TUHH
Hamburg University of Technology

Bachelorarbeit

Bruno Borchardt

Entwicklung eines Termersetzungssystems für assoziative und kommutative Ausdrücke

15. Juli 2021

betreut von:
PD Dr. Prashant Batra
Prof. Dr. Siegfried Rump

Technische Universität Hamburg (TUHH)
Hamburg University of Technology
Institut für Zuverlässiges Rechnen
21073 Hamburg

IRC

Versicherung an Eides statt

Ich, Bruno Borchardt, versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Rauris, am ....
(Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Die Manipulation symbolischer Ausdrücke ist in der Mathematik allgegenwärtig. Ist bekannt, dass ein Ausdruck A gleichbedeutend zu einem zweiten Ausdruck B ist, so kann in einem dritten Ausdruck C jedes Vorkommen von A durch B ersetzt werden. Wenn bekannt ist, dass $A = 4$ und $B = 2 \cdot 2$ die gleiche Bedeutung haben, kann beispielsweise der Ausdruck $C = \frac{4}{2}$ auch als $C' = \frac{2 \cdot 2}{2}$ geschrieben werden.
Oft ist für zwei Ausdrücke bekannt, dass sie gleichbedeutend sind, wenn beide jeweils einer bestimmten Struktur folgen. Im Beispiel kann C' auch als $C'' = 2$ geschrieben werden, da unabhängig konkreter Werte von x und y feststeht, dass der Ausdruck $\frac{x \cdot y}{x}$ auch als y geschrieben werden kann.
Solche Regeln von Hand anzuwenden, ist sowohl zeitaufwändig als auch fehleranfällig. Die Idee, Computer zu nutzen, um symbolische Ausdrücke zu manipulieren, ist deshalb fast so alt wie der Computer selbst. LISP ist als eine der ersten höheren Programmiersprachen bereits für diesen Zweck geschaffen worden [McC60]. Rückblickend war die symbolische Berechnung sogar vor der Erfindung des Computers ein wichtiger Teil dessen, was heute theoretische Informatik genannt wird. Herausstechend ist die Idee des Lambdakalküls von Church [Chu36].
Gegenwärtig gibt es eine Reihe etablierter Systeme zum symbolischen Rechnen. Wichtiger Bestandteil sind diese etwa in Mathematica [21c], Maple [21e] oder Matlab [21d]. Konzeptionelle Grundlage dieser Anwendungen ist die des Termersetzungssystems: Mit einer vom Nutzer oder Bibliotheksautor bestimmten Menge von Ersetzungsregeln wird ein gegebener Ausdruck so lange transformiert, bis keine Regel mehr anwendbar ist.

Beispiel 1.1. Es werden vier Ersetzungsregeln definiert.

$$
\begin{aligned}
x \cdot y + x \cdot z &= x \cdot (y + z) && (1)\\
\sqrt{x} &= x^{\frac{1}{2}} && (2)\\
(x^y)^z &= x^{y \cdot z} && (3)\\
(\sin x)^2 + (\cos x)^2 &= 1 && (4)
\end{aligned}
$$

Der folgende Term wird durch Anwendung der Regeln transformiert. Weiter werden Ausdrücke ohne Unbekannte ausgewertet.

$$
\begin{aligned}
3 \cdot (\sin(a+b))^2 + 3 \cdot \sqrt{(\cos(a+b))^4} &\overset{(1)}{=} 3 \cdot \left((\sin(a+b))^2 + \sqrt{(\cos(a+b))^4} \right)\\
&\overset{(2)}{=} 3 \cdot \left((\sin(a+b))^2 + \left((\cos(a+b))^4 \right)^{\frac{1}{2}} \right)\\
&\overset{(3)}{=} 3 \cdot \left((\sin(a+b))^2 + (\cos(a+b))^{4 \cdot \frac{1}{2}} \right)\\
&= 3 \cdot \left((\sin(a+b))^2 + (\cos(a+b))^2 \right)\\
&\overset{(4)}{=} 3 \cdot 1\\
&= 3
\end{aligned}
$$

Wird eine Ersetzungsregel angewandt, stehen die Variablen der Regel dabei stellvertretend für die entsprechenden Teile des zu transformierenden Ausdrucks. Formalisiert wird diese Beziehung in Abschnitt 2.3. Als Beispiel gilt für die erste Umformung $y = (\sin(a+b))^2$ mit y aus Regel (1). Anwendung finden Regeln ausschließlich von links nach rechts.

Der Einfluss von Termersetzungssystemen geht weit über die Anwendung durch Mathematiker oder Ingenieure mit direktem Interesse am vereinfachten Ausdruck hinaus. Interessant ist etwa die Formulierung von Ersetzungsregeln im Kontext eines optimierenden Compilers [19; Jon96]. Termersetzungssysteme können weiter direkt die Grundlage der Auswertung eines funktionalen Programms sein [Jon87].

1.1 Zielsetzung

Ziel der Arbeit ist Design und Umsetzung eines Termersetzungssystems zur Vereinfachung algebraischer Ausdrücke. Der Kern des Termersetzungssystems ist ein Algorithmus zur Erkennung eines bestimmten Musters in einem Term. Der Algorithmus soll ein Muster dabei nicht nur erkennen, wenn der Term die exakt identische Struktur aufweist. Bestimmte Äquivalenzklassen, wie etwa alle Permutationen der Parameter in einer kommutativen Funktion, sollen bereits auf der Mustererkennungsebene berücksichtigt werden. Die Formulierung einer Menge von Ersetzungsregeln für das Termersetzungssystem muss also entsprechend kompakt möglich sein. Damit ist das Ziel, die Mustererkennung möglichst schnell durchführen zu können, beim Treffen von Designentscheidungen in der Musterstruktur nicht ausschlaggebend, eine polynomielle Laufzeitkomplexität ist allerdings angestrebt.
Das Leistungsvermögen des entwickelten Termersetzungssystems wird in einer Anwendung zur Vereinfachung algebraischer Ausdrücke über den komplexen Zahlen getestet.

1.2 Aufbau der Arbeit

In Abschnitt 2 werden die Begriffe eingeführt, die zur Beschreibung eines zu transformierenden Ausdrucks, aber auch zur Beschreibung der Transformation selbst notwendig sind. Mit den dann etablierten Begriffen werden die Algorithmen zur Normalisierung ohne Mustererkennung aus Kapitel 3 und die Algorithmen zur Mustererkennung in Abschnitt 4 erläutert. Die tatsächliche Umsetzung und ihre Abweichungen von vorhergehenden Ideen ist in Kapitel 5 behandelt. Kapitel 6 fasst die Arbeit zusammen.

2 Grundlegende Definitionen

2.1 Term

Eine Menge von Termen T ist in dieser Arbeit immer in Abhängigkeit der nicht-leeren Mengen F und C, sowie der Stelligkeitsfunktion $arity\colon F \to \mathbb{N} \cup \{\omega\}$ definiert, ähnlich der Notation von Benanav et. al. in [BKN87]. F enthält die sogenannten Funktionssymbole. Beispiele für mögliche Elemente in F sind `sin` und `sqrt`, zudem auch Operatoren wie die Division, etwa geschrieben als `divide`. Die Stelligkeitsfunktion *arity* gibt für jedes Funktionssymbol an, wie viele Argumente erwartet werden. Eine mögliche Stelligkeit der genannten Beispielsymbole ist die folgende:

$$arity\ f = \begin{cases} 2 & f = \texttt{divide} \\ 1 & f \in \{\texttt{sin}, \texttt{sqrt}\} \end{cases}$$

Kann ein Funktionssymbol f beliebig viele Argumente entgegennehmen, wird gesagt, dass f variadische Stelligkeit hat oder variadisch ist. Die Stelligkeitsfunktion bildet f dann auf ω ab.
Die Menge C enthält die Konstantensymbole. Mit den genannten Beispielen für Funktionssymbole ergibt etwa $C = \mathbb{R}$ Sinn. Wichtig ist, dass im Folgenden nicht vorausgesetzt wird, dass zwangsweise jedem Konstantensymbol ein eindeutiger numerischer Wert zugeordnet werden kann[1].

Definition 2.1. Ein Term $t \in T(F, C)$ ist

- ein Konstantensymbol, also $t \in C$
- oder eine Funktionsanwendung des Funktionssymbols $f \in F$ mit $arity\ f \in \{n, \omega\}$ auf die Argumente $t_1, \ldots, t_n \in T(F, C)$, geschrieben $t = (f, t_1, \ldots, t_n)$

In Mengenschreibweise:

$$T(F, C) \coloneqq C \cup \{(f, t_1, \ldots, t_n) \mid f \in F,\ arity\ f\ \in \{n, \omega\},\ t_1, \ldots, t_n \in T(F, C)\}$$

Eine Funktionsanwendung wird in der Literatur oft mit dem Funktionssymbol außerhalb des Tupels geschrieben ([ODo77], [BKN87]), also $f(t_1, \ldots, t_n)$ statt $(f, t_1, \ldots, t_n)$. Zum deutlicheren Abheben von Funktionen, die auf Termen operieren, zu Termen selbst, wird diese Schreibweise hier keine Verwendung finden.

Beispiel 2.2. Beispielsweise lässt sich der Ausdruck $\frac{3}{\sin 1}$ in der formalen Schreibweise als Term mittels der Funktionssymbole `sin` und `divide` sowie den Konstantensymbolen 3 und 1 darstellen als $(\texttt{divide}, 3, (\texttt{sin}, 1))$. Ein Term kann als Baum[2] aufgefasst werden, siehe das aktuelle Beispiel in Abb. 2.1.1.

[1] Die Symbole unbekannten Wertes werden häufig von den Konstantensymbolen getrennt und Variablensymbole genannt. Diese Unterscheidung wird hier nicht getroffen, primär um die Definitionen einfach zu halten.

