then-else(true, X, Y) --> X
if-then-else(false, X, Y) --> Y
```

A.6. interpretierer.trs

Es ist möglich, einen Interpreter für TES als TES zu formulieren. Die Regeln in der Bibliothek `interpretierer.trs` sind genau das. Das Kopfsymbol ist $compute(X, Y)$. X ist dabei die kodierte Standardform eines Regelsatzes, Y die kodierte Standardform einer Instanz. Eine genaue Beschreibung der Funktionsweise des Interpretierers würde den Rahmen dieser Arbeit sprengen, die grundlegende Funktionsweise ist jedoch ähnlich zu der der hier vorgestellten Umgebung. Es wird nach einer passenden Stelle und einer passenden Regel für die Substitution gesucht und diese dann durchgeführt. In der Quelldatei sind auch noch zusätzliche Kommentare, die die Funktion jeder Prozedur kurz beschreiben.

```
// Berechnung von X auf Y
compute(X, Y) --> if-then-else(step(X, Y), compute(X,
     step(X, Y)), Y)

// Einmalige Anwendung von X auf Y
step(X, Y) --> if-then-else(inside(X, Y), rewrite(inside(X, Y),
     Y), false)

// False, falls keine linke Seite aus X in Y passt, sonst
// die entsprechende Regel
inside(X, Y) --> inside-helper(X, X, empty, subterms(Y))
inside-helper(X, empty, Y, empty) --> false
inside-helper(X, cons(cons(Y, Y), YS), Z, Z)
     --> if-then-else(match(Y, Z), cons(Y, Y),
     inside-helper(X, YS, Z, Z))
inside-helper(X, empty, Y, cons(Z, ZS))
     --> inside-helper(X, X, Z, ZS)

// Liste der Subterme von X in LO-Reihenfolge
subterms(var(X)) --> cons(var(X), empty)
subterms(cons(fun(X), Y)) --> cons(cons(fun(X), Y),
     subterms-list(Y))
subterms-list(empty) --> empty
```

```
subterms-list(cons(X, XS)) --> append(subterms(X),
     subterms-list(XS))

// Ersetze X durch Y in Z, vorausgesetzt der Term X passt in Z
rewrite(cons(X, Y), cons(Z, ZS)) --> if-then-else(match(X,
     cons(Z, ZS)), subst(Y, match(X, cons(Z, ZS))), cons(Z,
     rewrite-list(X, Y, ZS)))
rewrite-list(X, Y, cons(Z, ZS)) --> if-then-else(match(X, Z),
     cons(subst(Y, match(X, Z)), ZS), cons(Z, rewrite-list(X,
     Y, ZS)))

// False, falls X nicht auf Y passt, ansonsten die
// entsprechende Anpassung
match(var(X), Y) --> cons(cons(var(X), Y), empty)
match(cons(X, XS), cons(Y, YS)) --> if-then-else(equal(X, Y),
     match-list(XS, YS, empty), false)
match-list(empty, empty, X) --> X
match-list(cons(X, XS), cons(Y, YS), Z)
     --> if-then-else(match(X, Y), match-list(XS, YS, append(Z,
     match(X, Y))), false)

// subst(X, Y) - Wende Belegung Y auf Term X an
subst(var(X), cons(cons(var(Y), Z), ZS))
     --> if-then-else(equal(X, Y), Z, subst(var(X), ZS))
subst(cons(X, XS), Y) --> cons(X, map-subst(XS, Y))
map-subst(empty, X) --> empty
map-subst(cons(X, XS), Y) --> cons(subst(X, Y), map-subst(XS, Y))
```

A.7. ndet-int.trs

Neben dem vorherigen Interpretierer für Programme ist es ebenfalls möglich einen Interpretierer für den Interpretationsmodus als nichtdeterministisches Programm zu schreiben. Der Aufruf erfolgt ähnlich wie bei dem Interpretierer für Programme mit $compute(X, Y, Z)$, wobei X und Y wieder der Regelsatz und die Instanz in kodierter Standardform sind. Der bisher unbekannte Parameter Z ist eine Folge von Zahlen, die die Auswertungsreihenfolge vorgeben, also an welcher Stelle welche Regel angewendet wird.

```
compute(X, Y, empty) --> if-then-else(equal(zero,
     no-match-inside(X, Y)), Y, false)
compute(X, Y, cons(Z, ZS)) --> if-then-else(equal(zero,
     no-match-inside(X, Y)), Y, if-then-else(leq(Z,
     no-match-inside(X, Y)), check-one(X, apply(X, Y, Z), ZS),
     false))

no-match-inside(X, cons(Y , YS)) --> if-then-else(leq(suc(zero),
```

```
     no-match(X, cons(Y, YS))), no-match(X, cons(Y, YS)),
     list-no-match-inside(X, YS))
list-no-match-inside(X, empty) --> zero
list-no-match-inside(X, cons(Z, ZS)) --> add(no-match-inside(X,
     Z), list-no-match-inside(X, ZS))
no-match(empty, Y) --> zero
no-match(cons(cons(X, Y), XS), Z) --> if-then-else(match(X, Z),
     suc(no-match(XS, Z)), no-match(XS, Z))

apply(X, cons(Y, YS), Z) --> if-then-else(leq(Z, no-match(X,
     cons(Y, YS))), apply-imm(X,cons(Y, YS), Z), cons(Y,
     list-apply(X, YS, Z)))
list-apply(X, cons(Y, YS), Z) --> if-then-else(leq(Z,
     no-match-inside(X, Y)), cons(apply(X, Y), YS), cons(Y,
     list-apply(X, YS, minus(Z, no-match-inside(X, Y)))))
apply-imm(cons(cons(X, Y), XS), Z, Z) --> if-then-else(match(X,
     Z), if-then-else(equal(suc(zero), Z), subst(Y, match(X, Z)),
     apply-imm(XS, Z, minus(Z, suc(zero)))), apply-imm(XS, Z, Z))
```

A.8. maximum.trs

Die Funktion $max(X)$ findet die größte Zahl in einer mit *cons* aufgebauten Liste von Zahlen.

```
max(cons(X, XS)) --> max-helper(X, XS)
max-helper(X, empty) --> X
max-helper(X, cons(Y, YS)) --> if-then-else(leq(X, Y),
     max-helper(Y, YS), max-helper(X, YS))
```

B. Benutzungshinweise

Dieser Anhang enthält eine kurze Bedienungsanleitung für die Experimentierumgebung und erklärt die Programmfunktionen. Er stellt keinesfalls eine Programmieranleitung für das Programmieren mit Termersetzungssystemen dar.

Nach dem Start von KiTES erscheint das Hauptfenster, das bereit zur Regeleingabe ist (siehe Abbildung B.1).

Dort müssen in den Eingabebereich die Regeln des zu interpretierenden TES eingegeben gemäß der in Kapitel 4 angegebenen Grammatik angegeben werden, wie ebenfalls in der Abbildung zu sehen. Über das Menü „Bearbeiten“ lassen sich mitgelieferte Regeln laden. Es wird dabei eine neue `include`-Anweisung in das Regelsystem eingefügt.

Über einen Druck auf den Button „Interpreter starten“ öffnet sich nun das Interpreterfenster (siehe Abbildung B.2). Hier wird die Auswertung stattfinden. Doch zunächst ist ein Grundterm als Instanz des Regelsystems erforderlich. Die Instanz wird in den unteren Eingabebereich des Interpretationsfensters eingegeben.

Beim Öffnen des Interpreterfensters wird automatisch überprüft mit welchen Interpretationsmodi das TES kompatibel ist. Über das Menü „Interpretationsmodi“

Abbildung B.1.: Hauptfenster mit eingegebenem Regelsystem

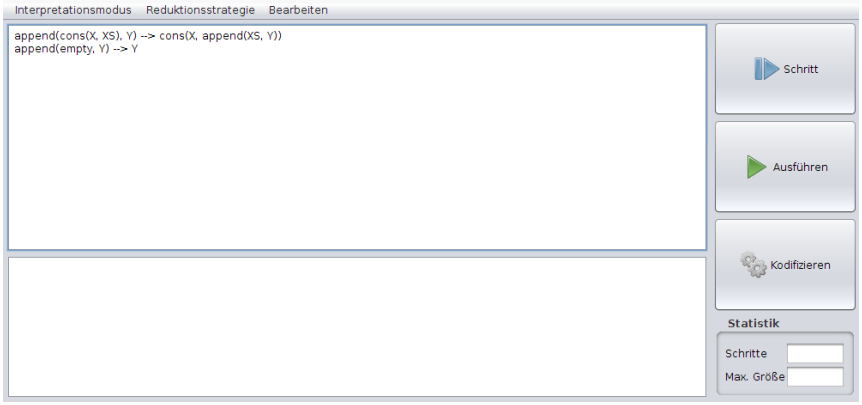

Abbildung B.2.: Interpretationsfenster ohne eingegebene Instanz

kann ausgewählt werden, welcher Modus verwendet werden soll. Der aktuell eingestellte Interpretationsmodus wird außerdem in der (im Screenshot nicht sichtbaren) Titelleiste angezeigt. Ist als Interpretationsmodus „Programm“ ausgewählt, so wird außerdem das Menü „Reduktionsstrategie“ freigeschaltet, mit dem zwischen vier verschiedenen Auswertungsstrategien gewählt werden kann.

Bei einem Lauf im Programm-Modus kann nun mit dem Button „Ausführen“ ein kompletter Lauf des TES angestoßen werden, also so lange Reduktionen durchgeführt werden, bis keine mehr möglich ist. Alternativ kann über den Button „Schritt“ eine schrittweise Ausführung begonnen werden. Diese Ausführung verhält sich zweigeteilt. Beim ersten Druck auf den Button wird die erste Reduktion durchgeführt, dies bedeutet im Fall der ersten Ausführung, dass die Instanz ausgegeben wird (siehe Abbildung B.3). Nach einem weiteren Druck auf den selben Button, dessen Beschriftung sich nun allerdings geändert hat zu „Ersetzung finden“, färbt die nächste Ersetzungsmöglichkeit ein. Die Beschriftung ändert sich nun wieder zu „Ersetzen“. Ein Druck auf den Button löst nun die Ersetzung aus, woraufhin sich wieder die Beschriftung zu „Ersetzung finden“ ändert, und so weiter, bis keine Ersetzungen mehr möglich sind.

In den anderen Interpretationsmodi ist keine automatische komplette Ausführung möglich, nur eine schrittweise. Beim ersten Druck auf den „Schritt“-Button wird wieder die Instanz ausgegeben. Hier wird aber der „Schritt“-Button deaktiviert und erfüllt keine Funktion mehr. Um mit der Ausführung fortzufahren muss nun auf einen der reduziblen, eingefärbten Teilterme geklickt werden. Es öffnet sich ein Popup-Menü mit den anwendbaren Regeln. Bei Klick auf eine, wird die entsprechende Regel angewendet und ein neuer Term der Ausgabe hinzugefügt, der ebenso reduziert werden kann. Der Knoten, in dem reduziert wurde, wird nun grün eingefärbt (siehe Abbildung B.4).

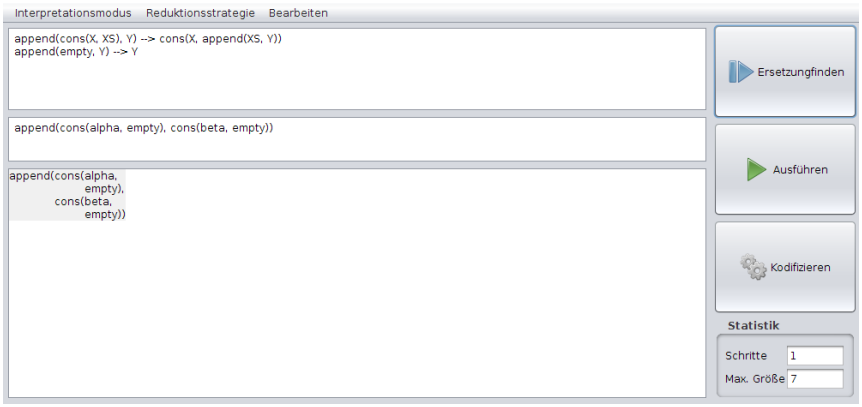

Abbildung B.3.: Interpretationsfenster mit Auswertung als Programm

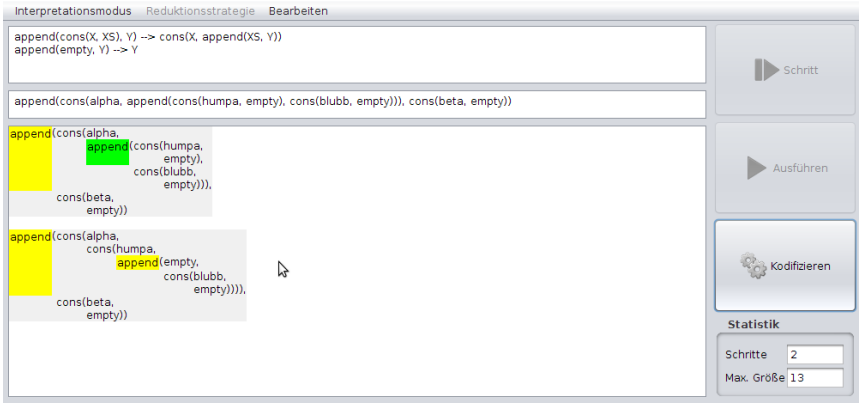

Abbildung B.4.: Interpretationsfenster mit Auswertung als Termersetzungssystem

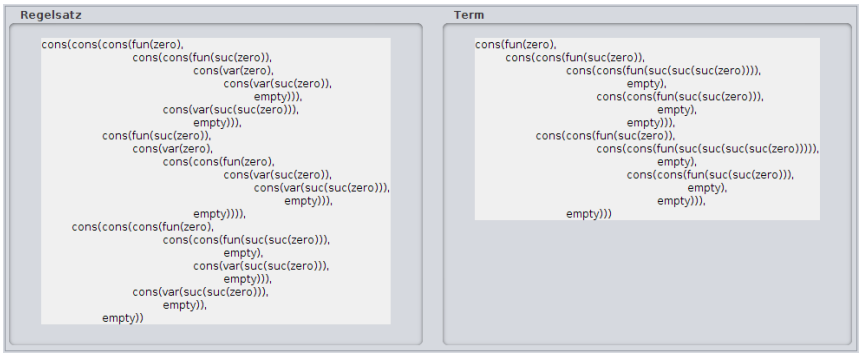

Abbildung B.5.: Ausgabe in kodierter Standardform

Die letzte Primärfunktion des Programms ist die Überführung eines Regelsystems und eines Terms in ihre kodierte Standardform, wie in Kapitel 6 beschrieben. Bei einem Druck auf den Button „Kodifizieren“ wird ein neues Fenster geöffnet, in dem im linken Teil die kodierte Standardform des Regelsatzes und im rechten Teil die kodierte Standardform der Instanz zu sehen sind.

Wie schon mehrfach erwähnt, findet die Ergebnisausgabe nicht in Form von Text statt, der einfach kopiert und eingefügt werden kann. Es gibt aber in jedem Fall eine spezielle „Ergebnis kopieren“-Funktion die über die rechte Maustaste erreichbar ist. Mit ihr kann das Ergebnis als Text in die Zwischenablage kopiert und in jedem beliebigen Programm eingefügt und weiterverwendet werden.

Zu beachten ist beim Programmieren auch, dass mehrfaches Einbinden von Regeln tatsächlich dazu führt, dass diese mehrfach vorhanden sind und damit die Ausführung im Programm-Modus nicht mehr möglich ist. Siehe dazu Abschnitt 3.3 über die Interpretation im Programmmodus.

Literatur

[1] Jürgen Avenhaus. *Reduktionssysteme*. Berlin: Springer, 1995.

[2] Franz Baader und Tobias Nipkow. *Term rewriting and all that*. Cambridge: CUP, 1999.

[3] Reinhard Bündgen. *Termersetzungssysteme: Theorie, Implementierung, Anwendung*. Braunschweig/Wiesbaden: Vieweg, 1998.

[4] Thorsten Juergeleit. *ANTLR plugin for Eclipse*. 2010. URL: `http://antlreclipse.sourceforge.net`.

[5] Jan Willem Klop und Roel de Vrijer. "First-order term rewriting systems". In: Terese. *Term Rewriting Systems*. Cambridge: CUP, 2003. Kap. 2, S. 24–60.

[6] Jan Willem Klop, Vincent van Oostrom und Roel de Vrijer. "Orthogonality". In: Terese. *Term Rewriting Systems*. Cambridge: CUP, 2003. Kap. 2, S. 88–149.

[7] Terence Parr. *ANTLR Parser Generator*. 2010. URL: `http://www.antlr.org`.

[8] Vincent van Oostrom und Roel de Vrijer. "Strategies". In: Terese. *Term Rewriting Systems*. Cambridge: CUP, 2003. Kap. 9, S. 475–548.