Fachbereich Informatik und Mathematik
Institut für Informatik

# Bachelorarbeit

**Aybike Demirsan**

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

GOETHE UNIVERSITÄT
FRANKFURT AM MAIN

Fachbereich Informatik und Mathematik
Institut für Informatik

Bachelorarbeit

# Automatisierung der Induktion in Diagrammbeweisen zur Korrektheit von Programmtransformationen mittels Terminierungsbeweisern für Termersetzungssysteme

Aybike Demirsan

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

# Danksagung

Ich möchte an dieser Stelle allen danken, die mir während dieser Arbeit beigestanden haben. Ich danke Louisa, dass sie mich mit Essen versorgt hat, und ebenso Steffen, dass er mir immer Nahrung brachte, als ich keine Zeit mehr dazu fand - ohne euch wäre ich womöglich verhungert.

Ich danke Ben, meiner Schwester Sibel und meinem Vater Tayfun für das, was sie in mir sahen - eine Informatikerin, noch lange bevor ich eine war.

Großer Dank gilt meinem Betreuer David Sabel, der stets Antworten auf all meine Fragen hatte. Und zu guter Letzt möchte ich von ganzem Herzen Klaudia Kolodziej danken, die mir zu jedem Zeitpunkt beigestanden hat - mit dir scheint jedes Ziel erreichbar, und ich weiß nicht, wie ich dir jemals gebührend danken könnte.

Erklärung gemäß DPO §11 Abs. 11

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

Frankfurt, den 9. Juli 2012

Aybike Demirsan

# Inhaltsverzeichnis

**Abbildungsverzeichnis** iii

**1 Einleitung** 1
1.1 Aufbau . . . . . 2

**2 Grundlagen** 4
2.1 Termersetzungssysteme . . . . . 5
2.2 Standardreduktion und Transformation . . . . . 6
2.2.1 Standardreduktionen . . . . . 7
2.2.2 Transformationen . . . . . 7
2.3 Äquivalenz . . . . . 7
2.3.1 Der Kontext . . . . . 8
2.3.2 Andere Äquivalenzen . . . . . 8
2.4 Die Diagrammmethode . . . . . 9
2.4.1 Forking-Diagramme . . . . . 10
2.4.2 TRS und ITRS . . . . . 11
2.5 AProVE . . . . . 12
2.6 Haskell . . . . . 18

**3 Automatisierung der Diagrammmethode** 20
3.1 Innermost-Terminierung . . . . . 20
3.2 Umformungen . . . . . 21
3.2.1 Von ARS hin zu ERSARS . . . . . 22
3.2.2 Von ERSARS zu ITRS . . . . . 25

**4 Implementierung** 28
4.1 Übersetzung $K_M$ . . . . . 29
4.2 Plusse entfernen . . . . . 30
4.3 Hauptaufrufe . . . . . 31

**5 Tests und Ergebnisse** 33
5.1 Vollständiger Satz von Forking-Diagrammen für $i\mathcal{S}llet$ . . . . . 33
5.2 Vollständiger Satz von Forking-Diagrammen für $i\mathcal{S}, cp$ . . . . . 33
5.3 Tests eines Diagrammsatzes . . . . . 34
5.3.1 Transformation CP . . . . . 34
5.3.2 Transformation LLET . . . . . 36

**6 Zusammenfassung und Ausblick** **41**
6.1 Zusammenfassung . . . . . 41
6.2 Fazit . . . . . 42
6.3 Ausblick . . . . . 42

**Literaturverzeichnis** **43**

# Abbildungsverzeichnis


2.1 AProVE Eingabe eines TRS . . . . . . . . . . . . . . . . . . . . . . . 14
2.2 AProVE widerlegte Terminierung TRS . . . . . . . . . . . . . . . . . 15
2.3 AProVE widerlegte Terminierung ITRS . . . . . . . . . . . . . . . . . 16
2.4 AProVE bewiesene Terminierung . . . . . . . . . . . . . . . . . . . . 17

5.1 Haskell-Eingabe: Forking-Diagramme, Ausgabe: NARS . . . . . . . . . 36
5.2 Haskell-Eingabe: Forking-Diagramme, Ausgabe: TRS . . . . . . . . . . 39
5.3 AProVE-Eingabe (TRS) soll auf Terminierung überprüft werden . . . 40
5.4 Terminierung erfolgreich von AProve gezeigt . . . . . . . . . . . . . . 40

# Kapitel 1

# Einleitung

Das Analysieren und automatische Anpassen von Programmen ist in der Informatik und vor allem in der künstlichen Intelligenz ein wichtiger Aspekt mit zunehmend großer Bedeutung. Hierbei bedarf es einer fundierten Grundlage von Korrektheit und Terminiertheit, die es zu beweisen gilt.

Ein großes Ziel der Informatik ist es, korrekte und fehlerfreie Software zu implementieren. Dieses Ziel ist noch lange nicht erreicht, und die Folgen sind immens:
Immer wieder entstehen Unglücke von unerwartet großem Ausmaß aufgrund fehlerhafter Software. Diese Fehler und Ausfälle bei technischen Geräten jeglicher Art können zu Schaden von mehreren Hundert Millionen Euro führen - wie am Beispiel des Ariane-5-Raketenabsturz 1996 (siehe [OGL06]) zu sehen war - bis hin zu Opfern medizinischer Unfälle aufgrund von fehlerhafter Software medizinischer Geräte (siehe Therac-25 Vorfall, [Lev93]).
Durch zunehmende Verbreitung von Software und Technik im Alltag und in der Forschung wächst die Verantwortung auf Seiten der Programmierer, korrekte Software zu gewährleisten und den Anspruch auf Korrektheit zu erhöhen.

Um die Korrektheit von Software und speziell den Optimierungs- und Übersetzungsprozess innerhalb von Compilern zu überprüfen, ist es unerlässlich, einen Gleichheitsbegriff für Programme zu definieren und Methoden und Techniken zum Nachweis der Gleichheiten (und damit verbunden auch oft der Korrektheit) zu entwickeln. Ziel dieser Arbeit ist es, einen kleinen Beitrag zu diesem großen Fernziel der Informatik zu leisten.

In dieser Arbeit werde ich kontextuelle Gleichheit als Gleichheitsbegriff verwenden. Kontextuelle Gleichheit besitzen zwei Programme, wenn sie sich in allen Kontexten - also Programmstücken, in die sie eingesetzt werden - gleich verhalten, was für deterministische Programmiersprachen nur heißt, in gleichem Kontext zu terminieren, bzw. nicht zu terminieren. Diese kontextuelle Gleichheit ist unabhängig von der Programmiersprache, was sie noch ausdrucksstärker macht.
Der Korrektheitsbeweis bezieht sich im Folgenden auf Programmtransformationen. Sie stellen den Wechsel von einem Programm in ein anderes dar. Um zu zeigen, dass

eine Programmtransformation nicht korrekt ist, muss man lediglich einen Kontext finden, für den die Gleichheit in Bezug auf die Terminiertheit eines Programms vor und nach der Transformation differiert, also im gleichen Kontext das eine Programm terminiert, wobei das andere nicht terminiert.
Um jedoch die Korrektheit zu beweisen, ohne unendlich viele Kontexte untersuchen zu müssen, bedarf es eines Korrektheitsbeweises in Form von Induktion. Diese Beweismethode lässt sich auch für nicht-deterministische Programmiersprachen verwenden.

Ziel dieser Arbeit ist, die Automatisierung des Nachweises von kontextueller Gleichheit voranzutreiben. Genauer soll die Automatisierung der sogenannten Diagrammmethode vollendet werden. Die Diagrammmethode berechnet zunächst Diagramme und in einem zweiten Schritt werden diese Diagramme in einem Induktionsbeweis wie Ersetzungsregeln verwendet, um den Korrekheitsbeweis einer Programmtransformation zu erbringen. Diesen letzten Teil werde ich (aufbauend auf der Arbeit [RSSS12]) automatisieren, indem die Diagramme als Termersetzungssystem kodiert werden und im letzten Schritt mithilfe des Terminierungsbeweisers (AProVE) dieses Ersetzungssystem automatisch auf Terminierung geprüft wird. Wird bestätigt, dass das Termersetzungssystem terminiert, so besteht der Rückschluss, dass die Anwendung aller Diagramme, die für den Induktionsbeweis verwendet werden, terminiert. Somit ist die Umkodierung in das Termersetzungssystem korrekt.

Diese Form der Beweisführung kann vielseitig eingesetzt werden und ist auch von Bedeutung für automatische Korrektheitsbeweise von Transformationen in Compilern.

## 1.1 Aufbau

Ziel dieser Arbeit ist die Automatisierung der Induktion in Diagrammbeweisen zur Korrektheit von Programmtransformationen mittels Terminierungsbeweisern für Termersetzungssysteme.

Dafür werde ich zunächst einige Grundlagen zu dieser Thematik klären. In Kapitel 2 stelle ich zunächst Reduktionssysteme vor, auf die die Kodierungen abzielen. Hierbei stehen vor allem Termersetzungssysteme (im Folgenden auch *TRS* genannt) und ihre Variation, die *TRS* (engl.: *integer term rewrite system*), im Vordergrund. Der simple Aufbau der Termersetzungssysteme, bestehend Regelsystem auf einer Reduktionsfolge, definiere ich genauer in Unterkapitel 2.1.

Danach betrachten wir Standardreduktionen und Transformationen, die wir später auf unseren Diagrammen anwenden. Ihre Funktionsweise wird in Unterkapitel 2.2

erläutert.

Für die Definition der Transformationen ist der Begriff der Gleichheit entscheidend. Auf dieser Äquivalenz aufbauend zeigt sich die Korrektheit der Programmtransformationen, die anhand der Gleichheit der Programme vor und nach der Transformation gezeigt wird. Dafür verwenden wir den Begriff der kontextuellen Gleichheit, bzw. kontextuellen Äquivalenz, der sich nur auf die Gleichheit zweier Programme in Bezug auf ihre Terminierung beziehen. Dieser Gleichheitsbegriff wird in Unterkapitel 2.3 beschrieben.

Damit sind die grundlegenden Vorkenntnisse gesetzt, um das Prinzip der Diagrammmethode in 2.4 zu erläutern. Die Diagrammmethode zeigt die Korrektheit der Programmtransformationen.
Es wird ein vollständiger Satz von Digrammen erstellt, was bedeutet, dass jede auftretende Überlappung durch mindestens ein Diagramm erfasst ist. Es soll gezeigt werden, dass durch Anwedung dieser Diagramme sukzessive eine erfolgreiche Reduktionsfolge gebaut werden kann, die zeigt, dass der Term terminiert. Dabei muss gezeigt werden, dass auch das Anwenden dieser Diagramme beendet werden kann, und diesen Terminierungsbeweis übernimmt der Terminierungsbeweiser AProVE, dessen Fähigkeiten und Funktionsweise im Abschnitt 2.5 eingeführt werden.
Die Kodierungen der ursprünglichen Reduktionsfolgen bis hin zu den Termersetzungssystemen werden in dieser Arbeit mit der funktionalen Programmiersprache Haskell realisiert, die ich in Abschnitt 2.6 vorstelle.

Diese Umformungen zwischen verschiedenen Ersetzungssystemen, die schließlich als Eingabe in AProVE münden, werden in Kapitel 3 schrittweise anhand von Beispielen erklärt.

Die Implementierung dieser Umformungen und ihre Besonderheiten finden sich in Kapitel 4.
Zuletzt beschreibe ich meine Tests und Ergebnisse in Kapitel 5 und schließe die Arbeit mit dem daraus gezogenen Fazit in Kapitel 6.2.

# Kapitel 2

# Grundlagen

Meine Arbeit beschäftigt sich mit der Automatisierung der Induktion in Diagrammbeweisen, die die Korrektheit von Programmtransformationen für Termersetzungssysteme nachweisen sollen. In den folgenden Unterkapiteln werden die benötigten Grundlagen für das Verständnis der Implementierung vorgestellt und erläutert.

Erst werden Reduktionssysteme allgemein erläutert (Kapitel 2.1), dann auf Termersetzungssysteme und ihre Spezialform, die Integer Termersetzungssysteme eingegangen. Ihr Aufbau wird beschrieben und ihre Merkmale definiert.
Dann werden in Kapitel 2.2 anhand von Beispielen Standardreduktionen und Transformationen näher erklärt, die für das Verständnis der Diagrammmethode elementar sind.
Um die Korrektheit einer Programmtransformation nachzuweisen, bedarf es lediglich kontextueller Äquivalenz (keine Gleichheit des gesamten Programmstücks in seiner Funktionsweise), deren Bedeutung in diesem Kapitel unter 2.3 erläutert wird.
Der Terminierungsbeweis bezieht sich auf Eingaben von Diagrammen, die in ITRS umgewandelt werden, um dann als Eingabe dem Terminierungsbeweiser AProVE übergeben zu werden. Die Darstellung solcher Diagramme und ihre Transformation von ARS-Form in NARS-Form und im letzten Schritt in Termersetzungssysteme werden in Kapitel 2.4 vorgestellt.
Im darauffolgenden Kapitel 2.5 werden der Terminierungsbeweiser AProVE und seine Fähigkeiten und Funktionsweisen vorgestellt und anhand eines Beispiels verdeutlicht, wie die umgewandelte Eingabe im TRS, bzw. ITRS Format von AProVE bearbeitet wird.

Zuletzt führt Kapitel 2.6 die funktionale Programmiersprache Haskell ein und zeigt ihre Besonderheiten sowie ihren Einsatz als Eingabe im Terminierungsbeweiser auf, da in Kapitel 4 die schrittweise Umwandlungen von ARS (oder NARS) in TRS (oder ITRS) in Haskell implementiert wurden.