[2] In der theoretischen Informatik auch Syntaxbaum oder AST (englisch für *Abstract Syntax Tree*)

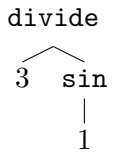

Abbildung 2.1.1: Baumdarstellung des Terms $(\texttt{divide}, 3, (\texttt{sin}, 1))$

Mit dem Kontext der Baumdarstellung lassen sich nun die folgenden Begriffe auf Terme übertragen. In der Funktionsanwendung $t = (f, t_1, \ldots, t_n)$ sind $t_1, \ldots, t_n$ die Kinder ihres Vaters t. Ein Kind ist allgemeiner ein Nachkomme. Nachkommen verhalten sich transitiv, also ein Nachkomme z des Nachkommen y von x ist auch ein Nachkomme von x. Umgekehrt ist x Ahne von y und z.

2.2 Funktionsauswertung

Die Erweiterung des Funktionssymbols zur Funktion, die von einem Raum Y^n nach Y abbildet, folgt mittels der *eval* Funktion frei nach O'Donnell [ODo77].

$$eval\colon \left(F \to \bigcup_{n\in\mathbb{N}} Y^n \to Y\right) \times (C \to Y) \to T \to Y$$

$$eval(u,v)\ t = \begin{cases} u\ f\ (eval(u,v)\ t_1, \ldots, eval(u,v)\ t_n) & t = (f, t_1, \ldots, t_n) \\ v\ t & t \in C \end{cases}$$

Gilt $arity\ f = n \in N$ für ein $f \in F$, ist zudem die Funktion $u\ f\colon Y^n \to Y$ in ihrer Definitionsmenge auf Dimension n eingeschränkt. Die Funktion u wird als Interpretation der Funktionssymbole F, die Funktion v als Interpretation der Konstantensymbole C und die Funktion $eval(u,v)\colon T \to Y$ als Interpretation der Terme T bezeichnet. Ein Element des Bildes einer Interpretation oder die Abbildung dahin wird als Auswertung bezeichnet.

Beispiel 2.3. Sei $F = \{\texttt{sum}, \texttt{prod}, \texttt{neg}\}$ und $C = \mathbb{N}$ mit $arity\ \texttt{sum} = arity\ \texttt{prod} = \omega$ und $arity\ \texttt{neg} = 1$. Für die Interpretation $eval(u,v)$ können u und v so gewählt werden, dass jeder Term in T zu einer ganzen Zahl $n \in \mathbb{Z}$ auswertbar ist.

$$u\ \texttt{sum}\ (y_1, \ldots, y_n) = \sum_{k=1}^{n} y_k$$
$$u\ \texttt{prod}\ (y_1, \ldots, y_n) = \prod_{k=1}^{n} y_k$$
$$u\ \texttt{neg}\ y = -y$$

$$v\ y = y$$

Hervorzuheben ist dabei, dass $u\ \texttt{neg}\colon \mathbb{Z} \to \mathbb{Z}$ nur eine ganze Zahl als Parameter erwartet, während u `sum` und u `prod` Tupel ganzer Zahlen beliebiger Länge abbilden können. Der Term $t = (\texttt{sum}, 3, (\texttt{prod}, 2, 4), (\texttt{neg}, 1))$ kann dann ausgewertet werden zu

$$\begin{aligned}
eval(u, v)\ t &= eval(u, v)(\texttt{sum}, 3, (\texttt{prod}, 2, 4), (\texttt{neg}, 1)) \\
&= u\ \texttt{sum}\ (eval(u, v)\ 3, eval(u, v)(\texttt{prod}, 2, 4), eval(u, v)(\texttt{neg}, 1)) \\
&= u\ \texttt{sum}\ (v\ 3, u\ \texttt{prod}\ (eval(u, v)\ 2, eval(u, v)\ 4), u\ \texttt{neg}\ (v\ 1)) \\
&= u\ \texttt{sum}\ (3, u\ \texttt{prod}\ (v\ 2, v\ 4), u\ \texttt{neg}\ 1) \\
&= u\ \texttt{sum}\ (3, u\ \texttt{prod}\ (2, 4), -1) \\
&= u\ \texttt{sum}\ (3, 8, -1) \\
&= 10
\end{aligned}$$

2.2.1 Konstruktor

Eine direkt aus der Struktur des Terms folgende Interpretation u_c für Funktionssymbole ist die des Konstruktors. Als Konstruktor eines Typs A wird allgemein eine Funktion bezeichnet, die nach A abbildet. Für typlose Terme ist diese Definition deshalb unzureichend. Insbesondere in funktionalen Sprachen wird das Konzept allerdings noch verfeinert. In Haskell transformiert ein Konstruktor Daten nicht im eigentlichen Sinn, sondern bündelt lediglich eine Kennzeichnung, die festhält, welcher Konstruktor genutzt wurde, mit den n übergebenen Argumenten zu einem Tupel der Größe $n + 1$ [21a]. Das entspricht exakt der Funktionsanwendung in einem Term, mit dem Funktionssymbol als Kennzeichnung. Dementsprechend folgt die Definition des Konstruktors.

Definition 2.4. Die Interpretation u_c angewendet auf ein Funktionssymbol f bildet das n-Tupel von Argumenten auf eine Funktionsanwendung von f mit den selben n Argumenten ab. Mit $f \in F$ und $arity\ f = n \in \mathbb{N}$ gilt

$$u_c\ f\colon T^n \to T,\ (t_1, \ldots, t_n) \mapsto (f, t_1, \ldots, t_n)$$

Mit einem beliebigen $v\colon C \to C'$ ändert die Interpretation $eval(u_c, v)\colon T(F, C) \to T(F, C')$ damit nur die Konstantensymbole eines Terms, lässt aber die sonstige Struktur

unverändert. Insbesondere ist $eval(u_c, v)\colon T \to T$ mit $v\ y = y$ die Identität. Die Interpretation u_c reicht für bestimmte Funktionssymbole aus, etwa kann so das Funktionssymbol `pair` ein Paar als Term darstellen.

$$u_c\ \texttt{pair}\colon T^2 \to T,\ (a, b) \mapsto (\texttt{pair}, a, b)$$

Äquivalent ist die Darstellung endlicher Mengen und Tupel mit den variadischen Funktionssymbolen `set` und `tup` möglich[3]. Hoffmann und O'Donnell [HO82] definieren den Konstruktor im selben Kontext.

Beispiel 2.5. Ein Graph $G = (V, E)$ wird als Paar der Menge von Knoten V und Menge der Kanten E definiert. Eine Kante ist dabei eine zweielementige Menge von Knoten. Der vollständige Graph auf drei Knoten ist $K_3 = (\{1, 2, 3\}, \{\{1, 2\}, \{2, 3\}, \{1, 3\}\})$. Mit der Interpretation der Funktionssymbole `pair` und `set` als Konstruktoren lässt sich K_3 auch als Term darstellen:

$$(\texttt{pair}, (\texttt{set}, 1, 2, 3), (\texttt{set}, (\texttt{set}, 1, 2), (\texttt{set}, 2, 3), (\texttt{set}, 1, 3))).$$

2.3 Muster

Bisher wurden die Objekte beschrieben, die in dieser Arbeit transformiert werden sollen. Die Transformationsregeln selbst lassen sich allerdings auch als Paare von bestimmten Termen darstellen. Zur Abgrenzung beider Konzepte werden die zu transformierenden Terme $t \in T(F, C)$ von hier an Literal genannt. Terme, die Teil einer Regeldefinition sind, werden Muster genannt. Die Menge der Muster $M(F, C)$ ist eine Obermenge der Literale, da sie deren Konstantensymbole um die Menge der Mustervariablen X erweitert[4]. Konkrete Elemente $\mathbf{x} \in X$ werden im folgenden **fett** geschrieben.

$$M(F, C) \coloneqq T(F, C \cup X)$$

Eine Ersetzungsregel für Literale $t \in T(F, C)$ hat die Form $(l, r) \in M(F, C) \times M(F, C)$. Die linke Seite l steht für das Muster, welches im Literal durch einen Ausdruck der Form der rechten Seite r ersetzt werden soll. Für die bessere Lesbarkeit wird statt (l, r) auch $l \mapsto r$ geschrieben.

Beispiel 2.6. Die Regel, die die Summe zweier identischer Terme x als Produkt von 2 und x transformiert, wird geschrieben als

$$(\texttt{sum}, \mathbf{x}, \mathbf{x}) \mapsto (\texttt{prod}, 2, \mathbf{x}).$$

In der Anwendung auf das Literal $t = (\texttt{sum}, (\texttt{sin}, 3), (\texttt{sin}, 3))$, wird t zu $t' = (\texttt{prod}, 2, (\texttt{sin}, 3))$ transformiert. Hervorzuheben ist dabei, dass die Mustervariable $\mathbf{x}$ selbst nicht mehr im Ergebnisterm vorkommt. Sie wurde durch den Teilterm ersetzt, der im Ursprungsliteral an der Stelle von $\mathbf{x}$ stand, nämlich $(\texttt{sin}, 3)$.

[3] Da eine Menge ihren Elementen keine Reihenfolge gibt, muss u_c `set` im Unterschied zu u_c `tup` prinzipiell nicht die ursprüngliche Argumentreihenfolge erhalten. In Kapitel 3.2 wird eine Größenrelation zur möglichen Umordnung diskutiert.

[4] Die Ergänzung der Funktionssymbole um Mustervariablen ist genau so möglich, wird aber vor allem, um die Notation übersichtlich zu halten, in den späteren Kapiteln nicht verfolgt.

Definition 2.7. Für ein Muster $p \in M(F, C)$ und ein Literal $t \in T(F, C)$ wird die Funktion $v_p \colon X \to T(F, C)$ als Match bezeichnet, wenn sie die Mustervariablen in p so durch Literale ersetzt, dass ein Term identisch zu t entsteht. Formal muss

$$eval(u_c, \tilde{v}_p)\ p = t$$

gelten. Die Hilfsfunktion $\tilde{v}_p$ erweitert dafür den Definitionsbereich von v_p um die Konstantensymbole C.

$$\tilde{v}_p\ c = \begin{cases} v_p\ c & c \in X \\ c & c \in C \setminus X \end{cases}$$

Im Folgenden wird der Begriff des Matches noch etwas weiter gefasst. Es werden nach wie vor die Mustervariablen durch Literale ersetzt, allerdings muss nicht direkt das Ergebnis der Ersetzung, sondern nur eine normalisierte Form des Ergebnisses mit dem Literal t übereinstimmen:

$$lit(v_p, p) = t$$

$$\text{mit } lit(v_p, p) \coloneqq normalize\ (eval(u_c, \tilde{v}_p)\ p).$$

Die Funktion $lit \colon (X \to T) \times M \to T$ wird als Musterinterpretation bezeichnet. Aus der Definition ist klar, dass $normalize \colon T \to T$ Terme auf Terme abbildet und ein Match v_p nur dann gefunden werden kann, wenn t im Bild von *normalize* liegt. Gedacht ist *normalize* als Mittel, um Unterschiede zwischen Termen zu reduzieren, die die Auswertung für eine gegebene Interpretation $eval(u, v)$ nicht ändern. Da das Ergebnis von *normalize* aber nur von dem übergebenen Term abhängig ist, können nie direkt Unterschiede zwischen mehreren Termen verglichen und missachtet werden. Ist die normalisierte Form t' eines Terms t ein Fixpunkt von *normalize*, ist in dem Kontext klar, dass t und t' die gleiche Auswertung mit der Interpretation $eval(u, v)$ besitzen, da auch $(normalize\ t') = normalize\ t$ gilt. Wäre $normalize\ t' \neq t'$, würde die Funktion nicht die ihr zugedachte Aufgabe erfüllen. Deshalb muss *normalize* eine Projektion sein, also eine Funktion, für die jeder Bildpunkt gleichzeitig ein Fixpunkt ist.
Welche Unterschiede *normalize* beseitigt, wird im folgenden Kapitel 3 festgelegt. Klar ist aber, dass, je nach Wahl von *normalize*, zwar ein einzelnes Muster sehr mächtig werden kann, also ein Match mit sehr vielen Termen möglich ist, das Finden des Matches im allgemeinen Fall allerdings immer aufwändiger wird.

Ein Matchalgorithmus ist eine Vorgehensweise für ein gegebenes Paar $(p, t) \in M(F, C) \times T(F, C)$ ein gültiges Match zu finden. Perfekt wird ein solcher Algorithmus dann genannt, wenn jedes mögliche Match gefunden wird. Im vorangegangenen Beispiel 2.6 ist v_p mit $v_p\ \mathtt{x} = (\mathtt{sin}, 3)$ ein Match.

3 Normalform

Das Kernthema dieser Arbeit ist die Vereinfachung von Termen. Eine Vereinfachung ist nur gültig, wenn sich die Bedeutung des vereinfachten Terms gegenüber der des ursprünglichen Terms nicht geändert hat. Da ein Term in sich keine Bedeutung trägt, muss eine Vereinfachung immer in Bezug auf eine Interpretation $eval(u, v)$ gesehen werden. Etwa kann der Ausdruck XAX^{-1} zu A vereinfacht werden, wenn $X, A \in \mathbb{C} \setminus \{0\}$, allerdings ist die Vereinfachung allgemein nicht möglich, sollten die Symbole X und A für Matrizen stehen.
Im Folgenden wird von der Assoziativität oder Kommutativität bestimmter Funktionssymbole gesprochen. Diese ist immer im Kontext der Interpretation $eval(u, v)$ zu sehen. Gleichzeitig ist aber auch klar, dass unabhängig von der Interpretation verschiedene Funktionssymbole die Rolle der Multiplikation übernehmen müssen, sollte sowohl skalare Multiplikation als auch Matrixmultiplikation im selben Term möglich sein. XAX^{-1} als Matrixmultiplikation könnte der Term $(\texttt{prod'}, X, A, (\texttt{pow}, X, -1))$ darstellen. Sind A und X Skalare, wäre der Ausdruck als $(\texttt{prod}, X, A, (\texttt{pow}, X, -1))$ schreibbar. Das Funktionssymbol `prod'` steht dann für ein nicht-kommutatives Produkt, während die Reihenfolge der Parameter in einer Funktionsanwendung von `prod` keine Rolle spielt.

In diesem Abschnitt werden einfache Termumformungen beschrieben, die isolierte Eingenschaften einzelner Funktionen ausnutzen. Ziel ist es, Äquivalenzklassen für die Erkennung von Mustern zu schaffen, die über die Austauschbarkeit jeder Mustervariable mit einem beliebigen Literal hinausgehen. Als Beispiel dient die Regel der Faktorisierung, normal geschrieben als $x \cdot y + x \cdot z = x \cdot (y + z)$. In der in Unterkapitel 2.3 etablierten Musterschreibweise, mit Mustervariablen **fett** geschrieben, wird daraus:

$$(\texttt{sum}, (\texttt{prod}, \mathbf{x}, \mathbf{y}), (\texttt{prod}, \mathbf{x}, \mathbf{z})) \mapsto (\texttt{prod}, \mathbf{x}, (\texttt{sum}, \mathbf{y}, \mathbf{z}))$$

Ziel ist, die Regel auf das Literal $(\texttt{sum}, (\texttt{prod}, a, b), (\texttt{prod}, a, c, d))$ anwendbar zu machen bzw. eine Regel formulieren zu können, die eine ähnliche Struktur hat und diesen Fall mit abdeckt. Würde das Literal geschrieben sein als

$$(\texttt{sum}, (\texttt{prod}, a, b), (\texttt{prod}, a, (\texttt{prod}, c, d))),$$

gäbe es ein Match v_p der linken Regelseite mit dem Literal, mit $v_p\ \mathbf{x} = a$, $v_p\ \mathbf{y} = b$ und $v_p\ \mathbf{z} = (\texttt{prod}, c, d)$. Ergebnis dieses Kapitels wird eine in Abschnitt 2.3 genutzte Projektion $normalize\colon T \to T$ sein, welche die Beispielregel auf das Beispielliteral in seiner ursprünglichen Form anwendbar macht.

Weiteres Ziel dieses Kapitels ist, dass möglichst viele Literale mit identischer Auswertung identische normalisierter Terme besitzen. So soll etwa die Normalisierung von $t_1 = (\texttt{sum}, a, b, c)$ identisch zur Normalisierung von $t_2 = (\texttt{sum}, b, a, c)$ identisch zur Normalisierung von $t_3 = (\texttt{sum}, (\texttt{sum}, b, a), c)$ sein. Je mehr Literale identischen Wertes auch zu identischen Termen normalisiert werden, desto besser können Muster

erkannt werden, in denen dieselbe Mustervariable mehrfach vorkommt. Interessant ist dabei, dass derselbe Effekt auch erreicht werden würde, wenn man eine Menge von Ersetzungsregeln um entsprechende normalisierende Ersetzungsregeln ergänzt. Wo die Grenze in der Arbeitsteilung von einer fest implementierten *normalize* Funktion zu den Regeln in einem Termersetzungssystem liegt, ist prinzipiell fast beliebig und in erster Linie eine Frage des Aufwandes, sowohl in Programmierung als auch Laufzeit. Etwa würde eine Darstellung natürlicher Zahlen ähnlich der Church-Numerale, wie sie in der Fachliteratur, etwa bei Baader und Nipkow [BN98b], üblich ist, erlauben, Rechenoperationen auf den natürlichen Zahlen komplett mit einer endlichen Menge von Mustern auszuwerten. Nachteile dieser Vorgehensweise wären allerdings eine langsamere Auswertung, mehr Speicherbedarf und nur sehr schwierig zu lesende Ergebnisse. Andersherum wäre etwa die Anwendung der ersten binomischen Formel prinzipiell auch in der *normalize* Funktion möglich, allerdings steht der Aufwand, manuell auf das Muster zu testen, nur möglicherweise minimalen Geschwindigkeitsvorteilen des Gesamtsystems gegenüber. Die Transformationen, die in diesem Kapitel der *normalize* Funktion zugewiesen werden, sollen also idealerweise nicht einfacher mit Mustern implementierbar sein.

In diesem Kapitel werden häufig Abschnitte der Parameter einer Funktionsanwendung beliebiger Länge der Form $t_i, \ldots, t_k$ vorkommen. Kompakt wird $ts...$ für den (möglicherweise leeren) Abschnitt des Funktionsanwendungstupels geschrieben. Das s in $ts...$ ist dann nicht als einzelnes Symbol zu lesen, sondern als Suffix um t in den Plural zu setzen. $(f, t_1, \ldots, t_k, a, t_{k+2}, \ldots, t_n)$ kann also äquivalent $(f, ts..., a, rs...)$ geschrieben werden, mit $(t_1, \ldots, t_k) = (ts...)$ und $(t_{k+2}, \ldots, t_n) = (rs...)$.

3.1 Assoziative Funktionsanwendungen

Die geschachtelte Anwendung einer assoziativen Funktion führt je nach Klammersetzung zu verschiedenen mathematisch äquivalenten Termen. Als Beispiel dient hier die Addition, dargestellt als Anwendung des Funktionssymbols `sum`. Die folgenden Ausdrücke sind paarweise verschiedene Terme, jedoch in ihrer Interpretation als Summe von a, b, c und d alle mathematisch äquivalent.

$$\begin{aligned}(\texttt{sum}, (\texttt{sum}, (\texttt{sum}, a, b), c), d) &= (\texttt{sum}, (\texttt{sum}, a, (\texttt{sum}, b, c)), d)\\ &= (\texttt{sum}, (\texttt{sum}, a, b), (\texttt{sum}, c, d))\\ &= (\texttt{sum}, a, (\texttt{sum}, b, (\texttt{sum}, c, d)))\\ &= \ldots\end{aligned}$$

Es gibt mehrere Optionen eine solche Schachtelung in einem Term zu normalisieren, also in eine eindeutige Form zu bringen. Die erste ist, festzulegen, dass in der normalisierten Form höchstens eines der beiden Argumente einer binären assoziativen Funktion wieder Anwendung desselben Funktionssymbols sein darf. Wählt man das zweite Argument dafür aus, wird die Summe in der Normalform dargestellt als $(\texttt{sum}, a, (\texttt{sum}, b, (\texttt{sum}, c, d)))$.

Ein Problem der Methode ist, dass nicht immer alle Argumente eines assoziativen Funktionssymbols direkt vorliegen.
Alternativ kann man die Summe von zwei Argumenten auch als Spezialfall einer Summe von $n \in \mathbb{N}$ Argumenten auffassen, gewohnt geschrieben als $\Sigma_{x \in \{a,b,c,d\}} x$. Dieser Weg wird im Folgenden gewählt, wobei die Darstellung als Term dann $(\texttt{sum}, a, b, c, d)$ ist. Assoziative Funktionen sind in der gewählten Darstellung damit variadisch. Eker ([Eke95]), Benanav ([BKN87]) und Kounalis ([KL91]) wählen ebenfalls diese Darstellung. Die Normalisierung von Funktionsanwendungen des assoziativen Funktionssymbols f bedeutet dann, geschachtelte Funktionsanwendungen in eine einzelne Funktionsanwendung zu übersetzen.

$$(f, as..., f(bs...), cs...) \mapsto (f, as..., bs..., cs...)$$

Der Spezialfall ist eine assoziative Funktionsanwendung mit nur einem Parameter. Diese kann immer zu dem Parameter selbst normalisiert werden.
Als Algorithmus dargestellt sind die Überlegungen in Algorithmus 1. Die Funktion $\tilde{u}$ kann hier und in den weiteren Algorithmen dieses Kapitels als „natürliche“ Interpretation der Menge von Funktionssymbolen gesehen werden, ähnlich u in Beispiel 2.3. Prinzipiell ist für die Gültigkeit des Kapitel aber egal, in welchem Kontext die Abbildungsvorschrift von $\tilde{u}$ Sinn ergibt oder ob ein solcher Kontext überhaupt existiert. Wichtig ist nur, dass $\tilde{u}$ über das gesamte Kapitel hinweg eine einheitliche Abbildungsvorschrift hat.

Algorithmus 1: *flatten*: $T \to T$

```
Input: t ∈ T(F, C)
if t = (f, t_1) mit ũ f assoziativ
    return t_1
else if t = (f, t_0, ..., t_{n-1}) mit ũ f assoziativ
    while t = (f, xs..., (f, ys...), zs...)
        t ← (f, xs..., ys..., zs...)
return t
```

3.2 Kommutative Funktionsanwendungen

Eine Normalform für kommutative Funktionsanwendungen erfordert eine totale Ordnung auf der Menge aller Terme $T(F, C)$.

Definition 3.1. Aufbauend auf einer totalen Ordnung von F sowie C, kann eine lexikographische Ordnung $<$ von T frei nach [Lau03] wie folgt definiert werden:

1. sind $c, \tilde{c} \in T$ Konstantensymbole, so ist die Ordnung identisch zu der Ordnung in C

2. sind $c, a, \in T$ sowie c ein Konstantensymbol und a eine Funktionsanwendung, gilt $c < a$

3. sind $a = (f, ts...), b = (g, rs...) \in T$ Funktionsanwendungen und ist $f < g$, gilt $a < b$
4. sind $a = (f, t_1, \ldots, t_n), b = (f, r_1, \ldots, r_m) \in T$ Funktionsanwendungen, ist die Ordnung wie folgt
   a) wenn $\exists k \leq \min(n, m) \colon \forall i < k \; t_i = r_i, \; t_k \neq r_k$, gilt $a < b \iff t_k < r_k$
   b) ist $n < m$ und $\forall i < n \colon t_i = r_i$, gilt $a < b$
   c) ist $n = m$ und $\forall i \leq n \colon t_i = r_i$, gilt $a = b$

Lemma 3.2. T hat genau dann ein Minimum, wenn C ein Minimum hat (1). T hat kein endlich großes Maximum (2).

Beweis. Teil (1) folgt daraus, dass die Konstantensymbole selbst bereits Terme kleiner als jede Funktionsanwendung sind. Ein Minimum von C ist damit gleichzeitig Minimum von T.
Teil (2) folgt aus einem Widerspruch. Angenommen es gäbe einen größten endlichen Term t. Der Term t' ist identisch zu t, nur die Konstantensymbole von t werden durch beliebige Funktionsanwendungen ersetzt. Der neue Term t' ist größer als t. □

Zur Normalisierung einer kommutativen Funktionsanwendung werden zuerst alle Parameter normalisiert, dann können die Parameter nach der lexikographischen Ordnung $<$ von T sortiert werden.

3.3 Teilweise Auswertung

Algorithmus 2: *combine*: $T \to T$

Input: $t \in T(F, C)$
if $eval(\tilde{u}, id)\ t = c \in C$
  return c
else if $t = (f, t_0, \ldots, t_{n-1})$ und $\tilde{u}\ f$ assoziativ
  if $\tilde{u}\ f$ kommutativ
    while $t = (f, us..., x, vs..., y, ws...)$ und $\tilde{u}\ f\ (x, y) = z \in C$
      $t \leftarrow (f, z, us..., vs..., ws...)$
  else
    while $t = (f, us..., x, y, vs...)$ und $\tilde{u}\ f\ (x, y) = z \in C$
      $t \leftarrow (f, us..., z, vs...)$

Mit der Darstellung einer assoziativen Funktion mit einem variadischen Funktionssymbol $f \in F$, kann eine Funktionsanwendung von f in bestimmen Fällen teilweise ausgewertet werden. Als Beispiel kann die Summe der Symbole 1, 3 und `a` geschrieben als $(\texttt{sum}, 1, 3, \texttt{a})$ zur Summe $(\texttt{sum}, 4, \texttt{a})$ transformiert werden. Gilt allgemein für ein Funktionssymbol

$f \in F$, dass $\tilde{u}$ f assoziativ ist, reicht es aus, zwei aufeinander folgende Argumente x und y in einer Funktionsanwendung von f zu finden, mit denen die Funktionsanwendung (f, x, y) auswertbar wäre. x und y können dann entsprechend ersetzt werden.

$$eval(\tilde{u}, id)\ (f, x, y) = z \in C \implies (f, us..., x, y, vs...) \mapsto (f, us..., z, vs...)$$

Ist $\tilde{u}$ f zudem kommutativ, müssen x und y nicht notwendigerweise direkt aufeinander folgen.

$$eval(\tilde{u}, id)\ (f, x, y) = z \in C \implies (f, us..., x, vs..., y, ws...) \mapsto (f, z, us..., vs..., ws...)$$

Eine normalisierte Funktionsanwendung enthält keine zwei auf diese Art ersetzbare Argumente x und y mehr. Weiter ist jede Funktionsanwendung, die als ganzes zu einer Konstante $z \in C$ auswertbar ist, ausgewertet.
Zusammengefasst sind die Überlegungen in Algorithmus 2. Die Verfahren zur Normalisierung assoziativer und kommutativer Funktionssymbole werden auch von Eker [Eke95] beschrieben.

3.4 Kombination der einzelnen Vereinfachungen

Algorithmus 3: *normalize*: $T \to T$

Input: $t \in T(F, C)$
if $t = (f, t_1, \ldots, t_n)$
  for $i \in \{1, \ldots, n\}$
    $t_i \leftarrow$ *normalize* t_i
$t \leftarrow$ *flatten* t
$t \leftarrow$ *combine* t
if $t = (f, t_1, \ldots, t_n)$ mit $\tilde{u}$ f kommutativ
  sortiere $t_1, \ldots, t_n$ lexikographisch nach Ordnung $<$
return t

Algorithmus 3 kombiniert die einzelnen Überlegungen dieses Kapitels: Zunächst werden alle Argumente einer Funktionsanwendung normalisiert, anschließend die Funktionsanwendung selbst.

4 Mustererkennung

In Kapitel 2.3 wurden die Konzepte des Musters und des Matches eingeführt. Letzteres ist insbesondere in der weiter gefassten Form relevant. Diese erlaubt, Muster mit strukturell nicht exakt identischen Literalen zu assoziieren, sofern die Unterschiede mit der Projektion $normalize\colon T \to T$ beseitigt werden können. Kapitel 3 definiert diese Projektion.
In diesem Kapitel wird ein Algorithmus entwickelt, der die Äquivalenzklassen der verschieden geschachtelten Funktionsanwendungen eines assoziativen Funktionssymbols mit denselben Argumenten, sowie die Äquivalenzklassen der verschieden permutierten Argumente in der Funktionsanwendung eines kommutativen Funktionssymbols beim Finden eines Matches berücksichtigt. Zunächst nicht verfolgt wird die teilweise Auswertung von *normalize* aus Abschnitt 3.3.

4.1 Grundstruktur

Dem Ergebnis eines Matchalgorithmus müssen zwei Dinge entnehmbar sein. Zum einen muss klar sein, ob ein Match $v_p\colon X \to T$ gefunden wurde. Wurde ein Match gefunden, muss zudem dessen Abbildungsvorschrift zurückgegeben werden. Der Rückgabetyp von Algorithmus 4 ist deswegen nicht nur das finale Match, sondern auch ein Wahrheitswert $b \in Bool := \{false, true\}$. Alternativ kann die Menge aller möglichen Matches zurückgegeben werden. Diese Idee wird im Folgenden nicht weiter verfolgt, da sie mit den Anforderungen an hier behandelte Muster auch im besten Fall schnell exponentielle Laufzeiten produziert[1]. Sind aber Mehrfachnennungen einer Mustervariable in einem Muster nicht erlaubt, haben Hoffman und O'Donnell in [CM 82] gezeigt, dass sehr effiziente Algorithmen zum gleichzeitigen Finden von Matches einer ganzen Menge von Mustern in allen Teiltermen eines Literals mit dieser Grundidee möglich sind.

Da eine Mustervariable in dieser Arbeit mehrfach in einem Muster vorkommen darf, muss ein Algorithmus beim Suchen nach einem Match $v_p\colon X \to T$ zu jedem Zeitpunkt wissen, für welche $x \in X$ das Match $v_p\ x$ bereits feststeht. v_p ist also nicht nur Rückgabewert eines Matchalgorithmus, sondern muss mit den Funktionswerten für bereits besuchte Mustervariablen auch Eingabe in den Algorithmus sein. In Algorithmus 4 wird v_p deswegen als partielle Funktion definiert, welche zu Beginn keine einzige Mustervariable nach T abbilden kann.

Wenn das Match streng definiert ist, also der Unterschied zwischen einem Muster p und einem Literal t für die Existenz eines Matches v_p ausschließlich darin bestehen darf, dass Teilterme von t in p durch eine Mustervariable repräsentiert werden, ist ein einfacher Matchalgorithmus fast trivial. Auf der Idee von *simpleMatchAlgorithm* basieren auch die späteren Algorithmen dieses Kapitels. Die Vorgehensweise ist, dass mit

[1] siehe Lemma 4.3

Algorithmus 4: *simpleMatchAlgorithmShell*: $M \times T \to (Bool, X \rightharpoonup T)$

Input: $p \in M$, $t \in T$
let $v_p \colon X \rightharpoonup T,\ x \mapsto \bot$
return $simpleMatchAlgorithm(p, t, v_p)$

Algorithmus 5: *simpleMatchAlgorithm*: $M \times T \times (X \rightharpoonup T) \to (Bool, X \rightharpoonup T)$

Input: $p \in M$, $t \in T$, $v_p \colon X \rightharpoonup T$
if $p \in X$ **and** $v_p\ p = \bot$
  $(v_p\ p) \leftarrow t$
  return $(true, v_p)$
else if $p \in X$ **and** $v_p\ p \neq \bot$
  return $(v_p\ p = t, v_p)$
else if $p \in C \setminus X$
  return $(p = t, v_p)$
else if $p = (f, p_0, \ldots, p_{m-1})$ **and** $t = (f, t_0, \ldots, t_{n-1})$
  for $k \in \{0, \ldots, n-1\}$
    $(success_k, v_p) \leftarrow simpleMatchAlgorithm(m_k, t_k, v_p)$
    if *not* $success_k$
      return $(false, v_p)$
  return $(true, v_p)$
else
  return $(false, v_p)$

einer Tiefensuche, die parallel durch Muster und Literal läuft, nach einem Unterschied zwischen beiden gesucht wird. Mustervariablen funktionieren dabei als Wildcard, wenn eine identische Mustervariable in der Tiefensuche vorher noch nicht gefunden wurde. Andernfalls vergleichen sie identisch zu dem Teilbaum, der mit dem ersten Vorkommen der Mustervariable verglichen wurde. Die Aufgabe, diese vorher begegneten Teilbäume zu speichern, übernimmt v_p, was erklärt, warum v_p auch als Parameter für Algorithmus 5 notwendig ist. Ist das gesamte Muster durchlaufen worden, ohne einen strukturellen Unterschied zum Literal zu finden, ist v_p das resultierende Match.

Lemma 4.1. Die Laufzeit von Algorithmus 4 ist linear abhängig von der Anzahl der Funktionssymbole und Konstantensymbole des Literals.

Beweis. Gibt es ein Match, wird jedes Funktionssymbol und Konstantensymbol des Literals höchstens einmal in *simpleMatchAlgorithm* abgelaufen. Wird eine Funktionsanwendung t im Literal parallel zu einer Mustervariable $\mathbf{x}$ im Muster abgelaufen, bleiben die Nachkommen von t unbesucht, wenn $\mathbf{x}$ noch nicht gematcht wurde. Andernfalls wird jeder Nachkomme von t höchstens einmal abgelaufen, um Gleichheit zu v_p $\mathbf{x}$ zu testen. Gibt es kein Match, wird das Literal so lange identisch zum anderen Fall abgelaufen, bis ein struktureller Unterschied festgestellt wurde. Dann bricht der Algorithmus ab. □

Definition 4.2. Die Instanz einer Mustervariable $\mathbf{x}$ wird als bindend bezeichnet, wenn sie in einer Tiefensuche durch das gesamte Muster als erste Instanz abgelaufen wird. Da Algorithmus 5 das Muster in einer Tiefensuche abläuft, ist die Bedingung $p \in X$ **and** $v_p\ p = \bot$ in der ersten Zeile damit genau dann wahr, wenn p bindend ist[2]. Weitere Instanzen von $\mathbf{x}$ im selben Muster werden als gebunden bezeichnet.

4.2 Multi-Mustervariablen

Von Anfang an werden Funktionssymbole in dieser Arbeit als möglicherweise variadisch definiert. Das ist insofern ein Problem, als dass Muster bisher immer nur eine feste Anzahl an Argumenten für jede Funktionsanwendung angeben können. Ist ein variadisches Funktionssymbol zudem assoziativ, ließe sich dieses Problem prinzipiell beheben, wenn Assoziativität im Matchalgorithmus berücksichtigt würde. Das Muster $\tilde{p} = (f, \mathbf{x}, \mathbf{y})$ würde für ein assoziatives Funktionssymbol f dann auch Literale wie $\tilde{t} = (f, a, b, c, d)$ matchen, mit verschiedenen Optionen für v_p, etwa $v_p\ \mathbf{x} = (f, a, b)$ und $v_p\ \mathbf{y} = (f, c, d)$. Ist auch die leere Funktionsanwendung (f) von f erlaubt[3], gibt es fünf verschiedene Matches v_p für $(\tilde{p}, \tilde{t})$ mit nicht-kommutativem f.

[2] Der Begriff bindend ist so zu verstehen, dass $\mathbf{x}$ nach Ablauf der ersten Instanz in Algorithmus 5 einen festen Wert v_p $\mathbf{x}$ hat, also für den spätere Teil des Musters an diesen Wert gebunden ist.

[3] Das ergibt dann Sinn, wenn f ein neutrales Element $e \in T$ besitzt, da $(f, as..., (f), bs...)$ mit *normalize* zu $(f, as..., bs...) = (f, as..., e, bs...)$ umgeformt wird. Im Folgenden wird von der Existenz eines Neutralen Elementes ausgegangen.

Lemma 4.3. Das Muster $p = (f, \mathbf{x}_1, \ldots, \mathbf{x}_m)$ hat mit dem Literal $t = (f, a_1, \ldots, a_n)$ genau $\binom{m+n-1}{n}$ mögliche Matches, wenn f assoziativ aber nicht kommutativ ist (1). Ist f assoziativ und kommutativ, existieren m^n mögliche Matches (2).

Beweis. (1): Es gibt m (möglicherweise leere) Abschnitte in den n Argumenten von t, welche jeweils eine Mustervariable $\mathbf{x}_i$ matchen. Stellt man eine Abschnittsgrenze mit einem Strich $\mid$ und ein Argument von t mit einem Stern $*$ dar, kann die Aufteilung der Argumente von t über ein String aus $m-1$ Strichen und n Sternen dargestellt werden. Als Beispiel ist $**\mid**$ der String zur Aufteilung von $\tilde{t}$ aus dem Anfang des Abschnittes zum beschriebenen Match v_p. Es gibt $\binom{m+n-1}{n}$ Möglichkeiten die n Sterne auf die $m+n-1$ möglichen Plätze zu verteilen.

(2): Jedes der n Argumente von t kann unabhängig der restlichen Argumente zu einer der m Mustervariablen gematcht werden. Insgesamt ergeben sich so m^n Kombinationen. □

Schon für nicht-kommutative aber assoziative Funktionssymbole f gibt es somit Muster p mit einer Anzahl möglicher Matches, die exponentiell mit der Größe des Literals steigt. Ist ein solches Muster p Teil eines größeren Musters p' und kommen Mustervariablen von p auch in anderen Teilen von p' vor, so ist nicht direkt ersichtlich, wie ein Algorithmus aussehen würde, der in P liegt und bestimmen kann, dass es kein Match für p' mit einem entsprechenden Literal gibt, bzw. das Match findet. Die Existenz eines solchen Algorithmus ist unwahrscheinlich: Benanav hat 1987 gezeigt, dass das Problem NP-vollständig ist [BKN87]. Von dem perfekten Matchalgorithmus wird aus diesem Grund abgesehen. Für viele Spezialfälle sind bessere Algorithmen möglich. Eine wichtige Klasse solcher Spezialfälle ist die, bei denen für den Autor klar ist, welche Mustervariable möglicherweise mehrere Parameter des Literals matchen soll. Würde bei der Ersetzung der ersten Binomischen Formel eine weitere Mustervariable **c** ergänzt, um die Binomische Formel auch in einer Summe mit mehr als drei Summanden zu erkennen, kann die Ersetzungsregel geschrieben werden als

$$(\texttt{sum}, (\texttt{pow}, \mathbf{a}, 2), (\texttt{prod}, 2, \mathbf{a}, \mathbf{b}), (\texttt{pow}, \mathbf{b}, 2), \mathbf{c}) \mapsto (\texttt{sum}, (\texttt{pow}, (\texttt{sum}, \mathbf{a}, \mathbf{b}), 2), \mathbf{c}).$$

Die Mustervariable **c** ist damit vom Autor des Musters ausschließlich dazu gedacht überbleibende Summanden „aufzusaugen“. Dieser Gedanke bleibt dem Algorithmus bisher verborgen. Die in dieser Arbeit gewählte Lösung zur Beschreibung von beliebig vielen Argumenten in einem Muster ist im Prinzip schon in Kapitel 3 eingeführt worden. Die Schreibweise $(f, ts...)$ als kompakte Alternative zu $(f, t_1, \ldots, t_n)$ hat viele der zur Beschreibung von Assoziativität gewünschten Eigenschaften. Ferner können so auch Muster mit nicht assoziativen variadischen Funktionssymbolen dargestellt werden. Eine Multi-Mustervariable der Form **xs**... kann also nicht nur genau ein Argument in einer Funktionsanwendung matchen, sondern beliebig viele, auch keins. Um den Matchalgorithmus nicht zu kompliziert zu gestalten, darf jede Multi-Mustervariable auf

der linken Seite einer Ersetzungsregel nur höchstens einmal vorkommen[4]. Die rigorose Beschreibung des Konzeptes gestaltet sich allerdings mit den bisher eingeführten Ideen schwierig, da eine Multi-Mustervariable nur Teil einer Funktionsanwendung ist und damit auch alleine keinen vollständigen Term repräsentiert. Konnte eine Matchfunktion $v_p \colon X \to T$ vorher einfach auf die Menge aller Terme abbilden, wäre dies nach Hinzufügen der Multi-Mustervariablen nicht mehr möglich. Entsprechend umständlicher würde auch die Beschreibung der Auswertung eines Musters werden.

Formal wird die Multi-Mustervariable damit nicht als neues Symbol in die Menge der Muster aufgenommen, sondern ist lediglich eine vereinfachende Schreibweise, die wie in Kapitel 3 immer für eine beliebige Anzahl an Teiltermen steht, in diesem Fall Mustervariablen. Ein Muster mit einer Multi-Mustervariable $\mathbf{xs}...$ repräsentiert formal unendlich viele konkrete Muster mit konkreten Mustervariablen $\mathbf{x_i}$:

$$\begin{aligned}(f, \mathbf{ts}...) = \{&(f),\\ &(f, \mathbf{x_1}),\\ &(f, \mathbf{x_1}, \mathbf{x_2}),\\ &(f, \mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}),\\ &\dots\}\end{aligned}$$

Für die folgenden Algorithmen dieses Kapitels sowie der echten Umsetzung ist es nicht praktikabel, diese Definition anzuwenden. Mit der Restriktion, dass jede Multi-Mustervariable auf der linken Seite einer Ersetzungsregel höchstens einmal vorkommen darf, ist eine einfache Verwaltung möglich. Für Funktionsanwendungen kommutativer Funktionssymbole in einem Muster muss lediglich zwischen *enthält eine Multi-Mustervariable* und *enthält keine Multi-Mustervariable* unterschieden werden. Multi-Mustervariablen in Funktionsanwendungen nicht-kommutativer Funktionssymbole haben eine eindeutige Position. Im Folgenden werden aber auch hier keine weiteren Parameter für Multi-Mustervariablen hinzugefügt. Alternativ wird für jeden tatsächlichen Term in den Argumenten einer solchen Funktionsanwendung festgehalten, ob er Nachfolger einer Multi-Mustervariable ist und weiter, ob an dem letzten Argument noch eine Multi-Mustervariable anschließt. Die Ersetzungsregel für die erste Binomische Formel, anwendbar auf Summen beliebiger Länge, wird demgemäß geschrieben als:

$$(\mathtt{sum}, (\mathtt{pow}, \mathbf{a}, 2), (\mathtt{prod}, 2, \mathbf{a}, \mathbf{b}), (\mathtt{pow}, \mathbf{b}, 2), \mathbf{cs}...) \mapsto (\mathtt{sum}, (\mathtt{pow}, (\mathtt{sum}, \mathbf{a}, \mathbf{b}), 2), \mathbf{cs}...).$$

Die Summe der linken Seite setzt sich für die folgenden Algorithmen dieses Kapitels dennoch nur aus drei Summanden zusammen. Die syntaktisch als Parameter geschriebenen $\mathbf{cs}...$ kommen hier in der linken Seite der Regel nur als Wahrheitswert vor, denn es gilt „Die Summe *enthält eine Multi-Mustervariable*“. Lediglich auf der rechten Seite ist relevant, um welche Multi-Mustervariable es sich handelt, sollte es mehrere geben. Dieses Kapitel befasst sich allerdings fast ausschließlich mit der linken Seite einer Regel.

[4] Das macht mehrere Multi-Mustervariablen in der selben Funktionsanwendung eines kommutativen Funktionssymbols auf der linken Seite einer Ersetzungsregel überflüssig. Diese Konstellation ist dementsprechend im Folgenden nicht berücksichtigt.

4.3 Kommutative Muster

Die Algorithmen 6, 7, 8, 9 und 10 bilden zusammen die Grundlage des finalen Matchalgorithmus dieser Arbeit. Der Startpunkt einer Matchsuche ist der Aufruf von *findMatch*. Hier wird im Kontrast zu *simpleMatchAlgorithm* nicht direkt ein Rekursionsaufruf durchgeführt, sondern abhängig von der Form der vorgefundenen Funktionsanwendung eine entsprechende Strategie für die Suche eines Matches gewählt. Die Algorithmen, die die entsprechenden Strategien implementieren, sind *findPermutation*, *findDilation* und *findIdentic*. Alle hier vorgestellten Strategien nutzen dabei Backtracking, um die verschiedenen Möglichkeiten zu testen. Sollte dabei die Notwendigkeit auftreten, für einen bereits gematchten Teil des Musters ein neues Match mit dem selben Literal zu finden, wird in allen drei Strategien *rematch* aufgerufen. Dieser Algorithmus ist ähnlich zu *findMatch*, erwartet aber, dass das übergebende Muster p bereits mit dem übergebenen Literal t gematcht ist. Die eigentliche Arbeit wird bei *rematch* an die selben Suchstrategien abgegeben. Da der Startpunkt dort aber davon abhängig ist, ob bereits eine bestimmte Zuordnung der Argumente als erfolgreich matchend festgehalten ist oder nicht, wird diese Information als letzter Parameter jeder Strategie mit übergeben.
Im Grundaufbau funktionieren alle Strategien gleich. Die Argumente $p_0, \ldots, p_{m-1}$ des Musters p werden in der vorliegenden Reihenfolge mit den Argumenten $t_0, \ldots, t_{n-1}$ des Literals t gematcht. Kann für das aktuelle Argument p_i kein Match mehr gefunden werden, wird versucht, die vorhergehenden Argumente $p_0, \ldots, p_{i-1}$ neu zu matchen, beginnend mit p_{i-1}. Mit welchen Argumenten t_k ein Match dabei erlaubt ist, variiert nach Strategie. Am stärksten eingeschränkt ist *findIdentic*. Das Musterargument p_i kann dort nur mit t_k gematcht werden, wenn $k = i$ gilt[5]. Am wenigsten eingeschränkt ist *findPermutation*. Hier kann jedes p_i mit jedem t_k gematcht werden, vorausgesetzt t_k ist noch nicht mit einem Argument aus $p_0, \ldots, p_{i-1}$ gematcht. In der Freiheit dazwischen steht *findDilation*, welche die Reihenfolge der p_i untereinander gleich halten muss, jedoch eine Lücke beliebiger Länge zwischen p_{i-1} und p_i erlaubt, sofern im Muster an dieser Stelle eine Multi-Mustervariable steht[6].
Abweichend von bisherigen Algorithmen wird von hier an im Pseudocode nicht mehr jede tatsächlich notwendige Information explizit übergeben. Beispielsweise muss v_p nach wie vor von jedem Funktionsaufruf aktualisiert werden, ist aber im Pseudocode der Algorithmen 6, 9, 10, etc. nicht länger explizit in Parameterliste oder als Rückgabewert erwähnt. Anstelle der konkreten Zuweisung eines Wertes zu einem Namen, dargestellt durch den Pfeil nach links „←“, wird die Veränderung einer solchen nicht explizit erwähnten Datenstruktur nur mit dem Wort „merke“ dargestellt.

4.3.1 findMatch und rematch

Algorithmus 6 ist in der Struktur ähnlich zu *simpleMatchAlgorithm*. Neben dem Auslagern der Rekursionsaufrufe in die verschiedenen Matchstrategien besteht ein Unterschied im Umgang mit Mustervariablen. Für *simpleMatchAlgorithm* wird getestet, ob

[5] Algorithmus 5 hat ausschließlich auf diese Weise nach einem Match gesucht.
[6] Wie in Abschnitt 4.2 erörtert, treten diese hier nicht als echte Argumente auf.

Algorithmus 6: *findMatch*: $M \times T \to Bool$

Input: $p \in M$, $t \in T$
if $p \in X$ **and** p bindend
  merke: v_p $p = t$
  return *true*
else if $p \in X$ **and** p gebunden
  return v_p $p = t$
else if $p \in C \setminus X$
  return $p = t$
else if $p = (f, p_0, \ldots, p_{m-1})$ **and** $t = (f, t_0, \ldots, t_{n-1})$
  if u f kommutativ
    return *findPermutation*$(p, t, false)$
  else if $t_1, \ldots, t_n$ enthalten Multi-Mustervariablen
    return *findDilation*$(p, t, false)$
  else if $m = n$
    return *findIdentic*$(p, t, false)$
return *false*

Algorithmus 7: *rematch*: $M \times T \to Bool$

Input: $p \in M$, $t \in T$
if $p = (f, p_0, \ldots, p_{m-1})$ **and** $t = (f, t_0, \ldots, t_{n-1})$
  if u f kommutativ
    return *findPermutation*$(p, t, true)$
  else if $t_1, \ldots, t_n$ enthalten Multi-Mustervariablen
    return *findDilation*$(p, t, true)$
  else if $m = n$
    return *findIdentic*$(p, t, true)$
return *false*

der Funktionswert v_p $\mathbf{x}$ für eine Mustervariable $\mathbf{x}$ bereits definiert ist, wenn $\mathbf{x}$ angetroffen wird und herausgefunden werden muss, ob die Instanz bindend oder gebunden ist. Das reicht für *findMatch* nicht, da dieser Algorithmus auch funktionieren muss, wenn die verschiedenen Matchstrategien Backtracking beinhalten, das heißt, *rematch* aufrufen. Der Algorithmus *rematch* ist fast identisch zur unteren Hälfte von *findMatch*, ruft die verschiedenen Matchstrategien allerdings mit *true* als letztem Parameter auf, was bedeutet, dass direkt zum Backtracking gesprungen wird.

4.3.2 findIdentic

Algorithmus 8: *findIdentic*: $M \times T \times Bool \to Bool$

```
Input: p = (f, p_0, ..., p_{n-1}) ∈ M, t = (f, t_0, ..., t_{n-1}) ∈ T,
       starteGematcht ∈ Bool
let i ← 0
if starteGematcht
    i ← n
    goto zurück
loop
matche p_i    while findMatch(p_i, t_i)
                  i ← i + 1
                  if i = n return true
zurück        do
                  if i = 0 return false
                  i ← i − 1
              while not rematch(p_i, t_i)
              i ← i + 1
```

Als Strategie mit den wenigsten Freiheiten ist die Umsetzung von *findIdentic* die kürzeste. Die Laufvariable i steht gleichzeitig als Index für die Argumente von Muster und Literal. Wenn der Aufruf aus *findMatch* erfolgt, wird mit $i = 0$ gestartet und in Abschnitt `matche` p_i versucht, für alle i bis $n-1$ p_i mit t_i zu matchen. Sollte das für ein i fehlschlagen, besteht für entsprechende Muster die Möglichkeit, dass eines der Argumente $p_j \in \{p_0, \ldots, p_{i-1}\}$ anders als bisher mit t_j gematcht werden kann, was dann das Match von p_i mit t_i ermöglicht. Gefunden wird p_j in Abschnitt `zurück`. Soll das gesamte Muster neu gematcht werden, startet *findIdentic* bei einem Aufruf durch *rematch* deswegen bei `zurück` und mit $i = n$.

Lemma 4.4. Die Laufzeitkomplexität von Algorithmus 8 bei der Suche eines Matches für ein Muster $p = (f, \mathbf{x}_0, \ldots, \mathbf{x}_{n-1})$ mit einem Literal $t = (f, t_0, \ldots, t_{n-1})$ ist in $\mathcal{O}(n)$, wenn jeder Parameter t_i von t nur $\mathcal{O}(1)$ Konstantensymbole und Funktionssymbole besitzt.

Beweis. Der Ausdruck *not rematch*(pi, ti) ist für kein i wahr, da *rematch* für Mustervariablen immer *false* zurückgibt. Die äußere Schleife wird folglich nur exakt einmal durchlaufen. Sowohl ein Durchlauf der **while**-Schleife als auch ein Durchlauf der **do-while**-Schleife ist in $\mathcal{O}(1)$, da jedes t_i nur $\mathcal{O}(1)$ Teilterme hat bzw. *rematch*(pi, ti) für Mustervariblen p_i direkt *false* zurückgibt. Entweder wird der Abschnitt `matche` p_i exakt n Mal abgelaufen und *true* zurückgegeben oder $n' < n$ Mal abgelaufen, woraufhin auch Abschnitt `zurück` n' Mal abgelaufen wird, bis *false* zurückgegeben wird. □

4.3.3 findPermutation

Algorithmus 9: *findPermutation*: $M \times T \times Bool \to Bool$

Input: $p = (f, p_0, \ldots, p_{m-1}) \in M$, $t = (f, t_0, \ldots, t_{n-1}) \in T$,
starteGematcht $\in Bool$
let $i \leftarrow 0$, $k \leftarrow 0$
if *starteGematcht*
  $i \leftarrow m$
  goto `zurück`
if $m > n$
  return *false*
matche p_i **while** $i < m$
  while $k < n$
    if t_k ist mit keinem Parameter von p gematcht
      if *findMatch*(p_i, t_k)
        goto `weiter`
    $k \leftarrow k + 1$
  zurück **if** $i = 0$
    return *false*
  $i \leftarrow i - 1$
  $k \leftarrow k'$ aus „p_i ist mit $t_{k'}$ gematcht“
  if not *rematch*(p_i, t_k)
    merke: p_i ist nicht mehr mit t_k gematcht
    $k \leftarrow k + 1$
    goto `matche` p_i
  weiter merke: p_i ist mit t_k gematcht
  $i \leftarrow i + 1$
  $k \leftarrow 0$
return p enthält eine Multi-Mustervariable **or** alle t_k wurden gematcht

Algorithmus 9 versucht, das Muster einer kommutativen Funktionsanwendung p auf ein Literal t der gleichen Form zu matchen. Die beiden Laufvariablen i und k sind Index der

Musterargumente $p_0, \ldots, p_{m-1}$, bzw. der Argumente des Literals $t_0, \ldots, t_{n-1}$. Für die Suche eines Matches wird zuerst versucht, p_0 mit t_0 zu matchen. Schlägt das fehl, wird k hochgezählt, bis p_0 ein t_k matchen kann oder jedes t_k getestet wurde. Im erfolglosen Fall wird die Suche beendet, da für p_0 alle verfügbaren Freiheitsgrade getestet wurden. Wurde p_0 erfolgreich mit t_k gematcht, wiederholt sich der Prozess für p_1, mit der Ausnahme, dass t_k jetzt nicht mehr als Matchkandidat zur Verfügung steht. Wird ein Argument gefunden, das p_1 matcht, wird $i = 3$ gesetzt und der Prozess wiederholt sich für p_3. Sollte die Suche für p_1 in dem Durchlauf erfolglos sein, heißt das allerdings nicht, dass kein Match von p und t möglich ist. Es besteht die Option, dass p_0 mit t_k noch auf eine andere Weise als die bisherige gematcht werden kann. Beinhaltet p_1 Mustervariablen, die in p_0 bindend vorkommen, eröffnet ein Ändern der Bindung möglicherweise neue Matchmöglichkeiten für p_1. Aus diesem Grund wird im Abschnitt `zurück` zuerst versucht, p_0 mit t_k zu rematchen. Sollte das fehlschlagen, ist es möglich, dass p_0 noch mit Parametern von t gematcht werden kann, die nach t_k aufgelistet sind, was wiederum p_1 erlauben würde, ein Match mit t_k zu testen. Auch diese Option wird getestet. Wurde für alle p_i ein Match gefunden, so ist ganz p mit ganz t gematcht, falls gleichzeitig alle t_k gematcht sind oder p eine Multi-Mustervariable enthält. Sollte keiner der beiden Fälle eintreten, ist an dieser Stelle kein Match von p und t möglich.

Lemma 4.5. Die Laufzeitkomplexität von Algorithmus 9 bei der Suche eines Matches für ein Muster $p = (f, \mathbf{x}_0, \ldots, \mathbf{x}_{m-1})$ mit einem Literal $t = (f, t_0, \ldots, t_{n-1})$ ist in $\mathcal{O}(n^m)$, wenn jedes Argument t_k von t nur $\mathcal{O}(1)$ Konstantensymbole und Funktionssymbole besitzt.

Beweis. Der Beweis erfolgt als Induktion über m. Für den Induktionsanfang mit $m = 1$ muss $\mathbf{x}_0$ höchstens mit allen n Argumenten des Literals verglichen werden. Unabhängig davon, ob die Instanz von $\mathbf{x}_0$ bindend oder gebunden ist, terminiert *findMatch*$(\mathbf{x}_0, t_k)$ in $\mathcal{O}(1)$, da alle t_k in ihrer Größe beschränkt sind.
Im allgemeinen Fall kann die Anwendung von *findPermutation* mit p und t auf höchstens m Anwendungen des Algorithmus mit Mustergröße $m-1$ zurückgeführt werden. Erneut kann $\mathbf{x}_0$ potenziell mit jedem t_k matchen. Das Match mit t_k erfolgt wie im Induktionsanfang beschrieben in $\mathcal{O}(1)$. Die anschließende Suche nach Matches für x_i mit $i > 0$ ist in der Komplexität äquivalent zu einem neuen Aufruf von *findPermutation* mit dem Muster $p' = (f, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1})$ und dem Literal $t' = (f, t_0, \ldots, t_{k-1}, t'_k, t_{k+1}, \ldots, t_{n-1})$, wobei t'_k ein spezieller Wert ist, mit dem ein Match zu jedem Muster in $\mathcal{O}(1)$ abgelehnt wird [7]. Gibt dieser Aufruf *false* zurück, gibt auch *rematch*$(\mathbf{x}_0, t_k)$ in $\mathcal{O}(1)$ *false* zurück. Der Übergang von t_k zu t_{k-1} erfolgt ebenfalls in $\mathcal{O}(1)$. Für jedes der bis zu n Matches von $\mathbf{x}_0$ mit einem t_k treten damit Laufzeitkosten von $\mathcal{O}(1)$ außerhalb der Rekursion auf. Jeder der n Rekursionsaufrufe hat nach Induktionshypothese eine Laufzeit in $\mathcal{O}(n^{m-1})$. Insgesamt ergibt sich so also eine Laufzeit in $n \cdot \mathcal{O}(1) \cdot \mathcal{O}(n^{m-1}) = \mathcal{O}(n^m)$. □

[7] Der Wert von t'_k ist sonst nicht notwendig, da in Bereich `matche` p_i durch die if-Abfrage der problematische Matchversuch mit t_k umgangen wird.

4.3.4 findDilation

Algorithmus 10 teilt die Musterargumente $p_0, \ldots, p_{m-1}$ in Blöcke der Form $p_i, \ldots, p_j$ ein. Ein solcher Block muss elementweise einen Block $t_k, \ldots, t_{k+j-i}$ der Argumente des Literals matchen. Die Aufteilung der Musterargumente in Blöcke ist dabei fest: Wenn zwischen p_{i-1} und p_i eine Multi-Mustervariable liegt, ist dort eine Blockgrenze. Aus diesem Grund wird in `matche` p_i nur mehr als ein Schleifendurchlauf erlaubt, wenn p_i einen neuen Block beginnt, t_k also nicht durch den bereits gematchen Beginn des Blockes festgelegt ist. Konnte p_i nicht gematcht werden, wird i im Bereich `zurück` so lange heruntergezählt, bis entweder *rematch* erfolgreich ist oder p_i das erste Element des aktuellen Blocks ist. Besonderes Verhalten tritt erneut für $i = 0$ auf. War jeder Matchversuch für p_0 erfolglos, gibt es keine Möglichkeit mehr, die Teilterme von p so zu „strecken“[8], dass ein Match mit t gefunden werden kann, ähnlich der Situation für p_0 in *findPermutation*. Anders als bei *findPermutation* wird k im Bereich `weiter` hochgezählt, da p_i nie ein Literal t_k matchen darf, wenn p_{i-1} bereits mit t_l gematcht ist und $l > k$.

Lemma 4.6. Sei p ein Muster bestehend aus einer Funktionsanwendung des Funktionssymbols f auf auf m Mustervariablen $\mathbf{x}_0, \ldots, \mathbf{x}_{m-1}$. Jede Mustervariable $\mathbf{x}_i$ ist Nachfolger einer Multi-Mustervariable und auf x_{m-1} folgt eine Multi-Mustervariable. Für $m = 3$ gilt also $p = (f, \mathbf{as}..., \mathbf{x}_0, \mathbf{bs}..., \mathbf{x}_1, \mathbf{cs}..., \mathbf{x}_2, \mathbf{ds}...)$. Die Laufzeitkomplexität von Algorithmus 10 bei der Suche eines Matches für ein Muster p mit einem Literal $t = (f, t_0, \ldots, t_{n-1})$ ist in $\mathcal{O}(n^m)$, wenn jedes Argument t_k von t nur $\mathcal{O}(1)$ Konstantensymbole und Funktionssymbole besitzt.

Beweis. Sei $D(m, n)$ die asymptotische Laufzeit von Algorithmus 10. Erneut wird ein Induktionsbeweis über m geführt. Mit $m = 1$ ist der Fall identisch zu dem Induktionsanfang des Beweises von Lemma 4.5, es gilt $D(1, n) = \mathcal{O}(n)$.

Für $m > 1$ wird $D(m, n)$ auf $D(m-1, n)$ zurückgeführt. Für jedes k muss nach erfolgreichem Match von $\mathbf{x}_0$ mit t_k versucht werden, die restlichen Mustervariablen $\mathbf{x}_1, \ldots, \mathbf{x}_{m-1}$ in den restlichen Argumenten $t_{k+1}, \ldots, t_{n-1}$ des Literals t zu matchen. Das entspricht einem erneuten Aufruf von *findDilation* mit neuem Muster p' ohne $\mathbf{x}_0$ und neuem Literal $t' = (f, t_{k+1}, \ldots, t, n-1)$. Im rechenaufwändigsten Fall passiert das für jedes k.

$$D(m, n) = \sum_{k=0}^{n-1} (\mathcal{O}(1) + D(m-1, n-k-1))$$

Trotz der vorgegebenen Reihenfolge der Matches von Musterargumenten im Literal, folgt

[8] daher der Name *findDilation*

Algorithmus 10: *findDilation*: $M \times T \times Bool \to Bool$

Input: $p = (f, p_0, \ldots, p_{m-1}) \in M$, $t = (f, t_0, \ldots, t_{n-1}) \in T$
let $i \leftarrow 0$, $k \leftarrow 0$
if *starteGematcht*
  $i \leftarrow m$
  goto `zurück`
if $m = 0$
  return *true*
if $n = 0$
  return *false*
matche p_i **if** $k < n$
  do
    if *findMatch*(p_i, t_k)
      goto `weiter`
    $k \leftarrow k + 1$
  while p_i ist Nachfolger einer Multi-Mustervariable **and** $k < n$
zurück **while** $i > 0$
  $i \leftarrow i - 1$
  $k \leftarrow k'$ aus „p_i ist mit $t_{k'}$ gematcht“
  if *rematch*(p_i, t_k)
    goto `weiter`
  else if p_i ist Nachfolger einer Multi-Mustervariable
    $k \leftarrow k + 1$
    goto `matche` p_i
return *false*
weiter merke: p_i ist mit t_k gematcht
$i \leftarrow i + 1$
$k \leftarrow k + 1$
if $i < m$
  goto `matche` p_i
else if $k < n$ **and** nach p_{m-1} folgt keine Multi-Mustervariable
  goto `zurück`
return *true*

die selbe Komplexitätsabschätzung wie für *findPermutation*.

$$\begin{aligned} D(m,n) &= \sum_{k=0}^{n-1}(\mathcal{O}(1) + D(m-1, n-k-1)) \\ &< \sum_{k=0}^{n-1}(\mathcal{O}(1) + D(m-1, n)) \\ &= \mathcal{O}(n) + \mathcal{O}(n) \cdot D(m-1, n) \\ &= \mathcal{O}(n) \cdot D(m-1, n) \\ &= \mathcal{O}(n^m) \end{aligned}$$

□

4.4 Bessere Laufzeit für kommutative Muster

Benanav zeigte 1987, dass auch das Matchproblem mit einem kommutativen Funktionssymbol NP-vollständig ist [BKN87]. Dennoch kann die Laufzeit von *findPermutation* für bestimmte Arten von Mustern verbessert werden. In diesem Abschnitt wird das versucht, indem a priori ausgeschlossen wird, dass bestimmte Reihenfolgen von Musterparametern erfolgreich matchen können. Diese Reihenfolgen müssen dann im Algorithmus nicht mehr geprüft werden. Voraussetzung dafür wird sein, dass sowohl Muster als auch Literal mit *normalize* aus Kapitel 3 normalisiert werden. Insbesondere die Sortierung nach der Relation $<$ aus Definition 3.1 ist relevant. Bestimmte Muster mit kommutativen Funktionssymbolen können mit dieser einfachen Maßnahme bereits in linearer Zeit mit einem Literal abgeglichen werden. Besteht ein Muster etwa aus der Anwendung eines kommutativen Funktionssymbols f auf m Argumente $p_1, \ldots, p_m$ und sind die Argumente p_i ausschließlich Funktionsanwendungen paarweise verschiedener Funktionssymbole, ist garantiert, dass die Argumente eines normalisierten Literals für ein Match in der selben Reihenfolge liegen müssen. Wenn auch alle Teilmuster p_i dieser Struktur folgen bzw. nicht kommutativ sind, wird ein Match zuverlässig bereits mit *findIdentic* (Algorithmus 8) gefunden[9]. Das liegt daran, dass für alle $i, j \in \{1, \ldots, m\}$ mit $i < j$ gilt, dass p_i ausschließlich Literale matchen kann, die vor jedes Literal sortiert werden, welches mit p_j gematcht werden könnte. Diese Idee der Ordnung von Mustern wird im Folgenden ausgeführt.

Definition 4.7. Man sagt, das Muster p_1 ist stark kleiner als das Muster p_2 oder $p_1 \prec p_2$, wenn für alle Matchfunktionen v_p gilt, dass $lit(p_1, v_p) < lit(p_2, v_p)$. Gilt immer $lit(p_1, v_p) \leq lit(p_2, v_p)$, sagt man p_1 ist stark kleiner-gleich als p_2 oder $p_1 \preceq p_2$.

Beispiel 4.8. Wenn `sin` $<$ `cos`, gilt $p_1 \prec p_2$ für $p_1 = (\texttt{pow}, (\texttt{sin}, \texttt{x}), 2)$ und $p_2 = (\texttt{pow}, (\texttt{cos}, \texttt{y}), 2)$. Der Beweis folgt mit Lemma 4.10.

[9] ohne Berücksichtigung von Matches, welche durch Assoziativität ermöglicht würden

Die Muster $\hat{p}_1 = (\texttt{pow}, \mathbf{x}, 2)$ und $\hat{p}_2 = (\texttt{pow}, \mathbf{y}, 3)$ sind zueinander nicht stark geordnet. Um das zu zeigen, werden drei Literale t_1, t_2 und t_3 mit $t_1 < t_2 < t_3$ genutzt. Können sowohl t_1 als auch t_3 mit $\hat{p}_1$ matchen und t_2 mit $\hat{p}_2$, sind die Muster nicht stark geordnet. Getrennte Matches reichen aus, da $\hat{p}_1$ und $\hat{p}_2$ keine gemeinsamen Mustervariablen besitzen. Mit $1 < 2 < 3$ erfüllen $t_1 = (\texttt{pow}, 1, 2)$, $t_2 = (\texttt{pow}, 2, 3)$ und $t_3 = (\texttt{pow}, 3, 2)$ die Bedingungen.

Lemma 4.9. Sei p ein beliebiges Muster ohne die Mustervariable $\mathbf{x}$, q ein Muster, welches nur aus der Mustervariable $\mathbf{x}$ besteht, und sei die Menge der Konstantensymbole C ohne Minimum. Es gilt $p \not\preceq q$, und $q \not\preceq p$.

Beweis. Mit beliebiger Matchfunktion v_p sei $t_2 = \mathit{lit}\ (p, v_p)$. Nach Lemma 3.2 gibt es die Literale $t_1 < t_2$ und $t_3 > t_2$. Mit v'_p identisch zu v_p, nur $v'_p\ x = t_1$, gilt

$$\mathit{lit}(p, v'_p) = t_2 > t_1 = \mathit{lit}(q, v'_p) \implies p \not\preceq q.$$

Mit v''_p identisch zu v_p, nur $v''_p\ x = t_3$, gilt

$$\mathit{lit}(p, v''_p) = t_2 < t_3 = \mathit{lit}(q, v''_p) \implies q \not\preceq p.$$

□

Einfach zu sehen ist, dass nur ein kleiner Teil der Muster zueinander stark geordnet ist, wenn die allgemeinste Form des Matches erlaubt wird. Problematisch ist dabei vor allem *combine* (Algorithmus 2) als Teil von *normalize*. Dadurch wird es möglich, Funktionsanwendungen als Muster mit Konstantensymbolen als Literal zu matchen. Da es aufwändig ist, überhaupt zu bestimmen, welche Muster Matches dieser Art erlauben, wird *normalize* hier abweichend von Kapitel 3 ohne *combine* angenommen. Nur Assoziativität und Kommutativität werden berücksichtigt.

Lemma 4.10. Für folgende Formen von normalisierten Mustern p und q gilt $p \prec q$:

1. p und q enthalten keine Mustervariablen und $p < q$
2. p ist ein Konstantensymbol aber keine Mustervariable und q ist eine Funktionsanwendung
3. p und q sind Funktionsanwendungen verschiedener Funktionssymbole f und g mit $f < g$
4. $p = (f, p_0, \ldots, p_{m-1})$ und $q = (f, q_0, \ldots, q_{n-1})$ sind Funktionsanwendungen desselben nicht-kommutativen Funktionssymbols f und einer der folgenden Punkte trifft zu
   a) $m < n$, $\forall j \in \{0, \ldots, m-1\} \colon p_j = q_j$, p hat keine Multi-Mustervariablen in seinen Argumenten und wenn q, dann erst nach q_m
   b) $\exists i < \mathit{min}\{n, m\} \colon p_i \prec q_i$, $\forall j \in \{0, \ldots, i-1\} \colon p_j = q_j$ und weder p noch q haben Multi-Mustervariablen in ihren Argumenten vor p_i bzw. q_i

Beweis.
In allen Fällen wird eine Mustervariable nur an Punkten erlaubt, die nicht zur Bestimmung der Ordnung von p und q unter der Relation $<$ beitragen bzw. die für kein Literal anstelle der Mustervariable die Ordnung der normalisierten Terme beeinflussen würden. Trivial ist das für Fall 1.
Für die restlichen Fälle muss klar sein, dass ein Literal der Form $(f, xs...)$ auch nach Normalisierung diese Form behält. Die Reihenfolge der Argumente und damit auch, ob f kommutativ ist, spielen keine Rolle. