Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Biologie und Informatik
Institut für Informatik

Diplomarbeit

Untersuchungen zu einigen Problemklassen des Kontext-Matching und Implementierung ausgewählter Algorithmen in der funktionalen Programmiersprache Haskell

Jörn Gersdorf

29. Juni 2004

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

Danksagung

Ich möchte mich bei all denjenigen Freunden bedanken, die mich während der Entstehung dieser Arbeit unterstützt und begleitet haben, besonders Heike Kramer und Frank Reffel.

Danielle - danke für all Deine Hilfe. Wie sonst auch hast Du mir während der stressigsten Phase der Entstehung dieser Arbeit so wundervoll zur Seite gestanden und mir stets neue Energie gegeben. Es ist wunderschön zu wissen, dass Du für mich da bist!

Mein besonderer Dank gilt Matthias Mann und Prof. Schmidt-Schauss für ihre hervorragende Betreuung.

Jörn Gersdorf

Erklärung gemäß DPO §11 Abs. 11

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

Frankfurt am Main, den 29. Juni 2004

J ö r n G e r s d o r f

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1. Motivation

Das Lösen von Unifikationsproblemen sowie der als eingeschränkte Unifikationsprobleme anzusehenden Matchingprobleme ist ein zentraler Prozess des automatisierten Schließens. Die Theorie, die sich mit diesen Problemen befasst - die Unifikationstheorie - abstrahiert die zugrundeliegenden Probleme und stellt einen formalen Rahmen dar, innerhalb dessen zentrale Begriffe wie Instanziierung oder Allgemeinster Unifikator definiert, Teilprobleme untersucht, Komplexitätsbetrachtungen angestellt und optimierte Lösungsalgorithmen für wichtige Teilinstanzen zur Verfügung gestellt werden.

Diese Diplomarbeit befasst sich mit einem kleinen Ausschnitt des umfangreichen Themengebietes: den Kontextmatchingproblemen, ihrer Komplexität sowie mit einigen Algorithmen, die Kontextmatchingprobleme lösen können. Die Algorithmen sind in der puren, statisch getypten, funktionalen Programmiersprache Haskell implementiert worden.

1.2. Einführung

Unifikationsprobleme befassen sich mit der Fragestellung, wie für zwei Terme s, t und das gegebene Problem $s \approx t$ eine Substitution σ für die freien Variablen in s und t zu finden ist, so dass $\sigma(s) = \sigma(t)$ gilt.

Dies bedeutet allgemein gesprochen, dass versucht wird, die beiden Terme s und t zu einer syntaktischen Identität zu bringen, indem manche Subausdrücke (die freien Variablen) durch andere Ausdrücke ersetzt werden. Dabei werden Terme aus n-ären Funktionssymbolen (incl. Konstanten mit Stelligkeit 0) und Variablen zusammengesetzt.

Beispiel 1.2.1. Das Unifikationsproblem $s \approx t$ mit $s := f(a, x)$ und $t := f(y, b)$ (mit f als 1-stelliger Funktion sowie a und b als Konstanten) stellt die Frage, ob es möglich ist, die Variablen x und y dergestalt durch neue Terme zu ersetzen, dass $s = t$ gilt. Dies ist möglich, indem x durch b und y durch a ersetzt werden. Wir erhalten damit die Lösung des Unifikationsproblems als Substitution $\sigma = \{\langle b/x\rangle, \langle a/y\rangle\}$. Diese Lösung wird als *Unifikator* bezeichnet.

Matchingprobleme sind eine Untermenge der Unifikationsprobleme; bei ihnen wird gefordert, dass der Term t keine freien Variablen enthält, so dass für die gesuchte Substitution σ nur noch $\sigma(s) = t$ gelten muss.

Syntaktische Unifikation. Diese Form der Unifikation wird auch als *syntaktische Unifikation erster Ordnung* bezeichnet. Dabei meint „syntaktische Unifikation", dass die Terme s und t durch Substitution syntaktisch identisch werden müssen. „Unifikation erster Ordnung" bedeutet, dass die freien Variablen nur Variablen erster Ordnung sein dürfen, d. h. insbesondere keine Variablen für Funktionen. So können die Terme $f(a, x)$ und $g(x, a)$ mittels syntaktischer Unifikation erster Ordnung nicht unifiziert werden, da $f \neq g$. Demgegenüber ist das Problem $f(a, x) \approx G(x, a)$ (wobei G eine Variable zweiter Ordnung ist) mit Unifikation höherer Ordnung lösbar, indem man die Substitution $\sigma = \{\langle a/x\rangle, \langle f/G\rangle\}$ als Unifikator wählt.

Gleichungsunifikation. Werden zusätzliche Identitäten oder Reduktionsregeln angegeben, die beschreiben, wann zwei Terme semantisch äquivalent sind, so spricht man von *Gleichungsunifikation.* Bei einem gegebenen Regelwerk E wird dann von der Substitution σ gefordert, dass $\sigma(s) =_E \sigma(t)$ gilt. Dies bedeutet, dass nach Anwendung der Substitution σ und einiger Regeln aus E die Terme s und t identisch werden. Im Fall der syntaktischen Unifikation gilt $E = \emptyset$.

Bei der Gleichungsunifikation wird also nicht syntaktische Gleichheit beider Gleichungsseiten gefordert, sondern Gleichheit modulo des Regelwerks E.

Beispiel. Wählt man $E := \{f(a, a) \leftrightarrow g(a, a)\}$, so ist das Unifikationsproblem $f(a, x) \approx g(x, a)$ bezüglich E lösbar mittels des E-Unifikators $\sigma = \{\langle a/x\rangle\}$, da gilt $\sigma(f(a, x)) = f(a, a) =_E g(a, a) = \sigma(g(x, a))$.

Allgemeinster Unifikator. Die Fragestellung, ob ein Unifikationsproblem grundsätzlich eine Lösung besitzt oder nicht, nennt man *Unifikations-Entscheidungsproblem.*

Besonders in praktischen Anwendungsfällen wird man im Fall der Lösbarkeit jedoch auch daran interessiert sein, durch den verwendeten Lösungsalgorithmus einen Unifikator, d. h. eine geeignete Substitution, geliefert zu bekommen. Allerdings gibt es Fälle, in denen ein Unifikationsproblem unendlich viele Unifikatoren besitzt. Ein Beispiel hierfür ist das Problem $f(x, y) \approx f(y, x)$, das gelöst werden kann, in dem x und y durch denselben Term ersetzt werden. Da es unendlich viele Terme gibt, gibt es auch unendlich viele Lösungen. Allerdings kann man diese Lösungsmenge auch durch Angabe eines *allgemeinsten Unifikators* spezifizieren, in diesem Fall $\sigma = \{\langle x/y\rangle\}$. Durch *Instanziierung* können alle Lösungen dargestellt werden: bei Wahl eines festen Terms s zum Beispiel durch $\sigma\{\langle s/x\rangle\} = \{\langle x/y\rangle, \langle s/x\rangle\}$.

1.3. Überblick

In Kapitel 2 werden die theoretischen Grundlagen für Unifikation und Matching gelegt. Dabei gehen wir über den abgegrenzten Bereich des Kontextmatchings hinaus, einerseits, weil viele allgemeine Begriffe und algorithmische Grundlagen auch für das Kontextmatching relevant sind, andererseits, um eine Einordnung von Kontextmatching in die globalere Theorie vornehmen zu können. Es werden nicht nur Definitionen und grundlegende Erkenntnisse sondern auch Algorithmen vorgestellt.

Danach wird in Kapitel 3 das Gebiet des Kontextmatchings untersucht. Zunächst wird auf die Besonderheiten dieser Problemklasse eingegangen, um dann die Komplexität verschiedener Subprobleme darzulegen. Im Anschluß werden Algorithmen zur Lösung von Kontextmatchingproblemen vorgestellt. Dabei werden die Algorithmen von ihrer theoretischen Seite beleuchtet.

Gegenstand von Kapitel 4 ist die Implementierung der Algorithmen aus Kapitel 3. Mit Ausnahme eines Algorithmus wurden alle dort vorgestellten Verfahrensweisen realisiert; hierbei kam die funktionale Programmiersprache Haskell zur Anwendung. Es werden sowohl die verwendeten Datenstrukturen als auch solche Ausschnitte aus dem Programmcode besprochen, die besonders interessant erscheinen.

Schließlich widmet sich Kapitel 5 der Analyse der Verhaltensweise des implementierten Systems und untersucht in einem Ausblick, welche Schlussfolgerungen aus der Analyse für weitere Verbesserungen der Algorithmen zur Lösung des Kontextmatching gezogen werden können.

KAPITEL 2

Grundlagen

Das Ziel dieses Kapitels ist es, einen allgemeinen Überblick über das Feld der Unifikationsprobleme zu geben. Wie später noch in Kapitel 3.1 erläutert wird, liegen die Kontextunifikations- und Matchingprobleme thematisch zwischen der syntaktischen Unifikation und der Unifikation zweiter Ordnung. Daher beschränken wir uns nicht auf das Feld der syntaktischen Unifikation, sondern stellen auch das allgemeinere Gebiet der Unifikation höherer Ordnung vor.

Wir beginnen mit einer allgemeinen Einführung in die Syntaxunifikation. In diesem Rahmen werden nicht nur Definitionen vorgenommen, sondern es werden auch Lösungsalgorithmen nebst Komplexitätsbetrachtungen untersucht.

Als Vorbereitung für die Unifikation höherer Ordnung wird in Abschnitt 2.3 eine kurze Einführung in den einfach getypten λ-Kalkül gegeben. Im Abschnitt 2.4 werden Unifikation und Matching dann formal definiert und es wird ein Überblick über mögliche Teilprobleme dieses Bereiches gegeben. Ferner wird die Unentscheidbarkeit der allgemeinen Unifikation gezeigt.

In Abschnitt 3.4 werden Lösungsalgorithmen für die allgemeine Unifikation vorgestellt. Obwohl unentscheidbar scheint es doch sinnvoll zu sein, über Lösungsverfahren für die Unifikation nachzudenken.

Wir schließen das Kapitel mit Betrachtungen über Matchingprobleme höherer Ordnung.

2.1. Syntaktische Unifikation

Die syntaktische Unifikation versucht, Terme mit Hilfe von Substitutionen freier Variablen zu unifizieren, das heißt syntaktisch identisch zu machen. Wir werden zunächst die grundlegenden Definitionen wiedergeben, danach einen einfachen Lösungsalgorithmus angeben, ihn auf Korrektheit untersuchen und schließlich seine Komplexität betrachten.

Die Darstellungen basieren auf [**Baad1998, Baad1999**].

2.1.1. Grundlegende Begriffe. Wir definieren zunächst eine Termalgebra.

Definition 2.1.1. Eine *Signatur* Σ ist eine Menge von Funktionssymbolen, wobei jedem $f \in \Sigma$ eine nichtnegative Ganzzahl $\mathtt{ar}(f)$ zugeordnet wird, die die *Stelligkeit* von f genannt wird. Ein Funktionssymbol mit $\mathtt{ar}(f) = 0$ wird

auch als *konstantes Symbol* bezeichnet. Die Menge der Funktionssymbole in Σ mit Stelligkeit n bezeichnen wir mit $\Sigma^n \subseteq \Sigma$.

Sei Σ nun eine solche Signatur und V eine Menge von *Variablen*, so dass $\Sigma \cap V = \emptyset$. Dann definieren wir die Menge der Σ-*Terme* (oder kurz Terme) $T(\Sigma, V)$ über V induktiv wie folgt:

(1) jede Variable ist ein Term: $\forall x \in V : x \in T(\Sigma, V)$
(2) die Applikation eines Funktionssymbols auf Terme ergibt wieder einen Term:

$$\forall n \geq 0, \forall f \in \Sigma^n, \forall t_1, \ldots, t_n \in T(\Sigma, V) : f(t_1, \ldots, t_n) \in T(\Sigma, V)$$

Wir werden im folgenden Teil die Buchstaben s, t und u für Terme, die Buchstaben x, y und z für Variablen sowie die Bezeichner f, g und h für Funktionssymbole verwenden. Weiter benutzen wir a, b und c für konstante Symbole (also z. B. $ar(a) = 0$).

Es gilt somit zum Beispiel $f(g(a, b), h(x)) \in T(\Sigma, V)$ mit $\Sigma = \{a, b, f, g, h\}$ und $V = \{x\}$. Dabei gilt weiter $\Sigma^0 = \{a, b\}$, $\Sigma^1 = \{h\}$, $\Sigma^2 = \{f, g\}$.

Definition 2.1.2. Die *Menge der Variablen* $\texttt{Var}(s)$ eines Terms $s \in T(\Sigma, V)$ ist definiert als

- $\texttt{Var}(x) := \{x\} \, \forall x \in V$
- $\texttt{Var}(f(s_1, \ldots, s_n)) := \bigcup_{1 \leq i \leq n} \texttt{Var}(s_i)$

Falls $\texttt{Var}(s) = \emptyset$, so wird s auch als *Grundterm* bezeichnet.

Für konkrete Diskussionen über Subterme ist es hilfreich, eine Notation für Positionen einzuführen:

Definition 2.1.3. Sei Σ eine Signatur und V eine Menge von Variablen mit $\Sigma \cap V = \emptyset$. Seien $s, t \in T(\Sigma, V)$ Terme.

(1) Die *Menge der Positionen* eines Terms s ist eine Menge von Strings aus natürlichen Zahlen $\texttt{Pos}(s)$, wobei jeder String genau eine Position im Termbaum von s und damit einen Subterm bezeichnet.
   - Falls $s = x$, dann ist $\texttt{Pos}(s) := \{\epsilon\}$
   - Falls $s = f(t_1, \ldots, t_n)$, dann ist

$$\texttt{Pos}(s) := \{\epsilon\} \cup \bigcup_{1 \leq i \leq n} \{ip | p \in \texttt{Pos}(t_i)\}$$

   Die Position ϵ wird auch als Wurzel bezeichnet.
(2) Damit können wir die *Menge der Subterme* definieren als $\texttt{Sub}(t) = \{s|_p | p \in \texttt{Pos}(t)\}$.
(3) Wir bezeichnen eine Position $p \in \texttt{Pos}(s)$ auch als *(Term-)Pfad*, wenn wir damit ausdrücken wollen, dass wir über die Knoten reden wollen, die in s auf dem Weg von der Wurzel bis zu p liegen.
(4) Für ein $p \in \texttt{Pos}(s)$ wird der Subterm an der Stelle p durch eine Induktion über die Länge von p definiert:
   - $s|_\epsilon := s$
   - $f(s_1, \ldots, s_n)|_{iq} := s_i|_q$

(5) Für einen Term s und eine Position $p \in \mathtt{Pos}(s)$ wird die Ersetzung eines Subterms in s durch einen anderen Term t an der Position p mit $s[t]_p$ bezeichnet und wie folgt definiert:
- $s[t]_\epsilon := t$
- $f(s_1, \ldots, s_n)[t]_{iq} := f\left(s_1, \ldots, s_{i-1}, s_i[t]_q, s_{i+1}, \ldots, s_n\right)$

(6) Seien s ein Term und $p \in \mathtt{Pos}(s)$. Dann definiert $s' := \frac{s}{p}$ denjenigen monadischen Term s', der durch das Aufsammeln sämtlicher Termknoten auf dem Weg von der Wurzel von s zu $s|_p$ entsteht. Wir bezeichnen $\frac{s}{p}$ als *Pfadterm*.

Bemerkung. Eine Position $p \in \mathtt{Pos}(s)$ definiert also einen Pfad durch s von der Wurzel von s ($s|_\epsilon$) bis zum Zielterm $s|_p$.

Beispiel. Sei $s = f(g(a,b), h(d))$. Dann ist $\mathtt{Pos}(s) = \{\epsilon, 1, 11, 12, 2, 21\}$. $s|_{12} = b$. $s[x]_1 = f(x, h(d))$. $\frac{s}{12} = f(g(b))$, $\frac{s}{1} = f(g)$, $\frac{s}{\epsilon} = f$.

Definition 2.1.4. Sei t ein Term. Seien s und t' Subterme von t mit Positionen p_s und $p_{t'}$, also $s = t|_{p_s}$, $t' = t|_{p_{t'}}$. Seien mit $p_s = i_1 \ldots i_n$ und $p_{t'} = j_1 \ldots j_m$ die Komponenten der Positionsstrings angegeben. Wir sagen, dass sich s *links* von t befindet, genau dann wenn $\exists k : 1 \leq k \leq \mathtt{max}(m,n) : i_k < j_k \wedge \forall l : 1 \leq l < k : i_l = j_l$. Der Begriff *rechts* ist analog definiert.

2.1.2. Substitutionen. Wir kommen nun zu einem für die Unifikation zentralen Konzept: der Substitution von Variablen durch beliebige andere Terme. Wir benötigen hierzu eine Substitutionsfunktion σ, die die Substitution erklärt.

Definition 2.1.5. Sei Σ eine Signatur, V eine von Σ disjunkte abzählbar unendliche Menge von Variablen. Eine *Substitution* ist eine Funktion $\sigma : V \to T(\Sigma, V)$, die endlich vielen $x \in V$ einen Wert $\sigma(x) \neq x$ zuweist und für alle anderen Variablen die Identität ist.

Die Menge von Variablen, für die σ nicht der Identität entspricht, nennt man *Domäne* von σ: $\mathtt{Dom}(\sigma) := \{x | x \in V, \sigma(x) \neq x\}$. Falls gilt $\mathtt{Dom}(\sigma) = \{x_1, \ldots, x_n\}$, so kann man die Substitution wie folgt notieren:

$$\sigma = \{\langle \sigma(x_1)/x_n \rangle, \ldots \langle \sigma(x_n)/x_n \rangle\}$$

Der *Wertebereich* von σ ist die Menge der Substitute:

$$\mathtt{Ran}(\sigma) := \{\sigma(x) | x \in \mathtt{Dom}(\sigma)\}$$

Der *Variablenwertebereich* von σ ist dagegen die Menge der Variablen, die im Wertebereich $\mathtt{Ran}(\sigma)$ auftauchen, also die Variablen, die durch σ (neu) eingeführt werden: $\mathtt{VRan}(\sigma) := \bigcup_{x \in \mathtt{Dom}(\sigma)} \mathtt{Var}(\sigma(x))$.

Bis jetzt ist die Substitution σ lediglich auf Variablen definiert. Die Definition kann jedoch auf natürliche Weise auf $T(\Sigma, V)$-Terme erweitert werden:

- Falls $t = x \in V$, so gilt $\sigma'(t) := \sigma(t)$
- Falls $t = f(s_1, \ldots, s_n)$, so gilt $\sigma'(t) := f(\sigma'(s_1), \ldots, \sigma'(s_n))$

In der Folge verwenden wir σ und σ' synonym und meinen in der Regel σ', wenn wir von σ reden. Manchmal schreibt man für $\sigma(t)$ auch $t\sigma$.

Eine Substitution mit $\mathtt{Dom}(\sigma) = \mathtt{Ran}(\sigma)$ nennt man eine *Variablenpermutation*. Ein Beispiel: $\sigma_1 = \{\langle y/x\rangle, \langle z/y\rangle, \langle x/z\rangle\}$ ist eine solche Variablenpermutation, $\sigma_2 = \{\langle x/y\rangle\}$ dagegen nicht, da $\mathtt{Dom}(\sigma_2) = \{y\} \neq \{x\} = \mathtt{Ran}(\sigma_2)$.

Definition. Die *Komposition* von zwei Substitutionen σ und θ ist definiert als $t\sigma\theta = (\theta(t))\sigma := \sigma(\theta(t))$. Anders ausgedrückt: $\sigma\theta(t) = (\sigma \circ \theta)(t) = \sigma(\theta(t))$.

Die Komposition $\sigma\theta$ ist eine Abbildung von V nach $T(\Sigma, V)$. Weiterhin gilt $\sigma\theta(x) = x$ für alle $x \in V \setminus (\mathtt{Dom}(\sigma) \cup \mathtt{Dom}(\theta))$. Damit ist $\sigma\theta$ wieder eine Substitution. Die Komposition von Substitutionen $\sigma\theta$ kann wie folgt berechnet werden:

(1) Wende σ auf jeden Term $t \in \mathtt{Ran}(\theta)$ an:
$$\theta' := \{\langle \sigma(t)/x\rangle \mid x \in \mathtt{Dom}(\theta), t = \theta(x)\}$$
Damit wird auf jeden Term $\theta(x)$, den die Substitution θ für eine Variable einsetzen würde, die Substition σ angewandt und anschließend eine neue Ersetzung definiert. Dies definiert im Grunde schon die Komposition der beiden Substitutionen.
(2) Entferne aus σ jede Bindung einer Variablen x, die bereits in θ vorkommt: $\sigma' := \sigma \setminus \{\langle \sigma(x)/x\rangle \mid x \in \mathtt{Dom}(\theta)\}$.
(3) Entferne aus θ' alle trivialen Bindungen: $\theta'' := \theta' \setminus \{\langle x/x\rangle \mid x \in \theta'\}$.
(4) Bilde die Vereinigung aus σ' und θ'': $\sigma\theta := \sigma' \cup \theta''$.

Definition. Eine Substitution heißt *idempotent*, wenn ihre mehrfache Anwendung auf einen beliebigen Term $t \in T(\Sigma, V)$ das Ergebnis nicht ändert: $t\sigma\sigma = t\sigma$.

Lemma. *Eine Substitution σ ist idempotent gdw.* $\mathtt{Dom}(\sigma) \cap \mathtt{VRan}(\sigma) = \emptyset$.

Beweis. Sei σ idempotent. Dann gilt für ein beliebiges $t \in T(\Sigma, V)$ $t\sigma\sigma = t\sigma$. Wir nehmen an $\mathtt{Dom}(\sigma) \cap \mathtt{VRan}(\sigma) \neq \emptyset$. Damit existiert ein $x \in \mathtt{Dom}(\sigma) \cap \mathtt{VRan}(\sigma)$. Wir nehmen weiterhin an, dass dieses x in t vorkommt. Damit kommt es auch in $t' := t\sigma$ vor. Da mit $x \in \mathtt{Dom}(\sigma)$ automatisch gilt $\sigma(x) \neq x$, aber auch $x \in \mathtt{VRan}(\sigma)$, kommt x auch wieder in t' vor. Bei einer weiteren Anwendung der Substitution $\sigma(t')$ wird die Variable erneut ersetzt, so dass gilt $t\sigma\sigma \neq t\sigma$, was ein Widerspruch ist.

Umgekehrt sei $\mathtt{Dom}(\sigma) \cap \mathtt{VRan}(\sigma) = \emptyset$. Sei t erneut ein $T(\Sigma, V)$-Term. Jede Variable $x \in \mathtt{Dom}(\sigma)$, die in t vorkommt, wird durch die Substitution durch einen Term s ersetzt, wobei s wegen der Voraussetzung die Variable x nicht mehr enthält. Daher kann eine erneute Anwendung von σ den Term $\sigma(t)$ nicht mehr verändern. □

Es stellt sich heraus, dass manche Substitutionen gleichermaßen Instanzen anderer Substitutionen sind.

Beispiel 2.1.6. Zum Beispiel kann man $\theta = \{\langle f(a)/x\rangle, \langle a/y\rangle\}$ auch darstellen als Instanz von $\sigma = \{\langle f(y)/x\rangle\}$, indem man die zusammengesetzte Substitution $\{\langle a/y\rangle\}\sigma$ bildet. σ ist dann allgemeiner als θ.

DEFINITION 2.1.7. Eine Substitution σ ist *allgemeiner* als eine Substitution θ, falls es eine Substitution τ gibt, so dass $\theta = \tau\sigma$. Wir schreiben $\sigma \lesssim \theta$.

Im Beispiel 2.1.6 gilt also $\sigma \lesssim \theta$, wobei $\tau = \{\langle a/y\rangle\}$.

LEMMA. *Die Relation $\lesssim$ auf Substitutionen ist eine Quasi-Ordnung.*

BEWEIS. Für die Reflexivität wählt man für τ die Identität.

Für die Transitivität ist zu zeigen, dass aus $\sigma_1 \lesssim \sigma_2$ und $\sigma_2 \lesssim \sigma_3$ folgt $\sigma_1 \lesssim \sigma_3$. Aus der Voraussetzung folgt zunächst $\sigma_2 = \tau_1\sigma_1$ und $\sigma_3 = \tau_2\sigma_2$. Dann aber ist $\sigma_3 = \tau_2\sigma_2 = \tau_2(\tau_1\sigma_1) = (\tau_1\tau_2)\sigma_1$. Das ist jedoch genau $\sigma_1 \lesssim \sigma_3$. □

Die Relation $\lesssim$ ist keine partielle Ordnung und daher auch nicht antisymmetrisch. Als Beispiel dafür wähle man $\sigma_1 = \{\langle x/y\rangle\}$ und $\sigma_2 = \{\langle y/x\rangle\}$. Es gilt nun $\sigma_2 = \sigma_2\sigma_1$ und damit $\sigma_1 \lesssim \sigma_2$ und andererseits aber auch $\sigma_2 \lesssim \sigma_1$ wegen $\sigma_1 = \sigma_1\sigma_2$.

2.1.3. Unifikation. Wir kommen nun zur formalen Definition von Unifikationsproblemen.

DEFINITION 2.1.8. Eine $T(\Sigma, V)$-*Unifikationsgleichung* ist ein Paar von $T(\Sigma, V)$-Termen, geschrieben $s \approx t$.

Ein $T(\Sigma, V)$-*Unifikationsproblem* Γ ist eine endliche Menge von Unifikationsgleichungen $\Gamma = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$.

Eine *Lösung* von Γ ist eine Substitution σ, so dass $\forall 1 \leq i \leq n : s_i\sigma = t_i\sigma$. Die Lösung wird auch *Unifikator* genannt. Die Menge aller Unifikatoren von Γ wird mit $U(\Gamma)$ bezeichnet. Γ ist lösbar oder unifizierbar falls $U(\Gamma) \neq \emptyset$.

Schränkt man Unifikationsprobleme hinsichtlich der freien Variablen auf der rechten Seite ein, erhält man die Klasse der Matchingprobleme:

DEFINITION 2.1.9. Ein *Matchingproblem* $\Gamma = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ ist ein Unifikationsproblem, bei dem die rechten Seiten der Gleichungen keine freien Variablen enthalten, d. h. alle t_i sind Grundterme. Damit ist ein solches Matchingproblem gelöst gdw. es eine Substitution σ gibt mit $\forall 1 \leq i \leq n : s_i\sigma = t_i$.

Unter allen Unifikatoren $\sigma \in U(\Gamma)$ gibt es Unifikatoren, die allgemeiner sind als andere.

DEFINITION 2.1.10. Ein Unifikator $\sigma \in U(\Gamma)$ ist ein *allgemeinster Unifikator* von Γ, falls gilt

$$\forall \sigma' \in U(\Gamma) : \sigma \lesssim \sigma'$$

Wir kommen zu einem wichtigen Sonderfall unter den Unifikationsproblemen.

DEFINITION 2.1.11. Ein Unifikationsproblem $\Gamma = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ heißt *gelöst*, wenn alle x_i paarweise verschieden sind und kein x_i in einem t_i vorkommt. In diesem Falle erhält man den kanonischen Unifikator

$$\sigma_\Gamma := \{\langle t_1/x_1\rangle, \ldots, \langle t_n/x_n\rangle\}$$

Eine freie Variable x in einem Unifikationsproblem Γ heißt *gelöst*, falls sie genau einmal in Γ vorkommt, nämlich auf der linken Seite einer Gleichung $x \approx t$ mit $x \notin Var(t)$.

σ_Γ ist stets eine idempotente Substitution. Diese Eigenschaft geht aber noch weiter, wie das folgende Lemma zeigt.

LEMMA 2.1.12. *Falls Γ ein gelöstes Unifikationsproblem ist, dann ist σ_Γ für alle Unifikatoren $\sigma \in U(\Gamma)$ eine allgemeinere Substitution: $\sigma = \sigma\sigma_\Gamma$ bzw. $\sigma_\Gamma \lesssim \sigma$.*

BEWEIS. Sei $\Gamma = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$. Wir zeigen, dass sich die Substitutionen σ und $\sigma\sigma_\Gamma$ bei allen Variablen $x \in V$ gleich verhalten, also dass $x\sigma = x\sigma\sigma_\Gamma$ gilt.

(1) Falls $x \in \{x_1, \ldots, x_n\}$. Sei $x = x_k$. Dann gilt wegen $\sigma \in U(\Gamma)$ und weil σ daher idempotent ist: $x\sigma = t_k\sigma = x\sigma\sigma_\Gamma$.
(2) Falls $x \notin \{x_1, \ldots, x_n\}$. Dann gilt wegen $x = x\sigma_\Gamma$: $x\sigma = x\sigma\sigma_\Gamma$.

□

LEMMA 2.1.13. *Falls Γ ein gelöstes Unifikationsproblem ist, so ist σ_Γ ein idempotenter allgemeinster Unifikator von Γ.*

BEWEIS. Die Idempotenz folgt direkt aus der Definition eines gelösten Unifikationsproblems. Wegen $x_i\sigma_\Gamma = t_i = t_i\sigma_\Gamma$ gilt $\sigma_\Gamma \in U(\Gamma)$ und σ_Γ ist ein Unifikator von Γ. Mit Lemma 2.1.12 gilt für alle $\sigma \in U(\Gamma)$: $\sigma_\Gamma \lesssim \sigma$. Damit ist σ_Γ auch allgemeinster Unifikator von Γ. □

LEMMA 2.1.14. *Seien σ und τ zwei allgemeinste Unifikatoren eines Unifikationsproblems Γ. Dann sind σ und τ identisch bis auf eine Komposition mit einer Variablenpermutation, d. h. $\sigma = \theta\tau$ und umgekehrt.*

BEWEIS. Der Beweis ist in [**Lass1987, Baad1999**] enthalten. □

2.2. Algorithmen für Syntaktische Unifikation

In diesem Abschnitt werden solche Algorithmen vorgestellt und untersucht, die syntaktische Unifikation lösen können. Mit der Existenz dieser Algorithmen ist die Entscheidbarkeit syntaktischer Unifikation gegeben. Betrachtungen über die Komplexität der Algorithmen runden den Abschnitt ab. Der Abschnitt beruht auf Darstellungen aus [**Baad1999**].

2.2.1. Robinson-Algorithmus. Einer der am häufigsten in symbolischen Berechnungsprogrammen implementierte Unifikationsalgorithmen ist derjenige von Robinson, der ihn bereits 1965 vorgestellt hat.

Grundidee. Die Idee ist, die beiden zu unifizierenden Terme simultan zu traversieren, wobei nicht der Syntaxbaum durchlaufen wird, sondern die textuelle Darstellung der Terme.

(1) Man setze jeweils eine Markierung an den Beginn jedes Terms.
(2) Man bewege die Markierungen simultan. Pro Schritt wird ein Symbol betrachtet. Fahre fort, bis beide Markierungen das Ende der Terme erreicht haben (in diesem Falle sind die Terme erfolgreich unifiziert) oder bis die Markierungen auf unterschiedliche Symbole zeigen.
(3) Falls keines der beiden Symbole eine Variable ist, dann ist die Unifikation gescheitert (da unterschiedliche Funktionssymbole).
(4) Sonst ist eines der beiden Symbole eine Variable x und das andere die Wurzel eines Subterms t.
   (a) Falls x in t vorkommt, ist die Unifikation gescheitert (da eine Schleife gefunden wurde).
   (b) Sonst füge $\{\langle t/x\rangle\}$ zur Lösungsmenge hinzu und ersetze x in den zu unifizierenden Termen durch t.

   Gehe zu Schritt 2.

Es stellen sich unmittelbar die Fragen nach Korrektheit, Vollständigkeit und Komplexität. Wir werden auf diese zurückkommen.

Das Vorgehen ist als Algorithmus 2.2.1 noch einmal in Pascal-ähnlichem Pseudocode dargestellt. Dabei ist die Lösungsmenge σ eine Liste von Termpaaren. Terme werden intern als Graph mit Zeigern realisiert. Der Algorithmus verwendet eine absteigende Rekursion, indem er sich von der Wurzel aus durch die Terme bewegt.

Unifikation durch Transformation. Unifikation kann sehr gut als eine Folge von Transformationsschritten, die auf einer Menge von Gleichungen operieren, dargestellt werden. Die einzelnen Schritte werden durch eine Menge von Regeln vorgegeben, die eine Bedingung und eine Folge definieren. Wie wir sehen werden, entspricht das hier vorgestellte Transformationssystem genau dem Robinson-Algorithmus.

Wegen Lemma 2.1.12 wissen wir genau, wie wir aus einem gelösten Unifikationsproblem einen idempotenten allgemeinsten Unifikator erhalten.

Wir führen zunächst das spezielle Unifikationsproblem $\bot$ ein, das keine Lösung besitzt.

Definition 2.2.1. Mit $\bot$ (gesprochen „bot") bezeichnen wir die Menge von Unifikationsproblemen, die keine Lösung besitzen. Wir verwenden $\bot$ auch als Symbol für ein beliebiges, aber festes Unifikationsproblem ohne Lösung.

Die Transformationsregeln in Tabelle 2.2.1 führen uns von einem beliebigen Syntaxunifikationsproblem zu einer solchen gelösten Form, falls eine existiert.

Definition. Das Symbol $\uplus$ bezeichnet die disjunkte Vereinigung.

Die einzelnen Regeln können wie folgt erklärt werden:

Algorithmus 2.2.1 rekursiv absteigender Robinson-Algorithmus

```
global σ: substitution; /* initialisiert als Identität */

procedure Unify(s: term, t: term) {
  // bereits gelöste Variablen eliminieren
  if (s ∈ V)
    s:=sσ;
  if (t ∈ V)
    t:=tσ;
  // (*)
  // Berechnungsstart
  if (s ∈ V ∧ s = t) then
    /* nothing */ ;  // Gleichung löschen
  else if (s = f(s_1,...,s_n) ∧ t = g(t_1,...,t_m)
           ∧n ≥ 0 ∧ m ≥ 0) {
    if (f = g)
      for i := 1 to n do
        Unify(s_i, t_i);  // Dekomposition
    else
      FAIL;  // kollidierende Symbole
  }
  else if (s ∉ V)
    Unify(t, s);  // Orientieren
  else if (s kommt in t vor)
    FAIL;  // occurs check
  else
    σ := σ{⟨t/s⟩};  // Lösungsmenge erweitern
}
```

Löschen	$\{t \approx t\} \uplus S$	$\Rightarrow$	S
Dekomposition	$\left\{f\left(\vec{t}\right) \approx f\left(\vec{s}\right)\right\} \uplus S$	$\Rightarrow$	$\{t_1 \approx s_1, \ldots, t_n \approx s_n\} \cup S$
Orientierung	falls $t \notin V$: $\{t \approx x\} \uplus S$	$\Rightarrow$	$\{x \approx t\} \cup S$
Variablen-elimination	falls $x \in \mathtt{Var}(S) \setminus \mathtt{Var}(t)$: $\{x \approx t\} \uplus S$	$\Rightarrow$	$\{x \approx t\} \cup S\{\langle t/x\rangle\}$
Kollision	$\left\{f\left(\vec{t}\right) \approx g\left(\vec{s}\right)\right\} \uplus S$	$\Rightarrow$	$\bot$, falls $f \neq g$ oder $\left\lvert\vec{t}\right\rvert \neq \left\lvert\vec{s}\right\rvert$
Occurs Check	$\{x \approx t\} \uplus S$	$\Rightarrow$	$\bot$, falls $x \in \mathtt{Var}(t) \wedge x \neq t$

Tabelle 2.2.1. Transformationsregeln für syntaktische Unifikation

Löschen: Entfernt triviale Gleichungen.

Dekomposition: Ersetzt Gleichungen zwischen Termen mit gleichem Funktionssymbol durch Gleichungen zwischen den Subtermen des Funktionssymbols.

Orientierung: Schiebt Variablen auf die linke Seite von Gleichungen.

Variablenelimination: Ersetzt Variablen durch Lösungen und entfernt dadurch gelöste Variablen aus dem restlichen Problem.

Kollision: Erkennt ein unlösbares Problem, wenn Funktionssymbole nicht zusammenpassen.

Occurs Check: Erkennt ein unlösbares Problem, falls eine Variable auf der linken und rechten Seite einer Gleichung auftaucht.

NOTATION 2.2.2. Wenn eine Transformationsregel das Unifikationsproblem Γ in das Problem Γ' überführt, so schreiben wir dies als $\Gamma \rightsquigarrow \Gamma'$.

Der Algorithmus, der sich aus den Transformationsregeln ergibt, ist nichtdeterministisch. Alle Regeln sind don't care-nichtdeterministisch, d. h. falls in irgendeinem Stadium zwei Regeln anwendbar sein sollten, kann der Algorithmus eine beliebige dieser Regeln wählen.

Die Regeln KOLLISION und OCCURS CHECK haben ihre Berechtigung aus folgenden beiden Lemmata.

LEMMA 2.2.3. *Eine Gleichung $f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)$ mit $f \neq g$ hat keine Lösung.*

BEWEIS. Sei σ eine beliebige Substitution.

$$\sigma(f(s_1, \ldots, s_n)) = f(s_1\sigma, \ldots, s_n\sigma) \neq g(t_1\sigma, \ldots, t_m\sigma)$$

□

LEMMA 2.2.4. *Eine Gleichung $x \approx t$, wobei $x \neq t$ und x in t vorkommt ($x \in Var(t)$), hat keine Lösung.*

BEWEIS. Weil $x \neq t$ gilt, muss t von der Form $t = f(t_1, \ldots, t_n)$ sein. Dann gilt $\exists 1 \leq i \leq n : x \in Var(t_i)$. Nun muss allerdings für eine beliebige Substitution σ gelten: $|x\sigma| \leq |t_i\sigma| < |t\sigma|$ und damit gilt auch $x\sigma \neq t\sigma$ für alle Substitutionen σ. □

2.2.1.1. *Vergleich mit dem Robinson-Algorithmus.* Bevor wir uns der Korrektheit des Transformationsalgorithmus zuwenden, vergleichen wir ihn zunächst einmal mit dem Robinson-Algorithmus (Algorithmus 2.2.1).

Wenn man sich eine Ausgabe der Argumente s und t der Prozedur `Unify` sowie der globalen Substitution σ an der Stelle (*) im Programmtext vorstellt, so kann man eine folgende Ausgabesequenz erhalten:

$$(\langle s_1, t_1, Id\rangle, \langle s_2, t_2, \sigma_2\rangle, \langle s_3, t_3, \sigma_3\rangle, \ldots)$$

Diese Sequenz kann auch durch die Transformationsregeln simuliert werden:

$$\left(\{s_1 \approx t_1\}, \{s_2 \approx t_2\} \biguplus P_2, \{s_3 \approx t_3\} \biguplus P_3, \ldots\right)$$

Dabei ist $s_i \approx t_i$ jeweils die Gleichung, auf der eine Transformationsregel angewandt wird und jedes σ_i ist identisch mit σ_{Γ_i}. Insbesondere endet der Algorithmus genau dann mit `FAIL`, wenn die Transformationsregeln mit $\bot$ enden. Ein Aufruf von `Unify` endet erfolgreich mit einer globalen Substitution σ_u genau dann, wenn die Transformationsregeln mit einem gelösten Unifikationsproblem enden, und dann gilt $\sigma_u = \sigma_\Gamma$.

Die Simulation des rekursiven Algorithmus kann dadurch erfolgen, indem man die (Multi-)Menge S als Keller auffasst, immer eine Regel auf die oberste Gleichung in S anwendet und die Regel LÖSCHEN nur dann anwendet, wenn t eine Variable ist. In diesem Rahmen gibt es in jedem Berechnungsschritt genau eine Transformationsregel, die angewendet werden kann.

Daher ist der nichtdeterministische Algorithmus aufgrund der Transformationsregeln eine allgemeinere Version des Algorithmus 2.2.1. Falls die Korrektheit der Transformationsregeln nachgewiesen ist, so ist auch die Korrektheit des rekursiven Algorithmus nachgewiesen.

2.2.1.2. *Korrektheit.* Wir zeigen zunächst, dass die Anwendung der einzelnen Regeln aus Tabelle 2.2.1 für jede Eingabe eines syntaktischen Unifikationsproblems terminiert. Dies ist nicht trivial, da die Regel VARIABLENELIMINATION das Unifikationsproblem zunächst vergrößert.

LEMMA 2.2.5. *Die Anwendung der Transformationsregeln terminiert bei allen Eingaben eines Unifikationsproblems* Γ*.*

BEWEIS. Wir geben zunächst eine Funktion an, die ein Komplexitätsmaß für ein Unifikationsproblem Γ berechnet. Dieses Komplexitätsmaß ist ein lexikographisch geordnetes Tripel (n_1, n_2, n_3) natürlicher Zahlen mit den folgenden Bedeutungen:

- n_1 ist Anzahl der noch nicht gelösten Variablen in Γ
- n_2 ist die Größe von Γ: $\sum_{(s \approx t) \in \Gamma} (|s| + |t|)$
- n_3 ist die Anzahl der Gleichungen $s \approx t$ in Γ

Wir zeigen nun, dass bei einer Folge von Anwendungen von Transformationsregeln die Folge der so definierten Komplexitätsmaße gemäß lexikographischer Ordnung ihrer Komponenten streng monoton fällt.

- Die Anwendung der Regeln OCCURS CHECK und KOLLISION führt direkt zu einem unlösbaren Problem.
- Die Anwendung von LÖSCHEN dekrementiert n_2, vergrößert n_1 aber nicht.
- Die Anwendung von DEKOMPOSITION vergrößert die Anzahl nicht gelöster Variablen (n_1) nicht, dekrementiert aber die Größe von Γ.
- Die Anwendung von ORIENTIERUNG läßt die Größe von Γ (n_2) unverändert, vergrößert n_1 nicht, aber dekrementiert n_3.
- Und schließlich dekrementiert die Anwendung der Regel VARIABLENELIMINATION die Anzahl ungelöster Variablen in Γ, mithin also n_1.

□

Zentral für den Korrektheitsbeweis ist die Eigenschaft der Transformationsregeln, dass sie die Menge der Unifikatoren nicht verändern.

LEMMA 2.2.6. *Falls ein Transformationsschritt das Unifikationsproblem* Γ *in das Unifikationsproblem* Γ' *transformiert (also* $\Gamma \rightsquigarrow \Gamma'$*), so gilt* $U(\Gamma) = U(\Gamma')$*.*

Beweis. In den Fällen Löschen, Dekomposition und Orientierung wird $U(\Gamma)$ offensichtlich nicht verändert. Gleiches gilt in den Fällen Occurs Check und Kollision (vergleiche Lemma 2.2.3 und Lemma 2.2.4).

Der einzige übrigbleibende Fall ist also die Regel Variablenelimination. Sei $\sigma \in U(\Gamma)$. Wir betrachten das gelöste Problem $\{x \approx t\}$ sowie die Substitution $\tau = \{\langle t/x \rangle\}$. Mit Lemma 2.1.12 erhalten wir $\sigma = \sigma\tau$ falls gilt $x\sigma = t\sigma$. Damit gilt:

$$\begin{aligned}
\sigma \in U\left(\{x \approx t\} \biguplus S\right) &\Leftrightarrow x\sigma = t\sigma \wedge \sigma \in U(S) \\
&\Leftrightarrow x\sigma = t\sigma \wedge \sigma\tau \in U(S) \\
&\Leftrightarrow x\sigma = t\sigma \wedge \sigma \in U(S\tau) \\
&\Leftrightarrow \sigma \in U\left(\{x \approx t\} \biguplus S\tau\right)
\end{aligned}$$

□

Aus diesem Lemma und Lemma 2.1.13 folgt unmittelbar

Lemma 2.2.7. *Falls die Anwendung der Transformationsregeln auf einem Unifikationsproblem* Γ *mit einem Unifikator* σ *termininiert, so ist* σ *ein allgemeinster idempotenter Unifikator von* Γ*.*

Es bleibt zu zeigen, dass die Transformationsregeln jedes Unifikationsproblem mit einer Lösung auch lösen, d. h. dass ein Algorithmus, der die Transformationsregeln benutzt, vollständig ist.

Lemma 2.2.8. *Falls ein Unifikationsproblem* Γ *lösbar ist, dann finden die Transformationsregeln die Lösung.*

Beweis. Wegen Lemma 2.2.6 genügt es, von einem Unifikationsproblem auszugehen, das durch erschöpfende Anwendung der Transformationsregeln aus Γ entstanden ist. Wir nennen dieses Unifikationsproblem Γ'. Wir haben zu zeigen, dass Γ' gelöst ist.

Γ' kann keine Gleichungen der Form $f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)$ enthalten, weil ansonsten Dekomposition anwendbar wäre. Es kann keine Gleichung der Form $f(s_1, \ldots s_n) \approx g(t_1, \ldots, t_m)$ geben, da ansonsten Kollision angewendet worden wäre. Eine Gleichung der Form $x \approx x$ kann wegen Löschen nicht existieren, eine Gleichung der Form $t \approx x$ mit $t \notin V$ nicht, da ansonsten Orientierung angewendet worden wäre. Eine Gleichung $x \approx t$ mit $x \in Var(t)$ existiert nicht wegen des Occurs Check. Ein zweifaches Auftreten einer Variablen x wird durch die Regel Eliminate ausgeschlossen. Daher ist Γ' in gelöster Form. □

Theorem 2.2.9. *Falls ein Unifikationsproblem* Γ *eine Lösung hat, dann hat es einen idempotenten allgemeinsten Unifikator.*

Beweis. Folgt direkt aus Korrektheit, Vollständigkeit und Terminierung der Transformationsregeln. □

Bemerkenswert an dem Beweis über die Transformationsregeln ist, dass die Regeln in nichtdeterministischer Weise anwendbar sind. Das heißt, jeder Algorithmus, der die Regeln implementiert und in *irgendeiner* Reihenfolge auf ein Unifikationsproblem anwendet, ist vollständig und korrekt. Falls das Problem lösbar ist, wird er die Lösung erreichen, und falls es nicht lösbar ist, so wird er mit $\perp$ enden. Allerdings sind nicht alle Transformationsreihenfolgen notwendigerweise gleich lang und nicht alle Reihenfolgen führen zu gleich großen Zwischentermen.

2.2.1.3. *Komplexität.* Die Komplexität der Transformationsregeln ist exponentiell in Laufzeit- und Speicherverhalten.

Beispiel 2.2.10. Um dies zu sehen, betrachte man das Unifikationsproblem

$$\{x_1 \approx f(x_0, x_0), x_2 \approx f(x_1, x_1), \ldots, x_n \approx f(x_{n-1}, x_{n-1})\}$$

Dieses Problem hat den allgemeinsten Unifikator

$$\{\langle f(x_0, x_0)/x_1\rangle, \langle f(f(x_0, x_0), f(x_0, x_0))/x_2\rangle, \ldots\}$$

der jedem x_i einen kompletten Binärbaum der Höhe i zuweist. Da allgemeinste Unifikatoren identisch sind bis auf Komposition mit einer Variablenpermutation ist jeder allgemeinste Unifikator exponentiell in der Eingabegröße.

Das Problem liegt darin, dass der Unifikator viele Duplikate derselben Subterme enthält. Dieses Problem läßt sich durch den Einsatz einer anderen Termdarstellungsweise mit gemeinsam geteilten Subtermen (*sharing*) beheben.

2.2.2. Verbesserungen des Grundalgorithmus. Ein Problem der Variante des Robinson-Algorithmus aus dem vorangehenden Kapitel ist sein möglicherweise exponentieller Platzverbrauch. Dieser rührt daher, weil die entstehenden Terme exponentiell wachsen können, wenn Variablen mehrfach im Unifikationsproblem vorkommen. In diesem Fall werden während des Lösungsprozesses gleiche Terme nämlich durch die Transformationsregel Variablenelimination kopiert, was wie in Beispiel 2.2.10 zu einer tatsächlich in ihrer Größe exponentiell wachsenden Lösung führen kann.

Dieses Problem kann man dadurch beseitigen, indem man für die Terme eine Graphdarstellung verwendet, die die gemeinsame Nutzung von Subtermen (*sharing*) unterstützt. Insbesondere Variablen dürfen in dieser Graphdarstellung nur einmal instanziiert werden. Mehrfache Nutzung derselben Variablen sollte durch Zeiger realisiert werden.

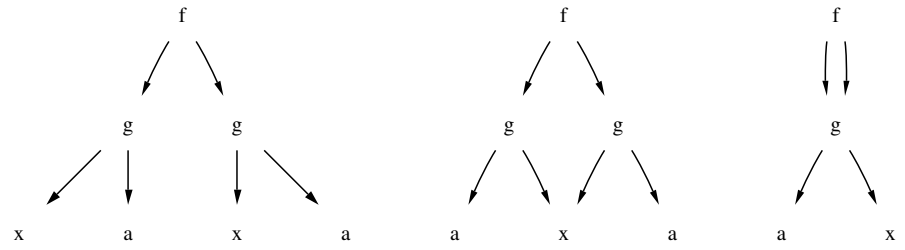
2.2.2.1. *Datenstruktur.*

Definition 2.2.11. Wir verwenden zur Termdarstellung einen gerichteten, azyklischen Graphen, genannt *Term-DAG*. Ein solcher Term-DAG ist ein Graph, dessen Knoten mit Funktionssymbolen und dessen Blätter mit Konstanten oder Variablen markiert sind. Die ausgehenden Kanten eines Knotens sind geordnet und die Anzahl ausgehender Kanten eines Knotens entspricht der Stelligkeit der Funktion dieses Knotens.