## 2.1 Termersetzungssysteme

Reduktionen sind Berechnungen, die in der Informatik eine große Rolle spielen. Man kann sie als sukzessive Berechnungen verstehen, die von einem Zustand in einen anderen führen.
Sie vereinfachen Terme, werden für die Normalformberechnung von Polynomen benötigt und für die Auswertung von Booleschen Ausdrücken verwendet, dabei entsprechen die einzelnen Reduktionsschritte gerade elementaren Berechnungen nach bestimmten Regeln.

Betrachten wir zunächst Terme. Terme sind wiederum syntaktisch korrekt gebildete Worte aus Variablen und Konstantensymbolen, die sich ausrechnen, nach bestimmten Regeln umformen und ineinander einsetzen lassen.
Formal: Eine *Signatur* $\Sigma$ ist eine endliche Menge von Funktionssysmbolen. Jedes Funktionssymbol $f \in \Sigma$ hat dabei eine feste Stelligkeit $ar(c)$. Für eine Signatur $\Sigma$ und eine unendliche Menge von Variablen $V$ ist die Menge der *Terme* $T(\Sigma, V)$ induktiv durch die folgenden Regeln definiert:

- jede Variable $x \in V$ ist auch ein Term (d.h. $x \in T(\Sigma, V)$)
- wenn $t_1, \ldots, t_n$ Terme sind und $f$ ein $n$-stelliges Funktionssymbol ist, dann ist auch $f(t_1, \ldots, t_n)$ ein Term.

Betrachten wir nun konkrete Reduktionsfolgen (RS).

**Definition 2.1** (Definition 3.1 aus [RSSS12]). *Sei* $\{\overset{T_1}{\Longrightarrow}, \ldots, \overset{T_k}{\Longrightarrow}\}$ *eine Menge von Transformationen. RS bestehen aus einer Folge von Elementen aus* $\overset{sr}{\Longleftarrow} \cup \bigcup_{1 \leq i \leq k}(\overset{T_i}{\Longrightarrow} \times \{i\}) \cup \{(a, a, \underline{id}) \mid a \in \mathcal{A}\}$*, wobei* $(a, a, \underline{id})$ *das höchstwertige Element ist und aufeinander folgende Elemente* $(e_1, e_2, d)(e_3, e_4, d')$ *nur zulässig sind, wenn* $e_2 = e_3$*.*

Allgemein ist ein Regelsystem R eine endliche Menge von Regeln, die die Reduktionsrelation $\overset{R}{\to}$ auf den Termen definiert.
In dieser Arbeit besteht das Regelsystem aus Ersetzungsregeln auf einer RS. Das sind Regeln in der Form $L \rightsquigarrow R$, wobei L und R RS sind.
Für eine Menge von Termersetzungsregeln $R = \{l_1 \to r_1, \ldots, l_n \to r_n\}$ ist die *Rewriting Relation* $\to_R$ formal definiert durch:

Sei $t$ ein Term und $t'$ ein Unterterm von $t$ an der Position $p$. Wenn es eine Instanzierung $\sigma$ gibt, so dass $\sigma(l_i) = t'$ gilt, dann $t \to_R s$, wobei $s$ der Term ist, der aus $t$ entsteht, indem man den Unterterm $t'$ and der Position $p$ durch $\sigma(r_i)$ ersetzt.

Für Termersetzungsregeln $R$ sagt man, $\to_R$ ist *terminierend*, wenn es keine unendliche lange Folge $t_1 \to_R t_2 \to_R \ldots$ (für beliebige Terme $t_i$) gibt.

In dieser Arbeit werden abstrakte Reduktionsfolgen (ARS) zum Schlussfolgern verwendet. Durch die ARS werden die konkreten Ausdrücke wegabstrahiert, was bedeutet, dass in den Reduktionssfolgen nur die beschrifteten Pfeile auftauchen, die Terme dazwischen sind jedoch in ARS nicht enthalten. Die Konstante A repräsentiert Antworten der Standardreduktionen und Transformationen.

Reduktionssysteme sind in der einfachsten Form Mengen von Objekten zusammen mit einer binären Relation. Dabei gilt zu untersuchen, ob die Reihenfolge der einzelnen elementaren Regeln beachtet werden muss, ob die Reduktion terminiert und ob das Ergebnis der Reduktion eindeutig ist, bzw. die Normalform vorliegt.
RSRS Ersetzungssysteme (engl.: rewrite system on reduction sequences) bestehen aus Reduktionsfolgen (RS) und einem Regelsystem $\rightarrow_R$ der Form $L \rightsquigarrow R$.

Als Untergruppe von Reduktionssystemen betrachten wir Termersetzungssysteme. Jene erlauben nur Regeln der Form $s \rightarrow t$, wobei s und t Terme sind. Eine Regel ist ein Paar $(s, t)$ von Termen und $FV(t) \subseteq FV(s)$, wobei $FV(x)$ die Menge der Variablen in einem Term darstellt.

Eine Erweiterung von Termersetzungssystemen sind die ITRS, die keine TRS sind, da in ITRS auf der rechten Seite der Termersetzungsregeln freie Variablen stehen dürfen, was in TRS nicht erlaubt ist. Diese freien Variablen werden dafür verwendet, Bedingungen (Constraints) aufzustellen. Freie Variablen stehen dabei für beliebige Zahlenkonstanten und dürfen vom Termersetzungssystem quasi geraten werden. Somit kann die Terminierung für die nun endlich vielen Reduktionen gezeigt werden.

Wenn Strukturen, z.B. TRS, keine unendliche Schachtelung von immer größeren Unterstrukturen enthalten können, sind sie noethersch. Für ITRS - allgemein für Ersetzungssysteme - ist der Terminierungsbegriff analog zum Terminierungsbegriff von TRS definiert: Es darf keine unendlich langen Folgen von Ersetzungsschritten geben. In diesem Kontext ist der Begriff des noetherschen TRS daher mit Terminiertheit gleichzusetzen, da das Reduktionssystem demnach nach endlich vielen Schritten terminiert.

## 2.2 Standardreduktion und Transformation

Dieser Abschnitt widmet sich Standardreduktionen und Transformationen.

### 2.2.1 Standardreduktionen

Definieren wir zunächst Standardreduktionen. Standardreduktionen reduzieren einen Ausdruck zu einem anderen. Dabei gibt es unterschiedliche Reduktionen, die durch Labels benannt werden.
Sei $Z$ die Menge der Ausdrücke (oder auch Programme) und $L$ die Menge der Labels. Dann sei $\xrightarrow{sr} \subseteq Z \times Z \times L$ die Standardreduktion. Die Menge der Labels enthält genau die Namen der Standardreduktionen. Für eine Reduktion mit Label schreiben wir $\xrightarrow{sr,l} \subseteq \xrightarrow{sr}$.

Die Auswertung eines Ausdrucks $z \in Z$ ist eine Folge von Standardreduktionen bis zu einer Antwort $a \in A \subseteq Z$. Wenn eine solche Antwort, also $z \xRightarrow{sr,*} a$, existiert, dann sagen wir, dass $z$ konvergiert. Die Konvergenz schreiben wir dann als $z \Downarrow$.
Existiert eine solche Antwort nicht, dann sagen wir: $z$ divergiert. Und schreiben $z \Uparrow$.

Durch diese sukzessiven Überführungen von einem Ausdruck in den nächsten ist die operationale Semantik in Form der Standardreduktion definiert.

### 2.2.2 Transformationen

Programmtransformationen transformieren ein Programm in ein anderes, sie sind also binäre Relationen auf Programmen. Diese Transformation gilt als korrekt, wenn das Programm vor und nach der Transformation gleich ist. Gleich ist hierbei unterschiedlich zu definieren, und für unsere Zwecke verwenden wir die kontextuelle Äquivalenz $\sim_c$ als Gleichheitsbegriff. Näheres zum Begriff der Äquivalenz folgt in Abschnitt 2.3.
Sei $\xrightarrow{T}$ eine Programmtransformation. $\xrightarrow{T} \subseteq Z \times Z$ ist also eine binäre Relation auf Ausdrücken. $\xrightarrow{T}$ wird *korrekt* genannt, wenn $\xrightarrow{T} \subseteq \sim_c$.

## 2.3 Äquivalenz

Wir führen den Begriff der kontextuellen Äquivalenz ein. Mit ihm lassen sich zwei Programme miteinander vergleichen, was für die Programmtransformationen relevant ist.
Dieser Begriff der kontextuellen Gleichheit wird bei Compilern relevant, bei denen Programmstücke miteinander verglichen werden. Sind beide Programmstücke kontextuell gleich, so ist die Korrektheit der Transformation, nämlich die Gleichheit des Programms vor und des Programms nach der Transformation, gewährleistet.
Dabei wird das Leibnitzsche Prinzip berücksichtigt, welches besagt, dass Gleiches nur mit Gleichem ersetzt wird.

Gleich bedeutet in diesem Kontext: Kontextuelle Äquivalenz oder Gleichheit besteht zwischen zwei Programmen, wenn sie sich in allen Kontexten gleich verhalten. Kontext bedeutet in diesem Fall ein Programm mit einem Loch, sodass ein Programm in dieses Loch eingesetzt werden kann.

### 2.3.1 Der Kontext

Bei deterministischen Programmiersprachen genügt es, zu überprüfen, ob sich beide Programme im gleichen Kontext gleich in Bezug auf ihre Terminierung verhalten. Also sind zwei Programme kontextuell nicht gleich, wenn es einen Kontext C gibt, der sie in ihrer Terminierung unterscheidet. Beispielsweise sind *True* und *False* nicht kontextuell gleich, weil man den folgenden Kontext C angeben kann:

C = if [.] then Endlosschleife else 0

Im Kontext C ist [.] das Loch, das durch Programme ersetzt wird. Somit terminiert das Programm (der Kontext) bei Ersetzung durch *True* nicht, bei Ersetzung durch *False* allerdings schon. Somit ist die kontextuelle Gleichheit für *True* und *False* widerlegt.

Insgesamt lässt sich die Kontextuelle Gleichheit daher definieren als:

**Definition 2.2** (Definition 2.3 aus [RSSS12]). *Die Kontextuelle Präordnung* $\leq_c$ *ist definiert als*

$$e_1 \leq_c e_2 \text{ genau dann, wenn für alle Kontexte } C : C[e_1]\Downarrow \Longrightarrow C[e_2]\Downarrow$$

*Zwei Ausdrücke* $e_1, e_2$ *sind kontextuell äquivalent* ($e_1 \sim_c e_2$)*, wenn sowohl* $e_1 \leq_c e_2$ *als auch* $e_2 \leq_c e_1$ *gilt.*

Der Nachweis Kontextueller Gleichheit kann sehr schwer sein, weil man alle Kontexte betrachten muss, was im Allgemeinen unendlich viele sind.

### 2.3.2 Andere Äquivalenzen

Kontextuelle Gleichheit lässt sich auch durch andere Methoden ermitteln. Beispiele hierfür wären passende Bisimulation, eine adäquate denotationale Semantik, logische Relationen und die axiomatische Semantik.

Die Bisimulation ist eine binäre Relation, die Zustände miteinander in Beziehung setzt, die sich gleich verhalten - kann also auch zwei Programme in eine Relation

setzen und somit verschiedene Formen von Gleichheit nachweisen. Bisimilar sind zwei Programme, wenn man sie durch Testen nicht unterscheiden kann. Bisimulation funktioniert als Beweismethode ohne Kontexte, sie kann somit unter anderem - je nach Definition der Bisimulation - zeigen, dass zwei Programme kontextuell gleich sind. Sie funktioniert allerdings ohne Kontexte und die Implikation gilt manchmal nur in die Richtung, dass Bisimulation kontextuelle Gleichheit impliziert, aber nicht umgekehrt. Selbst eine solche Bisimulation kann in syntaktisch komplexeren Sprachen schwer zu definieren sein.

Bei der denotationalen Semantik werden für alle Zustände durch Funktionen die Folgezustände definiert. Es gilt: Wenn zwei solcher Funktionen gleich definiert sind, also die gleichen Folgezustände haben, sind sie auch kontextuell gleich.
In dem Fall wurde eine adäquate denotationale Semantik gefunden. Man kann mit ihr also unter Umständen die kontextuelle Gleichheit nachweisen, allerdings lässt sie sich damit oft nicht widerlegen. Wenn sich Denotation und kontextuelle Gleichheit biimplizieren, spricht man von einer voll abstrakten denotationalen Semantik.

Logische Äquivalenz besteht, wenn zwei logische Ausdrücke den gleichen Wahrheitswert besitzen und axiomatische Semantik macht Aussagen über einzelne Eigenschaften von Programmen, ist also unter Umständen auch ein möglicher Ansatz zum Ermitteln kontextueller Gleichheit.

Die kontextuelle Gleichheit wird in dieser Arbeit dafür verwendet, die Korrektheit von Programmtransformationen zu zeigen, also die kontextuelle Gleichheit des Programms vor und nach der Transformation. Nur so kann gewährleistet werden, dass die letztendliche Terminierung, die für die ITRS, bzw. die TRS, gelten, auch für die ursprünglichen Diagramme gelten.

## 2.4 Die Diagrammmethode

Im Folgenden führe ich die Diagrammmethode ein, mit der die Korrektheit der Programmtransformationen bewiesen wird und die anschließend in Termersetzungssysteme umkodiert werden.
Die Diagrammmethode ist eine syntaktische Methode zum Nachweis der Korrektheit einer Programmtransformation. Wird die Korrektheit einer Programmtransformation T gezeigt, so bedeutet das, dass alle Paare in T kontextuell gleich sind.

Sei $\xrightarrow{T}$ eine Programmtransformation, eine binäre Relation auf Ausdrücken.
Man erstellt als ersten Schritt einen vollständigen Satz von Diagrammen, was bedeutet, dass jedes mögliche Auftreten von kritischen Paaren durch mindestens ein Diagramm erfasst wird.
Kritische Paare sind dabei die beiden Ausdrücke, die nach dem Anwenden der

Transformations- und der Standardreduktionsregel entstehen, also wäre in folgendem Fall

$$\begin{array}{ccc} e_1 & \xrightarrow{T} & e_2 \\ {\scriptstyle sr}\downarrow & & \\ e_3 & & \end{array}$$

das kritische Paar $(e_2, e_3)$. Diese kritischen Paare werden auch Überlappungen genannt.
Eine Überlappung zwischen eine Standardreduktion und einer Transformation ist also ein Ausdruck, auf den beide Regeln anwendbar sind.

Der nächste Schritt ist, für alle Ausdrücke $e_1, e_2$ und Kontexte $C$ unter $C[e_1] \xrightarrow{T} C[e_2]$ zu zeigen: Das Programm $C[e_1]$ konvergiert, genau dann, wenn das Programm $C[e_2]$ konvergiert. Die Umkehrrichtung - wenn $C[e_2]$ konvergiert, dann auch $C[e_1]$ - kann durch Umdrehen der Transformation behandelt werden, denn aus Symmetriegründen ist das Verfahren für diesen Fall analog.
Dafür betrachten wir eine konvergierende Reduktionsfolge für $C[e_1]$ und konstruieren per Induktion eine konvergierende Reduktionsfolge für $C[e_2]$, wobei wir die Forking-Diagramme als Termersetzungssystem verwenden.

### 2.4.1 Forking-Diagramme

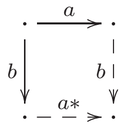
Forking-Diagramme (von engl. to fork = gabeln) sind Ersetzungsregeln $\text{L} \rightsquigarrow \text{R}$ auf abstrakten Reduktionsfolgen (ARS). Dabei kann die gegebene Reduktionsfolge L in die Reduktionsfolge R überführt werden. L und R bestehen dabei aus Standardreduktionen und Transformationen. Forking-Diagramme können auch in linearer Schreibweise dargestellt werden. Im Folgenden sieht man unser vorangegangenes Beispieldiagramm, ergänzt um die Reduktionsfolge R. Und dieses Forking-Diagramm ist rechts überführt in die entsprechende lineare Schreibweise zu sehen.

$$\begin{array}{ccc} e_1 & \xrightarrow{T} & e_2 \\ {\scriptstyle sr}\downarrow & & \downarrow{\scriptstyle sr} \\ e_3 & \xrightarrow{T} & e_4 \end{array} \quad \Longleftrightarrow \quad e_3 \xleftarrow{sr} e_1 \xrightarrow{T} e_2 \rightsquigarrow e_3 \xrightarrow{T} e_4 \xleftarrow{sr} e_2$$

Mit diesen Forking-Diagrammen will man die Transformationsschritte von der rechten auf die linke Seite schieben. Man wendet die Forking Diagramme also beispielsweise auf $A \xleftarrow{sr} \dots \xleftarrow{sr} . \xrightarrow{T}$ so lange an, bis man $A \xrightarrow{T} \dots \xrightarrow{T} . \xleftarrow{sr} \dots \xleftarrow{sr}$ erhält.
Mit Antwortdiagrammen will man die Transformationsschritte gänzlich eliminieren.

Wenn man die Antwortdiagramme ausreichend anwendet, erhält man eine (abstrakte) Reduktionsfolge der Form $A \xleftarrow{sr} \ldots \xleftarrow{sr}$, die die Konvergenz von $C[e_2]$ (s.o.) bezeugt.

Betrachten wir das folgende Beispieldiagramm.




## 2.4.2 TRS und ITRS

Gehen wir von Abschnitt 2.4.1 und den dort erwähnten Beispielen direkt über zu der Kodierung der TRS: $a*$ steht für die transitive Hülle von $\xrightarrow{a}$ Reduktionspfeilen, also beliebig viele oder keine $\xrightarrow{a}$ Reduktionspfeile. Um eine erlaubte TRS-Regel zu kodieren, nimmt man die aus dem Diagramm lesbare Reduktionsregel

$a(b(x)) \to b(a * (x))$

und erstellt aus ihr das TRS mit folgenden Regeln:

$a(b(x)) \to b(AStern(x))$
$AStern(x) \to a(AStern(x))$
$AStern(x) \to x.$

Dieses TRS terminiert nicht, da die zweite Regel unendlich oft anwendbar ist. Daraus entsteht die Notwendigkeit der ITRS (integer term rewrite system). Bei ITRS sind Zahlenvariablen und Bedingungen erlaubt. Somit kann man eine feste Zahl größer null auf der rechten Seite der Reduktionsregel raten und somit die Terminierung gewährleisten. In diesem Fall sähe das ITRS folgendermaßen aus:

$a(b(x)) \to b(AStern(y, x))$ mit y > 0
$AStern(n, x) \to a(AStern(n - 1, x))$ für n > 0
$AStern(0, x) \to x$

Aufgrund der transitiven Hüllen in den Diagrammen bedarf es in meiner Arbeit ITRS. Die Diagramme sind in ihrer Semantik nämlich in den folgenden Übersetzungen definiert (siehe [RSSS12] für nähere Details):
J(D) ergeben SRSARS *(simple rewriting system on abstract reduction sequences)*, die

keine Variablen oder transitive Hüllen besitzen, aber unter Umständen unendlich viele Regeln. Damit besteht der Bedarf nach einer Kodierung von NARS (die Übersetzung K(D) ), der *abstract reduction sequence with natural numbers.*

Dafür wurden für die Automatisierung des Terminierungsbeweises die ERSARS, die *encoded rewriting system on abstract reduction sequences*, eingeführt. Mit ihnen können ITRS kodiert werden, die zuvor aus NARS erstellt wurden.

In [RSSS12] wurde gezeigt: Wenn die Übersetzung J(D) für einen vollständigen Satz von Diagrammen für eine Transformation T leftmost terminierend ist, dann ist die Transformation T Konvergenz erhaltend.
Auch für die Übersetzung K(D) für die Diagramme gilt: Wenn die daraus entstehenden ERSARS leftmost terminierend sind, dann ist die Transformation T Konvergenz erhaltend.

Wir haben beim Nachweis der Terminierung also zwei Fälle, die wir unterscheiden:
Wir haben ein ARS, das keine Variablen oder transitive Hüllen besitzt. Es lässt sich als Diagramm darstellen, was nichts anderes ist als Ersetzungsregeln auf den ARS. Diese Diagramme lassen sich als SRSARS beschreiben, welche wir dann als ein TRS kodieren. Dieses TRS übergeben wir - mit gewähltem Eingabeformat TRS - unserem Terminierungsbeweiser AProVE. Zeigt jenes die Innermost-Terminierung für das TRS, dann ist die Innermost-Terminierung des SRSARS gezeigt und die Transformation ist Konvergenz erhaltend.
Es ist ersichtlich, dass sich ein ITRS auch in SRSARS kodieren lässt, wenn es keine transitiven Hüllen im ITRS gibt.

In unserem zweitem Fall haben wir ein ARS mit transitiven Hüllen. Dieses kodieren wir um in ein NARS. Die NARS lassen sich als ERSARS beschreiben - die Übersetzung K(D) für Diagramme ergibt eben ein ERSARS. Diese ERSARS lassen sich wiederum als ITRS darstellen, die wir dann in unseren Terminierungsbeweiser geben (diesmal entsprechend mit Eingabeformat „ITRS“). Zeigt der Terminierungsbeweiser AProVE die innermost Terminierung des ITRS, so entspricht dies der leftmost Terminierung des ERSARS und wieder ist gezeigt, dass die Transformation konvergenz-erhaltend ist.

## 2.5 AProVE

AProVE ist ein Werkzeug zum automatischen Nachweisen der Terminierung von Termersetzungssystemen und ein Akronym für „Automated Program Verification Environment“.
Es wird an der RWTH Aachen entwickelt und hat die Terminierungsanalyse bisher

für die Programmiersprachen Prolog und Haskell implementiert.
Die Terminierungsbeweise für objektiorientierte Sprachen wie Java sind aktueller Forschungsgegenstand.

Zur Entstehung von AProVE wurde die Entwicklungsumgebung Eclipse verwendet, sowie MiniSAT 2, einem ausgezeichneten SAT-Solver für das Lösen einiger Kodierungen von Suchproblemen.
Die Java Bibliothek SAT4J für Erfüllbarkeit wird ebenfalls für SAT-kodierte Suchprobleme verwendet, sowie für die Umwandlung von Aussagenlogischen Formeln in Konjunktive Normalformen durch Tseitins Algorithmus. Weitere von AProVE benutzte Softwaresysteme, die AProVEs Funktionsweise ermöglichen, sind ANTLR, GNU Getopt, Jdotty, Piccolo, SableCC und Yices.
AProVE kann für normale Termersetzungssysteme sowohl die Innermost Terminierung als auch die Terminierung ohne Strategie zeigen. Für meine Zwecke ist nur die innermost Terminierung relevant, da nur diese für ITRS (siehe Abschnitt 2.1) gezeigt werden kann, welche ausschließlich für meine Kodierung gebraucht wird.

Betrachten wir erneut das Beispiel-TRS und das daraus resultierende ITRS aus Kapitel 2.4. Das Beispiel-TRS war folgendes:

$a(b(x)) \to b(AStern(x))$
$AStern(x) \to a(AStern(x))$
$AStern(x) \to x.$

In AProVE Syntax entspricht die TRS Eingabe:

$(VAR\ x)$
$(RULES$
$a(b(x)) \to b(AStern(x))$
$AStern(x) \to a(AStern(x))$
$AStern(x) \to x)$

Diese Eingabe übergeben wir AProVE, siehe nachfolgende Abbildung 2.1.

Abbildung 2.1: AProVE Eingabe eines TRS

Übergeben wir dieses TRS an AProVE (im Eingabeformat „Term Rewrite System (WST)“, so widerlegt AProVE die Terminierung, wie zu erwarten war, da die Ersetzungsregel $AStern(x) \to a(AStern(x))$ unendlich oft angewandt werden kann.

0.26
**Termination w.r.t. Q** of the given *QTRS* could be disproven:

↳ **QTRS**

↳ `QTRS Reverse`

↳ `NonTerminationProof`

Q restricted rewrite system:
The TRS R consists of the following rules:

a(b(x)) → b(AStern(x))
AStern(x) → a(AStern(x))
AStern(x) → x

Q is empty.

We applied the QTRS Reverse Processor [REVERSE].

We used the non-termination processor [OPPELT08] to show that the SRS problem is infinite.

Found the self-embedding DerivationStructure:

AStern → a AStern

Abbildung 2.2: AProVE Ausgabe für eine widerlegte Terminierung des TRS

Die widerlegte Terminierung wird in AProVE begründet: Die unendlich oft anwendbare Regel $AStern(x) \rightarrow a(AStern(x))$ des Regelsystems wurde als Grund gefunden, siehe Abbildung 2.2.

Beheben wir diese Ursache und begrenzen die Anzahl der Regelanwendungen von $AStern(x) \rightarrow a(AStern(x))$, entsteht folgendes ITRS:

| | |
|---|---|
| $a(b(x)) \rightarrow b(AStern(y, x))$ | mit y > 0 |
| $AStern(n, x) \rightarrow a(AStern(n-1, x))$ | für n > 0 |
| $AStern(0, x) \rightarrow x$ | |

Verwendet man wieder das Eingabeformat „Term Rewrite System (WST)“ von AProVE, ist die entsprechende Eingabe des ITRS in AProVE Syntax:

$(VAR\ x\ y\ n)$
$(RULES$
$a(b(x)) \to b(AStern(y, x))$
$AStern(n, x) \to a(AStern((n-1), x)) \quad :|: n > 0$
$AStern(0, x) \to x)$

Und AProVE widerlegt die Terminierung, wie man in Abbildung 2.3 sehen kann, da eine Variable auf der rechten Seite im ITRS nicht wie in einem ITRS interpretiert wird, da das Eingabeformat „Term Rewrite System (WST)“ gewählt wurde.

0.19
**Termination** of the given *GTRS* could be disproven:

↳ **GTRS**
  ↳ CritRuleProof

Generalized rewrite system (where rules with free variables on rhs are allowed):
The TRS R consists of the following rules:

a(b(x)) → b(AStern(y, x))
AStern(n, x) → a(AStern, x)

The rule a(b(x)) → b(AStern(y, x)) contains free variables in its right-hand side. Hence the TRS is not-terminating.

Abbildung 2.3: AProVE Ausgabe für eine widerlegte Terminierung eines ITRS durch Interpretation als TRS

Ändert man nun den Eingabetyp von „Term Rewrite System (WST)“ zu „Integer Term Rewrite System“, so kann für die gleiche Eingabe Terminierung nachgewiesen werden. Die Ausgabe des Terminierungsbeweisers AProVE ist in Abbildung 2.4 zu sehen. Dass die Terminierung durch Änderung der Eingabeform nun von AProVE als bewiesen dargestellt wird, bezieht sich aber nur auf die sogenannte Innermost Ersetzungsrelation, die für ITRS verwendet wird. Sie eingeschränkter ist als die Ersetzungsrelation ohne Strategie und ist im Allgemeinen einfacher nachzuweisen als die allgemeine Terminierung (ohne Strategie). AProVE beschränkt sich also für ITRS auf den Nachweis dieses Terminierungsbegriffs.

1.14
**Termination** of the given *ITRS* could be proven:

↳ ITRS
  ↳ ITRStoIDPProof

ITRS problem:

The following function symbols are pre-defined:

!= ~ Neq: (Integer, Integer) -> Boolean
* ~ Mul: (Integer, Integer) -> Integer
>= ~ Ge: (Integer, Integer) -> Boolean
-1 ~ UnaryMinus: (Integer) -> Integer
| ~ Bwor: (Integer, Integer) -> Integer
/ ~ Div: (Integer, Integer) -> Integer
= ~ Eq: (Integer, Integer) -> Boolean
~ Bwxor: (Integer, Integer) -> Integer
|| ~ Lor: (Boolean, Boolean) -> Boolean
! ~ Lnot: (Boolean) -> Boolean
< ~ Lt: (Integer, Integer) -> Boolean
- ~ Sub: (Integer, Integer) -> Integer

Abbildung 2.4: Ausschnitt der AProVE Ausgabe für eine bewiesene Terminierung, Quelle: [JGT12, AProVE]

Für die allgemeine Rewriting-Relation gilt nämlich: $t \xrightarrow{R} t'$, wobei t von der Form $C[\sigma(l)]$ ist, C ist ein Kontext, $\sigma$ eine Substitution, und $l$ eine linke Seite der Regel und $t' = C[\sigma(r)]$, wobei $r$ die rechte Seite der Regel ist.

**Definition 2.3** (Definition 2.8 aus [SS12])**.** *Eine Substitution $\sigma$ ist dabei eine Funktion $\sigma : V \rightarrow Term\ (F, V)$ mit $\sigma(x) \neq x$ für endlich viele $x \in V$. $F$ ist dabei eine endliche Menge von Operatorsymbolen und $V$ steht für die Variablensymbole.*

Für die Innermost-Rewriting-Relation gelten die Regeln der Rewriting-Relation, mit einer zusätzlichen Einschränkung: Kein echter Unterterm von $\sigma(l)$ ist reduzierbar mit $\xrightarrow{R}$.

Beispiel: Sei $f(a) \rightarrow f(b)$ eine Reduktionsfolge mit einem Regelsystem bestehend aus den beiden Regeln $a \rightarrow b$ und $f(a) \rightarrow f(a)$. Dann würde $f(a)$ mit der Innermost-Rewriting-Relation zuerst die Regel $a \rightarrow b$ anwenden und somit zu $f(a) \rightarrow f(b)$ gelangen, was nicht weiter reduzibel ist.

Zur Funktionsweise von AProVE zählen direkte Terminierungsbeweise wie beispielsweise die Knuth-Bendix-Ordnung (Knuth-Bendix-order), der rekursiven Pfadordnung (recursive path order) und der polynomiellen Ordnung (polynomial order). Um die Fähigkeiten von AProVE zu erhöhen, wurden aber auch andere Techniken verwendet, wie die „dependency pair technique“, die einen Terminierungsbeweis ermöglichen, wo die anderen klassischen Techniken scheitern würden.
Da AProVE mithilfe dieser kombinierten Techniken das erfolgreichste Tool für Terminierungsbeweise von Termersetzungssystemen von 2004 - 2011 beim jährlichen „International Competition of Termination Tools “war, findet es in meiner Arbeit Verwendung als Terminierungsbeweiser.

AProVE arbeitet nicht nur mit Eingaben von Termersetzungssystemen, sondern auch mit Prolog, Haskell und Java Bytecode und außerdem mit Variationen von Termersetzungssytemen. Dabei werden die Eigenheiten der einzelnen Programmiersprachen berücksichtigt, wie die Auswertungsreihenfolge, die bei Haskell verzögert ist. Somit terminiert sie beispielsweise nicht für

```
[1..]
```

aber für

```
take 10 [1..]
```

schon, trotz der unendlichen Liste als Argument, da Argumente nur ausgewertet werden, wenn sie benötigt werden (sogenannte „call-by-need“ Auswertung). Dass diese Eigenschaft Haskells berücksichtigt wird, ist in unserem Kontext des Terminierungsnachweises relevant. So ist sichergestellt, dass der Terminierungsbeweiser bestehende Terminierung für einen Ausdruck erkennt, wenn sie in der entsprechenden Programmiersprache besteht.

## 2.6 Haskell

Mit Haskell werden in dieser Arbeit die Eingaben für AProVE erstellt. Haskell ist eine rein funktionale Programmiersprache, die auf dem Lambda-Kalkül basiert, welches als formale Sprache Funktionen und Parameter definiert und die Auswertungsregeln der Parameter beschreibt. Als funktionale Sprache besteht sie nur aus Funktionen, also ohne Prozeduren oder Methoden. Imperative Konstrukte wie Schleifen werden durch Rekursionen ersetzt.
Funktionen können normale Werte, Parameter anderer Funktionen oder Ergebnisse von Funktionsanwendungen sein.

Funktionen werden ohne Nebeneffekte berechnet, Werte werden also verändert zurückgegeben, doch der Zustand des Programms wird nicht verändert, was Programmbeweise deutlich vereinfachen, was vor allem für das Thema dieser Arbeit relevant ist. Gleiche Werte angewendet auf die gleiche Funktion liefern immer das gleiche Ergebnis. Diese Eigenschaft wird auch referentielle Transparenz genannt.

Operationen können keine Variablenwerte verändern und die Auswertungsreihenfolge von Haskell ist nicht strikt, daher existiert eine Implementierung von Haskell, dessen Auswertungsreihenfolge verzögert ist. So werden Ausdrücke nur bei Bedarf ausgewertet. Somit terminieren in Haskell dank dieser Auswertungsreihenfolge bestimmte Funktionsaufrufe, die ein nicht terminierendes Argument beinhalten.

Beispiel:

```
first x y = x
f = [x | x <- [1..]]

*Main> first 3 f
3
```

Da kein Bedarf besteht, wird das zweite Argument erst gar nicht ausgewertet und der Aufruf terminiert.
Pattern Matching ist die fallweise Definition von Funktionen, beispielsweise für Fakultät:

```
fakultaet 0 = 1
fakultaet x = x * (fakultaet (x-1))
```

In Haskell sind Typvariablen erlaubt, was die statische Typisierung etwas vereinfacht. Durch die statische Typisierung müssen alle Datentypen zur Laufzeit bereits definiert sein, sie werden zur Compilezeit überprüft. Dies hat den Vorteil, dass zur Laufzeit keine Typfehler entstehen.
Außerdem ermöglicht Haskell benutzerdefinierte Datentypen, die mit Konstruktoren definiert werden. Somit lässt sich beispielsweise der Datentyp Monat folgendermaßen definieren:

```
data Tag = Montag | Dienstag | Mittwoch | Donnerstag | Freitag
| Samstag | Sonntag
```

Somit lassen sich auch Datentypen definieren, die selbst rekursiv aus selbstdefinierten Datentypen bestehen.

# Kapitel 3

# Automatisierung der Diagrammmethode

Das Ziel dieser Arbeit ist, die manuellen Beweise der Diagrammmethode, die in 2.4 vorgestellt wurde, zu automatisieren, um die Korrektheit der Programmtransformationen zu zeigen.
Um das zu realisieren, muss die Innermost Terminierung des Termersetzungssystems gezeigt werden, das durch die Forking Diagramme dargestellt wurde. Den tatsächlichen Terminierungsbeweis führt dann ein Terminierungsbeweiser, in unserem Fall AProVE (siehe 2.5).

Das Terminierungsproblem kann in ITRS kodiert werden, sogenannte „integer term rewrite systems“. Da der später verwendete Terminierungsbeweiser AProVE lediglich die Innermost-Terminierung von ITRS zeigt, werden die ITRS entsprechend so kodiert, dass sie lediglich eine Innermost-Terminierung benötigen. In Verlauf des Kapitels werde ich auf Innermost-Terminierungen eingehen, sie definieren und ihren Zweck in dieser Arbeit erläutern.

Wie in Kapitel 2.4 erläutert, erhalten wir unser ITRS aus einem ERSARS, das wiederum ein umgewandeltes NARS darstellt. Diese einzelnen Umformungen von ARS in SRSARS, bzw. von NARS in ERSARS, und von SRSARS in TRS, bzw. von ERSARS in die gewollten ITRS, werde ich im darauffolgenden Unterkapitel 3.2 beschreiben.
Diese TRS und ITRS sind der letzte Schritt der Umformungen. Die Eingaben werden dem Terminierungsbeweiser AProVE übergeben, um die Terminierung der Diagrammbeweise zu zeigen oder zu widerlegen.

## 3.1 Innermost-Terminierung

Die Innermost-Ersetzungsrelation ist eine eingeschränkte Ersetzungsrelation, die nicht alle Ersetzungen erlaubt. Bei einer Innermost-Ersetzungsrelation dürfen

Terme entsprechend der Termersetzungsregeln ersetzt werden - solange kein echter Unterterm weiter reduzierbar ist.
Formal heißt das, dass ein TRS beispielsweise aus den Regeln

$$f(a) \xrightarrow{R} f(a)$$
$$a \xrightarrow{R} b$$

besteht. Die Ersetzungsrelation ist definiert als

$$t \xrightarrow{R} t' \ .$$

Nun sei $C$ ein Kontext, $\sigma$ eine Substitution, $l$ eine linke und $r$ eine rechte Seite der Regel. Sei weiterhin $t = C[\sigma(l)]$ und $t' = C[\sigma(r)]$.
In dem Fall kann bezüglich der Ersetzungsrelation $R$ (ohne Strategie) keine Terminierung nachgewiesen werden, da der Term $f(a)$ durch $f(a)$ unendlich oft ersetzt werden darf.
Betrachten wir nun jedoch die Innermost Strategie $\xrightarrow{IR}$. Sie ist ähnlich definiert wie $\xrightarrow{R}$:

$$t \xrightarrow{IR} t' \ .$$

$C$ sei wieder ein Kontext, $\sigma$ eine Substitution, $l$ eine linke und $r$ eine rechte Seite der Regel. Sei $t = C[\sigma(l)]$ und $t' = C[\sigma(r)]$.
Als weitere Regel gilt hier, dass nur reduziert werden darf, wenn kein echter Unterterm von $\sigma(l)$ reduzierbar ist mit $\xrightarrow{R}$.

Diese Regel der Innermost-Ersetzungsrelation führt zu einer nachweisbaren Terminierung unseres beispielhaften Termersetzungssystems, da nun stets die Regel $a \xrightarrow{R} b$ angewendet werden muss. Nun wird $f(a) \xrightarrow{IR} f(b)$ reduziert und das TRS terminiert, da $f(b)$ nicht mehr reduzibel und somit in der Normalform ist.

## 3.2 Umformungen

Betrachten wir nun die einzelnen Umformungsschritte von den anfänglich gegebenen ARS, bzw. NARS bis hin zu den TRS und ITRS Eingaben für den Terminierungsbeweiser.

### 3.2.1 Von ARS hin zu ERSARS

In diesem Abschnitt betrachten wir ARS und ihre Umformung in SRSARS, die Notwendigkeit von NARS und deren Umformung in ERSARS.

ARS sind *abstract reduction sequences*, also abstrakte Reduktionsfolgen. Terme und Reduktionen wurden in Kapitel 2.1 definiert, Reduktionsfolgen sind Folgen dieser einzelnen Reduktionsschritte auf Termen.
Abstrakt sind ARS, da konkrete Ausdrücke zwischen den einzelnen Reduktionsschritten wegabstrahiert wurden. Es stehen also keine Terme mehr zwischen den Reduktionspfeilen.
Eine Konstante A repräsentiert das erfolgreiche Ende einer Reduktionsfolge, die Antwort der Reduktion. Diese Antwort ist abstrakt, sie steht als Konstante A für jede beliebige Antwort.
Bei den in den vorigen Kapiteln betrachteten Forking-Diagrammen haben wir das Regelsystem $L \rightsquigarrow R$ betrachtet, bei dem $L$ und $R$ ARS darstellen.
ARS können in ihren Standardreduktionen Variablen besitzen von der Form $\xrightarrow{sr,x}$ mit $x$ als Variable. Solche Variablen stehen für beliebige Labels, wie in Abschnitt 2.2 vorgestellt wurde.
Zur Erinnerung: Für eine Reduktion mit Label schreiben wir $\xrightarrow{sr,l}$.

**Definition 3.1** (Definition 3.3 in [RSSS12]). *Eine* abstrakte Reduktionsfolge *(ARS) ist eine endliche Folge $I_n \dots I_1$, und eine* konvergierende ARS *(cARS) ist eine endliche Folge $AI_n \dots I_1$ bei der A eine Konstante ist, die eine Antwort repräsentiert, $n \geq 0$. Das Symbol $I_j$ kann entweder das Symbol $\xleftarrow{sr,l}$ sein mit $l \in \mathcal{L}$ als (gelabelte) Standardreduktion, das Symbol $\xleftarrow{sr,x}$, wobei $x$ eine Variable darstellt, $\xleftarrow{sr,\tau}$ mit $\tau \notin \mathcal{L}$, wobei $\tau$ für die Vereinigung von Labels steht, oder das Symbol $\xrightarrow{T_i}$, das für eine Transformation $T_i$ steht. Jedes dieser Symbole kann auch um + erweitert werden, was für die transitive Hülle der Reduktion oder Transformation steht.*
*Ein ARS oder cARS ohne Variablen nennen wir* Grund-ARS *oder* Grund-cARS, *und schreiben auch* gARS, *bzw.* gcARS. *Und wenn in einem Grund-ARS (beziehungsweise Grund-cARS) keine transitiven Hüllen, also keine +, vorkommen, nennen wir es* einfach.

Unser Ziel ist, die möglichen Variablen und transitiven Hüllen der ARS zu eliminieren.
Dafür übersetzen wir diese ARS nun in SRSARS, die *simple rewrite system on abstract reduction sequences.*

**Definition 3.2** (Definition 3.7 aus [RSSS12]). *Sei $D$ ein Menge von Ersetzungsregeln der Form $L \rightsquigarrow R$, mit $L$ und $R$ als einfache ARS. Sei weiterhin cARS($D$)*

*eine Menge von einfachen cARS, die aus den in D vorkommenden Symbolen erstellt werden kann. Dann ist das string rewriting system* $cARS(D), \xrightarrow{D}$ *ein einfaches Ersetzungssystem auf einer abstrakten Reduktionsfolge (SRSARS).*

Die Übersetzung J(D) von ARS in SRSARS ist für einen vollständigen Diagrammsatz definiert und ersetzt zunächst alle Variablen durch alle Kombinationen der Labels $l$, die im Diagrammsatz als $\xrightarrow{sr,l}$ vorkommen und zusätzlich durch $\tau$. Das Label *tau* repräsentiert dabei all diejenigen Standardreduktionen, die nicht im Diagrammsatz vorkommen und stellt damit eine Vereinigung von gelabelten Standardreduktionen dar.
In einem zweiten Schritt werden die transitiven Hüllen eliminiert: Jeder Pfeil $\xrightarrow{+,O}$ wird dabei durch jede mögliche Folge von $\xrightarrow{O}$ Pfeile ersetzt, wobei $O$ für Standardreduktionen ($\xrightarrow{sr,l}$) oder Transformationen ($\xrightarrow{T_i}$) stehen kann. Dieser Prozess erzeugt unendlich viele Diagramme.

Mit den SRSARS haben wir nun also die + und die Variablen getilgt. Das reicht jedoch nicht, da wir unter Umständen unendlich viele Regeln in den SRSARS haben. Da wir keine Eingabe von unendlich vielen Regeln formulieren können, können wir auf diese Weise auch keine Eingabe für unseren Terminierungsbeweiser erstellen.

Ein naiver Ansatz wäre - analog zum nicht terminierenden Beispiel in Abschnitt 2.4 - alle Regeln mit transitiven Hüllen mit den Regeln $\xrightarrow{T,+} \rightsquigarrow \xrightarrow{T}\xrightarrow{T,+}$ und $\xrightarrow{T,+} \rightsquigarrow \xrightarrow{T}$ für jedes vorkommende Symbol $\xrightarrow{T,+}$ zu ersetzen.
Dieser Ansatz würde die transitive Hülle wie gewünscht entfalten, aber wäre hinfällig, da er nicht terminiert.

Aus dieser Problematik entsteht die Notwendigkeit der *ERSARS*. Diese ERSARS kann man als analoge Kodierung zu SRSARS sehen, aber mit NARS als Ausgangspunkt statt ARS.

**Definition 3.3** (Definition 3.1 aus [RSSS12]). *Eine (respektive konvergierende)* abstrakte Reduktionsfolge mit natürlichen Zahlen *(NARS) (respektive cNARS) sei eine Folge* $I_n \dots I_1$ *(respektive* $AI_n \dots I_1$*), bei der A eine Antwort repräsentiert und jedes* $I_j$ *ein Symbol der Form* $\xleftarrow{sr,l}, \xrightarrow{T_i}, \langle w \rangle, \langle w, k \rangle, \langle w, \overline{k}+1 \rangle$ *sei, wobei* $l \in \mathcal{L} \cup \{\tau\}$, $w \in W$ *für eine Menge von Namen und W mit* $W \cap (\mathcal{L} \cup \{\tau\}) = \emptyset$, *und k entweder eine natürliche Zahl* ($k \in \mathbb{N}$) *sei oder eine* Zahlenvariable, *also eine Variable, die nur durch eine natürliche Zahl ersetzt werden darf.* $\overline{k}$ *sei immer eine Zahlenvariable.*

*Wir nennen eine NARS (respektive cNARS) eine* Grund-NARS, *genau dann, wenn sie keine Zahlenvariablen enthält.*

Mit diesen NARS können wir, wie in Kapitel 2.4 beschrieben, unendliche Folgen von Regelanwendungen verhindern durch die Bestimmung einer konstanten Zahl, bezie-

hungsweise einer Zahlenvariablen, die die Anzahl der Regelanwedungen bestimmt. Definieren wir nun die Ersetzungsregeln auf einem Grund-cNARS und kommen damit zu den ERSARS.

**Definition 3.4** (Definition 3.21 aus [RSSS12])**.** *Sei $D$ eine Menge von Regeln der Form $L \rightsquigarrow R$, bei der $L, R$ NARS sind. Sei weiterhin $gcNARS(D)$ die Menge aller Grund-cNARS, die durch Ersetzen der vorkommenden Symbole in $D$ durch irgendeine Substitution $\sigma$ gebildet werden können. Das Ersetzungssystem $(gcNARS(D), \xrightarrow{D})$ heißt dann* encoded rewriting system on abstract reduction systems *(ERSARS), wobei $\xrightarrow{D}$ definiert sei durch: Wenn $S = S'L'S''$, $L \rightsquigarrow R \in D$, und $\sigma$ eine Zahlensubstitution ist mit $\sigma(L) = L'$, dann gilt $S \xrightarrow{D} S'\sigma(R)S''$.*

Durch die Umformung von ARS in SRSARS haben wir also die transitiven Hüllen und die Variablen entfernt, und durch Umformen der ARS in NARS, also Bestimmung einer konstanten Zahl, die unendliche Folgen einer Regelanwendung umgeht, haben wir ein weiteres Hindernis erfolgreich überwunden.
Die Übersetzung von ARS in NARS wird dabei erneut für einen vollständigen Satz von Diagrammen durchgeführt:
Zunächst werden alle Variablen entfernt (analog zur Übersetzung in SRSARS), die Plusse auf den linken Seiten werden als Symbol nicht gänzlich getilgt, sondern mehr oder weniger beibehalten. Es werden Kontraktionsregeln hinzugefügt, die aus Folgen von gleichen Pfeilen einen „Pfeil mit Plus " erzeugen.
Plusse auf den rechten Seiten der Regeln werden mit der vorher besprochenen Methode behandelt: Sie werden durch neue Symbole ersetzt, die Zahlvariablen enthalten, und es werden die sogenannten Expansionsregeln hinzugefügt, die die (endliche) Entfaltung der transitiven Hüllen auf der Ebene des Ersetzungssystems durchführen.
Für nähere Informationen zu diesem Prozess siehe [RSSS12], Kapitel 3.

Diese NARS haben wir wiederum in ERSARS umgeformt, wofür wir lediglich ein Regelsystem zu den NARS hinzugefügt haben.

Kommen wir nun zu der Übersetzung K(D) für (Forking-)Diagramme D, die Diagramme auf ARS in Diagramme auf NARS übersetzen - womit wir bei den ERSARS angelangt sind.

**Definition 3.5** (Definition 3.22 aus [RSSS12])**.** *Für ein generelles Forking-Diagramm oder Antwort-Diagramm $L \rightsquigarrow R$ und $M \subseteq \mathcal{L}$ ist die Übersetzung $\mathcal{V}_M$ eine endliche Menge von Ersetzungsregeln auf Grund-ARS:*

$\mathcal{V}_M(S_L \rightsquigarrow S_R) := \bigcup \{V_{\neg M}(S_L) \rightsquigarrow V_{\neg M}(S_R) \mid V_{\neg M}$ *ist eine Variableninterpretation*$\}$

*Sei eine Menge $D = \bigcup_i \{L_i \rightsquigarrow R_i\}$ von generellen Forking-Diagrammen und Antwortdiagrammen gegeben und sei $M \subseteq \mathcal{L}$, mit $\mathcal{L}$ als Menge der Labels, dann sei die Übersetzung $\mathcal{K}_M(D)$ wie folgt definiert:*

*Zunächst werden alle Variablen interpretiert, daraus resultiert die Menge* $D' := \bigcup_i \{\mathcal{V}_M(L_i \rightsquigarrow R_i)\}$. *Für jede Regel* $L \rightsquigarrow R \in D'$ *enthält die Menge* $\mathcal{K}_M(D)$ *eine Regel* $\mathcal{K}_L(L) \to \mathcal{K}_R(R)$.

**Konstruktion von** $\mathcal{K}_L(L)$: *Sei* $L = I_n \dots I_1 \xrightarrow{T_i}$, *wobei* $I_j$ *entweder* $\xleftarrow{sr,l_j}$, *oder* $\xleftarrow{sr,l_j,+}$, *oder (für* $j = 1$*)* $I_j = A$ *ist. Sei* $K_j := \langle w_j \rangle$, *wenn* $I_j = \xleftarrow{sr,l_j,+}$, *sonst sei* $K_j := I_j$, *wobei* $w_j \in W$ *neue Namen darstellen, für jede neue Regel neu instanziiert. Dann setzen wir* $\mathcal{K}_L(L) := K_n \dots K_1 \xrightarrow{T_i}$. *Für jedes* $I_j$ *der Form* $\xleftarrow{sr,l_j,+}$ *fügen wir zwei sogenannte* Kontraktionsregeln *hinzu:* $\xleftarrow{sr,l_j} K_{j-1} \dots K_1 \xrightarrow{T_i} \rightsquigarrow K_j K_{j-1} \dots K_1 \xrightarrow{T_i}$ *und* $\xleftarrow{sr,l_j} K_j K_{j-1} \dots K_1 \xrightarrow{T_i} \rightsquigarrow K_j K_{j-1} \dots K_1 \xrightarrow{T_i}$.

**Konstruktion von** $\mathcal{K}_R(R)$: *Sei* $R = I_n \dots I_1$. *Wenn keines der* $I_j$ *ein* $+$ *enthält, dann ist die Übersetzung* $\mathcal{K}_R(R) := I_n \dots I_1$. *Ansonsten gibt es mindestens ein* $+$: *Sei* $L_j := \xrightarrow{T_i}$, *wenn* $I_j = \xrightarrow{T_i,+}$, $L_j := \xleftarrow{sr,l_j}$, *wenn* $I_j = \xleftarrow{sr,l_j,+}$ *und in allen anderen Fällen sei* $L_j := I_j$. *Sei weiter* $w'_j \in W$ *(für* $j \in \{1, \dots, n\}$*) neue Namen. Sei* $I_a$ *das rechteste* $I_j$, *das ein* $+$ *enthält, dann setzen wir* $\mathcal{K}_R(R) := \langle w'_a, k \rangle L_{a-1} \dots L_1$, *wobei* $k$ *eine Zahlenvariable sei. Für alle* $I_j$ *mit* $I_j = \xrightarrow{T_i,+}$ *oder* $I_j = \xleftarrow{sr,l_j,+}$ *fügen wir sogenannte* Expansionsregeln *hinzu:* $\langle w'_j, k+1 \rangle \rightsquigarrow \langle w'_j, k \rangle L_j$ *und* $\langle w'_j, 1 \rangle \rightsquigarrow L_n \dots L_j$. *Wenn ein* $m > j$ *existiert, wobei* $I_m$ *ein* $+$ *enthält, dann fügen wir für das kleinste* $m$ *dieser Art eine zusätzliche Expansionsregel hinzu:* $\langle w'_j, k+1 \rangle \rightsquigarrow \langle w'_m, k \rangle L_{m-1} \dots L_j$.

Und damit ist die Übersetzung in ERSARS definiert.

### 3.2.2 Von ERSARS zu ITRS

Betrachten wir nun die aus dem Abschnitt 3.2.1 erstellten ERSARS. Dort haben wir ERSARS bereits definiert.

Für einen vollständigen Satz an Diagrammen ist $M$ die Menge der Labels, die nicht in dem entsprechenden Diagramm vorkommen.

Die Symbole $\langle w_i \rangle$ und $\langle w'_j, k \rangle$ werden zusammen mit den zusätzlichen Regeln dafür verwendet, die $+$ Symbole auf den linken und rechten Seiten der Forking- und Antwort-Diagramme zu interpretieren. Es lässt sich leicht zeigen, dass jede Folge von Ersetzungen, die lediglich Kontraktionsregeln verwendet, endlich sein muss, und ebenso, dass jede Folge von Ersetzungen, die lediglich Expansionsregeln verwendet, ebenfalls endlich sein muss.

Diese ERSARS müssen nun lediglich noch umgeformt werden in ITRS, um sie AProVE als Eingabe übergeben zu können.

Ein Transformationsschritt $\xrightarrow{T_i}$ sei kodiert als einstelliges Funktionssymbol *ti*, ein Reduktionsschritt $\xleftarrow{sr,l_i}$ sei ebenfalls als einstelliges Funktionssymbol kodiert, nämlich *srli*, und das Antwortsymbol $A$ sei kodiert als Konstante $A$.

Ein Forking-Diagramm der Form $\xleftarrow{sr,l_1}\xrightarrow{T_1} \rightsquigarrow \xrightarrow{T_2}\xleftarrow{sr,l_2}$ wird kodiert als Termersetzungsregel $t1(srl1(X)) \rightarrow srl2(t1(X))$, wobei $X$ eine Variable sei.
Somit lässt sich ein vollständiger Satz von Diagrammen als eine Menge von Termersetzungsregeln beschreiben.
An der Implementierung eines Programms, das die vollständige Menge von Diagrammen berechnet, wird momentan gearbeitet. Mehr Informationen dazu unter [RSS11].

Betrachten wir ein Beispiel für ein Forking-Diagramm, das in ein ERSARS umgeformt, und dann von einem ERSARS in ein ITRS kodiert wird.
Sei folgendes Forking-Diagramm $D$ gegeben:

$$\begin{array}{ccc} \cdot & \xrightarrow{i\mathcal{S},llet} & \cdot \\ {\scriptstyle n,lll,+}\big\downarrow & \swarrow {\scriptstyle n,lll,+} & \\ \cdot & & \end{array}$$

Das Diagramm wird nun ersetzt durch die Regel $\langle w\rangle\xrightarrow{i\mathcal{S},llet} \rightsquigarrow \langle v,k\rangle$.
Da auf der linken Seite des Diagramms eine transitive Hülle ist, werden zwei Kontraktionsregeln hinzugefügt, nämlich:

$$\xleftarrow{n,lll}\xrightarrow{i\mathcal{S},llet} \rightsquigarrow \langle w\rangle\xrightarrow{i\mathcal{S},llet} \qquad \text{und} \qquad \xleftarrow{n,lll}\langle w\rangle\xrightarrow{i\mathcal{S},llet}\rightsquigarrow \langle w\rangle\xrightarrow{i\mathcal{S},llet}.$$

Da auch auf der rechten Seite eine transitive Hülle ist, werden auch Expansionsregeln hinzugefügt, nämlich:

$$\langle v,k+1\rangle \rightsquigarrow \langle v,k\rangle\xleftarrow{n,lll} \qquad \text{und} \qquad \langle v,1\rangle \rightsquigarrow \xleftarrow{n,lll}.$$

Dann ist für $\mathcal{L} = \{lll, llet, seq, \ldots\}$ und $M = \mathcal{L} \setminus \{lll, llet\}$ die Übersetzung $\mathcal{K}_M(D)$:

$$\begin{aligned} &\{\langle w\rangle\xrightarrow{i\mathcal{S},llet} \rightsquigarrow \langle v,k\rangle, \\ &\xleftarrow{n,lll}\xrightarrow{i\mathcal{S},llet} \rightsquigarrow \langle w\rangle\xrightarrow{i\mathcal{S},llet}, \\ &\xleftarrow{n,lll}\langle w\rangle\xrightarrow{i\mathcal{S},llet} \rightsquigarrow \langle w\rangle\xrightarrow{i\mathcal{S},llet}, \\ &\langle v,k+1\rangle \rightsquigarrow \langle v,k\rangle\xleftarrow{n,lll}, \\ &\langle v,1\rangle \rightsquigarrow \xleftarrow{n,lll}\}. \end{aligned}$$

Umgewandelt in ein ITRS als AProVE Eingabe wird für die Regel
$\langle w\rangle\xrightarrow{i\mathcal{S},llet} \rightsquigarrow \langle v,k\rangle$ die ITRS-Regel $iSllet(w(X)) \rightarrow v(K,X)$ mit $K > 0$

hinzugefügt.

Analog werden die anderen ERSARS-Regeln in Regeln des ITRS umgeformt.

Somit sieht das ITRS dann folgendermaßen aus:

$$iSllet(w(X)) \to v(K,X) \ \text{ mit } K > 0$$
$$iSllet(w(nlll(X))) \to iSllet(w(X))$$
$$iSllet(nlll(X)) \to iSllet(w(X))$$
$$v(K,X) \to nlll(v(K-1,X)), \ \text{ falls } K > 1$$
$$v(1,X) \to nlll(X)$$

Bei dieser Umformung in ein ITRS ist die erste Regel also die Kodierung des Diagramms, und die folgenden Regeln sind wie beschrieben Kontraktionsregeln und Expansionsregeln, die aufgrund der transitiven Hüllen im Forking-Diagramm hinzugefügt werden müssen.
Die erste Bedingung $K > 0$ stellt sicher, dass eine positive Zahl als Zahlenvariable gewählt wird, und die Bedingung $K > 1$ stellt sicher, dass K nach der Regelanwedung noch eine positive Zahl ist. Wenn $K = 1$, greift die letzte ITRS-Regel

$$v(1,X) \to nlll(X),$$

und damit ist der letzte Schritt der Entfaltung der transitiven Hüllen berechnet.

Diese Umformungen wurden in meiner Arbeit mithilfe von Haskell implementiert, was uns zum nächsten Schritt führt; der Implementierung in Kapitel 4.

# Kapitel 4

# Implementierung

Nachdem der genaue Verlauf der Umwandlungen beschrieben wurde, kommen wir nun zur Implementierung jener Umformungen.

Der Kern meiner Arbeit befasst sich mit der Implementierung der Umformungen aus Kapitel 3, also mit der schrittweisen Kodierung von ARS hin zu ITRS. Diese Umformungen wurden mit Haskell implementiert.

Ziel und Zweck meines Programms ist, eine Eingabe von Diagrammen nach den beschriebenen Methoden umzuformen, um als Ausgabe das ITRS für AProVE zu erhalten, dessen Terminierung mit AProVE getestet werden soll.
Betrachten wir zunächst die wichtigsten Datentypen:

```
data Label = Rule String | Tau | Var String |  Plus Label | PlusL Label
| PlusR Label
 deriving(Show,Eq)

data Item     = SR Label | Trans Label | Answer

data Diagram  =  [Item] :~~> [Item]
 deriving(Show,Eq)

data Term = TermVariable String | IntegerVariable String
| Fn String [Term]
 deriving(Eq,Show)

data Termersetzungsregel = Term :--> Term
 deriving(Eq,Show)
```

Damit sind die möglichen Label festgelegt: eine beliebige Reduktionsregel, Tau, eine Variable oder transitive Hüllen in Form von Plussen.

Der Datentyp Diagramm ist wie in unseren Forking-Diagrammen eine Regel auf einem ARS definiert, wobei das ARS durch den Datentyp Item festgelegt wird. Das ARS kann dabei aus einer Antwort und beliebig vielen SR Labels und Trans Labels bestehen.

*Wir betrachten die Implementierung in den folgenden Schritten:*

**Diagramme modifizieren** Erst betrachten wir die Modifizierung der Diagramme durch die Übersetzung $K_M$ aus dem Paper [RSSS12]. Dabei werden aus den gegebenen Diagrammen TRS-Regeln gebaut - noch ohne Kontraktions- und Expansionsregeln.

**Plusse entfernen** Hier werden die Plusse auf den linken und rechten Seiten der Diagramme entfernt.

**Hauptaufrufe** In diesem Abschnitt stelle ich die Hauptaufrufe mit ihren wichtigsten Komponenten vor.

Für den gesamten Code des Programms siehe [Dem12].

## 4.1 Übersetzung $K_M$

Für die Übersetzung der Diagramme in TRS werden zunächst alle Variablen (also variablen Labels) berechnet. Man möchte schließlich für die Variablen alle Labels aus $Q$ und zusätzlich das Label Tau einmal einsetzen, wobei $Q$ die Menge alles Labels ist, die in den Standardreduktionen vorkommen, und für gleiche Variablen innerhalb eines Diagramms immer das gleiche Label eingesetzt wird.
Diese Funktion (des Variablenberechnens) übernimmt im Programmcode die Funktion `computeVariables`, die als Eingabe den Datentyp `diagram` erwartet.

Im nächsten Schritt werde alle Labels für den gesamten Diagrammsatz berechnet - also eben die Menge Q. Sie fügt Tau automatisch an den Anfang der Liste, und führt dann die Hilfsfunktion `getLabels` , die die Labels für ein Diagramm berechnet, aus. Außerdem werden doppelte Einträge entfernt (durch `nub`), da wir keine doppelten Labels in unserer Menge Q erhalten wollen:

```
computeLabels diagrams = Tau:(nub $ concatMap getLabels diagrams)
```

Die Hilfsfunktion `getLabels` geht dabei sukzessive alle Fälle durch und konkateniert die Ergebnisse seiner Berechnungen der linken Regelseite des Diagramms mit denen der rechten Seite.

```
getLabels (links :~~> rechts) = getLabels' links ++ getLabels' rechts
getLabels' [] = []
getLabels' (x:xs) = (getLabels'' x) ++ (getLabels' xs)
getLabels'' (SR lab)   = getLabels''' lab
getLabels'' (Trans lab) = [] -- getLabels''' lab
getLabels'' Answer = []
getLabels''' (Rule x) = [Rule x]
getLabels''' (Plus lab) = getLabels''' lab
getLabels''' _       = [] --
```

Als nächster und letzter Schritt werden alle Kombinationen verwendet, da jede Kombination von Einsetzungen der Variablen abgefangen werden muss. Dies geschieht mit der Funktion `all_combinations`.
Fehlermeldungen fangen Fehleingaben wie Tau in der Eingabe ab.

Als wohl wichtigster Schritt dieses Abschnitts gilt die Funktion `convert`:

```
convert :: Diagram -> Termersetzungsregel
convert (l :~~> r) = (conv (reverse l)) :--> (conv (reverse r))
conv :: [Item] -> Term
```

Sie wandelt die Diagramme in Termersetzungsregeln um. Termersetzungsregeln bestehen aus einer Regel (im entsprechenden Datentyp als `:-->` dargestellt) auf zwei Termen.
Wieder werden linke und rechte Regelseite aufgeteilt und separat behandelt; die Hilfsfunktion `conv` wandelt dabei die Standardreduktionen und Transformationen und anderen Elemente der Liste `Item` in Terme um und fängt die einzelnen Fälle ab, wie man im Gesamtcode nachschlagen kann (siehe [Dem12]).

## 4.2 Plusse entfernen

Kommen wir nun zu dem wichtigen Schritt der Entfernung der Plusse: Hierin liegt eine der Besonderheiten des Programmcodes.
Um die Plusse zu entfernen, wird der dafür zuständigen Hauptfunktion `entfernePlusse` als Eingabe ein Diagrammsatz und eine Liste von Strings mit neuen Namen für die w-Symbole übergeben, und der Diagrammsatz ohne Plus und mit den entsprechenden zusätzlichen Regeln ausgegeben, also mit den entsprechenden Kontraktions- für die linken und den Expansionsregeln für die rechten Seiten.

Die Hauptfunktion sieht folgendermaßen aus:

```
entfernePlusse [] neueNamen = []
entfernePlusse (d:diagrams) neueNamen =
   let (neueDiagramme, restlicheNamen) =
   entfernePlusseEinDiagramm d neueNamen
   in neueDiagramme ++ (entfernePlusse diagrams restlicheNamen)

entfernePlusseEinDiagramm (links :~~> rechts) neueNamen =
  let (neueLinkeSeite,contraction_rules, restlicheNamen) =
   (entfernePlusLinks links neueNamen)
      (neueRechteSeite,expansion_rules, restlicheNamen') =
      (entfernePlusRechts rechts restlicheNamen)
   in ([neueLinkeSeite :~~> neueRechteSeite] ++
    contraction_rules ++ expansion_rules, restlicheNamen')
```

Dabei sieht man, dass `entfernePlusseEinDiagramm` auf beide Seiten der Regel angewandt wird; was notwendig ist, da für rechte und linke Seiten unterschiedliche Regeln hinzugefügt werden für transitive Hüllen. Für linke Seiten werden Kontraktionsregeln und für die rechten Seiten Expansionsregeln hinzugefügt.

Die genaue und aufwendige Implementierung der Kontraktions- und Expansionsregeln lassen sich im Gesamtcode nachsehen und entsprechen den in Kapitel 3 besprochenen Methoden.

## 4.3 Hauptaufrufe

In diesem Abschnitt möchte ich auf die Hauptaufrufe eingehen, die auch in Kapitel 5 bei den Tests zu sehen sein werden.

Betrachten wir zunächst den Hauptaufruf für Erstellung der NARS-Ausgabe: Hauptaufruf fuer NARS-Ausgabe Eingabe: Diagramme mit Variablen und Plussen Ausgabe: Diagramme in NARS-Form

```
createNARS :: Diagrams -> Diagrams
createNARS diagrams =
   let interpr_diagrams = variablen_interpretation diagrams
       neueNamen = ["w" ++ show i | i <- [1..]]
   in entfernePlusse interpr_diagrams neueNamen
```

Der Datentyp ist für `createNARS` für Eingabe und Ausgabe eine Liste von Diagrammen, also ein Diagrammsatz; die Eingabe besteht dabei wieder aus einer Liste von

Forking-Diagrammen, also mit möglichen Variablen und Plussen; und als Ausgabe werden Diagramme in NARS-Form erstellt.
Diese Funktion wird verwendet für den eigentlichen Hauptaufruf, nämlich die Funktion für die Ausgabe als ITRS:

```
printTRS diagrams = putStrLn (inputToTRS diagrams)
inputToTRS :: Diagrams -> String
inputToTRS diagrams =
let interpr_diagrams = createNARS diagrams
   ters = map convert interpr_diagrams
   sters = unlines $ map showTERS ters
 in header sters
```

Auch hier wird die im vorigen Abschnitt beschriebene Funktion `convert` verwendet, um Diagramme in Termersetzungsregeln umzuformen.

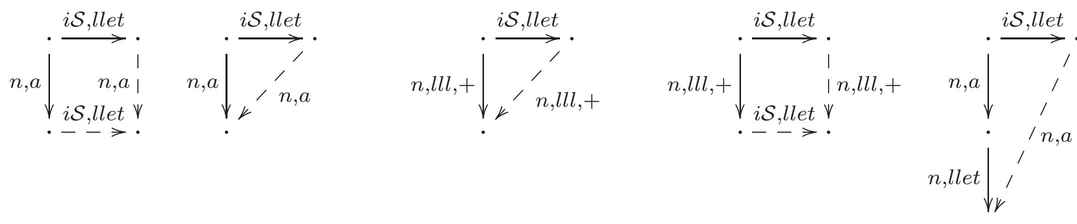
# Kapitel 5

# Tests und Ergebnisse

Die in Kapitel 4 beschriebene Implementierung habe ich getestet. Mit der WinGHCi von Haskell ([has12]) habe ich den Haskellcode getestet und anschließend überprüft, ob die aus der WinGHCi entstandene Ausgabe eine korrekte ITRS-Eingabe für den Terminierungsbeweiser AProVE ist und schließlich mit AProVE die Terminierung verifiziert.
Die Tests verliefen erfolgreich.

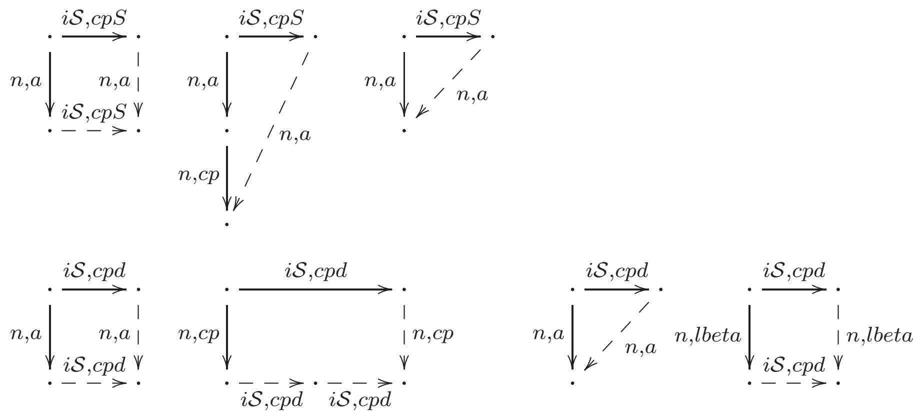
Betrachten wir nun Beispiele. In der Implementierung waren Beispieldiagramme, bzw. Beispieldiagrammsätze kodiert. Folgende Beispiele stammen aus dem Paper [SSSS08]. Das Verfahren der Diagramme ist dabei aus Symmetriegründen analog für die Umkehrung der Forking-Diagramme. Dafür werden lediglich die Transformationen umgedreht.

## 5.1 Vollständiger Satz von Forking-Diagrammen für $i\mathcal{S}llet$




## 5.2 Vollständiger Satz von Forking-Diagrammen für $i\mathcal{S}, cp$

Die Transformation ist aufgeteilt in $i\mathcal{S}, cpS$ und $i\mathcal{S}, cpd$.




## 5.3 Tests eines Diagrammsatzes

Nehmen wir nun die beiden in 5.1 und 5.2 genannten Diagrammsätze, um den gesamten Verlauf der Umwandlungen Schritt für Schritt zu zeigen, und die implementierten Funktionen des Programms zu testen.

Das Vorgehen ist folgendes:

1. Wir stellen die uns gegebenen Forking-Diagramme in Haskell dar.

2. Das von mir erstellte Programm wird genutzt, um die Ausgabe als NARS darzustellen, dies geschieht mithilfe des Befehls

   ```
   > createNARS diagrams
   ```

   `diagrams` steht dabei für eine Liste von Diagrammen, also einen Diagrammsatz.

3. Wir erzeugen mit dem Programm eine Ausgabe als TRS für AProVE. Dies geschieht mit dem Befehl

   ```
   > printTRS diagrams
   ```

   wobei `diagrams` wieder für eine Liste von Diagrammen steht.

4. Als letzten Schritt benutzen wir das TRS als Eingabe für AProVE und testen es auf Terminierung.

### 5.3.1 Transformation CP

Betrachten wir das Beispiel des Forking-Diagramms mit der Transformation *cp*. Diese ist aufgetrennt in zwei Transformationen, nämlich *cpS* und *cpd*.

Für die cpS Transformation haben wir folgende ARS:

$\xleftarrow{sr,a}\xrightarrow{CPS}\rightsquigarrow\xrightarrow{CPS}\xleftarrow{sr,a}$
$\xleftarrow{sr,CP}\xleftarrow{sr,a}\xrightarrow{CPS}\rightsquigarrow\xleftarrow{sr,a}$
$\xleftarrow{sr,a}\xrightarrow{CPS}\rightsquigarrow\xleftarrow{sr,a}$
$A\xrightarrow{CPS}\rightsquigarrow An$

Und für die cpd Transformation entsteht folgende ARS:

$\xleftarrow{sr,a}\xrightarrow{CPD}\rightsquigarrow\xrightarrow{CPD}\xleftarrow{sr,a}$
$\xleftarrow{sr,CP}\xrightarrow{CPD}\rightsquigarrow\xrightarrow{CPD}\xrightarrow{CPD}\xleftarrow{sr,CP}$
$\xleftarrow{sr,a}\xrightarrow{CPD}\rightsquigarrow\xleftarrow{sr,a}$
$\xleftarrow{sr,LBETA}\xrightarrow{CPD}\rightsquigarrow\xrightarrow{CPD}\xleftarrow{sr,LBETA}$
$A\xrightarrow{CPD}\rightsquigarrow A$

Für dieses ARS wenden wir nun die genannten Schritte zum Testen an.

1. Wir stellen das gegebene Forking-Diagramm in Haskell-Code dar und nennen es `cp_diagramme_forking`.

```
[[SR (Var "a"),Trans (Rule "CPS")] :~~>
[Trans (Rule "CPS"),SR (Var "a")],

[SR (Rule "CP"),SR (Var "a"),Trans (Rule "CPS")] :~~> [SR (Var "a")],

[SR (Var "a"),Trans (Rule "CPS")] :~~> [SR (Var "a")],

[Answer,Trans (Rule "CPS")] :~~> [Answer],

[SR (Var "a"),Trans (Rule "CPD")] :~~>
[Trans (Rule "CPD"),SR (Var "a")],

[SR (Rule "CP"),Trans (Rule "CPD")] :~~>
[Trans (Rule "CPD"),Trans (Rule "CPD"),SR (Rule "CP")],

[SR (Var "a"),Trans (Rule "CPD")] :~~> [SR (Var "a")],

[SR (Rule "LBETA"),Trans (Rule "CPD")] :~~>
[Trans (Rule "CPS"),SR (Rule "LBETA")],

[Answer,Trans (Rule "CPD")] :~~> [Answer]]
```

2. Wir erstellen mithilfe des Programms ein NARS aus dem gegebenen ARS,

und wenden den entsprechenden Befehl auf unserem gespeicherten Forking-Diagramm an.

```
> createNARS cp_diagramme_forking
```

Damit erhalten wir die NARS-Ausgabe, die in Abbildung 5.1 zu sehen ist.

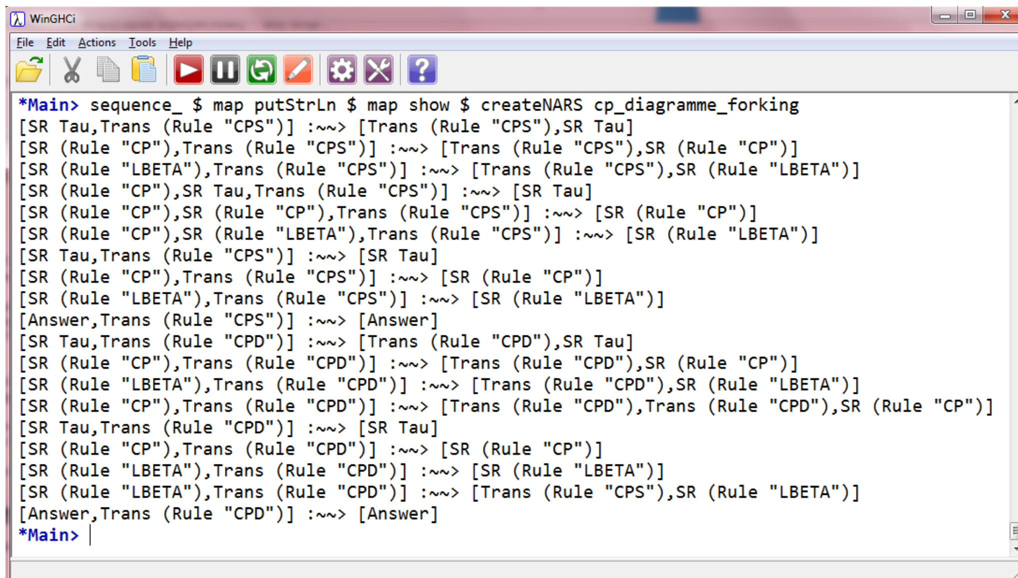



Abbildung 5.1: Haskell-Eingabe: Forking-Diagramme, Ausgabe: NARS

3. Nun erzeugen wir ein TRS für AProVE, indem wir den Befehl

   ```
   > printTRS cp_diagramme_forking
   ```

   ausführen und erzeugen die TRS-Ausgabe, die in Abbildung 5.2 zu sehen ist.

4. Als letzten Schritt geben wir nun das TRS in AProVE ein, um es auf Terminierung zu prüfen. Dies ist in Abbildung 5.3 zu sehen.

   Nach Bestätigen der Eingabe zeigt AProve die Terminierung, wie in Abbildung 5.4 zu sehen ist.

### 5.3.2 Transformation LLET

Wir wenden analog zu 5.3.1 das gleiche Verfahren an.

Die Forkingdiagramme und das Antwortdiagramm aus dem zweiten Diagrammsatz können in Haskellcode wie folgt repräsentiert werden:

```
[[SR (Var "a"),Trans (Rule "LLET")] :~~>
[Trans (Rule "LLET"),SR (Var "a")],
```

```
[SR (Var "a"),Trans (Rule "LLET")] :~~> [SR (Var "a")],

[SR (Plus (Rule "LLL")),Trans (Rule "LLET")] :~~>
[SR (Plus (Rule "LLL"))],

[SR (Plus (Rule "LLL")),Trans (Rule "LLET")] :~~>
[Trans (Rule "LLET"),SR (Plus (Rule "LLL"))],

[SR (Rule "LLET"),SR (Var "a"),Trans (Rule "LLET")] :~~>
[SR (Var "a")],

[Answer,Trans (Rule "LLET")] :~~> [Answer]]
```

Durch das von mir implementierte Programm lautet das daraus entstehende ER-SARS:

```
[[SR Tau,Trans (Rule "LLET")] :~~> [Trans (Rule "LLET"),SR Tau],

[SR (Rule "LLL"),Trans (Rule "LLET")] :~~>
[Trans (Rule "LLET"),SR (Rule "LLL")],

[SR (Rule "LLET"),Trans (Rule "LLET")] :~~>
[Trans (Rule "LLET"),SR (Rule "LLET")],

[SR Tau,Trans (Rule "LLET")] :~~> [SR Tau],

[SR (Rule "LLL"),Trans (Rule "LLET")] :~~> [SR (Rule "LLL")],

[SR (Rule "LLET"),Trans (Rule "LLET")] :~~> [SR (Rule "LLET")],

[WSymbol "w22",Trans (Rule "LLET")] :~~> [WPrimeSymbol "w24" VarK],

[SR (Rule "LLL"),Trans (Rule "LLET")] :~~>
[WSymbol "w22",Trans (Rule "LLET")],

[SR (Rule "LLL"),WSymbol "w22",Trans (Rule "LLET")] :~~>
[WSymbol "w22",Trans (Rule "LLET")],

[WPrimeSymbol "w24" VarKPlusOne] :~~>
[WPrimeSymbol "w24" VarK,SR (Rule "LLL")],

[WPrimeSymbol "w24" One] :~~> [SR (Rule "LLL")],

[WSymbol "w25",Trans (Rule "LLET")] :~~> [WPrimeSymbol "w28" VarK],

[SR (Rule "LLL"),Trans (Rule "LLET")] :~~>
[WSymbol "w25",Trans (Rule "LLET")],

[SR (Rule "LLL"),WSymbol "w25",Trans (Rule "LLET")] :~~>
[WSymbol "w25",Trans (Rule "LLET")],
```

```
[WPrimeSymbol "w28" VarKPlusOne] :~~>
[WPrimeSymbol "w28" VarK,SR (Rule "LLL")],

[WPrimeSymbol "w28" One] :~~> [Trans (Rule "LLET"),SR (Rule "LLL")],

[SR (Rule "LLET"),SR Tau,Trans (Rule "LLET")] :~~> [SR Tau],

[SR (Rule "LLET"),SR (Rule "LLL"),Trans (Rule "LLET")] :~~>
[SR (Rule "LLL")],

[SR (Rule "LLET"),SR (Rule "LLET"),Trans (Rule "LLET")] :~~>
[SR (Rule "LLET")],

[Answer,Trans (Rule "LLET")] :~~> [Answer]]
```

Dieses ERSARS können wir nun in ein ITRS als Eingabe für AProVE umwandeln. Somit entsteht folgende Eingabe für AProVE:

```
(VAR x k)
(RULES
TransLLET(SRTau(x)) -> SRTau(TransLLET(x))
TransLLET(SRLLL(x)) -> SRLLL(TransLLET(x))
TransLLET(SRLLET(x)) -> SRLLET(TransLLET(x))
TransLLET(SRTau(x)) -> SRTau(x)
TransLLET(SRLLL(x)) -> SRLLL(x)
TransLLET(SRLLET(x)) -> SRLLET(x)
TransLLET(w22(x)) -> w24(k-1,x) :|: k > 0
TransLLET(SRLLL(x)) -> TransLLET(w22(x))
TransLLET(w22(SRLLL(x))) -> TransLLET(w22(x))
w24(k,x) -> SRLLL(w24(k-1,x)) :|: k > 0
w24(1,x) -> SRLLL(x)
TransLLET(w25(x)) -> w28(k-1,x) :|: k > 0
TransLLET(SRLLL(x)) -> TransLLET(w25(x))
TransLLET(w25(SRLLL(x))) -> TransLLET(w25(x))
w28(k,x) -> SRLLL(w28(k-1,x)) :|: k > 0
w28(1,x) -> SRLLL(TransLLET(x))
TransLLET(SRTau(SRLLET(x))) -> SRTau(x)
TransLLET(SRLLL(SRLLET(x))) -> SRLLL(x)
TransLLET(SRLLET(SRLLET(x))) -> SRLLET(x)
TransLLET(Answer) -> Answer
)
```

Nach Eingabe in AProVE kann auch dieses die Innermost-Terminierung für das ITRS nachweisen.

Das Ergebnis meiner Tests und der vorangehenden Untersuchungen ist, dass AProVE genutzt werden kann, um die Korrektheit der Programmtransformation zu zeigen, da die Innermost-Terminierung das kodierten ITRS die Leftmost-Terminierung des ERSARS impliziert.

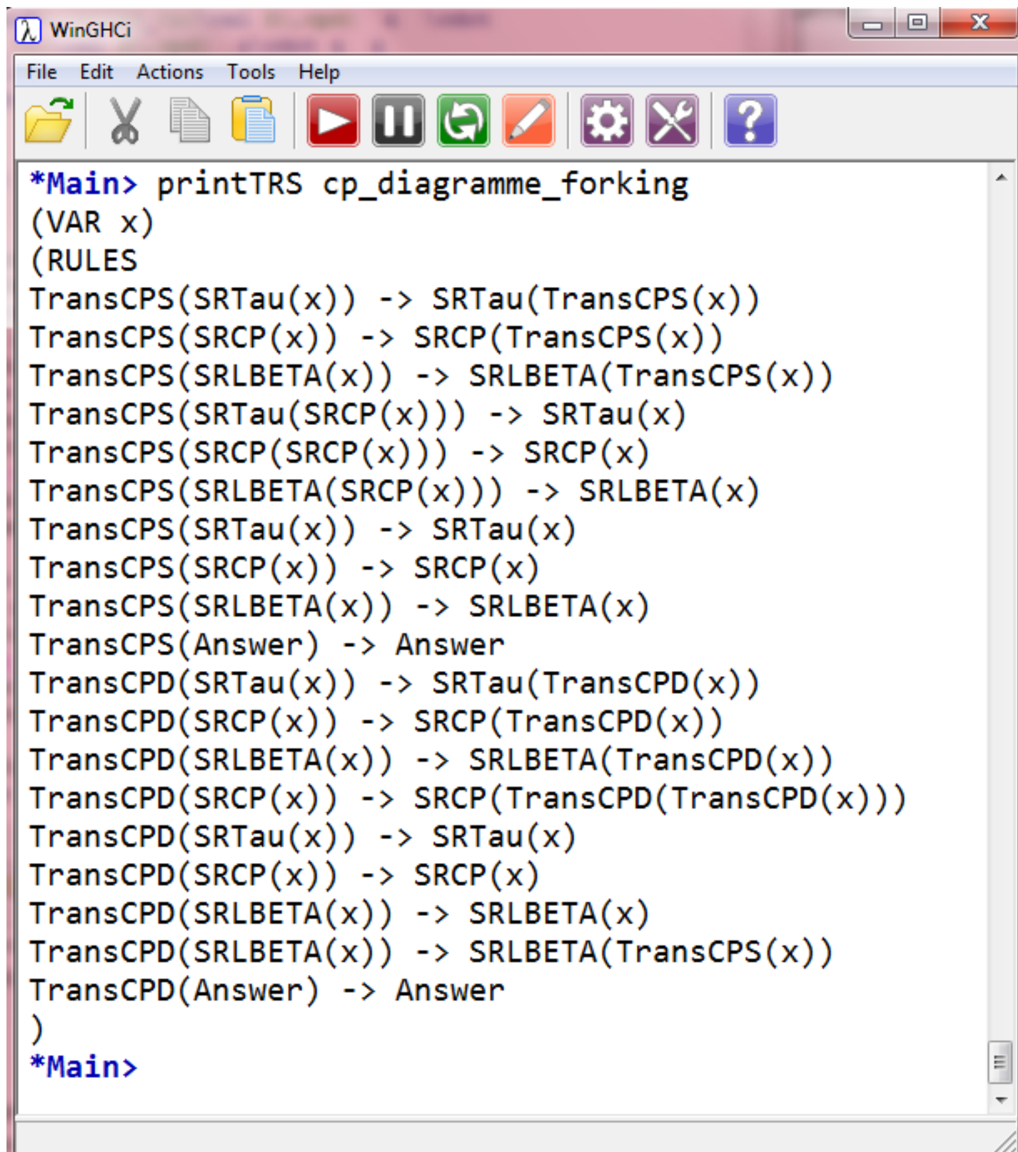



Abbildung 5.2: Haskell-Eingabe: Forking-Diagramme, Ausgabe: TRS

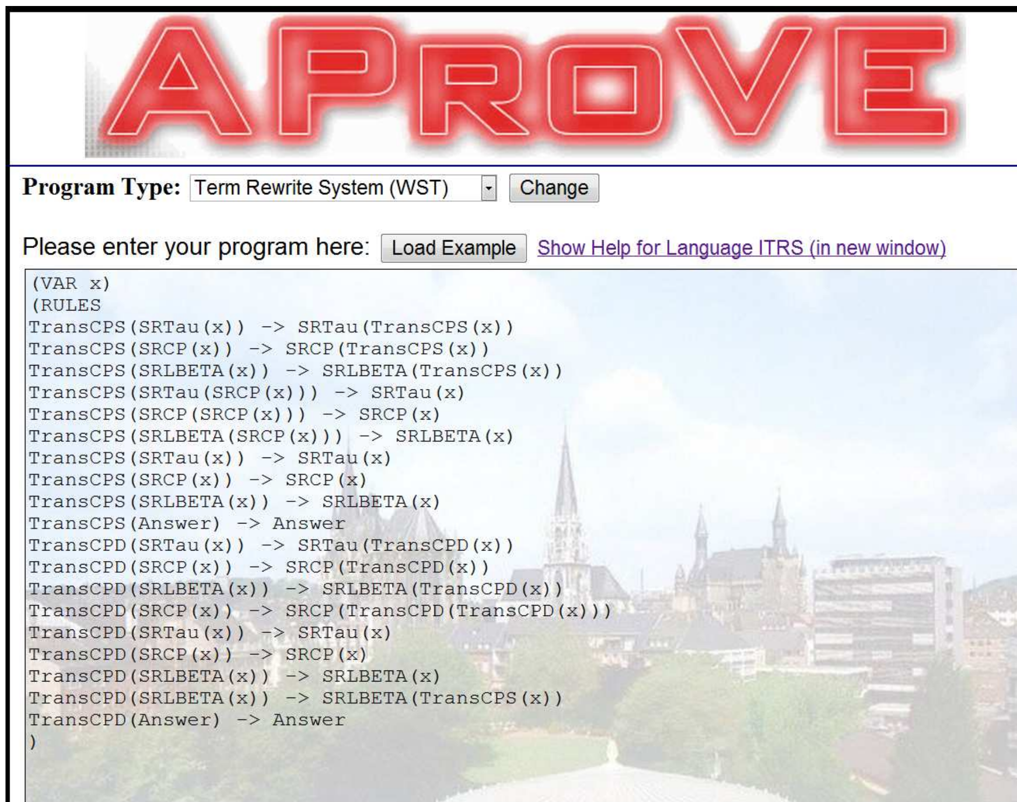



Abbildung 5.3: AProVE-Eingabe (TRS) soll auf Terminierung überprüft werden

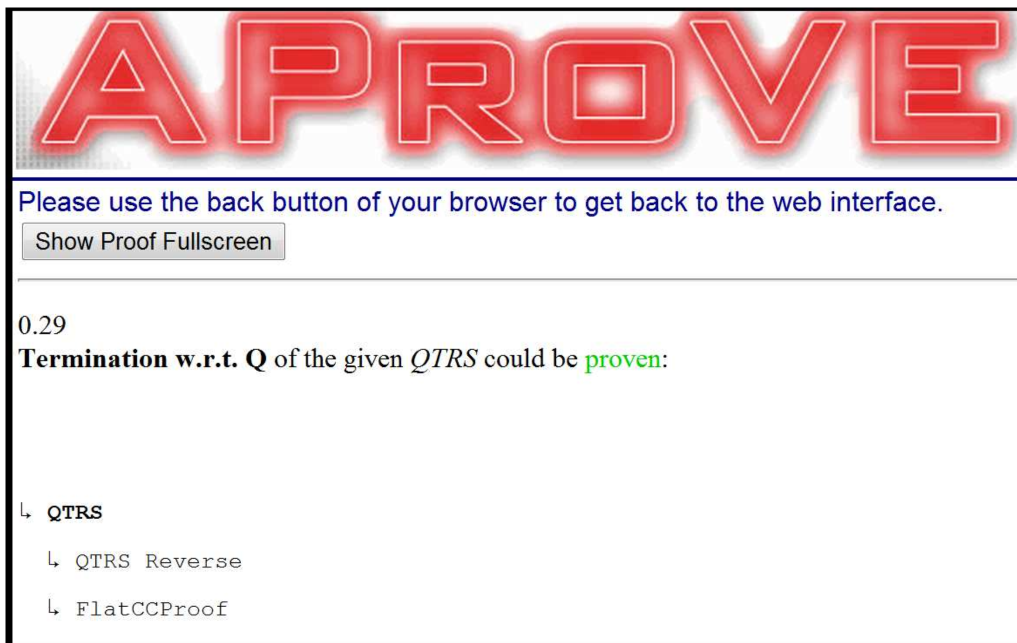



Abbildung 5.4: Terminierung erfolgreich von AProve gezeigt

# Kapitel 6

# Zusammenfassung und Ausblick

Mit diesem Kapitel möchte ich meine Arbeit abschließen.
Ich werde zusammenfassend die Schritte und den Schwerpunkt meiner Arbeit erläutern, anschließend ein Fazit aus den gewonnen Erkenntnissen ziehen und zuletzt einen Ausblick auf weitere Forschungen in der Richtung der Terminierungsbeweise und Korrektheitsnachweise geben.

## 6.1 Zusammenfassung

Blicken wir auf die Arbeit zurück und ziehen ein abschließendes Resümee.
Der Ausgangspunkt dieser Arbeit war die Problematik des Nachweises der Korrektheit von Programmen im Allgemeinen. Dafür wurde die Bedeutung verdeutlicht, die die Korrektheit von Programmen in unserem heutigen Zeitalter darstellt, zumal Software aus unserem Alltag und verantwortungsvollen Bereichen nicht wegzudenken ist und nicht an Glaubwürdigkeit verlieren darf, da schließlich immer größere Projekte mit ihrer Hilfe verwirklicht werden sollen und müssen.

Insbesondere bezog ich mich dabei auf Korrektheitsnachweise mit Hilfe eines Gleichheitsbegriffs, der den Prozess der Übersetzung innerhalb von Compilern überprüft. Ich führte dafür den Gleichheitsbegriff der kontextuellen Gleichheit ein, die als Methode zur Überprüfung von Korrektheit von Programmtransformationen dient und sich speziell auf die Terminierung zweier Programme bezieht.
Während es simple Lösungen zum Nachweis inkorrekter Transformationen gibt, ist der Nachweis der Korrektheit unentscheidbar, was direkt mit dem unentscheidbaren Halte- und Äquivalenzproblem zusammenhängt.
Dennoch gibt es Möglichkeiten, den Induktionsbeweis der Terminierung zu automatisieren, was ich mit dieser Arbeit zeigen wollte.

Genauer geht es hierbei um die Induktion der Diagrammbeweise, für die ich die Diagrammmethode ausführlich beschrieben und vorgestellt habe. Anhand von formalen Definitionen, unterstützenden Beispielen und der Implementierung meines

Programms habe ich die stückweise Umformung von gegebenen Diagrammen in Termersetzungssysteme erläutert und dokumentiert.

Dahinter steckte die Absicht, durch die nachgewiesene Terminierung des Termersetzungssystems rückschließend die Terminierung der Anwendung aller Diagramme zu beweisen.

Durch erfolgreiche Tests und die Darstellung dieser ist mir der Nachweis der Terminierung der sogenannten Forking-Diagramme gelungen. Diese Form der Beweisführung kann in weiteren Bereichen für Korrektheits- und Terminierungsbeweise eingesetzt werden.

## 6.2 Fazit

Die Frage nach der Terminierung von Programmen ist elementar in der Informatik und gehört in die Komplexitätsklassen der unentscheidbaren Probleme. Das bedeutet, dass es kein Entscheidungsverfahren gibt, das für jedes Programm die Frage nach der Terminierung beantworten kann.
Dennoch lassen sich Verfahren entwickeln, die durch Induktionsbeweise und geschickte Umformungen wertvolle Antworten auf die Frage nach Korrektheit und Terminierung finden können.
Da AProVE nur die Innermost-Terminierung von ITRS zeigen kann, ist die verwendete Kodierung in NARS wesentlich komplizierter als gedacht. Ansonsten stimmt das Ergebnis optimistisch bei der Forschung nach effizienten, verlässlichen Programmen und Software, deren Fähigkeiten und Grenzen sich besser einschätzen lassen.

## 6.3 Ausblick

Mit dieser Diagrammmethode ließen sich weitere, komplexere Diagramme untersuchen und auf Terminierung testen. Als weitere Arbeiten könnten auch die Methoden zur automatischen Terminierungsverifizierung erweitert werden, so dass zum Einen komplexere Diagramme verarbeitet werden können, zum anderen sollten weitere Beispiele und Beispielkalküle mithilfe des erstellten Programms getestet werden.

# Literaturverzeichnis


[Dem12] Demirsan, Aybike: *Implementierung BA*. 2012. – http://www.ki.informatik.uni-frankfurt.de/bachelor/programme/demirsan/

[has12] *Haskell*. 2012. – http://www.haskell.org/haskellwiki/Haskell

[JGT12] J. Giesl, P. Schneider-Kamp ; Thiemann, R.: *AProVE*. http://aprove.informatik.rwth-aachen.de/index.asp?subform=termination_proofs.html. Version: 2012. – Zugriff: 01.07.2012

[Lev93] Leveson, Nancy G.: An Investigation of the Therac-25 Accidents. In: *IEEE Computer* 26 (1993), S. 18–41

[OGL06] Ober, Iulian ; Graf, Susanne ; Lesens, David: A case study in UML model-based validation: The Ariane-5 launcher. In: *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006* Bd. 4037, 2006 (LNCS)

[RSS11] Rau, Conrad ; Schmidt-Schauss, Manfred: Towards Correctness of Program Transformations Through Unification and Critical Pair Computation / Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main. 2011 (41). – Frank report

[RSSS12] Rau, Conrad ; Sabel, David ; Schmidt-Schauss, Manfred: Encoding Induction in Correctness Proofs of Program Transformations as a Termination Problem. In: Moser, Georg (Hrsg.): *12th International Workshop on Termination, WST 2012, February 19-23, 2012, Obergurgl, Austria*, 2012, S. 74–78. – Published online and open access by G. Moser, Institute of Computer Science, University of Innsbruck, Austria

[SS12] Schmidt-Schauss, Manfred: *Skript „Reduktionssysteme und Termersetzung " zur Vorlesung „Automatische Deduktion", Sommersemester 2012*. Institut für Informatik. Fachbereich Informatik und Mathematik : htpp://www.ki.informatik.uni-frankfurt.de/SS2012/AD, 2012

[SSSS08] Schmidt-Schauss, Manfred ; Schütz, Marko ; Sabel, David: Safety of Nöcker's Strictness Analysis. In: *Journal of Functional Programming* 18 (2008), Nr. 04, S. 503–551. http://dx.doi.org/10.1017/S0956796807006624. – DOI 10.1017/S0956796807006624