Universität Stuttgart
Institut für Formale Methoden der Informatik
Abteilung Theoretische Informatik
Fakultät Informatik, Elektrotechnik und Informationstechnik
Universitätsstr. 38, 70569 Stuttgart

Diplomarbeit Nr. 2718

Erreichbarkeitsanalyse funktionaler Programme mit Grundtermersetzungssystemen

Andreas Gaiser

Studiengang:	Informatik
Prüfer:	Prof. Dr. Volker Diekert
Betreuer:	Dr. Stefan Schwoon (TU München)
Begonnen am:	15.01.2008
Beendet am:	16.7.2008
CR-Klassifikation:	D.2.4, D.2.5, F.4.2, F.4.3, F.3.2

Zusammenfassung

Diese Arbeit beschäftigt sich mit Erreichbarkeitsproblemen in der funktionalen Programmierung. Funktionalen Programmen kann mithilfe von Termersetzungssystemen eine operationale Semantik gegeben werden. Durch Einschränkung der Programmvariablen auf endliche Wertebereiche entstehen Grundtermersetzungssysteme, für die das Erreichbarkeitsproblem entscheidbar ist. Bestehende Ansätze benötigen viel Zeit und Speicherplatz, um die aus funktionalen Programmen entstehenden Grundtermersetzungssysteme verarbeiten zu können. Daher wurde eine effiziente Darstellung von Grundtermersetzungssystemen und ein dazugehöriger symbolischer Algorithmus für das Erreichbarkeitsproblem entwickelt.

Danksagung

Danken möchte ich vor allem Javier Esparza und Stefan Schwoon. Sie begleiteten mich während meines gesamten Studiums in Vorlesungen, Seminaren und studentischen Arbeiten. Sie verstanden es in bemerkenswerter Weise, mich für Model-Checking, formale Semantik und Automatentheorie zu begeistern. Während der Erstellung dieser Arbeit konnte ich mit allen Fragen und Problemen, waren sie auch noch so trivial, zu ihnen kommen. Besonders dankbar bin ich für die vielen erhellenden Gespräche, die manchmal ganze Nachmittage einnahmen.

Danken möchte ich auch Volker Diekert dafür, dass er die Durchführung dieser Arbeit möglich gemacht hat.

Meinen Eltern und meiner Schwester, die immer mit viel Liebe und ganzer Kraft für mich da waren, gilt der größte Dank. Ihnen möchte ich diese Arbeit widmen.

Inhaltsverzeichnis

1 Einführung **9**

2 Grundlagen **13**
2.1 Termersetzungssysteme 13
2.2 Baumautomaten 18
2.2.1 Eigenschaften 21
2.2.2 Entscheidungsprobleme 22
2.2.3 Minimale Baumautomaten 23

3 Funktionale Programmierung und Termersetzungssysteme **25**
3.1 Ein Modell für funktionale Programme 25
3.2 Testen funktionaler Programme 27

4 Erreichbarkeitsanalysen **29**
4.1 Nichtsymbolischer Ansatz 29
4.2 Motivation für einen symbolischen Ansatz 30
4.3 Symbolische Erreichbarkeitsanalyse 31
4.3.1 Symbolische Grundtermersetzungssysteme 31
4.4 Ein symbolischer Algorithmus 34
4.4.1 Terminierung und Korrektheit 37

5 Implementation **43**
5.1 Aufbau 43
5.2 Darstellung von Baumautomaten 45
5.2.1 Implementierte Operationen 46
5.3 Implementierung symbolischer Grundtermersetzungssysteme . 48
5.3.1 Die Klasse `Term` 48
5.3.2 Die Klassen `Rule` und `RewritingSystem` 49
5.4 Implementierung des symbolischen Algorithmus 49
5.4.1 Ein Iterationsschritt 50
5.4.2 Kopienverwaltung 53
5.4.3 Beispiel 54
5.4.4 Beschleunigung des Verfahrens 57
5.5 Implementierung des nichtsymbolischen Algorithmus 58
5.6 Reduktionsalgorithmen 58
5.6.1 Backward-Bisimulation 60
5.6.2 Forward-Bisimulation 63
5.6.3 Minimierung deterministischer Baumautomaten 64
5.7 Generierung von Gegenbeispielen 65

5.7.1 Annotation der Baumregeln 65
5.7.2 Rekonstruktion von Ableitungsfolgen 67
5.8 Benutzung des Prototypen 72
5.8.1 Syntax der Eingabedateien 73
5.8.2 Aufruf des Prototypen 76
5.9 Parser und Eingabedateien 77
5.9.1 Parser 77
5.9.2 Erstellung des Grundtermersetzungssystems 77

6 Untersuchungen und Anwendungen 79
6.1 Analyse von Testfällen 79
6.1.1 Vergleich symbolischer - nichtsymbolischer Algorithmus 81
6.1.2 Laufzeitintensive Testfälle 84
6.2 Einschränkungen des symbolischen Algorithmus 86
6.2.1 Teilentfaltung symbolischer Regeln 87
6.2.2 Simulation mehrfacher Variablenvorkommen mithilfe von Nachfolgermengen 88
6.2.3 Bewertung der Strategien 88
6.3 Modifikationen und Verbesserungen 89
6.3.1 Einfluss von Reduktionsalgorithmen 89
6.3.2 Weitere Verbesserungsstrategien 91
6.4 Anwendungen jenseits funktionaler Programmierung 94

7 Fazit 98

1 Einführung

Die funktionale Programmierung stellt ein sehr frühes Programmierparadigma dar. Als Stammvater aller funktionaler Programmiersprachen gilt das λ-Kalkül, welches bereits in den 1930er Jahren entwickelt wurde. Aus ihm entwickelten sich eine große Sprachfamilie, mit Vertretern wie Lisp, Scheme, ML oder Erlang. Funktionale Programmierung stellt den Begriff der Funktion in den Mittelpunkt des Programmiervorgangs: Programme sind Funktionsdefinitionen, Programmberechnungen entsprechen dem Auswerten von Funktionsausdrücken. In rein funktionalen Programmiersprachen fehlt zudem das Konzept der mehrfachen Zuweisung an Variablen, welches von besonderer Bedeutung für die konventionelle imperative Programmierung ist. Vielmehr kommen funktionale Programme dem Begriff der mathematischen Funktion nahe (siehe [ASS01]). Innerhalb dieser Arbeit möchten wir eine Möglichkeit entwickeln, Eigenschaften funktionaler Programme nachzuweisen. Wir sind vor allem an Erreichbarkeitsanalysen interessiert; dabei wird untersucht, zu welchen Ergebnissen sich eine vorgegebene Menge von Funktionsausdrücken auswerten lässt. Hierfür benötigen wir formale Methoden, um die Programmsemantik darstellen zu können.

Eine Möglichkeit, für funktionale Programmiersprachen eine operationale Semantik zu beschreiben, stellen Termersetzungssysteme dar. Definitionen eines funktionalen Programms lassen sich oftmals leicht als eine Menge von Termersetzungsregeln auffassen. Der Zustand eines funktionalen Programms in Form eines Funktionsausdrucks entspricht dann einem Term, die schrittweise Auswertung des Programms einer Anwendung von Termersetzungsregeln. Dadurch ist es möglich, Erreichbarkeitsanalysen funktionaler Programme auf das Erreichbarkeitsproblem von Termersetzungssystemen zu reduzieren.

[EHRS00] beschreibt einen Ansatz, um u.a. Erreichbarkeitsanalysen für imperative Programmiersprachen durchführen zu können: Die Programme werden als Wortersetzungssysteme mit sogenannter Präfixersetzung modelliert. Für Programmvariablen werden endliche Wertebereiche gewählt, so dass ein Wortersetzungssystem nur aus endlich vielen Regeln besteht. Das Erreichbarkeitsproblem hierfür ist in polynomieller Zeit lösbar, effiziente Algorithmen werden in [ES01] gegeben. Sie finden bereits praktische Anwendung, z.B. in jMoped, einem Werkzeug für die Verifikation von Java-Programmen, siehe [SBSE07].

Wortersetzungssysteme können als Spezialfälle von Termersetzungssystemen angesehen werden. Jedoch sind für das Behandeln von funktionalen Programmen Wortersetzungssysteme mit Präfixersetzung nicht ausreichend. Der Ansatz aus [EHRS00] kann aber dadurch auf den allgemeinen Fall übertragen werden, dass die Variablen des Termersetzungssystems auf

einen endlichen Wertebereich eingeschränkt werden. Dies kann dann in ein Grundtermersetzungssystem mit endlich vielen Regeln überführt werden. Für endliche Grundtermersetzungssysteme ist das Erreichbarkeitsproblem entscheidbar: In [Löd06] wird ein entsprechender Algorithmus vorgestellt. Die zu testende Erreichbarkeitseigenschaft Φ wird durch zwei Termmengen M und G dargestellt. Für M wird die Nachfolgermenge bezüglich des Grundtermersetzungssystems errechnet. Sie wird anschließend mit der Menge G geschnitten und das Ergebnis auf Leerheit geprüft. Abb. 1.1 zeigt eine schematische Darstellung des Vorgangs. Diese Problemstellung und die Vorgehensweise, sie zu lösen, hat viele Ähnlichkeiten mit Verfahren des **Model-Checkings**: Hierunter fallen Methoden, um Eigenschaften in Form einer Spezifikationsaussage für formale Strukturen wie Programme, Protokolle oder Schaltkreise nachzuweisen (siehe z.B. [CGP99]).
Problematisch ist, dass die in der Praxis interessanten funktionalen Programme zu sehr großen Grundtermersetzungssystemen führen. Dies verursacht bei dem bereits bestehenden Algorithmen für das Erreichbarkeitsproblem aus [Löd06] Speicher- und Geschwindigkeitsprobleme. Ähnliche Schwierigkeiten gibt es in fast allen Bereichen des Model-Checkings. Ein Ansatz, um ihnen zu begegnen, ist das **symbolische Model-Checking** (siehe [CGP99], [ES01]): Hierbei werden Mengen (z.B. von Zuständen) als aussagenlogische Formeln dargestellt. Meist werden dafür *Binary Decision Diagrams*, kurz BDDs, benutzt. Sie erlauben in vielen Fällen eine komprimierte Repräsentation der dargestellten Mengen. Ebenso ist es möglich, mit BDDs effizient boolesche Operationen durchzuführen; sie entsprechen Mengenoperationen wie z.B. Vereinigung oder Schnitt.
Wir fassen innerhalb dieser Arbeit den Begriff der **symbolischen Darstellung** allgemeiner: Wir verstehen darunter, dass eine Menge oder Relation nicht explizit, also Element für Element, abgespeichert wird, sondern mithilfe einer formalen Beschreibung. Dies kann zum einen aus Effizienzgründen nützlich sein, zum anderen die Möglichkeit eröffnen, auch mit unendlichen Mengen zu arbeiten. Ein Beispiel für eine einfache symbolische Darstellung sind Intervalle in $\mathbb{R}$: Es genügt die Angabe ihrer Suprema und Infima. Die für uns wichtigste symbolische Darstellungsart beschreibt Mengen durch Automaten: Die Sprache der von einem Automaten akzeptierten Elemente ist die darzustellende Menge.
Im Verlauf dieser Arbeit wurde eine symbolische Darstellung für jene Grundtermersetzungssysteme entwickelt, die bei der Behandlung von funktionalen Programmen entstehen. Sie basiert auf der symbolischen Darstellung von Termmengen durch Baumautomaten. Für als Baumautomaten repräsentierte Termmengen lassen sich Operationen wie Schnitt, Vereinigung oder der Test auf Gleichheit effektiv durchführen.
Ähnlich dem Ansatz aus [ES01] entstand ein Verfahren, im Folgenden meist **symbolischer Algorithmus** genannt, das die Erreichbarkeitsanalyse für symbolische Grundtermersetzungssysteme durchführt. Mithilfe eines in der Programmiersprache Java entwickelten Prototyps können so für einfache funktionale Programme Erreichbarkeitsanalysen durchgeführt werden.
In Kapitel 2 dieser Arbeit werden zunächst die formalen und theoretischen

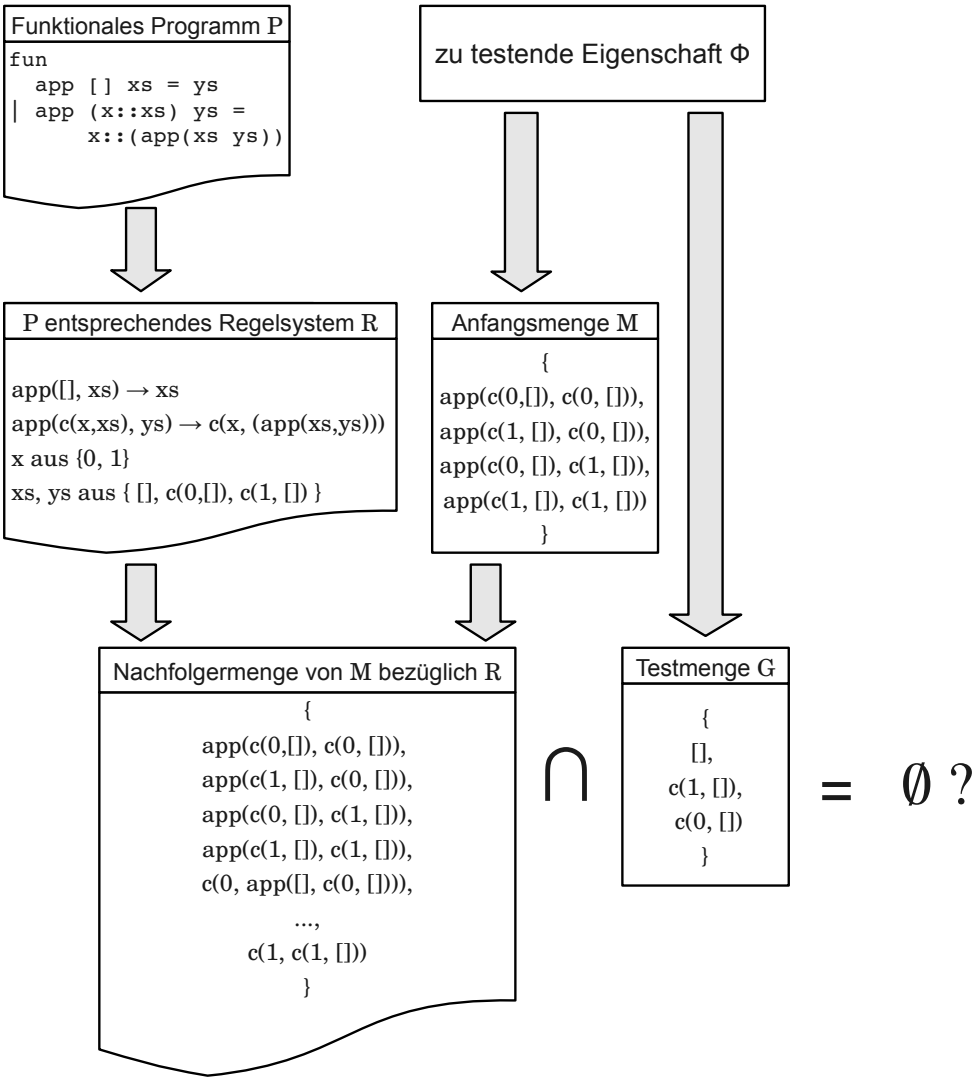


Abbildung 1.1: Schematische Darstellung einer Erreichbarkeitsanalyse.

Grundlagen des neu entwickelten Ansatzes eingeführt, um dann in Kapitel 3 beschreiben zu können, wie wir funktionale Programme als Termersetzungssysteme darstellen. In Kapitel 4 wird der Erreichbarkeitsalgorithmus aus [Löd06] und der in dieser Arbeit entwickelte symbolische Algorithmus vorgestellt. Kapitel 5 enthält Erläuterungen bezüglich des implementierten Prototyps und der in ihm verwendeten Techniken. Kapitel 6 vergleicht den symbolischen mit dem nichtsymbolischen Ansatz, bespricht Beschränkungen des entwickelten Verfahrens sowie mögliche Verbesserungen. Kapitel 7 schließlich zieht ein kurzes Resümee.

2 Grundlagen

Die wichtigsten theoretischen Hilfsmittel, die wir innerhalb dieser Arbeit verwenden, sind Termersetzungssysteme und Baumautomaten. In diesem Kapitel soll daher ein kurzer Überblick zu beiden Themen gegeben werden. Wir lehnen uns dabei an die Darstellungen aus [BN98] und [CDG+07] an.
Einige Bemerkungen zur Notation: Sind A und B Mengen und bezeichnet $\sim \subseteq A \times B$ eine binäre Relation, so bezeichnet $\sim^*$ den reflexiven und transitiven Abschluss von $\sim$. Den transitiven Abschluss bezeichnen wir mit $\sim^+$. Wir verwenden Relationensymbole auch in Infixschreibweise: $a \sim b :\Leftrightarrow (a, b) \in \sim$. Bezeichnet $\equiv$ eine Äquivalenzrelation über der Menge A, so ist $[a]_\equiv := \{x \in A \mid x \equiv a\}$ die Äquivalenzklasse von $a \in A$ bezüglich $\equiv$.
Ist M eine Menge, so bezeichnet $\mathcal{P}(M) := \{Y \subseteq M\}$ die Potenzmenge von M. Ist $f : A \to B$ eine partielle Abbildung von A nach B, so soll $f(a) = \perp$ für ein $a \in A$ bedeuten, dass f an der Stelle a undefiniert ist.

2.1 Termersetzungssysteme

Ein Termersetzungssystem stellt informell ausgedrückt eine Menge von Regeln dar, mithilfe derer Teilterme eines Terms durch andere Terme auf systematische Weise substituiert werden können.

Definition 1. Eine **Signatur** ist eine endliche Menge $\Sigma = \{f_1, ..., f_n\}$ zusammen mit einer Abbildung $arity : \Sigma \to \mathbb{N} \cup \{0\}$. Ein Element $f \in \Sigma$ bezeichnen wir als **Funktionssymbol**. □

Den Wert $arity(f)$ für $f \in \Sigma$ werden wir als die **Arität** von f bezeichnen. Funktionssymbole mit Arität $i \geq 0$ bezeichnen wir als **i-näres** oder **i-stelliges** Funktionssymbol. Funktionssymbole mit Arität 0 bezeichnen wir als **Konstantensymbol**. Wir führen zum Notieren der Aritäten eine spezielle Schreibweise ein: $c()$ bezeichnet dabei ein Konstantensymbol c, $g(\cdot)$ ein einstelliges Funktionssymbol g, $f(\cdot, \cdot)$ ein zweistelliges Funktionssymbol f usw. So bezeichnet z.B. $\Sigma = \{c(), g(\cdot), f(\cdot, \cdot)\}$ eine Signatur mit den oben angeführten Funktionssymbolen und Aritäten. Mit Σ^n bezeichnen wir die Menge aller n-nären Funktionssymbole der Signatur Σ.
Eine beliebige Menge $\mathcal{X}$ mit $\mathcal{X} \cap \Sigma = \emptyset$ bezeichnen wir als eine **für Σ passende Variablenmenge**.

Definition 2. Sei Σ eine Signatur, $\mathcal{X}$ eine Menge von Variablen. Die Menge $T(\Sigma, \mathcal{X})$, die **Terme der Signatur Σ über $\mathcal{X}$**, wird induktiv definiert durch:

1. Ein Element $x \in \mathcal{X}$ ist in $T(\Sigma, \mathcal{X})$.

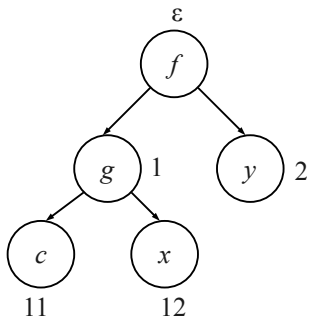


Abbildung 2.1: Darstellung des Terms $f(g(c,x),y)$ aus Beispiel 1 als Baum.

2. Sei $f \in \Sigma$ ein Funktionssymbol mit Arität $n \geq 0$, und gelte $t_1, ..., t_n \in T(\Sigma, \mathcal{X})$. Dann ist $f(t_1, ..., t_n) \in T(\Sigma, \mathcal{X})$. □

Wir nennen die t_i mit $i \leq n$ aus Definition 2 **direkte Teilterme** von $f(t_1, ..., t_n)$ und bezeichnen ihre Gesamtheit als

$$children(t) := \{t' \mid \exists n \in \mathbb{N} : \exists f \in \Sigma^n : (t = f(t_1, ..., t_n) \wedge \exists i \leq n : t' = t_i)\}.$$

Definieren wir die Relation $\sim \subseteq T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$ so, dass

$$\forall t, t' \in T(\Sigma, \mathcal{X}) : t' \sim t \;:\Leftrightarrow t' \text{ ist direkter Teilterm von } t,$$

so nennen wir den reflexiven und transitiven Abschluss von $\sim$ die **Teiltermrelation** $\leq \; := \; \sim^*$ von $T(\Sigma, \mathcal{X})$. Die Menge aller Variablen, die in einem Term t vorkommen, bezeichnen wir als $Var(t) := \{x \in \mathcal{X} \mid x \leq t\}$. Enthält ein Term keine Variablen, so bezeichnen wir ihn als **Grundterm**. Die Menge aller Grundterme wiederum bezeichnen wir mit $T(\Sigma, \emptyset)$ oder kurz $T(\Sigma)$. Wir erklären noch für eine Menge $M \in T(\Sigma, \mathcal{X})$ die Menge

$$SubTerms(M) := \{t \in T(\Sigma, \mathcal{X}) \mid \exists t' \in M : t \leq t'\},$$

also alle Teilterme, die in irgendeinem Term aus M vorkommen.
Eine anschauliche Darstellung von Termen sind gerichtete Bäume: Der einem Term entsprechende Baum enthält mit einem Funktionssymbol oder einer Variable beschriftete Knoten. Seine Kanten repräsentieren die direkte Teiltermrelation des Terms.

Beispiel 1. Sei die Signatur $\Sigma = \{f(\cdot,\cdot), g(\cdot,\cdot), h(\cdot), c()\}$ mit der Variablenmenge $\mathcal{X} = \{x, y, z\}$ gegeben. Dann ist z.B. $t = f(g(c,x),y)$ ein Element aus $T(\Sigma, \mathcal{X})$. Der dem Term entsprechende Baum ist in Abb. 2.1 dargestellt. □

Definition 3. Sei Σ eine Signatur, $\mathcal{X}$ eine dazu passende Variablenmenge. Die **Tiefe** eines Terms $depth(t)$ ist folgendermaßen definiert:

1. $t \in \Sigma^0 \cup \mathcal{X} \Rightarrow depth(t) := 0$.

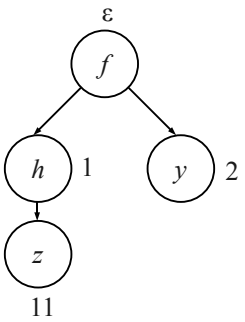


Abbildung 2.2: Darstellung der Termersetzung aus Beispiel 2.

2. $t = f(t_1, ..., t_n) \Rightarrow depth(t) := 1 + \max_{i \in \{1,...,n\}} depth(t_i)$. □

Durch die Baumdarstellung motiviert führen wir eine Nummerierung der in einem Term vorkommenden Funktionssymbole ein. Mit ihrer Hilfe können wir systematisch auf Teilterme zugreifen und Ersetzungen von Teiltermen vornehmen:

Definition 4. Sei Σ eine Signatur, $\mathcal{X}$ eine dazu passende Variablenmenge, und $s, t \in T(\Sigma, \mathcal{X})$. Dann ist die Menge der **Positionen** von s, $Pos(s) \subseteq \mathbb{N}^*$, folgendermaßen definiert:

1. $s \in \Sigma^0 \cup \mathcal{X} \Rightarrow Pos(s) := \{\epsilon\}$, wobei ϵ das leere Wort darstellt.
2. $s = f(s_1, ..., s_n) \Rightarrow Pos(s) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(s_i)\}$.

Des Weiteren ist der **Teilterm von** s **an der Position** p, bezeichnet durch $s|_p$, definiert durch

1. $s|_\epsilon := s$,
2. $f(s_1, ..., s_n)|_{iq} := s_i|_q$.

Sei $p \in Pos(s)$, so sei die **Ersetzung des Teilterms bei Position** p **durch** t **in** s, bezeichnet durch $s[t]_p$, definiert durch:

1. $s[t]_\epsilon := t$,
2. $f(s_1, ..., s_n)[t]_{iq} := f(s_1, ..., s_i[t]_q, ..., s_n)$. □

Beispiel 2. Wir betrachten nochmals den Baum aus Abb. 2.1 . Die Positionen sind neben den Knoten dargestellt. Der Wurzelknoten hat immer die Position ϵ. Beispielsweise gilt $t|_1 = g(c, x)$. Das Ergebnis der Ersetzung $t[h(z)]_1$ ist in Abb. 2.2 dargestellt. □

Wir werden im Folgenden nicht mehr zwischen der Darstellung eines Terms als Baum und der als Zeichenkette unterscheiden. Insbesondere wollen wir eine Darstellung eines Grundterms als Baum auch als **Grundbaum** bezeichnen. Mithilfe des Begriffs der Position lässt sich eine wichtige Termeigenschaft einfach definieren:

Definition 5. Ein Term $t \in T(\Sigma, \mathcal{X})$ ist **linear**, wenn für jede Variable $x \in \mathcal{X}$ höchstens eine Position $p \in Pos(t)$ existiert mit $t|_p = x$. □

Variablen und Konstanten haben viele Gemeinsamkeiten. Sie unterscheiden sich jedoch dadurch voneinander, dass Variablen, die in einem Term vorkommen, mittels Substitutionen durch andere Terme ersetzt werden können:

Definition 6. Sei Σ eine Signatur, $\mathcal{X}$ eine Menge von Variablen für Σ. Eine Abbildung $\sigma : \mathcal{X} \to T(\Sigma, \mathcal{X})$ ist eine **Substitution**, wenn $\sigma(x) \neq x$ für nur endlich viele $x \in \mathcal{X}$ gilt. Eine Substitution kann im Urbildbereich auf Terme $s \in T(\Sigma, \mathcal{X})$ zu einer Abbildung $\hat{\sigma} : T(\Sigma, \mathcal{X}) \to T(\Sigma, \mathcal{X})$ wie folgt erweitert werden:

1. $s = x \in \mathcal{X} \Rightarrow \hat{\sigma}(s) := \sigma(x)$
2. $s = f(s_1, ..., s_n) \Rightarrow \hat{\sigma}(s) := f(\hat{\sigma}(s_1), ..., \hat{\sigma}(s_n))$. □

Vereinfacht ausgedrückt können mithilfe von Substitutionen (endlich viele) Variablen eines Terms durch Teilterme ersetzt werden. Diese Teilterme dürfen durchhaus wieder Variablen enthalten. Wir werden im Folgenden nicht zwischen σ und $\hat{\sigma}$ unterscheiden: Wird σ auf einen zusammengesetzten Term oder eine Konstante angewandt, ist die Erweiterung $\hat{\sigma}$ gemeint. Die **Menge aller Substitutionen** über $T(\Sigma, \mathcal{X})$ wollen wir mit $Sub(T(\Sigma, \mathcal{X}))$ bezeichnen.

Beispiel 3. Wir betrachten den Term $f(g(c, x), y)$ aus Beispiel 1 und definieren eine Substitution $\sigma : \{x, y, z\} \to T(\Sigma, \mathcal{X})$ durch

$$\sigma(x) = c, \sigma(y) = f(z, z), \sigma(z) = z.$$

Dann ist $\hat{\sigma}(f(g(c, x), y)) = f(g(c, c), f(z, z))$. □

Termersetzungssysteme stellen eine Methode dar, auf einen Term bzw. eine Menge von Termen systematisch Ersetzungen für Teilterme anzuwenden.

Definition 7. Sei Σ eine Signatur, $\mathcal{X}$ eine Variablenmenge für Σ.
Eine **Ersetzungsregel** über $T(\Sigma, \mathcal{X})$ ist ein Tupel $(s, t) \in T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$, so dass gilt: $Var(t) \subseteq Var(s)$ und $s \notin \mathcal{X}$.
Ein **Termersetzungssystem** R über $T(\Sigma, \mathcal{X})$ ist eine Menge von Ersetzungsregeln über $T(\Sigma, \mathcal{X})$. Wir schreiben auch $s \to_R t$ wenn $(s, t) \in R$. Wir definieren weiterhin die Mengen $lhs(R) := \{t \in T(\Sigma, \mathcal{X}) \mid (t, t') \in R\}$ und $rhs(R) := \{t' \in T(\Sigma, \mathcal{X}) \mid (t, t') \in R\}$ der **linken** bzw. **rechten** Seiten des Termersetzungssystems R. □

Bei einer Regel $s \to_R t$ eines Termersetzungssystems R lassen wir oft den Index R weg, wenn der Kontext klar ist. Wir werden uns im Rahmen dieser Arbeit bis auf wenige Ausnahmen auf Termersetzungssysteme mit endlich vielen Ersetzungsregeln und einer endlichen Menge von Variablen beschränken. Wichtige Spezialfälle für Termersetzungssysteme sind die sogenannten **Grundtermersetzungssysteme**. Diese sind über $T(\Sigma, \emptyset)$ definiert, es kommen also in den Regeln keine Variablen vor. Der Name rührt daher, dass linke und rechte Seiten von Grundtermersetzungsregeln Grundterme darstellen.
Ein Termersetzungssystem induziert eine spezielle Relation über Termen:

Definition 8. Sei Σ eine Signatur und R ein Termersetzungssystem mit Regeln aus $T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$. Seien $t, t' \in T(\Sigma, \mathcal{X})$. Wir definieren die Relation $\hookrightarrow_R \subseteq T(\Sigma) \times T(\Sigma)$ wie folgt:

$$(t, t') \in \hookrightarrow_R$$

genau dann wenn

$$\exists (l, r) \in R : \exists \sigma \in Sub(T(\Sigma, \mathcal{X})) : \exists p \in Pos(t) : t|_p = \sigma(l) \wedge \sigma(t[r]_p) = t'. \quad \square$$

Wir schreiben auch $t \hookrightarrow_R t'$ für $(t, t') \in \hookrightarrow_R$. Der reflexive und transitive Abschluss von $\hookrightarrow_R$, die Relation $\hookrightarrow_R^*$, wird als **Ableitungsrelation** von R bezeichnet.

Da wir im Verlauf dieser Arbeit fast ausschließlich mit Grundtermersetzungssystemen arbeiten werden, geben wir eine weitere Definition für die Ableitungsrelation speziell für Grundtermersetzungssysteme, die ohne den Begriff der Substitution auskommt und mit der Definition 8 für den Speziallfall der Grundtermersetzungssysteme übereinstimmt.

Definition 9. Sei Σ eine Signatur, R ein Grundtermersetzungssystem mit Regeln aus $T(\Sigma) \times T(\Sigma)$. Seien $t, t' \in T(\Sigma)$. Wir definieren die Relation $\hookrightarrow_R \subseteq T(\Sigma) \times T(\Sigma)$ wie folgt:

$$(t, t') \in \hookrightarrow_R \;:\Leftrightarrow\; \exists (l, r) \in R : \exists p \in Pos(t) : t|_p = l \wedge t[r]_p = t'.$$

Wir schreiben auch $t \hookrightarrow_R t'$ für $(t, t') \in \hookrightarrow_R$. Der reflexive und transitive Abschluss von $\hookrightarrow_R$, die Relation $\hookrightarrow_R^*$, wird als **Ableitungsrelation** von R bezeichnet. $\square$

Beispiel 4. Sei $\Sigma = \{f(\cdot,\cdot), g(\cdot), a(), b()\}$ eine Signatur und R ein Grundtermersetzungssystem über Σ mit folgenden Regeln:

1. $a \to f(a, a)$,
2. $g(a) \to b$,
3. $f(a, a) \to g(g(a))$.

Nun gilt z.B. für den Term a: $a \hookrightarrow_R^* f(a, g(b))$. Eine Ableitungsfolge ist durch

- Regel 1: $a \hookrightarrow_R f(a, a)$,
- Regel 1: $f(a, a) \hookrightarrow_R f(a, f(a, a))$,
- Regel 3: $f(a, f(a, a)) \hookrightarrow_R f(a, g(g(a)))$,
- Regel 2: $f(a, g(g(a))) \hookrightarrow_R f(a, g(b))$ gegeben. $\square$

Durch die Ableitungsrelation $\hookrightarrow_R$ eines Grundtermersetzungssystems R wird eine für den weiteren Verlauf dieser Arbeit besonders wichtige Termmenge induziert:

Definition 10. Sei Σ eine Signatur, R ein Grundtermersetzungssystem über Σ. Sei weiterhin $M \subseteq T(\Sigma)$. Dann definieren wir die Menge **Post**(M) als

$$Post(M) := \{t \in T(\Sigma) \mid \exists t' \in M : t' \hookrightarrow_R t\}.$$

Desweiteren definieren wir die Menge **Post**$^*(M)$, die Menge aller **Nachfolgerterme** von M bezüglich R als

$$Post^*(M) := \bigcup_{i=0}^{n} Post^i(M).$$

□

Beispiel 5. Wir betrachten erneut das Grundtermersetzungssystem aus Bsp. 4. Es gilt z.B.: $Post(\{f(a,a)\}) = \{f(f(a,a),a), f(a,f(a,a)), g(g(a))\}$. Fügen wir dem System noch eine vierte Regel $b \to g(b)$ hinzu, so gilt beispielsweise: $Post^*(\{b\}) = \{g^i(b) \mid i \geq 0\}$. □

Analog zu $Post(M)$ und $Post^*(M)$ lassen sich die Mengen $Pre(M)$ und $Pre^*(M)$ definieren:

Definition 11. Sei Σ eine Signatur, R ein Grundtermersetzungssystem über Σ. Sei weiterhin $M \subseteq T(\Sigma)$. Dann definieren wir die Menge **Pre**(M) als

$$Pre(M) := \{t \in T(\Sigma) \mid \exists t' \in M : t \hookrightarrow_R t'\}.$$

Desweiteren definieren wir die Menge **Pre**$^*(M)$, die Menge aller **Vorgängerterme** von M bezüglich R als

$$Pre^*(M) := \bigcup_{i=0}^{n} Pre^i(M).$$

□

2.2 Baumautomaten

Baumautomaten können als Verallgemeinerung von endlichen Automaten für Zeichenketten aufgefasst werden. Sie akzeptieren Grundterme bzw. Grundbäume, ähnlich wie endliche Automaten Zeichenketten akzeptieren können. Es existieren viele verschiedene Realisierungen von Baumautomaten, die sich u.a. in Akzeptanzbedingungen, Abschlusseigenschaften oder der Menge der erkannten Baumsprachen unterscheiden. Der für uns wichtige Typ von Baumautomaten wird in der nächsten Definition eingeführt:

Definition 12. Ein **nichtdeterministischer Bottom-Up-Baumautomat** $\mathcal{A}$ **mit** ϵ**-Regeln** ist ein Tupel (Q, Σ, Q_f, δ), wobei gilt:

- Q ist eine endliche Menge von Zuständen,
- Σ ist eine Signatur, die mindestens eine Konstante enthält,
- $Q_f \subseteq Q$ ist die Menge der **akzeptierenden** Zustände,

- δ ist eine endliche Menge von Regeln der Form:
  - $f(q_1, ..., q_n) \to q$, mit $q_1, ..., q_n, q \in Q, f \in \Sigma^n$, oder
  - $q \to q'$, mit $q, q' \in Q$, die sogenannten **ϵ-Regeln**. □

Der Name Bottom-Up-Baumautomat rührt daher, dass Bäume durch den Automaten "von den Blättern" her abgeleitet werden, und zwar so, dass Teilterme durch Zustände ersetzt werden, bis zum Schluss der gesamte Baum auf einen Zustand reduziert worden ist. Eine Möglichkeit, Ableitungen oder Reduktionen von Bäumen durch einen Bottom-Up-Baumautomat zu definieren, besteht darin, die Regeln des Baumautomaten als Grundtermersetzungssystem zu interpretieren:

Definition 13. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat. Wir führen eine neue Signatur Σ' ein mit $\Sigma' = \Sigma \cup Q$, wobei für $q \in Q$ jeweils die Arität 0 festgelegt wird. Wir interpretieren δ als Grundtermersetzungssystem $R_{\mathcal{A}} := \delta$ über der Signatur Σ'. Dann gilt für $t \in T(\Sigma)$, dass $t \in T(\Sigma')$. Wir definieren die **Ableitungsrelation** $\to_{\mathcal{A}} \subseteq T(\Sigma') \times T(\Sigma')$ von $\mathcal{A}$ wie folgt:

$$\forall t, t' \in T(\Sigma') : t \to_{\mathcal{A}} t' :\Leftrightarrow t \hookrightarrow_{R_{\mathcal{A}}} t'.$$

Weiterhin definieren wir:

$$t \in T(\Sigma') \text{ kann in } \mathcal{A} \text{ zu Zustand } q \in Q \textbf{ reduziert} \text{ werden } :\Leftrightarrow t \to_{\mathcal{A}}^* q$$

($\hookrightarrow_R$ wurde in Def. 9 eingeführt.)
Ein Term $t \subseteq T(\Sigma)$ wird von $\mathcal{A}$ **akzeptiert** genau dann, wenn es ein $q \in Q_f$ gibt mit $t \to_{\mathcal{A}}^* q$. Eine Folge von Termen $(t, t_1, ..., t_n, t')$ mit $t \in T(\Sigma)$, $t' \in Q$, $t_1, ..., t_n \in T(\Sigma')$ mit $t \to_{\mathcal{A}} t_1 \to_{\mathcal{A}} ... \to_{\mathcal{A}} t_n \to_{\mathcal{A}} t'$ bezeichnen wir als **Reduktion** oder **Ableitung**. Sie ist **akzeptierend** wenn $t' \in Q_f$.
Die **von $\mathcal{A}$ akzeptierte Sprache** ist $\mathcal{L}(\mathcal{A}) := \{t \in T(\Sigma) \mid \exists q \in Q_f : t \to_{\mathcal{A}}^* q\}$. □

Wir wollen noch die Sprechweise einführen, dass ein Zustand q eines Baumautomaten $\mathcal{A}$ genau dann einen Term t **erkennt**, wenn es eine Reduktion für t gibt mit q als Endterm. Wir sagen dann auch, dass t zu q **abgeleitet** werden kann. Gelegentlich bezeichnen wir mit $\mathcal{L}(q)$ die Menge der Terme t, die von q erkannt werden.

Beispiel 6. Sei $\Sigma = \{f(\cdot,\cdot), g(\cdot), a(), b()\}$ eine Signatur. Wir betrachten den Baumautomaten $\mathcal{A} = \{\{q_1, q_2, q_f\}, \Sigma, \{q_f\}, \delta\}$ mit folgender Regelmenge δ:

1. $a \to q_1$,
2. $b \to q_1$,
3. $g(q_1) \to q_1$,
4. $f(q_1, q_1) \to q_2$,
5. $q_2 \to q_f$.

Wir geben eine Reduktion des Terms $f(g(b), g(a))$ in Baumform an:

$$\begin{array}{ccccccccccccc}
f & \rightarrow_{\mathcal{A}} & f & \rightarrow_{\mathcal{A}} & f & \rightarrow_{\mathcal{A}} & f & \rightarrow_{\mathcal{A}} & f & \rightarrow_{\mathcal{A}} & q_2 & \rightarrow_{\mathcal{A}} & q_f \\
\diagup\diagdown & & \diagup\diagdown & & \diagup\diagdown & & \diagup\diagdown & & \diagup\diagdown & & & & \\
g \quad g & & g \quad g & & g \quad g & & q_1 \quad g & & q_1 \quad q_1 & & & & \\
| \quad | & & | \quad | & & | \quad | & & | & & & & & & \\
b \quad a & & q_1 \quad a & & q_1 \quad q_1 & & q_1 & & & & & &
\end{array}$$

Die Reihenfolge der Ableitungsschritte ist nicht eindeutig, z.B. wäre auch eine Vertauschung der ersten beiden Ableitungsschritte möglich. Da $q_f \in Q_f$ gilt, stellt die obige Reduktion eine akzeptierende Reduktion dar. Damit gilt: $f(g(b), g(a)) \in \mathcal{L}(\mathcal{A})$. □

Es sollen noch besondere Klassen von Bottom-Up-Baumautomaten erwähnt werden:

Definition 14. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Bottom-Up-Baumautomat.

- $\mathcal{A}$ ist **deterministisch** genau dann, wenn er keine ϵ-Regeln enthält und für alle Regeln $r_1 = t_1 \rightarrow t_1'$, $r_2 = t_2 \rightarrow t_2'$ mit $r_1, r_2 \in \delta$, $r_1 \neq r_2$ gilt: $t_1 \neq t_2$.
- $\mathcal{A}$ ist **vollständig** genau dann, wenn es für jede Arität n, jedes Funktionssymbol $f \in \Sigma^n$ und für jede Zustandsfolge $(q_1, ..., q_n) \in Q^n$ ein $q \in Q$ und eine Regel $r \in \delta$ gibt mit $r = f(q_1, ..., q_n) \rightarrow q$. □

Wir merken außerdem an, dass es für jeden Baumautomaten $\mathcal{A}$ einen deterministischen Baumautomaten $\mathcal{A}'$ gibt mit $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. Der Beweis hierzu ist analog zu dem im Falle endlicher Wortautomaten (siehe [CDG+07]). Wir wollen aber die Konstruktion eines solchen deterministischen Baumautomaten angeben, da wir sie in modifizierter Form später verwenden werden.
Zunächst eine Bemerkung: Jeder Baumautomat $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ kann in einen Automaten umgewandelt werden, der keine ϵ-Regeln enthält und dieselbe Sprache erkennt. Dies kann z.B. dadurch geschehen, dass wir eine ϵ-Regel $q \rightarrow q'$ durch die Menge von Regeln $\{f(q_1, ..., q_n) \rightarrow q' \mid f(q_1, ..., q_n) \rightarrow q \in \delta\}$ ersetzen. Es stellt daher keine echte Einschränkung dar, wenn wir uns in folgenden Definitionen auf Baumautomaten ohne ϵ-Regel beschränken.

Definition 15. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln. Der **Potenzmengenautomat** $\mathcal{A}_\mathcal{P}$ für $\mathcal{A}$ ist definiert durch

$$\mathcal{A}_\mathcal{P} = (\mathcal{P}(Q), \Sigma, \{M \in \mathcal{P}(Q) \mid M \cap Q_f \neq \emptyset\}, \delta_\mathcal{P}),$$

wobei eine Regel $f(M_1, ..., M_n) \rightarrow M$ mit $M_1, ..., M_n, M \subseteq Q$ genau dann in $\delta_\mathcal{P}$ ist, wenn $M = \{q \in Q \mid \exists q_1 \in M_1 ... \exists q_n \in M_n : f(q_1, ..., q_n) \rightarrow q \in \delta\}$. □

Aus Effizienzgründen werden bei der Konstruktion des Potenzmengenautomaten in der vorliegenden Implementation nur produktive Zustände berücksichtigt (siehe Abschnitt 2.2.2), so dass die Automaten meist wesentlich weniger Zustände als $2^{|Q|}$ besitzen.

Eine wichtige Eigenschaft eines deterministischen Automaten $\mathcal{A}$ über der Signatur Σ besteht darin, dass es für einen Term $t \in T(\Sigma)$ höchstens einen Zustand q von $\mathcal{A}$ gibt, für den $t \rightarrow^*_{\mathcal{A}} q$ gilt. Ist der deterministische Automat zusätzlich vollständig, so gibt es genau einen solchen Zustand.
Gilt für eine Menge M von Bäumen bzw. Termen, dass es einen Bottom-Up-Baumautomaten $\mathcal{A}$ gibt mit $\mathcal{L}(\mathcal{A}) = M$, so nennen wir die Menge M **regulär**.
Im weiteren Verlauf der Arbeit sei mit Baumautomat oder auch nur Automat immer ein Bottom-Up-Baumautomat gemeint, soweit nicht anders angegeben.

2.2.1 Eigenschaften

Nicht nur in der Definition und Arbeitsweise ähneln Baumautomaten den endlichen Automaten für Zeichenketten; viele bekannte Algorithmen für endliche Wortautomaten können auf den Baumautomatenfall verallgemeinert werden. Ebenso gelten viele nützliche Eigenschaften der endlichen Wortautomaten auch für Baumautomaten. In diesem Abschnitt wollen wir einige der für uns wichtigsten Eigenschaften und Verfahren näher betrachten.
Zunächst bemerken wir, dass die regulären Baumsprachen unter booleschen Mengenoperationen abgeschlossen sind, wobei die Argumentation ähnlich der im Falle endlicher Automaten geführt wird:

Satz 1. Sei Σ eine Signatur. Seien $\mathcal{L}_1, \mathcal{L}_2 \in T(\Sigma)$ zwei reguläre Baumsprachen. Dann gilt: $\mathcal{L}_1 \cup \mathcal{L}_2$, $\mathcal{L}_1 \cap \mathcal{L}_2$ sowie $\overline{\mathcal{L}_1} = T(\Sigma) \setminus \mathcal{L}_1$ sind reguläre Baumsprachen.□

Beweis. Seien $\mathcal{A}_1 = (Q_1, \Sigma, F_1, \delta_1)$ und $\mathcal{A}_2 = (Q_2, \Sigma, F_2, \delta_2)$ Baumautomaten mit $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}_1$, $\mathcal{L}(\mathcal{A}_2) = \mathcal{L}_2$. O.B.d.A. soll $Q_1 \cap Q_2 = \emptyset$ gelten und die Baumautomaten ϵ-Regel-frei sein.

- $\mathcal{L}_1 \cup \mathcal{L}_2$:
  Wir konstruieren einen Automaten $\mathcal{A} = (Q_1 \cup Q_2, \Sigma, F_1 \cup F_2, \delta_1 \cup \delta_2)$. Nun gilt für alle $t \in T(\Sigma)$ wegen der Disjunktheit der Zustandsmengen:

$$\begin{aligned} t \in \mathcal{L}(\mathcal{A}) &\Leftrightarrow \exists q \in F_1 \cup F_2 : t \rightarrow^*_{\mathcal{A}} q \\ &\Leftrightarrow \exists q \in F_1 : t \rightarrow^*_{\mathcal{A}} q \vee \exists q \in F_2 : t \rightarrow^*_{\mathcal{A}} q \\ &\Leftrightarrow \exists q \in F_1 : t \rightarrow^*_{\mathcal{A}_1} q \vee \exists q \in F_2 : t \rightarrow^*_{\mathcal{A}_2} q \\ &\Leftrightarrow t \in \mathcal{L}(\mathcal{A}_1) \vee t \in \mathcal{L}(\mathcal{A}_2) \\ &\Leftrightarrow t \in \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2) \\ &\Leftrightarrow t \in \mathcal{L}_1 \cup \mathcal{L}_2. \end{aligned}$$

- $\mathcal{L}_1 \cap \mathcal{L}_2$: Wir konstruieren einen Automaten $\mathcal{A} = (Q, \Sigma, F, \delta)$ mit $Q = Q_1 \times Q_2$ und $F = F_1 \times F_2$. δ enthält für jedes $f \in \Sigma^n$ Regeln der Form

$$f((q_1, q_1'), ..., (q_n, q_n')) \rightarrow (q, q') \text{ mit } (q_1, q_1'), ..., (q_n, q_n'), (q, q') \in Q,$$

  für die gilt:

$$\exists f(q_1, ..., q_n) \rightarrow q \in \delta_1 \wedge \exists f(q_1', ..., q_n') \rightarrow q' \in \delta_2.$$

Wir simulieren also bei jedem Reduktionsschritt von $\mathcal{A}$ Reduktionsschritte in $\mathcal{A}_1$ und $\mathcal{A}_2$. Nun gilt für alle $t \in T(\Sigma)$:

$$\begin{aligned} t \in L(\mathcal{A}) &\Leftrightarrow \exists (q, q') \in F_1 \times F_2 : t \rightarrow^*_{\mathcal{A}} (q, q') \\ &\Leftrightarrow \exists q \in F_1 : t \rightarrow^*_{\mathcal{A}_1} q \wedge \exists q' \in F_2 : t \rightarrow^*_{\mathcal{A}_2} q' \\ &\Leftrightarrow t \in \mathcal{L}(\mathcal{A}_1) \wedge t \in \mathcal{L}(\mathcal{A}_2) \\ &\Leftrightarrow t \in \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) \\ &\Leftrightarrow t \in \mathcal{L}_1 \cap \mathcal{L}_2. \end{aligned}$$

- $\overline{\mathcal{L}_1}$:
  Wir können o.B.d.A. annehmen, dass $\mathcal{A}_1$ deterministisch und vollständig ist. Wir bilden nun einen ebenfalls deterministischen und vollständigen Baumautomaten $\overline{\mathcal{A}_1} = (Q, \Sigma, Q \setminus F_1, \delta)$. Wie schon erwähnt gilt für den deterministischen, vollständigen Baumautomaten $\overline{\mathcal{A}_1}$ und einen Term $t \in T(\Sigma)$: Es gibt genau einen Zustand $q \in Q$ mit $t \rightarrow^*_{\overline{\mathcal{A}_1}} q$. Daher gilt für alle $t \in T(\Sigma)$:

$$\begin{aligned} t \in L(\overline{\mathcal{A}_1}) &\Leftrightarrow \exists q \in Q \setminus F_1 : t \rightarrow^*_{\overline{\mathcal{A}_1}} q \\ &\Leftrightarrow \exists q \in Q \setminus F_1 : t \rightarrow^*_{\mathcal{A}_1} q \\ &\Leftrightarrow t \notin \mathcal{L}(\mathcal{A}_1) \\ &\Leftrightarrow t \notin \mathcal{L}_1. \end{aligned}$$

□

Wir bezeichnen den Automaten $\mathcal{A}$, der beim Beweis der Abgeschlossenheit des Schnitts benutzt wurde, auch als **Produktautomaten**, die dazu gehörigen Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ als **Faktorautomaten**. Meist schreiben wir für den Produktautomaten von $\mathcal{A}_1$ und $\mathcal{A}_2$ kurz $\mathcal{A}_1 \times \mathcal{A}_2$.

2.2.2 Entscheidungsprobleme

Problem 1. Ein Baumautomat $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ sei gegeben. Gilt $\mathcal{L}(\mathcal{A}) = \emptyset$?

Um dieses Problem effektiv lösen zu können, definieren wir eine spezielle Eigenschaft von Baumautomaten:

Definition 16. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat.
Einen Zustand $q \in Q$ nennen wir **produktiv**, wenn es einen Term $t \in T(\Sigma)$ gibt mit $t \rightarrow^*_{\mathcal{A}} q$.
Ein Zustand q ist **unproduktiv**, wenn er nicht produktiv ist.
Weiterhin nennen wir den Automaten $\mathcal{A}$ **bereinigt**, wenn er keinen unproduktiven Zustand $q \in Q$ enthält. □

Jeder Baumautomat $\mathcal{A}$ kann mithilfe des Algorithmus 2.3 zu einem bereinigten Automaten überführt werden, der dieselbe Sprache erkennt. Daraus ergibt sich sofort:

REMOVEUNPRODUCTIVESTATES(Automat $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$)

```
1  ▷ Erkennen der produktiven Zustände
2  rules := δ
3  productive := ∅
4  change := true
5  while change
6      do
7          change := false
8          for r ∈ rules
9              do
10                 if (r = f(q_1, ..., q_n) → q ∧ {q_1, ..., q_n} ⊆ productive)
                     ∨ (r = q' → q ∧ q' ∈ productive)
11                   then
12                       productive := productive ∪ {q}
13                       rules := rules \ {f(q'_1, ..., q'_n) → q ∈ δ}
14                       rules := rules \ {q̂ → q ∈ δ | q̂ ∈ Q}
15                       change := true
16 ▷ Unproduktive Zustände entfernen
17 return (Q ∩ productive, Σ, Q_f ∩ productive, δ \ rules)
```

Abbildung 2.3: Die Funktion REMOVEUNPRODUCTIVESTATES.

Satz 2. Problem 1 ist für jeden Baumautomaten $\mathcal{A}$ entscheidbar. □

Beweis. Nach Anwenden von Algorithmus 2.3 wird überprüft, ob der resultierende Automat mindestens einen akzeptierenden Zustand enthält. Wenn nicht, gib "Ja" aus, ansonsten "Nein". □

Durch die effektive Konstruktion von Baumautomaten für die obigen Mengenoperationen und den Leerheitstest ist auch das folgende Äquivalenzproblem effektiv berechenbar:

Problem 2. Zwei Baumautomaten $\mathcal{A}_1$, $\mathcal{A}_2$ seien gegeben.
Gilt $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$?

Für zwei reguläre Baumsprachen $\mathcal{L}_1, \mathcal{L}_2$ gilt $\mathcal{L}_1 = \mathcal{L}_2$ genau dann, wenn $(\mathcal{L}_1 \cap \overline{\mathcal{L}_2}) \cup (\mathcal{L}_2 \cap \overline{\mathcal{L}_1}) = \emptyset$, wobei wir sowohl sämtliche Mengenoperationen berechnen als auch den Leerheitstest effektiv entscheiden können. Daraus ergibt sich:

Korollar 3. Problem 2 ist entscheidbar. □

2.2.3 Minimale Baumautomaten

Für die im Verlauf dieser Arbeit entstandenen Implementierung wurde eine andere Möglichkeit als die obige gewählt, Äquivalenztests für Baumautomaten durchzuführen. Sie basiert auf der Beobachtung, dass sich für Bau-

mautomaten ähnlich wie im Falle endlicher Automaten eine Automaten-Minimalform definieren und berechnen lässt.
Wir erwähnen folgenden Satz ohne Beweis (siehe dazu z.B. [CDG+07]):

Satz 4. Für jeden Baumautomaten $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ existiert ein vollständiger, deterministischer Baumautomat $\mathcal{A}_{min}$ mit $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{min})$, der eine minimale Anzahl von Zuständen besitzt. Wir bezeichnen ihn als den **deterministischen Minimalautomat** von $\mathcal{A}$. Dieser Automat $\mathcal{A}'$ ist bis auf Umbenennung der Zustände eindeutig für jeden Automaten $\mathcal{A}'$ mit $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. □

Zu möglichen Algorithmen für die Erstellung von Minimalautomaten sei auf das Kapitel 5.6.2 verwiesen. Problematisch für die Berechnung ist, dass bei allen betrachteten Verfahren ein (nichtdeterministischer) Baumautomat zunächst deterministisch gemacht werden muss. Dies kann zu einer exponentiellen Aufblähung der Zustandsmenge führen. Für den Äquivalenztest wird Folgendes ausgenutzt:

Korollar 5. Zwei Automaten sind genau dann sprachäquivalent, wenn sie bis auf Umbenennung der Zustände denselben deterministischen Minimalautomaten besitzen. □

3 Funktionale Programmierung und Termersetzungssysteme

In diesem Kapitel soll erläutert werden, wie funktionalen Programmen eine operationale Semantik in Form von Termersetzungssystemen gegeben werden kann. Funktionale Programmierung stellt wie bereits erwähnt das Konzept der mathematischen Funktion und ihrer Auswertung in den Mittelpunkt der Programmerstellung. Ein funktionales Programm besteht aus einer oder mehrerer Funktionsdefinitionen. Eine Programmausführung besteht aus der Auswertung eines beliebig verschachtelten Funktionsaufrufs. Rein funktionale Programmierung verbietet die Verwendung von Zuweisungen, die bei imperativen Programmen eine äußerst wichtige Rolle spielen. Wir wollen zunächst ein Beispiel betrachten.

Beispiel 7. Im Folgenden ist eine einfache Funktionsdeklaration in der Programmiersprache ML gegeben, die eine Funktion `append` zur Konkatenation zweier Listen realisiert:

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

`[]` bezeichnet hier die leere Liste. Der Operator `::` fügt seinen ersten Operanden an den zweiten Operanden, eine Liste, an. Die Funktionsdefinition, eingeleitet durch das Schlüsselwort `fun`, besteht aus einer Fallunterscheidung der Argumente, die dem Funktionsnamen `fun` folgen (eigentlich stellt das Konstrukt ein Pattern-Matching dar). Ist das erste Argument die leere Liste, so ist das Ergebnis des Funktionsaufrufs das zweite Argument. Ist das erste Argument eine Liste, bestehend aus dem Listenkopf `x` und einer Restliste `xs`, so wird `x` an den Funktionswert von `append xs ys` angehängt. Für eine Einführung in ML sei beispielsweise auf [Bos95] verwiesen. □

Mit funktionaler Programmierung werden meist noch andere Konzepte verbunden, wie z.B. Funktionen höherer Ordnung oder *lazy evaluation*, bei der es möglich ist, Argumente einer Funktion erst dann auszuwerten, wenn sie unmittelbar benötigt werden. Diese Aspekte funktionaler Programme werden im Rahmen dieser Arbeit nicht weiter berücksichtigt.

3.1 Ein Modell für funktionale Programme

Wir werden im Folgenden funktionale Programme in abstrakter Form betrachten und definieren daher zunächst ein für uns geeignetes Modell.

Definition 17. Ein **funktionales Programm** ist ein Tupel $P = (\Sigma, \mathcal{X}, \Theta, \mathit{type}, \mathit{defs})$ mit

- einer Signatur Σ,
- einer Menge von Variablen $\mathcal{X}$,
- einer Menge von **Typen** $\Theta \subseteq \mathcal{P}(T(\Sigma))$,
- einer Abbildung $\mathit{type} : \mathcal{X} \to \Theta$.
- einem Termersetzungssystem defs von Termpaaren über $T(\Sigma, \mathcal{X})$, für deren Elemente $(t_l, t_r) \in T(\Sigma, \mathcal{X})$ gilt: $\mathit{Var}(t_l) \subseteq \mathit{Var}(t_r)$. □

Das Termersetzungssystem defs stellt gleichsam die Funktionsdefinitionen des Programms dar, wobei die Wertebereiche der Variablen bzw. Argumente der Funktionen durch die Angabe von Typen eingeschränkt werden. Wir bemerken, dass dieses Modell auch Nichtdeterminismus des Programms ermöglicht: Nichts spricht z.B. dagegen, dass ein Term $t \in T(\Sigma, \mathcal{X})$ als linke Seite zweier verschiedener Regeln des Termersetzungssystems defs vorkommt.
Funktionale Programme werden meist in interpretativer Manier gebraucht: Der Benutzer gibt einen funktionalen Ausdruck an, der mithilfe der Funktionsdefinitionen des Programms ausgewertet bzw. modifiziert wird. Dies geschieht solange, bis keine Funktionsdefinition mehr auf den Ausdruck anwendbar ist. Wir möchten das Verfahren auf unser Modell übertragen und bedienen uns dazu eines (potentiell unendlichen) Grundtermersetzungssystems. Hierzu müssen wir die Variablen auf ihre durch Typen gegebene Wertebereiche einschränken. Dies erreichen wir mithilfe der folgenden Definitionen:

Definition 18. Es sei $P = (\Sigma, \mathcal{X}, \Theta, \mathit{type}, \mathit{defs})$ ein funktionales Programm. Eine Substitution $\sigma \in T(\Sigma, \mathcal{X})$ heißt **gültig** für eine Regel $r = t_l \to t_r \in \mathit{defs}$ genau dann, wenn für alle $x \in Var(t_l)$ gilt: $\sigma(x) \in \mathit{type}(x)$. □

Definition 19. Es sei $P = (\Sigma, \mathcal{X}, \Theta, \mathit{type}, \mathit{defs})$ ein funktionales Programm. Dann bezeichnet R_P das Grundtermersetzungssystem über Σ mit allen Termpaaren $(t, t') \in T(\Sigma) \times T(\Sigma)$, für die eine Regel $r = t_l \to t_r \in \mathit{defs}$ existiert mit folgender Eigenschaft:

$$\exists \sigma \in \mathit{Sub}(T(\Sigma, \mathcal{X})) : \sigma \text{ ist gültig für } r \text{ und } \sigma(t) = t_l \text{ und } \sigma(t') = t_r. \qquad \square$$

Das Grundtermersetzungssystem R_P stellt also eine "Entfaltung" der Regeln aus defs dar, wobei berücksichtigt wird, dass die Variablen in den Regeltermen nicht durch beliebige Terme substituiert werden können, sondern nur durch Grundterme, die ihrem jeweiligen Typ entsprechen.

Definition 20. Es sei $P = (\Sigma, \mathcal{X}, \Theta, \mathit{type}, \mathit{defs})$ ein funktionales Programm und $t \in T(\Sigma)$ ein Grundterm. Wir ordnen t die Menge

$$C_{t,P} := \{t' \in \mathit{Post}^*(t) \mid \mathit{Post}^*(\{t'\}) = \{t'\}\}$$

zu und bezeichnen sie als die **Reduktionen** von t in P. □

Reduktionen sind also solche Nachfolgerterme von t, auf die keine Regel des funktionalen Programms bzw. des Grundtermersetzungssystems mehr anwendbar sind. Man kann sie als das *Resultat* von Funktionsauswertungen interpretieren. Man beachte, dass durchaus mehr als eine Reduktion pro Term und funktionalem Programm vorkommen kann. In diesem Fall zeigt das Programm **nichtdeterministisches** Verhalten. Ebenso ist es möglich, dass es für einen Term gar keine Reduktion in einem Programm gibt. In diesem Fall kann jeder Nachfolgerterm zu einem anderen weiter abgeleitet werden: Das Programm **terminiert nicht**.

Beispiel 8. Wir geben eine Möglichkeit an, Beispiel 7 zu modellieren, wobei wir die Menge der Listenelemente auf zwei Elemente 0 und 1 einschränken. Wir benutzen die Signatur $\Sigma = \{0(), 1(), [](), c(\cdot,\cdot), \mathit{append}(\cdot,\cdot)\}$. Die leere Liste entspricht der Konstanten $[]$. Eine Liste $[a_1, ..., a_n]$ wird durch den Term $c(a_1, ..., c(a_n, []))$ repräsentiert. Ein entsprechendes funktionales Programm $P = (\Sigma, \mathcal{X}, \Theta, \mathit{type}, \mathit{defs})$ ist durch folgende Bestandteile gegeben:

- $\mathcal{X} = \{x, xs, ys\}$,
- $\Theta = \{Consts, Lists\}$, wobei
- $Consts = \{0, 1\}$ und
- $Lists = \{\,[]\,\} \cup \{c(a_1, c(a_2, ..., c(a_n, []))...) \mid a_1, ..., a_n \in Consts, n \geq 1\}$,
- $\mathit{type} : \mathcal{X} \to \Theta$ mit $\mathit{type}(x) = Consts$, $\mathit{type}(xs) = \mathit{type}(ys) = Lists$,
- defs enthält zwei Regeln:
  - $\mathit{append}([], ys) \to ys$
  - $\mathit{append}(c(x, xs), ys) \to c(x, \mathit{append}(xs, ys))$.

Da $Consts$ unendlich viele Elemente enthält, enthält

$$R_P = \{\mathit{append}([], []) \to [], \mathit{append}([], c(0, [])) \to c(0, []), ...\} \cup$$

$$\{\mathit{append}(c(0, []), []) \to c(0, \mathit{append}([], [])),$$

$$\mathit{append}(c(0, []), c(0, [])) \to c(0, \mathit{append}(c(0, []), [])), ...\}$$

auch unendlich viele Regeln. Es gibt maximal eine Reduktion für jeden Term aus $T(\Sigma)$, da kein Term als linke Seite zweier verschiedener Regeln von R_P vorkommt. Z.B. ist die einzige Reduktion des Terms $c(0, [])$ der Term $c(0, [])$ selbst, die einzige Reduktion des Terms $\mathit{append}(c(0, []), c(1, []))$ ist der Term $c(0, c(1, []))$. □

3.2 Testen funktionaler Programme

Im Rahmen des Testens funktionaler Programme sollte insbesondere folgende Fragestellung beantwortet werden:

Problem 3. Gegeben sei ein funktionales Programm $P = (\Sigma, \mathcal{X}, \Theta, type, defs)$, eine **Anfangsmenge** $M \subseteq T(\Sigma)$ und eine **Testmenge** $G \subseteq T(\Sigma)$. Gilt $G \cap Post^*_{R_P}(M) = \emptyset$? □

Häufig besteht die Testmenge G aus Termen, die laut Spezifikation des Programms nicht ableitbar sein sollten.

Beispiel 9. Wir betrachten das Programm aus Beispiel 8. Zunächst definieren wir

$$Even = \{[]\} \cup \{c(a_1, c(a_2, ..., c(a_n, [])...)) \mid n \geq 0 \wedge \forall i \in \{1, ..., n\} : a_i \in \{0, 1\} \wedge 2 \mid n\}$$

und

$$Odd = \{c(a_1, c(a_2, ..., c(a_n, [])...)) \mid n \geq 1 \wedge \forall i \in \{1, ..., n\} : a_i \in \{0, 1\} \wedge 2 \not| n\},$$

also die Menge Mengen aller Listen gerader bzw. ungerader Länge.
Wir wählen als Anfangsmenge

$$M = \{append(t_1, t_2) \mid t_1, t_2 \in Even\},$$

als Testmenge G wählen wir Odd. Die Konkatenation zweier Listen gerader Länge ergibt wieder eine Liste gerader Länge. Sollte $append$ wirklich eine Funktion zum Konkatenieren zweier Listen definieren, so müsste gelten:

$$Post^*_{R_P}(M) \cap Odd = \emptyset.$$

Wäre dem nicht so, wäre der Schnitt leer. Anhand der Schnittelemente könnte dann gezielt nach dem Fehler der Funktionsdefinition gesucht werden. □

Das in Definition 19 eingeführte Grundtermersetzungssystem für ein funktionales Programm ist im Allgemeinen unendlich groß, zumindest, wenn wir für mindestens einen Typ eine unendliche Grundtermmenge wählen. Die Menge der Nachfolger $Post^*(\{t\})$ eines Terms t für ein Grundtermersetzungssystem mit unendlich vielen Regeln ist jedoch nicht berechenbar, da solche Systeme Turing-mächtig sind (siehe [BN98]).
Beschränken wir uns jedoch auf endliche Mengen von Typen und damit auf endliche Grundtermersetzungssysteme, so kann hierfür $Post^*(M)$ effektiv berechnet werden, wenn die Menge M in Form eines Baumautomaten vorliegt, also regulär ist. Algorithmen für diese Aufgabenstellung werden im nächsten Abschnitt vorgestellt.
Wir wollen noch anmerken, dass es für die Untersuchung funktionaler Programme auch sinnvoll sein kann, statt der Nachfolgermenge von Termen deren Vorgängermenge bezüglich eines Grundtermersetzungssystems zu berechnen. Beispielsweise könnte man daran interessiert sein, welche Terme bzw. Funktionsaufrufe einen bestimmten Term als Ergebnis haben. Wir sprechen hierbei analog zum $Post^*$-Problem vom Pre^*-Problem. Das Pre^*-Problem lässt sich mit denselben Algorithmen wie das $Post^*$-Problem berechnen: Lediglich die Rollen der linken und rechten Seiten der Regeln des Grundtermersetzungssystems müssen vertauscht werden. Wir werden daher im weiteren Verlauf nur die Berechnung von $Post^*$ betrachten; alle Erkenntnisse lassen sich direkt auf die Berechnung von Pre^* übertragen.

4 Erreichbarkeitsanalysen

Im vorigen Abschnitt wurde gezeigt, wie das Verhalten funktionaler Programme mithilfe von Grundtermersetzungssystemen modelliert werden kann. Besitzt das für ein funktionales Programm P definierte Grundtermersetzungssystem R_P nur endlich viele Regeln, existiert ein Algorithmus, der die Nachfolgermenge $Post^*(M)$ einer regulären Menge M von Termen effizient berechnen kann (siehe [Löd02]).
Um die Struktur von Grundtermersetzungssystemen auszunutzen, die bei der Modellierung funktionaler Programm entstehen, wurde im Rahmen dieser Arbeit eine symbolische Variante des Algorithmus aus [Löd02] bzw. [Löd06] entwickelt. Wir stellen zunächst den nichtsymbolischen Ansatz vor.

4.1 Nichtsymbolischer Ansatz

Wir formulieren zunächst das Problem 3 für den hier dargestellten Ansatz um:

Problem 4. Sei Σ eine Signatur. Es sei ein endliches Grundtermersetzungssystem R über Σ gegeben und ein Baumautomat $\mathcal{A}_0$ mit $\mathcal{L}(\mathcal{A}_0) = M$. Gesucht ist die Menge $Post^*(M)$ in Form eines Baumautomaten. □

Existiert ein Algorithmus für das obige Problem, ist damit auch gezeigt, dass die regulären Baumsprachen unter $Post^*$-Bildung bezüglich eines endlichen Grundtermersetzungssystems abgeschlossen sind.
Die Grundidee des Ansatzes aus [Löd02] lässt sich folgendermaßen darstellen: Kann in $\mathcal{A}_0$ die linke Seite einer Regel aus R zu einem Zustand q abgeleitet werden, dann sollte auch die rechte Seite der Regel zu q abgeleitet werden können. Dazu werden dem Automaten zu Beginn des Algorithmus Regeln und Zustände hinzugefügt: Für jede Regel $r = t_l \to t_r$ aus R existiert dann ein Zustand q_r in $\mathcal{A}_0$, zu dem nur die rechte Seite t_r der Regel abgeleitet werden kann. Entdeckt der Algorithmus nun einen Zustand q in $\mathcal{A}_0$ mit $t_l \hookrightarrow_R^* q$, wird $\mathcal{A}_0$ um eine Regel $q_r \to q$ zu einem Automaten $\mathcal{A}_1$ erweitert: Dann gilt $t_r \to_{\mathcal{A}_1}^* q$.
Ist ein Automat $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ und ein Regelsystem R über Σ gegeben, können wir ihn beispielsweise zu einem Automat $\mathcal{A}_0 = (Q \cup Q_R, \Sigma, Q_f, \delta \cup \delta_R)$ mit

- $Q_R = \{\ q_t \in T(\Sigma) \mid t \in SubTerms(rhs(R))\}$
- $\delta_R = \{\ f(q_{t_1}, ..., q_{t_n}) \to q_t \mid t = f(t_1, ..., t_n) \wedge q_t, q_{t_1}, ..., q_{t_n} \in Q_R\ \}$

erweitern. Für einen Term $t \in rhs(R)$ bezeichnet dann q_t den Zustand mit $t' \to_{\mathcal{A}}^* q_t \Leftrightarrow t' = t$, der oben gefordert wurde. Der Algorithmus ist in Abb.

CALCPOST(Automat $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$, Grundtermersetzungssystem R)

1 $Q' := Q \cup Q_R$
2 $\delta_0 := \delta \cup \delta_R$
3 $\mathcal{A}_0 := (Q_0, \Sigma, Q_f, \delta_0)$
4 $i := 0$
5 **while** $\exists\, q \in Q' : \exists\, t_l \to t_r \in R : (t_l \to^*_{\mathcal{A}_i} q \wedge t_r \not\to^*_{\mathcal{A}_i} q)$
6 **do**
7 $i := i + 1$
8 $\delta_i := \delta_{i-1} \cup \{\, q_{t_r} \to q \,\}$
9 $\mathcal{A}_i := (Q', \Sigma, Q_f, \delta_i)$
10 return $\mathcal{A}_i$

Abbildung 4.1: Algorithmus aus [Löd02].

4.1 dargestellt. Für Terminierung, Korrektheit und weitere Eigenschaften des Algorithmus sei auf [Löd02] verwiesen.

4.2 Motivation für einen symbolischen Ansatz

Schwierigkeiten des Ansatzes aus 4.1 bei der Anwendung auf funktionale Programme ergeben sich vor allem aus der Größe des dabei verwendeten Grundtermersetzungssystems. Wir fügen zu Beginn des Algorithmus für jede Regel des Systems Zustände und Regeln dem Anfangsautomat hinzu. Bei vielen Beispielen aus der Praxis funktionaler Programmierung, die z.B. Listen begrenzter Länge als Datenstrukturen verwenden, muss bei diesem Verfahren mit einem großen Anfangsautomat gerechnet werden: Meist besitzt er exponentiell viele Zustände in der Länge der Listen.

Beispiel 10. Wir greifen Beispiel 8 auf: Wählen wir für $Lists$ die Menge

$$\{[]\} \cup \{c(a_1, c(a_2, ..., c(a_n, []))...) \mid a_1, ..., a_n \in Consts, n \leq 2\}$$

$$= \{[], c(0, []), c(1, []), c(0, c(0, [])), c(0, c(1, [])), c(1, c(0, [])), c(1, c(1, []))\},$$

so besteht R_P aus insgesamt 35 entfalteten Regeln. Erhöht man die maximale Länge der Listen z.B. auf $n = 7$, so besteht R_p aus 130305 Regeln. Ein in kanonischer Weise erstellter Automat, der für jede dieser Regeln einen Zustand enthält, welcher genau die rechte Zustandsseite ableiten kann, enthält mehr als 2^{2n} Zustände. □

Sowohl in Anbetracht des Speicherverbrauchs als auch der Laufzeit stößt das Verfahren aus 4.1 daher bald an die Grenzen der praktischen Einsetzbarkeit.
Ein Spezialfall dieses Algorithmus wurde bereits für endliche Wortautomaten und Pushdown-Systeme in verschiedenen Model-Checking-Tools angewandt, siehe [EHRS00]. Wegen der auch bei diesem Spezialfall sehr großen

Zustandsräume der Wortautomaten wurde ein symbolisches Verfahren entwickelt, siehe [ES01]. Hierbei konnten mithilfe von BDDs große Zustands- als auch Regelmengen endlicher Automaten effizient dargestellt und manipuliert werden.
Im nächsten Abschnitt wird ein Algorithmus vorgestellt, der ähnliche Konzepte auf den hier betrachteten allgemeineren Fall überträgt, um so die Zustandsexplosion bei der Entfaltung der Termersetzungssysteme einzudämmen.

4.3 Symbolische Erreichbarkeitsanalyse

4.3.1 Symbolische Grundtermersetzungssysteme

Grundtermersetzungssysteme, die beim Untersuchen eines funktionalen Programms $P = (\Sigma, \mathcal{X}, \Theta, \mathit{type}, \mathit{defs})$ entstehen (siehe Def. 19), haben einen besonderen Aufbau: Jede Grundtermersetzungsregel entsteht durch Instantiierung eines Elements r aus defs. Die Grundtermersetzungsregeln, die zu r gehören, unterscheiden sich lediglich in den Termen, die für die Variablen des funktionalen Programms eingesetzt wurden. Der neu entwickelte Ansatz verfolgt die Idee, nicht das komplette Grundtermersetzungssystem aus Def. 19 zu berechnen. Stattdessen wird für jedes $r \in \mathit{defs}$ eine *symbolische Regel* erstellt, die alle zu r gehörenden Grundtermersetzungsregeln kompakt darstellen kann. Hierzu werden Baumautomaten verwendet.

Definition 21. Sei R ein Grundtermersetzungssystem über der Signatur Σ. Sei weiterhin $\mathcal{X}$ eine Menge von Variablen. Eine **symbolische** Regel von R ist ein Tupel $r = (t_l, t_r) \in T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$ mit einer Abbildung

$$dom_r : Var(t_l) \to \mathcal{P}(T(\Sigma)),$$

wobei gilt:

- $Var(t_r) \subseteq Var(t_l)$,
- $\forall x \in Var(t_l) : |dom_r(x)| < \infty$,
- t_l und t_r sind *lineare* Terme: Jede Variable kommt maximal einmal in t_l und maximal einmal in t_r vor.

Für eine symbolische Regel $r = (t_l, t_r)$ bezeichnen wir die Menge der Grundtermersetzungsregeln $t \to t' \in T(\Sigma) \times T(\Sigma)$, für die

$$\exists \sigma \in Sub(T(\Sigma, \mathcal{X})) : (\forall x \in \mathcal{X} : \sigma(x) \in dom_r(x)) \wedge \sigma(t_l) = t \wedge \sigma(t_r) = t'$$

gilt, als seine **entfalteten** Regeln. Wir bezeichnen sie auch mit $\mathit{Expand}(r)$. Eine Substitution $\sigma \in Sub(T(\Sigma, \mathcal{X}))$, für die $\forall x \in \mathcal{X} : \sigma(x) \in dom_r(x)$ gilt, heißt **gültig für** r. □

Im weiteren Verlauf werden wir die Schreibweisen $r = (t_l, t_r)$ und $r = t_l \to t_r$ für symbolische Regeln synonym gebrauchen. Wir wollen einige Beispiele

bezüglich der Linearität anführen: Die Regel $f(x) \to g(x,x)$ stellt keine wohldefinierte symbolische Regel dar, da die Variable x zweimal auf der rechten Regelseite vorkommt; hingegen ist beispielsweise die Regel $f(x,y) \to g(x)$ zulässig.

Definition 22. Sei R eine endliche Menge von symbolischen Regeln über der Variablenmenge $\mathcal{X}$ und der Signatur Σ, bei der für zwei symbolische Regeln $r_1 = (t_1, t_1') \in R$ und $r_2 = (t_2, t_2') \in R$ gilt:

$$\forall x \in \mathit{Var}(t_1) \cap \mathit{Var}(t_2) : \mathit{dom}_{r_1}(x) = \mathit{dom}_{r_2}(x).$$

Dann bezeichnen wir R als **symbolisches Grundtermersetzungssystem** und verwenden dafür eine gemeinsame partielle Wertebereichsabbildung

$$\mathit{dom}_R : \mathcal{X} \to \mathcal{P}(T(\Sigma)),$$

wobei für $x \in \mathcal{X}$ gilt:

$$\mathit{dom}_R(x) := \begin{cases} \mathit{dom}_r(x), & \text{wenn es ein } r = (t_l, t_r) \in R \text{ gibt mit } x \in \mathit{Var}(t_l) \\ \bot, & \text{sonst} \end{cases}$$

Durch die obige Einschränkung ist diese Abbildung wohldefiniert. □

Wir wollen unter der Relation $\hookrightarrow_R$ eines symbolischen Grundtermersetzungssystems die Ableitungsrelation verstehen, die durch die Gesamtheit der entfalteten Regeln von R induziert wird.

Definition 23. Um die entfalteten linken oder rechten Termseiten von symbolischen Regeln zu bezeichnen, führen wir für einen Term $t \in T(\Sigma, \mathcal{X})$ und ein symbolisches Grundtermersetzungssystem R über Σ mit der Variablenmenge $\mathcal{X}$ die Menge $\mathit{Expand}_R(t)$ ein. Sie ist definiert als die Menge aller Terme $t' \in T(\Sigma)$, für die

$$\exists \sigma \in \mathit{Sub}(T(\Sigma, \mathcal{X})) : (\forall x \in \mathcal{X} : \sigma(x) \in \mathit{dom}_R(x)) \wedge \sigma(t) = t'$$

gilt. Wir nennen $\mathit{Expand}_R(t)$ die **Entfaltung** von t bezüglich dem symbolischen Grundtermersetzungssystem R. □

Meist verzichten wir auf die Angabe des R-Index bei der Expand-Menge oder der Ableitungsrelation $\hookrightarrow$, wenn der Kontext klar ist.
Die Abbildung dom_r bzw. dom_R gibt den Wertebereich der Variablen an. Da dieser für jede Variable endlich ist, kann zu seiner Darstellung ein Baumautomat verwendet werden. In vielen praktischen Fällen ist eine solche Repräsentation kompakter als eine explizite Speicherung jedes einzelnen Terms des Wertebereichs. Auf dieser Beobachtung basiert der im nächsten Abschnitt eingeführte symbolische Algorithmus.

Beispiel 11. Es ist möglich, Listenterme von maximaler Länge $n \geq 0$ mit einem Baumautomat zu repräsentieren, der lediglich linear viele Zustände und Regeln in n hat. Beschränken wir uns auf Listen mit Elementen aus $\{u, v, w\}$, so ist ein möglicher Automat durch $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ mit einer Signatur $\Sigma = \{u(), v(), w(), c(\cdot,\cdot), []()\}$, $Q = \{q_0, q_1, ..., q_n, q_c, q_f\}$, $Q_f = \{q_f\}$ und der Regelmenge δ:

- $u \to q_c$
- $v \to q_c$
- $w \to q_c$
- $[] \to q_0$, $[] \to q_f$
- $c(q_c, q_0) \to q_1$, $c(q_c, q_1) \to q_2$, ..., $c(q_c, q_{n-1}) \to q_n$
- $c(q_c, q_0) \to q_f$, $c(q_c, q_1) \to q_f$, ..., $c(q_c, q_{n-1}) \to q_f$.

Allgemein lässt sich die Menge aller Terme einer Signatur Σ mit maximaler Tiefe n durch einen Automaten erkennen, dessen Zustandsanzahl in $\mathcal{O}(n)$ liegt. □

Wir wollen daher im weiteren Verlauf jeder Variable x, die im Zuge der Definition 21 auftaucht, einen Baumautomaten $\mathcal{A}_x$ zuweisen, für den gilt: $\mathcal{L}(\mathcal{A}_x) = dom(x)$. Es ist immer möglich, $\mathcal{A}_x$ mit nur einem akzeptierenden Zustand und ohne ϵ-Regeln zu konstruieren, was wir im weiteren Verlauf auch annehmen werden.

Beispiel 12. Wir betrachten das Beispiel 8 und modellieren die symbolische Regel

$$append(c(x, xs), ys) \to c(x, append(xs, ys)).$$

Der Wertebereich der Variablen x sei $dom(x) = \{0, 1\}$. Für die Variablen xs und ys wählen wir Listen der Länge maximal zwei, also $dom(xs) = dom(ys) = \{\,[]\,\} \cup \{c(a_1, c(a_2, ..., c(a_n, []))...) \mid a_1, ..., a_n \in Consts, 0 \leq n \leq 2\}$. Die gesamte symbolische Regel wird dargestellt durch

- ein Termpaar $(append(c(x, xs), ys), c(x, append(xs, ys)))$,
- drei Baumautomaten $\mathcal{A}_x, \mathcal{A}_{xs} = \mathcal{A}_{ys}$, die den Wertebereich der Variablen darstellen.

Eine mögliche Konstruktion des Automaten $\mathcal{A}_{xs}$ wird in Beispiel 11 erläutert.□

Mithilfe der Variablenautomaten ist es nicht schwierig, auf kanonische Weise für einen linearen Term t mit Variablen einen Baumautomaten zu konstruieren, der genau die Menge seiner entfalteten Terme erkennt, also $Expand(t)$. Hierbei wird ausgenutzt, dass eine Variable nur einmal in einem Term vorkommen kann. Diese Beobachtung wird uns später nützlich sein. Algorithmus GENERATETERMAUTOMATON in Abb. 4.3 verdeutlicht das Vorgehen. Auch diese Prozedur liefert Automaten zurück, die nur einen akzeptierenden Zustand enthalten.

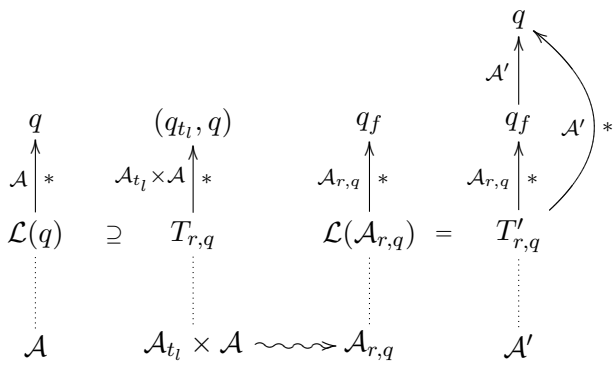


Abbildung 4.2: Schematische Darstellung zur Simulation rechter Regelseiten. Die Grafik zeigt die Termmengen auf, die von den Zuständen q, (q_{t_l}, q) und q_f in den beteiligten Automaten abgeleitet werden können. Mithilfe des Automaten $\mathcal{A}_{t_l} \times \mathcal{A}$ wird der Automat $\mathcal{A}_{r,q}$ erzeugt. Aus Zuständen und Regeln von $\mathcal{A}$ und $\mathcal{A}_{r,q}$ entsteht $\mathcal{A}'$, in dem q u.a. die Menge $T'_{r,q}$ erkennen kann. Zu beachten ist, dass der Zustand q sowohl in $\mathcal{A}$ als auch in $\mathcal{A}'$ vorkommt. Für weitere Erläuterungen siehe Text.

4.4 Ein symbolischer Algorithmus

Für die im vorigen Abschnitt entwickelten symbolischen Grundtermersetzungssysteme entwickeln wir nun einen Algorithmus, der das Erreichbarkeitsproblem löst.
Wir betrachten eine symbolische Regel $t_l \to t_r$ über einer Signatur Σ mit der Wertebereichsabbildung dom. Für den Term t_l lässt sich wie oben dargestellt ein Baumautomat $\mathcal{A}_{t_l}$ erstellen, der genau $Expand(t_l)$ erkennt. Es ist möglich, $\mathcal{A}_{t_l}$ mit nur einem akzeptierenden Zustand q_{t_l} zu erstellen. Sei $\mathcal{A}$ ein Baumautomat über Σ, q ein Zustand von $\mathcal{A}$. Wir betrachten den Produktautomat $\mathcal{A}_{t_l} \times \mathcal{A}$ und hierin den Zustand (q, q_{t_l}). Ein Term t kann nach (q, q_{t_l}) reduziert werden, wenn er in $\mathcal{A}_{t_l}$ nach q_{t_l} und in $\mathcal{A}$ nach q reduziert werden kann. Die Elemente aus $Expand(t_l)$, die zu Zustand q in $\mathcal{A}$ reduziert werden können, sind genau die Terme, die (q, q_{t_l}) in $\mathcal{A}_{t_l} \times \mathcal{A}$ erkennt. Wir bezeichnen diese Termmenge mit $T_{r,q} = Expand(t_l) \cap \mathcal{L}(q)$. Die diesen entsprechenden rechten Seiten der symbolischen Regel bezeichnen wir mit

$$T'_{r,q} = \{t \mid \exists t' \in T_{r,q} : (t', t) \in Expand(r)\}.$$

Es ist möglich, einen Baumautomaten $\mathcal{A}_{r,q}$ zu konstruieren, der genau die Termmenge $T'_{r,q}$ erkennt. Wir nehmen an, dass $\mathcal{A}_{r,q}$ nur einen akzeptierenden Zustand q_f enthält. Dazu werden die Zustände und Regeln von $\mathcal{A}_{t_l} \times \mathcal{A}$ verwendet. Der Automat $\mathcal{A}_{r,q}$ kann mit $\mathcal{A}$ zu einem neuen Automaten kombiniert werden: Die Zustände aus $\mathcal{A}_{r,q}$ werden als nichtakzeptierende Zustände $\mathcal{A}$ hinzugefügt, ebenso die dazugehörigen Regeln. Den so entstandenen Automaten, der Regeln und Zustände aus $\mathcal{A}$ und $\mathcal{A}_{r,q}$ besitzt, bezeichnen wir mit $\mathcal{A}'$. Wir fügen noch eine ϵ-Regel $q_f \to q$ in $\mathcal{A}'$ ein. Sie sorgt dafür, dass q in $\mathcal{A}$

nun auch $T'_{r,q}$ erkennt. Für eine Zusammenfassung der Vorgänge siehe auch Abb. 4.2.
Bei diesem Ansatz wurde die kompakte Darstellung der Regeln als Baumautomat ausgenutzt und kein explizites Entfalten des Grundtermersetzungssystems war nötig. Später werden wir manchmal davon sprechen, einem Automaten $\mathcal{A}$ Regeln und Zustände eines anderen Automaten $\tilde{\mathcal{A}}$ hinzuzufügen oder einzubauen; damit ist der obige Vorgang gemeint, bei dem streng genommen ein neuer Automat $\mathcal{A}'$ entsteht, der Regeln und Zustände aus beiden Automaten enthält. Diesen setzen wir dann im weiteren Verlauf mit $\mathcal{A}$ gleich; deshalb muss zwischen dem Zeitpunkt vor dem Einfügen von $\tilde{\mathcal{A}}$ und danach unterschieden werden, wenn von $\mathcal{A}$ die Rede ist.
Der hier untersuchte symbolische Algorithmus benutzt die beschriebene Vorgehensweise, um aus einem Baumautomaten $\mathcal{A}_0$ einen Automaten $\mathcal{A}_{post}$ mit $\mathcal{L}(\mathcal{A}_{post}) = Post^*(\mathcal{L}(\mathcal{A}_0))$ bezüglich einer Menge von symbolischen Regeln zu bilden: Für jede symbolische Regel $r = t_l \to t_r$ und jeden Zustand q wird die Menge $T'_{r,q}$ errechnet, der Automat $\mathcal{A}_{r,q}$ erstellt und auf die oben beschriebene Weise in $\mathcal{A}_0$ eingebaut. Beim Einbauen der Regeln und Zustände von $\mathcal{A}_{r,q}$ kann folgendes geschehen:

- Der Zustand q kann *zusätzliche* entfaltete linke Seiten symbolischer Regeln erkennen.
- Andere, bereits vor dem Einfügevorgang in $\mathcal{A}_0$ enthaltene Zustände q' können *zusätzliche* entfaltete linke Seiten symbolischer Regeln erkennen.
- Auch die *hinzugefügten* Zustände aus $\mathcal{A}_{r,q}$ können entfaltete linke Seiten symbolischer Regeln erkennen.

Wir bemerken noch, dass durch dieses Vorgehen weder Zustände noch Regeln aus $\mathcal{A}_0$ entfernt oder verändert werden: Die Menge der akzeptierenden Terme kann nur zunehmen.
Die Idee liegt nahe, das obige Verfahren iterativ zu verwenden: Wir erstellen im $i+1$-ten Iterationsschritt einen Automaten $\mathcal{A}_{i+1}$, der zunächst alle Regeln und Zustände von $\mathcal{A}_i$ enthält. Für jede symbolische Regel r und jeden Zustand q aus $\mathcal{A}_i$ wird das obige Verfahren durchgeführt, wobei die neu erzeugten Regeln und Zustände dem Automaten $\mathcal{A}_{i+1}$ hinzugefügt werden.
Problem beim Iterationsansatz ist, festzustellen, welche entfalteten rechte Seiten *neu erkannt* werden, d.h. erst in $\mathcal{A}_i$ und nicht bereits in $\mathcal{A}_{i-1}$ abgeleitet werden können. Außerdem kann das Einfügen neuer Zustände ohne weitere Prüfungen zu einem nichtterminierenden Verfahren führen, wie folgendes Beispiel zeigt:

Beispiel 13. Wir betrachten die einzelne symbolische Regel $r = f(x) \to f(f(x))$ über $\Sigma = \{f(\cdot,\cdot), a()\}$ mit $dom_r(x) = \{a\}$. Der Automat für $dom_r(x)$ sei

$$\mathcal{A}_x = (\{q_x\}, \Sigma, \{q_x\}, \{a \to q_x\}).$$

Ein Automat für alle linken Seiten der Regel (hier nur eine) ist:

$$\mathcal{A}_L = (\{q_x, q_L\}, \Sigma, \{q_L\}, \{a \to q_x, f(q_x) \to q_L\}).$$

Als Startautomat wählen wir

$$\mathcal{A}_0 = (\{q_a, q_f\}, \Sigma, \{q_f\}, \delta_0)$$

mit den Regeln $a \to q_a$, $f(q_a) \to q_f$ in δ_0. Wir bilden nun

$$\mathcal{A}_L \times \mathcal{A}_0 = (\{(q_L, q_f), (q_x, q_a)\}, \Sigma, \{(q_L, q_f)\}, \delta_\times), \text{ mit}$$

$$\delta_\times = \{a \to (q_x, q_a), f((q_x, q_a)) \to (q_L, q_f)\},$$

wobei wir nur die produktiven Zustände berücksichtigt haben. Wir erkennen, dass nur der Zustand q_f eine linke Seite der symbolischen Regel ableiten kann, da für q_a der Zustand (q_L, q_a) in $\mathcal{A}_L \times \mathcal{A}_0$ nicht produktiv ist. Daher müssen nur für den Zustand q_f zusätzliche rechte Seiten der Regel ableitbar sein. Wir erstellen aus $\mathcal{A}_L \times \mathcal{A}_0$ einen Automaten für die rechten Seiten von q_f. Hierbei benennen wir den Zustand (q_x, q_a) in q' um und kopieren die Automatenregeln, die eventuell in einer Reduktion zu (q_x, q_a) vorkommen:

$$\mathcal{A}_{r,q_f} = (\{q_1, q_2, q'\}, \Sigma, \{q_2\}, \{a \to q', f(q') \to q_1, f(q_1) \to q_2\})$$

Nun kombinieren wir $\mathcal{A}_{r,qf}$ und $\mathcal{A}_0$:

$$\mathcal{A}_1 = (\{q_a, q_f, q_1, q_2, q'\}, \Sigma, \{q_f\}, \delta_q), \text{ mit}$$

$$\delta_1 = \{a \to q_a, f(q_a) \to q_f, a \to q', f(q') \to q_1, f(q_1) \to q_2, q_2 \to q_f\}$$

$\mathcal{A}_1$ erkennt nun zusätzlich zu $\mathcal{L}(\mathcal{A}_0)$ den Term $f(f(a))$, also

$$\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_0) \cup Post(\mathcal{L}(\mathcal{A}_0)).$$

Wir bemerken, dass der Zustand q_1 in $\mathcal{A}_1$ genau den Term $f(a)$ erkennt, wie in $\mathcal{A}_0$ der Zustand q_f. Würden wir also den Vorgang des Schnitts wiederholen, würden wir auch für den Zustand q_1 in $\mathcal{A}_1$ wieder u.a. einen Zustand einfügen, der exakt $f(a)$ erkennt, um für die von ihm erkannte linke Seite der symbolischen Regel auch die rechte Seite zu erkennen. Der Automat würde also mit jedem Iterationsschritt weiter wachsen, und das Verfahren nie terminieren. □

Um dem abzuhelfen führen wir zunächst eine Erweiterung unserer bisherigen Notation ein: $\mathcal{A}_{r,q,i}$ soll den Automaten $\mathcal{A}_{r,q}$ bezeichnen, dessen Zustände und Regeln im Automaten $\mathcal{A}_i$ eingefügt wurden, also während des i-ten Iterationsschritts. Wir verwalten für jede symbolische Regel r eine Liste $Copies_r$. In dieser speichern wir die bereits eingefügten Automaten $\mathcal{A}_{r,q,i}$ und deren einstige akzeptierende Zustände q' ab. Die Liste besteht also aus Tupeln von Automaten und Zuständen der Art $(\mathcal{A}_{r,q,i}, q')$. Bevor Regeln und Zustände von $\mathcal{A}_{r,q,i}$ eingefügt werden, prüfen wir zunächst, ob ein zu ihm äquivalenter Automat nicht bereits früher erzeugt wurde: Das bedeutet, dass ein Tupel $(\mathcal{A}_{r,\tilde{q},j}, q') \in Copies_r$ existiert mit $j \leq i$ und $\mathcal{L}(\mathcal{A}_{r,q,i}) = \mathcal{L}(\mathcal{A}_{r,\tilde{q},j})$. Ist dem so, braucht nicht der gesamte Automat neu eingefügt zu werden: Lediglich die ϵ-Regel $q' \to q$ ist notwendig. Ist der Automat noch nicht erzeugt worden, so fügen wir ihn in $Copies_r$ ein.

Beispiel 14. Wir betrachten erneut das Beispiel 13 und verwalten nun eine Liste $Copies$ für die einzige symbolische Regel. Wir fügen also in diese Liste nach dem ersten Iterationsschritt das Tupel $(\mathcal{A}_{r,q_f,1}, q_2)$ mit dem Automaten

$$\mathcal{A}_{r,q_f,1} = (\{q_1, q_2, q'\}, \Sigma, \{q_2\}, \{a \to q', f(q') \to q_1, f(q_1) \to q_2\})$$

ein: Sollte später der Automat $\mathcal{A}_{r,q_f,1}$ wiederverwendet werden, so muss lediglich eine ϵ-Regel von q_2 zu dem jeweiligen Zustand hinzugefügt werden. Im nächsten Iterationsschritt sollten für q_1 eigentlich neue Zustände und Regeln eingefügt werden, um die (einzige) rechte Seite der symbolischen Regel zu erkennen. Jedoch ist mit $\mathcal{A}_{r,q_f,1}$ bereits ein Automat eingefügt worden, der exakt dieselbe Menge an Termen $\{f(f(a))\}$ erkennt. Daher wird das Ableiten der rechten Seiten durch eine ϵ-Regel $q_2 \to q_1$ simuliert. Da sonst keine neuen linken Seiten der symbolischen Regel erkannt werden, ergibt sich:

$$\mathcal{A}_2 = (\{q_a, q_f, q_1, q_2, q'\}, \Sigma, \{q_f\}, \delta_q), \text{ mit}$$

$$\delta_1 = \{a \to q_a, f(q_a) \to q_f, a \to q', f(q') \to q_1, f(q_1) \to q_2, q_2 \to q_f, q_2 \to q_1\}$$

Dieser Automat erkennt bereits genau $Post^*(\mathcal{L}(\mathcal{A}_0)) = \{f^i(a) \mid i \geq 1\}$. □

In Abb. 4.4 ist der gesamte symbolische Algorithmus dargestellt. Er benutzt die Funktionen aus Abb. 4.3. Für einen Baumautomaten $\mathcal{A}$ kann mittels der Funktionsaufrufe $\mathrm{states}(\mathcal{A})$, $\mathrm{accstates}(\mathcal{A})$ und $\mathrm{rules}(\mathcal{A})$ auf die Menge aller Zustände, aller akzeptierender Zustände bzw. aller Regeln von $\mathcal{A}$ zugegriffen werden. Es wird eine Signatur Σ vorausgesetzt, die allen vorkommenden Baumautomaten gemein ist.
Da wir in den nächsten Abschnitten aus Effizienzgründen eine etwas modifizierte Version des Algorithmus entwickeln werden, soll die in 4.4 vorkommende Prozedur RIGHTSIDES nicht im Detail angegeben werden. Sie liefert für einen Zustand (q, q') eines Produktautomaten, der eine bestimmte Menge linker Seiten einer symbolischen Regel erkennt, einen Automat, der genau die entsprechenden rechten Seiten der Regel erkennt. Eine Implementierung muss aus dem Produktautomaten die Zustände bestimmen, die bei der Ableitung der linken Seiten in den Zustand (q, q') die Variablen-Teile der linken Seiten erkennen. Diese muss beim Erstellen eines Automaten für die rechten Seiten an passender Stelle eingesetzt werden. Dabei hat man bei mehreren Variablen darauf zu achten, welche Kombination von Variablenzuständen möglich ist, siehe dazu Kapitel 5.4.1.

4.4.1 Terminierung und Korrektheit

Wir wollen zunächst für den Algorithmus aus Abb. 4.4 die Terminierung beweisen:

Satz 6. Der symbolische Algorithmus terminiert für alle Baumautomaten $\mathcal{A}_0$ und symbolische Grundtermersetzungssysteme R. □

TESTALREADYIN(Liste $Copies_r$, Automat $\mathcal{A}$)

```
1  for entry = (A_entry, q) ∈ Copies_r
2      do
3          if L(A_entry) = L(A)
4             then return q
5  return ⊥
```

GENERATETERMAUTOMATON(Term t)

```
1  if t ist Variable x
2     then
3          A := A_x
4  elseif t ist Konstante
5     then
6          q_t := newstate()
7          A := ({q_t}, Σ, {q_t}, {t → q_t})
8  elseif t = f(t_1, ..., t_n)
9     then
10         for t_i ∈ children(t)
11             do A_i := GENERATETERMAUTOMATON(t)
12         q_t := newstate()
13         newrules := {f(q_1, ..., q_n) → q_t | ∀i ≤ n : q_i ∈ accstates(A_i)}
14         A := ({q_t} ∪ ⋃_{i=1}^n states(A_i), Σ, {q_t}, newrules ∪ ⋃_{i=1}^n rules(A_i))
15
16 return A
```

Abbildung 4.3: Die Funktionen TESTALREADYIN und GENERATETERMAUTOMATON. Es wird vorausgesetzt, dass durch $\mathcal{A}_x$ für jede Variable x ein Baumautomat verfügbar ist, der nur einen einzigen akzeptierenden Zustand hat. Dann gilt, dass jeder von der Funktion GENERATETERMAUTOMATON zurückgegebene Term nur einen akzeptierenden Zustand besitzt. Die Funktion $\mathrm{newstate}()$ liefert einen neuen Zustand zurück, der noch in keinem anderen Automaten vorkommt. Für weitere Erklärungen siehe Text.

```
CALCPOST(Automat A_0 = (Q_0, Σ, Q_f, δ_0), Symbolisches Grundtermersetzungssystem R)
 1  for r = (t_l, t_r) ∈ R
 2      do
 3          ▷ Initialisiere Automatenlisten
 4          Copies_r := ∅
 5          ▷ Erstelle Automat für linke Seite der Regel
 6          A_{t_l} := GENERATETERMAUTOMATON(t_l)
 7          Weise q_r den akzeptierenden Zustand von A_r zu
 8  change := true
 9  i := 0
10  while change
11      do
12          change := false
13          newstates := ∅
14          newrules := ∅
15          for r = (t_l, t_r) ∈ R
16              do
17                  A_× := A_{t_l} × A_i
18                  for q ∈ A_i
19                      do
20                          if L((q, q_r)) ≠ ∅
21                            then
22                                  A_{r,q,i} := RIGHTSIDES((q, q_r), r, A_×)
23                                  Weise q_f den akzeptierenden Zustand von A_{r,q,i} zu
24                                  state := TESTALREADYIN(Copies_r, A_{r,q,i})
25                                  if state = ⊥
26                                    then
27                                          newstates := newstates ∪ states(A_{r,q,i})
28                                          newrules := newrules
29                                            ∪ rules(A_{r,q,i}) ∪ {(q_f → q)}
30                                          Copies_r := Copies_r ∪ (A_{r,q,i}, q_f)
31                                    else
32                                          newrules := newrules ∪ {state → q}
33                                  change := change ∨ (newrules ⊈ Q_i)
34          Q_{i+1} := Q_i ∪ newstates
35          δ_{i+1} := δ_i ∪ newrules
36          A_{i+1} := (Q_{i+1}, Σ, Q_f, δ_{i+1})
37          i := i + 1
38  return A_i
```

Abbildung 4.4: Ein symbolischer Algorithmus für $Post^*$, Erläuterungen im Text.

Beweis. Im Laufe des Algorithmus in Abb. 4.4 können dem Automaten $\mathcal{A}_i$ in Zeile 34 und 35 neue Zustände und Regeln aus den Mengen $newstates$ und $newrules$ hinzugefügt werden. Werden keine neuen Regeln hinzugefügt, gilt also $newrules \subseteq Q_i$, so behält $change$ den Wert $false$, und der Algorithmus terminiert. Für jede symbolische Regel $r = (t_l, t_r)$ mit vorkommenden Variablen $x_1, ..., x_n$ können nur endlich oft unterschiedliche Automaten für Teilmengen von $Expand(t_r)$ erstellt werden: Erkennt ein Zustand q im Automat $\mathcal{A}_i$ eine Teilmenge von $Expand(t_l)$, für die bereits dieselbe Menge rechter Seiten eingefügt wurde, so werden lediglich ϵ-Regeln hinzugefügt. Die Menge der für r erstellten Automaten kann mit

$$2^{|Expand(t_r)|} = 2^{(\prod_{i=1}^{n} |dom_r(x_i)|)}$$

abgeschätzt werden. Hier fließt ein, dass für alle x_i $|dom_r(x_i)| < \infty$ gilt. Damit werden nur endlich viele Zustände pro symbolischer Regel in den Automaten eingefügt. Daher können auch nur endlich viele ϵ-Regeln hinzugefügt werden, und es gibt einen Schleifendurchlauf mit $i \geq 0$, bei dem $newrules \subseteq Q_i$ gilt. □

Wir führen für die weiteren Sätze folgende Schreibweise ein:
Wir bezeichnen mit $\mathcal{A}_0 = (Q_0, \Sigma, Q_f, \delta_0)$ und $\mathcal{A}_k = (Q_k, \Sigma, Q_f, \delta_k)$ für alle $k \geq 1$ die entsprechenden Automaten aus dem Algorithmus in Abb. 4.3.
$\mathcal{L}_i(q)$ soll die Menge der von Zustand q im Automaten $\mathcal{A}_i$ erkannten Terme bezeichnen. Es soll $\mathcal{L}_i(q) = \emptyset$ für alle i gelten, für die q noch nicht in der Zustandsmenge von $\mathcal{A}_i$ enthalten ist. Es gilt für alle Zustände q und alle $n \geq 0$

$$\mathcal{L}_0(q) \subseteq \mathcal{L}_1(q) \subseteq ... \subseteq \mathcal{L}_n(q),$$

denn es werden im Verlaufe des Algorithmus keinerlei Zustände oder Regeln aus den Automaten $\mathcal{A}_i$ entfernt oder verändert.
Wir wollen mit $first(q)$ den Index $i \geq 0$ bezeichnen, der minimal ist mit der Eigenschaft, dass $\mathcal{A}_i$ den Zustand q bereits enthält.
Durch Satz 6 ist sichergestellt, dass nur endlich viele Automaten $\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n$ mit einem $n \geq 0$ erstellt werden, wobei der Algorithmus dem letzten Automaten $\mathcal{A}_n$ keine Regeln oder Zustände hinzufügt und somit terminiert. Wir erweitern diese Folge von Automaten, indem wir $\mathcal{A}_j = \mathcal{A}_n$ setzen für $j > n$. O.B.d.A. können wir annehmen, dass die Automaten $\mathcal{A}_{r,q,i}$, die während der Ausführung des Algorithmus erzeugt werden, ϵ-Regel-frei sind. Dies vereinfacht die Darstellung der folgenden Beweise. Zur Erinnerung: Die Regeln und Zustände eines Automaten $\mathcal{A}_{r,q,i}$ kommen zum Automaten $\mathcal{A}_i$ neu hinzu, damit Zustand q die entfalteten rechten Seiten der symbolischen Regel r erkennen kann, von denen er bereits die entsprechenden linken Seiten erkennt. Dies geschieht in Zeile 34 und 35 des Algorithmus in Abb. 4.4.

Satz 7. Für jeden Automaten $\mathcal{A}_i$ im Algorithmus aus Abb. 4.4 gilt:

$$\mathcal{L}(\mathcal{A}_i) \supseteq \bigcup_{k=0}^{i} Post^k(\mathcal{L}(\mathcal{A}_0)).$$

□

Beweis. Wir zeigen die Behauptung per Induktion nach i. Für $i = 0$ lautet die Aussage: $\mathcal{L}(\mathcal{A}_0) \supseteq \mathcal{L}(\mathcal{A}_0)$ und ist daher wahr.
Für $i > 0$ bemerken wir zunächst, dass wir im Laufe des Algorithmus keinerlei Zustände oder Regeln aus den Automaten $\mathcal{A}_i$ entfernen oder die Menge der akzeptierenden Zustände verändern. Lediglich neue Zustände und Regeln können dazukommen, daher gilt für alle $i \geq 1$, dass $\mathcal{L}(\mathcal{A}_i) \supseteq \mathcal{L}(\mathcal{A}_{i-1})$, womit durch die Induktionshypothese folgt:

$$\mathcal{L}(\mathcal{A}_i) \supseteq \bigcup_{k=0}^{i-1} Post^k(\mathcal{L}(\mathcal{A}_0)).$$

Wir betrachten daher einen Term $t \in Post^i(\mathcal{L}(\mathcal{A}_0))$. Für diesen Term gibt es eine Ableitungsfolge $t_0 \hookrightarrow_R t_1 \hookrightarrow_R \ldots \hookrightarrow_R t_{i-1} \hookrightarrow_R t$ im symbolischen Grundtermersetzungssystem R, mit $t_j \in Post^j(\mathcal{L}(\mathcal{A}_0))$ für alle $j \geq 0$. Wir können durch die Induktionshypothese annehmen, dass $t_{i-1} \in \mathcal{L}(\mathcal{A}_i)$ gilt. Wegen $t_{i-1} \hookrightarrow_R t$ gibt es einen Teilterm s und eine Position p mit $t_{i-1}[s]_p = t$. Ebenso existiert eine symbolische Regel $r = (t_l, t_r) \in R$ mit einer für r gültigen Substitution σ, so dass gilt: $t_{i-1}|_p = \sigma(t_l)$ und $s = \sigma(t_r)$. Wir fixieren eine Reduktion für $t_{i-1} \rightarrow^*_{\mathcal{A}_{i-1}} q_f$ mit einem akzeptierenden Zustand q_f. In ihr wurde insbesondere $t_{i-1}|_p$ auf einen Zustand q reduziert, also $t_{i-1}|_p \rightarrow^*_{\mathcal{A}_{i-1}} q$. Gilt $s \rightarrow^*_{\mathcal{A}_{i-1}} q$, so sind wir fertig. Andernfalls erkennt der Zustand (q, q_r) im Produktautomat $\mathcal{A}_{i-1} \times \mathcal{A}_r$ u.a. den Term $t_{i-1}|_p$. $t_{i-1}|_p$ stellt eine linke Seite der symbolischen Regel r dar, da $\sigma(t_l) = t_{i-1}|_p$ gilt. Daher wird entweder ein Automat $\mathcal{A}_{r,q',i}$ in $\mathcal{A}_i$ eingefügt, der s erkennt und damit auch q, oder eine ϵ-Regel eingefügt, so dass ebenso s durch q erkannt wird. Also gilt $s \rightarrow^*_{\mathcal{A}_i} q$ und $t = t_{i-1}[s]_p \rightarrow^*_{\mathcal{A}_i} q_f$. Damit gilt $t \in \mathcal{L}(\mathcal{A}_i)$ und somit die Behauptung. □

Satz 8. Gilt für einen Term t, dass $t \rightarrow^*_{\mathcal{A}_i} q$, und sei $\mathcal{A} = \mathcal{A}_{first(q)}$, so gibt es einen Term t' mit $t' \hookrightarrow^*_R t$ und $t' \rightarrow^*_{\mathcal{A}} q$. □

Beweis. Wir beweisen die Aussage per Induktion über i. Für $i = 0$ ist die Behauptung trivial: Es kann $t = t'$ gewählt werden.
Sei nun $i > 0$.
Wir betrachten eine kürzeste Reduktion für $t \rightarrow^*_{\mathcal{A}_i} q$. Wir führen den Beweis durch eine weitere Induktion über die Anzahl der Reduktionsschritte j, die Regeln aus $\delta_i \setminus \delta_{i-1}$ verwenden. Ist $j = 0$, so gilt $t \rightarrow^*_{\mathcal{A}_{i-1}} q$ und damit die Behauptung. Ist $j > 0$, so betrachten wir das erste Anwenden einer Regel $r = t_l \rightarrow_R t_r$ aus $\delta_i \setminus \delta_{i-1}$ in der fixierten Reduktion. Es gibt zwei Fälle:

- r ist keine ϵ-Regel: Dann gehört r zu einem in $\mathcal{A}_i$ eingebundenen Automaten $\mathcal{A}_{r,\tilde{q},k}$. Es existiert daher eine Position $p \in Pos(t)$ mit $t|_p \rightarrow^*_{\mathcal{A}_i} \tilde{q}$. Da $r \in \delta_i \setminus \delta_{i-1}$, gilt außerdem, dass es einen Term $\tilde{t}$ gibt mit $\tilde{t} \rightarrow^*_{\mathcal{A}_{i-1}} \tilde{q}$ und $t|_p \hookrightarrow^*_R \tilde{t}$. Dann gilt $t[\tilde{t}]_p \rightarrow^*_{\mathcal{A}_i} q$, wobei hierfür eine Reduktion existiert, die maximal $j-1$ Regeln aus $\delta_i \setminus \delta_{i-1}$ verwendet. Bezeichnet $\mathcal{A}$ den Automaten $\mathcal{A}_{first(q)}$, so gibt es nach der Induktionshypothese einen Term t' mit $t' \hookrightarrow^*_R t[\tilde{t}]_p$, für den $t' \rightarrow^*_{\mathcal{A}} q$ gilt. Da $t[\tilde{t}]_p \hookrightarrow^*_R t$ gilt, ist dieser Fall bewiesen.

- $r = q' \to q''$ ist ϵ-Regel: Es gibt daher eine Position p, so dass sich die Reduktion folgendermaßen darstellen lässt:

  $$t \to_{\mathcal{A}_{i-1}} \dots \to_{\mathcal{A}_{i-1}} t[q']_p \to_{\mathcal{A}_i} t[q'']_p \to_{\mathcal{A}_i} \dots \to_{\mathcal{A}_i} q$$

  Es gilt $t|_p \to^*_{\mathcal{A}_{i-1}} q'$. Gemäß der Induktionshypothese folgt daraus, dass es einen Term $\tilde{t}$ gibt mit $\tilde{t} \hookrightarrow^*_R t|_p$ und $\tilde{t} \to^*_{\mathcal{A}_{i-1}} q'$, der in $\mathcal{L}_{first(q')}(q')$ enthalten ist. $\tilde{t}$ stellt daher eine entfaltete rechte Seite einer symbolischen Regel dar. Da r in den Automaten $\mathcal{A}_i$ eingefügt wurde, existiert ebenso ein Term $\hat{t}$ mit $\hat{t} \hookrightarrow^*_R \tilde{t}$ und $\hat{t} \to^*_{\mathcal{A}_{i-1}} q''$. Wir wählen $t'' = t[\hat{t}]_p$. Es gibt eine Reduktion

  $$t'' \to_{\mathcal{A}_{i-1}} \dots \to_{\mathcal{A}_{i-1}} t[q'']_p \to_{\mathcal{A}_i} \dots \to_{\mathcal{A}_i} q,$$

  in der höchstens $j-1$ Regeln aus $\delta_i \setminus \delta_{i-1}$ vorkommen. Mit der Induktionshypothese folgt, dass es einen Term t' gibt mit $t' \to^*_{\mathcal{A}_{i-1}} q$ und

  $$t' \hookrightarrow^*_R t'' = t[\hat{t}]_p \hookrightarrow^*_R t[\tilde{t}]_p \hookrightarrow^*_R t,$$

  der in der Menge $\mathcal{L}_{first(q)}(q)$ enthalten ist. □

Damit können wir die Korrektheit des Verfahrens zeigen:

Korollar 9. Der Algorithmus aus Abb. 4.4 berechnet $Post^*(\mathcal{L}(\mathcal{A}_0))$. □

Beweis. Es gibt einen Index $k \geq 0$, so dass $\mathcal{L}(\mathcal{A}_k) = \mathcal{L}(\mathcal{A}_{k+1})$. Da

$$\mathcal{L}(\mathcal{A}_0) \subseteq \mathcal{L}(\mathcal{A}_1) \subseteq \dots$$

gilt, ist mit Satz 7 gezeigt, dass

$$\mathcal{L}(\mathcal{A}_k) \supseteq Post^*(\mathcal{L}(\mathcal{A}_0)).$$

Es werden $\mathcal{A}_0$ keine neuen akzeptierenden Zustände hinzugefügt. Aus Satz 8 folgt für jeden akzeptierenden Zustand q und jeden Term $t \in \mathcal{L}_k(q)$, dass es einen Term $t' \in \mathcal{L}_0(q)$ gibt mit $t' \hookrightarrow^*_R t$. Daraus ergibt sich

$$\mathcal{L}(\mathcal{A}_k) = \bigcup_{q \in Q_f} \mathcal{L}_k(q) \subseteq \bigcup_{q \in Q_f} Post^*(\mathcal{L}_0(q)) \subseteq Post^*(\mathcal{L}(\mathcal{A}_0)).$$

□

5 Implementation

Im Rahmen dieser Arbeit wurde ein einfacher Prototyp des neuen symbolischen Algorithmus in der Programmiersprache Java erstellt. Mit dessen Hilfe wurde Laufzeit und Speicherverbrauch analysiert und mit einer Implementierung des nichtsymbolischen Algorithmus aus [Löd06] verglichen. Das dabei entstandene Werkzeug erlaubt es, einfache funktionale Programme einzulesen, für eine spezifizierbare Menge von Anfangstermen Nachfolger- oder Vorgängermengen zu berechnen und das Ergebnis zu analysieren. Ebenfalls ist es möglich, für einen Term t aus der Nachfolgermenge anzugeben, wie ein Element aus der Menge der Anfangsterme zu t mittels Regeln des Termersetzungssystems abgeleitet werden konnte.

5.1 Aufbau

Wie schon in der Einleitung erwähnt wurde für den Prototypen eine Implementation in Java erstellt. Hierzu wurde die Möglichkeit der Unterteilung logisch trennbarer Programmteile in Packages ausgenutzt. Die Programmstruktur des Prototypen gliedert sich in folgende Bestandteile:

1. Eine Bibliothek für grundlegende Operationen auf Baumautomaten (Package `treeautomaton`),
2. eine Bibliothek zur Darstellung und Manipulation von symbolischen Grundtermersetzungssystemen (Package `rewriting`),
3. eine Bibliothek für die Implementierung verschiedener Reduktions- und Minimierungsalgorithmen auf Baumautomaten (Package `minimisation`),
4. ein Modul für die Generierung von Gegenbeispielen (Package `counterexamples`),
5. ein Modul für eine einfache Implementierung des Algorithmus aus [Löd02] (Package `nonsymbolic`),
6. ein Modul für den Ablauf des symbolischen Algorithmus (Package `symbolic`),
7. einen einfachen Parser für die Eingabedateien (Package `parser`) und
8. Dateien für die Programmlogik und das Erstellen von Grundtermersetzungssystemen aus Eingabedateien
   (Packages `control` und `inputprocessing`).

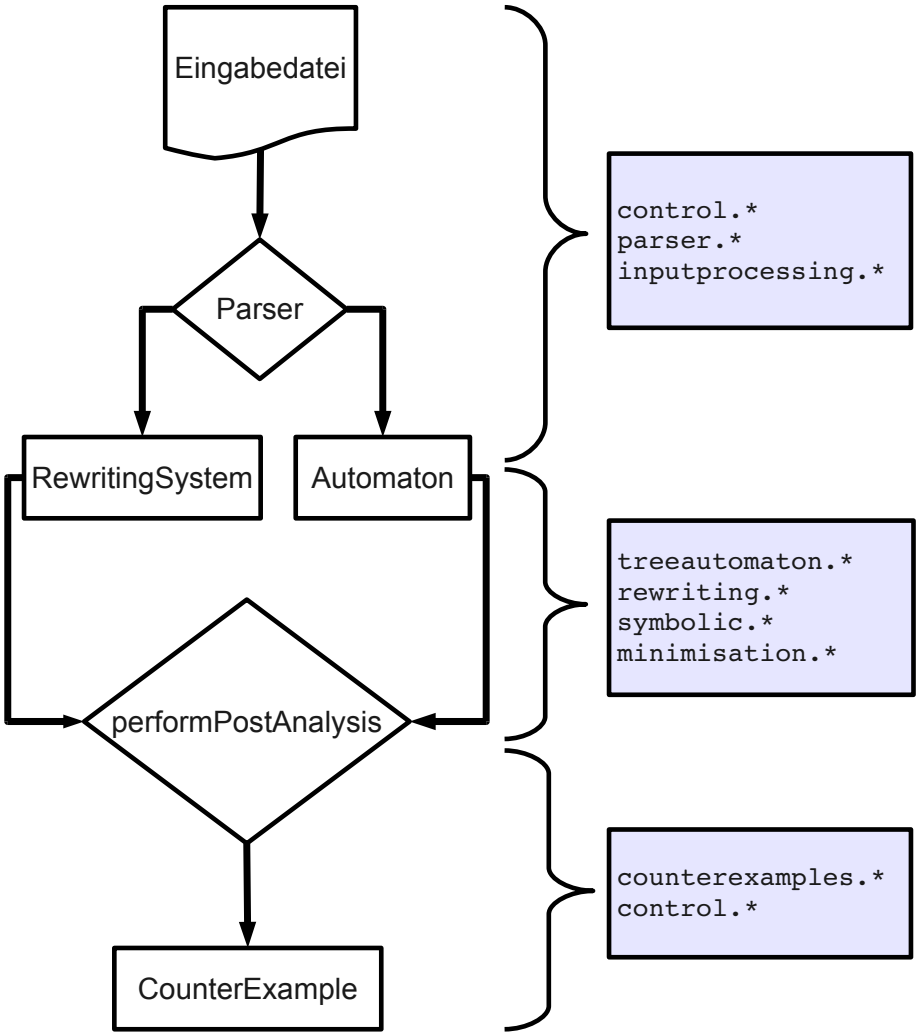


Abbildung 5.1: Schematischer Aufbau des Prototypen. Grob lässt sich der Ablauf einer Ausführung in die Phasen Parsen, Ausführen des symbolischen Algorithmus und eventuelle Berechnung von Gegenbeispielen gliedern. Am Rand sind die Packages angedeutet, die im jeweiligen Arbeitsschritt des Prototypen zum Einsatz kommen.

5.2 Darstellung von Baumautomaten

Das Package `treeautomaton` enthält als wichtigsten Bestandteil die Klasse `Automaton`. Instanzen von `Automaton` modellieren nichtdeterministische Bottom-Up-Baumautomaten. Sie werden initialisiert, indem ihnen zunächst Funktionssymbole (Klasse `FunctionSymbol`) zugewiesen werden, die sich in Namen und Arität unterscheiden können. Jedoch kann es nicht zwei Funktionssymbole mit demselben Namen und unterschiedlicher Arität geben. Mehrere `FunctionSymbol`-Instanzen bilden die Signatur des Baumautomaten. Weiterhin können dem Baumautomaten Zustände in Form von Instanzen der Klasse `State` und Regeln in Form von Instanzen der Klasse `Transition` hinzugefügt werden.
Eigene Klassen für Regeln und Zuständen erwiesen sich als sinnvoll, da sowohl für jeden einzelnen Zustand als auch für jede einzelne Regel Zusatzinformationen zu verwalten sind.
Ein wichtiger Unterschied der Implementierung zu den Ausführungen in 4.4 stellt die Entscheidung dar, Baumautomaten ohne ϵ-Regeln zu modellieren. Für Algorithmen wie die Produktautomatenkonstruktion oder Reduktionsalgorithmen ist die Behandlung von ϵ-Regeln unpraktisch und macht eine Implementierung kompliziert. Daher wurde der Klasse `Automaton` die Methode `simulateEpsTransition` hinzugefügt, mit der eine ϵ-Regel durch mehrere "gewöhnliche" Regeln simuliert werden kann (siehe 2.2).
Zwar ist es möglich, die Regelmenge eines Baumautomaten $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ als mehrdimensionale Matrizen vom Typ `boolean` darzustellen, z.B. für ein Funktionssymbol f mit Arität n als $(n+1)$-dimensionale Matrix M mit

$$M[i_1, ..., i_n, i] = \mathit{true} \Leftrightarrow f(q_{i_1}, ..., q_{i_n}) \rightarrow q_i \in \delta,$$

jedoch würde das bei fast allen Beispielen in der Praxis zu sehr großen und dünn besetzten Matrizen führen. Daher wurden Regeln des Automaten in der vorliegenden Implementation in Listen abgespeichert. Prinzipiell gibt es dafür mehrere Möglichkeiten, zum Beispiel:

- Für jeden Zustand q wird eine Liste der Regeln $f(q_1, ..., q_n) \rightarrow q$ abgespeichert,
- Für jedes Funktionssymbol f wird eine Liste von Regeln angelegt, die alle Regeln der Art $f(q_1, ..., q_n) \rightarrow q$ enthält; der jeweilige Listeneintrag enthält Verweise auf die Zustände $q_1, ..., q_n, q$.

In der Implementierung wurde eine Mischung beider Ansätze gewählt, da für unterschiedliche Algorithmen unterschiedliche Darstellungsformen Vorteile bringen. Abb. 5.2 zeigt die Darstellung der Regelmengen. Die einzelnen Instanzen von Zuständen und Regeln werden in den Listen nicht repliziert; vielmehr enthalten die Listen lediglich Referenzen auf die Instanzen. Mithilfe der Methodenfamilie `insertTransition` wird eine Regel in allen betreffenden Listen eingefügt.
Jeder Zustand besitzt außer der Liste der Regeln, die ihn als Zielzustand haben, noch unter anderem folgende Attribute:

- vier Flag-Bits: `reachable`, `updated`, `changed`, `addingnotpossible`,
- zwei Zustandsreferenzen: `productstate1` und `productstate2`,
- eine Objektreferenz `tag`,
- mehrere Attribute für Methoden der Zustandsminimierung sowie
- mehrere Attribute für die Verwaltung von Informationen des symbolischen Algorithmus.

Die Flag-Bits werden von Automatenoperationen und von den Algorithmen für $Post^*$ benutzt. `productstate1` und `productstate2` geben für einen Zustand q an, welche beiden Zustände in den Faktorautomaten diesen Zustand darstellen, falls q infolge einer Produktautomatenkonstruktion entstanden ist. Das `tag`-Objekt kann ebenso wie die Flag-Bits von verschiedenen Algorithmen als Zwischenspeicher verwendet werden. Daher ist zu beachten, dass der Inhalt des `tag`-Objekts sich nach Aufrufen von Methoden in `Automaton` ändern kann; hierbei ist die Dokumentation der einzelnen Methoden zu beachten.

5.2.1 Implementierte Operationen

Die Klasse `Automaton` unterstützt u.a. folgende Operationen:

- Schnitt zweier Automaten durch die statische Methode `intersect`,
- Berechnen aller Zustände, in die ein Term abgeleitet werden kann, durch die Methode `derive`,
- das vor allem im symbolischen Algorithmus notwendige Einbinden von Regeln und Zuständen eines Automaten in einen anderen durch die Methode `addAutomatonCopy`,
- eine Variante des Algorithmus aus Abb. 2.3 für das Entfernen unproduktiver Zustände durch die Methode `removeUnproductiveStates`,
- Entfernen von Zuständen, die nie innerhalb einer Reduktion vorkommen, welche in einem akzeptierenden Zustand endet, durch die Methode `removeUnderivableStates`,
- Deterministisch-Machen eines Baumautomaten mittels Potenzmengenkonstruktion durch die Methode `createDeterminizedAutomaton`.

Wir wollen kurz auf einige Besonderheiten der Methode `intersect` eingehen. Es ist nützlich, für einen Zustand im Produktautomaten zu wissen, aus welchen Zuständen er entstanden ist. Jeder Produktautomatenzustand stellt ein Paar von Zuständen aus den Faktorautomaten dar, siehe dazu Abschnitt 2.2.1. Bei einem Aufruf `intersect(a1, a2)` mit den Automaten `a1` und `a2` werden für jeden Zustand des Produktautomaten s die Attribute `productstate1` und `productstate2` gesetzt. Sie verweisen auf die

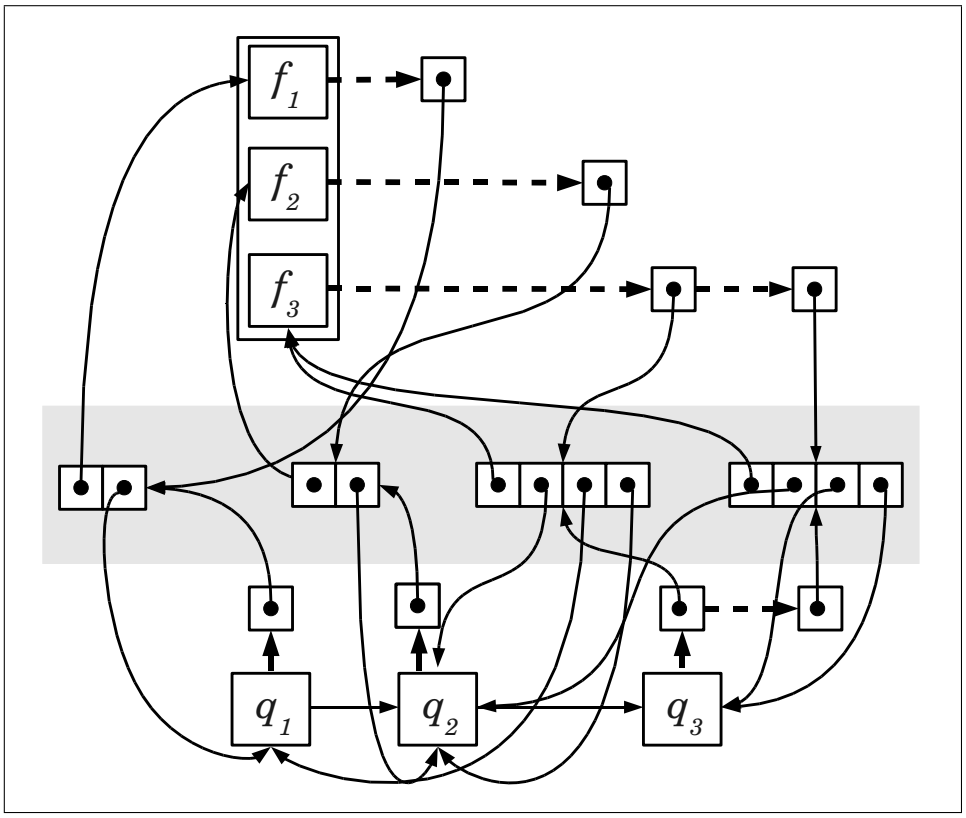


Abbildung 5.2: Beispiel für die Repräsentation der Automatenregeln in `Automaton`. Im grauen Bereich sind vier Regeln vorhanden, die eine Referenz auf ihr Funktionssymbol und Referenzen auf die an ihr beteiligten Zustände beinhalten. Zwei der Regeln besitzen eine Konstante als Funktionssymbol, so dass sie lediglich einen Zielzustand haben: $f_1 \rightarrow q_1$, $f_2 \rightarrow q_2$. Die anderen Regeln sind $f_3(q_2, q_1) \rightarrow q_2$ und $f_3(q_2, q_3) \rightarrow q_3$. Alle Zustände besitzen eine Regelliste, in denen Referenzen auf Regeln gespeichert werden, bei denen sie als Zielzustände vorkommen. Die Verzeigerungen dieser Regelliste sind im unteren Bereich gestrichelt dargestellt. Ebenso besitzt jedes Funktionssymbol eine Regelliste "seiner" Regeln, die im oberen Bereich ebenfalls gestrichelt dargestellt ist.

Zustände s_1 in `a1` bzw. s_2 in `a2`, für die gilt $s = (s_1, s_2)$. Außerdem initialisiert `intersect` die Hash-Tabelle `producthash`. Schlüssel hierfür stellen Zustandstupel dar (gekapselt in der Klasse `StateTupel`). Mit Hilfe des Hash kann durch die Kenntnis eines Zustandstupels (s_1, s_2) der Faktorautomaten auf den dazugehörigen Zustand im Produktautomaten zugegriffen werden.

5.3 Implementierung symbolischer Grundtermersetzungssysteme

Zur Darstellung und Manipulation von Termen und symbolischen Grundtermersetzungssystemen dient das Package `rewriting`. Es beinhaltet unter anderem die Klassen

- `Term` zur Repräsentation von Termen, sowohl Grundterme als auch Terme mit Variablen,
- `Rule` zur Repräsentation symbolischer Regeln eines Grundtermersetzungssystems und
- `RewritingSystem` zur Repräsentation eines symbolischen Grundtermersetzungssystems.

5.3.1 Die Klasse `Term`

Instanzen der Klasse `Term` stellen Baumknoten dar, die eine beliebige Menge von Kinderknoten derselben Klasse besitzen können. Die Anzahl der Kinder wird beim Konstruktoraufruf festgelegt und kann mittels der Methode `getArity` abgefragt, aber nicht mehr verändert werden. Jeder Knoten vom Typ `Term` besitzt außerdem ein Attribut `name` vom Typ String, das entweder ein Funktionssymbol oder einen Variablenbezeichner oder in bestimmten Situationen auch einen Sortenbezeichner beinhalten kann (siehe zu Letzterem Abschnitt 5.8).
Die Klasse `Term` wird unter anderem

- als Eingabe für Baumautomaten (Methode `Automaton.derive`),
- als linke oder rechte Seite einer symbolischen Regel eines Grundtermersetzungssystems und
- als Spezifikationsmittel für Termsorten (siehe 5.8) verwendet.

Ähnlich dem `tag`-Objekt bei der Klasse `Automaton` existiert für `Term` ein `value`-Objekt, das von Algorithmen zum Abspeichern von für sie wichtigen Objekten dienen kann. Der Knoten kann mittels der Methode `setVariable` als Variable gekennzeichnet werden; dann kann mittels `setVarId(int)` bzw. `getVarId()` die entsprechende Id der Variable gesetzt bzw. abgefragt werden. Das i-te Kind kann mittels `getChild(i)` bzw. `setChild(Term)` abgefragt bzw. gesetzt werden.

5.3.2 Die Klassen `Rule` und `RewritingSystem`

Instanzen der Klasse `Rule` stellen symbolische Regeln für Grundtermersetzungssysteme dar, die mit Def. 21 eingeführt wurden. Linke und rechte Seiten der Regeln werden als Instanzen von `Term` realisiert und können mittels `getLHS()` und `getRHS()` abgefragt werden. Die Klasse arbeitet stark mit `RewritingSystem` zusammen: Unter anderem wird eine Referenz auf das "aktuelle" `RewritingSystem` der Regel als Attribut verwaltet. Hierauf kann durch die Methoden `setRewritingSystem` und `getRewritingSystem` zugegriffen werden. Auch beim Konstruktoraufruf muss neben den beiden Termen für linke und rechte Seite ein `RewritingSystem`-Objekt übergeben werden. Eine Hauptaufgabe von `RewritingSystem` ist die Verwaltung der in seinen Regeln vorkommenden Variablen. Die möglichen Wertebereiche $dom_r(x)$ einer Variablen x einer symbolischen Regel werden in der Implementation auch mit *Sorten* bezeichnet. In vielen praktischen Beispielen kommen mehrere Variablen vor, die derselben Sorte angehören und daher lediglich *einen* gemeinsamen Automaten zur Repräsentation ihres Wertebereiches benötigen. Der Programmierer kann bei einem `RewritingSystem` eine neue Sorte registrieren, indem er die Methode `registerSort` mit einem Automaten aufruft, der exakt die Terme der Sorte erkennt. Die Methode liefert einen Integer-Wert, der die Sorte eindeutig kennzeichnet. Mittels `registerVariable(String, int)` können dann Variablen mit der übergebenen Sorte eingetragen werden; auch sie erhalten einen eindeutigen Integer-Bezeichner. Informationen über Variablen werden in der einfachen Datenstruktur `Variable` verwaltet. Statt für jede Variable x bei Aufruf von `registerSort` den übergebenen Automaten separat abzuspeichern, beinhaltet `RewritingSystem` ein einziges `Automaton`-Objekt `varaut`. In ihm sind alle Regeln und Zustände der Wertebereichsautomaten vereinigt vorhanden. Bei `registerSort` wird der übergebene Automat mit `varaut` vereinigt, und ein neuer Zustand wird der Sorte zugewiesen, der genau ihren Wertebereich erkennt. Dadurch können z.B. Zustände und Regeln von mehreren Sorten gemeinsam benutzt werden. Im späteren Verlauf des symbolischen Algorithmus wird `varaut` für Produktautomatenkonstruktionen benutzt, so dass ein kleinerer Automat Geschwindigkeitsvorteile bringen kann.

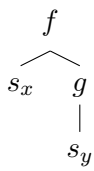
5.4 Implementierung des symbolischen Algorithmus

Kernstück der vorliegenden Implementation ist der Algorithmus aus 4.4. Mittels Aufruf von `performPostAnalysis` in der Klasse `Symbolic` des Package `symbolic` kann er gestartet werden. Der Algorithmus errechnet die Nachfolgermenge der Termmenge, die durch das erste Argument als Baumautomat gegeben ist bezüglich dem symbolischen Termersetzungssystem, das als zweites Argument übergeben wird. Im Folgenden soll die Realisierung der einzelnen Schritte des Algorithmus in der vorliegenden Implementierung aufgezeigt werden. Im Vergleich zum ursprünglichen Ansatz aus Abb. 4.4 wurden mehrere Modifikationen vorgenommen, die zum Ziel hatten, sowohl das

Laufzeitverhalten zu verbessern als auch den Speicherbedarf des Algorithmus zu senken.

5.4.1 Ein Iterationsschritt

Analog zu Abb. 4.4 wollen wir den Automaten $\mathcal{A}_i$ als jenen Automaten bezeichnen, der am Ende des i-ten Schleifendurchlauf, also nach dem i-ten Iterationsschritt, konstruiert wird.
Wir wollen in diesem Abschnitt vor allem erläutern, wie die Prozedur RIGHTSIDES in der vorliegenden Implementation realisiert wurde. Es wurde versucht, möglichst wenige Produkautomatenkonstruktionen durchzuführen, um Laufzeit und Speicherplatz zu sparen. Hierzu hilft uns die kompakte Darstellung aller Wertebereiche von Variablen im Automaten `varaut` eines `RewritingSystem` : Bei einem Interationsschritt des Algorithmus wird ein Produktautomat $P_i = \texttt{varaut} \times \mathcal{A}_i$ konstruiert, wobei wir alle Zustände aus `varaut` und $\mathcal{A}_i$ so behandeln, als wären sie akzeptierend. In P_i befinden sich dann nur akzeptierende Zustände. Die Sprache von P_i ist für uns nicht interessant, wohl aber seine Zustände und Regeln. Mit ihrer Hilfe kann errechnet werden, welche Zustände aus $\mathcal{A}_i$ eine Teilmenge des Wertebereichs einer Variablen x erkennen: Bezeichne q_x den ausgezeichneten Zustand für die Variable x in `varaut` und q einen Zustand in $\mathcal{A}_i$. q erkennt Terme aus dem Wertebereich von x, wenn der Zustand (q_x, q) aus P_i mindestens einen Term erkennt.
Statt also für jede Regel eine Produktautomatenkonstruktion durchzuführen, wird pro Schleifendurchlauf lediglich eine einzige Produktautomatenkonstruktion angewandt. Aus P_i kann nun geschlossen werden, welche Zustände aus $\mathcal{A}_i$ eine Teilmenge des Wertebereichs einer Variablen x erkennen: Bezeichne q_x den ausgezeichneten Zustand für die Variable x in `varaut` und q einen Zustand in $\mathcal{A}_i$, so erkennt q Terme aus dem Wertebereich von x, wenn der Zustand (q_x, q) aus P_i mindestens einen Term erkennt.
Wie kann mithilfe von P_i berechnet werden, welche entfalteten linken Seiten einer symbolischen Regel durch einen Zustand q aus $\mathcal{A}_i$ erkannt werden können? Betrachten wir als kleines Beispiel die symbolische Regel $r = f(x, g(y)) \rightarrow f(g(x), g(y))$, wobei als Wertebereich für x und y jeweils $\{a, g(a), g(g(a))\}$ angenommen wird. Nehmen wir weiterhin an, dass z.B. der Term $f(g(g(a)), g(g(g(a))))$ durch einen Zustand q in $\mathcal{A}_i$ erkannt wird. x wird durch $g(a)$, y durch $g(g(a))$ substituiert. Wir betrachten die möglichen Reduktionen für $f(g(g(a)), g(g(g(a)))) \rightarrow^*_{\mathcal{A}_i} q$ und erkennen, dass es eine Reduktion geben muss, bei der der Term



vorkommt; es gibt also zwei Zustände s_x, s_y in der Reduktion, die für die Variablen substituierten Terme erkennen. Des Weiteren gibt es Regeln, die aus

$f(s_x, g(s_y))$ den Zustand q ableiten können. Im Produktautomat P_i können wir sehen, dass die Zustände (q_x, s_x) und (q_y, s_y) eine nichtleere Menge von Termen erkennen können.
Unser Vorgehen ist daher, für jeden Zustand q in $\mathcal{A}_i$ und jede symbolische Regel $r = t_l \to t_r$ zu versuchen, "rückwärts" von q ausgehend eine Reduktion für t_l zu erstellen, so dass alle Funktionssymbole der Regel abgeleitet wurden, außer denen der Variablen. Die Zustände, die an Stelle der Variablen stehen, geben uns Aufschluss über die Elemente des Wertebereichs, für die t_l nach q abgeleitet werden kann. Diese benutzen wir für die Ableitung der entsprechenden Elemente aus $Expand(t_r)$ nach q, denn: Elemente aus $Expand(t_r)$ unterscheiden sich nur durch die Terme, durch die Variablen in t_r substituiert wurden. Mithilfe des Automaten P_i kann das Ableiten rechter Regelseiten ermöglicht werden.

Implementiertes Verfahren

Wir wollen zunächst ein ausführlicheres Beispiel untersuchen.

Beispiel 15. Wir betrachten den Automaten

$$\mathcal{A} = (\{q_1, q_2, q_3, q_4\}, \{a(), b(), f(\cdot), g(\cdot,\cdot)\}, \{q_3\}, \delta)$$

mit folgenden Regeln in δ:

1. $a \to q_1$
2. $b \to q_2$
3. $g(q_1, q_2) \to q_3$
4. $g(q_3, q_1) \to q_3$
5. $g(q_4, q_2) \to q_3$
6. $f(q_2) \to q_3$
7. $f(q_1) \to q_4$

Es sei außerdem die symbolische Regel $g(f(x), y) \to g(x, y)$ eines symbolischen Grundtermersetzungssystems gegeben, wobei die Variablen x und y den Wertebereich $\{a, b\}$ haben. Wir wollen für den Zustand q_3 entscheiden, welche Zustände als "Variablenzustände" für die Regel in Frage kommen: Welche Zustände v_1, v_2 gibt es, so dass $(g(f(v_1), v_2) \to^*_{\mathcal{A}} q_3$ und $\mathcal{L}(v_1) \cap \{a, b\} \neq \emptyset$ und $\mathcal{L}(v_2) \cap \{a, b\} \neq \emptyset$ gilt? Wir leiten also den Term $g(f(x), y)$ "rückwärts" ab und suchen zunächst eine Regel der Art $g(\cdot\,,\cdot) \to q_3$. Diese finden wir in den Regeln 3, 4 und 5. Bei Anwendung von Regel 3 erhalten wir den Term $g(q_1, q_2)$. Wir versuchen nun weiter, q_1 rückwärts abzuleiten und suchen eine Regel der Art $f(\cdot) \to q_1$, welche nicht existiert. Bei Anwendung von Regel 4 erhalten wir den Term $g(q_3, q_1)$, und auch q_3 können wir ableiten durch Regel 6. Dadurch erhalten wir den Term $g(f(q_2), q_1)$. Bei Anwendung von Regel 5 erhalten wir $g(q_4, q_2)$, hier können wir mit Regel 7 weiter ableiten zu $g(f(q_1), q_2)$.

Insgesamt erhalten wir zwei mögliche Zustandstupel (q_2, q_1) und (q_1, q_2), wobei das erste Element für die Variable x, das zweite für y steht. Zu beachten ist, dass z.B. die Kombination von q_2 für x und q_2 für y nicht unbedingt eine Teilmenge der erkannten linken Seiten von q_3 darstellt: Es ist wichtig, die Beziehungen bei der Ableitung in Form der Zustandstupel zu berücksichtigen. Mithilfe der Produktautomatenkonstruktion können wir noch feststellen, dass q_1, q_2 Terme aus dem Wertebereich der Variablen erkennen: Nehmen wir an, dass der Automat

$$\mathcal{A}_{var} = (\{q_{var}\}, \{a(), b()\}, \{q_{var}\}, \{a \to q_{var}, b \to q_{var}\})$$

gegeben ist, der den Wertebereich von x und y erkennt. Wir bilden den Automaten

$$P = (\{(q_{var}, q_1), (q_{var}, q_2)\}, \{a(), b()\}, \{(q_{var}, q_1), (q_{var}, q_2)\},$$
$$\{a \to (q_{var}, q_1), b \to (q_{var}, q_2)\}),$$

indem wir die Produktautomatenkonstruktion $\mathcal{A}_{var} \times \mathcal{A}$ durchführen und dabei alle Zustände von $\mathcal{A}_{var}$ und $\mathcal{A}$ als akzeptierend betrachten. q_1 und q_2 erkennen Terme aus dem Wertebereich der Variablen. Wir können mithilfe von P Automaten für die entsprechenden rechten Seiten der beiden Tupel konstruieren. Wir stellen beispielhaft den Automaten für das Tupel (q_2, q_1) dar:

$$\mathcal{A}_{(q_2,q_1)} = (\{q_f, (q_{var}, q_1), (q_{var}, q_2)\}, \{a(), b(), f(\cdot), g(\cdot,\cdot)\}, \{q_f\}, \delta_{(q_2,q_1)}), \text{ mit}$$
$$\delta_{(q_2,q_1)} = \{g((q_{var}, q_2), (q_{var}, q_1)) \to q_f, a \to (q_{var}, q_1), b \to (q_{var}, q_2)\}).$$

$\mathcal{A}_{(q_2,q_1)}$ erkennt im Beispiel nur den Term $g(b, a)$, die rechte Seite der entfalteten Regel $g(f(b), a) \to g(b, a)$. Wir können die Zustände und Regeln von P zur Konstruktion der Automaten für die zu erkennenden rechten Seiten der Regeln nutzen. □

Die möglichen Kombinationen von Variablenzuständen, mit denen auf diese Art nach q abgeleitet werden kann, werden für jeden Zustand gesammelt. Später werden mit ihrer Hilfe Automaten dargestellt, die die entsprechenden rechten Seiten erkennen. Die Methode `createVariableStates(State)` der Klasse `Rule` übernimmt das Sammeln der Variablenzustände: Sie liefert eine Liste von Zustandstupeln zurück, wobei jedes Tupel für jede in der Regel vorkommende Variable einen Zustand enthält. Alle linken Seiten einer Regel, die der Zustand q erkennt, lassen sich so kompakt durch eine Menge von Zustandstupeln und einen Term mit Variablen beschreiben: Sie repräsentieren Termmengen ähnlich der Darstellung symbolischer Regeln.
Ebenso einfach lässt sich für ein Zustandstupel ein Automat erstellen, der genau die entsprechenden rechten Seiten erkennt: Es müssen Regeln und Zustände für den "variablenfreien" Teil der rechten Seite einer symbolischen Regel hinzugefügt werden; anstelle der Variablen werden die entsprechenden Zustände der Tupel benutzt. Dies besorgt die Methode `createCopy` der Klasse `Rule`. Wird ein Automat für die erkannten rechten Seiten eines Zustands in $\mathcal{A}_i$ eingefügt, so benutzt er die Zustände von P_i mit, was zu einer Reduzierung

der eingefügten Zustände führt. Am Ende des Schleifendurchlaufs werden die benötigten Regeln und Zustände von P_i in dem Automaten $\mathcal{A}_i$ eingefügt.
Wir sind nun in der Lage, das Verfahren zur Simulation der Ableitung rechter Regelseiten darzustellen. Zunächst beschränken wir uns bei den Beschreibungen auf eine einzelne symbolische Regel $r = t_l \rightarrow t_r$. Wir nehmen an, dass $\mathcal{A}_i$ im Laufe der Ausführung bereits erstellt wurde, aber noch nicht $\mathcal{A}_{i+1}$. Außerdem benötigen wir noch einiges an Notation: Sei $k = |\,Var(t_l)|$. Die Menge aller in r vorkommenden Variablen sei mit $x_1, ..., x_k$ bezeichnet. q_{x_i} sei der ausgezeichnete Zustand der Variablen x_i im Variablenautomaten `varaut`. Wir bezeichnen mit $t[x_1 \mapsto q_1][x_2 \mapsto q_2]...[x_k \mapsto q_k]$ den Term, der aus t entsteht, wenn darin Variable x_i durch den Zustand q_i substituiert wurde für $i \in \{1, ..., n\}$. Wie bereits oben erläutert existiert ein Automat P_i, der aus dem Produkt von `varaut` und $\mathcal{A}_i$ entstanden ist.
Für jeden Zustand q in $\mathcal{A}_i = (Q, \Sigma, Q_f, \delta)$ führe Folgendes aus:

- Finde die Menge T aller Tupel $v = (q_1, ..., q_k)$ aus Q^k, für die gilt:

$$t_l[x_1 \mapsto q_1]...[x_k \mapsto q_k] \rightarrow^*_{\mathcal{A}_i} q$$

- Bilde für jedes Tupel v aus T einen Automaten $\mathcal{A}_v$, der alle Ausdrücke der Art $t_r[x_1 \mapsto (q_{x_1}, q_1)]...[x_k \mapsto (q_{x_k}, q_k)]$ auf einen akzeptierenden Zustand reduziert. $\mathcal{A}_v$ hat O.b.d.A. nur einen Endzustand q_v. Die Zustände (q_{x_i}, q_i) stammen aus P_i.

- Für jeden Automat $\mathcal{A}_v$ führe folgende Schritte aus:
  - Füge zu $\mathcal{A}_v$ die Regeln und Zustände aus P_i hinzu, die in einer Reduktion in P_i nach (q_{x_1}, q_1), ..., (q_{x_k}, q_k) vorkommen.
  - Falls bereits ein Tupel $(\mathcal{A}, q')$ in $Copies_r$ eingefügt wurde, bei dem $\mathcal{A}$ sprachäquivalent zu $\mathcal{A}_v$ ist: Füge lediglich die Regel $q' \rightarrow q$ ein.
  - Sonst: Füge Zustände und Regeln der erstellten Automaten $\mathcal{A}_v$ in $\mathcal{A}_{i+1}$ ein. q_v ist in $\mathcal{A}_{i+1}$ als nicht akzeptierend markiert. Füge Regeln $q_v \rightarrow q$ in $\mathcal{A}_{i+1}$ ein. Speichere das Tupel $(\mathcal{A}_v, q_v)$ in der Liste $Copies_r$.

Die Vereinigung aller Automaten $\mathcal{A}_v$, die für einen Zustand q und eine symbolische Regel r erzeugt werden, entsprechen den Automaten $\mathcal{A}_{r,q}$ aus dem vorherigen Kapitel. Werden mehrere symbolische Regeln verwendet, wird das Verfahren für alle weiteren Regeln wiederholt.

5.4.2 Kopienverwaltung

In Abschnitt 4.4 wurde bereits darauf hingewiesen, dass vor dem Einfügen von neuen Zuständen und Regeln getestet werden muss, ob nicht bereits derselbe Automat zu einem früheren Zeitpunkt eingefügt wurde. Hierzu verwaltet jedes Objekt der Klasse `Rule` eine Liste von Baumautomaten und Zuständen analog zu den Listen $Copies_r$ aus dem Algorithmus in Abb. 4.4 in Gestalt des Attributs `copies`. Die Einträge der Liste sind vom Typ `AutomatonCopy`. Dieser speichert im Attribut `fixed` einen Zustand aus den

$\mathcal{A}_i$-Automaten. Im Attribut `mindetaut` wird ein Baumautomat gespeichert. Das Tupel (`mindetaut`,`fixed`) (bzw. mehrere Tupel dieser Art mit dem gleichen `fixed`-Attribut) entspricht einem Tupel $(\mathcal{A}_{r,q,i}, q)$ aus den $Copies_r$-Listen (siehe 4.4).
Die Automaten in dieser Liste sind deterministisch gemacht und in Minimalform gebracht worden. Soll nun für einen Zustand q ein Automat in $\mathcal{A}_i$ eingefügt werden, der rechte Seiten einer symbolischen Regel erkennt, so wird der Automat zunächst deterministisch gemacht und minimiert. Anschließend wird er mit den Automaten in der Liste `copies` verglichen, die dieselbe Anzahl von Zuständen besitzen. Erkennen wir, dass derselbe Automat bereits eingefügt wurde, so simulieren wir lediglich ϵ-Regeln von `fixed` nach q. Ansonsten fügen wir den Automaten in `copies` und die entsprechenden Zustände und Regeln in $\mathcal{A}_i$ ein, wie bereits erläutert.

5.4.3 Beispiel

Wir wollen zur Veranschaulichung des implementierten Algorithmus ein Beispiel betrachten.
Wir wählen eine Signatur

$$\Sigma = \{a(), b(), nil(), c(\cdot,\cdot), append(\cdot,\cdot)\}.$$

Auf dieser aufbauend betrachten wir das symbolische Grundtermersetzungssystem R mit den Regeln

- $r_1 = append(nil, xs) \to x$
- $r_2 = append(c(x, xs), ys) \to c(x, append(xs, ys))$

Als Wertebereich für die Variable x wählen wir die Termmenge $C = \{a, b\}$ und für xs und ys die Termmenge $L = \{nil, c(a, nil), c(b, nil)\}$.
Die Implementierung erstellt nun einen Variablenautomaten

$$\mathcal{A}_{var} = (\{q_L, q_C, q_0\}, \Sigma, \emptyset, \delta_{var})$$

mit der Regelmenge

$$\delta_{var} = \{a \to q_C, b \to q_C, nil \to q_0, nil \to q_L, c(q_C, q_0) \to q_L\}$$

Der Zustand q_C erkennt genau die Menge C, der Zustand q_L genau die Menge L. Sowohl eine Darstellung des symbolischen Grundtermersetzungssystems als auch des Automaten $\mathcal{A}_{var}$ wird als Attribut einer `RewritingSystem`-Instanz abgespeichert. Hierbei entspricht $\mathcal{A}_{var}$ dem Attribut `varaut`. Ebenso werden Listen $Copies_{r_1}$ und $Copies_{r_2}$ als leere Listen initialisiert:

$$Copies_{r_1} = \emptyset, Copies_{r_2} = \emptyset$$

Diese entsprechen den `copies`-Attributen in der vorliegenden Implementierung.

Wir sind daran interessiert, von der Termmenge

$$M = \{append(c(b, nil), nil), append(c(a, nil), c(b, nil))\}$$

die Nachfolgerterme bezüglich R zu berechnen. Dazu codieren wir M als Automaten $\mathcal{A}_0 = (\{s_b, s_a, s_0, s_1, s_2, s_f\}, \Sigma, \{s_f\}, \delta_0)$ mit der Regelmenge

$$\delta_0 = \{b \to s_b, a \to s_a,$$

$$nil \to s_0, c(s_a, s_0) \to s_1, c(s_b, s_0) \to s_2,$$

$$append(s_2, s_0) \to s_f, append(s_1, s_2) \to s_f\}.$$

Dieses Beispiel wurde bewusst einfach gehalten, um ein gewisses Maß an Übersichtlichkeit zu erreichen. Ein großer Vorteil des Verfahrens besteht darin, eine große Menge von Termen mithilfe von Baumautomaten zu codieren, was jedoch das Beispiel hier unnötig kompliziert machen würde.

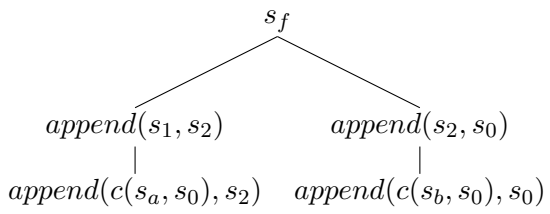
Ein Iterationsschritt

Zu Beginn des i-ten Iterationsschritts wird zunächst ein Automat P_i konstruiert, indem die Produktautomatenkonstruktion $\mathcal{A}_{var} \times \mathcal{A}_i$ ausgeführt wird, wobei alle Zustände in $\mathcal{A}_{var}$ und $\mathcal{A}_i$ als akzeptierend behandelt werden.
Im Weiteren wollen wir uns bei der Angabe der konstruierten Automaten auf die Nennung der Regeln und evtl. akzeptierender Zustände des Automaten beschränken. Die Regeln von P_1 sind

$$a \to (q_C, s_a), b \to (q_C, s_b), nil \to (q_0, s_0), nil \to (q_L, s_0),$$

$$c((q_C, s_a), (q_0, s_0)) \to (q_L, s_1), c((q_C, s_b), (q_0, s_0)) \to (q_L, s_2).$$

Zunächst betrachten wir die Zustände in $\mathcal{A}_0$, die eine Teilmenge von C erkennen: Dies sind s_a und s_b, da die Zustände $(q_C, s_a), (q_C, s_b)$ jeweils produktiv in P_1 sind. Bei der Menge L sind es die Zustände s_0, s_1 und s_f.
Der Algorithmus arbeitet nun jede Regel des symbolischen Grundtermersetzungssystems und jeden Zustand in $\mathcal{A}_0$ ab. Wir beginnen mit r_1 und suchen für jeden Zustand q, von q ausgehend, Reduktionen, die eine entfaltete linke Seite von r_1 erkennen können. Für r_1 gibt es im ersten Iterationsschritt keinen Zustand, der eine solche erkennt. Wir gehen über zu Regel r_2 und betrachten exemplarisch die Reduktionen für den Zustand s_f, die wir als Baum darstellen wollen, wobei jeder Pfad von der Wurzel zu einem Blatt eine Reduktion darstellt:



Die Blätter dieses Baumes stellen die Ergebnisse der Analyse des Zustandes s_f bezüglich der Regel r_2 dar. Sie sind mit Termen beschriftet, die die Form potentieller Kandidaten für linke Seiten der Regel darstellen. Für $append(c(s_a, s_0), s_2)$ wissen wir, dass s_0 eine Teilmenge des Wertebereiches von xs und s_2 eine Teilmenge des Wertebereichs von ys erkennt. Ebenso erkennt s_a eine Teilmenge des Wertebereichs von x. Für die Zustände im Term $append(c(s_b, s_0), s_0)$ gilt ähnliches.
Mit den Zustandstupeln $(x = s_a, xs = s_0, ys = s_2)$ und $(x = s_b, xs = s_0, ys = s_0)$ können wir nun folgende Regelmengen erstellen, die zur Ableitung der *rechten* Regelseiten dienen:

1. Für (s_a, s_0, s_2) die Regeln $append((q_L, s_0), (q_L, s_2)) \to p_1, c((q_C, s_a), p_1) \to p_{f_1}$,
2. Für (s_b, s_0, s_0) die Regeln $append((q_L, s_0), (q_L, s_0)) \to p_2, c((q_C, s_b), p_2) \to p_{f_2}$.

Hierbei werden einerseits neue Zustände p_1, p_2, p_{f_1} und p_{f_2} erzeugt, andererseits die Zustände aus dem Automaten P_1 wiederverwendet. Wir nennen die aus diesen beiden Regelmengen und den Regeln von P_1 entstehenden Automaten $\mathcal{C}_1$ und $\mathcal{C}_2$. Sie stellen Automaten der Art $\mathcal{A}_v$ der vorherigen Abschnitte dar. Ihre akzeptierenden Zustände sind jeweils p_{f_1} und p_{f_2}. Die Regeln und Zustände von $\mathcal{C}_1$ und $\mathcal{C}_2$ müssen mit den Automaten in $Copies_{r_2}$ auf Äquivalenz verglichen werden; jedoch ist $Copies_{r_2}$ zu diesem Zeitpunkt noch leer. Der Automat $\mathcal{A}_1$ wird sich demnach aus ganz neuen Zuständen (wie hier z.B. p_1 etc.), den Zuständen von $\mathcal{A}_0$ und einigen Zuständen aus P_1 zusammensetzen. Kein weiterer Zustand kann linke Seiten von r_2 ableiten, so dass wir den Automaten $\mathcal{A}_1$ mit folgenden Regeln aufbauen können:

$$\delta_1 = \delta_0 \cup$$

$$\{append((q_L, s_0), (q_L, s_2)) \to p_1, c((q_C, s_a), p_1) \to p_{f_1},$$
$$append((q_L, s_0), (q_L, s_0)) \to p_2, c((q_C, s_b), p_2) \to p_{f_2},$$
$$a \to (q_C, s_a), b \to (q_C, s_b), nil \to (q_0, s_0), nil \to (q_L, s_0),$$
$$c((q_c, s_b), (q_0, s_0)) \to (q_L, s_2),$$
$$p_{f_1} \to s_f, p_{f_2} \to s_f\}$$

Wir wollen anmerken, dass die Regel $c((q_C, s_a), (q_0, s_0)) \to (q_L, s_1)$ nicht in $\mathcal{A}_1$ übernommen wurde, da sie von den eingefügten Automaten für die rechten Seiten nicht benötigt wird.
In die Liste $Copies_{r_2}$ wurden zwei Einträge hinzugefügt, nämlich die Automaten $\mathcal{C}_1$ und $\mathcal{C}_2$. Implementierungstechnisch werden hierzu echte Kopien der Automaten angefertigt, d.h., sie haben keine Zustände oder Regeln mit $\mathcal{A}_0$ gemeinsam (sehr wohl jedoch mit P_1), insbesondere sind auch die akzeptierenden Zustände Kopien. Dies ist notwendig, da sich innerhalb der Automaten $\mathcal{A}_i$ die Termmenge vergrößern kann, die p_{f_1} bzw. p_{f_2} erkennt. Jedoch werden Referenzen auf die Zustände p_{f_1} und p_{f_2} aus $\mathcal{A}_0$ abgespeichert. Dafür wird

das bereits erwähnte `fixed`-Attribut der `copies`-Elemente verwendet. Sollte im weiteren Verlauf der Ausführung z.B. ein zu $\mathcal{C}_2$ äquivalenter Automat eingefügt werden, um für einen Zustand q das Ableiten rechter Regelseiten von r_2 zu simulieren, so wird lediglich eine ϵ-Regel von p_{f_1} nach q eingefügt. Der neue Automat $\mathcal{A}_1$ erkennt genau $Post(\mathcal{L}(\mathcal{A}_0)) \cup \mathcal{A}_0$.

Der Algorithmus würde nun mit weiteren Iterationsschritten fortfahren, da neue Regeln in $\mathcal{A}_1$ auftauchen. Für den nächsten Iterationsschritt müssen auch die Zustände $p_1, p_2, p_{f_1}, p_{f_2}, (q_L, s_0), ...$ untersucht werden. Einige von ihnen erkennen auch entfaltete linke Seiten der ersten symbolischen Regel, genauer gesagt p_1 und p_2. Außerdem wurden in die `copies`-Listen Einträge gespeichert, die auch berücksichtigt werden müssen. Die Vorgehensweise ist ansonsten analog zum ersten Iterationsschritt.

5.4.4 Beschleunigung des Verfahrens

Für jeden Zustand eines Automaten $\mathcal{A}_i$ werden ein `changed`- und ein `updated`-Flag verwaltet. Wird $\mathcal{A}_i$ während eines Schleifendurchlaufs eine Regel hinzugefügt, die den Zustand q als Zielzustand hat, so wird das `changed`-Flag von q auf `true` gesetzt: $\mathcal{L}(q)$ könnte sich verändert haben, und daher muss im nächsten Durchlauf getestet werden, ob neue linke Seiten einer Regel von q erkannt werden. Auch ein Zustand q' kann durch solch eine hinzugefügte Regel eventuell mehr linke Seiten erkennen, wenn es einen Term in $\mathcal{L}(q')$ gibt, bei dessen Reduktion q vorkommen kann. Daher wird am Ende eines Schleifendurchlaufs eine Erreichbarkeitsanalyse durchgeführt, bei der die Zustände q' mit dieser Eigenschaft identifiziert werden. Dies geschieht durch die Methode `Automaton.calculateTransitiveChangedStates`. Im nächsten Schleifendurchlauf wollen wir sowohl die `changed`-Informationen des letzten Schleifendurchlaufs verwenden als auch neue Informationen durch neu hinzugefügte Regeln und Zustände abspeichern. Daher wird ebenfalls am Ende eines Schleifendurchlaufs das `changed`-Flag eines Zustands abgespeichert in sein `updated`-Flag; dieses stellt die Information aus dem letzten Schleifendurchlauf dar und wird während dem Hinzufügen von Regeln und Zuständen nicht verändert.

Um die Zahl der Äquivalenztests gering zu halten, wird für jeden Zustand in $\mathcal{A}_i$ eine weitere Liste der Automaten verwaltet, die bereits *für diesen Zustand* eingefügt wurden. Diese werden zuerst auf Äquivalenz mit dem neu einzufügenden Automaten geprüft. Dies ist sinnvoll, da ein Zustand q durch zusätzlich eingefügte Regeln zwar oft mehr Terme erkennt, jedoch nicht unbedingt zusätzliche linken Seiten einer symbolischen Regel. Das bedeutet, dass bereits ein Automat an q "angehängt" wurde, der alle notwendigen rechten Regelseiten erkennt. Durch Untersuchen der q-Liste kann dies schnell festgestellt werden. Erst nach Überprüfen der Elemente aus der q-Liste werden die Automaten untersucht, die an andere Zustände angehängt wurden.

5.5 Implementierung des nichtsymbolischen Algorithmus

Zu Referenzzwecken wurde eine einfache Version des Algorithmus aus [Löd02] implementiert. Hierbei wurde der Ansatz verfolgt, die Regeln des symbolischen Regelsystems zu entfalten (siehe Abschnitt 4.1). Zu diesem Zweck befinden sich im Package `nonsymbolic` die Klassen `RuleExpander` und `ListExpander`. Wir beschränken uns beim nichtsymbolischen Ansatz auf symbolische Grundtermersetzungssysteme, deren Variablen als Wertebereich entweder $\{0, 1\}$ oder die Menge aller Listenterme der Form $c(a_1, c(a_2, ..., c(a_n, [])...))$ mit $a_i \in \{0, 1\}$ für alle $i \leq n$ mit einer maximalen, frei wählbaren Tiefe n besitzen. Trotz dieser Beschränkung konnten mit dieser Algorithmusversion viele Testfälle modelliert werden. Die Klasse `ListExpander` dient dazu, für Listenterme einer bestimmten Tiefe Zustände und Regeln in einen Automaten einzufügen, damit in diesem Automaten für jeden unterschiedlichen Listenterm t ein Zustand existiert, der nur t erkennt. Instanzen der Klasse `ListExpander` verwalten dazu ein zweidimensionales Array `elements`, in dem die Zustände gespeichert werden. `elements[i][j]` enthält eine Referenz auf den Zustand, der den Listenterm der Länge i enthält, dessen Binärdarstellung j entspricht. Z.B. würde für einen Term $c(0, c(1, c(0, c(1, []))))$ mit der Binärdarstellung $0101_2 = 5$ der zugehörige Zustand in `elements[4][5]` zu finden sein. Damit können wir einen Automaten um die notwendigen Zustände und Regeln für die Variablen von symbolischen Regeln anreichern.
Um auch Zustände für die entfalteten rechten Regelseiten selbst hinzufügen zu können, wird die Klasse `RuleExpander` benötigt. Sie verwendet das `elements`-Attribut einer `ListExpander`-Instanz und eine übergebene `Rule`-Instanz, um dem Automaten die entsprechenden Zustände und Regeln für die rechten Regelseiten hinzuzufügen. Objekte der Klasse `ExpandedRule` speichern die Information, welche Zustände welcher entfalteten Regel entsprechen.
Für die Ausführung des eigentlichen Algorithmus ist die Klasse `Nonsymbolic` zuständig: Sie leitet jede entfaltete linke Seite der Regeln ab, merkt sich, in welche Zustände M dabei reduziert werden konnte, und simuliert ϵ-Regeln zwischen den Zuständen aus M und den entsprechenden Zuständen der rechten Seiten.

5.6 Reduktionsalgorithmen

Bei starker Zunahme der Zustandszahl verlängert sich die Laufzeit des symbolischen Algorithmus: Für jeden Zustand muss festgestellt werden, welche linken Seiten einer Regel er erkennt. Ebenso können besonders bei einer symbolischen Regel, die mehr als eine Variable beinhaltet, auch viele Kombinationen von Zuständen für die Variablen auftreten (siehe 5.4.1), die mögliche Ableitungswege für linke Seiten der Regel darstellen. Daher lag der Gedanke nahe, nach Abarbeiten eines Iterationsschritts Reduktionsverfahren auf die

entstehenden Automaten anzuwenden, um dann mit einem verkleinerten Automaten weiterzuarbeiten. Unter einem Reduktionsverfahren für einen Baumautomaten $\mathcal{A}$ wollen wir in diesem Zusammenhang einen Algorithmus verstehen, der aus $\mathcal{A}$ einen Automaten $\mathcal{A}'$ erzeugt, wobei $\mathcal{A}'$

- dieselbe Sprache erkennt wie $\mathcal{A}$ und
- gleich viel oder weniger Zustände hat wie $\mathcal{A}$.

Bekanntes Beispiel eines Reduktionsverfahrens für endliche Wortautomaten ist die Minimierung deterministischer Automaten, bei der die Zustandszahl der Automaten minimiert wird. Ein analoges Verfahren existiert auch für Baumautomaten (siehe [CDG+07] und Abschnitt 5.6.2), wir benutzen eine Variante davon für den Äquivalenztest von Baumautomaten. Jedoch bringt das Determinieren eines großen Automaten, das vor einer solchen Minimierung geschehen muss, oftmals eine Zustandsexplosion mit sich. Daher ist die Minimierung eines deterministisch gemachten Baumautomaten in den meisten Fällen nicht praktikabel, wenn die Zustandszahl sehr groß ist. Die während dem symbolischen Algorithmus deterministisch gemachten und minimierten Zustände sind meist erheblich kleiner als die Gesamtautomaten, so dass hierbei keine starken Laufzeiteinbußen beobachtet wurden.
Nichtdeterministische Baumautomaten sind nur mit erheblichem Aufwand in eine Minimalform bezüglich der Zustandsanzahl zu überführen (siehe [MS72]). Es existieren jedoch schnell berechenbare Verfahren, die zwar nicht immer Automaten mit minimaler Zustandsanzahl liefern, jedoch oftmals eine sehr gute Annäherung.
Im Rahmen dieser Arbeit wurde ein Verfahren eingesetzt, dass sich auf eine spezielle Äquivalenzrelation auf Zuständen stützt, die sogenannte **Bisimulationsäquivalenz**. Bisimulationen wurden bereits für viele formale Modelle eingesetzt, z.B. allgemeine Graphen mit beschrifteten Kanten oder endliche Wortautomaten. [AKH06] und [HMM07] passten das Konzept der Bisimulationsäquivalenz auf Baumautomaten an. Mehrere mögliche Varianten wurden dabei eingeführt.
Im Rahmen dieser Arbeit kam vor allem die sogenannte **Backward**-Bisimulation zum Einsatz, für die Minimierung deterministischer Automaten wurde hingegen auf die sogenannte **Forward**-Bisimulation zurückgegriffen.
Beide Verfahren erstellen aus einer Äquivalenzrelation über den Zuständen eines Automaten einen neuen Automaten, den **Aggregations-Automat**:

Definition 24. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln und $\sim \subseteq Q \times Q$ eine Äquivalenzrelation über Q. Dann bezeichnen wir den Automaten $\mathcal{A}_\sim := (Q_\sim, \Sigma, Q_f^\sim, \delta_\sim)$, mit

- $Q_\sim = \{[q]_\sim \mid q \in Q\}$,
- $Q_f^\sim = \{[q]_\sim \mid q \in Q \wedge [q]_\sim \cap Q_f \neq \emptyset\}$ und
- $\delta_\sim = \{f([q_1], ..., [q_n]) \to [q] \mid \exists f(q_1, ..., q_n) \to q \in \delta\}$ □

als **Aggregations-Automaten** von $\mathcal{A}$.
Die Backward-Bisimulation ist für die Reduzierung der Automaten $\mathcal{A}_i$ des symbolischen Algorithmus besonders nützlich, da hierbei jeder Zustand einer Äquivalenzklasse dieselbe Menge von Termen erkennt. Der Repräsentant der Äquivalenzklasse im Aggregations-Automat erkennt daher auch genau diese Termmenge. So ist es insbesondere möglich, die $Copies$-Listen der symbolischen Regeln aktuell zu halten. Innerhalb dieser Listen werden neben Automaten auch Zustände aus den Automaten $\mathcal{A}_i$ im Attribut `fixed` gespeichert, die einfach durch den Repräsentanten ihrer Äquivalenzklasse im Aggregations-Automat ersetzt werden können.
Die Forward-Bisimulation hat diese Eigenschaft nicht. Daher ist sie für die Reduktion der $\mathcal{A}_i$-Automaten eher ungeeignet. Der Aggregations-Automat eines deterministischen vollständigen Automaten, der mithilfe der Forward-Bisimulation erzeugt wurde, hat jedoch die wichtige Eigenschaft, mit dem deterministischen Minimalautomaten isomorph zu sein (siehe [HMM07]). Wir nutzen daher Forward-Bisimulation für Äquivalenztests von Baumautomaten. Wir wollen die für uns wichtigsten Ergebnisse aus [HMM07] vorstellen und erläutern, wie die Verfahren in der vorliegenden Implementation benutzt wurden.
Nicht jeder Aggregations-Automat einer beliebigen Äquivalenzrelation über den Zuständen eines Automaten erkennt dieselbe Sprache wie der ursprüngliche Automat. Die beiden erwähnten Bisimulationsvarianten besitzen jedoch diese Eigenschaft.

5.6.1 Backward-Bisimulation

Das Prinzip der Berechnung einer Backward-Bisimulation lässt sich vereinfacht so zusammenfassen: Bei einem Vergleich zweier Zustände q, q' eines Baumautomaten wird untersucht, welche Automatenregeln $f(q_1, ..., q_n) \to q$ vorkommen. Die Zustände $q_1, ..., q_n$ werden dann mit den Zuständen $q'_1, ..., q'_n$ aus Regeln der Art $f(q'_1, ..., q'_n) \to q'$ verglichen. Wir schauen also "zurück" von einem Zustand q auf die Reduktionen, die zu q führen.

Definition 25. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln. Eine Äquivalenzrelation $\sim \subseteq Q \times Q$ ist eine **Backward-Bisimulationsäquivalenz**, wenn für alle q, q' mit $q \sim q'$ und alle Funktionssymbole $f \in \Sigma$ gilt: Existiert eine Regel

$$f(q_1, ..., q_n) \to q \in \delta,$$

so existiert auch eine Regel

$$f(q'_1, ..., q'_n) \to q' \in \delta,$$

mit $q_i \sim q'_i$ für alle $i \leq n$. □

Sind zwei Zustände backward-bisimulationsäquivalent, so gilt insbesondere:

Satz 10. Sei $\sim$ eine Bisimulationsäquivalenz auf den Zuständen eines Baumautomaten $\mathcal{A}$. Dann gilt für zwei Zustände q, q' aus $\mathcal{A}$: Wenn $q \sim q'$, so ist $\mathcal{L}(q) = \mathcal{L}(q')$. Die beiden Zustände erkennen also dieselbe Termmenge. □

Zustände, die zueinander backward-bisimulationsäquivalent sind, können daher zu einem Zustand zusammengefasst werden.
Wir erwähnen noch ohne Beweis:

Satz 11. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln.
Für $\mathcal{A}$ existiert eine **gröbste** Backward-Bisimulation $\sim_{max} \subseteq Q \times Q$, d.h. für jede Backward-Bisimulation $\sim \subseteq Q \times Q$ gilt: $\sim \subseteq \sim_{max}$. □

Die gröbste Backward-Bisimulation eines Automaten besitzt am wenigsten Äquivalenzklassen von allen Backward-Bisimulationen. Der aus dieser Relation berechnete aggregierte Automat hat also weniger Zustände als die aggregierten Automaten der anderen Backward-Bisimulationen.
Ein naiver Algorithmus zum Berechnen von $\sim_{max}$ ähnelt dem einfachen Minimierungsverfahren deterministischer Wortautomaten durch Partitionsverfeinerung (siehe beispielsweise [HU79]): Man startet mit einer anfänglichen Partition, die nur eine Klasse mit allen möglichen Zuständen enthält. Mithilfe des Kriteriums aus Def. 25 werden nun je zwei Zustände q, q', die sich in einer Klasse der Partition befinden, verglichen: Existiert für eine Regel $f(q_1, ..., q_n) \rightarrow q$ keine entsprechende Regel $f(q'_1, ..., q'_n) \rightarrow q'$, so dass sich die Zustände q_i, q'_i für alle $i \leq n$ jeweils in der gleichen Klasse der Partition befinden, so gehören beide Zustände nicht mehr derselben Klasse an und werden getrennt. Diese Überprüfungen werden für alle Zustandspaare q, q' solange wiederholt, bis sich keine Änderung der Partition mehr ergibt. Die Laufzeit des Algorithmus kann mit $\mathcal{O}(|Q|^2 \cdot |\delta|)$ abgeschätzt werden, wobei wir hierbei die Aritäten der Funktionssymbole nicht berücksichtigen. In [HMM07] wird ein effizienterer Algorithmus vorgestellt, der auf dem Partition-Refining-Algorithmus aus [PT87] beruht. Er wurde in der vorliegenden Implementation im Package `treeautomaton.minimisation.backward` als Klasse `BisiPartitions` realisiert. Grundgedanke des Algorithmus ist es, nicht pro Iterationsschritt die Regeln eines jeden Zustands mit denen eines jeden anderen zu vergleichen wie bei dem oben vorgestellten Ansatz. Vielmehr verwaltet der Algorithmus zwei Äquivalenzrelationen $\mathcal{P}, \mathcal{R} \subseteq Q \times Q$, die im Laufe des Algorithmus verfeinert werden. Wir wollen diese veränderten Relationen mit $\mathcal{P}_0 \supseteq \mathcal{P}_1 \supseteq ...$ und $\mathcal{R}_0 \supseteq \mathcal{R}_1 \supseteq ...$ bezeichnen. $\mathcal{R}_i$ stellt zu jedem Zeitpunkt i eine (nicht unbedingt echte) Verfeinerung von $\mathcal{P}_i$ dar, es gilt also $\mathcal{P}_i \subseteq \mathcal{R}_i$ für alle $i \geq 0$.

Definition 26. Es sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln. Eine Äquivalenzrelation $\sim \subseteq Q \times Q$ heißt **backwardstabil bezüglich einer Menge** $M \subseteq Q$, wenn für alle Zustände $q, q' \in M$ und alle Funktionssymbole $f \in \Sigma$ Folgendes gilt: Existiert eine Regel

$$f(q_1, ..., q_n) \rightarrow q \in \delta,$$

so existiert auch eine Regel

$$f(q'_1, ..., q'_n) \rightarrow q' \in \delta,$$

mit $q_i \sim q'_i$ für alle $i \leq n$. Weiterhin heißt $\sim$ **backwardstabil bezüglich einer Äquivalenzrelation** $\approx \subseteq Q \times Q$, wenn $\sim$ backwardstabil bezüglich jeder Klasse $[q]_\approx$, $q \in Q$, von $\approx$ ist. □

Ist eine Äquivalenzrelation zu sich selbst backwardstabil, so stellt sie eine Backward-Bisimulationsäquivalenz dar.

Die Idee des Algorithmus besteht darin, die Relationen von $\mathcal{R}_i$ bezüglich der gröberen Relationen $\mathcal{P}_i$ durch Verfeinerung backwardstabil zu machen. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln. Der Algorithmus startet mit $\mathcal{P}_0 = Q \times Q$ und der Relation $\mathcal{R}_0$ als gröbster Äquivalenzrelation, die zu Q stabil ist, d.h., aus $Q \times Q$ werden alle Zustandstupel (q, q') entfernt, die gegen die Bedingung aus Def. 26 bezüglich der Menge Q verstoßen. $\mathcal{R}_0$ stellt eine Verfeinerung von $\mathcal{P}_0$ dar. Solange $\mathcal{P}_i$ und $\mathcal{R}_i$ nicht dieselbe Relation darstellen, existiert immer mindestens eine Klasse $[q]_{\mathcal{R}_i} \subset [q]_{\mathcal{P}_i}$. Nehmen wir, dass gilt:

$$[q]_{\mathcal{P}_i} = \dot{\bigcup}_{i=1}^{n} [q_i]_{\mathcal{R}_i},$$

so wählt der Algorithmus hieraus ein $k \leq n$ mit $|[q_k]_{\mathcal{R}_i}| \leq |[q]_{\mathcal{P}_i}|/2$, also eine Klasse, die weniger als die Hälfte der Zustände von $[q]_{\mathcal{P}_i}$enthält, und verfeinert dadurch $\mathcal{P}_i$ zu $\mathcal{P}_{i+1}$: Die Klasse $[q]_{\mathcal{P}_i}$ wird in $\mathcal{P}_{i+1}$ durch die zwei Klassen $K_1 = [q_k]_{\mathcal{R}_i}$ und $K_2 = [q]_{\mathcal{P}_i} \setminus [q_k]_{\mathcal{R}_i}$ ersetzt. $\mathcal{R}_i$ war bereits backwardstabil bezüglich $\mathcal{P}_i$. Daher muss, um $\mathcal{R}_i$ backwardstabil bezüglich $\mathcal{P}_{i+1}$ zu machen, lediglich die Stabilität von $\mathcal{R}_i$ bezüglich K_1 und K_2 sichergestellt werden. Hier kommt die richtige Wahl von k ins Spiel: Wir können eine Verfeinerung von $\mathcal{R}_i$, die Relation $\mathcal{R}_{i+1}$, die bezüglich K_1 und K_2 stabil ist, in $\mathcal{O}(\rho \cdot |\delta(K_1)|)$ berechnen. Hierbei ist ρ die maximal vorkommende Arität in Σ und unter $\delta(K_1)$ wollen wir die Regeln in $\mathcal{A}$ verstehen, bei denen ein Zustand aus K_1 vorkommt. Die Verfeinerung bezüglich K_2 hängt nur von der Größe von K_1 ab, nicht jedoch von der Größe von K_2. Fallen am Ende des Algorithmus zum Zeitpunkt $n \geq 0$ dann $\mathcal{R}_n$ und $\mathcal{P}_n$ zusammen, ist mit $\mathcal{R}_n$ eine Bisimulationsäquivalenz gegeben. Durch die geschickte Wahl von k gilt für K_1 stets: $|K_1| \leq |(K_1 + K_2)|/2$. Daher kann ein Zustand maximal $\log |Q|$-mal in einer der K_1-Klassen enthalten sein, was nach genauerer Analyse des Algorithmus die Laufzeit $\mathcal{O}(\rho^2 \cdot |\delta| \cdot \log |Q|)$ ergibt (siehe [HMM07]).

Wichtig für die Effizienz des Algorithmus ist das Verfeinern der Relation $\mathcal{R}_i$ zu $\mathcal{R}_{i+1}$. $\mathcal{R}_{i+1}$ muss stabil sein bezüglich $\mathcal{P}_{i+1}$. Da sie schon stabil bezüglich $\mathcal{P}_i$ war, reicht es, die Stabilität bezüglich der beiden "neuen" Klassen K_1 und K_2 in $\mathcal{P}_{i+1}$ sicherzustellen. Hierbei gilt, dass $K_1 \cup K_2$ eine Klasse aus $\mathcal{P}_i$ darstellen, K_1 eine Klasse aus $\mathcal{R}_i$. Das Verfahren benutzt Hash-Tabellen und errechnet für jeden Zustand q aus Q einen Hashwert, basierend auf den Regeln aus $\delta(K_1)$ und Informationen aus früheren Iterationsschritten. Der Hashwert stellt ein Array v_q von Integer-Werten dar; für jede Regel $f(q_1, ..., q_n) \to q$ aus $\delta(K_1)$ wird eine Komponente von v_q inkrementiert. Die richtige Komponente wird durch die Klassenzugehörigkeiten der Zustände q_i ermittelt. Zustände, die denselben Hashwert besitzen, gehören dann in dieselbe Klasse. Wir wollen das Vorgehen an dieser Stelle nicht detaillierter behandeln und verweisen auf [HMM07].

Implementierung der Backward-Bisimulation

Zu Vergleichszwecken wurde sowohl der naive Ansatz als auch das Verfahren von [HMM07] implementiert. Letzteres benutzt einen Hash mit beliebiger Kapazität und perfekter Hashfunktion für das Hashen von Zuständen. Ein solcher konnte in der vorliegenden Implementierung nur durch einen gewöhnlichen Hash angenähert werden. Ebenso musste für jeden Zustand eine Hash-Tabelle angelegt werden, um Informationen über die untersuchten Regeln zu speichern. Das Verfahren benutzt in seiner ursprünglichen Form für diese Informationen ein Array von Integers für jeden Zustand q, das wir im vorigen Abschnitt mit v_q bezeichnet haben. Die Dimension von v_q kann in der Größenordnung von $|Q|^k$ liegen (z.B. wenn jede Klasse der Bisimulationsäquivalenz nur einen Zustand enthält), wobei k die maximale Arität der verwendeten Signatur darstellt. Dieses Array ist meist nur dünn besetzt, insgesamt gibt es nur maximal soviel Einträge $\neq 0$ wie Regeln im Baumautomaten. Da die Zustandszahl der Baumautomaten sehr groß werden kann, erscheint daher statt des Arrays eine Hash-Tabelle eine geeignete Darstellungsform zu sein. Jeder Zustand verwaltet diesen Hash in seinem Attribut `sobshash`. Ebenso enthält jeder Zustand die Attribute `pblock` und `rblock`, die Referenzen auf die Klassen von $\mathcal{P}_i$ und $\mathcal{R}_i$ darstellen, in denen sich der Zustand befindet.
Es ist möglich, den Algorithmus mit und ohne Backward-Bisimulation auszuführen. Ist sie aktiviert, wird jeder Automat $\mathcal{A}_i$ mithilfe der Backward-Bisimulationsäquivalenz reduziert. Danach ist es nötig, die `AutomatonCopy`-Instanzen in den symbolischen Regeln anzupassen: Die Referenz `fixed` muss nun durch seinen Repräsentanten im reduzierten Automaten ersetzt werden. Da bei der Backward-Bisimulationsäquivalenz die Zustände im aggregierten Automaten dieselbe Termmenge erkennen wie alle Zustände, deren Klasse sie repräsentieren, ist die Funktionalität des Algorithmus auch nach Einsetzen dieser Reduktionsmethode garantiert. Es wurden Experimente durchgeführt, inwiefern dieser Reduktionsalgorithmus die Laufzeit und das Speicherverhalten des Algorithmus verbessern kann, siehe dazu 6.3.1.

5.6.2 Forward-Bisimulation

Bei der **Forward-Bisimulation** vergleicht man die Automatenregeln der Form $f(q_1, ..., q_{i-1}, q, q_{i+1}, ..., q_n) \rightarrow q_{n+1}$ mit Regeln, die die Form $f(q'_1, ..., q'_{i-1}, q', q_{i+1}, ..., q'_n) \rightarrow q'_{n+1}$ haben. Hierbei blicken wir also gewissermaßen von einem Zustand aus in seine Zukunft.

Definition 27. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein Baumautomat ohne ϵ-Regeln. Eine Äquivalenzrelation $\sim \subseteq Q \times Q$ ist eine **Forward-Bisimulationsäquivalenz**, wenn für alle q, q' mit $q \sim q'$ und alle Funktionssymbole $f \in \Sigma$ gilt:

- $q \in Q_f \Leftrightarrow q' \in Q_f$,
- Existiert eine Regel

$$f(q_1, ..., q_{i-1}, q_i = q, q_{i+1}, ..., q_n) \rightarrow q_{n+1} \in \delta,$$

so existiert auch eine Regel

$$f(q'_1, ..., q'_{i-1}, q'_i = q', q'_{i+1}, ..., q'_n) \to q'_{n+1} \in \delta,$$

mit $q_j \sim q'_j$ für alle $j \leq n+1$. □

Da bei dieser Bisimulationsvariante nicht sichergestellt ist, dass zwei äquivalente Zustände auch dieselbe Termmenge erkennen, wird sie nicht für die Reduzierung der Automaten $\mathcal{A}_i$ verwendet. Es existiert ähnlich der Backward-Variante eine gröbste Forward-Bisimulationsäquivalenz für jeden Baumautomaten. Außerdem gilt folgende Aussage (für einen Beweis siehe [HMM07]):

Satz 12. Sei $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ ein deterministischer und vollständiger Baumautomat, und sei $\sim \subseteq Q \times Q$ die gröbste Forward-Bisimulation von $\mathcal{A}$. Dann gilt: Der Aggregationsautomat $\mathcal{A}_\sim$ von $\mathcal{A}$ ist der deterministische Minimalautomat von $\mathcal{A}$. □

Wir benutzen daher die Forward-Bisimulation zur Minimierung der deterministischen Automaten in den `copies`-Listen. Es gibt eine Version des Algorithmus für die Backward-Bisimulation auch für den Forward-Fall. Da wir Forward-Bisimulationen jedoch nur zur Minimierung von deterministischen Automaten einsetzen und für die effiziente Variante des Algorithmus ein vollständiger Automat vorliegen muss, der unter Umständen bedeutend mehr Speicherplatz belegt als ein unvollständiger, beschränken wir uns auf die naive Implementierung der Forward-Bisimulation, die wir auch einfach mit unvollständigen Automaten durchführen können; siehe dazu den folgenden Abschnitt.

5.6.3 Minimierung deterministischer Baumautomaten

Eigentlich ist für die Minimierung eines deterministischen Automaten $\mathcal{A} = (Q, \Sigma, Q_f, \delta)$ notwendig, dass er in vollständiger Form vorliegt. Das ist leicht zu bewerkstelligen, indem man dem Automaten einen nichtakzeptierenden *Garbage*-Zustand q_G und die Regelmenge

$$\{f(q_1, .., q_n) \to q_G \mid f \in \Sigma^n, q_1, ..., q_n \in Q \cup \{q_G\}, \nexists f(q_1, ..., q_n) \to q \in \delta\}$$

hinzufügt. Eine solche Aufblähung des Automaten möchte man meist vermeiden. Wir nehmen daher für deterministische Automaten einen *implizit* vorhandenen Garbage-Zustand q_G an und fügen keine zusätzlichen Regeln ein. Gibt es keine Regel für ein Tupel von Zuständen und ein Funktionssymbol, so wird nach q_G abgeleitet. Bei der Minimierung muss der Garbage-Zustand ebenso berücksichtigt werden: Ist ein Zustand q noch nicht als inäquivalent zu q_G erkannt, so muss jede Regel $r = f(q_1, ..., q, ..., q_n) \to q'$ daraufhin überprüft werden, ob q' nicht äquivalent mit q_G ist. Wir wissen, dass die Regel $f(q_1, ..., q_G, ..., q_n) \to q_G$ implizit in δ enthalten ist. Sind q' und q_G nicht äquivalent, so wissen wir, dass q und q_G ebenfalls nicht äquivalent sind.

5.7 Generierung von Gegenbeispielen

Beim Testen funktionaler Programme wird oft so vorgegangen, dass eine Testmenge G von Termen mit der Nachfolgermenge einer Termmenge M bezüglich eines Regelsystems $Post^*(M)$ geschnitten wird (siehe Abschnitt 3.2). Ist der Schnitt nicht leer, so könnte das bedeuten, dass die Terme im Schnitt eine bestimmte Eigenschaft einer gegebenen Spezifikation nicht erfüllen. Besonders interessant wäre es, von einem Element $t \in (G \cap Post^*(M))$ zu wissen, welches Element $t' \in M$ zu t abgeleitet werden konnte und welche Regeln dazu angewendet wurden, also eine Rekonstruktion der Ableitungsfolge. Wir wollen solche Terme t gemeinsam mit einer Ableitungsfolge informell als **Gegenbeispiele** bezeichnen und beschreiben, wie in der vorliegenden Implementation aus dem Automaten für $Post^*(M)$ Ableitungsfolgen rekonstruiert werden können. In der folgenden Darstellung erläutern wir das Verfahren unter Verwendung von ϵ-Regeln, um ein gewisses Maß an Anschaulichkeit zu erreichen: Wie bereits erwähnt wurden bei der Implementation des Prototyps keine ϵ-Regeln verwendet. Jedoch ließen sich die im Folgenden dargestellten Erkenntnisse auch auf die Implementierung übertragen, wie erläutert werden wird.

5.7.1 Annotation der Baumregeln

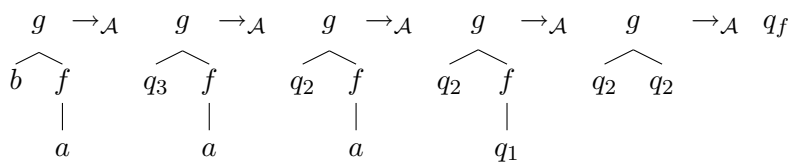
Der Algorithmus aus Abb. 4.4 fügt Zustände und Regeln in den Automaten $\mathcal{A}_0$ ein, von dessen Sprache $Post^*$ berechnet werden soll. Diese simulieren die Anwendung einer symbolischen Regel. Daher liegt nahe, eingefügte Regeln mit Zusatzinformationen zu versehen, durch die ersichtlich wird, aufgrund welcher symbolischen Regel sie jeweils eingefügt wurden. Beim Ableiten eines Terms t können diese Informationen genutzt werden, um eine mögliche Ableitungsfolge im Regelsystem zu finden, die ein Element des ursprünglichen Baumautomaten als Anfangsterm und t als Endterm hat.
Wir bemerken, dass nicht jede neu eingefügte Regel annotiert werden muss: Lediglich die ϵ-Regeln, die in Zeile 28 bzw. 32 des Algorithmus in Abb. 4.4 eingeführt werden, benötigen eine neue Beschriftung. Bei der Implementation wurden die Regeln beschriftet, die beim Simulieren von ϵ-Regeln erzeugt wurden. Sie haben insbesondere den Zielzustand der ϵ-Regel als Zielzustand. Die weiteren Ausführungen bezüglich Annotationen lassen sich auch auf diese Regeln übertragen.
Wir wollen im Folgenden den vom Algorithmus in Abb. 4.4 erzeugten Automaten mit $\mathcal{A}_{post}$ bezeichnen, die während dem Algorithmus erzeugten "Zwischenautomaten" wie schon im vorherigen Kapitel mit $\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n = \mathcal{A}_{post}$. Wir wollen ebenso vom Zeitpunkt i einer Ausführung des Algorithmus sprechen, wenn der Automat $\mathcal{A}_{i-1}$ im Laufe des Algorithmus bereits erzeugt wurde, jedoch noch nicht $\mathcal{A}_i$. Eine zum Zeitpunkt i eingefügte Regel ist noch nicht Bestandteil des Automaten $\mathcal{A}_{i-1}$, jedoch im Automat $\mathcal{A}_i$ vorhanden. Zum Zeitpunkt 0 eingefügte Regeln sind die Regeln von $\mathcal{A}_0$. Jede zum Zeitpunkt i eingefügte ϵ-Regel wird mit einem Tupel (i, r) beschriftet, wobei r die symbolische Regel r bezeichnet, aufgrund derer die ϵ-Regel erstellt wurde. Wir neh-

men weiterhin an, dass jede in einem Automaten $\mathcal{A}_i$ vorkommende ϵ-Regel nur durch Zeile 28 bzw. 32 des Algorithmus hinzugefügt wurde. Dies bedeutet insbesondere, dass die Automaten $\mathcal{A}_0$ und $\mathcal{A}_{r,q,i}$ ϵ-Regel-frei sind (siehe Abschnitt 4.4).
Wir fixieren nun für einen Term $t \in \mathcal{L}(\mathcal{A}_{post})$ eine mögliche Reduktion in einen akzeptierenden Zustand von $\mathcal{A}_{post}$. Wir können uns die Reduktion anhand des Terms graphisch veranschaulichen, indem wir ihn als Baum darstellen und die einzelnen Knoten mit den Zuständen versehen, in die sie während der Reduktion abgeleitet werden. Sind bei der Reduktion ϵ-Regeln beteiligt (bzw. die Regeln, durch die wir sie ersetzt haben), so werden wir diese ebenso als Annotation an den entsprechenden Knoten des Terms anbringen, und zwar in der Reihenfolge ihres Auftretens in der Reduktion.
Wir wollen dies anhand eines Beispiels verdeutlichen:

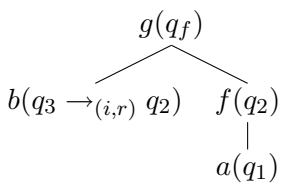
Beispiel 16. Wir betrachten einen Baumautomat $\mathcal{A}_{post} = (Q, \Sigma, Q_f, \delta)$ mit $Q = \{q_1, q_2, q_3, q_f\}$, $\Sigma = \{a(), b(), f(\cdot), g(\cdot,\cdot)\}$ und $Q_f = \{q_f\}$. δ enthalte die Regeln

- $a \to q_1$
- $f(q_1) \to q_2$
- $b \to q_3$
- $g(q_2, q_2) \to q_f$
- $q_3 \to q_2$

Wir wollen annehmen, dass die Regel $q_3 \to q_2$ zum Zeitpunkt i wegen Anwendung der symbolischen Regel r in den Automaten eingefügt wurde. Der Term $t = g(b, f(a))$ wird von $\mathcal{A}_{post}$ erkannt. Eine mögliche Reduktion ρ nach q_f ist gegeben durch



Wir benutzen die Reduktionsschritte, um einen annotierten Term t zu erstellen:



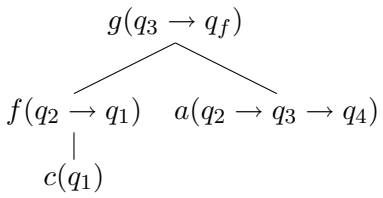
Jedes vorkommende Funktionssymbol wird mit den Zuständen annotiert, in die sie während der Reduktion abgeleitet wurden. Durch das Auftreten von ϵ-Regeln müssen auch diese in der Reihenfolge ihres Auftretens innerhalb der

Reduktion notiert werden. Im Beispiel finden wir am Knoten b die Information $q_2 \rightarrow_{(i,r)} q_3$, die auf die verwendete ϵ-Regel $q_3 \rightarrow q_2$ rückschließen lässt und gleichzeitig die Information enthält, dass diese zum Zeitpunkt i wegen der symbolischen Regel r dem Automaten hinzugefügt wurde. $\square$

Im weiteren Verlauf dieses Kapitels sprechen wir von einem bezüglich einer (festen) Reduktion ρ **annotierten** Term t, wenn er entsprechend Beispiel 16 mit Zusatzinformationen passend zur Reduktion ρ versehen wurde. Für eine bessere Übersicht bei der Darstellung werden wir oftmals Zeitpunkte und symbolischen Regeln (i, r) der Annotation weglassen. Ein Knoten kann als Annotationen einen einzelnen Zustand q oder eine Folge $q_1 \rightarrow q_2 ... \rightarrow q_n$ von Zuständen haben. Wir sprechen im zweiten Fall von dem Zustandspaar $q_i \rightarrow q_{i+1}$ als dem i-ten Element der Annotation. Dieses entspricht einer ϵ-Regel im Baumautomaten, mit der wir das Zustandspaar im weiteren Verlauf auch gleichsetzen möchten. Im ersten Fall hat die Annotation nur ein erstes Element q bzw. (q).

5.7.2 Rekonstruktion von Ableitungsfolgen

Wir wollen zunächst den Begriff einer **minimalen** ϵ-Regel eines annotierten Terms t einführen: Eine ϵ-Regel $r = q \rightarrow q'$, die als erstes Element der Beschriftung der Wurzel des Terms $t|_p$ vorkommt, und für die gilt, dass kein Knoten in $t|_p$ außer der Wurzel mit einer ϵ-Regel annotiert wurde, ist minimal bei p in t. Zum Beispiel sind im folgenden annotierten Term



die ϵ-Regeln $q_2 \rightarrow q_3$ beim a-Knoten (Position 2) sowie $q_2 \rightarrow q_1$ beim f-Knoten (Position 2) minimal, jedoch nicht die ϵ-Regel $q_3 \rightarrow q_f$ bei g (Position ϵ) oder die zweite ϵ-Regel $q_3 \rightarrow q_4$ bei b. Befindet sich bei Position p eine minimale ϵ-Regel, die durch die symbolische Regel r hinzugefügt wurde, so gilt, dass $t|_p$ eine entfaltete rechte Seite von r darstellt.
Um aus einem wie oben beschriebenen annotierten Term t eine gültige Ableitungsfolge im Grundtermersetzungssystem R zu erstellen, ist folgende Beobachtung wichtig: Kommt bei der Reduktion keine ϵ-Regel vor, die nicht schon im Automaten $\mathcal{A}_0$ vorhanden ist, so gilt, dass $t \in \mathcal{L}(\mathcal{A}_0)$. Gibt es jedoch eine einzige ϵ-Regel $q_1 \rightarrow q_2$, die in der Reduktion verwendet und zuerst in einem Automaten $\mathcal{A}_i$ mit $i \geq 1$ vorhanden ist, so bedeutet dies, dass wir aus dem Term t einen Vorgängerterm t' erzeugen können, für den $t' \in \mathcal{L}(\mathcal{A}_{i-1})$ und $t' \hookrightarrow_R t$ gilt. Dies lässt sich mithilfe des Minimalitäts-Begriffs von ϵ-Regeln auf Reduktionen mit mehr als einer ϵ-Regel übertragen. Abb. 5.3 verdeutlicht das Vorgehen: An der Position p des Terms t wird u.a. die ϵ-Regel $q_1 \rightarrow q_2$ verwendet. Nehmen wir an, dass diese zum Zeitpunkt i aufgrund der symbolischen

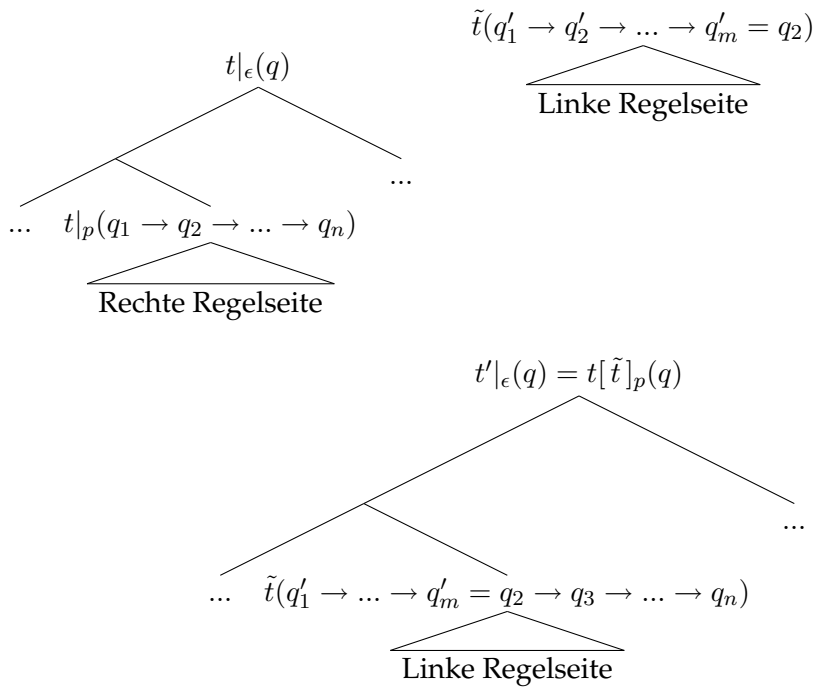


Abbildung 5.3: Rückwärtsersetzung.

Regel $r = (t_l, t_r)$ erzeugt wurde und sie minimal bei p ist. Dann stellt $t|_p$ eine entfaltete Regel einer rechten Seite r dar. Wir ersetzen daher $t|_p$ durch die entsprechende linke Seite der entfalteten Regel $\tilde{t}$, und es gilt $t[\,\tilde{t}\,]_p \hookrightarrow_R t$.

Wir werden das Vorgehen, aus t den Term $t[\tilde{t}]_p$ zu erzeugen, eine **Rückwärtsersetzung** nennen.

Eine Schwierigkeit besteht darin, dass wir aus der Struktur der symbolischen Regel r und des Terms t die entfaltete Regel rekonstruieren müssen, die für t' angewandt wurde, um daraus $\tilde{t}$ zu erhalten. Dazu müssen wir die jeweiligen Terme aus $t|_p$ extrahieren, die für die Variablen eingesetzt wurden. Der Algorithmus UNIFY in Abb. 5.4 zeigt auf, wie aus einem Grundterm g und einem linearen Term mit Variablen s die Menge der Teilterme von g errechnet werden kann, durch die die jeweiligen Variablen substituiert wurden: Wir betrachten beide Terme als Bäume und steigen "parallel" bei beiden von der Wurzel an abwärts, bis wir bei s auf eine Variable x stoßen; der entsprechend untersuchte Teilterm bei g wird an x gebunden. Da die Variable kein zweites Mal in s vorkommen kann, ist die Bindung korrekt, falls g auch wirklich eine Instantiierung von s darstellt. Wenn g keine Instanz von s ist, bricht der Algorithmus mit einer Fehlermeldung ab. Ansonsten gibt er eine Menge von Tupeln (x, t_x) zurück, wobei t_x die Substitution der Variablen x darstellt. Mithilfe von UNIFY$(t|_p, t_r)$ erhalten wie die für uns notwendigen Instanzen der Variablen und können die entsprechende Instanz des Term t_r erstellen, die wir mit $\tilde{t}$ bezeichnet haben.

```
UNIFY(Grundterm g, linearer Term s)
 1  if s ist Variable x
 2     then
 3           return {(x, g)}
 4  elseif s ist Konstante und g ist Konstante
 5     then
 6           return ∅
 7  elseif s = f(s_1, ..., s_n) und g = f(t_1, ..., t_n)
 8     then
 9           result := ∅
10           for i := 1.. n
11              do
12                 result := result ∪ UNIFY(t_i, s_i)
13           return result
14     else
15           Gib Fehler "g ist keine Instanz von s" aus und beende.
```

Abbildung 5.4: Algorithmus UNIFY; für Erläuterungen siehe Text.

Aufgrund der Arbeitsweise des Algorithmus aus Abb. 4.4 wissen wir, dass der neue Term $\tilde{t}$ ebenso wie $t|_p$ nach q_2 reduziert werden kann. Wir fixieren nun eine solche Reduktion $\tilde{t} \rightarrow^*_{\mathcal{A}_{post}} q_2$ und annotieren $\tilde{t}$ entsprechend dieser Reduktion wie in Beispiel 16 dargestellt. Wir übertragen die Annotation auf den Term $t' = t[\tilde{t}]_p$. Alle Annotationen von t' bis auf jene im Unterterm $t'|_p$ bzw. $\tilde{t}$ werden dazu von t übernommen. Enthielt die Annotation von t an der Position p außer $q_1 \rightarrow q_2$ noch weitere ϵ-Regeln $r_1, ..., r_n$, so müssen diese auch nach t' übernommen werden. In Abb. 5.3 besteht die Annotation von $\tilde{t}$ aus der Folge $q'_1 \rightarrow q'_2 \rightarrow ... \rightarrow q'_m = q_2$ von ϵ-Regeln.
War $q_1 \rightarrow q_2$ die einzige ϵ-Regel in der Annotation von $t|_p$, so wird $t'|_p$ nur mit $q'_1 \rightarrow q'_2 \rightarrow ... \rightarrow q'_m = q_2$ annotiert. Es ist in diesem Falle auch möglich, dass $\tilde{t}$ mit dem einzelnen Zustand (q_2) annotiert ist; dann wird $t'|_p$ nur mit (q_2) annotiert.
Gibt es mehrere ϵ-Regeln in der Annotation von $t|_p$, wird $q_1 \rightarrow q_2$ aus der Annotation entfernt und die restliche Folge $q_2 \rightarrow q_3 \rightarrow ... \rightarrow q_n$ mit der Regelfolge von $\tilde{t}$ zu $q'_1 \rightarrow q'_2 \rightarrow ... \rightarrow q'_m = q_2 \rightarrow q_3 \rightarrow ... \rightarrow q_n$ kombiniert. Diese Folge stellt dann die neue Annotation von $t'|_p$ dar. Insgesamt erhalten wir so mit t' wieder einen Term, der bezüglich einer Reduktion für $t' \rightarrow^*_{\mathcal{A}_{post}} q$ korrekt annotiert ist.
Wir sind daran interessiert, mit wiederholtem Anwenden dieses Verfahrens zu einem Term zu gelangen, der bereits von $\mathcal{A}_0$ erkannt wird. Durch Abspeichern der dazu durchgeführten Termersetzungen kann die Ableitungsfolge rekonstruiert werden.
Es ist im Allgemeinen nicht ausreichend, willkürlich mithilfe der Annotationen ϵ-Regeln auszuwählen und die entsprechenden Terme durch ihre

Vorgängerterme zu ersetzen; diese Vorgehensweise kann zu einer unendlichen Berechnung führen.

Beispiel 17. Wir betrachten drei symbolische Regeln eines symbolischen Grundtermersetzungssystems R über $\Sigma = \{a(), f(\cdot), g(\cdot), h(\cdot)\}$,

$$g(x) \to f(x), h(x) \to f(g(x)) \text{ und } f(f(x)) \to g(x),$$

wobei $x \in \{a\}$. Wir berechnen hierfür $Post_R^*(\{h(a)\})$. Unter anderem enthält diese Menge den Term $f(g(a))$. Für diesen existiert eine unendliche Rekonstruktion:

$$f(g(a)) \hookleftarrow_R f(f(f(a))) \hookleftarrow_R f(f(g(a))) \hookleftarrow_R f(f(f(f((a))))) \hookleftarrow_R ...,$$

also eine Ableitungsfolge, die nie zu einem Element aus der ursprünglichen Termmenge führt. Es existiert jedoch ebenso die endliche Rekonstruktion

$$f(g(a)) \hookleftarrow_R h(a).$$

Beide Rekonstruktionen können mit dem zuvor beschriebenen Verfahren abgeleitet werden. □

Um solche endlosen Berechnungen zu vermeiden, ist folgende Beobachtung hilfreich:

Satz 13. Sei t ein annotierter Term, und $q \to q'$ eine minimale ϵ-Regel von t bei Position p, die zum Zeitpunkt i aufgrund der symbolischen Regel $r = (t_l, t_r)$ eingefügt wurde. Dann existiert ein Term $\tilde{t}$, so dass $\tilde{t} \to t|_p$ eine entfaltete Regel von r darstellt. Weiterhin gibt es eine Reduktion der Art $\tilde{t} \to_{\mathcal{A}_{post}} ... \to_{\mathcal{A}_{post}} q'$, in der nur ϵ-Regeln vorkommen, die zu einem Zeitpunkt $j < i$ hinzugefügt wurden. □

Beweis. Die Existenz von $\tilde{t}$ als linker Seite einer entfalteten symbolischen Regel r ist durch die obige Konstruktion sichergestellt. $q \to q'$ wurde zum Zeitpunkt i eingefügt, und es gilt $t|_p \to^*_{\mathcal{A}_i} q'$. Da $t|_p$ eine rechte Seite einer entfalteten Regel $\tilde{t} \to t|_p$ darstellt, deren Ableitung nach q' durch Einfügen der ϵ-Regel $q \to q'$ ermöglicht wurde, muss zuvor auch die linke Seite von q' erkannt worden sein: $\tilde{t} \to^*_{\mathcal{A}_{i-1}} q'$. Daher existiert eine Reduktion für $\tilde{t}$ nach q' in $\mathcal{A}_{i-1}$, und alle ϵ-Regeln dieser Reduktion wurden zu einem Zeitpunkt $j \leq i-1$ eingefügt. □

Daher werden wir bei einer Rückwärtsersetzung von t nach $t' = t[\tilde{t}]_p$ darauf achten, für die Annotation von $t'|_p$ eine Reduktion auszuwählen, die selbst nur ϵ-Regeln enthält, die zu einem früheren Zeitpunkt als die ϵ-Regel $q_1 \to q_2$ eingefügt wurden.
Bei diesem Vorgehen kann nur endlich oft eine Rückwärtsersetzung angewandt werden: Angenommen, es gäbe eine unendliche Folge von Rückersetzungen. Wir stellen sie durch die Folge von annotierten Termen $t_0, t_1, t_2, ...$ dar. Bei jeder Rückersetzung wird eine ϵ-Regel eines Zeitpunktes i durch eine eventuell leere Menge von ϵ-Regeln älterer Zeitpunkte $j < i$ ersetzt. Wir

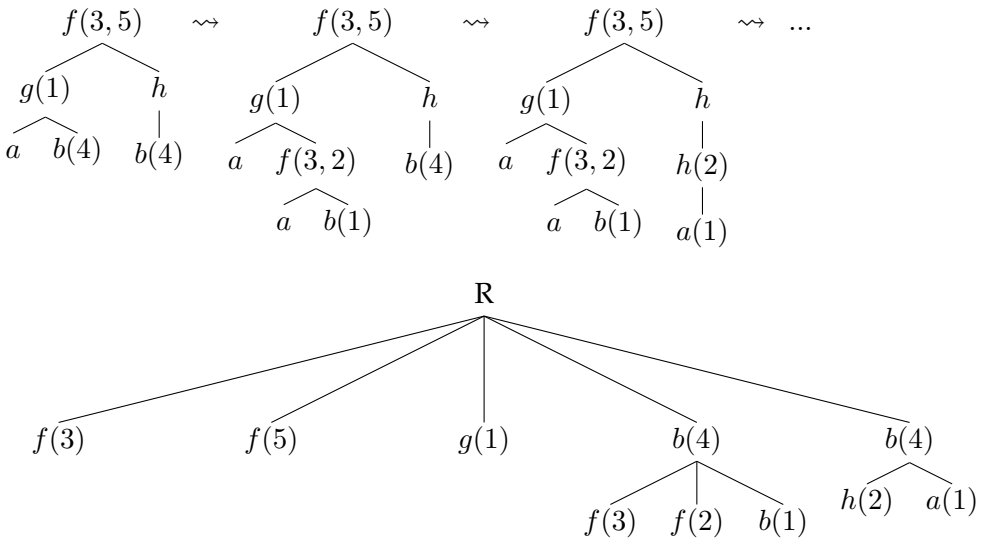


Abbildung 5.5: Beispiel zu endlosen Rückwärtsersetzungsfolgen. Die oberen Bäume stellen die annotierten Terme der Rückersetzungsfolge dar, wobei lediglich die Zeitpunkte des Einfügens der ϵ-Regel notiert wurden. Besitzt ein Knoten keine Annotation, so wurde während der Reduktion bei ihm keine ϵ-Regel angewandt.

können daher auf folgende Art einen Baum konstruieren: Von einem Wurzelknoten aus werden Nachfolgerknoten angelegt, die mit den Zeitpunkten der ϵ-Regeln von t_0 beschriftet werden, siehe dazu Abb. 5.5. Der Knoten der in t_0 ersetzten ϵ-Regel erhält dann ebenso Kinderknoten, die den in t_1 neu vorkommenden ϵ-Regeln entsprechen usw.
Der so entstehende Baum hat unendlich viele Knoten, sonst gäbe es nur endlich viele Rückersetzungen. Mithilfe des folgenden bekannten Satzes (siehe z.B. [BN98]) folgt daraus, dass es einen unendlichen Pfad im Baum geben muss. Jeder Kindknoten hat jedoch eine strikt kleinere Beschriftung als sein Elternknoten, so dass dies zum Widerspruch führt.

Satz 14. (Lemma von König) Ein gerichteter Baum mit endlichem Verzweigungsgrad hat genau dann unendlich viele Knoten, wenn er einen unendlichen Pfad enthält. □

Algorithmus GENERATECOUNTEREXAMPLE aus Abb. 5.6 erläutert schematisch den gesamten Vorgang der Gegenbeispielgenerierung. Er verwendet mehrere Funktionen und Prozeduren, deren Funktionsweise im Folgenden erläutert wird: Der Funktion FINDMINIMALEPSRULE wird ein annotierter Term t und ein Automat $\mathcal{A}$ übergeben. Sie liefert ein Tupel $(q \to q', i, r, p)$ zurück, so dass $q \to q'$ eine minimale ϵ-Regel in t an Position p ist, die zum Zeitpunkt i aufgrund der symbolischen Regel r in den Automat $\mathcal{A}$ eingefügt wurde. Die Prozedur ANNOTATE nimmt als Argumente einen Term t, eine Zustandsmenge M, ein $i \in \mathbb{N}$ und einen Automaten $\mathcal{A}$. Sie errechnet, soweit möglich, eine Reduktion ρ für $t \to^*_{\mathcal{A}} q$, wobei $q \in M$. Dabei dürfen nur ϵ-Regeln

vorkommen, die zum Zeitpunkt i oder früher zu $\mathcal{A}$ hinzugefügt wurden. Anschließend annotiert ANNOTATE den Term t entsprechend der Reduktion ρ. Der Prozeduraufruf INSTANTIATE(t, σ) erzeugt mithilfe einer von UNIFY erzeugten Tupelmenge σ die entsprechende Grundterminstanz des Terms t.

Aus den beiden zurückgegebenen Stacks von GENERATECOUNTEREXAMPLE kann die Ableitungsfolge des Gegenbeispiels rekonstruiert werden.

GENERATECOUNTEREXAMPLE(
Baumautomaten $\mathcal{A}_{post}$ und $\mathcal{A}_0$,
Symbolisches Grundtermersetzungssystem R,
Grundterm t)

```
 1  rulestack := ∅
 2  termstack := {t_0}
 3  procAnnotate(t_0, Q_f, ∞, A_post)
 4  j := 0
 5  t_j := t
 6  while t_j ∉ L(A_0)
 7        do
 8           (q → q', i, r = t_l → t_r, p) := FINDMINIMALEPSRULE(t_j, A_post)
 9           σ := UNIFY(t_r, t_j|_p)
10           t̃ := INSTANTIATE(t_l, σ)
11           ANNOTATE(t̃, {q'}, i − 1, A_post)
12           Entferne q → q' aus der Annotation von t_j an der Position p
13           Passe Annotation von t̃ an Position ϵ bezüglich t_j|_p an
14           PUSH(r, rulestack)
15           PUSH(t_j, termstack)
16           t_{j+1} := t_j[t̃]_p
17           j := j + 1
18  return (rulestack, termstack)
```

Abbildung 5.6: Algorithmus GENERATECOUNTEREXAMPLE; für Erläuterungen siehe Text. Zu beachten ist, dass die vorkommenden Terme annotiert sind, also Zusatzinformationen tragen.

5.8 Benutzung des Prototypen

Um den Algorithmus komfortabler an praktischen Beispielen austesten zu können, wurde ein Eingabeformat für funktionale Programme entwickelt. Mit dessen Hilfe sind sowohl die Bestandteile einfacher symbolischer Grundtermersetzungssysteme, also Sorten und Regeln, als auch Testmengen beschreibbar. Wir wollen in diesem Abschnitt die syntaktischen Bestandteile des Werkzeugs und seine Aufrufsparameter erläutern.

5.8.1 Syntax der Eingabedateien

Zur Veranschaulichung betrachten wir ein einfaches Beispiel, dessen symbolisches Grundtermersetzungssystem sich an 5.4.3 anlehnt. Wir erläutern die Syntax anhand der einzelnen Bestandteile des Beispiels:

```
// Symbole
symbols: a/0, b/0, c/2, nil/0, append/2;

// Sorten und Variablen
sorts: #const#0, #list#3;
sort #const = a(), b();
sort #list = nil(), c(#const, #list);
var xs is #list;
var ys is #list;
var x is #const;

// append-Regeln
rule append(nil(), :xs) => :xs;
rule append(c(:x, :xs), :ys) => c(:x, append(:xs, :ys));

// Spezifizierung der Testfälle
initterm: append(c(b(), :xs), nil());

// Spezifizierung der Testmenge
contains: c(a(), :xs) ?
```

Abbildung 5.7: Eingabedatei `append.rul`

Kommentare können im C++-Stil mit `//` eingeleitet werden; sie erstrecken sich dann über den Rest der Zeile.
Jede Eingabedatei beginnt mit der Definition einer Signatur in Form einer Liste. Sie wird eingeleitet durch das Schlüsselwort `symbols`, gefolgt von "`:`". Der Bezeichner eines Funktionssymbols kann eine beliebige Zeichenkette sein, bestehend aus Buchstaben und Zahlen, wobei zwischen Groß-und-Kleinschreibung unterschieden wird. Dies gilt auch für alle weiteren vorkommenden Bezeichner z.B. für Sorten oder Variablen. Zu jedem Funktionssymbol muss die passende Arität angegeben werden, was durch `/arity` geschieht. Beispielsweise definiert `append/2` ein 2-stelliges Funktionssymbol `append`. Abgeschlossen wird die Liste durch ein "`;`".
Als nächstes folgen Deklarationen für die Sorten, eingeleitet durch das Schlüsselwort `sorts` und ein ":". Sortenbezeichner werden mit einem "`#`" eingeleitet, gefolgt von beliebig vielen Buchstaben. In der Sortendeklaration muss außerdem für jede Sorte die maximale Tiefe seiner Terme angegeben werden. Dies geschieht durch Anhängen eines weiteren "`#`" und einer natürlichen Zahl. Durch diese Angabe ist sichergestellt, dass der Wertebereich der Sorten endlich bleibt. Auch hier muss mit einem "`;`" abgeschlossen werden.

Die Struktur der Sorten wird im folgenden Abschnitt festgelegt, den Sortendefinitionen. Eine Sortendefinition beginnt mit `sort` und dem Bezeichner der jeweiligen Sorte, gefolgt von einem "=". Nun kann eine Liste von Termen über der bereits definierten Signatur angegeben werden, in denen auch Sortenbezeichner vorkommen können. Abgeschlossen wird die Definition mit "`;`". Diese Terme stellen die der jeweiligen Sorte zugeordneten Termmengen dar, ihre Vereinigung bildet den Wertebereich der Sorte. Im Beispiel wird zunächst mit

```
sort #const = a(), b();
```

festgelegt, dass die einzigen Terme, die zur Sorte `#const` gehören, die Konstanten `a` und `b` sind. Konstanten müssen stets mit einem leeren Klammerpaar abgeschlossen werden. In der nächsten Zeile

```
sort #list = nil(), c(#const, #list);
```

wird `#list` definiert: Zunächst gehört `nil` zur Sorte `#list`. Ebenso sind auch alle Terme der Form `c`(t_1, t_2) von der Sorte `#list`, für die t_1 ein Term der Sorte `#const` und t_2 ein Term der Sorte `#list` ist. Damit haben wir die von uns bereits mehrfach benutzten Listenterme mit den Listenelementen `a` und `b` definiert. `nil` stellt die leerer Liste dar. Als nächstes erfolgt die Zuweisung von Sorten an die im Grundtermersetzungssystem vorkommenden Variablen. Eine Variablendeklaration wird eingeleitet durch das Schlüsselwort `var`, gefolgt von einem Variablenbezeichner. Dieser kann Buchstaben und Zahlen enthalten, muss jedoch mit einem Buchstaben beginnen. Es folgt das Schlüsselwort `is`, ein Sortenbezeichner und das "`;`". Z.B. weist die Deklaration

```
var ys is #list;
```

der Variablen `ys` die Sorte `#list` zu.
Nach Signatur, Sorten und Variablen folgen nun die Regeldefinitionen. Sie werden durch das Schlüsselwort `rule` eingeleitet, gefolgt von einem Term (der linken Seite der Regel) , einem "=>" und einem weiteren Term (der rechten Seite der Regel), gefolgt von einem "`;`". Bei Angabe von Termen ist zu beachten, dass Konstanten auch hier mit einem leeren Klammerpaar angegeben werden müssen. Bei Variablen muss vor dem Variablenbezeichner ein "`:`" angegeben werden. In den Termen der Regeldeklarationen können keine Sortenbezeichner vorkommen. Alle Funktionssymbole und Variablen müssen zuvor deklariert worden sein. Außerdem ist zu beachten, dass eine Variable nur maximal einmal auf der linken und einmal auf der rechten Seite vorkommen kann. Ebenso kann eine Variable nicht nur auf der rechten Seite einer Regel vorkommen.
Die Menge von Testfällen wird im nächsten Abschnitt spezifiziert, eingeleitet von Schlüsselwort `initterm`, gefolgt von einem "`:`" und einem Term, der Variablen beinhalten kann, wobei wiederum jede Variable nur einmal vorkommen kann. Abgeschlossen wird wieder mit einem "`;`". Aus diesem Term wird die Menge der Testfälle gewonnen. Z.B. wird mit

```
initterm: append(c(b(), :xs), nil());
```

eine Testfallmenge erzeugt, die aus allen Termen der Form $append(c(b, t_{xs}, nil))$ besteht, wobei für t_{xs} jeder Term aus dem Wertebereich von `xs` eingesetzt werden kann.
Ist die Nachfolgermenge erst einmal berechnet, so soll festgestellt werden können, ob innerhalb der Nachfolgermenge bestimmte Terme enthalten sind. Dies geschieht mit der `contains`-Anfrage, die durch das Schlüsselwort `contains` eingeleitet wird, dem ein Term mit ähnlichem Aufbau wie bei der `initterm`-Deklaration folgt, und die mit einem "`?`" abgeschlossen wird. Beispielsweise kann mit

```
contains: c(a(), :xs) ?
```

die Anfrage gestellt werden, ob ein Listenterm in der Nachfolgermenge enthalten ist, der mit der Konstante `a` beginnt.

Einschub: Sortendeklarationen

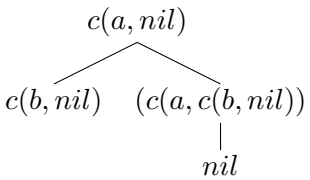
Die verwendete Art der Sortendefinition erlaubt es, auch mit solchen Sorten zu arbeiten, deren Terme einen komplizierteren Aufbau haben. Beispielsweise lässt sich eine Sorte von Binärbäumen `#tree` mit maximaler Tiefe 5, deren Knoten mit Listen beschriftet sind, folgendermaßen definieren:

```
// Symbole
symbols: a/0, b/0, c/2, nil/0, node/3;
// Sorten und Variablen
sorts: #const#0, #list#3, #tree#5;
sort #const = a(), b();
sort #list = nil(), c(#const, #list);
sort #tree = nil(), node(#list, #tree, #tree);
...
```

Hierbei verwenden wir die Konstante `nil` als Darstellung des "leeren" Binärbaums, während das 3-stellige Funktionssymbol `node` einen einzelnen Knoten eines Baumes darstellt. Seine Knotenbeschriftung findet sich in der ersten Stelle, die beiden Unterbäume in der zweiten und dritten. Beispielsweise stellt

$$node(c(a, nil), node(c(b, nil), nil, nil), node(c(a, c(b, nil)), nil, node(nil, nil, nil)$$

den Binärbaum



dar.

5.8.2 Aufruf des Prototypen

Wir führen das Beispiel vom vorherigen Abschnitt fort und wollen die dort definierte Eingabedatei verarbeiten. Für die Installation des Prototypen wird auf die der Installationsdatei beiliegende Dokumentation verwiesen. Der auszuführenden Datei `PostCalc` wird der Pfad zur Eingabedatei übergeben:

```
java PostCalc PFADZURDATEI [bisinaive | bisi]
```

Es kann außerdem ein zweites Kommandozeilenargument angegeben werden. Bei Angabe von `bisinaive` wird bei jedem Iterationsschritt des symbolischen Algorithmus eine Zustandsreduktion durch den naiven Backward-Bisimulationsalgorithmus durchgeführt, durch das Argument `bisi` wird die implementierte Variante aus [HMM07] verwendet (siehe dazu 5.6.1). Im vorliegenden Beispiel lauten die letzten Zeilen der Ausgabe:

```
...
Beginne Nachfolgerberechnung.
Der Schnitt ist leer!
```

Die Ausgabe macht Sinn, da wir lediglich solche Terme als Testfälle ausgewählt haben, die zu Listentermen der Art $c(b, ...)$ abgeleitet werden. Ändern wir jedoch die `contains`-Abfrage auf

```
contains: c(b(), c(a(), :xs)) ?
```

so erhalten wir beispielsweise folgende Ausgabe:

```
Beginne Nachfolgerberechnung.
Ende.

Der Schnitt ist NICHT leer, Gegenbeispiel:
c(b,c(a,c(b,nil)))

Ableitung des Zeugen:
c(b,c(a,c(b,nil)))
[append(nil,xs) => xs]
 <== c(b,c(a,c(b,append(nil,nil))))
[append(c(x,xs),ys) => c(x,append(xs,ys))]
 <== c(b,c(a,append(c(b,nil),nil)))
[append(c(x,xs),ys) => c(x,append(xs,ys))]
 <== c(b,append(c(a,c(b,nil)),nil))
[append(c(x,xs),ys) => c(x,append(xs,ys))]
 <== append(c(b,c(a,c(b,nil))),nil)
```

Nun gibt es Terme, die im Schnitt der Nachfolgermenge und der Testmenge liegen. Das Tool liefert uns den Terms $c(b, c(a, c(b, nil)))$ als Gegenbeispiel und gibt eine Rekonstruktion einer Ableitungsfolge an: Im ursprünglichen Automaten war z.B. der Term $append(c(b, c(a, c(b, nil))), nil)$ enthalten, der sich zum Gegenbeispiel ableiten lässt. Anzumerken ist, dass nicht unbedingt ein

minimales Gegenbeispiel in Bezug auf Anzahl der vorkommenden Funktionssymbole oder Tiefe des Terms ausgegeben wird. In diesem Fall wäre beispielsweise $c(b, c(a, nil))$ der Term eines Gegenbeispiels mit weniger Funktionssymbolen.

5.9 Parser und Eingabedateien

5.9.1 Parser

Da das Eingabeformat nur eine relativ einfache Grammatik benötigt, wurde der Parser mithilfe des `JavaCC`-Werkzeugs erzeugt, einem Top-Down-Parsergenerator für Java. Die Grammatikspezifikation ist in der Datei `parser.jj` im Package `parser` abgelegt. Eventuelle Erweiterungen lassen sich leicht einfügen.

5.9.2 Erstellung des Grundtermersetzungssystems

Das Package `inputprocessing` enthält Klassen, die das Erstellen des Grundtermersetzungssystems auf Basis einer Eingabedatei unterstützen. Hauptaufgabe ist hierbei das Erstellen des Variablenautomaten, da die Sorten ja auch vom Benutzer angegeben werden. Hierzu verwalten Instanzen der Klasse `SortAutomatons` eine Liste von `TermSort`-Instanzen. Jede `TermSort`-Instanz entspricht einer Sorte der Eingabedatei, von der unter anderem die maximale Tiefe gespeichert wird. Außerdem verwaltet `SortAutomatons` eine Automateninstanz `aut`, aus der später der Variablenautomat des symbolischen Grundtermersetzungssystems gewonnen wird.
Beim Lesen der Sortendeklarationen wird in `aut` für jede Sorte ein Zustand erzeugt und in der entsprechenden Instanz von `TermSort` abgespeichert. Beim Einlesen einer Sortendefinition wird für jeden Term der Termliste die Methode `addTermExpression` aufgerufen. Die Methode erstellt Regeln und Zustände, so dass der Term vom Zustand der übergebenen Sorte erkannt wird. Kommt ein Sortenbezeichner im Term vor, so wird er durch den Zustand der Sorte ersetzt.

Beispiel 18. Wir greifen einmal mehr das Beispiel aus 5.4.3 auf, das mit der Signatur $\Sigma = \{a(), b(), nil(), c(\cdot,\cdot), append(\cdot,\cdot)\}$ arbeitet. Wir wollen im weiteren Verlauf des Beispiels die Automaten meist nur durch ihre Regelmengen beschreiben, da Zustände und Regeln in die Automaten eingefügt werden können.
Zunächst beinhaltet der Automat `aut` der `SortAutomatons`-Instanz nach Verarbeiten der Sortendeklarationen zwei Zustände q_{const} und q_{list}, die ausgezeichneten Zustände für die Sorten `#const` und `#list`. Zunächst wird die Sortendefinition von `#const` gelesen; dem Automat werden folgende neuen Regeln hinzugefügt:

$$a \to q_{const}, b \to q_{const}$$

Nach dem Lesen der zweiten Sortendefinition kommen noch die Regeln

$$nil \to q_{list}, c(q_{const}, q_{list}) \to q_{list}$$

hinzu. □

Die von den ausgezeichneten Zuständen der Sorten erkannten Termmengen können unendlich groß sein. Daher wird die Tiefenangabe bei der Definition dazu benutzt, die Termmengen einzuschränken. Dies geschieht mit Hilfe der Methode `getSortAutomaton(TermSort)`: Sei q_s der ausgezeichnete Zustand der Sorte s mit der Tiefenbeschränkung d_s. Es wird ein Automat $\mathcal{A}_s$ mit einem akzeptierenden Zustand q_f erstellt, der *jeden* Term mit maximaler Tiefe d_s erkennt, der sich mit der Signatur des Grundtermersetzungssystems bilden lässt. Mit diesem Automaten und `aut` wird eine Produktautomatenkonstruktion durchgeführt, bei dem nur (q_f, q_s) als akzeptierender Zustand markiert wird. Dadurch kann der eigentliche Variablenautomat für ein `RewritingSystem` erstellt werden. Dies geschieht durch den Aufruf von `registerSorts(RewritingSystem)`, der alle Sorten im übergebenen Grundtermersetzungssystem einträgt.

Beispiel 19. Wir wollen die Sorte `#list` auf Terme der Tiefe ≤ 1 beschränken. Ein Automat $\mathcal{D}_1 = (\{q_0, q_1, q_f\}, \Sigma, \{q_f\}, \delta_{D_1})$, der alle Terme maximaler Tiefe 1 erkennt, besitzt z.B. folgende Regelmenge δ_{D_1}:

$$\delta_{D_1} = \{a \to q_0, b \to q_0, nil \to q_0, nil \to q_f,$$

$$c(q_0, q_0) \to q_1, append(q_0, q_0) \to q_1, c(q_0, q_0) \to q_f, append(q_0, q_0) \to q_f\}$$

Der Sortenautomat für `#list` hat folgende Regelmenge:

$$\{a \to (q_0, q_{const}), b \to (q_0, q_{const}), nil \to (q_0, q_{list}), nil \to (q_f, q_{list}),$$

$$c((q_0, q_{const}), (q_0, q_{list})) \to (q_f, q_{list})\}.$$

Akzeptierender Zustand dieses resultierenden Sortenautomaten ist (q_f, q_{list}).□

6 Untersuchungen und Anwendungen

Der symbolische Algorithmus wurde entwickelt, um die größten Probleme des nichtsymbolischen abzumildern: Lange Laufzeiten und hoher Speicherverbrauch. In diesem Kapitel werden zunächst praktische Testfälle dazu benutzt, den symbolischen mit dem nichtsymbolischen Ansatz zu vergleichen. Wir stellen Beispiele vor, in denen der symbolische Algorithmus nur wenig oder gar keine Verbesserung bringt. Auch prinzipielle Einschränkungen des symbolischen Ansatzes werden besprochen. Wir untersuchen den Einfluss von Verbesserungen, die den symbolischen Ansatz beschleunigen können. Schließlich erläutern wir weitere Einsatzmöglichkeiten des Verfahrens außerhalb der funktionalen Programmierung.

6.1 Analyse von Testfällen

Wir wählten einige einfache funktionale Programme als Eingabedateien für den Prototypen aus. Die Testfälle stammen vorwiegend aus dem Bereich der Listenverarbeitung, einige arbeiten mit Binärbäumen; siehe Abb. 6.1 für eine Übersicht der wichtigsten Beispiele. Alle dieser Testfälle haben ihren Ursprung in [Web04], wo sie als Übungsaufgaben für das automatische Beweissystem Isabelle dienen.
Die Wertebereiche der einzelnen Sorten (wie z.B. Listen) variieren von Problem zu Problem.
Für diese Beispiele wurde zu Testzwecken die Nachfolgermenge einer großen Menge von Termen berechnet, die sich als Baumautomaten mit kleiner Zustandszahl beschreiben ließen. Wir greifen als Beispiel die Datei `snoc.rul` heraus. Darin wird ein Funktionssymbol *snoc* definiert, das ein Element an das Ende einer Liste anfügt:

```
// Signatur und Sorten
symbols: 0/0, 1/0, c/2, nil/0, snoc/2;
sorts: #list#5, #const#1;
sort #const = 0(), 1();
sort #list =  nil(), c(#const, #list);
sort #errlist = c(#const, #errlist), c(0(), nil());

// Variablen
var l1 is #list;
var c1 is #const;
```

Testfallname	Beschreibung
snoc.rul	Anhängen von Elementen an das Ende einer Liste
replace.rul	Ersetzen von Elementen einer Liste durch andere
remove.rul	Entfernen von Listenelementen
reverse.rul	Listenelemente umkehren
occurs.rul	Das Vorkommen eines bestimmten Listenelementes zählen
flatten.rul	Verschachtelte Listen in eine flache Form bringen
sum.rul	Berechnen der Summe aller Listenelemente
first_pos.rul	Berechnen des ersten Vorkommens eines Listenelements
zip.rul	Zusammenfügen zweier Listen im Reißverschlussverfahren
union.rul	Vereinigung zweier als Listen dargestellter Mengen
tree_traversal.rul	Traversierung von Binärbäumen
binary_trees.rul	Zählen der Knoten vollständiger Binärbäume
merge.rul	Einfache Mische-Prozedur des Mergesort-Algorithmus
bdd.rul	Einfache Darstellung einer Datenstruktur für BDDs

Abbildung 6.1: Tabelle einiger untersuchter Testfälle.

```
var c2 is #const;
var el is #errlist;

// Regeln
rule snoc(nil(), :c1) => c(:c1, nil());
rule snoc(c(:c2, :l1), :c1) => c(:c1, snoc(:l1, :c2));

// Anfangsterme
initterm: snoc(:l1, 1());

// Testmenge
contains: :el ?
```

Dies stellt eines der kleinsten Beispiele mit nur zwei Regeln dar. Die Anfangsmenge besteht aus allen Termen der Art $snoc(t, 1)$, wobei t eine beliebige Liste der Länge maximal 5 ist (die maximale Listenlänge wurde bei den Testfällen variiert). Mithilfe der Testmenge wird überprüft, ob ein Term mit 0 als letztem Element ableitbar ist; dies würde auf einen Fehler der Funktionsdefinition hinweisen.

Untersucht wurden die Geschwindigkeiten, mit denen die verschiedenen Ansätze die Nachfolgermenge errechneten. Ebenso wurde gemessen, wieviel Zustände generiert wurden, um einen Ansatzpunkt für den Speicherverbrauch zu haben. Dazu wurden beim symbolischen Algorithmus sowohl die Zustände der $\mathcal{A}_i$-Automaten als auch die Zustände der Automaten in den $Copies_r$-Listen gezählt.

6.1.1 Vergleich symbolischer - nichtsymbolischer Algorithmus

Mit einer einfachen Implementierung des nichtsymbolischen Algorithmus (siehe 4.1) wurden einige Vergleiche durchgeführt. Hierbei ist anzumerken, dass diese Experimente eher Stichprobencharakter haben und nur Tendenzen aufzeigen können: Die Implementation des Algorithmus beruht auf dem Framework des symbolischen Algorithmus; daher konnten nicht alle möglichen Optimierungen ausgeschöpft werden. Insbesondere wird in [Löd02] eine mögliche Beschleunigung des Algorithmus besprochen, die hier nicht implementiert wurde. Jedoch werden bei diesem mindestens genau so viele Zustände erzeugt wie bei der hier vorgestellten Version des nichtsymbolischen Algorithmus.
Aus den Testfällen in Abb. 6.1 wurden die sechs Beispiele `first_pos.rul`, `remove.rul`, `merge.rul`, `reverse.rul`, `occurs.rul` und `snoc.rul` ausgewählt, also allesamt Fälle, bei denen Listen als Sorten bzw. Wertebereiche von Variablen dienen, die zwei Konstanten als Element besitzen.
Die Auswahl wurde getroffen, da die hier erstellte Implementierung des nichtsymbolischen Algorithmus nur Variablen mit Konstanten oder Listen als Wertemengen verarbeiten kann (siehe 4.1).
Die Testfälle wurden auf einem gewöhnlichen Linux-System mit 1 GB Hauptspeicher untersucht. Beim symbolischen Algorithmus wurde eine Backward-Bisimulationsreduzierung nach jedem Schleifendurchlauf durchgeführt. Beim nichtsymbolischen Algorithmus wurde lediglich die Zeit gemessen, die der erste Schleifendurchlauf benötigt. Wir untersuchten mehrere Testreihen, bei denen die Maximallänge der Listen für Variablen zwischen 5 und 9 variierte (siehe Abb. 6.2). Für den symbolischen Algorithmus wurde sowohl die Anzahl der im Endautomat enthaltenen Zustände als auch die der Automaten der $Copies_r$-Listen untersucht.
Dabei stellten wir fest, dass bei einer maximalen Listenlänge von fünf der nichtsymbolische Algorithmus den symbolischen Ansatz in puncto Geschwindigkeit noch leicht schlägt. Dies kann dadurch erklärt werden, dass für die Ausführung des nichtsymbolischen Algorithmus nach der Initialisierungsphase nur Ableitungsoperationen nötig sind, wohingegen beim symbolischen mehrere rechenintensive Schritte pro Zustand durchgeführt werden müssen, wie beispielsweise das Deterministisch-Machen von Automaten und Tests auf Automatenäquivalenz. Der Vorteil des symbolischen Algorithmus, mehrere entfaltete Regeln auf einmal zu behandeln, fällt bei den Testfällen mit kleineren Wertebereichen weniger ins Gewicht. Steigt die Listenlänge an, schlägt der symbolische Algorithmus seinen Gegenspieler deutlich, sowohl bezüglich Geschwindigkeit als auch bei der Anzahl generierter Zustände. Die Werte legen nahe, dass die Laufzeit und Zustandszahl des nichtsymbolischen Algorithmus bei den hier untersuchten sechs Testfällen exponentiell in der Länge der Listen wächst. Beim symbolischen Ansatz kann hingegen lineares Wachstum vermutet werden, sowohl bei den Zustandszahlen als auch bei den Geschwindigkeiten (siehe dazu Abb. 6.3).
Für die experimentellen Daten aus Abb. 6.1 wurde die Backward-Bisimulation

Dateiname	Listen-länge	Zeit in sec nichtsym.	Zeit in sec sym.	Zustände nichtsym.	Zustände sym.	Zustände in $Copies$
first_pos.rul	5	0,5	0,7 (1,2)	294	19 (37)	36
first_pos.rul	6	0,8	0,8 (1,6)	583	22 (46)	46
first_pos.rul	7	1,6	1,0 (1,8)	1160	25 (49)	57
first_pos.rul	8	6,4	1,3 (2,4)	2313	28 (55)	69
first_pos.rul	9	34,0	1,2 (2,3)	4618	31 (61)	82
occurs.rul	5	0,5	1,0 (2,9)	356	20 (46)	64
occurs.rul	6	1,7	1,3 (5,1)	709	23 (54)	83
occurs.rul	7	1,8	1,9 (5,6)	1414	26 (62)	104
occurs.rul	8	7,1	2,1 (7,3)	2823	29 (70)	127
occurs.rul	9	36,7	3,6 (9,0)	5640	32 (78)	152
remove.rul	5	0,5	1,2 (3,8)	418	20 (51)	65
remove.rul	6	1,5	1,4 (4,8)	835	23 (60)	84
remove.rul	7	3,5	2,1 (6,3)	1668	26 (69)	105
remove.rul	8	17,2	2,5 (8,5)	3333	29 (78)	128
remove.rul	9	90,2	3,0 (8,4)	6662	32 (87)	153
replace.rul	5	1,7	2,1 (6,3)	1291	60 (223)	279
replace.rul	6	7,5	3,2 (7,0)	2604	71 (266)	359
replace.rul	7	36,6	3,7 (9,1)	5229	82 (309)	447
replace.rul	8	180,1	4,1 (8,7)	10478	93 (352)	543
replace.rul	9	940,0	4,3 (9,2)	20975	104 (395)	647
reverse.rul	5	0,5	1,4 (2,8)	420	26 (65)	56
reverse.rul	6	0,8	2,0 (3,9)	837	31 (83)	73
reverse.rul	7	2,4	2,4 (4,9)	1670	36 (103)	92
reverse.rul	8	12,1	3,4 (6,4)	3335	41 (125)	113
reverse.rul	9	55,9	3,6 (7,5)	6664	46 (149)	136
snoc.rul	5	0,4	0,5 (1,0)	296	24 (53)	63
snoc.rul	6	0,5	0,7 (1,0)	585	28 (62)	81
snoc.rul	7	1,6	0,6 (2,2)	1162	32 (71)	101
snoc.rul	8	7,2	0,8 (1,5)	2315	36 (80)	123
snoc.rul	9	29,9	1,2 (2,1)	4620	40 (89)	147

Abbildung 6.2: Testergebnisse für den Vergleich zwischen symbolischem und nichtsymbolischem Ansatz, gemessene Geschwindigkeiten und Zustandszahlen. Die Geschwindigkeiten wurden auf die erste Stelle hinter dem Komma gerundet. In Klammern sind die Zahlen angegeben, die sich bei Weglassen der Backward-Bisimulation ergeben.

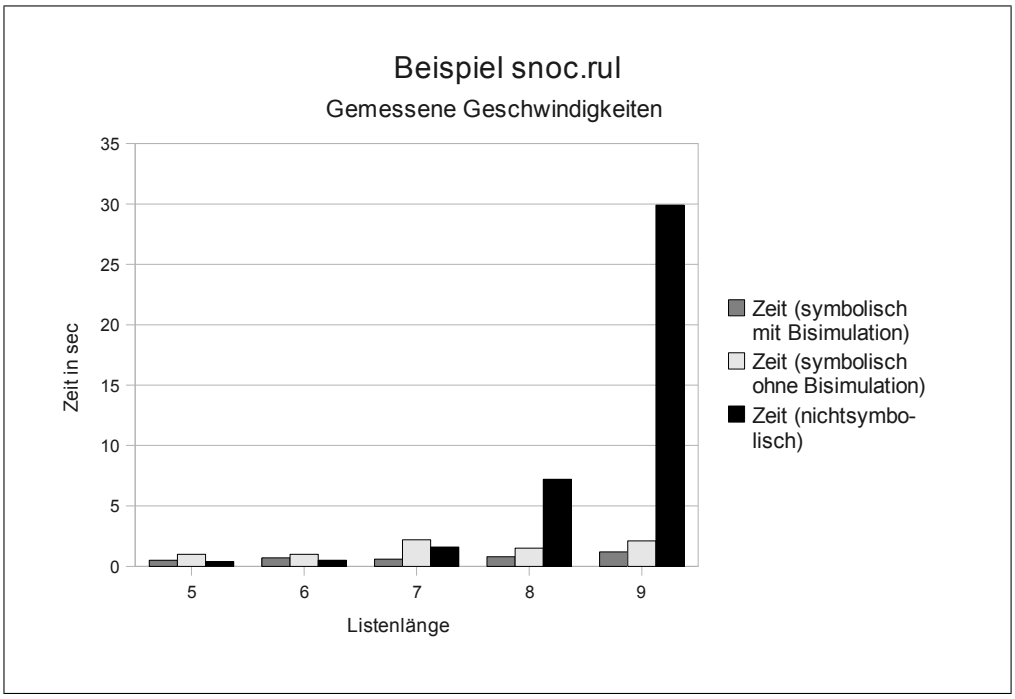


Abbildung 6.3: Laufzeiten bei `snoc.rul`.

als Reduktionsverfahren für den symbolischen Algorithmus angewandt; dadurch konnte die Geschwindigkeit des Verfahrens in vielen Fällen verdoppelt und die Anzahl der Zustände im Endautomaten verkleinert werden. Jedoch ist bei den hier untersuchten Beispielen der symbolische Algorithmus auch ohne Backward-Bisimulation dem nichtsymbolischen Algorithmus überlegen (siehe dazu die Werte in Klammern in Abb. 6.1 und Abschnitt 6.3.1).

Der symbolische Algorithmus konnte bei den untersuchten Testfällen ausnutzen, dass ein Baumautomat mit einer kleinen Anzahl von Zuständen existierte, um die gewünschte Nachfolgermenge darzustellen. Das Verfahren scheint eine sehr gute Annäherung an diese minimalen Automaten zu finden. Dies legen vor allem die Zustandszahlen in Kombination mit Backward-Bisimulation nahe.

Bei anderen Testfällen kann der nichtsymbolische Ansatz ab einer bestimmten Größe der Variablenwertebereiche nicht mehr praktikabel arbeiten. Beispielsweise wurde bei der Datei `flatten.rul` die darin definierte Funktion *flatten* auf alle verschachtelten Listen der maximalen Tiefe fünf angewandt, in denen 0 und 1 als "atomare" Elemente vorkommen (mehr als 2^{32} Terme). Auch die Variablen der symbolischen Regeln haben bei diesem Beispiel maximale Tiefe fünf. Die daraus resultierende Menge an entfalteten Regeln kann vom nichtsymbolischen Ansatz nur mit riesigem Zeit-und Speicheraufwand verarbeitet werden (der Automat hätte mehr als 2^{32} Zustände), im Gegensatz zum symbolischen Algorithmus: Die kompakte Darstellung der Wertemengen ermöglicht auch das Verarbeiten solcher Beispiele.

6.1.2 Laufzeitintensive Testfälle

Bei allen Testfällen aus 6.1 führte die Eigenschaft des symbolischen Algorithmus, die Anwendung einer symbolischen Regel ohne Untersuchen jeder dazugehörigen entfalteten Regel durchzuführen, zu einer starken Beschleunigung des Verfahrens. Dies war möglich, da viele Zustände mehrere entfaltete linke Seiten einer symbolischen Regel erkennen, und für diese gleichzeitig in Form neuer Regeln und Zustände die entsprechenden rechten Seiten eingefügt werden konnten. Mithilfe von Bisimulationsalgorithmen entstanden außerdem kleinere Automaten, die in vielen Fällen den minimalen nichtdeterministischen Baumautomaten in Bezug auf die Zustandsanzahl nahe kamen. Existiert jedoch für die zu berechnende Nachfolgermenge von Termen nur ein Automat mit besonders vielen Zuständen, so müssen auch bei Ausführung des symbolischen Ansatzes viele Zustände hinzugefügt werden. Die entstehenden Automaten können die Größe derer annehmen, die vom nichtsymbolischen Algorithmus zu Beginn seiner Ausführung bereits erstellt werden.
Es wurden hierzu Beispiele untersucht. Das Resultat dieser Testfälle ist oft ein besonders großer Automat, bei dem die einzelnen Zustände nur eine kleine Termmenge erkennen können. In der Praxis treten solche Fälle z.B. in Zusammenhang mit einer Gleichheitsrelation zwischen Termen auf. Wir betrachten hierzu ein symbolisches Grundtermersetzungssystem über der Signatur

$$\Sigma = \{a(), b(), nil(), c(\cdot,\cdot), test(\cdot,\cdot)\}$$

mit folgenden Regeln:

$$test(xs, ys) \to test(c(a, xs), c(a, ys)),$$

$$test(xs, ys) \to test(c(b, xs), c(b, ys)),$$

wobei den Variablen xs, ys als Wertebereich die Menge aller Listenterme der Form $c(a_1, c(a_2, ..., c(a_n, nil)...))$ mit $a_i \in \{a, b\}$ für alle $i \geq 0$ mit maximaler Länge $n \geq 0$ zugewiesen wird.
Die Nachfolger des Terms $test(nil, nil)$ sind alle Terme der Form $test(t_1, t_2)$ mit $t_1 = t_2$, wobei t_1 und t_2 Listen maximaler Länge n sind. Ein dazugehöriger Automat muss mindestens $2^{n+1} - 1$ verschiedene Zustände beinhalten: Jeder Term aus dem Wertebereich von x_1 und x_2 benötigt einen "eigenen" Zustand, der ihn und sonst keinen anderen Term erkennt. Nehmen wir an, es gäbe für ein t aus dem Wertebereich keinen solchen Zustand. Wir wissen, dass der Term $test(t, t)$ in einen akzeptierenden Zustand q_f reduziert werden kann, also existiert eine Reduktion, in der t auf einen Zustand q reduziert wird, und $test(q, t) \to^* q_f$. q akzeptiert laut Annahme aber noch mindestens einen anderen Term $t' \neq t$ aus dem Wertebereich von xs und ys, so dass auch $test(t', t)$ akzeptiert wird, was einen Widerspruch darstellt.
Das obige Beispielsystem wurde in Form der Eingabedatei `test.rul` getestet. Außerdem fand das Funktionssymbol $test$ in abgewandelter Form noch in der Datei `reversetest.rul` Verwendung, in der versucht wurde, eine einfache Eigenschaft einer Listenfunktion $reverse$ für Listen der Länge 3 und 4 nachzuweisen. Dazu wurde ein Funktionssym-

bol $reverse$ definiert, das die Elemente einer Liste umkehrt. Beispielsweise wertet sich $reverse(c(a_1, c(a_2, nil)))$ zu $c(a_2, c(a_1, nil))$ aus. Mit der bereits mehrfach definierten $append$-Funktion sollte nun getestet werden, ob $reverse(append(xs, ys)) = append(reverse(ys), reverse(xs))$ gilt, zumindest für xs, ys mit einem endlichen Wertebereich aus Listen. Um dies für den symbolischen Fall zu modellieren, wurden Techniken aus dem Abschnitt 6.2 verwendet; das $test$-Funktionssymbol ermöglichte es, die nichtlinearen Terme in dem Grundtermersetzungssystem zu simulieren.
Die Ergebnisse sind in Abb. 6.4 dargestellt. Für den nichtsymbolischen Algorithmus wurde wieder nur die Zeit für einen Iterationsschritt gemessen.
Bei der Ausführung des Algorithmus findet eine "Regelentfaltung" im Automaten statt: Die Zustände, die der nichtsymbolische Ansatz bereits am Anfang einfügen musste, werden vom symbolischen nach und nach eingefügt. Sie können dabei sogar die Zustandszahlen des nichtsymbolischen Ansatzes überschreiten. Dies liegt vor allem an den Zuständen in den $Copies$-Listen, die zusätzlich zum eigentlichen Automat der Nachfolgerterme abgespeichert werden. Dies stellt ein prinzipielles Problem bei der Beschränkung auf Baumautomaten dar: Es existieren keine Automaten mit weniger Zuständen für das Problem. Durch die anspruchsvolleren Operationen, die vom symbolischen Ansatz für jeden Zustand in jedem Schleifendurchlauf durchgeführt werden, kann in diesen Fällen der Algorithmus schlechtere Laufzeit- und Speicherplatzeigenschaften aufweisen als im nichtsymbolischen Fall. Der symbolische Ansatz führt unter anderem Äquivalenztests für jeden Zustand und jede Regel durch, sollte sich die erkannte Termmenge des Zustands seit der letzten Untersuchung verändert haben. Dies nimmt einen signifikanten Teil der Laufzeit in Anspruch. Die Reduktionen durch Bisimulationsäquivalenz können die Lage noch verschlimmern: Die Zustandsanzahl verringert sich durch ihren Einsatz kaum und die Durchführung des Reduktionsalgorithmus macht sich bei größer werdenden Automaten durch eine Verschlechterung der Laufzeit bemerkbar. Außerdem wachsen die $Copies_r$-Listen der symbolischen Regeln r des Grundtermersetzungssystems stark (siehe Zustandszahlen in Abb. 6.4), was sowohl ein Mehrfaches an Speicherplatznutzung als auch eine Geschwindigkeitsabnahme mit sich führt: Es werden mehr Äquivalenztests für Automaten nötig. Beim nichtsymbolischen Ansatz wurde im Testfall `reversetest.rul` die Möglichkeit ausgenutzt, nichtlineare Regeln für symbolische Grundtermersetzungssysteme zu benutzen. Dies ist für den symbolischen Algorithmus nicht möglich, siehe zu dieser Problematik auch den Abschnitt 6.2. Die Modellierung mit nichtlinearen Regeln wirkt sich besonders auf die Zahl der Zustände aus: Sie beträgt weniger als die Hälfte der Zustandszahl des symbolischen Algorithmus.
Beim Beispiel `test.rul` wurde für den nichtsymbolischen Algorithmus dasselbe Grundtermersetzungssystem verwendet wie für den symbolischen, also jeweils lineare Regeln verwendet. Dadurch baut der nichtsymbolische Ansatz zu Beginn der Ausführung einen sehr großen Automaten auf, wodurch er lange für die Berechnungen benötigt. Auch im symbolischen Fall werden relativ viele Zustände erzeugt, in diesem Beispiel jedoch nur ein Bruchteil der Anzahl seines Gegenspielers.

Testfall-name	Beschreibung	Zeit sym.	Zustände sym.	Zeit nichtsym.	Zustände nichtsym.
test.rul	Beispiel aus dem Text mit $n = 7$	12,1 sec	1024 +5610	> 10 min	> 160000
reversetest.rul	Listenlänge 3	65,0 sec	1918 +13009	27,5 sec	6053
reversetest.rul	Listenlänge 4	876,5 sec	7952 +64991	681,5 sec	25397

Abbildung 6.4: Laufzeitintensive Testfälle. Die Experimente wurden mit Backward-Bisimulation ausgeführt. Die Zustandszahlen des symbolischen Algorithmus teilen sich in Zustände des Endautomaten und Zustände der Automaten in den $Copies_r$-Listen auf.

Noch erwähnt werden sollte, dass die Effizienz der Algorithmen auch stark von den Anfangsautomaten, also der Anfangstermmenge, abhängt. Benötigt eine Darstellung der Termmenge, von der die Nachfolger berechnet werden sollen, bereits einen Automaten mit vielen Zuständen, so nähern sich Laufzeit und Speicherverbrauch für den symbolischen Algorithmus denen des nichtsymbolischen ebenfalls an: Auch hier findet meist eine weitere "Entfaltung" im Automaten statt. Leider lassen sich einige in der Praxis interessanten Fragen nur mit großen Anfangsautomaten lösen, da es für die Menge der Anfangsterme keine kleineren Baumautomaten gibt.
Wir können abschließend festhalten: Der symbolische Algorithmus kann mehr Speicherplatz in Form von Zuständen und mehr Zeitressourcen als der nichtsymbolische Ansatz benötigen. Dies ist vor allem dann der Fall, wenn es für die Nachfolgermenge keinen minimalen nichtdeterministischen Baumautomaten mit kleiner Zustandsmenge gibt. Zumindest bei den hier untersuchten Testfällen fielen die Einbußen des symbolischen Algorithmus nicht so schwer ins Gewicht wie beispielsweise bei 6.1.1 das Verhalten des nichtsymbolischen. Für den nichtsymbolischen Ansatz muss die Linearität symbolischer Regeln nicht gelten: So lassen sich funktionale Programme in einigen Fällen effizienter modellieren, siehe dazu auch den nächsten Abschnitt.

6.2 Einschränkungen des symbolischen Algorithmus

Wir haben für die Regeln eines symbolischen Grundtermersetzungssystems vorausgesetzt, dass sowohl linke als auch rechte Seiten linear sind. Eine Variable kann daher jeweils maximal einmal auf der linken und einmal auf der rechten Seite vorkommen. Dies schränkt die Anzahl der untersuchbaren funktionalen Programme stark ein. Wir betrachten als einfaches Beispiel die Definition einer Multiplikationsfunktion $mult$, wobei wir eine Additionsfunktion add als gegeben ansehen. Zahlen stellen wir als Terme $s^i(0)$ mit $0 \leq i \leq n \in \mathbb{N}$ und der Nachfolgerfunktion s dar. Dann sollte also gelten: $mult(s^i(0), s^j(0)) = s^{i \cdot j}(0)$. Eine mögliche Definition für $mult$ unter Benutzung der Funktion add mit

$add(s^i(0), s^j(0)) = s^{i+j}(0)$ wäre

$$mult(0, x) \to 0,$$

$$mult(s(x), y) \to add(y, mult(x, y)).$$

Bei der zweiten Regel kommt y zweimal auf der rechten Seite vor, daher lässt sich die Definition selbst einer so elementaren Funktion nicht direkt als symbolisches Grundtermersetzungssystem modellieren.
Besonders gravierend ist, dass die Einschränkungen es schwierig machen, im Termersetzungssystem z.B. zwei Funktionenaufrufe daraufhin zu untersuchen, ob sie dasselbe Ergebnis berechnen.
Jedoch ist es in vielen Fällen möglich, diese Beschränkung z.B. durch Hinzufügen von Regeln zu umgehen. Wir betrachten im Folgenden zwei Vorgehensweisen dazu.
Gut einsetzbar ist das Verfahren bei den meisten Programmen, die Listen elementweise verarbeiten und manipulieren (siehe Abb. 6.1), oder auch bei einfachen Programmen mit komplizierteren Datentypen wie Bäumen, die nur auf einzelne Baumknoten zugreifen.

6.2.1 Teilentfaltung symbolischer Regeln

Eine Teilentfaltung bietet sich an, wenn im ursprünglichen funktionalen Programm zwar eine Variable mehrfach auf einer Regelseite vorkommt, diese jedoch einen kleinen Wertebereich besitzt. Dann ist es möglich, die nichtlineare symbolische Regel durch mehrere lineare Regeln zu simulieren. Ebenso können einfache Prädikate über diesen Variablen errechnet werden. Als Beispiel betrachten wir das Gleichheits-Prädikat über der Termmenge $\{a, b, c\}$. Es soll durch das Funktionssymbol $equal$ dargestellt werden unter Verwendung der Konstanten $true$ und $false$ für die Wahrheitswerte "wahr" und "falsch". Dieses $equal$ soll also für Konstanten arbeiten im Gegensatz zu dem Funktionssymbol $equal$ des vorigen Abschnitts, das für Listen definiert wurde.
Das Prädikat kann durch die Regeln

$$equal(a, a) \to true,\ equal(a, b) \to false,\ equal(a, c) \to false,$$

$$equal(b, a) \to false,\ equal(b, b) \to true,\ equal(b, c) \to false,$$

$$equal(c, a) \to false,\ equal(c, b) \to false,\ equal(c, c) \to true$$

dargestellt werden. Hierbei wurde beispielsweise die mögliche nichtlineare Regel

$$equal(x, x) \to true$$

mit der Variable x durch drei Regeln $equal(a, a) \to true$, $equal(b, b) \to true$ und $equal(c, c) \to true$ ersetzt.

6.2.2 Simulation mehrfacher Variablenvorkommen mithilfe von Nachfolgermengen

Kommen Variablen mehrfach in einer Regelseite vor, die einen großen Wertebereich haben, würde der obige Ansatz zu einem System führen, das ähnliche Ausmaße wie das entfaltete Grundtermersetzungssystem annimmt. In gewissen Fällen kann jedoch mithilfe des Vorgangs der Nachfolgermengengenerierung und zusätzlichen Funktionssymbolen ein System ohne eine solche Regelentfaltung modelliert werden, mit dessen Hilfe die untersuchte Fragestellung ebenso gelöst werden kann.
Als Beispiel dazu betrachten wir ein funktionales Programm, von dem wir testen möchten, ob die darin definierten Funktionen f und g, die jeweils einen Listenausdruck der Form $c(a_1, c(a_2, ..., c(a_n, nil)...))$ mit $a_i \in \{a, b\}$ für $i \leq n$ mit $n \in \mathbb{N}$ als Argumente erhalten, dasselbe berechnen, also: Gilt $f(x) = g(x)$ für alle Listen x maximaler Länge n? Dazu greifen wir das $test$-Prädikat aus 6.1.2 auf und definieren sein Verhalten mit folgenden Regeln:

$$test(xs, ys) \rightarrow test(c(a, xs), c(a, ys)),$$

$$test(xs, ys) \rightarrow test(c(b, xs), c(b, ys)),$$

$$test(xs, ys) \rightarrow equal(f(xs), g(ys)).$$

Berechnen wir nun die Nachfolgermenge des Terms $test(nil, nil)$, so kann mit den ersten beiden Regeln zunächst eine Menge von Termen der Art $test(t_1, t_2)$ mit $t_1 = t_2$ generiert werden. Mithilfe der dritten Regel kann jeder dieser Terme zu einem Element der eigentlich zu untersuchenden Termmenge abgeleitet werden. Durch die ersten beiden Regeln und den Startterm wird sichergestellt, dass xs und ys gleiche Terme bezeichnen. Beim Generieren eines Gegenbeispiels können die Ableitungsschritte von $test(nil, nil)$ bis zum Term $equal(f(t_1), g(t_1))$ ignoriert werden.

6.2.3 Bewertung der Strategien

Für fast alle Testfälle aus 6.1 waren Teilentfaltungen symbolischer Regeln notwendig. Hierfür wurden Variablen mit kleinem Wertebereich verwendet, so dass die Größe der symbolischen Grundtermersetzungssysteme nur moderat anstieg. Die Probleme, die der zweite Ansatz zur Überwindung der Regel-Linearität mit sich bringt, wurden bereits in 6.1.2 angesprochen: Die Laufzeit des symbolischen Algorithmus nähert sich dem nichtsymbolischen an und kann sogar schlechter werden. Außerdem ist anzumerken, dass eine Modellierung durch zusätzliche Regeln nicht unbedingt zu leserlichen Eingabedateien beiträgt. Für viele Beispiele ist es zudem nicht unbedingt trivial bzw. einfach automatisierbar, ein nichtlineares Regelsystem durch Hinzufügen von Regeln in ein lineares umzuformen. Beispielsweise machten wir uns beim vorherigen Fall eine mögliche induktive Definition von Listentermen zunutze, um die Menge der Terme $test(t, t)$ mit t Listenterm zu erzeugen. Ein solch einfach strukturierter Aufbau ist nicht unbedingt für jeden Datentyp offensichtlich bzw. müsste noch zusätzlich angegeben werden.

6.3 Modifikationen und Verbesserungen

6.3.1 Einfluss von Reduktionsalgorithmen

Für den symbolischen Fall wurde der Einsatz von Backward-Bisimulationsäquivalenzen untersucht, die eine Reduktion der Zustandszahl ermöglichen können (siehe Abschnitt 5.6). Wir wollen in diesem Abschnitt mit Bisimulation immer die Backward-Bisimulation bezeichnen. Es wurden drei verschiedene Szenarien untersucht:

- keinerlei Anwendung von Bisimulationsalgorithmen,
- Anwendung des Bisimulationsalgorithmus gemäß [HMM07] nach jedem Schleifendurchlauf,
- Anwendung des naiven Bisimulationsalgorithmus nach jedem Schleifendurchlauf.

Es wurde die Zeit gemessen, die die einzelnen Algorithmenvarianten für die Durchführung der Iterationsschritte bis zur Sättigung der Baumautomaten benötigten, und die Anzahl der Zustände des resultierenden Automaten, siehe dazu Abb. 6.5. Hierbei wurden größere Wertebereiche als im vorherigen Abschnitt untersucht, z.B. Listen mit einer maximalen Länge von 35. Es zeigte sich, dass von den 14 untersuchten Beispielen

- in 10 Fällen durch Einsatz des naiven Bisimulationsalgorithmus die Laufzeit gegenüber den Varianten ohne Bisimulation verbessert werden konnte. Bei den drei Beispielen `flatten.rul`, `merge.rul` und `tree_traversal.rul`, die am meisten Laufzeit benötigten, konnte der naive Bisimulationsalgorithmus die Laufzeiten auf weniger als 15%, 40% bzw. 10% reduzieren.
- der Algorithmus nach [HMM07] zumindest in der vorliegenden Implementierung lediglich in einem Testfall effizienter war als seine naive Variante oder die Ausführung ohne Bisimulation. In allen anderen Fällen ist er zum Teil deutlich langsamer als die naive Variante. Selbst gegenüber der Ausführung ohne Bisimulation war er in sieben Fällen in puncto Geschwindigkeit unterlegen, wobei dies vor allem kleinere Beispiele betraf.

Das etwas überraschende Ergebnis bezüglich des Algorithmus nach [HMM07] könnte durch eine ineffiziente Implementierung erklärbar sein. Es ist jedoch auch wahrscheinlich, dass sich die Effizienz des Algorithmus im Vergleich zur naiven Variante erst bei Automaten mit deutlich höherer Zustandszahl auswirkt. Außerdem sorgen die vielen Hash-Operationen während der Ausführung des Algorithmus für eine Verlangsamung gegenüber der naiven Variante, die vergleichsweise einfache Datenstrukturen benötigt. Ebenso benötigt der Algorithmus zumindest bei den vorliegenden Beispielen mehr Speicher: Jeder Zustand besitzt eine eigene Hash-Tabelle und zum Bestimmen

Testfallname	Bisimulations-art	Zeit in sec	Zustandszahl des Endautomaten
bdd.rul	nobisi	10,8	173
bdd.rul	bisinaive	10,8	77
bdd.rul	bisi	14,4	77
binary_trees.rul	nobisi	27,9	1021
binary_trees.rul	bisinaive	13,7	364
binary_trees.rul	bisi	19,5	364
first_pos.rul	nobisi	3,2	112
first_pos.rul	bisinaive	2,9	50
first_pos.rul	bisi	4,6	50
flatten.rul	nobisi	453,2	1228
flatten.rul	bisinaive	66,2	412
flatten.rul	bisi	75,1	412
merge.rul	nobisi	167,2	4571
merge.rul	bisinaive	453,6	983
merge.rul	bisi	114,1	983
occurs.rul	nobisi	73,0	391
occurs.rul	bisinaive	62,1	148
occurs.rul	bisi	68,9	148
remove.rul	nobisi	111,3	141
remove.rul	bisinaive	119,2	50
remove.rul	bisi	7,7	50
replace.rul	nobisi	5,7	438
replace.rul	bisinaive	4,7	115
replace.rul	bisi	5,3	115
reverse.rul	nobisi	69,2	1918
reverse.rul	bisinaive	35,6	486
reverse.rul	bisi	783,9	486
snoc.rul	nobisi	44,6	278
snoc.rul	bisinaive	5,1	124
snoc.rul	bisi	7,1	124
sum.rul	nobisi	5,9	325
sum.rul	bisinaive	3,7	120
sum.rul	bisi	7,6	120
tree_traversal.rul	nobisi	184,6	1676
tree_traversal.rul	bisinaive	14,4	355
tree_traversal.rul	bisi	26,1	355
union.rul	nobisi	4,0	165
union.rul	bisinaive	3,2	52
union.rul	bisi	4,4	52
zip.rul	nobisi	9,1	836
zip.rul	bisinaive	9,2	709
zip.rul	bisi	24,1	709

Abbildung 6.5: Testergebnisse der verschiedenen Backward-Bisimulations-Varianten. “nobisi”: keine Anwendung von Backward-Bisimulation, “bisi”: modifizierter Algorithmus nach [HMM07],”bisinaive”: naive Variante.

der neuen Partitionen wird noch ein weiterer Hash benötigt, während der naive Algorithmus lediglich eine boolesche Matrix mit der Zustandszahl als Dimension benutzt. Daher wird beim nichtnaiven Ansatz häufig der Garbage-Collector der Java Virtual Machine aufgerufen. Außerdem müssen die Hash-Tabellen der Zustände beim Bilden der neuen Partitionen auf Gleichheit ihrer Einträge getestet werden, was nicht unbedingt in linearer Zeit erfolgen kann (siehe dazu 5.6.1).
Insgesamt legen die Ergebnisse der kleinen Testreihe jedoch die Vermutung nahe, dass die Anwendung von Backward-Bisimulation besonders bei umfangreichen Testfällen eine starke Geschwindigkeitssteigerung des Algorithmus verspricht: Je weniger Zustände der Automat enthält, desto schneller lässt sich ein Iterationsschritt durchführen. Die Größe der Automaten kann mithilfe der Bisimulationsalgorithmen in vielen untersuchten Fällen auf einen Bruchteil des ursprünglichen Automaten verkleinert werden: In der vorliegenden Testreihe betrug die Zustandszahl des mithilfe der Bisimulation erzeugten Endautomaten bei 13 von 14 Testfällen weniger als die Hälfte der Zustände des ohne Bisimulation erstellten Automaten.

Maximale Zustandszahlen bei Bisimulation

In Abb. 6.6 werden die Zustandszahlen der Endautomaten mit der maximalen Größe verglichen, die der Automat im Laufe des Algorithmus annehmen kann. Die maximale Zustandszahl kann in manchen Fällen mehr als das Dreifache der Zustandszahl des Endautomaten betragen: Die Automaten wachsen unter Einfluss der Bisimulation mit der Zeit nicht unbedingt monoton an, sondern können auch kleiner werden. Ähnliche Beobachtungen konnten auch beim Einsatz von BDDs, beispielsweise in Model-Checking-Tools, gemacht werden.

6.3.2 Weitere Verbesserungsstrategien

Bei Experimenten mit der Implementation des symbolischen Ansatzes konnten mehrere Faktoren ausgemacht werden, die großen Einfluss auf die Laufzeit haben und sich gegenseitig beeinflussen. Dazu gehören

1. die Anzahl der Zustände in dem konstruierten Automat,

2. die Anzahl der benötigten Äquivalenztests für Baumautomaten (siehe 5.4.2), und damit zusammenhängend die Größe der Listen $Copies_r$ jeder symbolischen Regel r,

3. die Anzahl der verschiedenen Reduktionen für einen Zustand q im Automaten, um Terme der entfalteten linken Seiten einer symbolischen Regel r abzuleiten.

Gibt es sehr viele verschiedene Reduktionen für q wie unter 3. beschrieben, so werden viele Automaten erzeugt, die auf Äquivalenz mit den Automaten in

Testfallname	Zustandszahl des Endautomaten	maximale Zustandszahl
bdd.rul	77	162
binary_trees.rul	364	452
first_pos.rul	50	95
flatten.rul	412	592
merge.rul	983	2214
occurs.rul	148	219
remove.rul	50	141
replace.rul	115	438
reverse.rul	486	565
snoc.rul	124	214
sum.rul	120	159
tree_traversal.rul	355	387
union.rul	52	63
worstcase.rul	1024	2049
zip.rul	709	776

Abbildung 6.6: Vergleich der Zustandszahlen der Endautomaten mit der maximalen Zustandszahl, die der Automat während der Durchführung des Algorithmus annimmt.

$Copies_r$ getestet werden müssen. Weniger schwer fiel die Produktautomatenkonstruktion ins Gewicht, da diese pro Schleifendurchlauf nur einmal durchgeführt werden musste. In Kapitel 5.4 wurden bereits mehrere Modifikationen des Algorithmus aus Abb. 4.4 vorgestellt, die den Ablauf des symbolischen Algorithmus beschleunigen können. Wir wollen hier noch zwei weitere Ansätze ausführen.

Eindämmung unnötiger Äquivalenztests

Der symbolische Algorithmus führt besonders bei Automaten mit vielen Zuständen häufig Äquivalenztests für Baumautomaten durch. In Kapitel 5.4 wurde gezeigt, wie durch das `changed`-Flag bei einem Iterationsschritt nur diejenigen Zustände untersucht werden müssen, deren erkannte Termmenge sich während dem letzten Schritt eventuell vergrößert hat. Damit können Äquivalenztests eingespart werden.
Wir untersuchen nun eine Verfeinerung dieses Verfahrens, bei dem der Einfluss der Regeln untereinander berücksichtigt wurde. Wir wollen sie innerhalb dieses Abschnitts als **verfeinerte** Methode bezeichnen. Das oben beschriebene Verfahren, welches lediglich das `changed`-Flag benutzt, wollen wir entsprechend als **grobes** Verfahren bezeichnen.
Zunächst erläutern wir die Idee der verfeinerten Methode. Wir betrachten als Beispiel die Regeln $r_1 = f(x) \rightarrow g(x)$ und $r_2 = g(x) \rightarrow h(x)$. x hat hierbei einen beliebigen endlichen Wertebereich. Die Regel r_1 beeinflusst die Regel r_2 folgendermaßen: Erkennt ein Zustand q in einem Automaten z.B. lediglich

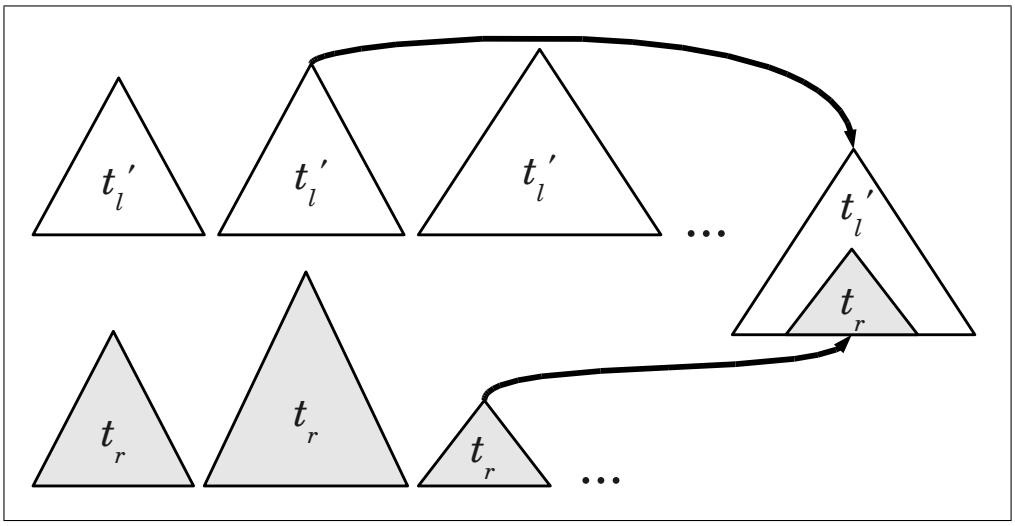


Abbildung 6.7: Beeinflussung symbolischer Regeln. Oben sind entfaltete linke Seiten der Regel r', unten entfaltete rechte Seiten von r dargestellt.

eine Teilmenge von $Expand(f(x))$, so erkennt er bei Anwendung des symbolischen Algorithmus nach einem Iterationsschritt die entsprechende Teilmenge von $Expand(g(x))$. Dadurch erst kann Regel r_2 auf q angewandt werden. Genauer gesagt sprechen wir davon, dass eine symbolische Regel $r = t_l \to t_r$ eine zweite Regel $r' = t_l' \to t_r'$ **beeinflusst**, wenn

$$Expand(SubTerms(t_l')) \cap Expand(t_r) \neq \emptyset,$$

siehe dazu auch Abb. 6.7).
Zu Beginn des symbolischen Algorithmus wird beim verfeinerten Verfahren die Beeinflussungs-Relation mittels der Methode `detectRuleInfluence` errechnet, die Bestandteil der Klasse `RewritingSystem` ist. Die Relation wird in Form einer booleschen Matrix gespeichert. Jeder Zustand eines Automaten erhält Flags in Form des booleschen Arrays `ruleupdate`. Sie zeigen an, ob er in dem jüngsten Iterationsschritt mehr Terme erkennt als vorher, und welche Regeln die neuen Terme beeinflussen können.
Wenn dem Automaten neue Zustände hinzugefügt werden, werden diese Flags aktualisiert, und zwar für die bereits vorhandenen Zustände, die mit den neuen durch ϵ-Regeln verbunden werden. Dazu wird die errechnete Relation benutzt. Durch `calculateTransitiveRuleUpdates` werden die Informationen an die anderen Zustände weiterpropagiert, die ebenfalls neue Terme erkennen könnten. Wird zum Simulieren einer symbolischen Regel lediglich eine ϵ-Regel $q \to q'$ eingefügt, da die benötigten Zustände und Regeln bereits zuvor in den Automaten eingefügt wurden, wird eine Überabschätzung des Regeleinflusses benutzt: In diesem Fall kann es nämlich sein, dass q' durch die ϵ-Regel mehr erkennt als nur entfaltete rechte Seiten einer symbolischen Regel. Wir nehmen hier den schlechtesten Fall an und setzen alle Einträge des Arrays `ruleupdate` für q' auf $true$.

Aufgrund dieser Überabschätzung ist die Geschwindigkeitszunahme durch die verfeinerte Methode im Vergleich zur speziellen Methode eher gering. Auch das Propagieren von Informationen im Baumautomat und das Berechnen der Relation ist zeitaufwändig. Bei Experimenten kamen sowohl Beispiele einer Geschwindigkeitssteigerung von bis zu ca. 10% als auch Verschlechterungen in ungefähr demselben Rahmen vor.
Jedoch könnte es möglich sein, durch eine genauere Unterscheidung, welche Terme wegen neuer Automatenregeln vom jeweiligen Zustand erkannt werden können, die Laufzeit des Algorithmus zu verbessern: So könnte besser entschieden werden, ob ein Zustand neue entfaltete linke Seiten einer symbolischen Regel ableiten kann. Das Problem besteht darin, die Termmengen effizient zu berechnen; hierfür wurde noch keine befriedigende Lösung gefunden.

Vorzeitiger Abbruch

Bis jetzt haben wir erst nach Berechnung der gesamten Nachfolgermenge der Testfälle begonnen, nach Termen aus der Testmenge zu suchen. Es spricht jedoch nichts dagegen, diese Suche bereits während der Ausführung des Algorithmus, z.B. nach jedem Schleifendurchlauf, durchzuführen. Dadurch kann unter Umständen bereits nach wenigen Iterationsschritten ein Gegenbeispiel gefunden und ausgegeben werden, und oftmals muss dadurch nur ein Bruchteil des Automaten aufgebaut werden. Dies ist für den nichtsymbolischen Algorithmus weniger effizient, da die Anzahl der Zustände im Automaten hier schon von Anfang an sehr hoch ist und lediglich ϵ-Regeln hinzugefügt werden. Seine Laufzeit kann jedoch durch einen vorzeitigen Abbruch ebenso vermindert werden.
Experimente mit den oben angegebenen Testfällen zeigen, dass in vielen Fällen eine starke Verkürzung der Laufzeit durch dieses Verfahren ermöglicht wird. Natürlich ist eine Verbesserung aber in erster Linie von der Art der Testmenge abhängig: Wenn im Extremfall der Schnitt von Nachfolgermenge und Testmenge leer sein sollte, muss durch die nach jedem Schleifendurchlauf durchgeführte Schnittbildung mit leicht längerer Laufzeit des Algorithmus gerechnet werden.

6.4 Anwendungen jenseits funktionaler Programmierung

Hauptziel dieser Arbeit ist es, Algorithmen für die Analyse funktionaler Programme zu entwickeln. Der hierzu entwickelte Ansatz erlaubt es jedoch auch, Probleme aus anderen Bereichen zu behandeln. Genauer gesagt fallen hierunter Anwendungen von Termersetzungssystemen, bei denen es sinnvoll sein kann, die darin vorkommenden Variablen auf einen endlichen Wertebereich von Termen einzuschränken. Wir wollen dazu ein Beispiel betrachten, bei dem der Ansatz das Beweisen eines einfachen Satzes erleichtert. Hierbei wollen wir

für alle natürlichen Zahlen $s_1, s_2, s_3 \in \mathbb{N}$ beweisen, dass

$$((s_1 + s_2) + s_3) = (s_1 + (s_2 + s_3)),$$

dass also die Addition assoziativ ist. Wir arbeiten mit der Signatur $\Sigma = \{0(), s(\cdot), add(\cdot,\cdot)\}$, die wir später noch erweitern werden. Natürliche Zahlen werden als Terme $s^i(0)$ mit $i \geq 0$ codiert, die Addition wird durch das Funktionssymbol add mit folgenden Regeln dargestellt:

$$add(0, x) \to x,$$

$$add(s(x), y) \to s(add(x, y)),$$

wobei $x, y \in \{0, s(0), s^2(0), ...\}$. Sie entsprechen der induktiven Definition von $add(x, y)$:

1. Ist $x = 0$, so gilt $add(x, y) = y$.
2. Ist $x = z + 1$ für ein $z \in \mathbb{N}_0$, so gilt $add(x, y) = 1 + add(z, y)$.

Wir wollen die Behauptung nun durch Induktion über s_1 beweisen. Dazu muss als Induktionsanfang zunächst die Aussage für $s_1 = 0$ bewiesen werden. Dazu bedienen wir uns folgender Eingabedatei:

```
symbols: 0/0, s/1, add/2, s2/0, s3/0;
sorts: #number#3;
sort #number = 0(), s(#number),
               s2(), s3(), add(#number, #number);
var n1 is #number;
var n2 is #number;
rule add(0(), :n1) => :n1;
rule :n1 => add(0(), :n1);
rule add(s(:n1), :n2) => s(add(:n1, :n2));
rule s(add(:n1, :n2)) => add(s(:n1), :n2);
initterm: add(add(0(), s2()), s3());
contains: add(0(), add(s2(), s3())) ?
```

Es werden zwei "symbolische" Variablen `s2` und `s3` eingeführt. Außerdem kommen die Regeln für das Funktionssymbol add auch in umgekehrter Ableitungsrichtung vor: Wir betrachten diese symbolischen Regeln als Kongruenzen (siehe [BN98]), und berechnen gleichermaßen $Post^*$ als auch Pre^*. Als Sorte `#number` wählen wir nicht nur Terme der Art $s^i(0)$ mit $i \geq 0$, sondern auch Terme mit dem Funktionssymbol add, die ebenfalls Zahlen darstellen bzw. sich zu solchen auswerten lassen. Die Tiefe der Terme ist auf drei begrenzt, was für unsere Zwecke ausreicht. Wir erhalten als Ausgabe das Gegenbeispiel

```
add(0,add(s2,s3))
[add(0,n1) => n1]
 <== add(0,add(add(0,s2),s3))
[n1 => add(0,n1)]
 <== add(add(0,s2),s3)
```

Damit ist eine Beweisfolge für den Induktionsanfang gegeben: $add(add(0, s_2), s_3) = add(0, add(add(0, s_2), s_3)) = add(0, add(s_2, s_3))$ Den Induktionsschritt modellieren wir mit folgender Eingabedatei:

```
symbols: 0/0, s/1, add/2, s1/0, s2/0, s3/0;
sorts: #number#3;
sort #number = 0(), s(#number),
     s1(), s2(), s3(), add(#number, #number);
var n1 is #number;
var n2 is #number;
rule add(0(), :n1) => :n1;
rule :n1 => add(0(), :n1);
rule add(s(:n1), :n2) => s(add(:n1, :n2));
rule s(add(:n1, :n2)) => add(s(:n1), :n2);
rule add(add(s1(), s2()), s3()) =>
     add(s1(), add(s2(), s3()));
rule add(s1(), add(s2(), s3())) =>
     add(add(s1(), s2()), s3());
initterm: add(s(s1()), add(s2(), s3()));
contains: add(add(s(s1()), s2()), s3()) ?
```

Hierbei wird die Induktionshypothese

$$add(s_1, add(s_2, s_3)) = add(add(s_1, s_2), s_3)$$

durch ein zusätzliches Regelpaar simuliert. Als Ableitung eines Gegenbeispiels wird

$$add(s(s_1), add(s_2, s_3)) = s(add(s_1, add(s_2, s_3))) =$$

$$s(add(add(s_1, s_2), s_3)) = add(s(add(s_1, s_2)), s_3)$$

zurückgegeben. Beim zweiten Ableitungsschritt wird die Induktionshypothese benutzt. Da die Termtiefe der Sorten beschränkt ist, kann eventuell ein existierender Beweis in Form einer Ableitung im Grundtermersetzungssystem nicht gefunden werden. Eine Erhöhung der maximalen Sortentiefe ist in solchen Fällen sinnvoll, um mehr Fälle abzudecken.
Bei Untersuchungen dieser Anwendungsart mithilfe des symbolischen Algorithmus fiel auf, dass nach einiger Zeit die einzelnen Iterationsschritte immer mehr Zeit in Anspruch nahmen. Dieses Phänomen ist einerseits auf Regeln mit mehreren Variablen mit großem Wertebereich zurückzuführen, andererseits auf Regeln, bei denen Teilmengen ihrer entfalteten linken Seiten von einer großen Zahl von Zuständen erkannt wird. Im obigen Beweis muss z.B. die Regel $n_1 \rightarrow add(0, n_1)$ für alle Zustände simuliert werden, die irgendeinen Term mit Maximaltiefe der Sorte von n_1 erkennen. Dadurch steigt die Zahl der neu eingefügten Zustände im Automaten oftmals rapide an.
Meist reicht eine begrenzte Zahl von Iterationsschritten aus, um eine Ableitung zu finden (siehe dazu auch 6.3.2). Um Beweise mit nichtlinearen symbolischen Regeln herzuleiten muss jedoch in der Regel auf den nichtsymbolischen

Algorithmus zurückgegriffen werden. Ab einer gewissen maximalen Tiefe der Variablenterme ist das nichtsymbolische Verfahren jedoch auch hier nur noch bedingt einsetzbar.
Diese Technik kann auch dazu benutzt werden, Eigenschaften von *funktionalen Programmen* per vollständiger Induktion nachzuweisen. In den von uns untersuchten Fällen war dazu meist nötig, weitere Eigenschaften der untersuchten Funktionen dem Regelsystem in Form von zusätzlichen Regeln bekannt zu machen, wie beispielsweise die Assoziativität oder Kommutativität von Funktionen; ansonsten konnte meist keine Beweisableitung gefunden werden.

7 Fazit

Der symbolische Ansatz für die Erreichbarkeitsanalyse von Grundtermersetzungssystemen kann den nichtsymbolischen Algorithmus in den Bereichen Geschwindigkeit und Speicherverbrauch übertreffen, vor allem im Falle einfacher funktionaler Programme. Selbst bei symbolischen Grundtermersetzungssystemen, die eine sehr große Menge entfalteter Regeln enthalten, kann das Verfahren effizient Nachfolgermengen berechnen: Einfache Erreichbarkeitseigenschaften können in diesen Fällen meist schnell und effizient geprüft werden. Die Generierung von Gegenbeispielen hilft beim Programmverstehen. Sorten können durch ihre Modellierung mittels Baumautomaten flexibel und komfortabel angegeben werden.
Die Probleme beim symbolischen Ansatz ergeben sich zum einen aus den Beschränkungen, die für die möglichen symbolischen Grundtermersetzungssysteme gelten. Damit das Verfahren korrekt bleibt, müssen die Regelterme linear sein. Nur einfache funktionale Programme lassen sich so ohne größere Probleme und Modifikationen modellieren. Es musste festgestellt werden, dass die Performanz des Verfahrens bei stark steigender Zustandszahl des Automaten abnimmt: Für jeden Zustand müssen umfangreiche Berechnungen vorgenommen werden. Auch bei Anfangstermmengen, die nur durch sehr große Baumautomaten repräsentierbar sind, ist das Verfahren weniger effizient. Es wurden Ansätze erörtert, wie die Auswirkungen dieser Probleme abgemildert werden können. Insbesondere das Einschränken unnötiger Äquivalenztests durch geschicktes Sammeln von Informationen könnte die Laufzeit des Verfahrens reduzieren.
Des Weiteren wurde aufgezeigt, dass die Anwendung sich nicht auf das Untersuchen funktionaler Programme beschränken muss. Auch beispielsweise bei Fragestellungen der automatischen Beweisführung kann sie hilfreich sein.
Im Verlauf dieser Arbeit haben wir uns auf nichtdeterministische Bottom-Up-Baumautomaten als einziges Mittel zur symbolischen Repräsentation von Mengen bschränkt. Dies erlaubt uns zwar, die Wertebereiche von Variablen sehr flexibel zu gestalten. Jedoch kommen in der Praxis häufig Probleme vor, bei denen die so erstellten Baumautomaten sehr viele Zustände besitzen, die ihrerseits nur wenig Terme erkennen. Dies resultiert in einer schlechten Performanz des Verfahrens. Möglicherweise ist es in der Zukunft sinnvoll zu untersuchen, ob für bestimmte funktionale Programme, bei denen nur spezielle Wertebereiche für Variablen vorkommen, eine Modellierung dieser Wertebereiche mithilfe anderer Formalismen wie beispielsweise BDDs möglich und sinnvoll ist. Dadurch könnte man Automaten mit symbolischen Zuständen erhalten, die selbst mehrere "entfaltete" Zustände repräsentieren können, und somit auch die Größe des Baumautomaten reduzieren.

Abbildungsverzeichnis

1.1 Erreichbarkeitsanalyse - Übersicht 11

2.1 Darstellung des Terms $f(g(c,x),y)$ aus Beispiel 1 als Baum. . . . 14
2.2 Darstellung der Termersetzung aus Beispiel 2. 15
2.3 Die Funktion REMOVEUNPRODUCTIVESTATES. 23

4.1 Algorithmus aus [Löd02]. 30
4.2 Iterationsschritt - Übersicht 34
4.3 TESTALREADYIN und GENERATETERMAUTOMATON 38
4.4 Symbolischer Algorithmus für $Post^*$ 39

5.1 Architektur des Prototypen 44
5.2 Automatenregeln in `Automaton` 47
5.3 Rückwärtsersetzung 68
5.4 Algorithmus UNIFY 69
5.5 Beispiel zu endlosen Rückwärtsersetzungsfolgen 71
5.6 Algorithmus GENERATECOUNTEREXAMPLE 72
5.7 Eingabedatei `append.rul` 73

6.1 Tabelle der Testfälle 80
6.2 Testergebnisse symbolischer/nichtsymbolischer Ansatz (Geschwindigkeit) 82
6.3 Schaubild `snoc.rul` 83
6.4 Laufzeitintensive Testfälle 86
6.5 Testergebnisse Backward-Bisimulation 90
6.6 Vergleich der Zustandszahlen von Endautomat mit maximaler Zustandszahl 92
6.7 Beeinflussung symbolischer Regeln 93

Literaturverzeichnis

[AHU75] AHO, ALFRED, JOHN E. HOPCROFT und JEFFREY ULLMAN: *The design and analysis of computer algorithms*. Addison-Wesley, 1975.

[AKH06] ABDULLA, PAROSH AZIZ, LISA KAATI und JOHANNA HÖGBERG: *Bisimulation Minimization of Tree automata*. Proc. CIAA'06-11th International Conference on Implementation and Application of Automata, 2006.

[ASS01] ABELSON, HAROLD, GERALD J. SUSSMAN und J. SUSSMAN: *Struktur und Interpretation von Computerprogrammen: Eine Informatik-Einführung (Springer-Lehrbuch)*. Springer, September 2001.

[BN98] BAADER, FRANZ und TOBIAS NIPKOW: *Term Rewriting and All That*. Cambdridge University Press, 1998.

[Bos95] BOSWORTH, RICHARD: *A Practical Course in Functional Programming Using ML*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[CDG+07] COMON, H., M. DAUCHET, R. GILLERON, C. LÖDING, F. JACQUEMARD, D. LUGIEZ, S. TISON und M. TOMMASI: *Tree Automata Techniques and Applications*. `http://www.grappa.univ-lille3.fr/tata`, 2007.

[CGP99] CLARK, EDMUND M., ORNA GRUMBERG und DORON A. PELED: *Model Checking*. The MIT Press, 1999.

[EHRS00] ESPARZA, JAVIER, DAVID HANSEL, PETER ROSSMANITH und STEFAN SCHWOON: *Efficient Algorithms for Model Checking Pushdown Systems*. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, Band 1855, Seiten 232–247. Springer, 2000.

[ES01] ESPARZA, JAVIER und STEFAN SCHWOON: *A BDD-based Model Checker for Recursive Programs*. In: BERRY, GÉRARD, HUBERT COMON und ALAIN FINKEL (Herausgeber): *Proceedings of CAV 2001*, Band 2102 der Reihe *Lecture Notes in Computer Science*, Seiten 324–336. Springer, July 2001.

[HMM07] HÖGBERG, JOHANNA, ANDREAS MALETTI und JONATHAN MAY: *Backward and Forward Bisimulation Minimisation of Tree Automata*. In: HOLUB, JAN und JAN ZDÁREK (Herausgeber): *Implementation and Application of Automata, 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers*, Band

4783 der Reihe *Lecture Notes in Computer Science*, Seiten 109–121. Springer, 2007.

[HU79] HOPCROFT, J.E. und J.D. ULLMAN: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[KS03] KRÜGER, GUIDO und THOMAS STARK: *Handbuch der Java-Programmierung*. Addison-Wesley, 2003.

[LÖ2] LÖDING, CHRISTOF: *Ground Tree Rewriting Graphs of Bounded Tree Width*. In: *STACS '02: Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, Seiten 559–570, London, UK, 2002. Springer.

[Löd02] LÖDING, CHRISTOF: *Model-Checking Infinite Systems Generated by Ground Tree Rewriting*. In: NIELSEN, MOGENS und UFFE ENGBERG (Herausgeber): *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, Band 2303 der Reihe *Lecture Notes in Computer Science*, Seiten 280–294. Springer, 2002.

[Löd06] LÖDING, CHRISTOF: *Reachability Problems on Regular Ground Tree Rewriting Graphs*. Theory Comput. Syst., 39(2):347–383, 2006.

[Mar05] MARYNS, HENDRIK: *On the Implementation of Tree Automata: Limitations of the Naive Approach*. In: *Proceedings of the Fifth Workshop on Treebanks and Linguistic Theories (TLT 2006)*, Prague, Czech Republic, 2005.

[MS72] MEYER, ALBERT R. und LARRY J. STOCKMEYER: *The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space*. In: *13th Annual Symposium on Switching and Automata Theory, 25-27 October 1972, The University of Maryland, USA*, Seiten 125–129. IEEE, 1972.

[PT87] PAIGE, ROBERT und ROBERT E. TARJAN: *Three partition refinement algorithms*. SIAM J. Comput., 16(6):973–989, 1987.

[SBSE07] SUWIMONTEERABUTH, DEJVUTH, FELIX BERGER, STEFAN SCHWOON und JAVIER ESPARZA: *jMoped: A Test Environment for Java Programs*. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, Band 4590 der Reihe *Lecture Notes in Computer Science*, Seiten 164–167. Springer, 2007.

[Web04] WEBER, TJARK ET AL.: *Isabelle/HOL Exercises*. `http://isabelle.in.tum.de/exercises/`, 2004.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

_____

(Andreas Gaiser)