Beispiel. Jeder Knoten eines Term-DAGs kann wieder als die Wurzel eines Terms angesehen werden. Wir verwenden daher Knoten eines Term-DAGs und den durch den Knoten dargestellten Term synonym.

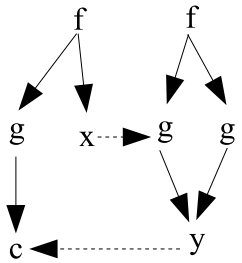
Der entscheidende Unterschied zwischen einem Term und einem Term-DAG ist, dass innerhalb des Term-DAGs Subterme gemeinsam genutzt werden können. Damit kann ein und derselbe Term verschiedene Darstellungen als Term-DAG besitzen.

Wir betrachten zum Beispiel den Term $f\left(g\left(x,a\right),g\left(x,a\right)\right)$, der die folgenden Darstellungen haben kann:



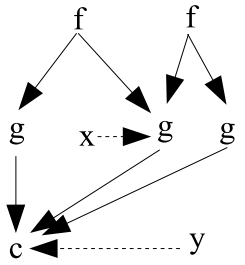
Ein solcher Term-DAG kann in $O\left(n\right)$, wobei n die Länge der Termdarstellung als String ist, erstellt werden [**Baad1999**], indem man beim Parsen des Terms einen Trie benutzt, um Variablennamen zu speichern. Der Trie wird verwendet, um mehrfach vorkommende Variablennamen in der textuellen Darstellung des Terms auf eine einzige Variableninstanz im Graphen zurückzuführen.

Wir nehmen für die folgenden Darstellungen an, dass die Eingabe für den zu erstellenden Unifikationsalgorithmus eine Term-DAG-Darstellung der beiden zu unifizierenden Terme mit gemeinsam genutzten Variablen ist. Ein Beispiel für eine solche Eingabe ist das Unifikationsproblem $f\left(x,g\left(c\right)\right)\approx f\left(g\left(y\right),g\left(y\right)\right)$, das in der folgenden Darstellung zu sehen ist.



Wir nehmen an, dass es eine Funktion `parents` $\left(n\right)$ gibt, die für einen Knoten n in $O\left(1\right)$ eine Liste aller Väter des Knotens liefert. Hierzu kann die Liste der Väter im Knoten selbst gespeichert werden.

Substitution wird im Term-DAG dadurch realisiert, dass bei Instanziierung einer Bindung einer Variablen alle Zielpunkte der eingehenden Kanten der Variablen auf das Bindungsziel umgelenkt werden. Im o. g. Beispiel sähe das dann so aus:



Beispiel 2.2.12. Das in Beispiel 2.2.10 genannte Unifikationsproblem kann auch folgendermaßen formuliert werden:

$$\begin{aligned} s_n(x) &= f(x_1, f(x_2, f(\ldots, x_n)\ldots)) \\ t_n(x) &= f(f(x_0, x_0), f(f(x_1, x_1), f(\ldots, f(x_{n-1}, x_{n-1}))\ldots)) \\ s_n(x) &\approx t_n(x) \end{aligned}$$

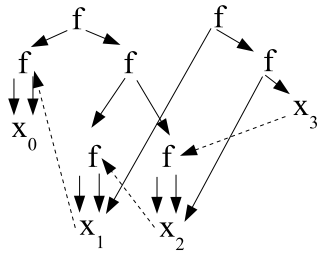
Wählt man nun zum Beispiel $s_3(x) \approx t_3(x)$, so hat man das Problem

$$f(x_1, f(x_2,x_3)) \approx f(f(x_0, x_0), f(f(x_1, x_1), f(x_2, x_2)))$$

Der Unifikator für dieses Problem ist

$$\{\langle f(x_0, x_0)/x_1\rangle, \langle f(x_1, x_1)/x_2\rangle, \langle f(x_2, x_2)/x_3\rangle\}$$

(Substitution in triangulärer Darstellung). Das folgende Schaubild verdeutlicht die Situation:



Wie in Beispiel 2.2.10 ist zu erkennen, dass die Lösung ein vollständiger Binärbaum der Höhe 3 ist. Durch die gemeinsame Nutzung der Variablen innerhalb des Term-DAGs ist eine Lösung jedoch in linearem Platz möglich.

Ein Problem bleibt jedoch auch bei dieser Variante bestehen: der um die Verarbeitung von Term-DAGs angepaßte Algorithmus 2.2.1 ist immer noch exponentiell. Dies zeigt das Beispiel

$$f(s_n(x), f(s_n(y), x_n)) \approx f(t_n(x), f(t_n(y), y_n))$$

Der Algorithmus unifiziert zunächst $s_n(x)$ und $t_n(x)$, danach $s_n(y)$ und $t_n(y)$. Zum Schluß werden x_n und $y_n$unifiziert, was dazu führt, dass dieselben Teilbäume wieder und wieder unifiziert werden, obwohl diese Subterme bereits unifiziert wurden.

Algorithmus 2.2.2 quadratische syntaktische Unifikation

```
global δ: Term-DAG; /* gemeinsam genutzte Variablen */
global σ: substitution; /* initialisiert als Identität */

procedure Unify(s: node, t: node) {
  // auf Gleichheit testen
  if (s = t)
    /* nothing */
  else if (s = f(s_1, ..., s_n) ∧ t = g(t_1, ..., t_m)
           ∧n ≥ 0 ∧ m ≥ 0) {
    if (f = g)
      for i := 1 to n do
        Unify(s_i, t_i);  // Dekomposition
    else
      FAIL;  // kollidierende Symbole
  }
  else if (s ∉ V)
    Unify(t, s);  // Orientieren
  else if (s kommt in t vor)
    FAIL;  // occurs check
  else
    σ := σ{⟨t/s⟩};  // Lösungsmenge erweitern

  // s und t sind nun unifiziert und können
  // zusammengefaßt werden
  δ:=Merge(s, t, δ);
}
```

2.2.2.2. *Quadratischer Algorithmus.* Um nun das exponentielle Verhalten dämpfen zu können, müssen wir dafür sorgen, dass bereits unifizierte Subprobleme nicht erneut besucht und unifiziert werden. Indem bereits unifizierte Terme zusammengeführt werden und man eine Überprüfung auf Gleichheit der Termknoten vor der Unifikation durchführt, erspart man sich diese Mehrfacharbeit.

Das Zusammenführen von Termknoten ist durch einfaches Umsetzen der Zeiger möglich. Seien s, t unifizierte Terme. Sei $\texttt{parents}(s) = \{f_1, \ldots, f_n\}$. Durch folgende Prozedur wird der Term s isoliert und der ehemalige Term s durch die Struktur von t ersetzt. Wir bezeichnen diese Prozedur als `Merge(`s`, `t`, `δ`)`, wobei δ der zu verändernde Term-DAG ist.

- $\forall f_i \in \texttt{parents}(s)$: ersetze die Kante $f_i \rightarrow s$ durch eine Kante $f_i \rightarrow t$
- $\texttt{parents}(t) := \texttt{parents}(t) \cup \texttt{parents}(s)$
- $\texttt{parents}(s) := \emptyset$

Dies führt zu Algorithmus 2.2.2. Seine Komplexität wird im Folgenden betrachtet.

Das Aufstellen des Term-DAGs benötigt Zeit proportional zur Länge des Eingabestrings, also $O(n)$.

Jeder Aufruf von `Unify` isoliert einen Knoten (durch die Verwendung der Funktion Merge). Es kann also insgesamt nicht mehr als n Aufrufe von `Unify` geben, wobei n die Anzahl der Knoten im Term-DAG ist.

Jeder einzelne Aufruf verrichtet konstante Arbeit, außer dem Aufruf des Occurs Check, der den Graphen von t traversiert (wiederum nicht mehr als n Knoten), und dem Aufruf von `Merge`, der höchstens n Kanten verschiebt.

Damit erhält man die Komplexität von $O\left(n^2\right)$.

Die Korrektheit hängt ab von der Korrektheit der Funktion `Merge`. Einen Beweis hierzu liefert [**Baad1999**, Kapitel 2.3.2].

2.2.2.3. *Lineare Algorithmen.* Durch weitere Anpassungen am Algorithmus kann die Komplexität auf ein Maß Nahe der Linearität gedrückt werden. In der Tat zeigt [**Baad1998**, Kapitel 4.8.2] einen Algorithmus, der auf Term-DAGs basiert und die Komplexität $O(mG(n))$ besitzt, wobei m die Anzahl der Kanten und n die Anzahl der Knoten des Term-DAGs bezeichnen. Die Funktion $G(n)$ist eine extrem langsam wachsende Funktion:

$$G\left(\underbrace{2^{2^{\cdots^2}}}_{\text{n-mal}}\right) = n+1$$

[**Baad1999**] stellt noch einen anderen Ansatz vor, in dem das Unifikationsproblem auf die Konstruktion von Äquivalenzklassen zurückgeführt wird. Dabei sind die Klassen die Terme, die unifiziert werden müssen. Der in jedem Durchgang durchgeführte occurs-Check wird durch eine einzelne Prüfung des Graphen auf Azyklität ersetzt. Dieser Algorithmus erreicht eine Komplexität von $O(n\alpha(n))$, wobei α die Inverse der Ackermann-Funktion ist.

Ein Algorithmus, der tatsächlich Linearzeit-Komplexität erreicht, wurde von Paterson und Wegman [**Pat1978**] vorgestellt.

2.3. Der einfach getypte λ-Kalkül

Wir kommen nun zu einem Bereich, den man als Verallgemeinerung der syntaktischen Unifikation auffassen kann. Die Syntaxunifikation läßt als Instanzen von Variablen lediglich einfache Terme zu. Hebt man diese Einschränkung auf und läßt zusätzlich Funktionen als Werte für Variablen zu, so kommt man zur Unifikation höherer Ordnung. So kann man das Problem $f(a,x) \approx y(a,b)$ nur unifizieren, indem man für die Variable y Funktionen zuläßt, in diesem Fall die Funktion f.

Für die weiteren Erläuterungen wird der einfach getypte λ-Kalkül als Grundlage verwendet. In diesem Abschnitt werden seine grundlegenden Eigenschaften eingeführt.

Die Darstellungen in den folgenden Abschnitten basieren im Wesentlichen auf [**Dow2001**] und [**Schm2003-2**].

2.3.1. Typen.

DEFINITION 2.3.1. Es gebe eine endliche Menge von *Grundtypen* (atomare Typen). Dann sind Typen induktiv definiert durch folgende Regeln:

(1) Ein Grundtyp ist ein Typ.
(2) Falls S und T Typen sind, so ist auch $S \to T$ ein Typ.

BEISPIEL. Bei einer Grundmenge $A = \{Int, Float\}$ ist also die folgende Menge eine Teilmenge aller aus diesen beiden Grundtypen zu konstruierenden Typen:

$$\{Int,\ Float,\ Int \to Float, Float \to Float,$$
$$Float \to Int, Int \to (Int \to Int), (Int \to Int) \to Int\}$$

NOTATION 2.3.2. Typen der Form $T_1 \to (T_2 \to \ldots (T_{n-1} \to T_n))$ werden auch als $T_1 \to T_2 \to \ldots \to T_{n-1} \to T_n$ geschrieben.

Für die Klassifikation von Unifikations- und Matchingproblemen wird oftmals der Begriff der Ordnung eines Typs bzw. eines Terms verwendet.

DEFINITION 2.3.3. Falls T ein Typ ist, dann ist die *Ordnung* von T wie folgt definiert:

- $\mathtt{ord}(T) = 1$, falls T ein Grundtyp ist.
- $\mathtt{ord}(T_1 \to T_2) = \mathtt{max}\{1 + \mathtt{ord}(T_1), \mathtt{ord}(T_2)\}$

Damit hat der Typ $T_1 = (Int \to Int) \to Int$ die Ordnung $\mathtt{ord}(T_1) = 3$. Der Typ $T_2 = Int \to (Int \to Int)$ hat dagegen die Ordnung $\mathtt{ord}(T_2) = 2$ und wird kurz als $T_2 = Int \to Int \to Int$ geschrieben.

2.3.2. Terme.

Ist im Folgenden ein Term mit einem Superskript τ versehen, so bezeichnet dieser den Typ des Terms.

DEFINITION 2.3.4. Einfach getypte λ-Terme vom Typ τ sind induktiv wie folgt definiert:

- Konstanten vom Typ τ sind Terme, z. B. f^τ.
- Variablen sind Terme, z. B. x^τ.
- Falls $s^{\tau' \to \tau}$ und und $t^{\tau'}$ Terme sind, so ist auch $(s\,t)^\tau$ ein Term. Terme dieser Form werden Applikationen genannt.
- Falls x^{τ_1} eine Variable ist und t^{τ_2} ein Term, so ist auch $(\lambda x.t)^\tau$ ein Term, falls zusätzlich gilt: $\tau = \tau_1 \to \tau_2$. Terme dieser Form werden Abstraktionen genannt.

Die Menge der einfach getypten Terme vom Typ τ läßt sich damit auch kürzer definieren als

$$Term^\tau ::= f^\tau | x^\tau | \left(Term^{\tau' \to \tau}\, Term^{\tau'}\right) | \lambda x^{\tau_1}.Term^{\tau_2}$$

NOTATION 2.3.5. Wir verwenden die folgenden Schreibweisen:

- Mit $\mathtt{type}(t)$ wird der Typ eines Terms bezeichnet.

- Die Ordnung $\texttt{ord}\,(t^\tau)$ eines Terms t ist die Ordnung $\texttt{ord}\,(\tau)$ seines Typs τ.
- Im Folgenden werden die Begriffe Term und λ-Term synonym verwendet.
- Wir verwenden für $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.t$ die Schreibweise $\lambda x_1, \ldots, x_n.t$.
- Wir benutzen oftmals den Bezeichner ι als Symbol für einen beliebigen, aber eindeutigen Grundtyp.

Wir nehmen an, dass zu jedem Grundtyp g ein konstanter Term t_g existiert. Damit gibt es zu jedem zusammengesetzten Typ $T_1 \to \ldots \to T_n \to g$ einen geschlossenen Term $\lambda x_1, \ldots, x_n.t_g$, wobei $\texttt{type}\,(x_i) = T_i$, $\forall 1 \leq i \leq n$.

Definition. Die Größe eines Terms t, bezeichnet mit $|t|$, ist wie folgt definiert:

- $|c| := 1$ (c Konstante)
- $|x| := 1$ (x Variable)
- $|(s\,u)| := |s| + |u|$
- $|\lambda x.s| := |s|$

Definition 2.3.6. Ein *Kontext* ist ein Term, in dem ein spezieller Operator $\Box$ (Loch) genau ein einziges mal vorkommt. Wir verwenden für einen Kontext die Notation $C\,[\Box] = C\,[]$.

Ein Kontext $C\,[]$ kann auf einen Term t angewandt werden. Wir schreiben dies als $C\,[t]$. In diesem Falle wird das Loch in C durch den Term t syntaktisch ersetzt.

2.3.3. Substitutionen, α-Äquivalenz. Im Folgenden bezeichnen c ein konstantes Symbol, x eine Variable sowie s, t und u Terme.

Definition 2.3.7. Sei t ein Term. Dann ist $\texttt{Var}\,(t)$ die *Menge aller Variablen*, die in t vorkommen. Diese Menge wird induktiv definiert:

- $\texttt{Var}\,(c) = \emptyset$
- $\texttt{Var}\,(x) = \{x\}$
- $\texttt{Var}\,((s\,u)) = \texttt{Var}\,(s) \cup \texttt{Var}\,(u)$
- $\texttt{Var}\,(\lambda x.s) = \{x\} \cup \texttt{Var}\,(s)$

Mit $\texttt{FV}\,(t)$ bezeichnen wir die *Menge aller freien Variablen*, d. h. der Variablen, die in t vorkommen, aber nicht gebunden sind. Diese Menge ist ebenfalls induktiv definiert:

- $\texttt{FV}\,(c) = \emptyset$
- $\texttt{FV}\,(x) = \{x\}$
- $\texttt{FV}\,((s\,u)) = \texttt{FV}\,(s) \cup \texttt{FV}\,(u)$
- $\texttt{FV}\,(\lambda x.s) = \texttt{FV}\,(s) \setminus \{x\}$

Terme t mit $\texttt{FV}\,(t) = \emptyset$ werden auch als *geschlossen* bezeichnet.

Die Menge der freien Variablen eines Terms wird noch bei der Definition der Unifikationsprobleme von Bedeutung sein .

Definition 2.3.8. Eine *Substitution* ist eine endliche Menge von Paaren der Form

$$\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$$

wobei jeder Term t_i vom gleichen Typ wie die Variable x_i ist:

$$\forall 1 \leq i \leq n : type\,(x_i) = \tau_i = type\,(t_i)$$

Notation 2.3.9. Eine zu $\{\langle x_1,t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$ äquivalente Schreibweise ist die Darstellung $\{\langle t_1/x_1\rangle, \ldots, \langle t_n/x_n\rangle\}$, die eher betont, dass zum Beispiel t_1 für x_1 einzusetzen ist.

Mit Hilfe von Substitutionen können Terme verändert werden, indem Variablen durch λ-Terme ersetzt werden. Dabei muss man allerdings beachten, dass man in λ-Abstraktionen keine freien Variablen einfängt.

Beispiel 2.3.10. Wenn man die Substitution $\lambda x.x\,\{\langle y/x\rangle\}$ ausführt, ist nicht $\lambda x.y$ das Ergebnis, sondern $\lambda x.x$. Im ersten Falle hätte man x aus der Bindung herausgelöst.

Bei der Substitution $\lambda y.x\,\{\langle y/x\rangle\}$ ist das korrekte Substitionsergebnis $\lambda z.y$ und nicht $\lambda y.y$. Im letzteren Fall wäre die freie Variable x eingefangen worden.

Wegen dieser Problematik werden bei der Substitution in λ-Abstraktionen in den folgenden Regeln die gebundenen Variablen vor der eigentlichen Substition umbenannt.

Definition 2.3.11. (Substitution in λ-Termen)

Sei t ein Term. Sei $\sigma = \{\langle t_1/x_1\rangle, \ldots, \langle t_n/x_n\rangle\}$ eine Substitution. Dann ist der durch Substitution entstandene Term $t' = \sigma\,(t) = t\sigma$ wie folgt definiert:

- $c\sigma = c$
- $x\sigma = \begin{cases} t_i, \text{ falls } x = x_i \\ x, \text{ sonst} \end{cases}$
- $\sigma\,(s\,u) = (s\sigma\; u\sigma)$
- $\sigma\,(\lambda x.s) = \lambda y.s\sigma\,\{\langle y/x\rangle\}$, wobei y eine neue Variable mit gleichem Typ wie x ist. y kommt weder in t vor noch in $t_1, \ldots, t_n$ und ist verschieden von den $x_1, \ldots, x_n$.

Definition. Die *Größe einer Substitution* σ, bezeichnet mit $|\sigma|$, ist wie folgt definiert:

$$|\sigma| = |\{\langle t_1/x_1\rangle, \ldots, \langle t_n/x_n\rangle\}| := \sum_{i=1}^{n} |t_i|$$

Terme können in α-Äquivalenzklassen aufgeteilt werden; diese sind definiert aufgrund von Termen modulo der Umbenennung gebundener Variablen. Damit können Terme auf syntaktischer Ebene als gleich betrachtet werden, wenn es bis auf Umbenennung gebundener Variablen keine Unterschiede gibt.

Definition 2.3.12. (α-Äquivalenz).

Die *α-Äquivalenz* (Äquivalenz modulo Umbenennung gebundener Variablen) wird induktiv definiert:

- $c =_\alpha c$
- $x =_\alpha x$
- $(s\,u) =_\alpha (s'\,u')$ genau dann wenn $s =_\alpha s' \wedge u =_\alpha u'$.
- $\lambda x.s =_\alpha \lambda y.u$ genau dann wenn $\langle x, z\rangle\, s =_\alpha \langle y, z\rangle\, u$ gilt für ein z mit $x \neq z \neq y$, welches weder in s noch in u vorkommt.

Beispiel. Die Terme $\lambda x.x$ und $\lambda y.y$ sind α-äquivalent.

2.3.4. Reduktion. Reduktion überführt einen Term in einen semantisch äquivalenten Term.

Definition 2.3.13. (Reduktion)

Eine Reduktion $\rightarrow$ ist eine binäre Relation auf einer Menge A: $\rightarrow \subseteq A \times A$. Anstelle von $(a, b) \in A$ schreibt man $a \rightarrow b$ und sagt „a reduziert zu b".

Wenn wir $a \xrightarrow{*} b$ schreiben, so verwenden wir mit $\xrightarrow{*}$ die reflexive und transitive Hülle von $\rightarrow$ und meinen, dass es eine endliche Folge von Reduktionen gibt, die a zu b überführt. Wir geben die notwendigen Definitionen wie in [**Baad1998**] an.

Definition 2.3.14. (Reduktionsnotation)

reflexive Identität	$\xrightarrow{0} := \{(x, x)\,|x \in A\}$
i-fache Komposition	$\xrightarrow{i+1} := \xrightarrow{i} \circ \rightarrow$
transitive Hülle	$\xrightarrow{+} := \bigcup_{i>0} \xrightarrow{i}$
reflexive transitive Hülle	$\xrightarrow{*} := \xrightarrow{+} \cup \xrightarrow{0}$
reflexive Hülle	$\xrightarrow{=} := \rightarrow \cup \xrightarrow{0}$
Inverse	$\leftarrow := \xrightarrow{-1} := \{(x, y)\,|x \rightarrow y\}$
symmetrische Hülle	$\leftrightarrow := \rightarrow \cup \leftarrow$
transitive symmetrische Hülle	$\xleftrightarrow{+} := (\leftrightarrow)^+$
reflexive transitive symmetrische Hülle	$\xleftrightarrow{*} := (\leftrightarrow)^*$

Man kann Reduktion als eine Form der Berechnung betrachten, wobei in jedem Berechnungsschritt eine Reduktion durchgeführt wird. Gibt es keinen möglichen Reduktionsschritt mehr, so sagt man, der Term habe *Normalform* erreicht:

Notation 2.3.15. Einige Grundbegriffe im Rahmen von Reduktionssystemen:

- t heißt *reduzierbar* gdw. es ein u gibt, so dass $t \rightarrow u$.
- t ist in *Normalform* gdw. es kein u gibt mit $t \rightarrow u$.
- u ist eine *Normalform* von t gdw. $t \xrightarrow{*} u$ und u ist eine Normalform. Falls t eine eindeutig bestimmte Normalform besitzt, so wird diese mit $t\downarrow$ bezeichnet.

- s und t heißen *vereinigbar* gdw. es ein u gibt mit $s \xrightarrow{*} u \xleftarrow{*} t$. In diesem Falle schreiben wir $s \downarrow t$.

DEFINITION 2.3.16. Eine Reduktion $\rightarrow$ kann eine oder mehrere der folgenden Eigenschaften besitzen:

Church-Rosser: es gilt $x \overset{*}{\leftrightarrow} y \Rightarrow x \downarrow y$
konfluent: es gilt $y_1 \xleftarrow{*} x \xrightarrow{*} y_2 \Rightarrow y_1 \downarrow y_2$
normalisierend: jeder Term t besitzt eine Normalform, d. h. es gibt keinen Term t' mit $t \rightarrow t'$.
terminieren: es gibt keine unendlich absteigende Kette $t_1 \rightarrow t_2 \rightarrow \ldots$
konvergent: $\rightarrow$ ist konfluent und terminierend.

Es zeigt sich, dass die Church-Rosser-Eigenschaft und Konfluenz miteinander einhergehen [**Baad1998**].

Im λ-Kalkül [**Bar1984**] werden die Reduktionsregeln β, η und $\overline{\eta}$ definiert.

DEFINITION 2.3.17. ($\beta\eta$-Reduktion)

Sei C ein beliebiger Kontext.

- (β) $C[(\lambda x.t)\ s] \rightarrow C[t\{\langle s/x\rangle\}]$
- (η) $C[\lambda y.(t\,y)] \rightarrow C[t]$, falls $y \notin FV(t)$
- $(\overline{\eta})$ $C[t] \rightarrow C[\lambda y.(t\,y)]$, falls t keine Abstraktion ist, $type(t) = \tau_1 \rightarrow \tau_2$ kein Grundtyp und t ist in $C[t]$ eine maximale Abstraktion. Die Variable y muss eine frische Variable sein mit passendem Typ τ_1.

β-Reduktion reduziert Applikationen und führt diese damit aus. η-Reduktion entfernt Bindungen aus Abstraktionen und führt damit neue freie Variablen ein. Die $\overline{\eta}$-Reduktion ist die Umkehrung der η-Reduktion und wird daher auch als η-Expansion bezeichnet. Sie führt neue gebundene Variablen ein und macht damit implizite Argumente eines Terms sichtbar.

NOTATION 2.3.18. Kann ein Term nicht weiter durch $\overline{\eta}$-Reduktion reduziert werden, so ist dieser Term in *η-Langform*. Kann ein Term weder durch β- noch durch $\overline{\eta}$-Reduktion reduziert werden, so ist dieser Term in *$\beta\overline{\eta}$-Normalform*. Diese wird auch als η-Lang-β-Normalform bezeichnet.

THEOREM 2.3.19. *Die $\beta\eta$-Reduktion ist streng normalisierend und konfluent auf getypten Termen (und weist somit die Church-Rosser-Eigenschaft auf). Damit hat jeder Term t eine eindeutige $\beta\eta$-Normalform $t\downarrow$ (modulo α-Äquivalenz)* [**Bar1984**].

LEMMA 2.3.20. *Sei t ein λ-Term in Normalform vom Typ $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \iota$. Dann hat t die Form*

$$t = \lambda x_1, \ldots, x_n.(h\,u_1\,u_k)$$

mit $k \leq n$ und h als Kopfsymbol. h ist entweder eine Variable oder eine Konstante.

LEMMA 2.3.21. *Sei* $t = \lambda x_1, \ldots, x_n.\,(h\,u_1\,u_k)$ *ein* λ*-Term in Normalform vom Typ* $\tau_1 \to \tau_2 \to \ldots \to \tau_m \to \iota$*. Dabei sei* $n \leq m$*, d. h. die Stelligkeit von* t *ist höchstens erreicht. Dann hat* t *die* η*-Lang-*β*-Normalform*

$$t_{\beta\overline{\eta}} = \lambda x_1, \ldots, x_n, x_{n+1}, \ldots, x_m.\,\left(h\,\overline{u_1}\,\ldots\,\overline{u_k}\,\overline{x_{n+1}}\,\ldots\,\overline{x_m}\right)$$

Dabei seien die $\overline{u_i}$ *die* η*-Lang-*β*-Normalformen der* u_i *und die* $\overline{x_i}$ *die* η*-Lang-*β*-Normalformen der* x_i*.*

2.4. Unifikation und Matching höherer Ordnung

Die Definitionen für Unifikations- und Matchingprobleme höherer Ordnung sind bis auf die Definition der Gleichheit identisch mit denen aus dem Bereich der syntaktischen Unifikation.

DEFINITION 2.4.1. (Unifikationsproblem)

Eine Gleichung ist ein Paar von Termen s, t und wird $s \approx u$ geschrieben. Ein *Unifikationsproblem* ist eine endliche Menge $\{s_1 \approx u_1, \ldots, s_n \approx u_n\}$ von Gleichungen. Eine Lösung, genannt *Unifikator*, eines solchen Problems ist eine Substitution σ dergestalt, dass für jede Gleichung $s \approx u$ des Problems $\sigma(s) =_{\beta\eta} \sigma(u)$ gilt. Die substituierten Terme einer Gleichung müssen also dieselbe Normalform haben.

Die Matchingprobleme sind eine Untermenge der Unifikationsprobleme: die rechten Seiten der Gleichungen der Gleichungsmenge werden in Bezug auf die freien Variablen eingeschränkt.

DEFINITION. Die *Ordnung* einer Instanz eines Unifkations- oder Matchingproblems ist die maximale Ordnung seiner freien Variablen.

LEMMA 2.4.2. *Eine Instanz eines Matchingproblems höherer Ordnung* Γ *ist genau dann lösbar, wenn eine geschlossene Lösung existiert.*

BEWEIS. Die Rückrichtung ist trivial, da jede geschlossene Lösung von Γ zeigt, dass Γ lösbar ist.

Sei umgekehrt $\sigma = \{\langle t_1/x_1\rangle, \ldots, \langle t_n/x_n\rangle\}$ eine beliebige Lösung von Γ, d. h. eine Lösung, die nicht notwendigerweise geschlossen ist. Seien die $x_i\,(1 \leq i \leq n)$ die freien Variablen von Γ.

Wir betrachten die Menge der freien Variablen in den substituierten Termen $t_1, \ldots, t_n$: $\theta = FV(\{t_1, \ldots, t_n\})$. Wir wählen nun Konstanten $c_1, \ldots, c_{|\theta|}$ mit entsprechenden Typen, die nicht auf den rechten Seiten von Γ auftauchen. Solche Konstanten existieren, denn wir haben angenommen, dass es für jeden Grundtyp g eine Konstante c_g gibt. $\texttt{const}_\theta(v)$ liefere aus dieser Menge von Konstanten diejenige, die der freien Variablen v zugeordnet ist. Wir definieren nun die Substitution $\phi = \{\langle \texttt{const}_\theta(v)/v\rangle \,|\, v \in \theta\}$, d. h. die freien Variablen in der nicht geschlossenen Lösungssubstitution σ werden ersetzt durch typkonforme frische Konstanten. Die Substitution $\phi\sigma$ ist ebenfalls eine Lösung von Γ, jedoch eine geschlossene. □

2.4.1. Entscheidbarkeit. Unifikationsprobleme erster Ordnung sind entscheidbar, Unifikationsprobleme höherer Ordnung dagegen nicht. D. h. es existiert kein Algorithmus, der für ein beliebiges Unifikationsproblem berechnet, ob eine Lösung existiert oder nicht.

Es gibt jedoch Untermengen der Unifikationsprobleme, die durch Einschränkungen hinsichtlich der zugelassenen Unifikationsprobleme oder der zulässigen Lösungsmenge gebildet werden und dadurch entscheidbar werden.

Zum Beweis der Unentscheidbarkeit verwenden wir den Beweis aus [**Dow2001**]. Wir reduzieren hierzu ein anderes unentscheidbares Problem auf die Unifikationsprobleme höherer Ordnung: Hilberts zehntes Problem. Falls dann die Unifikationsprobleme entscheidbar sind, so ist es Hilberts zehntes Problem durch Instanzreduktion auch.

THEOREM 2.4.3. *(Matiyacevich-Robinson-Davis). Hilberts zehntes Problem ist unentscheidbar. Das heißt es gibt keinen Algorithmus, der für zwei beliebige Polynome $P(X_1, \ldots, X_n)$ und $Q(X_1, \ldots, X_n)$ mit natürlichen Koeffizienten berechnet, ob es eine Lösung mit natürlichen Zahlen $k_1, \ldots, k_n$ gibt, so dass gilt*

$$P(k_1, \ldots, k_n) = Q(k_1, \ldots, k_n)$$

Für die Reduktion von Hilberts zehntem Problem benötigen wir eine Zahlendarstellung natürlicher Zahlen im einfach getypten λ-Kalkül.

DEFINITION 2.4.4. (Church-Zahlen)

Eine natürliche Zahl n wird wie folgt als Church-Zahl $\overline{n}$ dargestellt:

$$\overline{n} = \lambda x.\lambda f.\,(f\ (f \ldots (f\, x)))$$

Dabei kommt das Funktionssymbol f genau n-mal in $\overline{n}$ vor. Es gilt $\mathtt{type}(\overline{n}) = \iota \to (\iota \to \iota) \to \iota$.

Mit Church-Zahlen kann wie folgt gerechnet werden.

PROPOSITION 2.4.5. *Die Addition und Multiplikation unter Church-Zahlen $\overline{n}$ und $\overline{m}$ ist definiert als*

$$\mathtt{add} = \lambda n, m.\lambda x.\lambda f.\,(n\ (m\, x\, f)\ f)$$

$$\mathtt{mult} = \lambda n, m.\lambda x.\lambda f.\,(n\, x\ (\lambda z.\,(m\, z\, f)))$$

Die Normalform des Terms $\mathtt{add}\,\overline{n}\,\overline{m}$ ist $\overline{n+m}$, die Normalform des Terms $\mathtt{mult}\,\overline{n}\,\overline{m}$ ist $\overline{n \cdot m}$.

Damit kann jedes Polynom mit natürlichen Koeffizienten in natürlichen Zahlen in der Church-Zahl-Darstellung berechnet werden: Für jedes Polynom P existiert ein λ-Term p, so dass die Normalform von $p\,\overline{m_1} \ldots \overline{m_n}$ genau $\overline{P(m_1, \ldots, m_n)}$ ist.

Es gilt nun, das polynomielle Problem $P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)$ mit der Lösung $m_1, \ldots, m_n$ auf ein Unifikationsproblem zu reduzieren. Wir benutzen hierzu das Unifikationsproblem

$$\Gamma_{Hilbert} = \{p\, X_1 \ \ldots \ X_n \approx q\, X_1 \ \ldots \ X_n\} \tag{2.4.1}$$

Dabei sind p und q die Polynome P bzw. Q in Church-Darstellung wie in 2.4.5.

Offensichtlich ist in diesem Falle die Substitution $\{\langle\overline{m_1}/X_1\rangle, \ldots \langle\overline{m_n}/X_n\rangle\}$ eine Lösung von $\Gamma_{Hilbert}$.

Der umgekehrte Fall ist nicht ganz so einfach, da es Lösungen von $\Gamma_{Hilbert}$ gibt, die keine Church-Zahlen sind und damit auch keine Lösung von Hilberts zehntem Problem. Man kann die Lösungsmenge des Unifikationsproblems jedoch dadurch einschränken, indem man zusätzliche Gleichungen zu $\Gamma_{Hilbert}$ hinzufügt, die $X_1, \ldots, X_n$ auf Terme beschränken, die Church-Zahlen sind.

Proposition 2.4.6. *Ein Term t vom Typ $\iota \to (\iota \to \iota) \to \iota$ ist genau dann eine Church-Zahl, wenn $\sigma = \{\langle t/X\rangle\}$ eine Lösung der Gleichung des Unifikationsproblems Γ_{Church} ist:*

$$\Gamma_{Church} = \{\lambda z.\, (X\ z\ (\lambda y\, y)) \approx \lambda z.z\}$$

Beweis. Durch Induktion nach der Struktur des Terms t. □

Theorem 2.4.7. *Es gibt keinen Algorithmus, der als Eingabe ein Unifikationsproblem hat und der berechnet, ob dieses eine Lösung besitzt oder nicht.*

Beweis. Wir reduzieren Hilberts zehntes Problem auf Unifikation und nehmen als Probleminstanz $P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)$ an. Hierzu ergänzen wir das Unifikationsproblem $\Gamma_{Hilbert}$ um weitere Gleichungen, die garantieren, dass gefundene Substitutionen für die freien Variablen $X_1, \ldots, X_n$ Church-Zahlen darstellen:

$$\begin{aligned}\Gamma_{HilbertChurch} = \{ \quad & p\, X_1 \ \ldots \ X_n \approx q\, X_1 \ \ldots \ X_n, \\ & \lambda z.\, (X_1\ z\ (\lambda y\, y)) \approx \lambda z.z \\ & \vdots \\ & \lambda z.\, (X_n\ z\ (\lambda y\, y)) \approx \lambda z.z\ \}\end{aligned}$$

□

Die Terme p und q stellen wieder die entsprechenden Polynome der Hilbertinstanz dar.

Beweis. Falls die Probleminstanz eine Lösung $m_1, \ldots, m_n$ hat, so ist

$$\{\langle\overline{m_1}/X_1\rangle, \ldots, \langle\overline{m_n}/X_n\rangle\}$$

eine Lösung von $\Gamma_{HilbertChurch}$. Ist nun umgekehrt $\{\langle t_1/X_1\rangle, \ldots, \langle t_n/X_n\rangle\}$ eine Lösung von $\Gamma_{HilbertChurch}$, so garantiert Proposition 2.4.6 zunächst, dass es sich bei den Termen $t_1, \ldots, t_n$ um Churchzahlen handelt, wobei gelte

$t_i = \overline{m_i}$ $(\forall i : 1 \leq i \leq n)$. Die m_i sind natürliche Zahlen und eine Lösung der Hilbertinstanz. □

2.4.2. Unifikation zweiter Ordnung.

DEFINITION 2.4.8. Ein *Unifikationsproblem n-ter Ordnung* ist ein Unifikationsproblem, in dem man als freie Variablen lediglich Variablen n-ter Ordnung zuläßt.

Goldfarb zeigte 1981, dass bereits Unifikation zweiter Ordnung unentscheidbar ist, indem Hilberts zehntes Problem auf Unifikation reduziert wird.

Im obigen Beweis haben die Church-Zahlen $X_1, \ldots, X_n$ den Typ

$$\iota \to (\iota \to \iota) \to \iota$$

und sind von Ordnung 3.

Goldfarb verwendet eine abgewandelte Version der Church-Zahlen, indem eine Zahl m nun durch $\lambda x.\,(f\ (f\ \ldots\ (f\,x)))$ dargestellt wird [**Gold1981**]. Dabei ist f ein Konstante vom Typ $\iota \to \iota$. Diese Zahlen besitzen nunmehr den Typ $\iota \to \iota$ und sind von zweiter Ordnung.

PROPOSITION 2.4.9. *Ein Term t vom Typ $\iota \to \iota$ ist genau dann eine Goldfarb-Zahl, wenn $\sigma = \{\langle t/X\rangle\}$ eine Lösung der Gleichung des Unifikationsproblems $\Gamma_{Goldfarb}$ ist:*

$$\Gamma_{Goldfarb} = \{g\,a\ (X\,a) \approx X\ (g\,a\,a)\}$$

BEWEIS. Durch Induktion nach der Struktur des Terms t. □

PROPOSITION 2.4.10. *Die Addition unter Goldfarb-Zahlen $\overline{n}$ und $\overline{m}$ ist definiert als*

$$\mathtt{add} = \lambda n, m.\lambda x.\,(n\ (m\,x))$$

Die Multiplikation wird als ein Unifikationsproblem dargestellt:

$$\begin{aligned}\Gamma_{Goldfarb-Mult} \quad = \quad & \{Y\,a\,b\ (g\ (g\ (X_3\,a)\ (X_2\,b))\ a) \approx \\ & g\ (g\,a\,b)\ (Y\ (X_1\,a)\ (g\,a\,b)\ a)\} \\ & \cup \{Y\,b\,a\ (g\ (g\ (X_3\,b)\ (X_2\,a))\ a) \approx \\ & g\ (g\,b\,a)\ (Y\ (X_1\,b)\ (g\,a\,a)\ a)\end{aligned}$$

$\Gamma_{Goldfarb-Mult}$ *hat eine Lösung*

$$\sigma_{Goldfarb-Mult} = \{\langle\overline{a_1}/X_1\rangle, \langle\overline{a_2}/X_2\rangle, \langle\overline{a_3}/X_3\rangle, \langle u/Y\rangle\}$$

gdw. $a_1 \cdot a_2 = a_3$.

Für einen Beweis siehe [**Gold1981**].

THEOREM 2.4.11. *Unifikation zweiter Ordnung ist unentscheidbar.*

Beweis. Wir reduzieren Hilberts zehntes Problem auf Unifikation und nehmen als Probleminstanz $P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)$ an. Diese Probleminstanz kann dargestellt werden durch ein System von Gleichungen der folgenden Form:

$$\begin{aligned} X_i + X_j &= X_k \\ X_i \cdot X_j &= X_k \\ X_i &= p \end{aligned}$$

Beispielsweise läßt sich das Problem $2(x_1)^2 + x_0 = 3x_0$ darstellen als

$$\begin{aligned} x_1 \cdot x_1 &= x_q \\ x_q + x_q &= x_{dq} \\ x_{dq} + x_0 &= p \\ x_0 + x_0 &= x_{0d} \\ x_0 + x_{0d} &= x_{03} \\ x_{03} &= q \\ p &= q \end{aligned}$$

Damit läßt sich nun ein Unifikationsproblem erstellen, das Hilberts zehntes Problem lösen würde.

Man füge für jede Variable X_i eine Unifikationsgleichung hinzu, die garantiert, dass X_i eine Goldfarbzahl ist (siehe Proposition 2.4.9).

Für jede Additionsgleichung des Systems $(X_i + X_j = X_k)$ füge eine Unifikationsgleichung der Form $(\lambda n, m.\lambda x.\,(n\ (m\,x)))\ X_i\,X_j \approx X_k$ hinzu.

Für jede Multiplikationsgleichung des Systems $(X_i \cdot X_j = X_k)$ füge die beiden entsprechenden Unifikationsgleichungen aus Proposition 2.4.10 hinzu.

Und für jede Gleichung der Form $X_i = p$ füge die Gleichung $X_i \approx \overline{p}$ hinzu. □

2.5. Algorithmen

Obwohl Unifikation höherer Ordnung im Allgemeinen unentscheidbar ist, lohnt es sich dennoch, entsprechende Algorithmen zu betrachten. Zum einen ist die Unifikation semi-entscheidbar, d. h. man kann einen Algorithmus schreiben, der eine Eingabe daraufhin untersucht, ob sie eine Lösung für ein gegebenes Unifikationsproblem ist. Der Algorithmus terminiert mit ja, falls es so ist, kann aber auf der anderen Seite endlos laufen, falls der Untersuchungskandidat keine Lösung ist.

In einem ersten Schritt kann man also einen Algorithmus entwerfen, der zunächst alle möglichen korrekt getypten Einsetzungen (gemäß einfach getyptem λ-Kalkül) für die freien Variablen generiert und diese danach gegen das zu lösende Unifikationsproblem testet. Ein solcher Algorithmus hat einen brute force-Ansatz und verwendet das Unifikationsproblem rein passiv zum Test der Lösungskandidaten. In einem weiteren Verbesserungsschritt kann

man dann versuchen, bei der Suche nach geeigneten Lösungskandidaten Informationen aus dem Unifikationsproblem selbst zu verwenden, um den Suchraum einzuschränken.

2.5.1. Generierung aller Terme. Wir erinnern an Lemma 2.4.2, das besagte, dass es stets einen geschlossenen Unifikator für ein Unifikationsproblem gibt, falls das Problem lösbar ist. In diesem Sinne reicht es aus, nur solche Substitutionen $\{\langle t_1/X_1\rangle, \ldots, \langle t_n/X_1\rangle\}$ zu testen, bei denen die Terme $t_1, \ldots, t_n$ geschlossen sind. Weiterhin ist diese Substitution auch dann eine Lösung, wenn man die t_i durch entsprechenden η-Lang-β-Normalformen ersetzt.

Wir wissen, dass ein Term t vom Typ $\tau_1 \to \ldots \to \tau_n \to \iota$ in η-Lang-β-Normalform die Form $\lambda x_1, \ldots x_n.\,(h\, u_1 \ldots u_k)$ besitzt, wobei die x_i den entsprechenden Typ τ_i besitzen, h entweder eine der Variablen x_i ist oder eine Konstante und die Anzahl und Typen der u_i von h abhängen.

Wollen wir nun alle η-Lang-β-Normalformen für einen solchen Term t aufzählen, so können wir beginnen, indem wir alle möglichen Kopfsymbole h wählen und entsprechende Terme mit Variablen $H_1, \ldots, H_{ar(h)}$ generieren:

$$\lambda x_1, \ldots, x_n.\,\big(h\ (H_1\, x_1 \ldots x_n)\ \ldots\ \big(H_{ar(h)}\, x_1 \ldots x_n\big)\big)$$

Anschließend werden entsprechende Terme für die Variablen H_i generiert. Diese besitzen eine funktionale Abhängigkeit, da die Variablen $x_1, \ldots, x_n$ in den zu generierenden Termen für die Variablen H_i vorkommen könnten. Würde man die funktionale Abhängigkeit nicht explizit darstellen, so würde man Variablen bei der Substitution der H_i einfangen.

Definition 2.5.1. Wir generieren alle η-Lang-β-Normalformen ausgehend von einem Term t mit der folgenden Regel. Sei X dabei eine freie Variable in t vom Typ $\tau_1 \to \ldots \to \tau_n \to \iota$, h eine der Variablen x_i oder eine Konstante mit Zieltyp ι und die H_i sind frische Variablen mit zu h passendem Typ.

$$\frac{t}{t\left\{\left\langle \lambda x_1, \ldots x_n.\,\big(h\ (H_1\, x_1 \ldots x_n)\ \ldots\ \big(H_{ar(h)}\, x_1 \ldots x_n\big)\big)/X\right\rangle\right\}}$$

Mit Induktion nach der Größe von t kann man zeigen, dass ausgehend von einer Variablen X mit $\mathtt{type}\,(X) = \tau$ alle geschlossenen η-Lang-β-Normalformen vom Typ τ mit der Regel generiert werden.

Bei der Generierung gibt es neben dem don't care-Nichtdeterminismus bei der Wahl der Variablen X auch einen don't know-Nichtdeterminismus bei der Wahl des Kopfsymbols h. Dadurch wird bei der Implementierung Backtracking benötigt.

Definition. (Trivialer Unifikationsalgorithmus)

Der triviale Unifikationsalgorithmus generiert für alle freien Variablen X eines Unifikationsproblems höherer Ordnung Γ alle möglichen geschlossenen Terme in η-Lang-β-Normalform und testet für jede Kombination, ob Γ gelöst wurde.

KOLLISION	$\{\lambda x_1,\ldots,x_n.(f\,s_1\ldots s_k) \approx \lambda x_1,\ldots,x_n.(g\,t_1\ldots t_l)\} \uplus S$	$\Rightarrow$	$\bot$
DEKOMPOSITION	$\{\lambda x_1,\ldots,x_n.(f\,s_1\ldots s_k) \approx \lambda x_1,\ldots,x_n.(f\,t_1\ldots t_k)\} \uplus S$	$\Rightarrow$	$\{\lambda x_1,\ldots x_n.s_1 \approx \lambda x_1,\ldots x_n.t_1,\ldots, \lambda x_1,\ldots x_n.s_k \approx \lambda x_1,\ldots x_n.t_k\} \cup S$
ORIENTIERUNG	$\{\lambda x_1,\ldots,x_n.(f\,s_1\ldots s_k) \approx \lambda x_1,\ldots,x_n.(X\,t_1\ldots t_l)\} \uplus S$	$\Rightarrow$	$\{\lambda x_1,\ldots,x_n.(X\,t_1\ldots t_k) \approx \lambda x_1,\ldots,x_n.(f\,s_1\ldots s_l)\} \cup S$
GENERIERUNG	S	$\Rightarrow$	$S\,\{\langle\lambda y_1,\ldots y_p.(h\ (H_1\,y_1\ldots y_p)\ldots (H_r\,y_1\ldots y_p))\,/X\rangle\}$
	falls $\lambda x_1,\ldots,x_n.(X\,s_1\ldots s_k) \approx \lambda x_1,\ldots,x_n.(f\,t_1\ldots t_l) \in S$, h ist eine der Variablen $y_1,\ldots,y_p$ oder die Konstante f		

TABELLE 2.5.1. Transformationsregeln für Huet's Algorithmus

2.5.2. Huet's Algorithmus. Der vorgestellte triviale Unifikationsalgorithmus durchsucht den gesamten Raum aller möglichen typgerechten Termersetzungen für die freien Variablen eines Unifikationsproblems (eingeschränkt nur auf Terme in η-Lang-β-Normalform). Er nutzt bei der Generierung der Terme aber keine Informationen, die er aus dem gestellten Unifikationsproblem selbst gewinnen könnte, um den Suchraum einzuschränken.

Diese Informationen versucht Huet's Algorithmus auszunutzen, um effizienter zu agieren als der triviale Unifikationsalgorithmus.

DEFINITION 2.5.2. Wir nennen einen Term t *starr*, wenn sein Kopfsymbol eine Konstante oder eine gebundene Variable ist. Wir nennen t *flexibel*, falls sein Kopfsymbol eine freie Variable ist.

Jede Gleichung $(s \approx t) \in \Gamma$ fällt unter einen der Gleichungstypen starr-starr (sowohl s als auch t sind starre Terme), starr-flexibel (s ist starr, t ist flexibel), flexibel-starr oder flexibel-flexibel.

2.5.2.1. *Transformationsregeln.* Wir geben nun in Tabelle 2.5.1 die Transformationsregeln an, die Huet's Algorithmus bestimmen.

2.5.2.1.1. Die Regeln KOLLISION und DEKOMPOSITION. Diese beiden Regeln betreffen starr-starre Gleichungen. Sie entsprechenden in ihrer Funktion ihren Pendants beim syntaktischen Unifikationsalgorithmus. Falls die Kopfsymbole der beiden Terme ungleich sind, so wird der Algorithmus mit dem unlösbaren Unifikationsproblem $\bot$ abgebrochen, da es in diesem Falle keine Lösung gibt.

Falls die Kopfsymbole gleiche Funktionssymbole sind, so müssen die Funktionsargumente gleich sein. In diesem Falle greift die Regel DEKOMPOSITION.

2.5.2.1.2. Die Regeln ORIENTIERUNG und GENERIERUNG. Diese beiden Regeln befassen sich mit starr-flexiblen bzw. flexibel-starren Gleichungen. Die Regel ORIENTIERUNG transformiert eine starr-flexible Gleichung in eine flexibel-starre Gleichung.

Die Regel Generierung bedarf einer genaueren Betrachtung. Sie betrifft flexibel-starre Gleichungen der Form

$$\lambda x_1, \ldots, x_n.\,(X\; s_1 \ldots s_k) \approx \lambda x_1, \ldots, x_n.\,(f\; t_1 \ldots t_l)$$

In diesem Falle können wir die freie Variable X wie im trivialen Algorithmus mit einem beliebigen Term der Form

$$\lambda y_1, \ldots y_p.\,(h\;\; (H_1\; y_1 \ldots y_p) \ldots (H_r\; y_1 \ldots y_p))$$

vom Typ $\texttt{type}\,(X)$ ersetzen. Wir können jedoch den Nichtdeterminismus etwas einschränken, da wir wissen, dass das Unifikationsproblem nicht mehr lösbar sein wird, falls wir für das Kopfsymbol h eine Konstante $g \neq f$ wählen. Daher ist h beschränkt auf eine der Variablen $y_1, \ldots y_p$ oder auf die Konstante f. Falls eine der Variablen gewählt wird, spricht man auch von einer *Projektion*, falls die Funktion f gewählt wird, von einer *Imitation*.

Die Regel Generierung führt einen *don't know* Nichtdeterminismus in die Transformationsregeln ein, da man nicht genau weiß, welches Kopfsymbol h zu wählen ist. Dies kann dazu führen, dass man den Transformationsweg zurückverfolgen muss (*Backtracking*).

2.5.2.1.3. Flexibel-flexible Gleichungen. Die in Tabelle 2.5.1 aufgeführten Regeln behandeln nicht den Fall flexibel-flexibler Gleichungen der Form $\lambda x_1, \ldots, x_n.\,(X\; s_1 \ldots s_k) \approx \lambda x_1, \ldots, x_n.\,(Y\; t_1 \ldots t_l)$.

In diesem Falle kann es bei der Wahl von Termen für die Variablen X oder Y keine Einschränkungen mehr geben und man muss alle Terme wie im trivialen Unifikationsalgorithmus generieren.

Falls man jedoch nicht an allen Lösungen interessiert ist, sondern nur an der Lösbarkeit des Unifikationsproblems, ist diese Generierung nicht notwendig, wie Huet's Lemma 2.5.4 zeigt. Dafür benötigen wir zunächst eine Definition eines gelösten Unifikationsproblems.

Definition 2.5.3. Ein Unifikationsproblem Γ heißt *gelöst*, wenn es nur noch flexibel-flexible Gleichungen enthält.

Lemma 2.5.4. *(Huet) Jedes gelöste Unifikationsproblem besitzt eine Lösung.*

Beweis. Wir haben angenommen, dass es für jeden Grundtyp g eine Konstante c_g mit Typ $\texttt{type}\,(c_g) = g$ gibt. Wir definieren nun σ als die Substitution, die jeder Variablen X vom Typ $\tau_1 \to \ldots \to \tau_n \to \iota$ einen Term der Form $\lambda x_1, \ldots, x_n.c_\iota$ zuweist, wobei gilt $\forall 1 \leq i \leq n : \texttt{type}\,(x_i) = \tau_i$ und $\texttt{type}\,(c_\iota) = \iota$.

Sei nun Γ ein gelöstes Unifikationsproblem. Dann ist σ eine Lösung von Γ. Denn jede Gleichung der Form

$$\lambda x_1, \ldots, x_n.\,(X\; s_1 \ldots s_k) \approx \lambda x_1, \ldots, x_n.\,(Y\; t_1 \ldots t_l)$$

wird durch σ gelöst:

$$(\lambda x_1, \ldots, x_n.\,(X\; s_1 \ldots s_k))\,\sigma = \lambda x_1, \ldots, x_n.c_\iota = (\lambda x_1, \ldots, x_n.\,(Y\; t_1 \ldots t_l))\,\sigma$$

□

2.5.2.2. *Korrektheit.* Wir zeigen die Korrektheit des Transformationssystems aus Tabelle 2.5.1.

PROPOSITION. *Durch Anwendung der Regeln* KOLLISION, DEKOMPOSITION *und* ORIENTIERUNG *auf ein Unifikationsproblem* Γ *verändert sich die Lösungsmenge nicht.*

BEWEIS. Durch Inspektion. □

PROPOSITION. *Die Anwendung der Regeln* KOLLISION *und* DEKOMPOSITION *auf ein Unifikationsproblem* Γ *terminiert nach endlich vielen Schritten. Das dadurch berechnete Unifikationsproblem* Γ' *besitzt keine starr-starren Gleichungen mehr.*

BEWEIS. Wir wählen als Komplexitätsmaß die Anzahl der Funktionssymbole in Γ. Wären noch starr-starre Gleichungen in Γ' enthalten, könnte entweder KOLLISION oder DEKOMPOSITION angewandt werden. □

PROPOSITION 2.5.5. *Falls ein Unifikationsproblem* Γ *durch die Transformationsregeln* KOLLISION, DEKOMPOSITION, ORIENTIERUNG *und* GENERIERUNG *in ein gelöstes Unifikationsproblem* Γ' *transformiert werden kann, so ist* Γ *unifizierbar.*

BEWEIS. Wir zeigen die Behauptung mittels Induktion über die Anzahl n der Transformationsschritte. Falls $n = 0$, so gilt die Behauptung wegen Huet's Lemma.

Sei also $n > 0$. Ist der erste Transformationsschritt die Regel KOLLISION, so ist Γ nicht unifizierbar. Ist die erste Regel DEKOMPOSITION oder ORIENTIERUNG, so wenden wir die Induktionsbehauptung an.

Falls die erste angewandte Regel GENERIERUNG ist, dann hat das sich dadurch ergebende Unifikationsproblem Γ' nach Induktionsbehauptung die Lösung σ'. Die Substitution definiert durch

$$\sigma = \sigma' \{\langle \lambda y_1, \ldots y_p.\, (h\ (H_1\, y_1 \ldots y_p) \ldots (H_r\, y_1 \ldots y_p))\rangle / X\}$$

ist eine Lösung für Γ. □

PROPOSITION 2.5.6. *Falls ein Unifikationsproblem* Γ *unifizierbar ist, also einen Unifikator* σ *besitzt, so berechnen die Transformationsregeln* KOLLISION, DEKOMPOSITION, ORIENTIERUNG *und* GENERIERUNG Γ *ein gelöstes Unifikationsproblem* Γ'.

BEWEIS. Wir wenden zunächst die Regeln KOLLISION und DEKOMPOSITION erschöpfend auf Γ an. Dieser Prozess terminiert, wie wir gezeigt haben, und er ist lösungserhaltend. Wir haben als Ergebnis ein Unifikationsproblem Γ_1, das keinerlei starr-starren Gleichungen mehr enthält. Falls Γ_1 gelöst ist, so setzen wir $\Gamma' := \Gamma_1$.

Anderenfalls enthält Γ_1 starr-flexible oder flexibel-starre Gleichungen. Starr-flexible Gleichungen werden mittels der Regel ORIENTIERUNG in flexibel-starre transformiert.

Wir wenden Induktion über die Größe der Substitution σ an.

Es existiert also eine Gleichung der Form

$$\lambda x_1, \ldots, x_n.\,(X\, s_1 \ldots s_k) \approx \lambda x_1, \ldots, x_n.\,(f\, t_1 \ldots t_l)$$

Sei nun $\langle \lambda y_1, \ldots, x_k.\,(h\, u_1 \ldots u_r)\,/X\rangle \in \sigma$.

Die Regel GENERIERUNG transformiert die Gleichung $\lambda x_1, \ldots, x_n.\,(X\, s_1 \ldots s_k) \approx \lambda x_1, \ldots, x_n.\,(Y\, t_1 \ldots t_l)$ in die Gleichung

$$\begin{array}{c}\lambda x_1, \ldots, x_n.\,(X\, s_1 \ldots s_k) \approx \lambda x_1, \ldots, x_n.\,(f\, t_1 \ldots t_l)\\ \{\langle \lambda y_1, \ldots y_p.\,(h\ (H_1\, y_1 \ldots y_p) \ldots (H_r\, y_1 \ldots y_p))\,/X\rangle\}\end{array}$$

Für das so gewonnene Unifikationsproblem Γ_2 ist die Substitution σ_2 eine Lösung:

$$\sigma_2 := (\sigma \setminus \{\langle X\sigma/X\rangle\}) \cup \{\langle y_1, \ldots, y_p.u_1/H_1\rangle, \ldots, \langle y_1, \ldots, y_p.u_r/H_r\rangle\}$$

Es gilt $|\sigma_2| < |\sigma|$ und wir wenden die Induktionsbehauptung an. □

2.6. Matching höherer Ordnung

DEFINITION 2.6.1. (Matchingproblem höherer Ordnung)

Ein *Matchingproblem höherer Ordnung* (higher order matching problem, *HOMP*) Γ der Form $\Gamma = \{s_1 \approx u_1, \ldots, s_n \approx u_n\}$ ist ein Unifikationsproblem mit folgender Einschränkung: $\forall i : 1 \leq i \leq n$ sind die Terme s_i, u_i vom Grundtyp und alle Terme u_i sind geschlossen. Die Lösung eines HOMP ist ein Unifikator gemäß Definition 2.4.1.

Wir bezeichnen die linke Seite eines Matchingproblems auch als *Anfrage*, die rechte Seite als *Daten*.

Wir betrachten in diesem Abschnitt lediglich das Matching zweiter Ordnung, da dieses in bezug auf die Mächtigkeit unmittelbar oberhalb des Kontextmatchings liegt.

BEMERKUNG. Ein Matchingproblem zweiter Ordnung ist ein Matchingproblem, dessen freie Variablen höchstens zweiter Ordnung und dessen gebundene Variablen höchstens dritter Ordnung sind.

Huet und Lang zeigten die Entscheidbarkeit von Matching zweiter Ordnung [**Huet1978**]:

PROPOSITION 2.6.2. *(Huet und Lang) Matching zweiter Ordnung ist entscheidbar. Es existiert also ein Algorithmus, der für ein beliebiges Matchingproblem zweiter Ordnung Γ beantwortet, ob es einen Unifikator σ gibt oder nicht.*

BEWEIS. Wir zeigen, dass Huet's Algorithmus (Tabelle 2.5.1) für Matchingprobleme zweiter Ordnung stets terminiert und verwenden dazu das Komplexitätsmaß (r, v, o), wobei r die Summe der Größe der rechten Seiten des Problems ist und v die Anzahl der Variablen. o bezeichnet die Anzahl der Gleichungen, die einer Orientierung bedürfen. Die lexikographische Ordnung von (r, v, o) fällt bei Anwendung der Transformationsregeln streng monoton.

Die Regel KOLLISION läßt alle Komponenten auf 0 sinken. Die Regel DEKOMPOSITION verringert r. Die Regel ORIENTIERUNG läßt r und v unverändert, dekrementiert aber o.

Bei der Regel GENERIERUNG unterscheiden wir zwischen den Fällen der Imitation und der Projektion.

Im Falle der Imitation folgt dieser im nächsten Schritt stets eine DEKOMPOSITION, so dass die Größe der rechten Seiten r kleiner wird.

Im Falle der Projektion hat diese die Form $\lambda x_1, \ldots, x_n.x_i$, da wir es mit einem Matchingproblem zweiter Ordnung zu tun haben. Damit werden keine neuen freien Variablen eingeführt und v sinkt. □

2.6.1. Weitere Ergebnisse. Dowek zeigte 1992, dass Matching dritter Ordnung entscheidbar ist. Padovani [**Pad2000**] konnte 1997 beweisen, dass Matching vierter Ordnung ebenfalls entscheidbar ist, ein Ergebnis, dass Comon und Jurski mit einer andere Beweistechnik über die Konstruktion eines Baumautomaten ebenfalls erhielten. Für Ordnungen höher als vier ist die Frage zur Zeit ungeklärt [**Dow2001**].

Matching zweiter und dritter Ordnung ist $\mathcal{NP}$-vollständig, wie Comon und Jurski zeigten [**Com1997**].

KAPITEL 3

Kontext-Matching

Kontext-Matching ist eine Erweiterung des syntaktischen Matching dergestalt, dass man nun neben Variablen erster Ordnung Kontextvariablen zuläßt. Kontextvariablen können dabei instanziiert werden durch Kontexte, das heißt Terme mit einem speziellen Operator, dem Loch (siehe Definition 2.3.6).

Wir beginnen das Kapitel mit einer Einordnung von Kontextmatching in die im vorangegangenen Kapitel vorgestellten Probleme Syntaxmatching und Matching zweiter Ordnung. Wir fahren fort mit einigen grundlegenden Erkenntnissen, die in diesem Kapitel verwendet werden.

Wir untersuchen dann in Abschnitt 3.3 einige eingeschränkte Kontextmatchingprobleme auf ihre Komplexität.

Im Abschnitt 3.4 betrachten wir schließlich algorithmische Möglichkeiten, Kontextmatching zu lösen.

3.1. Einordnung

Im Gegensatz zu Matchingproblemen erster Ordnung, in denen Variablen lediglich durch Terme instanziiert werden und daher nur als Blätter im Termbaum auftreten können, können Kontextvariablen als monadische Operatoren an beliebiger Stelle im Term auftauchen. Sie erlauben es, beliebig tiefe Ausdrücke in nur einem Schritt zu matchen. Damit kann Kontextmatching in der Tat als Erweiterung des Syntaxmatchings angesehen werden.

Auf der anderen Seite ist Kontextmatching eine Einschränkung des Matchings zweiter Ordnung, denn ein Kontext $C\,[\Box]$ kann aufgefaßt werden als eine 2nd-order-Funktion vom Typ $\iota \to \iota$. Dabei darf die gebundene Variable nur einmal auf der rechten Seite auftauchen.

Damit kann man den Kontext $C\,[\Box]$ übersetzen in eine Abstraktion$\lambda\Box.t$, wobei der Term t die gebundene Variable $\Box$ nur einmal enthalten darf.

Definition 3.1.1. Eine *Kontextvariable* $X\,[]$ ist eine Variable vom Typ $\iota \to \iota$ mit genau einem Vorkommen des Arguments und kann einen *Kontext* binden.

Die Anwendung einer Kontextvariable $X\,[(f\,g)]$ läßt sich damit übersetzen nach $(X\ (f\,g))$, d. h. die Applikation der Kontextvariablen X auf das Argument $(f\,g)$. Im Falle eines konkreten Kontexts $C\,[(f\,g)]$ ergibt dies $((\lambda\Box.t)\ (f\,g))$.

Übersetzt man ein Kontextmatchingproblem, um es mit einem Matchingalgorithmus für Matching zweiter Ordnung zu lösen, kann man zum Teil Lösungen erhalten, die im Rahmen des Kontextmatchings nicht zulässig sind, da in der lösenden Substitution Variablen mehrfach gebunden sein könnten.

Aus Sicht der Komplexität bedeutet dies zunächst, dass Kontextmatching zwischen dem in Linearzeit lösbaren Syntaxmatching und dem $\mathcal{NP}$-vollständigen Matching zweiter Ordnung steht. Ziel vieler Untersuchungen ist es, diese Grenze schärfer ziehen zu können, indem Einschränkungen der Probleme auf beiden Seiten des Komplexitätsniveaus vorgenommen werden.

3.2. Grundlegendes

Wir erweitern zunächst den Termbegriff aus Kapitel 2 um Kontextvariablen. Viele Definitionen sind analog.

Definition 3.2.1. Ein $T(\Sigma, V, \mathcal{C})$-*Kontextterm* ist ein Term erster Ordnung erweitert um Kontextvariablen. Dabei verwenden wir die Signatur Σ, eine abzählbar unendliche Menge von individuellen Variablen V und eine abzählbar unendliche Menge von Kontextvariablen $\mathcal{C}$ mit $\Sigma \cap V \cap \mathcal{C} = \emptyset$. Die Menge der Kontextterme ist induktiv definiert:

(1) jede Variable ist ein Term: $\forall x \in V : x \in T(\Sigma, V, \mathcal{C})$
(2) das Loch ist ein Term: $\Box \in T(\Sigma, V, \mathcal{C})$
(3) die Applikation eines Funktionssymbols auf Terme ergibt wieder einen Term:

$$\forall n \geq 0, \forall f \in \Sigma^n, \forall t_1, \ldots, t_n \in T(\Sigma, V, \mathcal{C}) : f(t_1, \ldots, t_n) \in T(\Sigma, V, \mathcal{C})$$

(4) Eine Kontextvariable angewandt auf einen Term ist ein Term: $\forall X \in \mathcal{C}, t \in T(\Sigma, V, \mathcal{C}) : X[t] \in T(\Sigma, V, \mathcal{C})$

Bemerkung 3.2.2. Wir bezeichnen ein $t \in (\Sigma, V, \mathcal{C})$ als Kontextterm oder kurz als Term. Ein t mit genau einem Vorkommen des Lochs $\Box$ bezeichnen wir kurz als *Kontext*.

Definition 3.2.3. Analog zu Definition 2.1.2 definieren wir $Var(t)$ als die Menge der Variablen und $CVar(t)$ als die Menge der Kontextvariablen in t.

Definition. Eine *Termgleichung* ist eine Gleichung $s \approx t$ zwischen zwei Kontexttermen $s, t \in T(\Sigma, V, \mathcal{C})$. Eine *Kontextgleichung* ist eine Gleichung zwischen zwei Kontexten.

Wir erweitern den Begriff Kontext auf mehrere „Löcher" und führen den Begriff des Multikontexts ein.

Definition 3.2.4. Ein *Multikontext* ist ein Term mit mehreren Löchern. Jedes Loch darf im Term genau einmal vorkommen; die Reihenfolge der Löcher ist dabei relevant. Wir bezeichnen Multikontexte mit $C[\Box, \ldots, \Box]$. Innerhalb von C gibt es keine Löcher, außer die, die innerhalb der eckigen Klammern stehen.

Wir schreiben $\Box^k$ für k Löcher und $\Box_i$ für das i-te Loch.

Multikontexte können wiederum als Terme zweiter Ordnung aufgefasst werden. Für einen Multikontext $C[\Box_1, \Box_2]$ ergibt sich ein Term $\lambda\Box_1, \Box_2.t$ mit Typ $\iota \to \iota \to \iota$, wobei die gebundenen Variablen $\Box_1$ und $\Box_2$ jeweils nur einmal in der richtigen Reihenfolge im Term t auftauchen dürfen.

Multikontexte sind ein rein abstraktes Konzept; daher erlauben wir auch keine Multikontextvariablen. Kontextvariablen können stets nur durch einen „einfachen" Kontext instanziiert werden.

Wir erweitern die Definition der Substitution aus Kapitel 2.1.2, um auch Kontextvariablen substituieren zu können:

Definition 3.2.5. Sei Σ eine Signatur, V eine abzählbar unendliche Menge von Variablen und $\mathcal{C}$ eine abzählbar unendliche Menge von Kontextvariablen mit $\Sigma \cap V \cap \mathcal{C} = \emptyset$. Eine *Substitution* ist eine Funktion $\sigma : V \cup \mathcal{C} \to T(\Sigma, V, \mathcal{C})$, die endlich vielen $x \in V$ einen Wert $\sigma(x) \neq x$ zuweist und x für alle anderen Variablen. Analog wird endlich vielen $X \in \mathcal{C}$ ein Wert $\sigma(X) \neq X$ zugewiesen und allen anderen Kontextvariablen $X[\Box]$.

Substitutionen können keine Löcher einführen oder entfernen. Wir erweitern die Definition von σ auf $T(\Sigma, V, \mathcal{C})$-Terme t:

- Falls $t = x \in V$, so gilt $\sigma'(t) := \sigma(t)$
- Falls $t = X \in \mathcal{C}$, so gilt $\sigma'(X[s]) := (\sigma(X))[\sigma'(s)]$
- Falls $t = f(s_1, \ldots, s_n)$, so gilt $\sigma'(t) := f(\sigma'(s_1), \ldots, \sigma'(s_n))$

Wir verwenden für *Matchingprobleme* die um Kontextvariablen erweiterte Definition 2.6.1 aus Kapitel 2.6.

Wir zeigen nun, dass man Matchingprobleme Γ mit $|\Gamma| \geq 2$ auf Matchingprobleme mit nur einer Termgleichung zurückführen kann und dass man jede Termgleichung in eine Kontextgleichung umwandeln kann.

Proposition 3.2.6. *Sei $h \in T(\Sigma, V, \mathcal{C})$ mit $\mathtt{ar}(h) \geq 2$. Dann gilt:*

(1) *Jede Konjunktion von Termgleichungen kann in eine äquivalente einzige Termgleichung umgeschrieben werden.*
(2) *Jede Termgleichung kann in eine äquivalente Kontexttermgleichung umgeschrieben werden.*

Beweis. Sei $k := \mathtt{ar}(h) \geq 2$.

Zum ersten Fall: Sei $s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n$ die Menge der umzuschreibenden Termgleichungen. Dann ist die äquivalente Termgleichung wie folgt:

$$h(s_1, \ldots, s_{k-1}, h(s_k, \ldots, h(s_{n-k+1}, \ldots, s_n))) \approx h(t_1, \ldots, t_{k-1}, h(t_k, \ldots, h(t_{n-k+1}, \ldots, t_n)))$$

Zum zweiten Fall: Sei $s \approx t$ die Termgleichung. Dann ist

$$h\Big(\underbrace{\Box, s, \ldots, s}_{k\text{-fach}}\Big) \quad = \quad h(\Box, t, \ldots, t)$$

□

Jede Gleichung aus Multikontexten kann in eine Konjunktion von Kontextgleichungen umgewandelt werden:

PROPOSITION. *Sei* $C[\Box^m] \approx D[\Box^n]$ *eine Gleichung aus Multikontexten. Diese ist entweder unlösbar oder sie kann ersetzt werden durch eine Konjunktion aus Term- und Kontextgleichungen. Die Ersetzung kann in linearer Zeit und linearem Platz geschehen.*

BEWEIS. Betrachten wir eine Gleichung aus zwei Multikontexten.

$$C[f(C_1[\Box], C_2[\Box])] \approx D[g(D_1[\Box], D_2[\Box])]$$

Die Löcher der linken Seite müssen in ihrer Position den Löchern der rechten Seite entsprechen. Die Funktionssymbole und die Löcher schließen die Kontexte C_1, C_2 sowie D_1, D_2 ein, so dass die Symbole am längsten gemeinsamen Präfix der beiden Löcher jeder Seite gleich sein müssen. Das bedeutet, dass für Lösbarkeit der Gleichung $f = g$ gelten muss.

In diesem Falle kann die Gleichung aber durch folgende Konjunktion ersetzt werden:

$$C[] \approx D[] \wedge C_1[] \approx D_1[] \wedge C_2[] \approx D_2[]$$

Dieses Prinzip funktioniert auch bei mehr als zwei Löchern analog, ebenso, falls die Stelligkeit $\mathtt{ar}(f) = \mathtt{ar}(g) > 2$ ist. Es ist nicht notwendig, dass f und g in jedem Argument ein Loch besitzen. Zwei Argumente mit Löchern reichen aus, um die Position der Funktionssymbole eindeutig festzulegen. □

DEFINITION 3.2.7. Der *Variablenpräfix* eines Vorkommens einer (individuellen oder Kontext-) Variablen in einem Kontextterm ist die Sequenz der Kontextvariablen auf dem Pfad von der Wurzel des Terms bis einschließlich zu der Variablen.

BEISPIEL. Im Term $X[f(a, y, Y[g(x)])]$ sind die Präfixe wie folgt: $\mathtt{pref}(X) = X$, $\mathtt{pref}(y) = Xy$, $\mathtt{pref}(Y) = XY$ und $\mathtt{pref}(x) = XYx$.

3.3. Komplexität einiger Probleme

3.3.1. Lineares Kontextmatching.

DEFINITION 3.3.1. Ein Kontextmatchingproblem ist *linear*, wenn jede Variable höchstens einmal vorkommt.

BEISPIEL 3.3.2. Das Kontextmatchingproblem

$$\{X[f(a,b), Y[a]] \approx g(f(a,b), f(a)), Z[f(x)] \approx f(f(a))\}$$

ist linear, das Problem

$$\{X[f(a,b), Y[a]] \approx g(f(a,b), f(a)), Y[f(x)] \approx f(f(a))\}$$

hingegen nicht, da die Variable Y zweimal vorkommt.

Man kann lineares Kontextmatching so auffassen, dass (normale) Variablen Platzhalter für Kontexte sind.

Definition 3.3.3. Ein Kontextmatchingproblem heißt *monadisch*, falls alle in ihm vorkommenden Funktionssymbole höchstens die Stelligkeit 1 besitzen.

Lemma. *Selbst im „einfachen" Fall des linearen monadischen Kontextmatchings kann es bereits exponentiell viele Lösungen geben.*

Beweis. Als Beispiel betrachte man das folgende lineare monadische Kontextmatchingproblem:

$$\{X_1 [Y_1 [h (X_2 [Y_2 [h (\ldots h (X_n [Y_n [a]]))]])]] \approx g (h (g (h \ldots h (g (a)))))\}$$

Auf beiden Seiten der Gleichung existiert die gleiche Anzahl des Funktionssymbols h, so dass die Kontextvariablen X und Y lediglich das Funktionssymbol g matchen können. Damit gibt es für jedes Paar X_i, Y_i genau zwei Lösungsmöglichkeiten: $\{\langle \Box / X_i \rangle , \langle g (\Box) / Y_i \rangle\}$ sowie $\{\langle g (\Box) / X_i \rangle , \langle \Box / Y_i \rangle\}$. Beide Substitutionspaare können unabhängig voneinander kombiniert werden, so dass sich 2^n Lösungsmöglichkeiten ergeben.

□

Korrolar 3.3.4. *Lineares Kontextmatching ist in $\mathcal{P}$.*

Beweis. Lineares Kontextmatching kann durch dynamisches Programmieren gelöst werden. Der Algorithmus, der in Kapitel 3.4.1 vorgestellt wird, benötigt hierzu $O\left(n^3\right)$ Zeit und $O\left(n^2\right)$ Platz. □

3.3.2. Geschichtetes Kontextmatching.

Definition 3.3.5. Ein Kontextmatchingproblem heißt *geschichtet*, falls jedes Auftreten einer individuellen oder einer Kontextvariablen denselben Variablenpräfix besitzt.

Beispiel. Das Kontextmatchingproblem

$$\{X [g (Y [h (z, c)] , Y [x, h (c, z)])] \approx g (f (h (c, c)) , f (f (a) , h (c, c)))\}$$

ist geschichtet, da es folgende Variablenpräfixe aufweist: $\texttt{pref}(X) = X$, $\texttt{pref}(Y) = XY$, $\texttt{pref}(x) = XYx$ und $\texttt{pref}(z) = XYz$. Dagegen ist das Kontextmatchingproblem

$$\{X [g (Y (h (z, c))) \approx g (f (h (c, c))) , Y [h (f (a) , x) \approx f (h (f (a) , b))]]\}$$

nicht geschichtet, da für die erste Gleichung $\texttt{pref}(Y) = XY$ und die zweite Gleichung $\texttt{pref}(Y) = Y$ gilt.

Theorem 3.3.6. *Geschichtetes Kontextmatching ist $\mathcal{NP}$-vollständig.*

Beweis. Wir reduzieren das Problem $3SAT$, welches als $\mathcal{NP}$-vollständig bekannt ist, auf Kontextmatching. Im Folgenden skizzieren wir hierzu den Beweis aus [**Schm2003**]. □

Definition. $3SAT$ ist das Erfüllungsproblem für boolesche Formeln in konjunktiver Normalform, bei der jede Klausel genau 3 Literale enthält.

Beispiel 3.3.7. $(x \vee y \vee z) \wedge (\neg x \vee z \vee a)$ ist eine $3SAT$-Formel mit den Variablen $\{a, x, y, z\}$.

Lemma 3.3.8. *$3SAT$ ist $\mathcal{NP}$-vollständig.*

Beweis. Siehe [**Hop1994**, Kapitel 13.2]. □

Konstruktion des geschichteten Kontextmatchingproblems. Wir nehmen an, dass die zu reduzierende Instanz I_{3SAT} von $3SAT$ aus den Klauseln $c_1, \ldots, c_m$ besteht und in diesen Klauseln die (negierten oder nicht-negierten) Variablen $x_1, \ldots, x_n$ vorkommen. Diese aus m Klauseln bestehende Formel übersetzen wir in ein geschichtetes Kontextmatchingproblem aus $m+1$ Gleichungen:

$$P_{3SAT} := \{s_{0,1} \approx t_{0,\epsilon}, \ldots, s_{m,1} \approx t_{m,\epsilon}\}$$

Jede der m Klauseln von I_{3SAT} entspricht so einer Gleichung des Kontextmatchingproblems P_{3SAT}.

Die erste Gleichung $s_{0,1} \approx t_{0,\epsilon}$ ist dabei eine gesonderte Gleichung, die dafür sorgt, dass die Lösungen des Kontextmatchingproblems P nur gültigen Besetzungen der Variablen des $3SAT$-Problems entsprechen. Wir werden diese Gleichung im Folgenden *Wahrheitsgleichung* nennen.

Im weiteren Verlauf indiziert der Bezeichner i Gleichungen bzw. Klauseln $(0 \leq i \leq m)$ und der Bezeichner j Variablen $(1 \leq j \leq n)$.

In den Kontextgleichungen von P_{3SAT} werden die Kontextvariablen X_j und Y_j $(1 \leq j \leq n)$ verwendet. Die Wahrheitsgleichung wird dafür sorgen, dass die beiden Kontextvariablen jeweils nur durch zwei Kontexte substituiert werden können. Jede der beiden Kombinationen repräsentiert dabei einen Wahrheitswert:

$$\begin{aligned} \texttt{True} &\equiv \{\langle \Box, X_j \rangle, \langle g(\Box, c)/Y_j \rangle\} \\ \texttt{False} &\equiv \{\langle g(\Box, c)/X_j \rangle, \langle \Box/Y_j \rangle\} \end{aligned} \tag{3.3.1}$$

Wir werden die Wahrheitswerte `True` und `False` mit den jeweiligen Substitutionen aus Gleichung 3.3.1 identifizieren.

Jede Gleichung von P_{3SAT} besteht aus $n+1$ Schichten. Dabei repräsentieren die n oberen Schichten einer Gleichung $s_{i,0} \approx t_{i,\epsilon}$ die n Variablen von I_{3SAT}. Die unterste Schicht (genannt *Ergebnisschicht*) kodiert, ob die Klausel c_i durch die Belegungen der Kontextvariablen X_j, Y_j in den höheren Schichten erfüllt wird oder nicht. Kommt die Variable x_j in Klausel c_i vor, so wird in Schicht j eine sogenannte *aktive* Schicht verwendet, ansonsten eine *passive* Schicht.

Eine *passive Schicht* j in Gleichung i (d. h. x_j kommt in Klausel c_i nicht vor) sorgt dafür, dass X_j und Y_j nur durch Kontexte substituiert werden können, die den Wahrheitswerten `False` oder `True` entsprechen (vgl. Gleichung 3.3.1). Die Wahrheitsgleichung besteht lediglich aus passiven Schichten und sorgt somit global dafür, dass X_j und Y_j $(1 \leq j \leq n)$ nur durch gültige Kontexte ersetzt werden können.

In einer *aktiven Schicht* j in Gleichung i (d. h. x_j oder $\neg x_j$ kommt in Klausel c_i vor) wird eine Entscheidung über die Besetzung von X_j und Y_j (entweder `True` oder `False`) getroffen. Die Summe der Entscheidungen aller Schichten $1, \ldots, n$ führt zu einer bestimmten Ergebnisschicht $n+1$, die aussagt, ob, basierend auf den Besetzungen der Kontextvariablen X_j und Y_j in den oberen Schichten, die Klausel c_i erfüllt wird oder nicht. Falls c_i erfüllt ist, wird die Ergebnisschicht so gewählt, dass die Kontextgleichung $s_{i,1} \approx t_{i,\epsilon}$ gelöst werden kann, ansonsten so, dass sie nicht gelöst werden kann.

Die Ergebnisschicht wird also abhängig von den Entscheidungen über die Kontextvariablenbesetzungen der oberen Schichten konstruiert. Dafür werden die in höheren Schichten getroffenen Entscheidungen in einem Stringliteral α aufgezeichnet. Jede Stelle j des Stringliterals α repräsentiert ein Literal der Klausel c_i mit Variable x_j. Falls Klausel c_i die Variable x_j nicht enthält, so wird $\alpha\,|_j := *$ gesetzt ($\alpha\,|_j$ bezeichne die j-te Stelle des Strings α). Falls die Klausel c_i die Variable x_j in einem Literal enthält, das unwahr unter der gewählten Substitution für X_j und Y_j wird, so wird $\alpha\,|_j := 0$ gesetzt. Wird das Literal wahr unter der gewählten Substitution, so wird $\alpha\,|_j := 1$ gesetzt.

$$a\,|_j := \begin{cases} * & c_i \text{ enthält Variable } x_j \text{ nicht.} \\ 0 & x_j \text{ ist in Literal, das unwahr unter der Besetzung} \\ & \text{von } X_j, Y_j \text{ wird.} \\ 1 & x_j \text{ ist in Literal, das wahr unter der Besetzung} \\ & \text{von } X_j, Y_j \text{ wird.} \end{cases}$$

Für die Wahrheitsgleichung wird eine erfüllbare Ergebnisschicht gebildet. Für alle anderen Gleichungen wird nur dann eine erfüllbare Ergebnisschicht gewählt, falls der String α, der zu der Ergebnisschicht führte, mindestens eine 1 enthält. Dies entspricht der Aussage, dass mindestens ein Literal der Klausel c_i wahr ist, womit die gesamte Klausel c_i wahr ist.

Damit erhalten wir folgende rekursive Definition der Terme auf beiden Seiten des zu konstruierenden Kontextmatchingproblems P_{3SAT}:

$$s_{i,j} := \begin{cases} f\left(X_j\left[g\left(g\left(x_{i,j}, Y_j\left[s_{i,j+1}\right]\right), y_{i,j}\right)\right]\right) & \text{aktiv: falls } i \neq 0 \text{ und } x_j \text{ in } c_i \\ & \text{vorkommt} \\ f\left(X_j\left[Y_j\left[s_{i,j+1}\right]\right]\right) & \text{passiv: sonst} \end{cases}$$

$$t_{i,\alpha} := \begin{cases} f\left(g\left(g\left(g\left(c, t_{i,\alpha 0}\right), g\left(t_{i,\alpha 1}, c\right)\right), c\right)\right) & \text{aktiv: } i \neq 0 \text{ und } x_{|\alpha|+1} \text{ ist} \\ & \text{positiv in } c_i \\ f\left(g\left(g\left(g\left(c, t_{i,\alpha 1}\right), g\left(t_{i,\alpha 0}, c\right)\right), c\right)\right) & \text{aktiv: } i \neq 0 \text{ und } x_{|\alpha|+1} \text{ ist} \\ & \text{negativ in } c_i \\ f\left(g\left(t_{i,\alpha *}, c\right)\right) & \text{passiv: sonst} \end{cases}$$

Dabei gilt $0 \leq i \leq m$ und $1 \leq j = |\alpha| + 1 \leq n$. Im Falle $i = 0$ (Wahrheitsgleichung) wird stets eine passive Schicht gewählt. Für die Ergebnisschicht $n+1$ werden folgende Terme gesetzt:

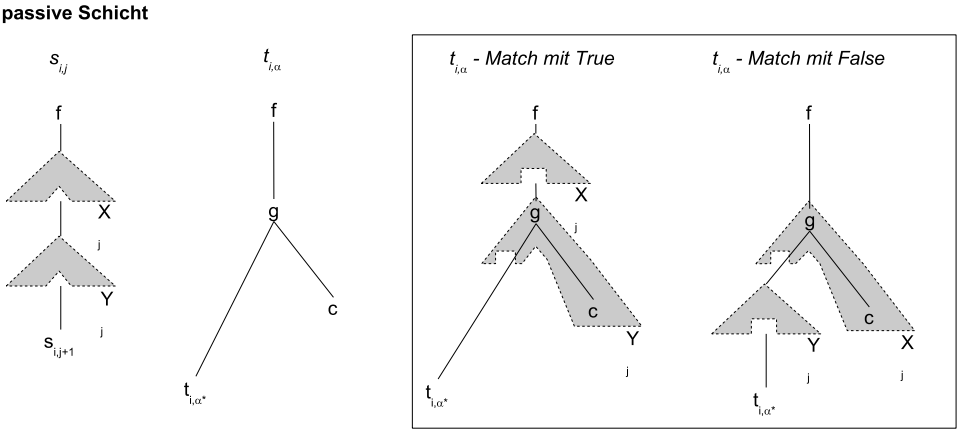


ABBILDUNG 3.3.1. passive Schichten

$$s_{i,n+1} := a$$

$$t_{i,\alpha} := \begin{cases} a & \text{falls } i = 0 \text{ oder } \alpha \text{ mindestens eine 1 enthält} \\ b & \text{sonst} \end{cases} \tag{3.3.2}$$

Beweis der Reduzierbarkeit von I_{3SAT} auf P_{3SAT}. Wir haben zu zeigen, dass P_{3SAT} dann und nur dann eine Lösung hat, falls I_{3SAT} lösbar ist. Weiterhin müssen wir zeigen, dass I_{3SAT} in polynomieller Zeit auf P_{3SAT} reduzierbar ist.

Wir betrachten zunächst $s_{i,j}$ und $t_{i,\alpha}$ und stellen fest, dass beide Terme in allen Fällen $1 \leq i \leq n$ nur dann gematcht werden können, wenn gilt $j = |\alpha| + 1$. Das gilt, da auf beiden Seiten der Gleichungen $s_{i,j} \approx t_{i,\alpha}$ dieselbe Anzahl von Bezeichnern f vorkommt. Damit sind die Positionen von f fixiert, und es kann kein f durch eine der Kontextvariablen X_j, Y_j gematcht werden. Die einzelnen Schichten jeder Gleichung sind somit getrennt und wir können jede Schicht gesondert untersuchen.

Wir betrachten zunächst eine einzelne passive Schicht j. In einer solchen Schicht sind lediglich zwei Lösungen möglich:

- (Abbildung 3.3.1) `True` $\equiv \{\langle \square, X_j \rangle, \langle g(\square, c)/Y_j \rangle\}$
- (Abbildung 3.3.1) `False` $\equiv \{\langle g(\square, c)/X_j \rangle, \langle \square / Y_j \rangle\}$

Wir erinnern uns, dass die Wahrheitsgleichung $s_{0,1} \approx t_{0,\epsilon}$ nur passive Schichten enthält. Damit kann jedes der Kontextvariablenpaare (X_j, Y_j) nur eine der beiden o. g. Besetzungen haben. Jede der beiden Besetzungen entspricht einem Wahrheitswert `True` oder `False`.

Wir kommen nun zu einer aktiven Schicht. Auch hier gibt es - wie in Abbildungen 3.3.2 und 3.3.3 gezeigt - zwei mögliche Lösungen. Wir untersuchen zunächst den Fall einer positiven aktiven Schicht. Im Fall der ersten Lösungsmöglichkeit ($(X_j, Y_j) \equiv$ `True`) wird die nächste untere Schicht $s_{i,j+1}$

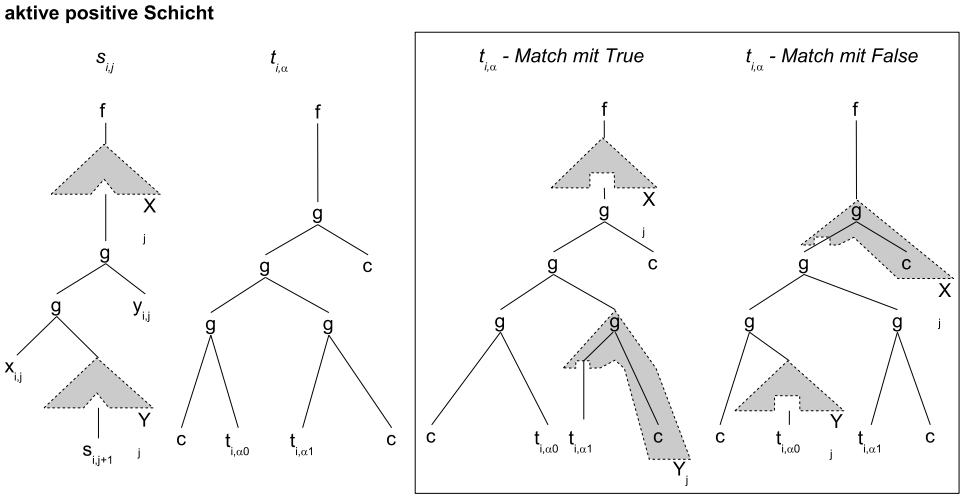


Abbildung 3.3.2. aktive positive Schichten

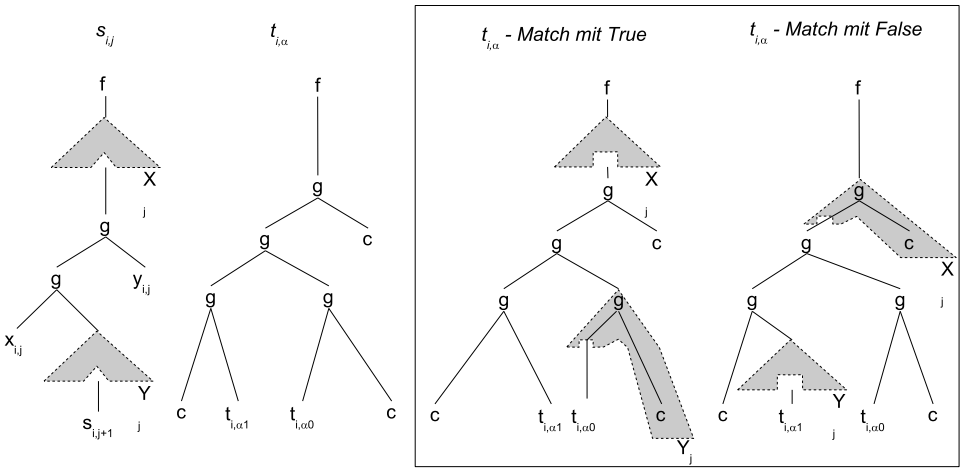


Abbildung 3.3.3. aktive negative Schichten

mit $t_{i,\alpha 1}$ gematcht, wodurch ein positiver Ergebnisbeitrag für die Ergebnisschicht geleistet wird (da 1 mit α konkateniert wird). Bei der zweiten Lösungsmöglichkeit ($(X_j, Y_j) \equiv$ `False`) matcht die nächstfolgende Schicht $s_{i,j+1}$ den Subterm $t_{i,\alpha 0}$, womit α ein negativer Beitrag für die Ergebnisschicht hinzugefügt wird.

Bei einer negativen aktiven Schicht erfolgt das Matching äquivalent, mit dem Unterschied, dass die Konkatenation des Strings α symmetrisch gedreht wird (siehe Abbildung 3.3.3).

Wir können also feststellen, dass in der Ergebnisschicht die einzelnen Stellen von α mit den Wahrheitswerten der Literale in der Klausel c_i korrespondieren.

Gemäß Gleichung 3.3.2 schließen wir jede Matchinggleichung in der Ergebnisschicht mit einem lösbaren Matchingproblem $a \approx a$ ab, falls α mindestens eine 1 enthält (was der Situation entspricht, dass c_i mindestens ein Literal enthält, das den Wert `True` hat). Ansonsten wird die Ergebnisschicht mit einem nichtlösbaren Matchingproblem $a \approx b$ belegt, was folgender Situation entspricht: Die Wahl der Substitutionen für (X_j, Y_j), die zu dieser Ergebnisschicht geführt haben, lassen die Klausel c_i nicht wahr werden.

Dies bedeutet, dass jede Matchinglösung einer Gleichung $s_{i,1} \approx t_{i,\epsilon}$ einer Wahrheitsbelegung entspricht, die die Klausel c_i wahr werden läßt. Umgekehrt kann eine solche Wahrheitsbelegung in eine Substitution für $\{X_j, Y_j | 1 \leq j \leq n\}$ „übersetzt" werden, so dass diese Substitution das Matchingproblem $s_{i,1} \approx t_{i,\epsilon}$ löst. Wird also das geschichtete Matchingproblem P_{3SAT} gelöst, so löst jede gefundene Substitution - als Wahrheitswerte interpretiert -, auch das Ursprungsproblem I_{3SAT}.

Wir haben noch zu zeigen, dass die Reduktion von I_{3SAT} auf P_{3SAT} in polynomieller Zeit möglich ist. Betrachtet man die Konstruktion der einzelnen Kontextgleichungen von P_{3SAT}, so stellt man fest, dass die Tiefe jeder Gleichung proportional zur Anzahl der Variablen n ist. Die Anzahl der Gleichungen ist proportional zur Anzahl der Klauseln m. Weiter stellen wir fest, dass jede der rechten Seiten in höchstens drei Schichten (den aktiven Schichten) verzweigt, da es höchstens drei Variablen pro Klausel c_i gibt. In den passiven Schichten von $t_{i,\epsilon}$, die die in Klausel c_i nicht vorkommenden Variablen repräsentieren, gibt es keine Verzweigung. Damit ergibt sich ein nicht-exponentieller Vergrößerungsfaktor von 2^3.

Insgesamt zeigt sich, dass das so entstehende Kontextmatchingproblem eine quadratische Vergrößerung der $3SAT$-Instanz ist und I_{3SAT} somit polynomial-Zeit-reduzierbar auf P_{3SAT} ist.

Damit wurde gezeigt, dass geschichtetes Kontextmatching $\mathcal{NP}$-vollständig ist.

3.3.3. Andere Komplexitätsergebnisse.

3.3.3.1. *Simultanes geschichtetes monadisches Kontextmatching.*

Definition 3.3.9. Ein Kontextmatchingproblem heißt *simultanes geschichtetes monadisches Kontextmatchingproblem*, wenn es geschichtet ist und alle Funktionssymbole eine Stelligkeit von höchstens 1 haben.

In [**Schm2003**] wird gezeigt, dass simultanes geschichtetes monadisches Kontextmatching in $\mathcal{P}$ ist. Dazu wird ein entsprechender Transformationsalgorithmus verwendet, der insbesondere die Tatsache benutzt, dass in dem gegebenen eingeschränkten Problem keine Verzweigungen in den Termen möglich sind, da die Funktionssymbole höchstens die Stelligkeit 1 besitzen.

Dieser Fall unterscheidet sich schon dadurch von anderen Kontextmatchingproblemen, dass ein aus mehreren Gleichungen bestehendes Kontextmatchingproblem in Ermangelung eines zweistelligen Funktionssymbols nicht auf ein Kontextmatchingproblem mit nur einer Gleichung zurückgeführt werden kann (vgl. Proposition 3.2.6).

Das heißt, dass für eine Gleichung $s_1 X_1 \ldots s_n X_n \approx t$ (wobei die s_i und t beliebige Terme ohne Kontextvariablen sind) und eine Lösung σ stets gilt $|(X_1 X_n)\,\sigma| = |t| - |s_1 \ldots s_n|$. Die Zahl $|t| - |s_1 \ldots s_n|$ wird *Substitutionslänge* genannt. Falls diese negativ ist, ist die entsprechende Gleichung unlösbar. Gleiches gilt, wenn es zwei Gleichungen im Kontextmatchingproblem mit derselben Menge Kontextvariablen gibt, die Substitutionslänge aber differiert.

Der in [**Schm2003**] vorgestellte Transformationsalgorithmus löst simultanes geschichtetes monadisches Kontextmatching in $O\left(n^3\right)$ Zeit und $O\left(n^2\right)$ Platz.

3.3.3.2. *Kontextmatching mit Comon's Restriktion.*

Definition 3.3.10. Ein Kontextmatchingproblem gehorcht Comon's Restriktion, falls jede Kontextvariable auf denselben Term angewandt wird.

Man kann Comon's Restriktion als einen Spezialfall von geschichteten Kontextmatchingproblemen ansehen. Zusätzlich zu der Bedingung, dass der Präfix aller Variablen gleich sein muss (wie bei geschicheten Problemen), wird gefordert, dass alle Terme, die in Kontexten eingesetzt werden, identisch sind.

Beispiel. Folgendes Kontextmatchingproblem ist zwar geschichtet, gehorcht aber nicht Comon's Restriktion, da die in X und Y eingesetzten Terme nicht identisch sind:

$$\{X\,[f\,(Y\,[c])] \approx t_1, X\,[g\,(Y\,[h])] \approx t_2\}$$

Dagegen fällt das Problem

$$\{X\,[f\,(Y\,[c])] \approx t_1, Z\,[g\,(Y\,[c])] \approx t_2\}$$

in die Klasse der Kontextmatchingprobleme mit Comon's Restriktion.

Repräsentiert man ein Kontextmatchingproblem mit Comon's Restriktion P durch einen Graphen mit vollständiger gemeinsamer Nutzung von Subtermen, so taucht jede Kontextvariable lediglich einmal im Problem auf. Daher kann der Algorithmus für lineares Kontextmatching verwendet werden, was zur gleichen Zeit- und Platzkomplexität führt.

Kontextmatching mit Comon's Restriktion ist daher in $\mathcal{P}$.

3.3.3.3. *2-Duplikat-Kontextmatching.*

Definition 3.3.11. Ein Kontextmatchingproblem P ist in der Klasse der *2-Duplikat-Kontextmatchingprobleme*, wenn jede individuelle und jede Kontextvariable höchstens zweifach in P vorkommen.

Theorem 3.3.12. *2-Duplikat-Kontextmatching ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zitieren den Beweis aus [**Schm2003**], der das als $\mathcal{NP}$-vollständig bekannte Problem positives $1-in-3SAT$ auf 2-Duplikat-Kontextmatching zurückführt.

DEFINITION. $1-in-3SAT$ ist das Erfüllungsproblem für boolesche Formeln in konjunktiver Normalform, bei der jede Klausel genau drei Literale enthält. Gesucht wird eine Besetzung der Variablen dergestalt, dass in jeder Klausel c_i des Problems genau ein Literal wahr ist, so dass die Konjunktion aller Klauseln $c_1 \wedge \ldots \wedge c_n$ wahr ist.

Wir konstruieren ein 2-Duplikat-Kontextmatchingproblem P_{3SAT}, dessen Lösung eine Lösung für das $1-in-3SAT$-Problem I_{3SAT} ist. Dabei wird der Wahrheitswert True mit $f(a)$ und der Wahrheitswert False mit a repräsentiert. Wir werden jede *logische* Variable x_j in I_{3SAT} durch n_j *individuelle* Variablen in P_{3SAT} repräsentieren. Dabei ist n_j die Anzahl der Vorkommen der Variablen x_j in I_{3SAT}.

BEISPIEL. Sei I_{3SAT} wie folgt:

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_3 \vee \neg x_4)$$

Dann wird die logische Variable x_1 in P_{3SAT} durch die individuellen Variablen y_{11}, y_{12} und y_{13} repräsentiert.

Die Konstruktion von P_{3SAT} funktioniert wie folgt:

Für jede logische Variable x_j wird eine Kontextgleichung mit all ihren zugehörigen individuellen Variablen $y_{j1}, \ldots, y_{jn_j}$ hinzugefügt:

$$X_j \left[g\left(y_{j1}, \ldots, y_{jn_j}\right) \right] \approx h\left(g\left(a, \ldots, a\right), g\left(f\left(a\right), \ldots, f\left(a\right)\right)\right) \tag{3.3.3}$$

Mit Hilfe dieser Gleichung wird sichergestellt, dass alle individuellen Variablen $y_{j1}, \ldots, y_{jn_j}$ stets denselben Wahrheitswert besitzen, d. h. alle individuellen Variablen gleichzeitig True oder False sind, da sie zusammen die logische Variable x_j repräsentieren.

Für jede Klausel $c_i = (x_{i_1} \vee x_{i_2} \vee x_{i_3})$ fügen wir eine weitere Kontextgleichung hinzu:

$$\begin{aligned} &Y_i \left[g\left(y_{i_1 k_1}, y_{i_2 k_2}, y_{i_3 k_3}\right) \right] \approx \\ &\quad h\left(g\left(f\left(a\right), a, a\right), g\left(a, f\left(a\right), a\right), g\left(a, a, f\left(a\right)\right)\right) \end{aligned} \tag{3.3.4}$$

Dabei wird jedes der in c_i vorkommenden Literale genau einmal in jeder Verzweigung auf True gesetzt.

Damit wird jede individuelle Variable y_{ij} genau zweimal in P_{3SAT} verwendet - einmal in einer Gleichung vom Typ 3.3.3 und einmal in einer Gleichung vom Typ 3.3.4. P_{3SAT} ist demnach ein 2-Duplikat-Kontextmatchingproblem.

Insgesamt wird mit P_{3SAT} auch I_{3SAT} gelöst. Die Reduktion von I_{3SAT} auf P_{3SAT} erfolgt offensichtlich in polynomieller Zeit.

□

3.4. Algorithmen

Wir kommen nun zu Algorithmen, die Kontextmatching bzw. bestimmte Teilprobleme von Kontextmatching lösen können. In diesem Kapitel werden wir diese Algorithmen zunächst theoretisch vorstellen. Die Implementierungen in Haskell, die im Rahmen dieser Diplomarbeit entstanden sind, sind Gegenstand von Kapitel 4.

3.4.1. Lineares Kontextmatching. Wie bereits in Kapitel 3.3.1 angedeutet, kann lineares Kontextmatching durch dynamisches Programmieren gelöst werden. Der hier vorgestellte Algorithmus zeigt auch, dass lineares Kontextmatching in $\mathcal{P}$ ist.

Um ein lineares Kontextmatchingproblem $P = \{s \approx t\}$ zu lösen, stellen wir eine Tabelle auf, deren Zeilen mit sämtlichen Subtermen von s und deren Spalten mit sämtlichen Subtermen von t markiert sind. Wir lösen P, indem wir die Tabelle rekursiv von unten beginnend füllen, d. h. zunächst diejenigen Zellen bearbeiten, die sich auf die Blätter der Terme s und t beziehen, um dann während der Auflösung der Rekursion höherliegende Subterme zu betrachten.

Im Ansatz aus [**Schm2003**] wird durch den Algorithmus lediglich festgestellt, ob P lösbar ist. Wir sind jedoch auch an den Lösungen von P interessiert, falls es welche gibt. Daher halten wir während der Ausführung des Algorithmus - analog zu [**Stu2003**] - die Lösung des aktuellen Subproblems in einer Menge von logischen Formeln fest. Diese werden als *semigelöste Formen* bezeichnet.

Definition 3.4.1. Sei $t \in T(\Sigma, V, \mathcal{C})$ ein Kontextterm. Seien weiterhin $x \in V$ und $X \in \mathcal{C}$. Eine logische Formel ist in *semigelöster Form*, falls sie lediglich Gleichungen der Formen $x \approx t$ oder $X \approx t\,[\Box]$ sowie die logischen Operatoren $\wedge$, $\vee$, $\bot$ und $\top$ enthält.

Wir nennen unseren Algorithmus LCM und schreiben für die semigelöste Form der Problems $\{s \approx t\}$ einfach $LCM(\{s \approx t\})$. LCM ist in Algorithmus 3.4.1 dargestellt.

Proposition. *Der Algorithmus $LCM(\{s \approx t\})$ löst lineares Kontextmatching für das Problem $P = \{s \approx t\}$.*

Beweis. Durch Induktion über die Struktur der Terme und die Beobachtung, dass sämtliche Fälle abgedeckt sind. □

Sei n nun die Größe des Problems P. Wir nehmen an, dass gleiche Subterme gemeinsam genutzt werden (sharing). Es gibt $O(n^2)$ Tabelleneinträge. Jede Tabellenzelle benötigt dann höchstens $O(n)$ Platz. Insgesamt ergibt dies $O(n^3)$ als Speicherkomplexität.

Bei der Berechnung benötigt LCM für jede Tabellenzelle höchstens $O(n)$ Zeit, so dass die Zeitkomplexität ebenfalls $O(n^3)$ beträgt.

Algorithmus 3.4.1 Der Algorithmus $LCM(\{s \approx t\})$

(1) s ist eine individuelle Variable, d. h. $s = x$. Falls t kein Kontext ist, d. h. t enthält kein Loch $\Box$, so setze $LCM(\{s \approx t\}) := x \approx t$. Ansonsten setze $LCM\{s \approx t\} := \bot$.

(2) s ist das Loch, d. h. $s = \Box$. Falls $t = \Box$, so setze $LCM(s \approx t) := \top$, sonst $LCM(s \approx t) := \bot$.

(3) s und t sind Applikationen, d. h. $s = f(s_1, \ldots, s_n)$ und $t = g(t_1, \ldots, t_m)$. Falls $f = g$ und $n = m$, so setze $LCM(s \approx t) := LCM(s_1 \approx t_1) \wedge \ldots \wedge LCM(s_n \approx t_n)$, sonst $LCM(s \approx t) := \bot$.

(4) s ist eine Instanziierung einer Kontextvariablen, d. h. $s = X[s']$. Dann wählen wir sämtliche Kontexte C und alle Terme t', so dass gilt $t = C[t']$ und schreiben $LCM(s \approx t)$ wie folgt:

$$LCM(s \approx t) := \bigvee_{\{C \in \mathcal{C}, t' \in T(X,V,\mathcal{C}) | t = C[t']\}} X \approx C[\Box] \wedge LCM\left(s' \approx t'\right)$$

3.4.2. Allgemeines Kontextmatching. Obwohl allgemeines Kontextmatching als $\mathcal{NP}$-vollständig bekannt ist [**Schm1998**], ist es interessant, allgemeine Algorithmen für dieses Problem zu entwickeln und zu untersuchen, da eventuell viele Fälle der Praxis durch einen solchen Algorithmus zu lösen sind. [**Schm2003**] stellt einen solchen Algorithmus in Form eines Transformationsalgorithmus vor und gibt Optimierungsmöglichkeiten für einige Sonderfälle an.

Wir betrachten zunächst die Transformationsregeln, die in Tabelle 3.4.1 angegeben sind.

Die Regeln operieren auf einer Konjunktion von Gleichungen zwischen Multikontexttermen. Die rechte Seite der Gleichungen ist dabei stets geschlossen.

Alle Regeln - außer Variablen-Split - sind sogenannte don't care-nichtdeterministische Regeln, d. h. sie können in beliebiger Reihenfolge erschöpfend auf das Kontextmatchingproblem angewandt werden.

Die Regeln Kontextelimination und Variablen-Split unterscheiden sich nur in einem Punkt: Während bei der Kontextelimination die Kontextvariable X bereits fest an den Kontext C gebunden ist und damit auch t' festgelegt ist, ist diese Bindung beim Variablen-Split noch nicht gegeben. Ein passender Kontext C muss daher erraten werden.

Daraus resultiert der don't know-Nichtdeterminismus von Variablen-Split: Die Wahl des Kontexts C kann über die Lösbarkeit des Problems entscheiden und zu verschiedenen Lösungen führen. Daher muss jede Anwendung dieser Regel zurückverfolgbar sein (Backtracking).

Definition 3.4.2. Ein Kontextmatchingproblem

$$P = \{x_1 \approx t_1 \wedge \ldots \wedge x_n \approx t_n \wedge X_1 \approx C_1[\Box] \wedge X_m \approx C_m[\Box]\}$$

heißt *gelöst*, wenn alle x_i $(1 \leq i \leq n)$ sowie alle X_j $(1 \leq j \leq m)$ paarweise verschieden sind und weder ein x_i noch ein X_j in einem t_i oder C_j vorkommt. In diesem Fall erhält man die kanonische Substitution

$$\sigma_P := \{\langle t_1/x_1 \rangle, \ldots, \langle t_n/x_n \rangle, \langle C_1/X_1 \rangle, \ldots, \langle C_m/X_m \rangle\}$$

DEKOMPOSITION	$\left\{f\left(\vec{s}\right) \approx f\left(\vec{t}\right)\right\} \uplus S$, falls $n := \left\lvert\vec{t}\right\rvert = \left\lvert\vec{s}\right\rvert$	$\Rightarrow$	$\{s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n\} \cup S$
KOLLISION	$\left\{f\left(\vec{t}\right) \approx g\left(\vec{s}\right)\right\} \uplus S$	$\Rightarrow$	$\bot$, falls $f \neq g$ oder $\left\lvert\vec{t}\right\rvert \neq \left\lvert\vec{s}\right\rvert$
LOCHDEKOMPOSITION	$\{\Box \approx C[\Box]\} \uplus S$	$\Rightarrow$	S, falls $C[\Box] = \Box$, $\bot$ sonst.
MULTIKONTEXT-KOLLISION	$\{C[\Box^n] \approx D[\Box^m]\} \uplus S$, falls $n \neq m$	$\Rightarrow$	$\bot$
TERMVERSCHMELZUNG	$\{x \approx s \wedge x \approx t\} \uplus S$	$\Rightarrow$	$\{x \approx s\} \cup S$, falls $s = t$, $\bot$ sonst.
KONTEXTVERSCHMELZUNG	$\{X \approx C[\Box] \wedge X \approx D[\Box]\} \uplus S$	$\Rightarrow$	$\{X \approx C[\Box]\} \cup S$, falls $C[\Box] = D[\Box]$, $\bot$ sonst.
KONTEXTELIMINATION	$\{X \approx C[\Box] \wedge X[s] \approx t\} \uplus S$	$\Rightarrow$	$\{X \approx C[\Box] \wedge s \approx t'\} \cup S$, falls $s \neq \Box$ und $t = C[t']$, $\bot$ sonst.
VARIABLEN-SPLIT	$\{X[s] \approx t\} \uplus S$, falls $s \neq \Box$ und $t = C[t']$	$\Rightarrow$	$\{X \approx C[\Box] \wedge s \approx t'\} \cup S$

TABELLE 3.4.1. Transformationsalgorithmus für Kontextmatching

3.4.2.1. *Korrektheit des Algorithmus.*

THEOREM 3.4.3. *Der Transformationsalgorithmus, definiert durch die Regeln in Tabelle 3.4.1, der die don't care-nichtdeterministischen Regeln erschöpfend anwendet und die Anwendungen der Regel* VARIABLEN-SPLIT *zurückverfolgt, findet alle Lösungen von Kontextmatchingproblemen.*

Wir zeigen die Korrektheit des Theorems durch einige Lemmata.

LEMMA 3.4.4. *Die don't care-nichtdeterministischen Regeln* DEKOMPOSITION, KOLLISION, LOCHDEKOMPOSITION, MULTIKONTEXT-KOLLISION, TERMVERSCHMELZUNG, KONTEXTVERSCHMELZUNG *und* KONTEXTELIMINATION *verändern die Lösungsmenge nicht.*

BEWEIS. Durch Inspektion der Regeln. □

Wir müssen nun noch zeigen, dass die don't know-nichtdeterministische Regel VARIABLEN-SPLIT das Kontextmatchingproblem dergestalt transformiert, dass seine Lösungen weiterhin anwendbar bleiben.

LEMMA 3.4.5. *Sei P ein Kontextmatchingproblem und seien $P_1, \ldots, P_n$ Kontextmatchingprobleme, die durch Anwendung der Regel* VARIABLEN-SPLIT *auf alle passenden Gleichungen aus P hervorgegangen sind. Dann ist eine Substitution eine Lösung von P genau dann wenn sie auch eine Lösung für einige P_i $(1 \leq i \leq n)$ ist.*

Beweis. Sei σ eine Substitution, die P löst. Wir betrachten eine auf die Regel Variablen-Split passende Gleichung aus P. Diese hat die Form $X[s] \approx t$. Die Kontextvariable X an der Wurzel der linken Seite muss mit Hilfe von σ durch einen Kontext C ersetzt werden. Weil σ eine Lösung von P ist, gilt auch $X[s]\sigma = t$. In diesem Falle ist C festgelegt und da für jede Wahl von C eine passende Anwendung von Variablen-Split existiert, gilt auch $X\sigma = C[\Box]$ und $s\sigma = t$.

Ist andererseits σ eine Substitution, die Lösung eines P_i ist, so muss gelten $\langle C[\Box]/X\rangle \in \sigma$ und wegen $t = C[t']$ sowie $s \approx t'$ auch $X[s]\sigma = t = C[t']$. Damit ist σ auch Lösung von P. □

Wir zeigen nun die Vollständigkeit der Regeln.

Lemma 3.4.6. *Falls ein Kontextmatchingproblem P nicht gelöst ist (vgl. Definition 3.4.2), dann ist eine der Transformationsregeln aus Tabelle 3.4.1 anwendbar.*

Beweis. Falls es in P Gleichungen mit einer unterschiedlichen Anzahl von Löchern auf der linken und rechten Seite gibt, so wird die Regel Multikontext-Kollision angewandt. Nach einer endlichen Anzahl von Schritten haben alle Gleichungen die gleiche Anzahl Löcher.

Falls die linke Seite einer Gleichung ein Loch $\Box$ ist, wenden wir Lochdekomposition an. Falls die Wurzel der linken Seite ein Funktionssymbol f ist, wenden wir Dekomposition oder Kollision an.

In jedem Fall sind alle Gleichungen nach erschöpfender Anwendung dieser drei Regeln von der Form $x \approx t$ oder $X[s] \approx t$.

Auf eine Gleichung der Form $X[s] \approx t$ wird Variablen-Split angewandt. Danach können die bereits genannten Regeln wiederum angewendet werden.

Nach vollständiger Anwendung aller Regeln haben sämtliche Gleichungen die Form $x \approx t$ oder $X \approx C[\Box]$. Falls nun eine Variable x oder X mehrfach in den Gleichungen auftaucht, so kann entweder die Regel Kontextverschmelzung oder Termverschmelzung verwendet werden.

Falls keine der Regeln mehr anwendbar ist, so ist das Kontextmatchingproblem gelöst. □

Abschließend fehlt noch, die Terminierung des Transformationsalgorithmus zu zeigen.

Lemma 3.4.7. *Der Transformationsalgorithmus, der durch die Regeln in Tabelle 3.4.1 definiert wird, terminiert in $O(n)$. Dabei ist n ein Maß berechnet aus dem Kontextmatchingproblem.*

Beweis. Wir bilden die Summe aus der Anzahl von Funktionssymbolen und Variablen im Kontextmatchingproblem sowie der Anzahl der Argumente ungleich dem Loch $\Box$ von Funktionssymbolen und Kontextvariablen. Wir verwenden diese Summe als Maß.

Die don't care-Transformationsregeln dekrementieren die Anzahl der Funktionssymbole und Variablen und erhöhen die Anzahl der Argumente nicht.

Die Regel VARIABLEN-SPLIT (don't know-nichtdeterministisch) lässt die Anzahl der Funktionssymbole und Variablen unverändert, dekrementiert aber die Anzahl der Argumente. □

3.4.2.2. *Komplexität.* Die Regel VARIABLEN-SPLIT führt durch ihren don't know-Nichtdeterminismus zu einem exponentiellen Verhalten des Algorithmus. Darüber hinaus können exponentiell viele Lösungen entstehen.

Ist man nur an der Lösbarkeit des Kontextmatchingproblems interessiert, nicht aber an den Lösungen selbst, so können Gleichungen der Form $x \approx t$ und $X \approx C\,[\Box]$ aus dem Kontextmatchingproblem entfernt werden, sofern die Variablen x bzw. X nur einmal im Problem auftauchen.

3.4.3. Korrespondierende Positionen. Im Falle von Multikontexten (Termen mit mehreren Löchern) führt die Anwendung der Regel Dekomposition in Kombination mit der Regel VARIABLEN-SPLIT letztendlich dazu, dass Löcher, die in parallelen Subtermen auftauchen, getrennt werden. Löcher auf der linken Seite einer Gleichung stehen jedoch in einer eins-zu-eins Korrespondenz zu den Löchern der rechten Seite, da man sie als implizit gebundene Variablen ansehen kann.

Damit sind Löcher ein Beispiel für (trivial) korrespondierende Positionen in einer Termgleichung, da sie Positionen im Matchingproblem gleichermaßen fixieren. Bildhaft gesprochen können mit korrespondierenden Positionen die Termgraphen beider Seiten einer (Multi-)Kontextgleichung aneinander ausgerichtet werden, so dass man Split-Punkte erhält, an denen man die Kontextgleichung in eine Konjunktion mehrerer Gleichung aufteilen kann. Der don't know-Nichtdeterminismus von VARIABLEN-SPLIT wird dadurch vermieden.

Zunächst jedoch definieren wir den Begriff korrespondierender Positionen formal.

DEFINITION 3.4.8. Sei P ein Kontextmatchingproblem, $s \approx t$ eine Gleichung aus P. Sei weiter p_s eine Position in s und p_t eine Position in t.

Die Positionen p_s und p_t *korrespondieren für eine Substitution* σ, falls $s\,[\Box]_{p_s}\,\sigma = t\,[\Box]_{p_t}$ und $s|_{p_s}\sigma = t|_{p_t}$. Sie sind *korrespondierende Positionen in* P, falls p_s und p_t für alle Lösungen σ von P korrespondieren.

Wir nennen die Wurzeln der linken und rechten Seite einer Kontextgleichung sowie die Positionen von Löchern *trivial korrespondierend*, alle anderen Positionen dementsprechend *nichttrivial korrespondierend*.

Im Falle von unlösbaren Problemen korrespondieren alle Positionen miteinander: Es existiert keine lösende Substition σ, also korrespondieren alle Positionen für alle Lösungen (nämlich keine) von P.

Sollten wir korrespondierende Positionen p_s und p_t kennen, so können wir diese Information nutzen, um die Termgleichung an den korrespondierenden Positionen aufzuteilen:

SPLIT	$\{s\,[s|_{p_s}] \approx t\,[t|_{p_t}]\} \uplus S$, falls p_s und p_t nicht-trivial korrespondieren.	$\Rightarrow$	$\left\{s\,[\Box]_{p_s} \approx t\,[\Box]_{p_t} \wedge s|_{p_s} \approx t|_{p_t}\right\} \cup S$

Trivial korrespondierende Positionen führen in der Regel SPLIT zurück zum Ausgangsproblem und werden daher ausgeschlossen, um die Terminierung weiterhin garantieren zu können.

Die Regel SPLIT kann bei Kenntnis korrespondierender Positionen somit ohne Backtracking erschöpfend in beliebiger Reihenfolge auf das Problem angewendet werden und vermeidet damit in solchen Fällen das exponentielle Verhalten von VARIABLEN-SPLIT.

PROPOSITION 3.4.9. *Die Regel* SPLIT *verändert die Lösungsmenge nicht.*

BEWEIS. Ergibt sich unmittelbar aus der Definition des Begriffs korrespondierende Position. □

PROPOSITION 3.4.10. *Der Transformationsalgorithmus, der durch die Regeln in Tabelle 3.4.1 ergänzt um die Regel* SPLIT *definiert wird, terminiert in $O(n)$. Dabei ist n ein Maß berechnet aus dem Kontextmatchingproblem.*

BEWEIS. Analog zu Lemma 3.4.7. Die Regel SPLIT dekrementiert die Anzahl der Argumente von Funktionssymbolen und Variablen ungleich dem Loch, verändert aber die Anzahl der Funktionssymbole und Variablen nicht. □

Wir nehmen nun an, wir könnten korrespondierende Positionen berechnen. Dies führt zu folgender Komplexitätsbetrachtung:

THEOREM 3.4.11. *Sei C eine Klasse von Kontextmatchingproblemen, die geschlossen ist unter der Anwendung der Transformationsregeln. Falls ein Orakel existiert, das für Instanzen $P \in C$ in $O\left(n^k\right)$ stets korrespondierende Positionen findet, dann löst der Transformationsalgorithmus Matchingprobleme in C in $O\left(n^{k+1}\right)$ und C liegt in $\mathcal{P}$.*

Es wird in diesem Theorem gefordert, dass das Orakel stets korrespondierende Positionen findet. Solche Positionen existieren jedoch nicht immer.

3.4.4. Berechnung korrespondierender Positionen. Wir wenden uns nun dem Problem zu, korrespondierende Positionen in einem Kontextmatchingproblem zu finden. [**Schm2003**] stellt dazu drei Methoden vor. Die Erste verwendet Linearisierung des Problems, die Zweite ein diophantisches Gleichungssystem und die Dritte berechnet korrespondierende Positionen aus bereits bekannten korrespondierenden Positionen, insbesondere aus Löchern.

3.4.4.1. *Linearisierung.* Ausgangspunkt dieser Methode ist die Erkenntnis, dass man eine beliebige Kontextgleichung $s \approx t$ durch Linearisierung der Variablen auf Unlösbarkeit testen kann.

Dazu transformiert man die Gleichung $s \approx t$ zunächst durch Variablenumbenennung aller individuellen und Kontextvariablen in eine lineare Gleichung $s' \approx t$, so dass jede Variable höchstens einmal vorkommt. Jede Lösung des Ursprungsproblems $s \approx t$ ist auch eine Lösung von $s' \approx t$. Ist jedoch das linearisierte Problem unlösbar, so ist klar, dass auch das Ursprungsproblem unlösbar ist: Sei x eine beliebige Variable, die in s n-fach $(n > 1)$ vorkommt. Dann wurde diese Variable in s' durch n Variablen der Form $x'_1, \ldots, x'_n$ ersetzt. $s' \approx t$ ist unlösbar. D. h. es existiert keine Besetzung für $x'_1, \ldots, x'_n$, so dass $s' \approx t$ lösbar wäre. Insbesondere gilt dies auch für den Fall $x'_1 = \ldots = x'_n$. Dies aber gilt für die Ursprungsgleichung $s \approx t$.

Wir können nun den Algorithmus $LCM\left(\{s \approx t\}\right)$ aus Kapitel 3.4.1 verwenden, um korrespondierende Positionen zu entdecken. Da es sich bei LCM um dynamisches Programmieren handelt, können wir die berechnete Lösungstabelle betrachten. Tragen wir die linke Seite einer Kontextmatchinggleichung in Zeilen und die rechte Seite in Spalten ab, so betrachten wir solche Zeilen, in denen genau eine Tabellenzelle mit einem Wert gleich $\top$ existiert.

Uns interessiert in diesem Falle also nicht die Lösung des linearisierten Problems $s' \approx t$, sondern nur die Lösbarkeit. Daher vereinfachen wir LCM gemäß Algorithmus 3.4.2 so, dass nur noch die Lösbarkeit protokolliert wird, nicht jedoch die Lösungen. Damit sinkt der Platzbedarf des Algorithmus von $O\left(n^3\right)$ auf $O\left(n^2\right)$. Wir nennen diesen Algorithmus *LLCM*.

Wir betrachten nur Zeilen mit genau einem $\top$ und keine Spalten mit dieser Eigenschaft, da letztere zu einem don't know-nichtdeterministischen Verhalten führen. Beispielsweise sind Zeilen, die eine Variable repräsentieren, in jeder Spalte mit einem $\top$ versehen, da eine Variable jeden Term matcht. Betrachtet man nun zwei Spalten, in denen jeweils nur ein $\top$ in einer Zelle mit derselben Variablen steht, so bedeutet dies, dass die Variable entweder die eine oder die andere Spalte matcht.

Beispiel 3.4.12. Ein Beispielproblem hierfür ist $\{C\left[f\left(x\right)\right] \approx h\left(f\left(a\right), f\left(b\right)\right)\}$. In diesem Fall sieht die Ergebnistabelle wie folgt aus:

	$h\left(f\left(a\right), f\left(b\right)\right)$	$f\left(a\right)$	a	$f\left(b\right)$	b
$C\left[f\left(x\right)\right]$	$\top$	$\top$	$\bot$	$\top$	$\bot$
$f\left(x\right)$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
x	$\top$	$\top$	$\top$	$\top$	$\top$

x wird durch a oder b gematcht. Die Lösung des Problems lautet:

$$\left\{\left\{\left\langle h\left(\Box, f\left(b\right)\right)/C\right\rangle, \left\langle a/x\right\rangle\right\}, \left\{\left\langle h\left(f\left(a\right), \Box\right)/C\right\rangle, \left\langle b/x\right\rangle\right\}\right\}$$

Lässt man Spalten mit nur einem $\top$ als korrespondierende Position zu, so werden $\langle a/x\rangle$ bzw. $\langle b/x\rangle$ fälschlicherweise als korrespondierende Positionen erkannt. Im Fall $\langle a/x\rangle$ lautet die Argumentation: Hier müssten gemäß Definition 3.4.8 $C\left[f\left(\Box\right)\right]\sigma = h\left(f\left(\Box\right)\right)$ und $x\sigma = a$ für alle Lösungssubstitutionen

Algorithmus 3.4.2 Der Lösungs-Algorithmus $LLCM(\{s \approx t\})$

(1) s ist eine individuelle Variable, d. h. $s = x$. Setze $LLCM(\{s \approx t\}) := \top$.
(2) s ist das Loch, d. h. $s = \square$. Falls $t = \square$, so setze $LLCM(s \approx t) := \top$, sonst $LLCM(s \approx t) := \bot$.
(3) s und t sind Applikationen, d. h. $s = f(s_1, \ldots, s_n)$ und $t = g(t_1, \ldots, t_m)$. Falls $f = g$ und $n = m$, so setze $LLCM(s \approx t) := LLCM(s_1 \approx t_1) \wedge \ldots \wedge LLCM(s_n \approx t_n)$, sonst $LLCM(s \approx t) := \bot$.
(4) s ist eine Instanziierung einer Kontextvariablen, d. h. $s = X[s']$. Dann wählen wir sämtliche Kontexte C und alle Terme t', so dass gilt $t = C[t']$ und schreiben $LLCM(s \approx t)$ wie folgt:

$$LLCM(s \approx t) := \bigvee_{\{C \in \mathcal{C}, t' \in T(X,V,\mathcal{C}) | t = C[t']\}} LLCM(s' \approx t')$$

σ gelten. Im Fall der ersten Substitution gilt dies auch, nicht aber bei der zweiten, da $h(f(a), \square) \neq h(f(\square), f(b))$ und $b \neq a$.

Eine weitere Änderung betrifft die Regel für individuelle Variablen (Regel 1). In Algorithmus 3.4.1 enthält diese Regel eine zusätzliche Überprüfung auf Kontexte. Führt man diese Überprüfung ein, so werden korrespondierende Positionen falsch erkannt, wie das Beispiel $\{f(x, \square) \approx f(\square, b)\}$ zeigt. Wir geben die Lösungstabelle an für den Fall, dass die Kontextüberprüfung enthalten ist:

	$f(\square, b)$	$\square$	b
$f(x, \square)$	$\bot$	$\bot$	$\bot$
x	$\bot$	$\bot$	$\top$
$\square$	$\bot$	$\top$	$\bot$

In diesem Fall matcht x nur die Konstante b auf der rechten Seite, was dazu führt, dass x und b als korrespondierende Positionen erkannt werden, was aber offensichtlich falsch ist. Lassen wir jedoch das Matching von Grundkontexten durch individuelle Variablen wie in Algorithmus 3.4.2 zu, so sieht die Lösungstabelle wie folgt aus und x besitzt keine korrespondierende Position:

	$f(\square, b)$	$\square$	b
$f(x, \square)$	$\bot$	$\bot$	$\bot$
x	$\top$	$\top$	$\top$
$\square$	$\bot$	$\top$	$\bot$

Die Linearisierung kann natürlich den Lösungsraum vergrößern, da durch sie identische Variablen in unabhängige Variablen aufgespalten werden. Insofern kann die Anwendung von $LLCM$ zur Berechnung korrespondierender Positionen nur ein notwendiges, nicht aber ein hinreichendes Kriterium sein und die durch diesen Algorithmus berechneten korrespondierenden Positionen können auch falsch sein.

Bemerkung. Man kann die Linearisierung auch intuitiv wie folgt beschreiben. Gegeben sei ein Kontextmatchingproblem P:

$$P := \{X\left[f\left(Y\left[a\right], Y\left[Z\left[g\left(x, b\right)\right]\right], Z\left[f\left(a, x, c\right)\right]\right) \approx t\right]\}$$

Durch Linearisierung entsteht P':

$$P' := \left\{X\left[f\left(Y_1'\left[a\right], Y_2'\left[Z_1'\left[g\left(x_1', b\right)\right]\right], Z_2'\left[f\left(a, x_2', c\right)\right]\right)\right] \approx t\right\}$$

Intuitiv würde man nun in t nach einem Subterm, der f matcht, suchen, indem man nach einem Knoten f im Termbaum sucht, in dessen ersten Argument ein a, in dessen zweiten Argument ein g und in dessen dritten Argument wiederum ein f auftritt.

Die Vorgehensweise, in der Ergebnistabelle von $LLCM$ nach Zeilen mit nur einem $\top$ zu suchen, entspricht genau diesem Vorgehen. Betrachtet man beispielsweise eine Zeile, in der nur ein $\top$ steht, und sei diese Zelle mit den Termen s' und t' markiert, so bedeutet dies, dass der Subterm s' von s im Term t nur als t' auftaucht, unabhängig davon, wie individuelle oder Kontextvariablen besetzt werden.

3.4.4.2. *Lineare Gleichungssysteme.* Wir betrachten eine Möglichkeit, korrespondierende Positionen zu finden, die gewissermaßen orthogonal zu der Berechnung mit Hilfe der Linearisierung liegt: Galt es dort noch, die Positionen von Funktionssymbolen zu betrachten, so beschäftigt sich dieser Ansatz mit ihrer Anzahl.

Definition. Sei t ein Kontextterm und f ein Funktionssymbol. Dann bezeichnet $\mathtt{occ}\left(f, t\right)$ die Anzahl, mit der f in t vorkommt.

Sei $P := \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ ein Kontextmatchingproblem, σ eine (unbekannte) Lösung von P und x_1 eine Variable, die in P vorkommt. Dann führen wir eine neue Variable $x_{1,f}$ ein, die die Anzahl der Vorkommen von f in der Substitution $\sigma\left(x_1\right)$ bezeichnen soll. Ist σ bestimmt, so gilt $x_{1,f} = \mathtt{occ}\left(f, x_1\sigma\right)$.

Seien $x_1, \ldots, x_m$ die Variablen in $P \in T\left(\Sigma, V, \mathcal{C}\right)$. Sei ϕ die Anzahl der in P vorkommenden verschiedenen Funktionssymbole. Wir betrachten nun die i-te Gleichung $s_i \approx t_i$ aus P und definieren die folgende Gleichung in den natürlichen Zahlen:

$$EQ\left(f, i\right) \equiv \mathtt{occ}\left(f, s_i\right) + \mathtt{occ}\left(x_1, s_i\right) \cdot x_{1,f} + \ldots + \mathtt{occ}\left(x_m, s_i\right) \cdot x_{m,f} = \mathtt{occ}\left(f, t_i\right)$$

Das Gleichungssystem, das für jedes in P vorkommende Funktionssymbol $f \in \Sigma$ über alle Gleichungen $s_i \approx t_i$ $(1 \leq i \leq n)$ erstellt wird, stellt sicher, dass auf linker und rechter Seite der Kontextgleichungen die gleiche Anzahl von Funktionssymbolen f vorkommt. Wir erhalten also für jedes f ein lineares Gleichungssystem natürlicher Zahlen: $EQ\left(f, i\right)\ \forall 1 \leq i \leq n$.

Diese ϕ Gleichungssysteme liefern uns ein notwendiges Kriterium für korrespondierende Positionen. Wir nehmen an, wir wollen in der Gleichung $s \approx t$ korrespondierende Positionen finden. Dazu wählen wir zunächst eine beliebige fixe Position p_s in s. Wir iterieren nun über alle möglichen Positionen p_t in t, wenden die Regel Split an und erhalten das Problem P'.

Zu P' wird das zugehörige lineare Gleichungssystem aufgestellt und auf Lösbarkeit geprüft. Falls nur eine Position p_t existiert, so dass das Gleichungssystem lösbar ist, sind p_s und p_t korrespondierende Positionen.

Diese Methode ist also geeignet, den Nichtdeterminismus der SPLIT-Regeln zu reduzieren, indem für die Wahl eines Kontextes C, an dessen Stelle ein SPLIT erfolgen soll, ein notwendiges Kriterium bereitgestellt wird.

Entscheidend für diese Methode ist, dass die durch $EQ(f, i)$ definierten Gleichungssysteme entscheidbar sind. Beim definierten Gleichungssystem handelt es sich um ein Diophantisches Gleichungssystem. Beschränkt man den Lösungsraum auf nichtnegative natürliche Zahlen, so ist dieses Problem $\mathcal{NP}$-vollständig. Erlaubt man jedoch beliebige natürliche Zahlen als Lösung, so gibt es Algorithmen, die in polynomieller Zeit die Lösbarkeit entscheiden.

In einem Spezialfall benötigt man die Lösung über lineare Gleichungssysteme nicht, und zwar in dem Fall, dass das Funktionssymbol f in s_i und t_i in gleicher Anzahl vorkommt. Da f nicht in den Substitutionen auftauchen darf und $x_{j,f} = 0$ für alle j $(1 \leq j \leq m)$, ergibt sich eine natürliche Korrespondenz der betroffenen Positionen. Wir nennen die Vorgehensweise analog zu diesem Spezialfall KORRESPONDIERENDE FUNKTIONSSYMBOLE.

3.4.4.3. *Abgeleitete korrespondierende Positionen.* Kennt man bereits korrespondierende Positionen in einer Kontextgleichung $s \approx t$, so kann man diese Information nutzen, um neue korrespondierende Positionen zu finden. Wir kennen insbesondere die trivial korrespondierenden Positionen (die Termwurzel und Löcher).

Ziel dieses Abschnitts ist es, Multikontextgleichungen in Gleichungen zwischen Kontexten mit höchstens einem Loch zu transformieren. Durch die Regeln BODENDEKOMPOSITION und MULTIKONTEXT-DEKOMPOSITION, die in diesem Abschnitt vorgestellt werden, erreicht man in Kombination mit den anderen bereits bekannten Transformationsregeln, dass die Kontextgleichungen nur noch ein Loch besitzen, und dass dieses Loch lediglich in einem Subterm der Form $X\,[]$ auftauchen kann. Durch die beiden zusätzlichen Regeln, die don't care-nichtdeterministisch sind, kann die Anwendung der don't know-nichtdeterministischen Regel VARIABLEN-SPLIT vermindert werden.

BODENDEKOMPOSITION. Betrachtet man eine Multikontextgleichung

$$C\left[\Box^k, s, \Box^l\right] \approx D\left[\Box^k, t, \Box^l\right]$$

so stellt man fest, dass die in den beiden Kontexten enthaltenen Löcher Positionen im Term fixieren. Sind die Wurzeln der Terme s und t nun mit derselben Funktion gekennzeichnet (d. h. $s = E[s_1, \ldots, s_n]$ und $t = E[t_1, \ldots, t_n]$ mit $E[\Box_1, \ldots, \Box_n] = f(\Box_1, \ldots, \Box_n)$), und besitzt diese Funktion in mindestens einem Fall in s und t Löcher in denselben Argumenten, so ist wegen der eins-zu-eins-Korrespondenz von Löchern klar, dass $s \approx t$ gelten muss.

Diese Aussage formalisiert die folgende Proposition:

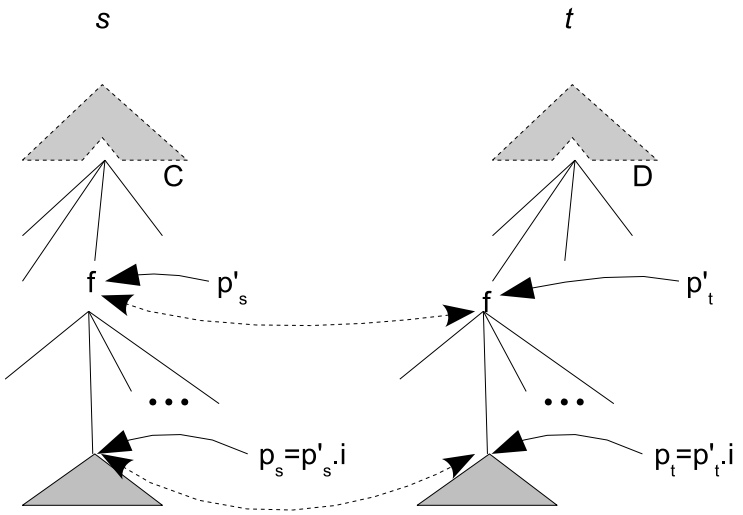


ABBILDUNG 3.4.1. Illustration zu Proposition 3.4.13

PROPOSITION 3.4.13. *Sei $s \approx t$ eine Kontextgleichung. Seien p'_s, p'_t, $p_s := p'_s.i$ und $p_t := p'_t.j$ Positionen in s respektive in t, so dass s ein Funktionssymbol f an Stelle p'_s hat. p_s und p_t sind korrespondierende Positionen in $s \approx t$, genau dann wenn p'_s und p'_t korrespondierende Positionen sind sowie $i = j$.*

BEMERKUNG. Die Forderung, dass p'_s und p'_t korrespondierende Positionen sein müssen, impliziert, dass t das Funktionssymbol f an Stelle p'_t hat.

Die Proposition ist in Abbildung 3.4.1 noch einmal illustriert.

Die Erkenntnis aus dieser Proposition kann man verwenden, um Funktionssymbole, die sich direkt oberhalb korrespondierender Positionen befinden, wie beispielsweise korrespondierende Löcher, zu dekomponieren. In der neuen Regel BODENDEKOMPOSITION werden innerhalb eines Schritts die Regeln SPLIT und DEKOMPOSITION zusammengefaßt:

BODEN-DEKOM-POSITION	$\{C[\Box^k, f(s_1,\ldots,s_n), \Box^l] \approx D[\Box^k, f(t_1,\ldots,t_n), \Box^l]\} \uplus S$ falls $s_j = t_j = \Box_i$ für i, j $(1 \le j \le n)$	$\Rightarrow$	$\{C[\Box^{k+l+1}] \approx D[\Box^{k+l+1}], s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup S$
BODEN-KOLLISION	$\{C[\Box^{k_1}, f(s_1,\ldots,s_m), \Box^{l_1}] \approx D[\Box^{k_2}, g(t_1,\ldots,t_n), \Box^{l_2}]\} \uplus S$ falls $s_{j_1} = \Box_i$ für $1 \le j_1 \le m$, $t_{j_2} = \Box_i$ für $1 \le j_2 \le n$ und $f \neq g$ oder $j_1 \neq j_2$	$\Rightarrow$	$\bot$

Dabei ist entscheidend, dass als Argument unmittelbar unterhalb des Funktionssymbols f mindestens ein Loch vorkommt. Die als Löcher vorhandenen Funktionsargumente müssen auf linker wie rechter Seite in den gleichen Argumenten sitzen (daher die Forderung $s_j = t_j = \Box_i$). Die Forderung aus Proposition 3.4.13, dass p'_s und p'_t korrespondierende Positionen sind, wäre

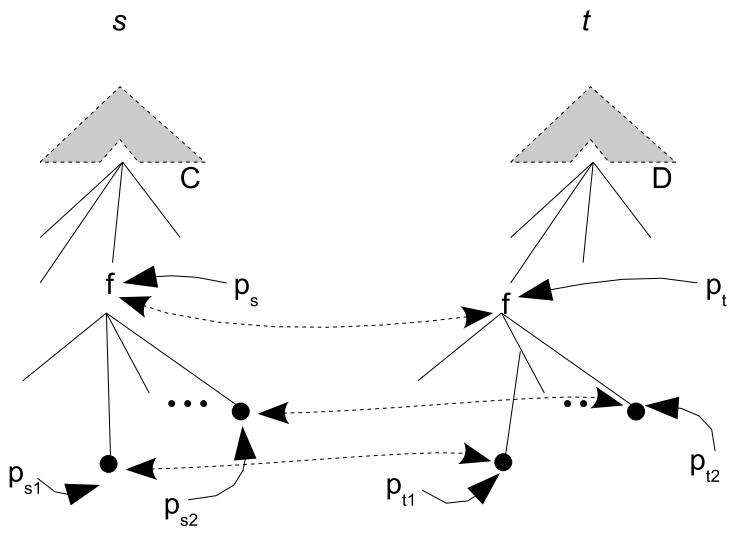


Abbildung 3.4.2. Multikontext-Dekomposition

ohne diese Eigenschaft nicht erfüllt und das Funktionssymbol f nicht fixiert, d. h. keine korrespondierende Position.

Multikontext-Dekomposition. Betrachtet man Multikontexte mit mehr als einem Loch, so stellt man fest, dass Termknoten, die den längsten Präfix von zwei oder mehr Löchern darstellen, korrespondieren. Da der Termbaum nur durch Funktionssymbole verzweigen kann, muss ein solcher Termknoten ein Funktionssymbol f sein. Die Situation ist in Abbildung 3.4.2 beispielhaft dargestellt. Es folgt eine Formalisierung.

Proposition 3.4.14. *Seien p_{s_1} und p_{t_1} sowie p_{s_2} und p_{t_2} verschiedene Paare korrespondierender Positionen in einer Kontextgleichung $s \approx t$. Sei weiter p_s (p_t) der längste gemeinsame Präfix von p_{s_1} (p_{t_1}) und p_{s_2} (p_{t_2}). Dann sind p_s und p_t korrespondierende Positionen in $s \approx t$.*

Da an den Stellen p_s und p_t Funktionssymbole stehen müssen, kann man diese Eigenschaft für eine weitere Regel ausnutzen:

MULTI-KONTEXT-DEKOM-POSITION	$C\ f\ C_1\ \Box^{k_1}\ ,\ldots,C_n\ \Box^{k_n} \approx D\ f\ D_1\ \Box^{k_1}\ ,\ldots,D_n\ \Box^{k_n} \uplus S$ falls $\exists i,j : 1 \leq i,j \leq n$, $i \neq j$, $k_i \geq 1$ und $k_j \geq 1$	$\Rightarrow$	$\{C\,[\Box] \approx D\,[\Box]\,,$ $C_1\ \Box^{k_1} \approx D_1\ \Box^{k_1}\ ,$ $\ldots, C_n\ \Box^{k_n} \approx D_n\ \Box^{k_n}$ $\cup S$
MULTI-KONTEXT-KOLLISION	$C\ f\ C_1\ \Box^{k_1}\ ,\ldots,C_m\ \Box^{k_m} \approx D\ g\ D_1\ \Box^{l_1}\ ,\ldots,D_n\ \Box^{l_n} \uplus S$ falls (1) $\exists i,j : 1 \leq i,j \leq m$ mit $i \neq j$, $k_i \geq 1$ und $k_j \geq 1$, (2) $\exists i,j : 1 \leq i,j \leq n$ mit $i \neq j$, $l_i \geq 1$ und $l_j \geq 1$ und (3) $f \neq g$ oder $\exists i : 1 \leq i \leq m$, $k_i \neq l_j$	$\Rightarrow$	$\bot$

Bei der MULTIKONTEXT-KOLLISION sorgen Bedingungen (1) und (2) dafür, dass es unterhalb der Funktionssymbole f bzw. g jeweils mindestens zwei Löcher in unterschiedlichen Argumenten gibt. Dadurch wird sichergestellt, dass f und g die Funktionssymbole sind, die die Wurzel des längsten gemeinsamen Präfix der Löcher darstellen. In Bedingung (3) werden die beiden Möglichkeiten zusammengefasst, die eine Lösung der Kontextgleichung verhindern: unterschiedliche Funktionssymbole an der Wurzel des längsten gemeinsamen Präfixes sowie unterschiedlich viele Löcher in Funktionsargumenten auf linker und rechter Seite.

PROPOSITION 3.4.15. *In einer Kontextgleichung $s \approx t$ kann es nur eine Funktion f mit den in der Regel* MULTIKONTEXT-DEKOMPOSITION *geforderten Bedingungen geben.*

BEWEIS. Sei $s \approx t$ eine Kontextgleichung. Die Bedingung in der Regel MULTIKONTEXT-DEKOMPOSITION besagt, dass wir eine Funktion f suchen, die im Termbaum links und rechts (vgl. Definition 2.1.4) von sich keinerlei Löcher aufweist, aber mindestens in zweien ihrer Argument-Subterme ein Loch besitzt. Sei f nun diejenige Funktion in s, die unter den bedingungserfüllenden Funktionen die geringste Tiefe im Termbaum hat. Sei g eine beliebige andere Funktion in s, die die gewünschten Eigenschaften ebenfalls besitzt. Dann muss g wegen der Voraussetzung eine größere Tiefe als f haben, mithin also in einem der Argument-Subterme von f enthalten sein. Dann aber hat g im Termbaum links oder rechts von sich ein Loch, da f in mindestens zwei Argumenten ein solches hat. □

Nachdem die in diesem Abschnitt vorgestellten Regeln BODENDEKOMPOSITION und MULTIKONTEXT-DEKOMPOSITION erschöpfend angewandt wurden, wird das Kontextmatchingproblem keine echten Multikontexte sondern nur noch gewöhnliche Term- und Kontextgleichungen (mit nur einem Loch) enthalten.

3.4.4.4. Korrespondierende Lochpfade. Wir haben mit der Regel Bodendekomposition bereits die grundlegende Eigenschaft von Löchern ausgenutzt, dass sie auf natürliche Weise korrespondierende Positionen schaffen. In dieser Regel haben wir lediglich Funktionssymbole direkt oberhalb von Löchern betrachtet.

Wir können jedoch auch die Pfadterme (vgl. Definition 2.1.3) betrachten, die durch Löcher induziert werden.

Proposition 3.4.16. *Sei $s \approx t$ eine Kontextgleichung. Seien weiter $p_s \in Pos(s)$ und $p_t \in Pos(t)$ mit $s|_{p_s} = \Box_k$ und $t|_{p_t} = \Box_k$, d. h. p_s und p_t bezeichnen jeweils die Position des k-ten Lochs auf beiden Seiten der Kontextgleichung. Dann gilt:*

(1) *Die Pfade p_s und p_t korrespondieren in dem Sinne, dass es eine Substitution σ' gibt mit $\frac{s}{p_s}\sigma' = \frac{t}{p_t}$, falls eine Substitution σ existiert mit $s\sigma = t$. σ' ist eine Einschränkung von σ auf die Variablen und Kontextvariablen, die in $\frac{s}{p_s}$ vorkommen. Weiter ersetzt σ' Kontextvariablen durch entsprechende monadische Kontexte.*
(2) *Sind p'_s und p'_t korrespondierende Positionen in $\frac{s}{p_s}$ und $\frac{t}{p_t}$, so existieren entsprechende korrespondiere Positionen in s und t, die auf den Pfaden p_s und p_t liegen.*

Beweis. Wir betrachten Eigenschaft 1. Wir nehmen an, dass σ eine Lösung von $s \approx t$ ist. Wir berechnen die Einschränkung σ':

$$\sigma'(x) := \begin{cases} \sigma(x) & \text{falls } x \in Var\left(\frac{s}{p_s}\right) \\ \tau(x,\sigma) & \text{falls } x \in CVar\left(\frac{s}{p_s}\right) \\ x & \text{sonst} \end{cases}$$

τ ist dabei eine Operation, die eine Substitution monadisiert:

$$\tau(X,\sigma) := \frac{t|_{\texttt{SubstPos}(X,s,t,\sigma)}}{hp}$$

Dabei bezeichnet $\texttt{SubstPos}(X,s,t,\sigma)$ die Position von X in t bezüglich der Substitution σ, d. h. die Stelle der Wurzel der Substitution $\sigma(X)$ im Term $s\sigma$. Diese kann leicht berechnet werden, da wir die komplette Lösungssubstitution σ kennen. hp ist die Position des Lochs im Kontext C: $C[\Box]|_{hp} = \Box$.

Aus der Definition von σ' folgt, dass $\frac{s}{p_s}\sigma' = \frac{t}{p_t}$ gilt.

Wir zeigen nun Eigenschaft 2. Seien also p'_s und p'_t korrespondierende Positionen in den beiden Pfadtermen $\frac{s}{p_s}$ und $\frac{t}{p_t}$ wie beschrieben. Dann berechnen wir zunächst die entsprechenden Positionen $\hat{p}_s$ und $\hat{p}_t$. $\hat{p}_s$ erhalten wir, indem wir den $|p'_s|$-Präfix von p_s nehmen, $\hat{p}_t$ analog.

Wir haben noch zu zeigen, dass $\hat{p}_s$ und $\hat{p}_t$ korrespondieren. Täten sie dies nicht, so gäbe es eine Lösung $\hat{\sigma}$ von $s \approx t$, so dass gilt $s[\Box]_{\hat{p}_s}\hat{\sigma} = t[\Box]_{\hat{p}_t}$ und $s|_{\hat{p}_s}\hat{\sigma} \neq t|_{\hat{p}_t}$. Sei $\hat{\sigma}$ eine solche Lösung. Wir berechnen die Einschränkung σ von $\hat{\sigma}$. Da p'_s und p'_t korrespondieren, gilt nach Eigenschaft 1: $\frac{s}{p_s}[\Box]_{p'_s}\sigma = \frac{t}{p_t}[\Box]_{p'_t}$ und $\frac{s}{p_s}|_{p'_s}\sigma = \frac{t}{p_t}|_{p'_t}$. Da $\hat{\sigma}$ Lösung ist, gilt $s\hat{\sigma} = t$. Mit der Definition

der Einschränkung muss dann aber auch $s\,[\Box]_{\hat{p}_s}\,\hat{\sigma} = t\,[\Box]_{\hat{p}_t}$ und $s|_{\hat{p}_s}\hat{\sigma} = t|_{\hat{p}_t}$ gelten, was einen Widerspruch zur Definition von $\hat{\sigma}$ in der Voraussetzung darstellt. □

Man kann also gewissermaßen die Termpfade $\frac{s}{p_s}$ und $\frac{t}{p_t}$ übereinanderlegen und fixieren. Dies liegt daran, dass Wurzel und Löcher auf natürliche Weise korrespondieren und dadurch den Pfad durch $s\sigma$ und t fixieren.

Die Proposition erlaubt uns, die Termpfade $\frac{s}{p_s}$ und $\frac{t}{p_t}$ unabhängig von den Termen s und t zu betrachten und korrespondierende Positionen zu finden, indem wir das monadische Kontextmatchingproblem $\frac{s}{p_s} \approx \frac{t}{p_t}$ betrachten.

Das monadische Kontextmatchingproblem ist in der Regel wesentlich kleiner als das ursprüngliche Problem. Man kann nun einfach die Berechnung korrespondierender Positionen in $\frac{s}{p_s}$ und $\frac{t}{p_t}$ durch Linearisierung wie in Kapitel 3.4.4.1 durchführen. Durch die wesentlich kleineren Terme ist die Laufzeit des Linearisierungsalgorithmus entsprechend kürzer.

Die dort gefundenen korrespondierenden Positionen p'_s und p'_t (die stets von der Form $1*$ sind) können auf die ursprünglichen Positionen in s und t zurückgerechnet werden, indem man die $|p'_s|$- bzw. $|p'_t|$-Präfixe von p_s bzw. p_t nimmt.

Die Berechnung korrespondierender Lochpfade erbringt also nicht mehr korrespondierende Positionen als das Verfahren der Linearisierung, rechtfertigt sich aber dadurch, dass es auf kleineren Termen operiert.

Wegen der Abhängigkeit von der LINEARISIERUNG gilt auch bei der Methode der KORRESPONDIERENDEN LOCHPFADE, dass sie ein notwendiges, aber keine hinreichendes Kriterium für die Korrektheit der gefundenen korrespondierenden Position ist.

3.4.4.5. *Verbesserung der Anwendbarkeit.* Die Regeln BODENDEKOMPOSITION und -KOLLISION sowie die Berechnung KORRESPONDIERENDE LOCHPFADE benötigen die Existenz von Löchern. Wir können die Anwendbarkeit dieser Regeln erhöhen, indem wir eine neue Regel einführen, die Konstanten durch Löcher ersetzt.

Wie unmittelbar ersichtlich ist, korrespondiert eine Konstante (d. h. Funktion mit Stelligkeit 0) a in einer Kontextgleichung $s \approx t$, wenn a sowohl in s als auch in t genau ein einziges mal vorkommt. In diesem Fall kann a auf beiden Seiten durch ein Loch □ ersetzt werden.

KONSTANTEN-ELIMINATION	$\{C\,[a] \approx D\,[a]\} \uplus S$, falls a weder in $C\,[\Box]$ noch in $D\,[\Box]$ vorkommt.	$\Rightarrow \quad \{C\,[\Box] \approx D\,[\Box]\} \cup S$

Durch Einführung der KONSTANTENELIMINATION ergeben sich potentiell mehr Möglichkeiten, andere Regeln anzuwenden. Man hätte auch BODENDEKOMPOSITION und -KOLLISION sowie KORRESPONDIERENDE LOCHPFADE entsprechend um die Anwendbarkeit auf korrespondierende Konstanten erweitern können, dies hätte jedoch die Regelbeschreibungen übermäßig kompliziert.

KAPITEL 4

Implementierung

Nachdem Algorithmen zur Lösung von Kontextmatchingproblemen auf theoretischer Ebene eingeführt wurden, beschäftigt sich das nun folgende Kapitel mit der Implementierung der vorgestellten Verfahren in der funktionalen Programmiersprache Haskell.

Wir geben zunächst einen Überblick über die Zielsetzungen unserer Implementierung sowohl auf funktionaler wie nichtfunktionaler Ebene. Im Anschluss führen wir kurz das Paradigma funktionaler Programmiersprachen ein.

Es folgt die Betrachtung der verwendeten Basisarchitektur, insbesondere der verwendeten Datenstrukturen zur Repräsentation von Kontexttermen, um schließlich die Implementierung der zuvor eingeführten Algorithmen zur Lösung des generellen und linearen Kontextmatchings vorzustellen.

4.1. Softwaredesign

Die Aufgabe von Design im Allgemeinen nach [**Oes2001**] ist, für ein Problem unter Berücksichtigung vorgegebener Rahmenbedingungen ein Lösungskonzept zu finden. Das Problem selbst wird durch die funktionalen Anforderungen an den zu entwickelnden Algorithmus beschrieben. Die Rahmenbedingungen stellen nichtfunktionale Anforderungen dar, z. B. das Antwort- oder Laufzeitverhalten und Vorgaben wie der zu verwendende Entwicklungsprozess.

Ein durchdachtes Softwaredesign ist wichtig, weil es die prinzipiellen Möglichkeiten und Einschränkungen der Problemlösung definiert und insofern den mit dem erarbeiteten Lösungsansatz erreichbaren Lösungsraum einschränkt. Ziel sollte es daher sein, ein Design zu entwickeln, das eine langfristige Flexibilität in der Weiterentwicklung sowie einen hohen Wiederverwendungsgrad erzielt.

Wichtige Parameter, die ein Softwaredesign beachten sollte, sind Abstraktion, Modularisierung sowie der Grad der Kopplung und Kohäsion zwischen einzelnen Komponenten des Systems.

Abstraktion bedeutet dabei die Trennung eines grundlegenden algorithmischen oder strukturellen Prinzips von der Anwendung auf eine spezifische Probleminstanz. Dabei sind zwei wesentliche Formen der Abstraktion zu unterscheiden: Einerseits die funktionale Abstraktion, bei der das algorithmische Verhalten eines Prozesses in einer Funktion gekapselt wird (wie beispielsweise die Addition in einer Additionsfunktion), andererseits die Datenabstraktion, bei der die Struktur von Daten abstrahiert wird (als

Beispiel sei hier eine generische Baumdatenstruktur genannt, mit deren Hilfe man Baumstrukturen über beliebige Datentypen darstellen kann).

Modularisierung ermöglicht es, ein zu erstellendes System in eigenständige Komponenten zu zerlegen, die eine wohldefinierte Zuständigkeit haben und aus einer von außen sichtbaren Modulschnittstelle sowie einer unsichtbaren Modulimplementierung bestehen. Im Allgemeinen ist ein Modul keine unabhängige Komponente, sondern greift seinerseits wieder auf die Dienste anderer Module zurück. Module dienen als Strukturierungsmittel des zu entwickelnden Softwaresystems und sind damit die Basis für die Entwicklung komplexer Softwaresysteme. Sie sind überschaubare, voneinander weitgehend unabhängige und separat übersetzbare Teile eines Programms. Entscheidend ist die Trennung der Schnittstelle und der Implementierung der in einem Modul enthaltenen Algorithmen und Datenstrukturen (Geheimnisprinzip). Diese ermöglicht es, Module weitgehend unabhängig voneinander zu entwickeln, separat zu testen und relativ leicht auszutauschen.

Unter Kopplung im Rahmen von Softwareentwürfen versteht man das Maß der Verbundenheit der einzelnen Module und Komponenten untereinander. Beispielsweise sind Module stark gekoppelt, falls sie auf gemeinsame Variablen zugreifen sollten. Wenig miteinander gekoppelte Module sind dagegen größtenteils unabhängig, indem sie ein starkes Geheimnisprinzip einsetzen.

Kohäsion bezeichnet dagegen das Maß der Geschlossenheit der Beziehungen zwischen den einzelnen Teilen einer Komponente bzw. eines Moduls. Ein hohes Maß an Kohäsion liegt vor, wenn ein Modul eine einzige logische Funktion implementiert.

Die Qualität eines Softwareentwurfs wird weitgehend davon bestimmt, wie verständlich er einerseits und wie stark adaptierbar er andererseits ist. Die Verständlichkeit bestimmt, wie leicht ein Entwurf nachträglich geändert werden kann. Sie beeinflusst damit wesentlich die Wartbarkeit des Softwareprodukts. Dabei spielen nicht nur die Namensgebung und Dokumentation eine Rolle, sondern auch die Kopplung und Kohäsion der Softwarebestandteile. Die Adaptierbarkeit eines Entwurfs sagt aus, wie leicht man einen Designentwurf auf andere Problemstellungen anwenden kann und gibt somit den Grad seiner Wiederverwendbarkeit und Erweiterbarkeit an.

4.1.1. Ziele. Das im Rahmen der Implementierung zu entwickelnde Softwaredesign soll nicht nur die strukturellen Qualitätsmerkmale Verständlichkeit und Adaptierbarkeit berücksichtigen, sondern auch dem Merkmal Effizienz im Sinne von Laufzeit- und Speichereffizienz genügen. Daraus ergeben sich die wesentlichen Ziele:

(1) Die entwickelten Algorithmen sollen weitgehend unabhängig von der Repräsentation der Kontextterme sein.
(2) Die Lösungsmengen können bereits im linearen Fall exponentiell groß sein. Um den Speicherbedarf beherrschbar zu halten, darf nicht erst die gesamte Lösungsmenge berechnet werden, bevor die Antwort auf ein Kontextmatchingproblem geliefert wird.

(3) Im Falle des generellen Kontextmatchings sind die Regeln nichtdeterministisch anzuwenden, womit die Reihenfolge ihrer Anwendung nicht vorgegeben ist; diese Reihenfolge kann jedoch die Laufzeit der Berechnung beeinflussen. Daher sollten verschiedene Regelauswahlstrategien möglich sein.

4.1.2. Anforderungen. Die spezifischen Anforderungen an das Softwaredesign ergeben sich direkt aus den formulierten Zielen.

(1) Eine Schnittstelle muss die Implementierung der Kontextterme verbergen. Die Implementierung der Algorithmen darf nicht direkt von den Datentypen der Terme abhängig sein.
(2) Teilmengen der Lösungen dürfen erst auf Anforderung berechnet werden (lazy evaluation, streaming). Erst nach Konsumption einer Teillösung (in unserem Falle einer möglichen Substitution) wird die nächste Teillösung auf Anforderung berechnet.
(3) Es sollte eine Heuristik entwickelt werden, die über die Anwendung einer bestimmten Regel aus einer Menge potentiell möglicher Regeln entscheidet. Die Heuristik sollte dabei getrennt sein von der Regelimplementierung, um zwischen den Regelprämissen und der Regelanwendung unterscheiden zu können.

Eine nichtfunktionale Anforderung an das Design stellt die Entscheidung für die Programmiersprache Haskell [**Pey2002**] dar. Die heute zur Verfügung stehenden Haskell-Compiler implementieren den Haskell98-Standard in vollem Umfang und bieten darüber hinaus oftmals Spracherweiterungen. Wir wollen diese Erweiterungen nur dort nutzen, wo es sinnvoll oder notwendig erscheint.

Wir beschränken uns daher auf die praktische Erweiterung durch Multiparameter-Typklassen. Diese erlauben die Parameterisierung von Typklassen mit mehr als einem Parameter, wie es beispielsweise bei unseren Baumtypen notwendig sein wird: ein Parameter der Typklasse `ContainerTree` ist der Baumtyp, der zweite ist der Typ der in den Baumknoten gespeicherten Elemente.

Darüberhinaus benötigen wir die Erweiterung der funktionalen Abhängigkeiten ([**Jon2000**]), die im Zusammenhang mit Multiparameter-Typklassen nützlich sein können.

Beide Erweiterungen werden sowohl durch den Compiler „Glasgow Haskell Compiler" [**GHC2004**] als auch durch den Interpreter „Hugs" [**Jon2003**] unterstützt.

4.2. Funktionales Programmieren

Dieser Abschnitt stellt das funktionale Programmierparadigma kurz vor.

4.2.1. Funktionen. Ausgangspunkt des funktionalen Programmierens ist der mathematische Begriff einer Funktion. Eine Funktion überführt Werte aus ihrem Definitionsbereich $\mathbb{D}$ in Werte ihres Wertebereichs $\mathbb{W}$. Hat man also mit $x \in \mathbb{D}$ und $y \in \mathbb{W}$ zwei variable Größen und kann man jedem x genau ein y zuordnen, so nennt man y eine Funktion von x und schreibt $y = f(x)$. Auch Funktionen von mehreren veränderlichen Größen $x_1, \ldots, x_n$ sind möglich und werden $y = f(x_1, \ldots, x_n)$ geschrieben.

Die Forderung, dass eine Funktion eine eindeutige Zuordnung eines Definitionsbereichs auf einen Wertebereich repräsentiert, stellt sicher, dass die Anwendung einer bestimmten Funktion auf dasselbe Argument stets den gleichen Wert liefert.

Wir können damit bereits einfache Abstraktionen wie das Doppelte einer Zahl definieren:

$$double(x) := x + x$$

Auch rekursive Funktionen, bei denen die Funktion sich selbst wieder in ihrer Definition verwendet, sind möglich:

$$ntimes(n, x) := \begin{cases} 0 & \text{falls } n = 0 \\ x + ntimes(n-1, x) & \text{sonst} \end{cases}$$

Bei der Berechnung einer Funktion wird dann die Funktion entsprechend ihrer Argumente ausgewertet. Ein Beispiel für $ntimes(2, 3)$:

$$\begin{aligned} ntimes(2,3) &\rightarrow 3 + ntimes(1,3) \\ &\rightarrow 3 + 3 + ntimes(0,3) \\ &\rightarrow 3 + 3 + 0 \\ &\rightarrow 3 + 3 \\ &\rightarrow 6 \end{aligned}$$

4.2.2. Referenzielle Transparenz. Die durch den mathematischen Funktionsbegriff geforderte Eigenschaft von Funktionen, die selben Argumente stets auf den selben Wert abzubilden, führt zum Begriff der referenziellen Transparenz. Dieser bedeutet, dass ein Ausdruck stets und nur allein von seinen Teilausdrücken abhängt. Mit anderen Worten ist ein Ausdruck nur eine bestimmte Repräsentation eines Wertes. Zum Beispiel repräsentieren die Ausdrücke $ntimes(2, 3)$ und $2 + 4$ denselben Wert, nämlich 6[1].

Der wichtigste Aspekt der referenziellen Transparenz ist die Eigenschaft der Unabhängigkeit von externen Einflüssen. Funktionale Programmiersprachen bauen auf dem Prinzip der referenziellen Transparenz auf. Ein Beispiel in der Programmiersprache Java verdeutlicht, warum globale Variablen in Verbindung mit destruktiven Änderungen an diesen zum Verlust der referenziellen Transparenz führen können (siehe Algorithmus 4.2.1).

[1] Dabei ist 6 natürlich auch nur eine Repräsentation des abstrakten Wertes, der hinter dem „Konzept" 6 steht.

Algorithmus 4.2.1 Referentielle Integrität: Beispiel

```
public class RefIntegrity {
  public static int times = 0;

  public static int f(int a) {
    times++;
    return (a*times);
  }

  public static void main(String[] args) {
    int r1 = f(2) + f(1);
    int r2 = f(1) + f(2);
    System.out.println(r1); // ==> 4
    System.out.println(r2); // ==> 11
  }
}
```

Der Wert der Ausdrücke $f\,(1)$ und $f\,(2)$ ist an keiner Stelle des Programms der Gleiche, da die globale Variable `times` einen *Seiteneffekt* einführt. Funktionale Programmiersprachen ohne Seiteneffekte werden als *pur* bezeichnet.

4.2.3. Auswertungsreihenfolgen. Im Allgemeinen existieren mehrere Wege, einen einzigen Ausdruck auszuwerten, d. h. seinen Wert zu berechnen. Wir betrachten kurz die beiden „Extremformen": die Anwendungsordnung und die Normalordnung. Betrachten wir eine Funktion $double\,x = x + x$. Berechnet man nun $double\ (3 + 3)$, so gibt es folgende Möglichkeiten der Auswertung:

Anwendungsordnung	Normalordnung
$double\ (3 + 3)$	$double\ (3 + 3)$
$\rightarrow double\,6$	$\rightarrow (3 + 3) + (3 + 3)$
$\rightarrow 6 + 6$	$\rightarrow 6 + (3 + 3)$
$\rightarrow 12$	$\rightarrow 6 + 6$
	$\rightarrow 12$

4.2.3.1. *Anwendungsordnung.* Von Anwendungsordnung (auch Applikationsordnung genannt) spricht man, wenn bei einer Funktionsanwendung $f\,x$ zunächst das Argument x zu einem Wert ausgewertet und das Ergebnis in die Definitionsgleichung der Funktion f eingesetzt wird.

4.2.3.2. *Normalordnung.* Die Normalordnung geht dagegen umgekehrt vor: Bei der Applikation $f\,x$ wird das Argument nicht ausgewertet, sondern direkt in die Definitionsgleichung von f eingesetzt. Das Argument wird erst dann ausgewertet, wenn die Auswertung der Funktion f während der Berechnung danach verlangt.

Nicht jede Auswertungsreihenfolge muss terminieren. Wir betrachten die Funktion K, die wie folgt definiert wird

$$K\,x\,y = x$$

Sei y nun ein Term, dessen Auswertung nicht terminiert, wie zum Beispiel $t = t$. Dann terminiert die Auswertung in Anwendungsordnung nicht, da zunächst die beiden Argumente x und y von K ausgewertet werden. Die Normalordnung hingegen kann zu einem Wert reduzieren, da y in der Definitionsgleichung von K nicht benötigt wird.

Man kann zeigen, dass die Normalordnung eines Ausdrucks stets terminiert, falls dieser Ausdruck mit einer beliebigen Auswertungsreihenfolge zu einem Wert reduziert werden kann [**Schm2000**].

4.2.4. Sharing. Wir haben jedoch in dem Beispiel der Auswertung von *double* $(3 + 3)$ in Normalordnung gesehen, dass es durch Kopieren von Subtermen zu Doppelberechnungen kommen kann. In diesem Beispiel wird der Argumentterm $(3 + 3)$ doppelt in die Definitionsgleichung von *double* eingesetzt, so dass dieser Term während der Reduktion zweifach ausgewertet wird.

Daher wird in den aktuellen Implementierungen funktionaler Programmiersprachen *Sharing* verwendet. Dabei werden Terme nicht in Form von Termbäumen sondern als Termgraphen dargestellt, in denen mehrfach auftretende Subterme durch Referenzen auf denselben Knoten des Graphen modelliert werden. Dadurch steht das Berechnungsergebnis eines solchen Knotens sofort auch allen anderen Knoten zur Verfügung, die diesen Knoten referenzieren.

Man kann zeigen, dass die Normalordnung in Verbindung mit Sharing höchstens genau so viele Berechnungsschritte benötigt wie die Anwendungsordnung [**Schm2000**].

Funktionale Programmiersprachen, die die Anwendungsordnung verwenden, werden als *strikt* bezeichnet; solche, die die Normalordnung verwenden, heißen *nicht-strikt* oder auch *verzögernd auswertend*.

4.3. Basisarchitektur

Wir legen unser Augenmerk nun auf einen Vertreter der funktionalen Programmiersprachen - die nicht-strikte pure funktionale Sprache *Haskell*. Wir verzichten an dieser Stelle auf eine Einführung in Haskell und verweisen stattdessen auf [**Bird1998**] und [**Hud2000**].

4.3.1. Modulüberblick. Wir verwenden eine überaus praktische Erweiterung des Haskell98-Standards [**Pey2002**] - hierarchische Modulnamen. Damit ist es – korrespondierend zu den entsprechenden Mechanismen in Programmiersprachen wie Java oder C# – möglich, Modulen eine hierarchische Struktur zu geben, die sich auch in der Verzeichnisstruktur des Sourcecodes widerspiegelt.

In Schaubild 4.3.1 wird zunächst die Modulstruktur - ohne die Module selbst - vorgestellt. In diesem Abschnitt befassen wir uns zunächst mit den Modulen in den Paketen `Gersdorf.Data` und `Gersdorf.ContextMatching.Base`, in denen die Module zusammengefasst sind, die sowohl für das lineare als auch das generelle ContextMatching

`Gersdorf`			
	`ContextMatching`		
		`Base`	Basisdatenstrukturen für Terme, Substitutionen etc.
		`Linear`	Algorithmen für lineares Contextmatching
		`General`	Algorithmen für generelles Contextmatching
		`General` `Rules`	Regeln für generelles Kontextmatching
		`General` `CorrespondingPositions`	Algorithmen zur Berechnung korrespondierender Positionen
		`Test`	Testalgorithmen
	`Data`		Allgemeine Datenstrukturen
	`Algorithms`		Allgemeine Algorithmen

Abbildung 4.3.1. Modulüberblick

benötigt werden. Dies sind insbesondere die Datenstrukturen für Bäume, Terme und Substitutionen.

Für die wesentlichen Datenstrukturen wurden Fassaden in Form von Typklassen erstellt, um einen generischen Zugriff auf sie zu ermöglichen und die entwickelten Algorithmen weitgehend unabhängig von der tatsächlichen Repräsentation, beispielsweise der Terme, zu machen. Wir stellen die Typklassen in den folgenden Abschnitten kurz vor und betrachten dann die spezifischen Vor- und Nachteile der einzelnen Implementierungen.

4.3.2. Basisdatenstrukturen. Im Package `Gersdorf.Data` sind allgemeine Basisdatenstrukturen implementiert. Dies betrifft insbesondere die Datenstrukturen für Bäume, da die zu verarbeitenden Kontextterme im wesentlichen Bäume sind.

Wir verwenden mehrere verschiedene Baumrepräsentationen. Bevor wir zu den Implementierungdetails kommen, betrachten wir die Typklassen, die für die algebraischen Datentypen rund um Bäume entwickelt wurden.

4.3.2.1. *Bäume.* Einen Überblick über die Typklassen rund um Bäume gibt Abbildung 4.3.2. Tabelle 4.3.1 erläutert die einzelnen Typklassen.

Bäume können unter verschiedenen Aspekten betrachtet werden. Zum einen definieren Bäume eine Struktur. Die Struktureigenschaft wird über die Typklassenhierarchie, die mit der zentralen Typklasse `Tree` beginnt, abgebildet. Zum anderen enthalten Bäume Nutzdaten. Diese Sichtachse wird über die Typen abgeleitet von der Typklasse `ContainerTree` dargestellt. Schließlich können sie Informationen enthalten, die sowohl von der Struktur als auch vom Inhalt abhängen. Dies wird über die Typen ab der Typklasse `ContextableTree` abgedeckt.

Zentrales Element der Hierarchie ist die Typklasse `Tree`. Alle Bäume sind Mitglieder dieser Typklasse, die das Navigieren in Richtung der Kinder über die Operation `getChildren` erlaubt. Sie wird erweitert durch die Typklasse `NavigatableTree`, die die Fähigkeit der Navigation in Richtung der Vaterknoten hinzufügt. Die direkte Adressierung von Knoten wird

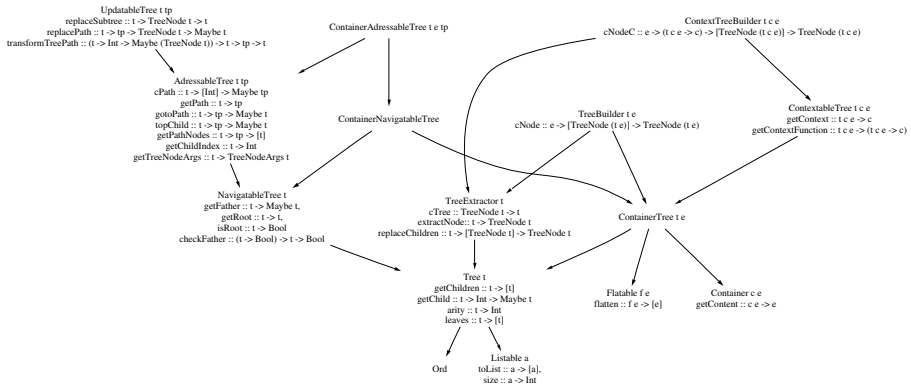


Abbildung 4.3.2. Typklassen der Basisdatenstrukturen

Typklasse	Bedeutung und Fähigkeiten
`Tree`	Basistypklasse. Erlaubt Navigation in Richtung Kinder. Reiner Strukturbaum.
`NavigatableTree`	Navigation in Richtung Vater und in Richtung Kinder.
`AdressableTree`	Navigation und Adressierung mit Hilfe von `TreePath`s.
`UpdatableTree`	Navigation, Adressierung und Strukturänderungen.
`ContainerTree`	Baum mit Inhalt und Navigation in Richtung Kinder.
`ContainerNavigatableTree`	Wie `NavigatableTree` und `ContainerTree`.
`ContainerAdressableTree`	Wie `AdressableTree` und `ContainerTree`.
`ContextableTree`	Baum mit Inhalt, Navigation in Richtung Kinder und Kontexten in den Knoten (Daten abhängig von der Baumstruktur).
`TreeExtractor`	Baum mit der Fähigkeit, Teilbäume zu extrahieren.
`TreeBuilder`	Baum mit der Fähigkeit, Baumknoten zu erzeugen.
`ContextTreeBuilder`	Wie `ContextableTree` und `TreeBuilder`.

Tabelle 4.3.1. Übersicht über Typklassen von Bäumen

durch Typen, die Instanzen der Typklasse `AdressableTree` sind, ermöglicht. `AdressableTree`s bieten für jeden Knoten eine Adresse in Form eines Pfades wie in Definition 2.1.3 an, über die ein Knoten des Baums adressiert werden kann.

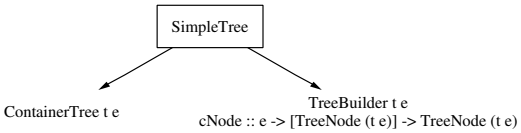


ABBILDUNG 4.3.3. SimpleTree

Um einen Baum aufbauen zu können, muss ein Datentyp Mitglied der Typklasse `TermBuilder` sein. Über die Funktion `cNode` in Verbindung mit `cTree` kann dann sukzessive ein Baum aufgebaut werden, wie das folgende Beispiel zeigt:

```
let b=(cTree (cNode 1 [cNode 2 [], cNode 3 []]))
        :: SimpleTree Int
```

4.3.2.2. *SimpleTree.* Die erste Implementierung von Bäumen - `SimpleTree` genannt - nutzt eine Darstellung, in der der Baum durch einen normalen Haskell-Datentyp repräsentiert wird. Dies entspricht der naheliegensten Umsetzung:

```
data SimpleTree a = SimpleTree {
    value :: a,                 -- Inhalt
   children :: [SimpleTree a] -- Verweise auf Kindknoten
}
```

Wir verwenden in vielen Fällen Haskell-Records für Datentypen, da diese zum einen selbstdokumentierend sind und zum anderen leichtere Anpassungen erlauben als die Verwendung „normaler" Konstruktoren. Benutzt man statt Records einfache Konstruktoren, so muss man bei einer kleinen Änderung eines Datentyps in der Regel alle Funktionen ändern, in denen gegen den Datentyp ein Patternmatching vorgenommen wird. Dies ist bei Verwendung von Records im Allgemeinen nicht notwendig außer bei Funktionen, die eine neue Instanz eines Datums erzeugen.

4.3.2.3. *ExtendedTree.* Die Darstellung eines Baums als `SimpleTree` hat den Nachteil, dass, ausgehend von einem Baumknoten, eine Navigation nur in Richtung der Kinder möglich ist, nicht jedoch in Richtung des Vaters. In vielen praktischen Anwendungsfällen ist es jedoch wünschenswert, genau diese Navigationsachse zur Verfügung zu haben. Hierfür ist es notwendig, in einem Knoten nicht nur Verweise auf die Kinder sondern auch auf den Vater zu speichern:

```
data ExtendedTree a = ExtendedTree {
   value :: a,
   children :: Array Int (ExtendedTree a),

   father :: Maybe (ExtendedTree a),
   childIndex :: Int,
   path :: StandardTreePath
}
```

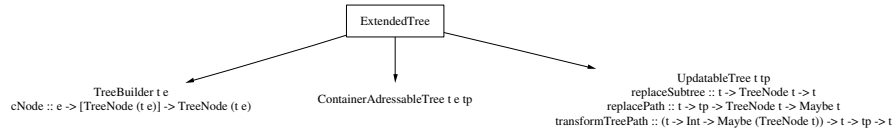


Abbildung 4.3.4. ExtendedTree

Die Bedeutung der einzelnen Felder wird im Folgenden beschrieben: Der Vaterknoten wird in einem Datum vom Typ `Maybe (ExtendedTree a)` gespeichert. Ist dieser Wert `Nothing`, so existiert kein Vater des beschriebenen Knoten, und der Knoten ist mithin die Wurzel des Baums.

Problematisch ist zunächst, einen Baum aus Knoten dieses Typs zu konstruieren, da bei der Konstruktion eines gegebenen Knotens sowohl der Vater- als auch die Kindknoten bekannt sein müssen. Haskell bietet jedoch durch das rekursive `let` (in Verbindung mit verzögert auswertenden Funktionen) eine Lösung. Wir definieren die Funktion `cNode` wie folgt (nur relevante Auszüge):

```
-- ggf. Paar aus Vaterknoten und Kindindex
-- des zu konstruierenden Knoten.
type TreeNodeArgs tb = Maybe (tb, Int)
-- nimmt TreeNodeArgs und konstruiert einen
-- Baum.
type TreeNode tb = TreeNodeArgs tb -> tb
cNode :: e -> [TreeNode (t e)] -> TreeNode (t e)
cNode v cs f =
  let node = ExtendedTree {
               value = v,
               father = case (f) of
                 Nothing            -> Nothing
                (Just (father, _)) -> Just father,
              children = listArray (1, length cs)
                  (map (\ (etcc, n)
                    -> etcc
                       (Just (node, n)))
                       (zip cs [1..])),
                -- weitere Felder
                -- (hier ausgelassen)
              }
  in  node
```

Dabei sind die Parameter `v` der Knoteninhalt, `cs` die Knotenkinder und `f` ein Paar aus Knotenvater und Kindindex des neu zu schaffenden Knotens.

Mit Hilfe der Funktion `cNode` kann nun ein Baum von `TreeNode`s erzeugt werden. Damit wir einen kompletten Baum erhalten, muss noch der Vater der Wurzel festgelegt werden, wofür die Funktion `cTree` existiert:

```
cTree :: TreeNode t -> t
cTree tn = tn Nothing
```

Wir bezeichnen den Vorgang, einen Teilbaum eines Baumes als eigenständigen Baum zu betrachten, als *Extraktion.*

Soll ein Teilbaum extrahiert werden, so muss darauf geachtet werden, dass die Vaterbeziehungen im kompletten Teilbaum neu berechnet werden müssen, damit kein Knoten des extrahierten Teilbaums über die Vaterbeziehung wieder in den ursprünglichen Baum verweisen kann. Das liegt am Sharing, welches in Haskell implementiert ist. Beachtet man dies nicht, so kann sich folgende beispielhafte Situation ergeben: Sei $t := f(g(x))$ ein Ausdruck, der als `ExtendedTree` dargestellt wird. Dann gilt in symbolischer Schreibweise `father` $f(g(x))$`=Nothing`, `father` $g(x)$`=Just` $f(g(x))$ und `father` x`=Just` $g(x)$. Wir extrahieren nun den Teilbaum $g(x)$ und erhalten $g'(x')$. Berechnen wir die Vaterbeziehungen nicht komplett neu, sondern setzen nur den Vater des Wurzelknotens von $g(x)$ auf `Nothing`, so gilt zwar `father` $g(x)$`=Nothing`, da aber durch das Sharing $x = x'$ gilt, auch `father` x'`=Just` $g(x)$ und mithin `father (father` x`)=Just` $f(g(x))$.

Mit der durch das Sharing notwendigen Neuberechnung erhält man eine Laufzeit von $O(m)$, wobei m die Größe des zu extrahierenden Teilbaums ist.

```
extractNode :: t
  -> TreeNode t
extractNode t =
  cNode (getContent t)
        (map extractNode (getChildren t))
```

Die Felder `children`, `childIndex` und `path` verdienen eine nähere Betrachtung. Um in einem Baum einen Knoten eindeutig zu referenzieren, bietet sich eine Darstellung wie in Definition 2.1.3 an, in der der Pfad von der Wurzel zum Knoten als ein String von Integern verstanden wird, wobei jeder Integer die Auswahl einer ausgehenden Kante eines Knotens bedeutet.

Exakt auf diese Weise ist ein `StandardTreePath` gebildet:

```
data StandardTreePath = StandardTreePath {
   stpPath :: [Int],
   stpHash :: Int32
} deriving Show
```

Die Hash-Komponente dient dem Gleichheitstest von `StandardTreePath`s in konstanter Zeit (sobald der Hash einmal berechnet wurde, was durch verzögerte Auswertung bei der ersten Anforderung der Komponente `stpHash` erfolgt).

In jedem Knoten eines `ExtendedTree`s wird der zugehörige `StandardTreePath` gespeichert. Um eine schnelle Navigation mit Hilfe der TreePaths innerhalb des `ExtendedTree`s zu erreichen, werden die Kindverweise nicht in einer Liste, sondern in einem `Array` als `children` festgehalten, um einen $O(1)$-Zugriff auf einzelne Kinder zu erhalten.

Oft ist die Information nützlich, das wievielte Kind seines Vaters ein betrachteter Knoten ist. Daher halten wir dieses Datum in dem Feld `childIndex`

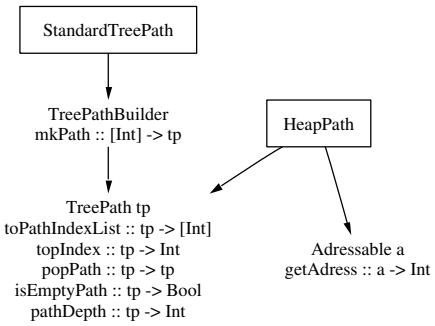


ABBILDUNG 4.3.5. TreePath

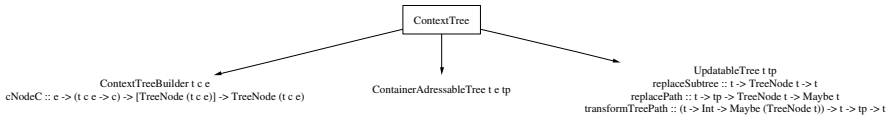


ABBILDUNG 4.3.6. ContextTree

fest. Man könnte diese Information zwar auch aus dem letzten Listenelement des `path`s erhalten; dies wäre jedoch nur in $O(|p|)$ möglich, wobei p der Pfad selbst ist.

4.3.2.4. *ContextTree.* In vielen Situationen ist es notwendig, Baumanalysen in Abhängigkeit nicht nur des Knoteninhalts sondern auch der Baumstruktur durchzuführen. Ein einfaches Beispiel hierfür ist das Zählen der Knotenanzahl eines Teilbaums.

Diese kontextabhängig zu berechnenden Daten werden im Folgenden *Kontexte* genannt und sind nicht mit Kontexten aus der Termbegrifflichkeit zu verwechseln.

`ContextTree`s sind eine Weiterentwicklung der `ExtendedTree`s und haben folgenden Datentyp:

```
data ContextTree c a = ContextTree {
  value :: a,
  context :: c,
  children :: Array Int (ContextTree c a),
  -- ab hier: Verwaltungsinformationen!
  father :: Maybe (ContextTree c a),
  childIndex :: Int,
  path :: StandardTreePath,
  contextFunction :: ContextTree c a -> c
}
```

In den Knoten wird sowohl der berechnete `context` als auch ein Verweis auf die berechnende Funktion gespeichert, die `contextFunction`. Damit ist der Typ der knotenerzeugenden Funktion `cNode` wie folgt verändert:

```
cNodeC :: e
```

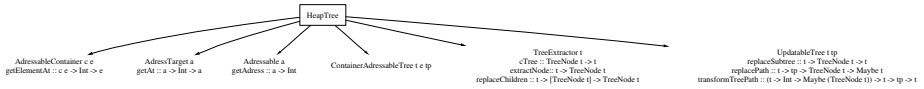


Abbildung 4.3.7. HeapTree

```
-> (t c e -> c) -- contextFunction
-> [TreeNode (t c e)]
-> TreeNode (t c e)
```

Der Name der Funktion musste verändert werden, da Haskell keine identischen Funktionsnamen mit unterschiedlichen Signaturen erlaubt. Die Funktionsweise ist im Wesentlichen wie die der Funktion `cNode`, nur dass die `contextFunction` im Knoten gesichert wird und mit Hilfe der `contextFunction` der Wert des Kontextes berechnet wird:

```
cNodeC v ctf cs f =
  let node = ContextTree {
               value = v,
               context = ctf node,
               contextFunction = ctf,
               -- weitere Felder wie bei
               -- cNode (ausgelassen)
             }
  in  node
```

Der wesentliche Vorteil des Typs `ContextTree` ist, dass die zu berechnenden Kontextinformationen durch die verzögerte Auswertung von Haskell nur einmal - beim ersten Zugriff auf das Knotenelement `context` - berechnet werden. Bei weiteren Zugriffen ist die Information bereits berechnet und steht sofort zur Verfügung.

4.3.2.5. *HeapTree.* Im Gegensatz zu den bisherigen Baumdarstellungen repräsentiert der Datentyp `HeapTree` einen Baum in einem Array:

```
type HeapAddr =Int
data HeapCell a
 = HeapCell  (Maybe HeapAddr)  – Evtl. Adresse des Vaters
             a                 – Inhalt des Knotens
             [HeapAddr]        – Adressen der Kinder

data HeapTree a
 = HeapTree  HeapAddr          – Adresse der Baumwurzel
             (Array HeapAddr   – Array mit Baumknoten
             (HeapCell a))
```

Ein `HeapTree` ist also ein Array von `HeapCell`s, wobei jede Zelle optional eine Vaterzelle und mehrere Kindzellen besitzen darf. Durch diese manuell vorgenommene Verzeigerung wird der Baum im Array dargestellt.

Vorteilhaft an der Repräsentation als `HeapTree` ist, dass die einzelnen Knoten des `HeapTree`s direkt mit einer `HeapAddr` (also einem `Integer`) adressierbar sind und damit jeder einzelne Knoten in $O(1)$ erreichbar ist

Typklasse	Bedeutung und Fähigkeiten
`TermTypeable`	Test auf Termknotentypen wie z. B. Test, ob ein Knoten eine Variable ist etc.
`TermIterator`	Term hat spezifisches `fold`.
`Term`	Zentrale Term-Typklasse. Bietet alle Operationen eines `NavigatableTree`s sowie zusätzliche Analysefunktionen, z. B. ob ein Term ein Grundterm ist.
`AnalysisTerm`	Ein Term, der besondere (rechenintensive) Analysefunktionen bereitstellt.
`PositionTerm`	Term, der positionsabhängige Operationen und Termänderungsfunktionen anbietet. Zum Beispiel kann ein Subterm mit der Funtion `makeContext` durch ein Loch ersetzt werden.
`AdressableTerm`	Term, bei dem die Termknoten direkt (in $O(1)$) adressiert werden können und der dementsprechende Operationen anbietet.
`SubTermBuilder`	Datenstruktur, die Termknoten (Funktionen, Variablen, Kontextvariablen Löcher) sowie Strukturen aus Termknoten erzeugen kann.
`TermBuilder`	Datenstruktur, die aus einer Termknotenstruktur einen Term erzeugen kann.

TABELLE 4.3.2. Übersicht über Typklassen von Termen

[**Mar2002**]. Dies ist von entscheidender Bedeutung für den Algorithmus zur Lösung linearer Kontextmatchingprobleme, wie wir noch sehen werden. Dabei ist es wichtig zu beachten, dass normale Haskell-Arrays nicht veränderlich (immutable) sind, so dass ein Update auf eine einzelne Arrayzelle die Komplexität $O(n)$ besitzt.

`HeapTree`s benutzen eine gesonderte Instanz eines `TreePath`s: `HeapPath`s (siehe Abbildung 4.3.5). Ein `HeapPath` kapselt neben dem Pfad auch die `HeapAddr` eines Knotens, so dass dieser in $O(1)$ adressierbar ist.

4.3.3. Terme. Das Package Terme `Gersdorf.ContextMatching.Base` beheimatet Terme und ihre Repräsentationen. Sie werden mit Hilfe der bereits dargestellten Baum-Datenstrukturen modelliert. Wir definieren Terme als Bäume über Knoten des Typs `TermType`.

```
data TermType = TermHole
              | TermVariable String
              | TermContextVariable String
              | TermFunction String
```

Wir betrachten zunächst Terme allgemein und die Bedeutung der einzelnen Typklassen in diesem Zusammenhang.

Eine Übersicht über die Typklassenstruktur der Terme bieten Abbildung 4.3.8 sowie Tabelle 4.3.2.

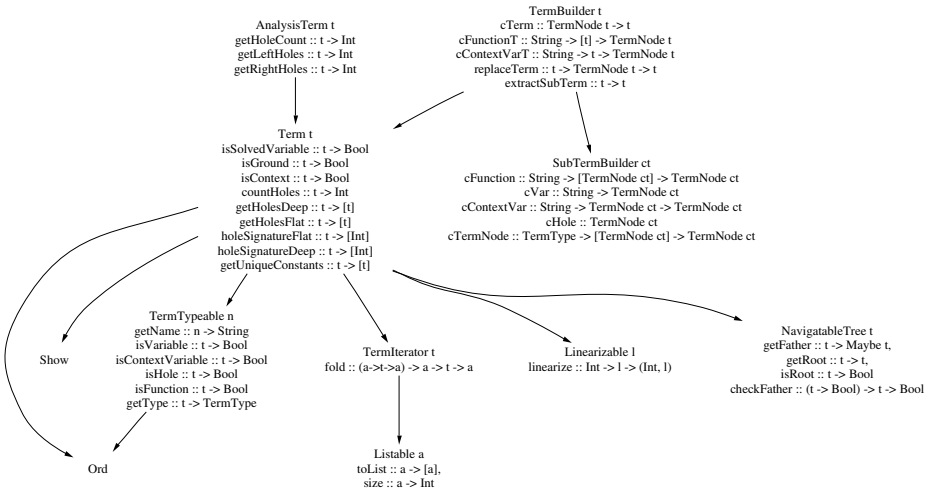


Abbildung 4.3.8. Typklassenhierarchie der Terme

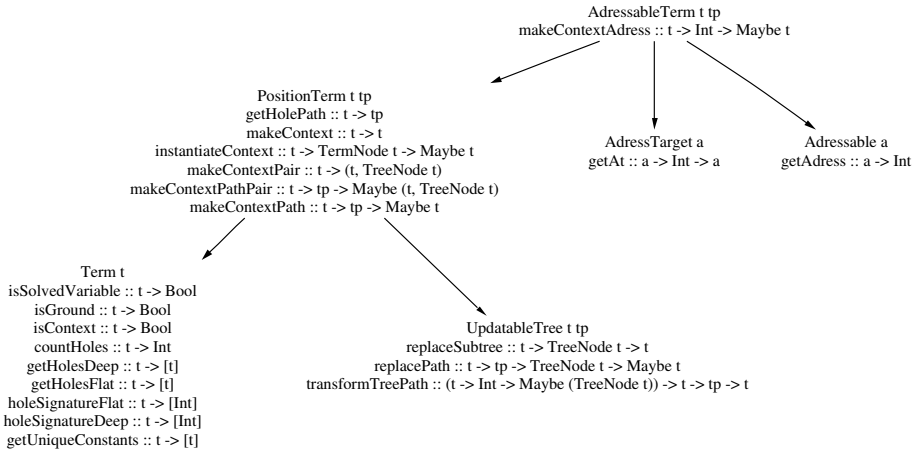


Abbildung 4.3.9. Typklassenhierarchie von `PositionTerm`s

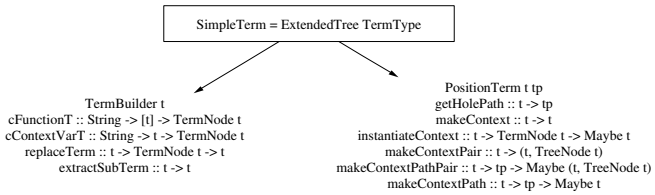


Abbildung 4.3.10. SimpleTerm

Es existieren entsprechend der aufgezeigten Baumdarstellungen unterschiedliche Termrepräsentationen: `SimpleTerm`, `ExtendedTerm` und `HeapTerm`. Sie erhalten ihre charakteristischen Eigenschaften durch die verschiedenen unter ihnen liegenden Baumtypen.

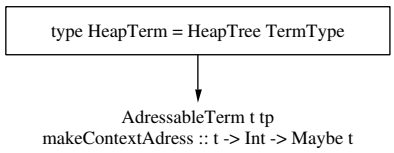


Abbildung 4.3.11. HeapTerm

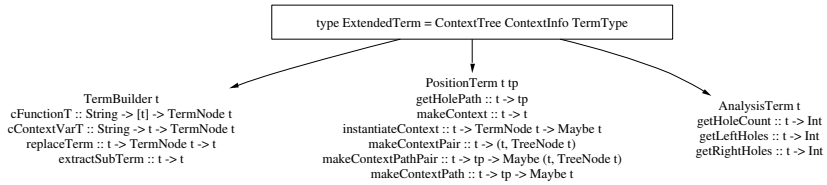


Abbildung 4.3.12. ExtendedTerm

```
type SimpleTerm = ExtendedTree TermType
type HeapTerm = HeapTree TermType
type ExtendedTerm = ContextTree ContextInfo TermType
```

In den implementierten Algorithmen werden vornehmlich Terme der Typen `ExtendedTerm` und `HeapTerm` eingesetzt. Im Rahmen des linearen Kontextmatchings verwenden wir `HeapTerm`s wegen ihrer Eigenschaft, dass einzelne Termknoten direkt mittels eines Integers in $O(1)$ adressiert werden können. Bei der Implementierung des generellen Kontextmatchings benutzen wir `ExtendedTerms`, da diese schnelle Analysemöglichkeiten durch ihre Mitgliedschaft in der Typklasse `AnalysisTerm` bereitstellen.

4.3.4. Substitutionen. Eine Substitution ist eine Menge von Ersetzungen, die entweder eine Variable durch einen Term oder eine Kontextvariable durch einen Term mit einem Loch ersetzen. Dazu gibt es die primitive Definition eines `SubstitutionAtom`s:

```
data Term t => SubstitutionAtom t = SubstBot
                  | SubstTop
                  | SubstVar TermType t
                  | SubstContextVar TermType t
```

Sei S die Menge von Substitutionsatomen über einen Termtyp. Dann gilt die folgende Logik:

`SubstBot` $\wedge\, s, s \in S$	$\equiv$	`SubstBot`
`SubstBot` $\vee\, s, s \in S$	$\equiv$	s
`SubstTop` $\wedge\, s, s \in S$	$\equiv$	s
`SubstTop` $\vee\, s, s \in S$	$\equiv$	s

Dies entspricht der Erkenntnis, dass ein `SubstTop` keinen Anteil an der Lösung besitzt, wohingegen ein `SubstBot` die Lösung ungültig macht.

Substitutionen selbst sind definiert als Listen von `SubstitutionsAtom`en:

```
newtype (Term t) =>
  Substitution t = Substitution [SubstitutionAtom t]
```

4.3.4.1. *Typklassen.* Wichtig sind nun die beiden Typklassen `SubstitutionGenerator` und `SubstitutionBuilder`. Ein `SubstitutionBuilder` ist ein algebraischer Datentyp, der Substitutionen aus Und- und Oder-Kombinationen von `SubstitutionAtom`en aufbauen kann. Es ergibt sich damit eine komplexe Struktur von logischen Substitutionsgleichungen. Ein `SubstitutionBuilder` dagegen ist ein Datentyp, der Substitutionen in der oben genannten (flachen) Form generiert. Von Bedeutung ist die geforderte Eigenschaft der Funktion `generate`, die Liste der Substitutionen verzögert aufzubauen, so dass der Funktionsaufrufer den Kopf der Liste bereits frühzeitig konsumieren kann.

```
class (Term t) =>
  SubstitutionGenerator sg t | sg -> t where
    simplify :: sg -> sg
    generate :: sg -> [Substitution t]
class (PositionTerm t, SubstitutionGenerator s t) =>
      SubstitutionBuilder s t where
  substTop :: s
  substBot :: s
  substVar :: t -> t -> s
  substContextVar :: t -> t -> Int -> s
  substAnd :: [s] -> s
  substOr :: [s] -> s
```

Wir verwenden bei der Definition des `SubstitutionGenerators` eine Erweiterung des Haskell98-Standards [**Pey2002**]: funktionale Abhängigkeiten (`sg -> t`) [**Jon2000**]. Diese werden benötigt bei Multiparametertypklassen, die Funktionen definieren, bei denen Parametertypen als Ergebnisparameter angegeben werden, nicht jedoch innerhalb der Eingabeparameter einer Funktion (wie bei der Funktion `generate :: sg -> [Substitution t]`).

Nach unserer Definition besagt `sg -> t`, dass der Typ des `SubstitutionGenerators` eindeutig den Typ der generierten Terme bestimmt. Dies ist auch fachlich sinnvoll.

Die Typklasse `SubstitutionSet` dient der internen Operation mit einer Menge von Substitutionen. Eine Instanz dieser Typklasse ist eine Abbildung von Termtypen auf Terme. Sie ist wie folgt definiert:

```
class (Term t) => SubstitutionSet s t where
  emptySubstitution :: s t
  hasSubstitution :: s t  -> TermType -> Bool
  bindSubstitution :: s t  -> TermType -> t  -> s t
  lookupSubstitution :: s t -> TermType -> Maybe t
  lookupSubstitutionST ::
    s t -> TermType -> Maybe ExtendedTerm
```

4.3.4.2. *SimpleSubstitution.* Es existieren nun zwei Implementierungen von `SubstitutionBuildern`. Die zunächst entwickelte heißt `SimpleSubstitution`. Sie zeichnet sich dadurch aus, bei jeder Anwendung der Funktionen `substAnd` und `substOr` bereits eine flache Darstellung der Substitutionen in den beiden folgenden Datentypen zu berechnen:

```
newtype (PositionTerm t) =>
  SubstitutionSet t =
    SubstitutionSet [SubstitutionAtom t]
newtype (PositionTerm t) =>
  SubstitutionAlt t = SubstitutionAlt
    [SubstitutionSet t]
```

Dabei stellt die Struktur `SubstitutionAlt` eine Disjunktion der in ihr enthaltenen `SubstitutionSets` dar. Ein `SubstitutionSet` ist dagegen eine Konjunktion der in ihr enthaltenen `SubstitutionsAtome` und stellt damit eine Substitution selbst dar.

```
substAnd2 :: SimpleSubstitutionAlt
  -> SimpleSubstitutionAlt
  -> SimpleSubstitutionAlt
substAnd2
  (SubstitutionAlt dss1) (SubstitutionAlt dss2) =
  SubstitutionAlt [ SubstitutionSet (css1 ++ css2) |
    (SubstitutionSet css1) <- dss1,
    (SubstitutionSet css2) <- dss2 ]
substOr2 :: SimpleSubstitutionAlt
  -> SimpleSubstitutionAlt
  -> SimpleSubstitutionAlt
substOr2
  (SubstitutionAlt dss1) (SubstitutionAlt dss2) =
  SubstitutionAlt (dss1 ++ dss2)
```

Es stellt sich heraus, dass diese Form der Darstellung bei der Berechnung der Lösungen des linearen Kontextmatchings zu speicher- und rechenintensiv ist, da durch die unmittelbare Berechnung aller Lösungen das Sharing von Teillösungen in den konjugierten bzw. disjungierten Substitutionen verloren geht.

4.3.4.3. *SemiSolvedSubstitution.* Daher hält die Implementierung eines `SubstitutionBuilders` in Form der `SemiSolvedSubstitution` die originale Form der Disjunktionen und Konjunktionen aufrecht, bis es zu einem Aufruf der Funktion `generate` kommt. Erst dann wird die Substitution in eine flache Form gebracht.

```
data (PositionTerm t) =>
  SemiSolvedSubstitution t =
        SemiBot
      | SemiTop
      | SemiVar TermType t
      | SemiContextVar TermType t
```

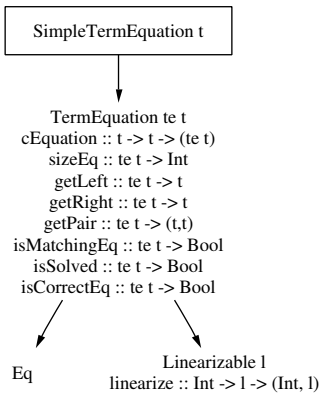


ABBILDUNG 4.3.13. SimpleTermEquation

```
      | SemiAnd (SemiSolvedSubstitution t)
                (SemiSolvedSubstitution t)
      | SemiOr (SemiSolvedSubstitution t)
               (SemiSolvedSubstitution t)
substAnd2 s1 s2 =
  if (isFlatBot s1 || isFlatBot s2) then SemiBot
    else SemiAnd s1 s2
substOr2 s1 s2 =
  if (isFlatBot s1 && isFlatBot s2) then SemiBot
    else SemiOr s1 s2
```

Bei der Generierung kommt es damit zur Auflösung der Disjunktionen und Konjunktionen, wie der folgende Codeausschnitt zeigt:

```
generate'' (SemiAnd s1 s2) =
  [ Substitution (css1 ++ css2) |
     (Substitution css1) <- (generate' s1),
     (Substitution css2) <- (generate' s2) ]
generate'' (SemiOr s1 s2) =
  (generate' s1) ++ (generate' s2)
```

Durch diese Lösung kommt es erst während der Generierung zu einer Duplizierung von gemeinsam genutzten Subtermen. Da der Aufrufer der Funktion `generate` diese jedoch sofort auswerten und konsumieren kann, kann er durch seine Verarbeitung den Platzbedarf steuern.

4.3.5. Termgleichungen und Unifikationsprobleme. Wir führen zwei weitere Typklassen ein, um Termgleichungen und Unifikationsprobleme darstellen zu können. Diese beiden Typklassen haben beide sehr simple Implementierungen, sind aber hilfreich für die Abstrahierung und Strukturierung des Programmes.

Eine `SimpleTermEquation` ist ein Paar von Termen mit einigen hilfreichen Funktionen wie in Abbildung 4.3.13 dargestellt.

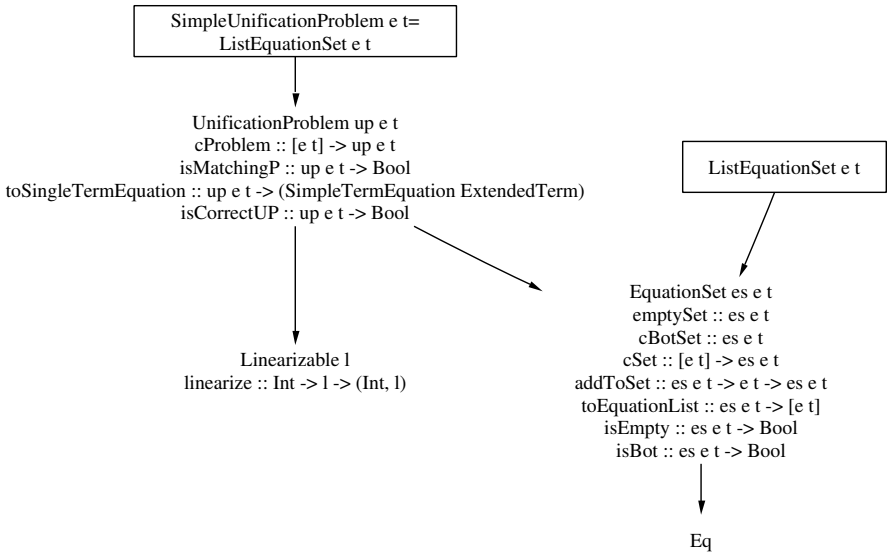


Abbildung 4.3.14. SimpleUnificationProblem

In Abbildung 4.3.14 sehen wir die Typklassenhierarchie ausgehend vom Datentyp `SimpleUnificationProblem`. Ein Unifikationsproblem also ist eine Menge von `TermEquation`s gemeinsam mit nützlichen Funktionen.

4.3.6. Testterme. Zu Testzwecken und um das Verhalten der implementierten Algorithmen zu studieren ist es notwendig, große Terme generieren zu können. Im Rahmen dieser Diplomarbeit wurden hierzu zwei Funktionalitäten entwickelt.

4.3.6.1. *Linear exponentielle Terme.* Es wurde eine Funktion zur Generierung linearer Kontextmatchingprobleme mit exponentiellem Ergebniswachstum gemäß Lemma 3.3.1 implementiert. Diese Probleme zeichnen sich dadurch aus, dass sie in Abhängigkeit von der Anzahl n der generierten Variablen eine exponentielle Anzahl möglicher Lösungen 2^n haben.

Die Kontextmatchingprobleme haben die Form

$$\{X_n\,[Y_n\,[X_{n-1}\,[Y_{n-1}\,[\ldots X_1\,[Y_1\,[h\,(a)]]]]]]] \approx g\,(h\,(g\,(h\,(\ldots g\,(h\,(a))))))\}$$

Beispiel 4.3.1. Wir geben eine Beispielanwendung der Funktion `linearExponential` an:

```
> :type linearExponential
linearExponential :: Int -> ContextUP
> linearExponential 2
[X2[Y2[h(X1[Y1[h(a())]])]] =? g(h(g(h(a()))))
```

Dieser Termgenerator wird primär zur Erzeugung großer Terme mit vielen Lösungen für den Test des Algorithmus zur Lösung linearer Kontextmatchingprobleme genutzt.

4.3.6.2. *Zufallstermgenerator.* Zum Test der Algorithmen zur Lösung des generellen Kontextmatching wurde ein randomisierter Termgenerator entwickelt, der Kontextmatchingprobleme mit genau einer Kontextmatchinggleichung generiert. Die Beschränkung auf nur eine Kontextmatchinggleichung ist keine Einschränkung, da gemäß Proposition 3.2.6 ein Kontextmatchingproblem mit mehreren Gleichungen stets in ein äquivalentes mit nur einer Gleichung überführt werden kann. Der umgekehrte Schritt kann mit einer Anwendung der `Dekomposition` gemäß Tabelle 3.4.1 ebenfalls vollzogen werden.

Das Package `Gersdorf.ContextMatching.Base.TermGenerator` beinhaltet den implementierten Termgenerator. Dieser bietet die folgende Schnittstelle an:

`generateUP`	`TermGeneratorParameter -> IO (ListEquationSet SimpleTermEquation ExtendedTerm)`
`generateUPn`	`TermGeneratorParameter -> Int -> IO [ListEquationSet SimpleTermEquation ExtendedTerm]`
`unsafeGenerateUP`	`TermGeneratorParameter -> ListEquationSet SimpleTermEquation ExtendedTerm`
`unsafeGenerateUPn`	`TermGeneratorParameter -> [ListEquationSet SimpleTermEquation ExtendedTerm]`

Die Funktionen `generateUP` und `generateUPn` operieren in der IO-Monade, da sie Zufallswerte zur Termgenerierung benötigen. Die unsafe-Varianten der beiden Funktionen sind per `unsafePerformIO` aus der IO-Monade herausgeholte Ergebnisse der beiden erstgenannten Funktionen.

Die Termgenerierung wird durch einen `TermGeneratorParameter` von außen gesteuert. Wir betrachten diesen Parameter:

```
data TermTypeParameter =
  TermTypeParameter {
    termTypeCount :: Int,
    termTypeWeight :: Weight
  }
```

Parameter	Bedeutung
`termSize`	Termgröße der rechten Termgleichungsseite in Anzahl Termknoten.
`variableSize`	Termgröße der linken Termgleichungsseite in Anzahl Termknoten.
`varParameter`	Parameterisierung für Variablen.
`cvarParameter`	Parameterisierung für Kontextvariablen.
`funcParameter`	Parameterisierung für Funktionen.
`holeParameter`	Parameterisierung für Löcher.
`maxOccurenceVar`	Maximale Anzahl von Variablen im generierten Term.
`maxOccurenceCVar`	Maximale Anzahl von Kontextvariablen im generierten Term.
maxArity	Maximale Stelligkeit von Funktionen.
`randomRangeF`	Funktion, die per Zufall einen Wert aus einem Wertebereich wählt.
`distributionF`	Funktion, die per Zufall r Punkte auf n Stellen verteilt.

Tabelle 4.3.3. TermGeneratorParameter

Parameter	Bedeutung
`termTypeCount`	Anzahl verschiedener Instanzen des Termtyps (z. B. Anzahl verschiedener Funktionen).
`termTypeWeight`	Gewicht des Termtyps. Bestimmt Häufigkeit, mit der z. B. eine Funktion gewählt wird.

Tabelle 4.3.4. TermTypeParameter

```
data TermGeneratorParameter =
  TermGeneratorParameter {
    termSize :: Int,
    variableSize :: Int,
    varParameter :: TermTypeParameter,
    cvarParameter :: TermTypeParameter,
    funcParameter :: TermTypeParameter,
    holeParameter :: TermTypeParameter,
    maxOccurenceVar :: Int,
    maxOccurenceCVar :: Int,
    maxArity :: Int,
    randomRangeF :: (Int, Int) -> IO Int,
    distributionF :: Int -> Int -> IO [Int]
  }
```

Die Bedeutung der einzelnen Parameter ist in den Tabellen 4.3.3 und 4.3.4 beschrieben.

Die Implementierung beruht auf Algorithmus 4.3.1. Sie verwendet eine `StateMonade` [**Jon1995**], die einerseits die genannte Parameterisierung

kapselt und andererseits über die Anzahl bereits generierter Variablen und Kontextvariablen buchführt, um den Parametern `maxOccurenceVar` und `maxOccurenceCVar` Rechnung tragen zu können.

In Algorithmus 3.4.1 wurde zur Vereinfachung von der Betrachtung der beiden Parameter `maxOccurenceVar` und `maxOccurcenceCVar` abgesehen.

Wie man sich leicht überzeugen kann, werden mit Hilfe des angegebenen Algorithmus stets lösbare Unifikationsprobleme erzeugt.

Beispiel 4.3.2. Beispielgenerierung eines Unifikationsproblems:

```
> let smallRTP = TermGeneratorParameter {
                   termSize = 12,
                   variableSize = 8,
                   varParameter =
                    TermTypeParameter
                      { termTypeCount = 2,
                        termTypeWeight = 20 },
                   maxOccurenceVar = 3,
                   cvarParameter =
                    TermTypeParameter
                      { termTypeCount = 3,
                        termTypeWeight = 20 },
                   maxOccurenceCVar = 2,
                   funcParameter =
                    TermTypeParameter
                      { termTypeCount = 5,
                       termTypeWeight = 50 },
                   holeParameter =
                    TermTypeParameter
                      { termTypeCount = 1,
                       termTypeWeight = 10 },
                   maxArity = 2,
                  randomRangeF = randomRIO,
                 distributionF = distributeIO }
> let up = unsafeGenerateUP smallRTP
> up
[f2(f3(f4(f1(f3(v0)))),X2[f0()]) =?
f2(f3(f4(f1(f3(f1(f0()))))),f2(f0(),f1(f1(f0()))))]
> solveGCM paperStrategy up
[ { <f1(f0())/v0>, <f2(f0(),f1(f1(_)))/X2> },
{ <f1(f0())/v0>, <f2(_,f1(f1(f0())))/X2> } ]
```

4.4. Lineares Kontextmatching

4.4.1. Überblick. Die Implementierung des linearen Kontextmatchings ist die Umsetzung des in Algorithmus 3.4.1 beschriebenen Verfahrens. Im Rahmen der Diplomarbeit wurde darüber hinaus auch der vereinfachte Entscheidungsalgorithmus 3.4.2 separat implementiert. Man kann das

Algorithmus 4.3.1 Termgenerator

```
berechneTerm(r:Anzahl Termknoten) {
  r == 1: * Wähle zufällig ein Loch, eine Variable
            oder eine Funktion mit Stelligkeit 0
  r > 1:  * Wähle zufällig eine Funktion mit
            Stelligkeit in [1,(r-1)] oder eine
            Kontextvariable
          * Sei a die Stelligkeit des gewählten
            Termtyps
          * Verteile (r-1) Knoten auf die
            Funktionsparameter [p1,..,pa]
          * Berechne rekursiv berechneTerm(pi)
            für jeden Parameter pi
}
berechneGrundTerm(r:Anzahl Termknoten) {
  r == 1: * Wähle zufällig eine Funktion mit
            Stelligkeit 0
  r > 1:  * Wähle zufällig eine Funktion mit
            Stelligkeit in [1,(r-1)]
          * Sei a die Stelligkeit des gewählten
            Termtyps
          * Verteile (r-1) Knoten auf die
            Kindparameter [p1,..,pa]
          * Berechne rekursiv
            berechneGrundTerm(pi) für jeden
            Parameter pi
}

1) Berechne zunächst linke Seite:
   lt = berechneTerm(variableSize)
2) Zähle nun die Vorkommen einzelner
   Variablen und Kontextvariablen in lt
3) verteile (termSize-variableSize)
   gemäß der Vorkommensverteilung auf
   die einzelnen Variablen und Kontextvariablen
4) Rufe für jede in lt vorkommende Variable
   und Kontextvariable berechneGrundTerm
   auf mit der in 3) zugewiesenen Termgröße
5) Substituiere in lt die Variablen und
   Kontextvariablen durch die in 4) berechneten
   Terme und erhalte damit rt
6) Gib {lt ≈ rt} zurück.
```

Entscheidungsproblem auch auf einfache Weise durch den ersten Algorithmus lösen. Da dieser jedoch die Lösungsmengen berechnen muss, ist er bei Berechnung aller Lösungen um Faktor n speicherintensiver als der vereinfachte Algorithmus. Darüber hinaus ist die einfache boolesche Logik,

die der Entscheidungsalgorithmus benötigt, wesentlich performanter als die mengenorientierte Logik der Substitutionen.

4.4.2. Dynamisches Programmieren. Algorithmus 3.4.1 verwendet die Technik des dynamischen Programmierens. Diese Technik zeichnet sich dadurch aus, dass sie, ähnlich dem Divide and Conquer-Ansatz, ein Gesamtproblem in mehrere Teilprobleme unterteilt. Um mehrfach benötigte Teillösungen nicht mehrfach berechnen zu müssen, werden sie in einer Tabelle gespeichert.

Wir verwenden eine Funktion höherer Ordnung, um das dynamische Programmieren allgemein zu beschreiben und verwenden hierzu den Ansatz aus [**Rabh1999**, 9.2]. Das Modul `Gersdorf.Algorithms.Dynamic` enthält die Implementierung.

Wir definieren zunächst den Datentyp für die Lösungstabelle, die die Lösungen der Subprobleme enthalten soll:

```
newtype (Ix it) =>
  DynTable it et = DynTable (Array it et)
```

Dabei ist `et` der Typ der Lösung. Da in Haskell Arrays jedoch normalerweise unveränderlich sind, d. h. die Inhalte der einzelnen Zellen bereits bei Erstellung des Arrays bekannt sein müssen, stellt sich die Frage, wie man an diese Inhalte kommen soll, da man sie zunächst berechnen muss. Hier kommt die verzögerte Auswertung zum Tragen: Man legt einfach eine noch unausgewertete Anwendung der Funktion, die die entsprechende Sublösung berechnet, auf die richtigen Argumente in jede Zelle. Genau dies tut die Funktion `dynamic`:

```
dynamic :: (Ix it) =>
  (DynTable it et -> it -> et)
  -> (it, it)
  -> DynTable it et
dynamic compute bnds = t
  where t =
    DynTable (array bnds
      (map (\coord -> (coord, compute t coord))
      (range bnds)))
```

Die Funktion `dynamic` erhält eine Funktion `compute`, die für einen bestimmten Index (das heißt für das mit diesem Index bezeichnete Subproblem) eine Lösung berechnet. Eine Anwendung von `compute` auf die entsprechenden Argumente wird in jede Zelle eingetragen.

Die Lösung des Gesamtproblems oder eines bestimmten Subproblems ist damit denkbar einfach: man liest aus der gewünschten Zelle des Arrays:

```
dynamicSolve :: (Ix it) =>
  DynTable it et
  -> it
  -> et
dynamicSolve (DynTable arr) coord = arr ! coord
```

4.4.3. Implementierung. Die Tatsache, dass zum dynamischen Programmieren in der Form, wie im vorigen Abschnitt vorgestellt, ein Array notwendig ist und eine Funktion, die auf den Array-Indizes operiert, macht deutlich, dass eine Termdarstellung der Form `SimpleTerm` nicht verwendet werden kann, um lineares Kontextmatching zu implementieren. Wir benötigen eine Termdarstellung, die Mitglied der Typklasse `PositionTerm` ist, damit wir Subterme gezielt adressieren können.

Bei unserem Ansatz zur Lösung linearer Kontextmatchingprobleme $\{s \approx t\}$ stellen wir eine Tabelle der Dimensionen $(|s|,|t|)$ auf, wobei in Zelle (1,1) das Problem $\{s \approx t\}$ steht. In Zelle (i,j) steht damit jeweils ein Subproblem. Bezeichnen wir mit $((i,j),p)$ das in Zelle (i,j) befindliche Kontextmatchingproblem p und weiterhin mit $t|_i$ den mit i adressierten Subterm von t, so läßt sich der Inhalt des Arrays wie folgt beschreiben:

$$\{((i,j),p)\,|p = \{s|_i \approx t|_j\}, 1 \leq i \leq |s|, 1 \leq j \leq |t|\}$$

Damit wird die Implementierung denkbar einfach. Wir geben die Berechnungsfunktion in Algorithmus 4.4.1 an.

4.5. Generelles Kontextmatching

4.5.1. Überblick. Im Rahmen der Diplomarbeit wurde der Algorithmus zur Lösung genereller Kontextmatchingprobleme, wie er in Kapitel 3.4.2 vorgestellt wurde, in Haskell implementiert. Dabei wurden alle dort vorgestellten Regeln umgesetzt, mit Ausnahme der Berechnung korrespondierender Positionen mittels diophantischer Gleichungen (Kapitel 3.4.4.2). Zusätzlich wurde jedoch ein Spezialfall korrespondierender Positionen mittels diophantischer Gleichungen realisiert: Korrespondierende Funktionssymbole. Die implementierten Regeln sind in Tabelle 4.5.1 noch einmal namentlich aufgeführt.

4.5.2. Backtracking. Der Transformationsalgorithmus verwendet u. a. Regeln, die einen don't know-Nichtdeterminismus einführen, z. B. Variablen-Split. Dies bedeutet, dass bei der Anwendung einer solchen Regel verschiedene Regelanwendungsschritte ausgewählt werden können. Jedoch führt nicht jeder dieser Schritte tatsächlich zu einer Lösung des Problems. Stellt der Algorithmus fest, dass er in einer Sackgasse angekommen ist, so muss er in seinem Lösungsweg zurückgehen und einen anderen Schritt ausprobieren.

4.5.2.1. *Einfacher Backtracking-Algorithmus.* Dies bedeutet, dass unser Algorithmus in systematischer Art und Weise nach möglichen Lösungen des Kontextmatchingproblems suchen muss. Allgemein ausgedrückt benötigen wir einen Algorithmus, der für Probleme mit den folgenden Charakteristika geeignet ist:

- Es existiert ein Raum möglicher Lösungen. Die potentiellen Lösungen werden als Knoten bezeichnet.

Algorithmus 4.4.1 Berechnungsfunktion für lineares Kontextmatching

```
computeLCM :: HeapTerm -> HeapTerm
  -> DynTable (Int, Int) SubstitutionType
  -> (Int, Int) -> SubstitutionType
computeLCM left right dt (i, j) =
  simplify $ process leftSub rightSub
  where
  leftSub = getAt i left
  rightSub = getAt j right

  process :: HeapTerm -> HeapTerm -> SubstitutionType
  process t1 t2 | isHole t1 && isHole t2 = substTop
     | isVariable t1 && not (isContext t2) =
         substVar leftSub rightSub
     | isFunction t1 && isFunction t2
       && (getName t1 == getName t2) &&
       (arity t1 == arity t2) =
         subSolution leftSub rightSub
     | isContextVariable t1 =
         contextSolution leftSub rightSub
     | otherwise = substBot

  contextSolution :: HeapTerm -> HeapTerm
    -> SubstitutionType
  contextSolution s t =
   substOr $ map (\subt -> (sol subt)) (toList t)
    where
    sol subt = substAnd ((cVar subt):
      [dynamicSolve dt
      (getAdress contextChild, getAdress subt)])
    contextChild = head $ getChildren s
   cVar subt = substContextVar s t (getAdress subt)

  subSolution :: HeapTerm -> HeapTerm
    -> SubstitutionType
  subSolution father1 father2 = substAnd
   (map ( \(ht1,ht2) ->
     dynamicSolve dt (getAdress ht1, getAdress ht2))
   (zip (getChildren father1) (getChildren father2)))
```

- Es existiert eine Funktion, die ausgehend von einem Knoten die möglichen Schritte zu einem oder auch mehreren anderen Knoten beschreibt.
- Es existiert ein initialer Knoten.
- Es existieren ein oder mehrere Lösungsknoten.

Probleme dieser Problemklasse können mit einem sogenannten Backtracking-Algorithmus gelöst werden.

Regel	Kapitel	Modul
Dekomposition	3.4.2	`Decompose`
Kollision	3.4.2	`Decompose`
Lochdekomposition	3.4.2	`HoleDecompose`
Multikontext-Kollision	3.4.2	`MultiContextCollision`
Termverschmelzung	3.4.2	`SolvedDummy`
Kontextverschmelzung	3.4.2	`SolvedDummy`
Kontextelimination	3.4.2	`ContextElimination`
Variablensplit	3.4.2	`VariableSplit`
Split	3.4.3	`Split`
Bodendekomposition	3.4.4.3	`BottomDecompose`
Bodenkollision	3.4.4.3	`BottomDecompose`
Konstanten-Elimination	3.4.4.5	`ConstantEliminate`
Multikontext-Dekomposition	3.4.4.3	`MultiContextDecompose`
Multikontext-Kollision	3.4.4.3	`MultiContextDecompose`

Alle Regeln befinden sich im Paket
`Gersdorf.ContextMatching.General.Rules`.
Darüber hinaus wurden die folgenden Methoden zur Berechnung korrespondierender Positionen umgesetzt. Diese befinden sich im Paket `Gersdorf.ContextMatching.General.CorrespondingPositions`.

Methode	Kapitel	Modul
Linearisierung	3.4.4.1	`CorrespondingPosLinear`
Korrespondierende Lochpfade	3.4.4.4	`CorrespondingPosHoles`
Korrespondierende Funktionssymbole	3.4.4.2	`CorrespondingPosFunctions`

Tabelle 4.5.1. Implementierte Regeln

Wir verwenden eine abgewandelte Version der Funktion höherer Ordnung für Backtracking-Algorithmen aus [**Rabh1999**] und stellen zunächst die ursprüngliche Version kurz vor.

Im Paket `Gersdorf.Algorithms.Backtracking` wird die folgende Funktion definiert:

```
solveBacktrack :: (Eq node) =>
  (node -> [node])
  -> (node -> Bool)
  -> node
  -> [node]
```

Wir beschreiben die Parameter der Funktion. Die Funktion ist polymorph über Knotentypen und benötigt eine Transitionsfunktion, die ausgehend von einem aktuellen Knoten mögliche neue Knoten berechnet. Dabei ist die Menge neuer Knoten als alternativ anzusehen. Die zweite Funktion gibt an, ob ein Knoten eine Lösung ist oder nicht (Prädikatfunktion). Falls ja, wird

Algorithmus 4.5.1 Backtracking nach [Rabh1999]

```
solveBacktrack succ goal cur
  = search (push cur (emptyStack::ListStack node))
    where
    search st
      | isStackEmpty st = []
      | goal (top st)   = top st:(search (pop st))
      | otherwise       =
          let next = top st
          in  search (foldr push (pop st) (succ next))
```

dieser Knoten in die Ergebnisliste eingetragen (Rückgabewert). Der dritte Parameter ist der Startknoten.

Der Algorithmus funktioniert so, dass als Initialisierung der Startknoten auf einen Stack gelegt wird. Danach wird auf dem Stack operiert. Der oberste Knoten wird vom Stack genommen und nun als aktueller Knoten bezeichnet. Es wird mit Hilfe der Prädikatfunktion geprüft, ob der aktuelle Knoten eine Lösung ist. Falls ja, wird dieser an die Lösungsliste angehangen und die Operation mit dem Stack fortgesetzt. Im anderen Falle wird der aktuelle Knoten der Transitionsfunktion übergeben. Diese berechnet eine Menge weiterer Knoten. Die Menge wird auf den Stack gelegt und der Algorithmus beginnt von vorne. Der Algorithmus läuft solange, bis der Stack leer ist.

Die Implementierung in Haskell ist in Algorithmus 4.5.1 abgebildet.

Zur Implementierung unseres Algorithmus verwenden wir aus zwei Gründen eine erweiterte Form des vorgestellten Verfahrens. Zum einen benötigen wir statistische Informationen über die Regelabarbeitung des Algorithmus. Dabei sollen sowohl die Lösungsknoten als auch der gesamte Transformationsablauf berücksichtigt werden, also auch solche Knotentransformationen, die zu keiner Lösung führen, die aber dennoch die Laufzeit beeinflussen. Insbesondere möchten wir wissen, wie oft welcher Transformationstyp angewendet wurde, um alle Lösungen eines Problems zu berechnen.

Zum anderen führt die Regel Split in Verbindung mit den beiden Verfahren zur Berechnung korrespondierender Probleme Linearisierung und Korrespondierende Lochpfade einen bedingten don't know-Nichtdeterminismus ein. Wir definieren zunächst verschiedene Typen des Determinismus und stellen dann einen Datentyp vor, der diese abbilden kann. Im Anschluss daran werden wir den erweiterten Backtrackingalgorithmus skizzieren.

Definition 4.5.1. Eine Transformation verhält sich *deterministisch*, wenn sie lediglich einen Knoten als Ausgabe berechnet. Sie verhält sich *don't care-nichtdeterministisch*, wenn sie eine Menge von Knoten als Ausgabe berechnet, von denen jeder Knoten zur gleichen Lösungsmenge führt. Die Transformation verhält sich *don't know-nichtdeterministisch*, wenn sie eine Menge von Knoten berechnet, deren Knoten zu unterschiedlichen Lösungsmengen führen können. Sie verhält sich *bedingt don't know-nichtdeterministisch*, wenn sie eine Menge von Knoten berechnet, deren Knoten entweder zur vollständigen Lösungsmenge oder in eine Sackgasse ohne Lösungsbeitrag führen.

Bemerkung. Bei einer don't care-nichtdeterministischen Transformation ist es also für die Korrektheit und Vollständigkeit der weiteren Lösungsberechnung gleichgültig, welcher Knoten gewählt wird. Bei einer don't know-nichtdeterministischen Transformation kann Backtracking notwendig sein, da die Ergebnisknotenmenge zu unterschiedlichen Lösungen führen kann. Bei einer bedingten don't know-nichtdeterministischen Transformation kann ein beliebiger Knoten aus der Ergebnisknotenmenge gewählt werden, der zu einer Lösung führt. Hat man einen solchen Knoten gefunden, so können die restlichen Knoten unbetrachtet bleiben, da diese zu keiner anderen als zu der bereits gefundenen Lösung führen.

4.5.2.2. `BacktrackingChoice`. Um deterministische sowie don't know-nichtdeterministische Regeln modellieren zu können, führen wir den Datentyp `BacktrackingChoice` (im Paket `Gersdorf.Data.BacktrackingChoice`) ein. Dieser ist definiert als:

```
data BTType = BTOne | BTMany | BTDet
  deriving Eq

data BacktrackingChoice a =
  BTChoice {
    btType    :: BTType,
    btEntries :: [a]
  }
```

Tabelle 4.5.2 erläutert die unterschiedlichen Typen. Offensichtlich ist `BacktrackingChoice` eine Instanz von `Functor`:

```
instance Functor BacktrackingChoice where
  fmap f b = b { btEntries = map f (btEntries b) }
```

Beinhaltet `BacktrackingChoice` mit dem Typ `BTMany` mehr als eine Alternative, so sind dies Alternativen in einem don't know-nichtdeterministischen Sinne: Nachdem Alternative 1 vollständig berechnet wurde, ist die Berechnung also in jedem Fall mit Alternative 2 fortzusetzen, da diese einen weiteren Lösungsbeitrag liefern könnte. Ist dagegen der Typ `BTOne`, so liegen bedingt don't know-nichtdeterministische Alternativen vor. Führt Alternative 1 in eine Sackgasse, so ist die Suche mit Alternative 2 fortzusetzen. Führt sie hingegen zu einer Lösung, so können die anderen Alternativen unbeachtet bleiben. Im Falle des Typs `BTDet` kann ein beliebiger der vorliegenden Knoten zur Berechnung ausgewählt werden. In unserer Implementierung wählen wir stets den ersten Knoten, da die Auswahl der Knoten an anderer Stelle vorgenommen wird (siehe Kapitel 4.5.3.4).

4.5.2.3. *Erweiterter Backtracking-Algorithmus.* Wir kommen nun zu unserer Implementierung eines Backtracking-Algorithmus, der neben Determinismus, don't care-Nichtdeterminismus und don't know-Nichtdeterminismus auch bedingten don't know-Nichtdeterminismus unterstützt. Wir geben zunächst einen Algorithmus in Pseudocode an (Algorithmus 4.5.2).

Konstruktor	Bedeutung
BTOne	Menge bedingt don't know-nichtdeterministischer Knoten
BTMany	Menge don't know-nichtdeterministischer Knoten
BTDet	Menge don't care-nichtdeterministischer Knoten (falls nur ein Knoten gegeben ist, ist dies ein deterministischer Knoten)

TABELLE 4.5.2. `BacktrackingChoice`

Wir versehen in dem Algorithmus jeden Knoten vom Typ `BacktrackingChoice` mit einem Marker, der angibt, ob dieser Knoten bereits zu einer Lösung beigetragen hat. Das ist für die Behandlung von `BTOne`-Knoten von Bedeutung, die den bedingten don't know-Nichtdeterminismus abbilden.

Da wir, abhängig von unserer Aufgabenstellung, nur die Lösungsknoten berechnen wollen oder aber zusätzlich statistische Informationen über die Anzahl angewendeter Transformationen, ändern wir den Typ der Backtracking-Funktion so, dass sie sämtliche Knoten auf dem Weg der Berechnung zurückgibt und eine vorgelagerte Fassadenfunktion entscheidet, was mit den Knoten geschieht, die keine Lösungsknoten sind. Dies bedeutet keinen Mehraufwand, da die Knotenmenge ohnehin erzeugt werden muss. Es ergibt sich der allgemeine Typ:

```
solveBacktrackR2 ::
    (node -> BacktrackingChoice node)
    -> (node -> Bool)
    -> node
    -> [node]
```

Gegenüber `solveBacktrack` hat sich äußerlich nur der Typ der Transformationsfunktion so geändert, dass diese nun auch bedingt don't know-nichtdeterministische Transformationsfunktionen unterstützt. Der Ergebnistyp hat sich nach außen hin nicht geändert, wohl aber seine Semantik: Beinhaltete die erzeugte Liste bei `solveBacktrack` noch ausschließlich die Zielknoten, so gibt `solveBacktrackR2` auch alle anderen betrachteten Knoten auf dem Weg der Ergebnisberechnung zurück.

Im Wesentlichen ist `solveBacktrackR2` also die Umsetzung von Algorithmus 4.5.2. Die Funktion ist als Algorithmus 4.5.3 abgebildet.

Die andere Änderung ist die Angabe einer Transformationsfunktion, die einen Knoten vom Typ `node` in einen Knoten vom Typ `g` transformiert. Diese Funktion wird nur auf Lösungsknoten vor der Ausgabe der Ergebnisliste angewandt und ermöglicht es, für die Arbeitsknoten andere Typen zu nutzen als für die Ergebnisknoten.

Wir geben die modifizierte Funktion in Algorithmus 4.5.3 an.

Die eigentliche Backtrackingfunktion, die nur die Lösungsknoten - auf einen anderen Typ transformiert - zurückgibt, ist dann entsprechend einfach zu implementieren:

Algorithmus 4.5.2 Erweiterter Backtrackingalgorithmus

```
0) Lege Startknoten s als BTMany([s], False)-
   Multiknoten auf den Stack
1) Falls Stack leer: Ende.
2) Nimm Multiknoten N vom Stack.
     WENN (N nichtleer) DANN
       WENN (N vom Typ BTOne
             && N hat Marker True) DANN
         propagateStackSolution
         goto 1)
       SONST
         Nimm Knoten n aus N
         WENN (n Zielknoten) DANN
           n zu Ergebnis zufügen
           Marker von N auf True setzen
           N auf Stack
           goto 1)
         SONST
           N auf Stack
           Nachfolgeknotenmenge von n berechnen
           Nachfolgeknotenmenge mit Marker False
             auf Stack legen
           goto 1)
     SONST {- N ist leer -}
       WENN (N hat Marker True) DANN
         propagateStackSolution
         goto 1)
       SONST
     goto 1)

propagateStackSolution:
  WENN (Stack nichtleer) DANN
    nimm N2 vom Stack
    Setze Marker von N2 auf True
    lege N2 auf Stack
```

```
solveBacktrackSolution ::
    (node -> BacktrackingChoice node)
    -> (node -> Bool)
    -> (node -> g)
    -> node
    -> [g]
solveBacktrackSolution
  getSucc isGoal transformGoal cur =
    map transformGoal
    $ filter isGoal
    $ solveBacktrackR2 getSucc isGoal cur
```

Algorithmus 4.5.3 erweiterter Backtrackingalgorithmus (Haskell)

```
solveBacktrackR2 getSucc isGoal cur =
    search (push startNode
           (emptyStack::ListStack
             (BacktrackStackNode node)))
    where
    startNode = BacktrackStackNode {
              bsnChoice = cBacktrackingDet cur,
                  bsnHasSolution = False }

    search st
        | isStackEmpty st = []
        | noChoice curBT  =
            let nextStack =
              if (bsnHasSolution curStackChoice)
                then propagateSolution (pop st)
                else (pop st)
            in  search nextStack
        | (isBTOne curBT || isBTDet curBT)
          && (bsnHasSolution curStackChoice) =
          let nextStack = propagateSolution (pop st)
            in  search nextStack
        | isGoal curNode =
            let updStackNode =
              curStackChoice {
                bsnChoice = popChoice curBT,
                bsnHasSolution = True  }
          in  curNode:(search (push updStackNode st))
        | otherwise     =
            let nextBT          = getSucc curNode
                updStackChoice  =
                  curStackChoice {
                    bsnChoice = popChoice curBT }
                nextStackChoice =
                    BacktrackStackNode {
                      bsnChoice = nextBT,
                      bsnHasSolution = False }
          in  curNode:(search (push nextStackChoice
                              $ push updStackChoice
                                    $ pop st))
        where
        curStackChoice = top st
        curBT          = bsnChoice curStackChoice
       curNode       = fromJust $ topChoice curBT
        propagateSolution st =
          topUpdate
          (\bsn -> bsn { bsnHasSolution = True }) st

data BacktrackStackNode node =
    BacktrackStackNode {
        bsnChoice :: BacktrackingChoice node,
        bsnHasSolution :: Bool
   }
```

`solveBacktrackSolution` hat einen zusätzlichen Parameter vom Typ `(node -> g)` – eine Funktion, die Lösungsknoten in geeigneter Weise transformiert. Dementsprechend gibt `solveBacktrackSolution` eine Liste von Lösungsknoten vom Typ `g` zurück.

Neben `solveBacktrackSolution` haben wir eine Funktion implementiert, die statistische Informationen während des Ablaufs der Backtrackingfunktion berechnet. Wir geben hier lediglich den Typ an; die Implementierung findet sich im Anhang (Kapitel 7.1.1).

```
solveBacktrackStatistic :: (Ord transformType) =>
  (node -> BacktrackingChoice node)
   -> (node -> Bool)
   -> (node -> transformType)
    -> node
    -> BacktrackingStatistic transformType

data (Ord transformType) =>
  BacktrackingStatistic transformType =
    BacktrackingStatistic {
        btsTransformMap ::
          !(TreeFiniteMap transformType Int),
        btsSteps        :: !Int,
        btsNoSolutions  :: !Int
    }
```

Die Striktheitsannotationen im Konstruktor sind notwendig, um bei umfangreichen Problemberechnungen den Speicherbedarf niedrig zu halten.

4.5.3. Implementierung. Wir beginnen mit einem groben Überblick über die Verarbeitungsweise (siehe Abbildung 4.5.1). Diese beginnt mit einem ungelösten Kontextmatchingproblem. Wie bereits in Kapitel 4.3.5 ausgeführt, sind Unifikationsprobleme Instanzen der Typklasse `UnificationProblem` und damit Instanzen der Klasse `EquationSet`. Ein `EquationSet` ist nichts anderes als eine Menge von Gleichungen. Wir operieren nun im gesamten Algorithmus auf einem Paar aus einer Menge von Gleichungen (den ungelösten Gleichungen) und einer Menge von Substitutionen (den gelösten Gleichungen). Dafür existiert eine Subklasse von `EquationSet`: `EnvEquationSet`[2].

Zu Beginn der Verarbeitung besteht das `EnvEquationSet` aus einer Menge von (ungelösten) Gleichungen sowie einer leeren Menge von Substitutionen. Es beschreibt den aktuellen Zustand des Algorithmus und ist daher der Knotentyp für den Backtracking-Algorithmus. Die Prädikatfunktion, die entscheidet, ob ein Knoten eine Lösung ist, prüft lediglich im aktuellen `EnvEquationSet`, ob es keine ungelösten Gleichungen mehr gibt.

[2] `Env` steht für Environment

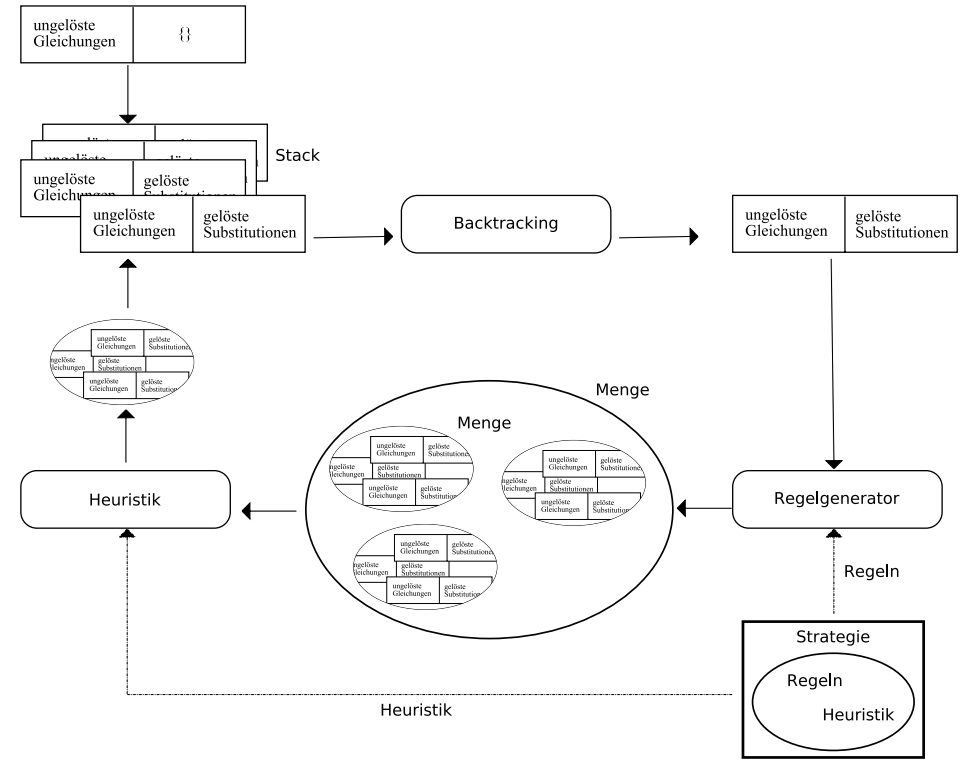


Abbildung 4.5.1. Verarbeitungsüberblick

Die Verhaltensweise des Algorithmus kann von außen durch eine *Strategie* parameterisiert werden. Eine Strategie besteht aus einer Menge von Regeln sowie einer Heuristik. Eine *Regel* berechnet ausgehend von einem `EnvEquationSet` eine oder mehrere neue `EnvEquationSet`s, die der Lösung näher kommen. Dabei implementiert jede der Regeln eine oder mehrere Transformationsregeln gemäß Tabelle 4.5.1. Eine Regel kann abhängig vom zu bearbeitenden `EnvEquationSet` anwendbar sein oder nicht.

Eine *Heuristik* ist eine Funktion, die aus einer Menge anwendbarer Regeln die tatsächlich anzuwendende auswählt. Sie bestimmt somit die Reihenfolge der Regelanwendung und kann entscheidend die Laufzeit der Gesamtberechnung beeinflussen.

Das initiale `EnvEquationSet` wird auf den Stack gelegt. Damit beginnt die eigentliche Verarbeitungsschleife. Ist das aktuelle `EnvEquationSet` eine Lösung, so wird diese ausgegeben. Ist es keine Lösung, so wird der Regelgenerator verwendet, der jede Regel aus der Strategie auf das `EnvEquationSet` anwendet.

Jede Regel kann eine Menge von `EnvEquationSet`s produzieren. Daher gibt der Regelgenerator eine Menge von Mengen von `EnvEquationSet`s aus. Aus diesen wählt die Heuristik eine Menge aus, die wiederum auf den Stack gelegt wird. Damit beginnt der Algorithmus von Neuem, bis der Stack leer ist.

Wichtig ist, zu beachten, dass Haskell eine verzögert auswertende Sprache ist. Dies bedeutet, dass der Regelgenerator zwar so programmiert ist, als ob er sämtliche möglichen Regelanwendungen direkt bei der ersten Anwendung ausgäbe. Die tatsächliche Berechung der Ergebnisse der Regelanwendung erfolgt jedoch erst bei Anforderung derselben.

4.5.3.1. *Voraussetzungen.* Vor der nun folgenden Diskussion der für die Umsetzung des Transformationsalgorithmus wesentlichen Datenstrukturen und Funktionen führen wir einige Basisdatenstrukturen ein.

4.5.3.1.1. `EnvEquationSet`. Ein `EnvEquationSet` ist auf der einen Seite ein `EquationSet` und beinhaltet als solches eine Menge ungelöster Kontextgleichungen. Auf der anderen Seite ist es auch ein `SubstitutionSet` und beinhaltet als solches Substitutionen von bereits gelösten Variablen und Kontextvariablen.

```
class (EquationSet es eq t, SubstitutionSet (es eq) t) =>
  EnvEquationSet es eq t where
    scanUpdate ::
      es eq t
      -> (eq t
        -> [BacktrackingChoice (RuleResult eq t)])
      -> [BacktrackingChoice (es eq t)]
```

Die Funktion `scanUpdate` nimmt ein `EnvEquationSet` sowie eine Funktion `f` als Parameter. Das `EnvEquationSet` beinhaltet die Menge der ungelösten Kontextgleichungen sowie die Menge der Substitutionen bereits gelöster Variablen und Kontextvariablen. `scanUpdate` iteriert über jede ungelöste Kontextgleichung und übergibt diese der Funktion `f`.

`f` ist eine Funktion, die eine Regel auf eine Termgleichung anwendet. Sie gibt eine Menge von `BacktrackingChoices` zurück. Jede Alternative einer `BacktrackingChoice` ist ein mögliches Ergebnis einer Regelanwendung (`RuleResult`). Eine `BacktrackingChoice` modelliert also den möglichen don't know-Nichtdeterminismus einer Regel.

Der mögliche don't care-Nichtdeterminismus wird durch die Liste modelliert.

Beispiel. Gibt die Funktion f eine Liste

```
[BTChoice {btType=BTDet, btEntries=[rr1, rr2]},
 BTChoice {btType=BTMany, btEntries=[rr3, rr4]},
 BTChoice {btType=BTOne, btEntries=[rr4, rr5]]
```

zurück, so bedeutet dies, dass in einem don't care-Sinne ein beliebiges der Regelresultate `rr1` oder `rr2`, in einem don't know-Sinne die Regelresultate `rr3` oder `rr4` oder in einem bedingt don't know-Sinne die Resultate `rr5` oder `rr6` ausgewählt werden können. Dies bedeutet: hat man sich einmal für `rr1` oder `rr2` entschieden, so bleibt diese Entscheidung bestehen, d. h. es gibt keinen Weg zurück. Entscheidet man sich dagegen für `rr3` oder `rr4`, so kann man - falls sich diese Entscheidung als falsch erweisen sollte oder man weitere Lösungen berechnen möchte - die Berechnung mit `rr4` oder

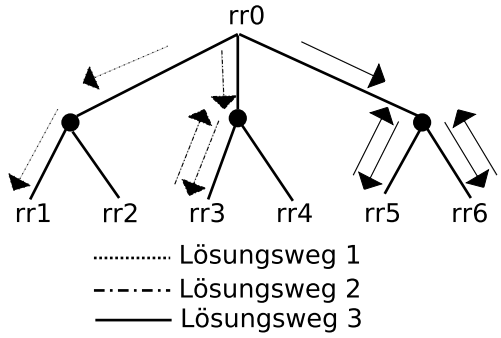


Abbildung 4.5.2. Backtracking

`rr3` neu aufsetzen. Wählt man `rr5` oder `rr6`, so wird die Berechnung mit `rr5` nur dann versucht, falls `rr4` keinen Lösungsbeitrag zur Gesamtlösung beifügt. Abbildung 4.5.2 illustriert diese Darstellung.

`scanUpdate` konsolidiert die Anwendung von `f` auf jede Termgleichung des `EnvEquationSets` und gibt eine Liste don't care-alternativer don't know-Regelanwendungsresultate in Form von `EnvEquationSets` zurück. Jedes der `EnvEquationSets` ist das Ergebnis einer Regelanwendung, d. h. sowohl die Menge der ungelösten Gleichungen als auch die Menge der Substitutionen gelöster Variablen und Kontextvariablen kann geändert worden sein.

Eine hilfreiche Typklasse ist `EnvPosEqSet`, die verschiedene Typklassen in sich vereinigt:

```
class (EnvEquationSet es eq t,
  PositionTerm t tp,
  AnalysisTerm t,
  TreeExtractor t,
  TermBuilder t,
  SubstitutionGenerator (es eq t) t)
  => EnvPosEqSet es eq t tp
```

4.5.3.2. *Strategien.* Wir beginnen den tieferen Einblick in die Arbeitsweise mit einem Blick auf Strategien. Eine Strategie ist wie folgt definiert:

```
data (EnvPosEqSet es eq t tp) =>
  Strategy es eq t tp =
    Strategy {
      strategyHeuristic :: HeuristicFunction,
      strategyRules :: [Rule es eq t tp]
    }
```

Eine Strategie ist also eine Menge von Transformationsregeln gemeinsam mit einer Heuristikfunktion, die unter einer Menge von anwendbaren Transformationsregeln eine auswählt. Wie sie dies tut, kann damit von außen bestimmt werden.

Algorithmus 4.5.4 applyRule und RuleApplicationResult

```
data (EnvPosEqSet es eq t tp) =>
  RuleApplicationResult es eq t tp =
  RuleApplicationResult {
    appliedRule :: Rule es eq t tp,
   appliedResult :: [BacktrackingChoice (es eq t)]
}
applyRule :: (EnvPosEqSet es eq t tp) =>
  es eq t
  -> Rule es eq t tp
  -> RuleApplicationResult es eq t tp
applyRule eset r =
  let (RuleFunction rf) = ruleF r
  in  RuleApplicationResult {
        appliedRule   = r,
       appliedResult = filter (not . noChoice) (rf eset)
     }
```

4.5.3.3. *Regeln.* Eine Transformationsregel wie sie in Tabelle 4.5.1 angegeben ist, wird durch einen Datentyp `Rule` (Paket `Gersdorf.Context-Matching.General.Rule`) beschrieben:

```
data (EnvPosEqSet es eq t tp) =>
  Rule es eq t tp =
    Rule {
      ruleName :: String,
      ruleF :: RuleFunction es eq t tp
    }
newtype (EnvEquationSet es eq t, PositionTerm t tp) =>
  RuleFunction es eq t tp =
    RuleFunction (es eq t ->
                  [BacktrackingChoice (es eq t)])
```

Eine Regel besteht aus einem Namen sowie einer Funktion, die ein `EnvEquationSet` (also ein Paar aus ungelösten Gleichungen und Substitutionen bereits gelöster Variablen und Kontextvariablen) in eine Liste alternativer `BacktrackingChoices` transformiert. Ausgehend von einem Zustand während der Berechnung - dem aktuellen `EnvEquationSet` - berechnet die Regel alternative - möglicherweise don't know-nichtdeterministische - neue `EnvEquationSets`.

Wir geben in Algorithmus 4.5.4 die Funktion `applyRule` an, die eine Regel auf ein `EnvEquationSet` anwendet und ein Regelresultat berechnet. `applyRule` filtert bereits alle ergebnislosen Regelanwendungsversuche aus der Resultatmenge.

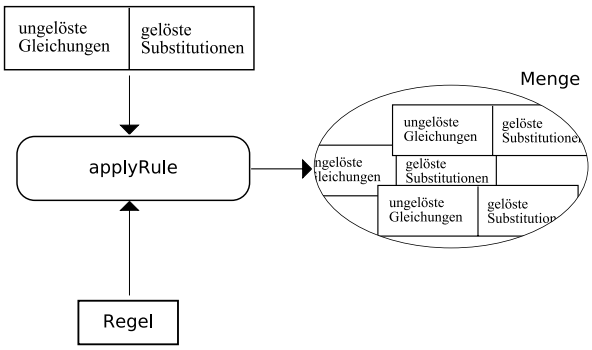


Abbildung 4.5.3. applyRule schematisch

Algorithmus 4.5.5 einfachste Heuristik: `take1Heuristic`

```
take1Heuristic rs =
  let filterEmptyRules =
       filter (any (not . noChoice) . appliedResult) rs
      firstResult  = head filterEmptyRules
      resultChoice =
        HeuristicChoice {
         chosenRule   = appliedRule firstResult,
        chosenResult = head $ appliedResult firstResult
        }
  in  resultChoice
```

4.5.3.4. *Heuristiken.* Eine Heuristik ist eine selektierende Funktion, die aus einer Menge von Regelresultaten eine Regel auswählt und ein Resultat (also eine Menge von `EnvEquationSets`) zurückliefert.

```
type HeuristicFunction =
  (EnvPosEqSet es eq t tp) =>
  [RuleApplicationResult es eq t tp]
  -> HeuristicChoice es eq t tp
data (EnvPosEqSet es eq t tp) =>
  HeuristicChoice es eq t tp =
  HeuristicChoice {
    chosenRule   :: Rule es eq t tp,
    chosenResult :: BacktrackingChoice (es eq t)
  }
```

Mit Algorithmus 4.5.5 ist die einfachste Inkarnation einer Heuristikfunktion angegeben. Diese wählt stets das erste mögliche Regelresultat aus.

Da die Eingabe einer Heuristikfunktion - die Liste `[RuleApplicationResult es eq t tp]` - generiert wird, indem die in einer Strategie aufgelisteten Regeln in der dort angegeben Reihenfolge auf das aktuell zu bearbeitende `EnvPosEqSet` angewendet werden, ist die Eingabeliste aus `RuleApplicationResults` ebenfalls in dieser Reihenfolge.

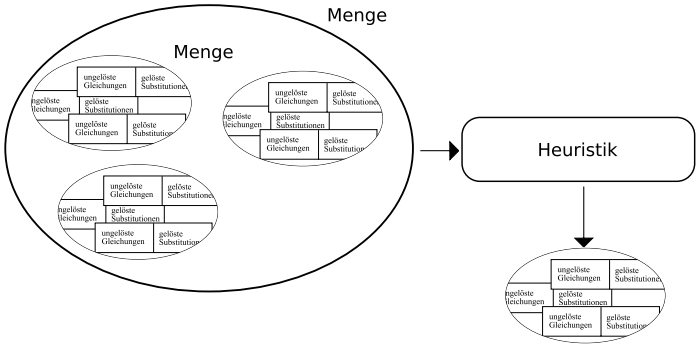


ABBILDUNG 4.5.4. Heuristik schematisch

Dies bedeutet für die einfache Heuristik `take1Heuristic`, dass sie stets die erste anwendbare Regel gemäß der Regelreihenfolge in der aktuellen Strategie auswählt. Insofern kann man diese Heuristik durch die Reihenfolge der Regeln in der Strategie parameterisieren und z. B. durch Eintrag der Regel VARIABLESPLIT als letzte Regel dafür sorgen, dass diese nur dann verwendet wird, wenn keine andere Regel anwendbar ist.

4.5.4. Implementierung der Transitionsregeln. Wir kommen nun zur Implementierung der in Tabelle 4.5.1 angegebenen Transitionsregeln.

4.5.4.1. *Hilfsfunktionen.* Für die Implementierung der Transformationsregeln wurden verschiedene vereinfachende Hilfsfunktionen entwickelt. Diese entstanden aus der Beobachtung, dass alle zu implementierenden Regeln auf jeweils nur einer Kontextgleichung aus der Menge der ungelösten Kontextgleichungen sowie gegebenenfalls den bereits gelösten Variablen und Kontextvariablen operieren.

Für das Ergebnis der Transformationsregeln kann einer der folgenden drei Fälle eintreten:

(1) Die Regel ist nicht anwendbar, weil die Voraussetzungen nicht zutreffen (`RRNothing`).
(2) Die Regel führt zu dem Ergebnis, dass das Kontextmatchingproblem unlösbar ist (`RRBot`).
(3) Die Regel ersetzt die durch sie betrachtete Kontextgleichung durch keine, eine oder mehrere neue Kontextgleichungen und/oder löst eine oder mehrere Variablen oder Kontextvariablen (`RResult`).

Diese drei Ergebnisvarianten werden durch den Datentyp `RuleResult` (Paket `Gersdorf.ContextMatching.General.EnvEquationSet`) modelliert:

```
data (Term t, TermEquation e t) =>
  RuleResult e t =
    RRNothing
    | RRBot
    | RResult {
```

Regel	Kategorie
DEKOMPOSITION	deterministisch
KOLLISION	deterministisch
LOCHDEKOMPOSITION	deterministisch
MULTIKONTEXT-KOLLISION	deterministisch
TERMVERSCHMELZUNG	deterministisch
KONTEXTVERSCHMELZUNG	deterministisch
KONTEXTELIMINATION	deterministisch
VARIABLENSPLIT	don't-know-nichtdeterministisch
SPLIT	don't care-, don't know- oder bedingt don't know-nichtdeterministisch (abhängig von verwendeter Funktion zur Bereitstellung korrespondierender Positionen)
BODENDEKOMPOSITION	don't-care-nichtdeterministisch
BODENKOLLISION	don't-care-nichtdeterministisch
KONSTANTEN-ELIMINATION	deterministisch
MULTIKONTEXT-DEKOMPOSITION	deterministisch
MULTIKONTEXT-KOLLISION	deterministisch

TABELLE 4.5.3. Kategorisierung der Transitionsregeln

```
      rrNewEquations :: [e t],
      rrSolved       :: [(TermType, t)]
    }
```

Wir verwenden zur Erzeugung von Regeln drei Hilfsfunktionen, die drei verschiedenen Regeltypen entsprechen. Die Regeltypen werden hinsichtlich ihres Grades an Nichtdeterminismus unterschieden. Tabelle 4.5.3 kategorisiert die implementierten Regeln.

4.5.4.1.1. (Bedingt) don't know-nichtdeterministische Regeln. Diese zeichnen sich dadurch aus, dass ihr Regelergebnis unter Umständen Backtracking benötigt:

```
cSimpleRule :: (EnvPosEqSet es eq t tp) =>
  String
  -> (es eq t -> eq t
      -> [BacktrackingChoice (RuleResult eq t)])
  -> Rule es eq t tp
cSimpleRule n f = Rule {
     ruleName = n,
     ruleF = RuleFunction simpleRuleWrapper
   }
   where
   simpleRuleWrapper eset = scanUpdate eset (f eset)
```

Die Funktion `cSimpleRule` nimmt zwei Parameter: den Namen der Regel sowie eine Funktion `f`. `f` operiert auf einer Kontextgleichung (daher

der zweite Parameter von `f`) und erhält zusätzlich Zugang zum aktuellen `EnvPosEqSet`, um auf bereits gelöste Variablen und Kontextvariablen zugreifen zu können. `f` liefert als Ergebnis eine Liste don't care-nichtdeterministischer don't know-nichtdeterministischer Regelresultate.

Interessant ist die Anwendung der Funktion `scanUpdate`, die über sämtliche ungelösten Gleichungen des übergebenen `EnvPosEqSet` iteriert und diese durch das Ergebnis der Anwendung von `f` auf jede der Gleichungen ersetzt (vgl. Kapitel 4.5.3.1.1).

Mit `cSimpleRule` können sowohl der deterministische als auch alle drei nichtdeterministischen Fälle abgebildet werden.

4.5.4.1.2. Don't care-nichtdeterministische Regeln. Regeln, die nur don't care-nichtdeterministisch sind, können durch die Funktion `cSimpleRuleDet` erzeugt werden. Die Implementierung delegiert an `cSimpleRule`. Wir geben lediglich den Typ von `cSimpleRuleDet` an:

```
cSimpleRuleDet :: (EnvPosEqSet es eq t tp) =>
  String
  -> (es eq t -> eq t -> [RuleResult eq t])
  -> Rule es eq t tp
```

4.5.4.1.3. Deterministische Regeln. Deterministische Regeln liefern maximal ein Regelresultat und sind damit ein Spezialfall der don't care-nichtdeterministischen Regeln. Sie können über die Funktion `cSimpleRule1` erzeugt werden:

```
cSimpleRule1 :: (EnvPosEqSet es eq t tp) =>
  String
  -> (es eq t -> eq t -> RuleResult eq t)
  -> Rule es eq t tp
```

4.5.4.2. Dekomposition *und* Kollision. Wir beginnen mit der einfachen Implementierung der Dekomposition, die auf einer Termgleichung operiert und die Wurzeln der Terme beider Seiten betrachtet. Sind beide Wurzeln mit dem gleichen Funktionssymbol bezeichnet, so wird der Term dekomponiert.

```
decomposeRule :: (EnvPosEqSet es eq t tp) =>
  Rule es eq t tp
decomposeRule = cSimpleRule1 "decompose/collision"
                             decompose

decompose :: (EnvPosEqSet es eq t tp) =>
  es eq t
  -> eq t
  -> RuleResult eq t
decompose _ eq
  | isApplicable
    && isFunction r
    && (arity l == arity r)
    && (getType l == getType r)
```

```
      = cRuleResult (childrenExtract l r) []
  | isApplicable
      = RRBot
  | otherwise
      = RRNothing
  where
  (l, r) = getPair eq
  isApplicable = isFunction l
```

Beispielhaft für die anderen Regeln geben wir eine kurze Beschreibung der Funktionsweise an. Die Dekomposition ist eine deterministische Regel, daher konstruieren wir sie mit Hilfe der Hilfsfunktion `cSimpleRule1`. Sie operiert - wie alle anderen Regeln auch - auf einer einzigen Termgleichung aus der Menge der ungelösten Gleichungen. Ferner erhält sie Zugriff auf das aktuelle `EnvPosEqSet` und damit u. a. auf die Menge der bereits gelösten Variablen und Kontextvariablen. Diese wird jedoch ignoriert.

Sie prüft, ob die Regel anwendbar ist, indem zum einen getestet wird, ob das Wurzelsymbol der linken Termgleichungsseite eine Funktion ist. Falls nun das Wurzelsymbol der linken mit dem Wurzelsymbol der rechten Seite in Typ, Name und Stelligkeit übereinstimmt, so findet eine Dekomposition statt und die untersuchte Gleichung wird durch Gleichungen der Wurzelkinder ersetzt. Ist die Regel anwendbar, aber eine der weiteren Bedingungen wird nicht erfüllt, so führt dies zum Ergebnis $\perp$ (`RRBot`). Dies entspricht der Kollision. In allen anderen Fällen tut die Regel nichts (`RRNothing`).

Immer dann, wenn Subterme als neue Gleichungen im `RuleResult` zurückgegeben werden, ist die Extraktion dieser Subterme notwendig, damit nachfolgende Regelanwendungen durch Navigation in Richtung des Termvaters nicht den bisherigen Termvater erreichen. Daher ist in solchen Situationen stets die Anwendung einer der Funktionen `childrenExtract`, `extractNode` oder `extractSubTerm` notwendig.

Wir lassen die Definition der `xyzRule`-Funktionen in Zukunft aus, da diese analog zu `decomposeRule` mit Hilfe der Funktionen `cSimpleRule1`, `cSimpleRuleDet` und `cSimpleRule` zu definieren sind.

4.5.4.3. Lochdekomposition. Die Lochdekomposition wird ähnlich wie die Dekomposition implementiert:

```
holeDecompose :: (EnvEquationSet es eq t) =>
  es eq t
  -> eq t
  -> RuleResult eq t
holeDecompose _ eq
  | isApplicable
    && isHole r  = cRuleResult [] []
  | isApplicable = RRBot
  | otherwise    = RRNothing
  where
  (l, r) = getPair eq
  isApplicable = isHole l
```

4.5.4.4. Multikontext-Kollision. Auch die Multikontext-Kollision ist sehr direkt zu implementieren. Zu beachten ist hierbei der potentiell sehr teure Aufruf von `countHoles`. Die Kosten des Aufrufes werden durch die Implementierung der Terme als `ContextTree`s (`ExtendedTerm`) gemildert, da diese in ihrem Kontext eine einmal berechnete Anzahl von Löchern zwischenspeichern (vgl. Kapitel 4.3.3).

```
multiCtxCollision :: (EnvEquationSet es eq t) =>
  es eq t
  -> eq t
  -> RuleResult eq t
multiCtxCollision _ eq
  | countHoles l /= countHoles r = RRBot
  | otherwise                    = RRNothing
  where
  (l, r) = getPair eq
```

4.5.4.5. Term- *und* Kontextverschmelzung. Auch die Implementierung dieser Regel ist trivial. Es genügt, zu prüfen, ob die linke Seite eine gelöste Variable (d. h. eine Variable oder Kontextvariable) ist und ob eine ggf. bereits vorhandene Lösung mit der erneut gefundenen übereinstimmt:

```
solvedDummy :: (EnvEquationSet es eq t) =>
  es eq t
  -> eq t
  -> RuleResult eq t
solvedDummy eset eq
  | isApplicable &&
    hasSubstitution eset (getType l) =
      let boundInstance =
          case (lookupSubstitution eset (getType l)) of
               Nothing  ->
                 error "Could not find Subst."
               (Just t) -> t
      in  if (boundInstance == r)
             then cRuleResult [] []
             else RRBot
  | isApplicable = cRuleResult []
                               [(getType l, r)]
  | otherwise    = RRNothing
  where
  (l, r) = getPair eq
  isApplicable = isSolvedVariable l
```

Der Fehlerfall „`Could not find Subst.`" kann nicht auftreten, da zuvor das Vorhandensein der Substitution im `Guard` überprüft wurde.

4.5.4.6. Kontextelimination. Die Kontextelimination ist anwendbar, wenn die Termwurzel der linken Seite der Kontextgleichung eine bereits gelöste Kontextvariable `X` ist. In diesem Falle suchen wir den TreePath des Lochs innerhalb von `X` (`getHolePath`) und im Term der rechten Seite der

Kontextgleichung denjenigen Subterm `t'`, der denselben TreePath besitzt (`gotoPath`).

Wir müssen darüberhinaus prüfen, ob die bereits gefundene Lösung von `X` - instanziiert mit `t'` - mit der rechten Seite der Kontextgleichung übereinstimmt. Ist dies nicht der Fall, ist das Ergebnis $\perp$ (`RRBot`). Ansonsten wird die Kontextvariable eliminiert und die untersuchte Gleichung durch eine Kontextgleichung aus Kind von `X` und Subterm `t'` ersetzt.

```
ctxElimination :: (EnvPosEqSet es eq t tp) =>
  es eq t
  -> eq t
  -> RuleResult eq t
ctxElimination eset eq
  | isApplicable =
      let maybe_sol   =
            do eq_C     <- lookupSubstitution
                           eset
                           (getType eq_X)
               holePath <- getHolePath eq_C
               eq_t'    <- gotoPath eq_t
                                    holePath
               eq_C_t'  <- instantiateContext
                           eq_C
                           (extractNode eq_t')
               if (eq_C_t' /= eq_t)
                 then Nothing
                 else return
                        (cRuleResult
                         [cEquation eq_s eq_t']
                          [])
      in  case (maybe_sol) of
            Nothing    -> RRBot
            (Just sol) -> sol
  | otherwise = RRNothing
  where
  eq_X = getLeft eq
  eq_s = head $ getChildren eq_X
  eq_t = getRight eq
  isApplicable =
    isContextVariable eq_X
    && hasSubstitution eset (getType eq_X)
    && not (isHole eq_s)
```

4.5.4.7. Variablen-Split. Die einzige don't know-nichtdeterministische Regel, die zu implementieren ist, ist die Regel Variablen-Split. Diese Transformationsregel ist immer anwendbar, wenn die Wurzel der linken Termgleichungsseite eine Kontextvariable ist.

Wie sich herausstellt, können die nichtdeterministisch zu betrachtenden alternativen Termgleichungen auf einfache Weise mittels Haskell-List-Comprehensions generiert werden. Hierzu verwenden wir eine weitere Hilfsfunktion - `permuteContextTerms`:

```
permuteContextTerms :: (PositionTerm t tp,
                        TreeExtractor t) =>
  t
  -> [(t, t)]
```

Sie ist im Modul `Gersdorf.ContextMatching.Base.PositionTerm` definiert und nimmt als Argument einen `PositionTerm`. Für jede mögliche Position `p`, an der ein Subterm von `t` durch ein Loch ersetzt werden kann, wird ein Listenelement aus einem Paar $\left(\mathtt{t}\,[\Box]_{\mathtt{p}}\,, \mathtt{t}|_{\mathtt{p}}\right)$ generiert.

Variablen-Split rät die korrekte Position des Lochs in einer instanziierten Kontextvariablen. Dieses Erraten führt zu einem don't know-Nichtdeterminismus, der durch ein Datum vom Typ `BacktrackingChoice` modelliert wird.

```
variableSplit :: (EnvPosEqSet es eq t tp) =>
  es eq t
  -> eq t
  -> [BacktrackingChoice (RuleResult eq t)]
variableSplit eset eq
  | isApplicable
    && (not $ isHole eq_s) =
      let eq_X_just = cTerm
                    $ cContextVar (getName eq_X)
                                  cHole
          eq_s_extr = cTree $ extractNode eq_s
          result    =
            cBacktrackingMany $
             [ cRuleResult
                 [cEquation eq_X_just eq_C,
                  cEquation eq_s_extr eq_t']
                 []
             | (eq_C, eq_t') <- permuteContextTerms eq_t ]
      in  [result]
  | otherwise  = [cBacktrackingNo]
  where
  eq_X         = getLeft eq
  eq_s         = head $ getChildren eq_X
  eq_t         = getRight eq
  isApplicable = isContextVariable eq_X
```

4.5.4.8. Split. Split ist eine Regel, die davon ausgeht, dass man im Besitz eines Paares korrespondierender Positionen ist. In diesem Fall ist die Ausführung der Regel recht trivial.

Da es verschiedene Methoden gibt, korrespondierende Positionen zu berechnen, wurde die Implementierung so gewählt, dass sie mit einer Funktion zur Berechnung korrespondierender Positionen parameterisierbar ist:

```
split :: (EnvPosEqSet es eq t tp,
          TreePath tp2) =>
  CorrPosProvider t tp2
  -> es eq t
  -> eq t
  -> [BacktrackingChoice (RuleResult eq t)]
```

Ihrer Signatur nach ist die Funktion `split` don't know-Nichtdeterministisch. Dies hängt jedoch von der verwendeten Funktion zur Berechung korrespondierender Positionen ab. Ist diese dergestalt, dass die von ihr berechneten Positionen stets korrekt sind, so verhält sich `split` don't care-nichtdeterministisch, im anderen Falle bedingt don't know-nichtdeterministisch.

Die Parameterisierung erfolgt durch einen `CorrPosProvider`, der einen Namen besitzt und eine Funktion zur Berechnung korrespondierender Positionen bereitstellt. Ferner gibt die Recordkomponente `cppBacktracking` an, ob die berechneten Positionen immer korrekt sind und daher kein Backtracking benötigten (don't care-Nichtdeterminismus) oder ob sich unter diesen auch fälschlicherweise als korrespondierend entdeckte Positionen befinden (bedingter don't know-Nichtdeterminismus). Durch die Unterstützung des bedingten don't know-Nichtdeterminismus können also auch solche `CorrPosProvider` verwendet werden, die korrespondierender Positionen nur hinsichtlich eines notwendigen, nicht aber eines hinreichenden Kriteriums berechnen können.

```
data (Term t, TreePath tp2) =>
  CorrPosProvider t tp2 =
    CorrPosProvider {
      cppName :: String,
      cppBacktracking :: Bool,
      cppFunction :: CorrPosFunction t tp2
    }

newtype (Term t, TreePath tp2) =>
  CorrPosFunction t tp2 =
    CorrPosFunction ((t,t) -> [(tp2, tp2)])

cCorrPosProvider name f backtrack =
  CorrPosProvider {
    cppName = name,
    cppFunction = CorrPosFunction f,
    cppBacktracking = backtrack
  }
```

Eine `CorrPosFunction` nimmt ein Paar von Termen (die linke und rechte Seite einer Kontextgleichung) und gibt eine Liste gefundener korrespondierender Positionen in Form von Paaren von `TreePaths` zurück. Damit ist die Implementierung von `split` vorgegeben:

```
split cpp eset eq
  | isApplicable =
      let cResult :: (TermEquation eq t,
                      PositionTerm t tp,
                      TermBuilder t) =>
            (t, t)
            -> RuleResult eq t
          cResult (eq_s', eq_t') =
            let eq_s_Ctx = makeContext eq_s'
                eq_t_Ctx = makeContext eq_t'
               eq_t'_extr = extractSubTerm eq_t'
               eq_s'_extr = extractSubTerm eq_s'
            in  cRuleResult
                  [cEquation eq_s_Ctx eq_t_Ctx,
                   cEquation eq_s'_extr eq_t'_extr]
                  []
      in  if (cppBacktracking cpp)
            then [cBacktrackingOne
                    (map cResult corrTerms)]
          else map (cBacktrackingDet . cResult)
                   corrTerms
  | otherwise    = [RRNothing]
  where
  (l, r) = getPair eq
  corrTerms = getCorrTerms l r
              (applyCorrPosProvider cpp eq)
  isApplicable = not $ null corrTerms
```

Dabei ist `applyCorrPosProvider :: (TermEquation eq t, TreePath tp2) => CorrPosProvider t tp2 -> eq t -> [(tp2, tp2)]` eine Hilfsfunktion zur Anwendung eines `CorrPosProviders` auf eine Kontextgleichung.

Abhängig davon, ob im `CorrPosProvider` angegeben wurde, dass Backtracking benötigt wird, wird das Ergebnis aufbereitet.

Die Funktion `getCorrTerms` liefert anhand zweier Terme (linke und rechte Termgleichungsseite) sowie einer Liste entsprechend korrespondierender Positionspaare eine Liste derjenigen durch die Positionen adressierten Subterme, deren Positionen nichttrivial korrespondieren.

```
getCorrTerms :: (PositionTerm t tp,
                 TreePath tp2) =>
  t
  -> t
  -> [(tp2, tp2)]
```

```
  -> [(t, t)]
getCorrTerms l r =
  filter (not . isTriviallyCorresponding)
  . map (getCorrTermPair (l, r))
  . filter (\(p1, p2) ->
            not (isEmptyPath p1
                 || isEmptyPath p2))

isTriviallyCorresponding (l, r) =
  (isRoot l && isRoot r) || isHole l
```

Die Filterung durch die Funktion `isEmptyPath` erfolgt, um nicht unnötigerweise trivial korrespondierende Terme zu erzeugen. Diese sollen möglichst frühzeitig erkannt werden.

Die eigentliche Umsetzung der `TreePath`s korrespondierender Positionen geschieht in der folgenden Funktion. Sie konvertiert zunächst die beiden Pfade in den Pfadtyp der Terme. Das ist notwendig, weil die Berechnungsroutine zur Ermittlung der korrespondierenden Positionen die Terme gegebenenfalls in eine andere Darstellung konvertiert und sie die zu dieser Darstellung passenden Instanzen von `TreePath` zurückgegeben hat.

```
getCorrTermPair :: (PositionTerm t tp,
                    TreePath tp2) =>
  (t, t)
  -> (tp2, tp2)
  -> (t, t)
getCorrTermPair (l, r) (lp, rp) =
  let nlp = cNewPath l lp
      nrp = cNewPath r rp
      cNewPath t tp =
        case (cPath t (toPathIndexList tp)) of
          Nothing -> error ("Path not found")
          (Just ntp) -> ntp
      nlt = getNewTerm l nlp
      nrt = getNewTerm r nrp

      getNewTerm t tp =
        case (gotoPath t tp) of
          Nothing -> error ("Subterm not found")
          (Just nt) -> nt
  in  (nlt, nrt)
```

Wir beschreiben nun die Implementierung der Methoden zur Ermittlung korrespondierender Positionen. Im Rahmen dieser Diplomarbeit wurden die Methoden Linearisierung, Korrespondierende Lochpfade sowie Korrespondierende Funktionssymbole implementiert.

4.5.4.8.1. Linearisierung. Wie bereits in Kapitel 3.4.4.1 beschrieben, beruht die Anwendung der Linearisierung darauf, aus einer eventuell nichtlinearen Kontextgleichung durch Umbenennung von Variablen und Kontextvariablen eine lineare Gleichung zu formen. Es werden dann mit Hilfe des Algorithmus zur Lösung linearer Kontextmatchingprobleme und der Tabelle, die während der Lösung mittels dynamischen Programmierens erstellt wird, korrespondierende Positionen der linearisierten Gleichung gesucht.

Wir berechnen zunächst Paare korrespondierender Terme, da die Umwandlung in ein Paar korrespondierender `TreePath`s durch die Funktion `getPath` sehr einfach ist. Weil der Algorithmus zur Lösung linearer Kontextmatchingprobleme voraussetzt, dass die Terme als `HeapTerm`s vorliegen, wird zunächst eine Konvertierung und anschließend eine Linearisierung vorgenommen.

Die Methode zur Linearisierung durch `linearize` ändert sämtliche Namen von Variablen und Kontextvariablen, indem sie eine fortlaufende Nummer an die Namen anfügt.

Es folgt die Lösung des linearen Kontextmatchingproblems durch die Bildung der Tabelle für den Ansatz des dynamischen Programmierens. In dieser Tabelle werden Zeilen mit nur einem Eintrag `True` gesucht. Die Subterme, die dieser Tabellenzelle entsprechen, sind korrespondierende Subterme.

```
correspondingTermsLinear :: (Term t) =>
  (t, t)
  -> [(HeapTerm, HeapTerm)]
correspondingTermsLinear (l,r) =
  let heq :: SimpleTermEquation HeapTerm
      heq   = cEquation lht rht
      lht   = toHeapTerm l
      rht   = toHeapTerm r

      linEq = snd $ linearize 0 heq
      linL  = getLeft linEq
      linR  = getRight linEq

      dynTable =
        dynamicTableLinearCMDecision (lht, rht)

      n    = size linL
      m    = size linR

     getRow i    = map (\j -> dynamicSolve dynTable
                                          (i, j))
                       [1..m]
     toIntList bs = map (\b -> if (b) then 1 else 0)
                        bs
      exactly1Checked is = ((foldl (+) 0 is) == 1)
       exactly1Row = exactly1Checked
                   . toIntList
                   . getRow
```

```
        singleRows = filter exactly1Row [1..n]
        getPair index selector builder =
         map (\(i, j) -> (getAt lht i, getAt rht j))
          $ map (buildPair selector builder ) index
        where
          buildPair selector builder i =
          let j = fromJust (elemIndex True (selector i))
                    + 1
            in  builder (i, j)

        pairsRows = getPair singleRows getRow id
  in  pairRows

correspondingPosLinear :: (Term t) =>
  (t, t)
  -> [(HeapPath, HeapPath)]
correspondingPosLinear =
  map (\(l, r) -> (getPath l, getPath r))
  . correspondingTermsLinear
```

Wie bereits in Kapitel 3.4.4.1 besprochen, benötigt das Verfahren der Linearisierung gegebenenfalls Backtracking, da es nur ein notwendiges Kriterium für die tatsächlich im Ursprungsproblem korrespondierenden Positionen darstellt. Das Verhalten der Linearisierung ist dementsprechend bedingt don't know-nichtdeterministisch. Daraus ergibt sich die folgende Definition des `CorrPosProviders`:

```
corrPosLinearProvider :: (Term t) =>
  CorrPosProvider t HeapPath
corrPosLinearProvider =
  cCorrPosProvider "correspondingLinear"
                   correspondingPosLinear
                   True
```

4.5.4.8.2. Korrespondierende Lochpfade. Die Korrespondierenden Lochpfade können ebenso wie die Linearisierung Positionen fälschlicherweise als korrespondierend erkennen. Daher ist ihr Verhalten bedingt don't know-nichtdeterministisch.

Die Berechnung korrespondierender Positionen mit Hilfe von Lochpfaden beruht auf den trivial korrespondierenden Löchern und in der gegenwärtigen Implementierung auf der Linearisierung.

```
correspondingPosHoles :: (PositionTerm t tp) =>
  (t,t)
  -> [(tp, tp)]
correspondingPosHoles p =
  let mHoles       = matchingHoles p
      corrMonadics = correspondingMonadics mHoles
      corrPairs = zip mHoles
                  (correspondingPairs corrMonadics)
```

```
  in  concat
      $ map calcCorrespondingPaths corrPairs
```

Die Methode sucht mittels der Hilfsfunktion `matchingHoles` zunächst Paare korrespondierender Löcher im übergebenen Termpaar. Ausgehend von diesen und ihren Pfaden zur Termwurzel werden durch `correspondingMonadics` monadische Terme gebaut. Aufbauend auf diesen Paaren korrespondierender Lochpfade erfolgt die Berechnung korrespondierender Positionen in der Funktion `calcCorrespondingPaths`.

Wir geben hier nur die Typen der beiden Funktionen `matchingHoles` und `correspondingMonadics` an:

```
matchingHoles :: (Term t) =>
  (t, t)
  -> [(t, t)]

correspondingMonadics :: (PositionTerm t tp) =>
  [(t, t)]
  -> [(t, t)]
```

Die eigentlich zentralen Funktionen sind einerseits die Funktion `correspondingPairs`, die mittels Linearisierung korrespondierende Positionen findet, andererseits `calcCorrespondingPaths` und `findCorrespondingPaths`, die deren Ergebnisse aufbereiten.

```
correspondingPairs :: (PositionTerm t tp) =>
  [(t, t)]
  -> [[(HeapPath, HeapPath)]]
correspondingPairs = map correspondingPosLinear
```

Wie zu erkennen ist, wird die Berechnung der korrespondierenden Positionen auf den monadisierten Termen an die Linearisierung deligiert. Die nunmehr berechneten korrespondierenden Positionen basieren auf den monadisierten Termen und müssen durch `calcCorrespondingPaths` wieder in `TreePath`s der ursprünglichen Terme umgerechnet werden.

```
calcCorrespondingPaths :: (PositionTerm t tp) =>
  ((t,t), [(HeapPath, HeapPath)])
  -> [(tp, tp)]
calcCorrespondingPaths ((l, r), pths) =
  let (lp, rp) = unzip pths
  in  zip (findCorrespondingPaths l lp)
          (findCorrespondingPaths r rp)

findCorrespondingPaths :: (PositionTerm t tp) =>
  t
  -> [HeapPath]
  -> [tp]
findCorrespondingPaths t pths =
  let r         = getRoot t
      pilist    = toPathIndexList (getPath t)
```

```
        depths    = map pathDepth pths
        origPaths = map (\i -> fromJust
                               $ cPath r
                               $ take i pilist)
                        depths
 in  origPaths
```

Während `calcCorrespondingPaths` eher Hilfscharakter besitzt, übernimmt die Funktion `findCorrespondingPaths` den eigentlichen Teil der Berechnung. Sie ermittelt insbesondere die Tiefe (Länge) der `TreePaths`, um mit ihrer Hilfe die Originalpfade zu den Löchern durch Abschneiden auf die korrekten Termpositionen umzurechnen.

4.5.4.8.3. Korrespondierende Funktionssymbole. Wir implementieren zusätzlich die Methode Korrespondierende Funktionssymbole, die darauf beruht, dass auf beiden Seiten einer Termgleichung genau einmal auftretende Funktionssymbole in natürlicher Weise korrespondierende Positionen einführen (vgl. Kapitel 3.4.4.2).

Die Besonderheit dieser Methode zur Berechnung korrespondierender Positionen in Vergleich zu den bisher genannten ist, dass sie in Verbindung mit Split don't care-nichtdeterministisch ist, da sie stets korrekte korrespondierende Positionen in Bezug auf die Eingabegleichung berechnet.

```
corrPosUniquesProvider :: (PositionTerm t tp) =>
  CorrPosProvider t tp
corrPosUniquesProvider =
  cCorrPosProvider "correspondingPosUniques"
                   correspondingPosUniques
                   False

correspondingPosUniques :: (PositionTerm t tp) =>
    (t,t) -> [(tp, tp)]
correspondingPosUniques (l,r) =
    let uniqueFunctions t =
         uniquesBy (\a b -> compare (getType a)
                                    (getType b))
          $ filter isFunction
          $ toList t
        functionsR = uniqueFunctions r
        functionsL = uniqueFunctions l

        uniqueTermPairs =
         zipEqualsBy (\a b -> compare (getType a)
                                      (getType b))
                     functionsL
                     functionsR
   in  map (scalarApply getPath) uniqueTermPairs
```

4.5.4.9. Bodendekomposition *und* -kollision. Die Bodendekomposition ist in ihrer Implementierung eine der umfangreicheren Regeln. Sie

Algorithmus 4.5.6 BODENDEKOMPOSITION UND -KOLLISION

(1) Sei $l \approx r$ die Termgleichung. Seien $l = C\left[\Box^k, f\left(s_1, \ldots, s_n\right), \Box^l\right]$ und $r = D\left[\Box^k, f\left(t_1, \ldots, t_n\right), \Box^l\right]$.
(2) Berechne für $l \approx r$ die Liste aller Lochpaare. Dies ist möglich, da die Anzahl der Löcher in l und r identisch ist.
(3) Betrachte von nun an nur noch die Lochpaare, bei denen bei beiden Löchern die Väter jeweils Funktionen sind.
(4) Berechne nun für jedes der Lochpaare (lh, rh) das Resultat. Falls
   (a) sowohl lh als auch rh das j-te Kind ihrer Väter lhf und rhf sind,
   (b) lhf und rhf denselben TERMTYPE haben,
   (c) folgendes gilt: Seien $h_{l_1}, \ldots, h_{l_m}$ die Löcher, die unmittelbare Kinder von lhf sind. $l_1, \ldots, l_m$ seien die Nummern der Löcher. $h_{r_1}, \ldots, h_{r_m}$ sei analog definiert. Dann muss $l_1 = r_1, \ldots, l_m = r_m$ gelten.
   (d) die flache Lochsignatur von lhf und rhf identisch ist,

   dann ist das Resultat die Menge der Gleichungen bestehend aus den Kindpaaren von lhf und rhf sowie der Kontexte C und D, in denen lhf bzw. rhf durch das Loch ersetzt wurden: $\left\{s_1 \approx t_1, \ldots, s_n \approx t_n, C\left[\Box^{k+l+1}\right] \approx D\left[\Box^{k+l+1}\right]\right\}$. Andernfalls ist das Ergebnis $\perp$.

Da es innerhalb von $l \approx r$ mehrere Lochpaare geben kann, auf die die genannten Bedingungen zutreffen, kann es auch mehrere unabhängige Resultatmöglichkeiten geben.

ist eine don't care-nichtdeterministische Regel und liefert als solche mehrere mögliche Resultate, die als alternativ anzusehen sind. In der Verabeitung darf daher jedes der Resultate erschöpfend verwendet werden.

Die Regel ist anwendbar, wenn auf linker wie rechter Termgleichungsseite die gleiche Anzahl von Löchern > 0 vorhanden ist und weder linke noch rechte Seite das Loch selbst sind.

Wir geben zunächst die Regel in Algorithmus 4.5.6 an.

DEFINITION. Die *flache Lochsignatur* eines Terms t ist der String, der entsteht, indem man für jedes Kind von t, welches ein Loch ist, eine 1 einsetzt und für jedes andere Kind eine 0.

BEISPIEL. Sei $t = f\left(a, \Box, g\left(\Box\right)\right)$. Dann ist die flache Lochsignatur von t der String 010.

Algorithmus 4.5.6 wird in der Implementierung noch um ein Detail erweitert. Da in der Liste aller Lochpaare Löcher stehen können, die Kinder desselben Subterms sind, werden solche „mehrfach vorhandenen" Löcher gefiltert.

Die Hauptfunktion dient als Fassade, die die Anwendbarkeit von BODENDEKOMPOSITION überprüft und die genannte Lochpaarliste berechnet:

```
bottomDecompose :: (EnvPosEqSet es eq t tp) =>
  es eq t
```

```
    -> eq t
    -> [RuleResult eq t]
  bottomDecompose eset eq
   | isApplicable =
       let holePairs = zip (getHolesDeep l)
                           (getHolesDeep r)
           onlyFunctionHolePairs =
             filter (\(lh, rh) ->
                         (isFatherFunction lh)
                         && (isFatherFunction rh))
                    holePairs
         isFatherFunction h = checkFather isFunction h
       in  calcRuleResult onlyFunctionHolePairs
    | otherwise = [RRNothing]
    where
    (l, r) = getPair eq
    isApplicable =
      let leftHoles = countHoles l
      in  (leftHoles > 0)
          && (leftHoles == countHoles r)
          && (not (isHole l) && not (isHole r))
```

Die Funktion `calcRuleResult` dient als Filterinstanz, die unterhalb eines Subterms mehrfach vorhandenen Löcher zu filtern:

```
  calcRuleResult holePairs =
    calcRuleResult' 1 Set.emptySet [] holePairs
    where
    calcRuleResult' holeNum procSet res [] = res
    calcRuleResult' holeNum procSet res
                    (p@(lh, rh):hss) =
     let (processedHN, result) = calcSingleResult p
         newprocSet = Set.listToSet processedHN
         unionedSet = Set.union procSet
                                newprocSet
      in  if (Set.elementOf holeNum procSet)
            then calcRuleResult' (holeNum+1)
                    procSet res hss
            else calcRuleResult' (holeNum+1)
                    unionedSet (result:res) hss
```

`calcSingleResult` ist die Implementierung von Algorithmus 4.5.6:

```
  calcSingleResult :: (AnalysisTerm t, PositionTerm t tp,
              TreeExtractor t, TermEquation eq t) =>
    (t, t)
    -> ([Int], RuleResult eq t)
  calcSingleResult p@(lh, rh) =
    let leftF  = fromJust (getFather lh)
```

```
        rightF = fromJust (getFather rh)
        leftHN = holeNumbersFlat leftF
        rightHN = holeNumbersFlat rightF
        childEqs = childrenExtract leftF rightF
        eq_C'    = makeContext leftF
        eq_D'    = makeContext rightF
        eq_C'_D' = cEquation eq_C' eq_D'
    in  if ((getChildIndex lh == getChildIndex rh)
            && (getType leftF == getType rightF)
            && (holeSignatureFlat leftF ==
                holeSignatureFlat rightF)
            && (leftHN == rightHN)
          then (leftHN,
                cRuleResult (eq_C'_D':childEqs) [])
          else (leftHN, RRBot)

holeNumbersFlat t = map ((+) 1 . getLeftHoles)
                        (getHolesFlat t)
```

Die Implementierung ist so entwickelt, dass sie theoretisch sämtliche Anwendungsfälle von Bodendekomposition in einer Termgleichung berechnet. Durch die verzögerte Auswertung von Haskell wird jedoch erreicht, dass nur diejenigen tatsächlich berechnet werden, die ausgewertet werden sollen.

4.5.4.10. Konstanten-Elimination. Die Konstanten-Elimination erfordert die Suche nach identischen Konstanten auf beiden Seiten der betrachteten Kontextgleichung, und zwar solcher Konstanten, die sowohl links als auch rechts des Gleichheitszeichens genau einmal auftreten. Die Funktion `getMatchingConstants` findet solche Konstantenpaare. Auf eine nähere Betrachtung ihrer Implementierung wird an dieser Stelle verzichtet.

```
constantEliminate :: (EnvPosEqSet es eq t tp) =>
  es eq t
  -> eq t
  -> RuleResult eq t
constantEliminate _ eq
  | isApplicable =
     let positions   = map (scalarApply getPath)
                           constantPairs
         (newl, newr) = createHoledTerm (getPair eq)
                                        positions
      in  cRuleResult [cEquation newl newr] []
  | otherwise   = RRNothing -- rule not applicable
  where
  isApplicable = not (null constantPairs)
  constantPairs = getMatchingConstants eq
```

Die Funktion `createHoledTerm` nimmt ein Paar von Termen und `TreePaths` und ersetzt die Terme an den Stellen der `TreePaths` durch Löcher:

Algorithmus 4.5.7 MULTIKONTEXT-DEKOMPOSITION UND -KOLLISION

(1) Sei $s \approx t$ die zu betrachtende Kontextgleichung. Sei weiter $s = C\left[f\left(C_1\left[\square^{k_1}\right], \ldots, C_n\left[\square^{k_n}\right]\right)\right]$ und $t = D\left[f\left(D_1\left[\square^{k_1}\right], \ldots, D_n\left[\square^{k_n}\right]\right)\right]$

(2) Sammle in s und t alle Subterme f, die Funktionen sind und folgende Eigenschaften besitzen:
   (a) Im Termbaum existieren links und rechts von f keine Löcher.
   (b) f hat eine Stelligkeit von mindestens 2.
   (c) In mindestens zwei Argument-Subtermen von f existiert mindestens ein Loch.

(3) Sei $\mathcal{M}$ die Menge aller Paare der in Schritt 2 gefundenen Funktionssubterme. Wegen Proposition 3.4.15 kann es jedoch nur eine solche geben.

(4) Für das gefundene Paar (t_1, t_2) berechnen wir wie folgt das Ergebnis:
   - Falls $\texttt{TermType}\,(t_1) \neq \texttt{TermType}\,(t_2)$, dann ist das Ergebnis $\perp$.
   - Sei l_1 die tiefe Lochsignatur von t_1, l_2 analog. Falls $l_1 \neq l_2$, so ist das Ergebnis $\perp$.
   - Sonst wird die untersuchte Termgleichung $s \approx t$ ersetzt durch die Gleichungen

$$\left\{C\left[\square\right] \approx D\left[\square\right], C_1\left[\square^{k_1}\right] \approx D_1\left[\square^{k_1}\right], \ldots, C_n\left[\square^{k_n}\right] \approx D_n\left[\square^{k_n}\right]\right\}$$

```
createHoledTerm :: (PositionTerm t tp,
                    SubTermBuilder t) =>
  (t, t)
  -> [(tp, tp)]
  -> (t, t)
createHoledTerm ts [] = ts
createHoledTerm (l,r) ((tpl, tpr):tps) =
  let newl = fromJust $ replacePath l tpl (cHole)
      newr = fromJust $ replacePath r tpr (cHole)
  in  createHoledTerm (newl, newr) tps
```

4.5.4.11. MULTIKONTEXT-DEKOMPOSITION *und* -KOLLISION. Aufgrund von Proposition 3.4.15 ist MULTIKONTEXT-KOLLISION eine deterministische Regel. Die Bedingungen an die Termgleichung sind jedoch recht komplex, was sich auch auf den zu entwickelnden Regelalgorithmus auswirkt.

Algorithmus 4.5.7 gibt das Verfahren an.

DEFINITION. Die *tiefe Lochsignatur* eines Terms t ist der String, der dadurch entsteht, dass man für jedes Kind von t, welches in beliebiger Tiefe ein Loch besitzt, eine 1 einsetzt und für jedes andere Kind eine 0.

BEISPIEL. Sei $t = f\,(a, \square, g\,(\square))$. Dann ist die tiefe Lochsignatur von t der String 011.

Wir betrachten nun die Implementierung:

```
multiContextDecompose :: (EnvPosEqSet es eq t tp) =>
  es eq t
  -> eq t
  -> RuleResult eq t
multiContextDecompose _ eq
  | isApplicable =
      let scoreCandidate :: (TermEquation eq t,
                             PositionTerm t tp,
                             TreeExtractor t) =>
            (t, t)
            -> RuleResult eq t
          scoreCandidate (l, r) =
            if ((getType l == getType r)
                && (holeSignatureDeep l
                    == holeSignatureDeep r))
              then let eq_C_D =
                         cEquation (makeContext l)
                                   (makeContext r)
                       eq_cs  = childrenExtract l r
                   in  cRuleResult (eq_C_D:eq_cs) []
              else RRBot
        result = scoreCandidate (fromJust cand1,
                                 fromJust cand2)
      in  result
  | otherwise = RRNothing
  where
  (l, r) = getPair eq
  cand1 = doubleHoleFunction l
  cand2 = doubleHoleFunction r
  leftH = getHoleCount l
  isApplicable = (leftH == getHoleCount r)
                 && (leftH >= 2)
                 && (isJust cand1
                     && isJust cand2)
```

Die Funktion berechnet zunächst mit Hilfe von `doubleHoleFunction` das potentielle Kandidatenpaar gemäß Schritt 2 aus Algorithmus 4.5.7 und wendet darauf die Funktion `scoreCandidate` an, die Schritt 4 des Algorithmus implementiert.

```
doubleHoleFunction :: (Term t) =>
  t
  -> Maybe t
doubleHoleFunction rt =
  let ts = filter (\t -> isFunction t
                  && (fst $ numberHolePaths t) >= 2)
           $ allHolesTerms rt
  in  if (null ts)
```

```
            then Nothing
            else Just $ head ts
```

`doubleHoleFunction` errechnet unter Verwendung von `allHolesTerms` alle Subterme des übergebenen Terms t, die alle in t enthaltenen Löcher umfassen. Diese werden daraufhin gefiltert, ob sie eine Funktion sind und in mindestens zwei Argument-Subtermen ein Loch besitzen.

```
allHolesTerms :: (Term t) =>
  t
  -> [t]
allHolesTerms t = allHolesTerms' [] t
  where
  allHolesTerms' accum t
    | isContextVariable t =
      allHolesTerms' (t:accum) (head $ getChildren t)
    | isFunction t        =
        let cs                = getChildren t
            (nh, signature10) = numberHolePaths t

            processTermList accum [] = accum
            processTermList accum (c:cs) =
              processTermList
                 (allHolesTerms' accum c)
                 cs

            restTerms =
              if (nh == 0)
                then processTermList (t:accum)
                                     cs
                else if (nh == 1)
                 then case (findIndex ((/=) 0)
                                signature10) of
                         Nothing  ->
                           error "No idx /= 0."
                         (Just i) ->
                      case (getChild t (i+1)) of
                             Nothing ->
                              error "No child."
                             (Just c) ->
                        allHolesTerms' (t:accum) c
                  else t:accum
        in  restTerms

    | otherwise = t:accum
```

Die Funktion traversiert über die Subterme des Terms t und verwendet dabei die Akkumulatortechnik, um eine Liste aufzubauen und dadurch verschachtelte `appends (++)` zu vermeiden [**Bird1998**]. Bei Kontexttermen taucht sie tiefer in den Term ein, bei Funktionen traversiert sie nur dann

tiefer, wenn es nur ein Argument-Subterm gibt mit einer Lochanzahl größer Null. In diesem Fall wird dieser Argument-Subterm zur Traversion gewählt.

```
numberHolePaths :: (Term t) =>
  t
  -> (Int, [Int])
numberHolePaths t =
  let signature10 = map signum
                       (holeSignatureDeep t)
  in  (foldl (+) 0 signature10, signature10)
```

Die Funktion `numberHolePaths` ist eine Hilfsfunktion, die in einem Paar einerseits die Anzahl der direkten Kinder des Argumentterms, die Löcher enthalten, und andererseits eine Signatur der Argumentlöcher zurückliefert.

KAPITEL 5

Analyse

Im vorangegangenen Kapitel wurde ausführlich besprochen, welche der Algorithmen aus Kapitel 3 auf welche Weise implementiert wurden. Wir wollen uns nun in diesem Kapitel mit dem Verhalten des so implementierten Systems zur Lösung von Kontextmatchingproblemen widmen.

Das vorgestellte und implementierte System zur Lösung von Kontextmatchingproblemen wurde mit verschiedenen Strategien und Problemen mit Augenmerk auf seine Ausführungsdauer untersucht.

Wir betrachten zunächst die linearen Kontextmatchingprobleme separat, da für diese ein optimierter Algorithmus realisiert wurde.

Bei der daran anschließenden Betrachtung der Implementierung zur Lösung genereller Kontextmatchingprobleme stand die Frage im Vordergrund, welche Regeleinführungen zu welchen Laufzeitauswirkungen im Sinne der Anzahl der Regelanwendungen führen. Dabei wurden verschiedene Termklassen untersucht.

Testsystem war in allen Fällen ein Rechner mit AMD K6II-CPU (400 MHz) und 392 MByte RAM. Als Compiler wurde der Glasgow Haskell Compiler in Version 6.2.1 verwendet.

5.1. Lineares Kontextmatching

5.1.1. Testszenario. Wir betrachten die Laufzeit des Algorithmus für einen speziellen Typ linearer Probleme, der sich dadurch auszeichnet, dass er eine exponentielle Anzahl Lösungen in Abhängigkeit seiner Größe liefert. Dieser Typ wurde in Beispiel 2.2.10 vorgestellt.

Beispiel 5.1.1. Wir geben kurz ein Beispiel an für die Größe 4:

```
*Test.TestGHCI> linearExponential 4
[X4[Y4[h(X3[Y3[h(X2[Y2[h(X1[Y1[h(a())]])]])]])]]
=? g(h(g(h(g(h(g(h(a()))))))))]
```

Wir testeten lineares Kontextmatching mit verschiedenen Problemgrößen.

Problem-größe	Anzahl Lösungen	Laufzeit		
		LCM (komplett)	LCM (1. Lösung)	LLCM
5	32	0.2	0.28	0.075
7	128	0.7	0.34	0.14
8	256	1.5	0.37	0.18
9	512	3.7	0.46	0.25
10	1.024	9.5	0.51	0.3
12	4.096	59.5	0.71	0.38
15	32.768	13:47.5	0.74	0.5
20	1.048.576	n/a	1.62	0.9
50	2^{50}	n/a	17.73	8.67
100	2^{100}	n/a	2:24.1	1:02.4

TABELLE 5.1.1. Ergebnisse lineares Kontextmatching (Laufzeiten in der Form min:sec.msec)

Termprofil	Anzahl Lösungen	Laufzeit LCM (komplett)
varRTP	3.3	0.10
smallRTP	2	0.06
cvarRTP	180.1	2.32
manyFuncRTP	2.8	0.20
defaultRTP	7.5	0.13
arity	3.1	0.20

TABELLE 5.1.2. Lineares Kontextmatching mit Zufallstermen (Laufzeiten in der Form min:sec.msec)

5.1.2. Ergebnisse. Es wurden zwei Algorithmen implementiert - LCM (Algorithmus 3.4.1) und LLCM (Algorithmus 3.4.2). Während der Erste die Lösungen des übergebenen Problems berechnet, ist der Zweite ein reiner Entscheidungsalgorithmus, der lediglich angibt, ob das Problem lösbar ist.

Wir fassen die Ergebnisse in Tabelle 5.1.1 zusammen.

Die Zeiten in der Tabelle stellen Reallaufzeiten dar. Sie wurden durch Haskell-Routinen gemessen und sind Ergebnisse der Durchschnittsbildung eines fünfzigmaligen Tests.

Die Zeiten zeigen, dass LLCM auch mit sehr großen Problemen gut umgehen kann. Die Berechnung der tatsächlichen Lösungen benötigt weitaus mehr Zeit. Dies liegt zum einen daran, dass die booleschen Operationen, die in LLCM verwendet werden, wesentlich kostengünstiger sind als die notwendigen Operationen der Formel-Logik zur Darstellung der semigelösten Formen in LCM, zum anderen aber auch daran, dass LLCM lediglich die erste Lösung zur Bestätigung der Lösbarkeit finden muss, LCM jedoch alle.

Diese Vermutung bestätigt sich durch einen Test mit LCM, bei dem lediglich die erste Lösung berechnet wird.

In Tabelle 5.1.2 geben wir der Vollständigkeit halber die Laufzeiten des Algorithmus zur Lösung linearer Kontextmatchingprobleme mit zufällig erzeugten Termen an. Die Termprofile sind `TermGeneratorParameter` gemäß Kapitel 4.3.6.2 und werden im folgenden Abschnitt (Tabelle 5.2.2) näher erläutert. Die Messungen erfolgten durch fünfzigfache Generierung von fünfzig zufälligen Kontextmatchingproblemen, ihrer Linearisierung und Bildung des Durchschnitts über die Laufzeiten zu ihrer Lösung.

5.2. Generelles Kontextmatching

5.2.1. Testszenario. Wie bereits in Kapitel 4.5.3.2 beschrieben, kann das implementierte Verfahren zur Lösung von Kontextmatching mit Strategien parameterisiert werden. Unser Testszenario umfasst die Verwendung von vier verschiedenen Strategien, die im Folgenden näher vorgestellt werden.

Das generelles Kontextmatching wurde mit Hilfe von zufälligen Termen, die der in Kapitel 4.3.4 vorgestellte Termgenerator erzeugte, getestet. Dabei wurden jeweils fünfzig Terme erzeugt und mit jeder zu testenden Strategie gelöst. Diese Vorgehensweise wurde fünfzig mal wiederholt. Danach wurden Durchschnittswerte über alle getesteten Termtyp-Strategie-Paare gebildet.

5.2.1.1. *Strategien und Heuristiken.* Die Tests wurden mit fünf Strategien durchgeführt, die in Tabelle 5.2.1 dargestellt sind. Alle fünf Strategien verwenden die in Kapitel 4.5.3.4 vorgestellte einfache Heuristik `take1Heuristic`, die unter den anwendbaren Regeln stets die Erste auswählt. Daher ist die Reihenfolge der Regeleintragung in die Strategie von erheblicher Bedeutung. Das Modul `Gersdorf.ContextMatching.Test.TestStrategies` enthält die genannten Strategien.

Folgende Überlegungen spielten bei der Festlegung, nur `take1Heuristic` zu verwenden, eine Rolle:

(1) `take1Heuristic` entscheidet ausschließlich aufgrund der Regelreihenfolge, d. h. die Heuristik benötigt keinerlei Informationen über das zu bearbeitende Problem oder die Ausgabe einer Regelanwendung auf ein Problem und ist insofern rechenunintensiv.
(2) Bei Heuristiken, die Einblick nehmen in Regelresultate, führt dies dazu, dass die Regelresultate aller verfügbaren Regeln auch tatsächlich berechnet werden müssen. Dieser zusätzliche Aufwand läuft jedoch dem Ziel, mit wenig Aufwand zu einer Lösung des betrachteten Kontextmatchingproblems zu kommen, zumindest teilweise zuwider.
(3) Bereits mit der Regelreihenfolge allein kann man eine Regelpriorisierung festlegen, z. B. dergestalt, dass Variablen-Split nur dann angewendet wird, wenn keine andere Regel anwendbar ist (durch Eintragung als letzte Regel in eine Strategie). Eine kostenbasierte Heuristik, die einzelnen Regeln Kosten - eventuell in Abhängigkeit von Termgrößen oder -eigenschaften - zuweist und daraufhin die Regel mit den lokal geringsten Kosten auswählt, scheint im Vergleich hierzu wenig Vorteile zu versprechen.

Strategie	Heuristik	Regeln
`baseStrategy`	`take1Heuristic`	Term- u. Kontextverschmelzung Bot-Elimination Dekomposition Lochdekomposition Kontextelimination Multikontext-Kollision Variablen-Split
`noSplitStrategy`	`take1Heuristic`	Term- u. Kontextverschmelzung Bot-Elimination Dekomposition Lochdekomposition Kontextelimination Multikontext-Kollision Multikontext-Dekomposition Bodendekomposition Variablen-Split
`paperStrategy`	`take1Heuristic`	Term- u. Kontextverschmelzung Bot-Elimination Dekomposition Lochdekomposition Kontextelimination Multikontext-Kollision Multikontext-Dekomposition Split: Linearisierung Bodendekomposition Variablen-Split
`extendedStrategy`	`take1Heuristic`	Term- u. Kontextverschmelzung Bot-Elimination Dekomposition Lochdekomposition Kontextelimination Multikontext-Kollision Konstantenelimination Multikontext-Dekomposition Split: Korrespond. Lochpfade Split: Linearisierung Bodendekomposition Variablen-Split
`extended2Strategy`	`take1Heuristic`	Term- u. Kontextverschmelzung Bot-Elimination Dekomposition Lochdekomposition Kontextelimination Multikontext-Kollision Konstantenelimination Multikontext-Dekomposition Split: Korrespond. Funktionen Split: Korrespond. Lochpfade Split: Linearisierung Bodendekomposition Variablen-Split

Tabelle 5.2.1. Getestete Strategien

Entscheidend ist also die Reihenfolge der Regeln in den einzelnen Strategien. Die Regel VARIABLEN-SPLIT kommt in allen Strategien als Letzte vor, da sie durch ihren don't know-Nichtdeterminismus sehr teuer ist. Die sehr einfachen Regeln wie DEKOMPOSITION sind dagegen weit vorne eingeordnet, da sie zum einen don't care-nichtdeterministisch und zum anderen sehr preiswert in der Anwendung sind.

Wir beschreiben kurz die Eigenschaften der einzelnen Strategien:

`baseStrategy`**:** Die Strategie beinhaltet lediglich die in Tabelle 3.4.1 dargestellten Basisregeln des Transformationsalgorithmus für Kontextmatchingprobleme. Die Regeln formen ein Regelwerk zur vollständigen und korrekten Lösung von Kontextmatchingproblemen.

`noSplitStrategy`**:** In dieser Strategie werden die beiden Regeln MULTIKONTEXT-DEKOMPOSITION und BODENDEKOMPOSITION hinzugefügt. Da die Regel BODENDEKOMPOSITION don't know-nichtdeterministisch ist, wird sie geringer priorisiert als MULTIKONTEXT-DEKOMPOSITION.

`paperStrategy`**:** Diese Strategie umfasst alle Regeln, die in dem dieser Diplomarbeit zugrundeliegenden Papier [**Schm2003**] vorgestellt wurden, mit Ausnahme der Methode diophantischer Gleichungen zur Ermittlung korrespondierender Positionen. Insofern wird die `noSplitStrategy` erweitert um SPLIT: LINEARISIERUNG.

`extendedStrategy`**:** In dieser Strategie wurden die zusätzlichen Regeln KONSTANTENELIMINATION und SPLIT: KORRESPONDIERENDE LOCHPFADE zur `paperStrategy` hinzugefügt. Da KORRESPONDIERENDE LOCHPFADE korrespondierende Positionen mit geringerem Aufwand als LINEARISIERUNG ermittelt, hat diese Regel höhere Priorität. Die KONTEXTELIMINATION besitzt eine entsprechend hohe Priorität, um rechtzeitig für die Regeln BODENDEKOMPOSITION und MULTIKONTEXT-DEKOMPOSITION die notwendigen Voraussetzungen (passende Löcher) zu schaffen.

extended2Strategy: Im Vergleich zur `extendedStrategy` wird die Regel SPLIT: KORRESPONDIERENDE FUNKTIONEN hinzugefügt. Diese hat eine höhere Priorität als die anderen SPLIT-Regeln, da sie don't-care-nichtdeterministrisch ist.

5.2.1.2. *Termtypen.* Die vorgestellten Strategien wurden an zufällig mit dem bereits in Kapitel 4.3.6.2 vorgestellten Termgenerator generierten Kontextmatchingproblemen getestet. Der Termgenerator ist - wie bereits besprochen - parameterisierbar. Wir stellen die verschiedenen verwendeten Parameterisierungen und mithin die dadurch eingeführten Termtypen in Tabelle 5.2.2 vor.

Die unterschiedlichen Termtypen legen jeweils den Schwerpunkt auf eine bestimmte Eigenschaft von Termen, sei es die Anzahl der Kontextvariablen, sei es die Anzahl individueller Variablen oder aber die Stelligkeit von Funktionen. Mit dieser Auswahl sollte ein breites Spektrum unterschiedlicher Terme abgedeckt sein.

TermTyp	arity	cvar	default	hole	manyFunc	small	var
termSize	50	50	50	50	50	15	50
variableSize	20	20	20	20	20	8	20
Var: #/Weight	4/30	2/20	2/20	2/20	4/70	2/20	4/70
cVar: #/Weight	3/20	4/70	3/20	3/50	3/20	3/20	3/20
Func: #/Weight	4/50	5/50	5/50	5/50	30/50	5/50	5/50
Hole: #/Weight	1/10	1/20	1/10	1/30	1/10	1/10	1/10
maxVar	5	2	5	5	10	3	10
maxCVar	2	4	2	2	2	2	2
maxArity	10	3	3	3	3	2	3

TABELLE 5.2.2. Termtypen

5.2.2. Ergebnisse. Für den Test wurden fünfzig mal fünfzig Terme jedes genannten Termtyps per Zufall erzeugt. Jedes Fünfzigerpack von Termen wurde mit jeder der vorgestellten Strategien gelöst; dabei wurden Statistikdaten gesammelt (`solveBacktrackStatistic`, siehe Kapitel 7.1.1). Das Ergebnis der Tests ist in Tabelle 5.2.3 zu sehen.

Während der Ausführung des Lösungsalgorithmus wurden auch Daten über die zufällig generierten Unifikationsprobleme gesammelt, welche in Tabelle 5.2.4 dargestellt sind. Die in der Tabelle abgetragenen Größen sind Durchschnittswerte über alle 2.500 (50 mal 50) generierten Probleme. Neben der durchschnittlichen Größe und Anzahl der Lösungen eines generierten Kontextmatchingproblems sind auch für die Knotentypen Funktion, Variable und Kontextvariable die Anzahl insgesamt vorkommender Knotentypen und die Anzahl unterschiedlicher Knotentypen aufgezeigt. Beispielsweise wäre im Term $f(f(x))$ die Anzahl der Funktionen gleich 2, die Anzahl unterschiedlicher Funktionen jedoch gleich 1.

In Tabelle 5.2.5 wird die prozentuale Einsparung in Transformationsschritten durch Einsatz der Strategie `extended2Strategy` im Vergleich zur Strategie `baseStrategy` dargelegt. Dabei wird differenziert nach den unterschiedlichen generierten Termtypen.

Man erkennt, dass durch Einsatz fortgeschrittener Transformationsregeln wie BODENDEKOMPOSITION, MULTIKONTEXT-DEKOMPOSITION sowie die unterschiedlichen SPLIT-Regeln ein erhebliches Maß an Einsparungen erzielt werden kann; im Testdurchschnitt kam es zu 83,7% Einsparungen. Dabei sind je nach Termtyp unterschiedlich große Einsparungen zu erzielen.

Betrachtet man die unterschiedlichen Termtypen und ihre Einsparungen, so stellt man fest, dass die Einsparung umso größer ist, je mehr Kontextvariablen und Löcher in einem Problem vorhanden sind. Mehr Kontextvariablen führen zu mehr potentieller Einsparung der SPLIT-Regeln, da durch diese VARIABLEN-SPLITS gespart werden. Mehr Löcher führen zu mehr Einsatz der Regeln BODENDEKOMPOSITION und MULTIKONTEXT-DEKOMPOSITION.

Ein Blick in Tabelle 5.2.3 verrät zudem, dass der Unterschied zwischen der `extended2Strategy` und der `paperStrategy` minimal ist. Dabei ist jedoch zu beachten, dass hier nicht der Gesamtberechnungsaufwand, sondern nur

Strategie	1	2	3	4	5	6	7	8	9	10	11	12
Durchschnitt aller Termtypen												
base	1403,6	439,6	433,1	4,8	13,2	0	0	0	0	0	0	512,0
noSplit	1130,3	378,1	362,6	1,8	8,9	0	0,2	0	0	0	1,2	376,5
paper	229,2	74,2	81,1	1,0	1,9	0	0,2	0	1,3	0	0,06	68,5
extended	229,2	74,2	80,7	1,2	1,8	0,4	0,2	0,7	0,5	0	0,05	68,5
extended2	229,0	74,1	80,7	1,3	1,8	0,4	0,2	0,6	0,4	0,2	0,05	68,3
cvarRTP												
base	5660,6	1844,7	1615,6	16,1	63,9	0	0	0	0	0	0	2119,3
noSplit	4602,2	1591,0	1366,5	2,6	43,0	0	0,27	0	0	0	1,9	1596,0
paper	846,9	298,7	247,0	1,5	8,5	0	0,3	0	2,7	0	0,2	287,0
extended	846,6	298,6	246,2	1,7	8,5	1,0	0,3	1,6	0,8	0	0,2	286,7
extended2	845,0	297,9	246,1	1,8	8,5	1,0	0,3	1,2	0,6	0,9	0,2	285,7
defaultRTP												
base	408,7	105,9	177,6	1,5	0,6	0	0	0	0	0	0	122,1
noSplit	347,3	94,1	158,2	1,5	0,5	0	0,1	0	0	0	1,1	90,7
paper	90,3	21,2	49,1	0,8	0,3	0	0,1	0	1,1	0	0,02	16,8
extended	90,4	21,2	48,8	0,9	0,3	0,3	0,1	0,5	0,6	0	0,02	16,8
extended2	90,4	21,2	48,8	0,9	0,3	0,3	0,1	0,5	0,5	0,1	0,02	16,8
holeRTP												
base	510,9	118,2	200,5	5,3	0,7	0	0	0	0	0	0	185,2
noSplit	301,0	83,8	126,2	3,4	0,4	0	0,4	0	0	0	2,1	83,7
paper	66,3	18,7	25,7	1,9	0,1	0	0,4	0	1,0	0	0,02	17,5
extended	66,4	18,7	25,5	2,0	0,1	0,2	0,4	0,6	0,3	0	0,02	17,5
extended2	66,4	18,7	25,5	2,0	0,1	0,2	0,4	0,6	0,3	0,07	0,02	17,5
smallRTP												
base	44,6	15,2	13,0	0,3	0,3	0	0	0	0	0	0	14,7
noSplit	40,9	14,0	12,0	0,4	0,3	0	0,01	0	0	0	0,3	12,9
paper	17,1	5,1	6,9	0,3	0,1	0	0,01	0	0,8	0	0,02	2,9
extended	17,2	5,1	6,4	0,6	0,1	0,4	0,05	0,4	0,3	0	0,02	2,9
extended2	17,3	5,1	6,4	0,7	0,1	0,4	0,06	0,4	0,2	0,3	0,02	2,9
varRTP												
base	393,3	113,7	158,8	0,8	0,5	0	0	0	0	0	0	118,5
noSplit	360,0	107,7	149,9	1,0	0,5	0	0,04	0	0	0	0,8	99,1
paper	125,5	27,4	76,8	0,5	0,2	0	0,04	0	1,1	0	0,02	18,4
extended	125,6	27,4	76,7	0,6	0,2	0,2	0,05	0,3	0,7	0	0,02	18,4
extended2	125,7	27,4	76,7	0,6	0,2	0,2	0,05	0,3	0,6	0,04	0,02	18,4

Tabelle 5.2.3. Ergebnisse generelles Kontextmatching

1=# Schritte, 2=Term- u. Kontextverschmelzung, 3=Dekomposition, 4=Lochdekomposition, 5=Kontextelimination, 6=Konstantenelimination, 7=Multikontext-Dekomposition, 8=Split: Korrespondierende Lochpfade, 9=Split: Linearisierung, 10=Split: Korrespondierende Funktionen, 11=Bodendekomposition, 12=Variablen-Split

Ø-Wert/Termtyp	`cVar`	`default`	`hole`	`small`	`var`
Größe	41,0	44,7	44,9	11,0	43,0
# Lösungen	50,5	5,7	5,6	2,0	7,1
# Funktionen	53,6	59,9	58,3	16,7	57,1
# untersch. Funktionen	4,7	4,8	4,8	4,0	4,8
# Löcher	2,4	1,4	3,4	0,5	0,9
# Variablen	1,0	1,5	1,1	0,5	3,1
# untersch. Variablen	0,8	1,0	0,9	0,5	2,2
# Kontextvariablen	4,0	1,9	2,0	1,4	1,9
# untersch. Kontextvar.	2,7	1,6	1,7	1,2	1,6

TABELLE 5.2.4. Auswertungen über generierte Terme

Termtyp	Schritte `base-Strategy`	Schritte `extended2-Strategy`	Ersparnis in Prozent
Ø alle Termtypen	1403,6	229,0	83,7
`cVar`	5660,6	845,0	85,1
`default`	408,7	90,4	77,9
`hole`	510,9	66,4	87,0
`small`	44,6	17,3	61,2
`var`	393,3	125,7	68,0

TABELLE 5.2.5. Einsparungen in Schritten `extendedStrategy` vs. `baseStrategy`

die Anzahl an Transformationsschritten gemessen wurde. Dementsprechend wird der geringere Berechnungsaufwand von SPLIT: KORRESPONDIERENDE FUNKTIONEN und SPLIT: KORRESPONDIERENDE LÖCHER gegenüber SPLIT: LINEARISIERUNG nicht berücksichtigt.

Da die SPLIT-Regeln jedoch gemäß Tabelle 5.2.3 äußerst sparsam angewendet werden (im Bereich 0,8-2,7 pro Problemlösung) und bereits damit im Übergang von der `noSplitStrategy` zur `paperStrategy` beeindruckende Schrittzahlreduzierungen erreicht werden, fällt der geringere Berechnungsaufwand von SPLIT: KORRESPONDIERENDE LÖCHER nicht stark ins Gewicht, zumal diese Methode zur Ausnutzung korrespondierender Löcher im Ergebnis die gleichen Korrespondenzen findet wie die LINEARISIERUNG. Gleiches gilt für SPLIT: KORRESPONDIERENDE FUNKTIONEN.

KAPITEL 6

Zusammenfassung und Ausblick

6.1. Zusammenfassung

In dieser Diplomarbeit wurde zunächst eine Einführung in das Gebiet der Unifikationstheorie gegeben, um dann zum Teilgebiet des Kontextmatchings zu kommen. Dieses wurde in das Gesamtgebiet der Unifikation eingeordnet. In Anlehnung an [**Schm2003**] wurde die Komplexität einiger Einschränkungen des Kontextmatchings betrachtet. Insbesondere wurde ein Algorithmus zur Lösung linearer Kontextmatchingprobleme in polynomieller Zeit vorgestellt.

Es folgte die Einführung des Transformationsalgorithmus aus [**Schm2003**] zur Lösung allgemeiner Kontextmatchingprobleme, wobei nach und nach verbesserte Transformationsregeln für einzelne spezielle Problemsituationen vorgestellt wurden. Über [**Schm2003**] hinausgehend wurden die Regeln Split: Korrespondierende Lochpfade und Konstantenelimination vorgestellt.

Im Rahmen der Diplomarbeit wurden die genannten Algorithmen in der funktionalen Programmiersprache Haskell implementiert, wobei auf eine einfache Erweiterbarkeit um neue Transformationsregeln sowie alternative Heuristiken zur Auswahl der in einem Schritt anzuwendenden Transformationsregel geachtet wurde.

Die Implementierung (und damit auch die in ihr implementierten Algorithmen) wurde mit Hilfe von zufällig erzeugten Termen auf ihre Leistungsfähigkeit getestet. Hauptaugenmerk lag dabei darauf, inwiefern sich Regeln, die über die Basisregeln aus Tabelle 3.4.1 hinausgehen, positiv auf die Anzahl der Transformationsschritte auswirken.

6.2. Ergebnisse

Das Ergebnis ist beeindruckend: durch die Einführung komplexerer Transformationsregeln ließen sich in unseren Testfällen bis zu 87% der Transformationsschritte einsparen, im Durchschnitt immerhin noch 83%. Speziell komplexere Kontextmatchingprobleme mit einer größeren Anzahl an Kontextvariablen profitieren hiervon.

Insbesondere die Erkennung korrespondierender Positionen in Verbindung mit der Regel Split führte zu erheblichen Verbesserungen.

Die implementierten Algorithmen zur Erkennung korrespondierender Positionen stellen teilweise nur ein notwendiges Kriterium für die Existenz korrespondierender Löcher dar. Dies kann zu fehlerhaften Erkennungen

solcher Positionen führen. Wie sich in unseren Tests zeigte, scheint das jedoch kein gravierendes Problem zu sein, da die entsprechenden SPLIT-Transformationen ohnehin äußerst sparsam eingesetzt werden.

6.3. Ausblick

Interessant wäre die Untersuchung der Anwendbarkeit von Kontextmatching auf praktische Probleme wie der Übersetzung von XPath-Ausdrücken in Kontextmatching-Anfragen. Leider sind hierzu umfangreiche Änderungen am Lösungsalgorithmus notwendig: Zum einen erlaubt XML in seinen Baumknoten, die den Funktionen im Kontextmatching entsprechen, unterschiedliche Stelligkeiten. Funktionen jedoch haben eine feste Stelligkeit, so dass der Algorithmus so angepasst werden müsste, dass Funktionen mit variabler Argumentanzahl unterstützt werden. Zum anderen erlaubt XML auch - insbesondere bei Attributen - mehrere äquivalente Argumentreihenfolgen. Beispielsweise ist ein XML-Knoten `<Name Alter=''23'' Wohnort=''Frankfurt'' />` äquivalent zu `<Name Wohnort=''Frankfurt'' Alter=''23'' />`. Ähnliches gilt auch - je nach XSchema - für die Reihenfolge von Kindknoten.

Für weitere Verbesserungen des Transformationsalgorithmus wäre es interessant, das Verfahren zur Erkennung korrespondierender Positionen mittels linearer Gleichungssysteme (Kapitel 3.4.4.2) zu implementieren, da dieser andere korrespondierende Positionen als die LINEARISIERUNG erkennen könnte.

Darüber hinaus könnte man das Verfahren KORRESPONDIERENDE LOCHPFADE dadurch verbessern, dass in ihm statt der LINEARISIERUNG ein spezielles Verfahren zu Lösung monadischer Kontextmatchingprobleme mit nur einer Gleichung eingesetzt wird. Dadurch könnte der bedingte don't know-Nichtdeterminismus durch einen don't care-Nichtdeterminismus ersetzt werden. Da diese Regel jedoch nach unserem Testszenario nur äußerst selten angewendet wird, ist nicht mit wesentlichen Laufzeitreduzierungen zu rechnen.

Wesentlichere Verbesserungen sind zu erwarten, wenn intelligentere Mechanismen zur Regelauswahl (Heuristiken) entwickelt würden. Dabei ist jedoch darauf zu achten, dass Heuristiken Informationen zu ihrer Entscheidungsfindung benötigen. Die Kosten der Informationsgewinnung dürfen nicht größer sein als die potentielle Laufzeiteinsparung durch bessere Regelauswahlen. Insofern ist das Heranziehen von Regelanwendungsresultaten zur Entscheidungsfindung ungeeignet für eine Heuristik. In dieser Diplomarbeit wurde zunächst auf eine solche Entwicklung besserer Heuristiken als der vorgestellten einfachen verzichtet; die notwendigen Voraussetzungen hierfür wurden jedoch mit der Implementierung geschaffen.

KAPITEL 7

Anhang

7.1. Funktionen

Wir geben in diesem Kapitel der Vollständigkeit halber Funktionen und Datentypen an, die im Hauptteil bewusst weggelassen sind.

7.1.1. solveBacktrackStatistic.

```
solveBacktrackStatistic :: (Ord transformType) =>
  (node -> BacktrackingChoice node)
  -> (node -> Bool)
  -> (node -> transformType)
  -> node
  -> BacktrackingStatistic transformType
solveBacktrackStatistic
  getSucc isGoal getTransformType cur =
  let nodeStream       =
        solveBacktrackR2 getSucc isGoal cur
      emptyBTStatistic =
        BacktrackingStatistic {
          btsTransformMap = emptyFM,
          btsSteps = 0,
          btsNoSolutions = 0 }

     calculateStatistic [] btStatistic = btStatistic
     calculateStatistic (node:nodes) btStatistic
        | isGoal node =
            let goalStatistic =
                  nextStatistic {
                    btsNoSolutions =
                   1 + (btsNoSolutions nextStatistic)
                   }
            in  (calculateStatistic  nodes)
                $! goalStatistic
        | otherwise =
            (calculateStatistic nodes)
            $! nextStatistic

        where
       nextTransformType = getTransformType node
```

```
        nextStatistic =
          (updateBTStatistic nextTransformType)
          $! btStatistic

  in  calculateStatistic nodeStream emptyBTStatistic

updateBTStatistic :: (Ord transformType) =>
  transformType
  -> BacktrackingStatistic transformType
  -> BacktrackingStatistic transformType
updateBTStatistic tt bts =
    let newMap = (updateFM tt 1 (\i -> 1 + i))
                           (btsTransformMap bts)
    in  BacktrackingStatistic {
            btsSteps = (1 + btsSteps bts),
            btsTransformMap =  newMap,
            btsNoSolutions = btsNoSolutions bts
        }

data (Ord transformType) =>
  BacktrackingStatistic transformType =
    BacktrackingStatistic {
        btsTransformMap ::
          !(TreeFiniteMap transformType Int),
        btsSteps        :: !Int,
        btsNoSolutions  :: !Int
    } deriving Show
```

7.2. Beispielberechnung

In diesem Kapitel lösen wir das Problem

$$\{X_2\,[X_1\,[X_1\,[f_3\,(\square, X_1\,[f_2\,(f_0)])]]] \approx f_1\,(f_4\,(f_4\,(f_3\,(\square, f_2\,(f_0)))))\}$$

mit den verschiedenen in Tabelle 5.2.1 aufgeführten Strategien.

7.2.1. baseStrategy. Wir zeigen zunächst stets den jeweiligen Weg zu einer Lösung, wie ihn das implementierte System zur Lösung von Kontextmatching-Problemen ausgibt (Funktion `solveGCMVerbose`). Diese Ausgabe zeigt nur diejenigen Transformationsschritte, die zu der jeweils gefundenen Lösung führten. Eventuell vorher oder nachher durchgeführte Schritte, die zu keiner Lösung führten, werden nicht gezeigt.

```
ComplexNode. Taken: 10.
Current: <[] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
Steps:
[Identity] <[X2[X1[X1[f3(_,X1[f2(f0())])]]] =?
  f1(f4(f4(f3(_,f2(f0()))))))] :: {  }>
[VariableSplit] <[X2[_] =? f1(f4(f4(_))),
  X1[X1[f3(_,X1[f2(f0())])]] =? f3(_,f2(f0()))] :: {  }>
[solvedDummy] <[X1[X1[f3(_,X1[f2(f0())])]] =?
```

# Schritte	64
Identität	1
Term- u. Kontextverschmelzung	20
Dekomposition	4
Lochdekomposition	2
Kontextelimination	7
Variablen-Split	30

Tabelle 7.2.1. `solveGCMStatistic baseStrategy`

```
  f3(_,f2(f0()))] :: { (X2,f1(f4(f4(_)))) }>
[VariableSplit] <[X1[_] =? _,
  X1[f3(_,X1[f2(f0())])] =? f3(_,f2(f0()))]
  :: { (X2,f1(f4(f4(_)))) }>
[solvedDummy] <[X1[f3(_,X1[f2(f0())])] =?
  f3(_,f2(f0()))] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[ctxElimination] <[f3(_,X1[f2(f0())]) =?
  f3(_,f2(f0()))] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[decompose/collision] <[_ =? _,X1[f2(f0())] =?
  f2(f0())] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[holeDecompose] <[X1[f2(f0())] =? f2(f0())]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[ctxElimination] <[f2(f0()) =? f2(f0())]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[decompose/collision] <[f0() =? f0()]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[decompose/collision] <[]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
```

Zur Übersicht sind die statistischen Informationen (Funktion `solveGCMStatistic`) in Tabelle 7.2.1 zusammengefasst.

7.2.2. noSplitStrategy.

```
ComplexNode. Taken: 10.
Current: <[] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
Steps:
[Identity] <[X2[X1[X1[f3(_,X1[f2(f0())])]]] =?
  f1(f4(f4(f3(_,f2(f0()))))))] :: {  }>
[Bottom Decompose/Collision] <[X2[X1[X1[_]]] =?
  f1(f4(f4(_))),_ =? _,X1[f2(f0())] =? f2(f0())] :: {  }>
[holeDecompose] <[X2[X1[X1[_]]] =? f1(f4(f4(_)))
  ,X1[f2(f0())] =? f2(f0())] :: {  }>
[VariableSplit] <[X2[_] =? f1(f4(f4(_))),
  X1[X1[_]] =? _,X1[f2(f0())] =? f2(f0())] :: {  }>
[solvedDummy] <[X1[X1[_]] =? _,
  X1[f2(f0())] =? f2(f0())] :: { (X2,f1(f4(f4(_)))) }>
[VariableSplit] <[X1[_] =? _,X1[_] =? _,
  X1[f2(f0())] =? f2(f0())] :: { (X2,f1(f4(f4(_)))) }>
[solvedDummy] <[X1[_] =? _,X1[f2(f0())] =? f2(f0())]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[solvedDummy] <[X1[f2(f0())] =? f2(f0())]
```

# Schritte	36
Identität	1
Term- u. Kontextverschmelzung	16
Dekomposition	2
Lochdekomposition	1
Kontextelimination	1
Bodendekomposition	1
Variablen-Split	14

Tabelle 7.2.2. `solveGCMStatistic noSplitStrategy`

```
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[ctxElimination] <[f2(f0()) =? f2(f0())]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[decompose/collision] <[f0() =? f0()]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[decompose/collision] <[]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
```

7.2.3. paperStrategy.

```
ComplexNode. Taken: 12.
Current: <[] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
Steps:
[Identity] <[X2[X1[X1[f3(_,X1[f2(f0())])]]] =?
  f1(f4(f4(f3(_,f2(f0()))))))] :: {  }>
[Split: correspondingLinear] <[X2[X1[X1[_]]] =?
  f1(f4(f4(_))),f3(_,X1[f2(f0())]) =? f3(_,f2(f0()))] :: {  }>
[decompose/collision] <[_ =? _,X1[f2(f0())] =? f2(f0())
  ,X2[X1[X1[_]]] =? f1(f4(f4(_)))] :: {  }>
[holeDecompose] <[X1[f2(f0())] =? f2(f0())
  ,X2[X1[X1[_]]] =? f1(f4(f4(_)))] :: {  }>
[Split: correspondingLinear] <[X1[f2(_)] =? f2(_)
  ,f0() =? f0(),X2[X1[X1[_]]] =? f1(f4(f4(_)))] :: {  }>
[decompose/collision] <[X1[f2(_)] =? f2(_)
  ,X2[X1[X1[_]]] =? f1(f4(f4(_)))] :: {  }>
[Bottom Decompose/Collision] <[X1[_] =? _,_ =? _,
  X2[X1[X1[_]]] =? f1(f4(f4(_)))] :: {  }>
[solvedDummy] <[_ =? _,X2[X1[X1[_]]] =?
  f1(f4(f4(_)))] :: { (X1,_) }>
[holeDecompose] <[X2[X1[X1[_]]] =? f1(f4(f4(_)))]
  :: { (X1,_) }>
[VariableSplit] <[X2[_] =? f1(f4(f4(_))),
  X1[X1[_]] =? _] :: { (X1,_) }>
[solvedDummy] <[X1[X1[_]] =? _]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[ctxElimination] <[X1[_] =? _]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[solvedDummy] <[]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
```

# Schritte	22
Identität	1
Term- u. Kontextverschmelzung	6
Dekomposition	2
Lochdekomposition	2
Kontextelimination	4
Split: Linearisierung	2
Bodendekomposition	1
Variablen-Split	4

Tabelle 7.2.3. `solveGCMStatistic paperStrategy`

# Schritte	20
Identität	1
Term- u. Kontextverschmelzung	6
Lochdekomposition	2
Kontextelimination	4
Konstantenelimination	1
Multikontext-Dekomposition	1
Bodendekomposition	1
Variablen-Split	4

Tabelle 7.2.4. `solveGCMStatistic extendedStrategy`

7.2.4. extendedStrategy.

```
ComplexNode. Taken: 10.
Current: <[] :: { (X1,_), (X2,f1(f4(f4(_)))) }>
Steps:
[Identity] <[X2[X1[X1[f3(_,X1[f2(f0())])]]]
  =? f1(f4(f4(f3(_,f2(f0()))))))] :: {  }>
[constantEliminate] <[X2[X1[X1[f3(_,X1[f2(_)])]]]
  =? f1(f4(f4(f3(_,f2(_)))))] :: {  }>
[Multicontext Decompose/Collision] <[X2[X1[X1[_]]]
  =? f1(f4(f4(_))),_ =? _,X1[f2(_)] =? f2(_)] :: {  }>
[holeDecompose] <[X2[X1[X1[_]]] =? f1(f4(f4(_)))
  ,X1[f2(_)] =? f2(_)] :: {  }>
[Bottom Decompose/Collision] <[X1[_] =? _,_ =? _
  ,X2[X1[X1[_]]] =? f1(f4(f4(_)))] :: {  }>
[solvedDummy] <[_ =? _,X2[X1[X1[_]]] =?
  f1(f4(f4(_)))] :: { (X1,_) }>
[holeDecompose] <[X2[X1[X1[_]]] =?
  f1(f4(f4(_)))] :: { (X1,_) }>
[VariableSplit] <[X2[_] =? f1(f4(f4(_)))
  ,X1[X1[_]] =? _] :: { (X1,_) }>
[solvedDummy] <[X1[X1[_]] =? _]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[ctxElimination] <[X1[_] =? _]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
[solvedDummy] <[]
  :: { (X1,_), (X2,f1(f4(f4(_)))) }>
```

Kategorie	base	noSplit	paper	extended	extended2
# Schritte	33.864	33.864	2.387	2.387	2.274
Identität	1	1	1	1	1
Term- u. Kontextverschm.	14.983	14.983	828	828	782
Dekomposition	2.151	2.151	70	70	71
Lochdekomposition	0	0	0	0	0
Kontextelimination	1.790	1.790	0	0	0
Konstantenelimination	0	0	0	0	0
Multikontext-Dekomposition	0	0	0	0	0
Split: Linearisierung	0	0	1	1	1
Split: Korresp. Funktionen	0	0	0	0	9
Bodendekomposition	0	0	0	0	0
Variablen-Split	14.939	14.939	1.487	1.487	1.410

Tabelle 7.2.5. `solveGCMStatistic`

7.2.5. extended2Strategy. Die extended2Strategy hat bei dem zu lösenden Problem denselben Lösungsweg wie extendedStrategy. Um einen Unterschied zwischen beiden Strategien zu demonstrieren, betrachten wir nun das Problem

```
[X1[X2[X0[f4(X3[X2[f2(f4(f2(f4(f4(f2(f2(f3(f4(v0),
  f2(f2(f4(v0))))))))))]])]]] =?
f1(f4(f4(f3(f2(f4(f4(f3(f3(f3(f2(f2(f4(f2(f4(f4(
  f2(f2(f3(f4(f1(f2(f2(f0())))),
  f2(f2(f4(f1(f2(f2(f0()))))))))))))))),f0()),
  f4(f0())),f4(f4(f0()))))),f0())))]
```

Das Problem besitzt 22 Lösungen. Tabelle 7.2.5 zeigt den Unterschied in der Anzahl Regelanwendungen.

Abbildungsverzeichnis

Index

Literaturverzeichnis

[Baad1998] Franz Baader und Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, UK, 1998.

[Baad1999] Franz Baader und Walter Snyder. Unification Theory. In Alan Robinson und Andrei Voronkow, Herausgeber, *Handbook of automated Reasoning.* Elsevier Science Publishers B. V., 1999.

[Bar1984] Henk P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

[Bird1998] Richard Bird. *Introduction to Functional Programming using Haskell, Second Edition.* Pearson Education, Essex, UK, 1998.

[Com1997] Hubert Comon und Yan Jurski. Higher-order matching and tree automata. In *Proc. Conf. on Computer Science Logic (CSL-97)*, Lecture Notes in Computer Science, 1414:157-176. Springer, Aarhus, 1997.

[Dow2001] Gilles Dowek. Higher-Order Unification and Matching. In Alan Robinson und Andrei Voronkow, Herausgeber, *Handbook of automated Reasoning*, Band 2, Kapitel 16, Seiten 1009-1062. North-Holland, 2001.

[GHC2004] GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 6.2.* http://www.haskell.org/ghc/docs/latest/html/users_guide/users_guide.html, 2004.

[Gold1981] W. D. Goldfarb. The undecidability of the second-order unification problem, *Theoretical Computer Science,* 13:225-230, 1981.

[Hop1994] John E. Hopcroft, Jeffrey Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* 3. Auflage. Addison-Wesley, Bonn, 1994.

[Hos2000] Haruo Hosoya und Benjamin Pierce. *Regular Expression Pattern Matching for XML*, 2000.

[Hud2000] Paul Hudak. *The Haskell School of Expression: learning functional programming through multimedia.* Cambridge University Press, Cambridge, UK, 2000.

[Huet1978] Gérard Huet und Bernhard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica,*11:31-35, 1978.

[Jon1995] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. *First international Spring School on Advanced Functional Programming Techniques*, Lecture Notes in Computer Science, 925. Springer, Bastad, Schweden, 1995.

[Jon2000] Mark P. Jones. Type Classes with Functional Dependencies. *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, Lecture Notes in Computer Science, 1728. Springer, Berlin, 2000.

[Jon2003] Mark P. Jones et al. *The Hugs 98 User's Guide.* http://cvs.haskell.org/Hugs/pages/users_guide/index.html, 2003.

[Lass1987] J.-L. Lassez, M. Maher und K. Mariott. Unification revisited. In J. Minker, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufman, Los Altos, Kalifornien, 1987.

[Mar2002] Simon Marlow. Antwort auf „Random questions after a long haskell coding day". *Mail auf Mailingliste „Haskell-Cafe".* http://www.haskell.org/pipermail/haskell-cafe/2002-January/002614.html, 2002.

[Oka1998] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK, 1998.

[Oes2001] Bernd Oestereich. *Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language.* Oldenburg Wissenschaftsverlag, München, Wien, 2001.

[Pad2000] Vincent Padovani. Decidability of fourth-order matching. *Mathematical Structures in Computer Science*, 10(3):361-372, 2000.

[Pat1978] Mike S. Paterson und Mark N. Wegman. Linear unification. *Journal Computer and System Sciences*, 16:158-167, 1978.

[Pey2002] Simon Peyton-Jones et al. *Haskell 98 Language and Libraries. The revised report.* http://www.haskell.org/onlinereport/, 2002.

[Rabh1999] Fethi Rabhi und Guy Lapalme. *Algorithms: a functional approach.* Pearson Education, Essex, UK, 1999.

[Schm2000] Manfred Schmidt-Schauß. *Funktionale Programmierung I.* Johann Wolfgang Goethe-Universität Frankfurt, Vorlesung, Sommersemester 2000.

[Schm2003-2] Manfred Schmidt-Schauß. Decidability of Arity-Bounded Higher-Order Matching. *Lecture Notes in Computer Science*, 2741:488-502. Springer, Berlin, 2003.

[Schm2003] Manfred Schmidt-Schauß und Jürgen Stuber. On the complexity of linear and stratified context matching problems. *ACM TOCS*, accepted for publication. 2004.

[Schm1998] Manfred Schmidt-Schauß und Klaus U. Schulz. On the exponent of periodicity of minimal solutions of context equations. In *Proc. 9th Int. Conf. on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, 1379:61-75. Springer, Tsukuba, Japan, 1998.

[Stu2003] Jürgen Stuber und K. Vikram. *Experiences with an Implementation of Context Matching.* Draft.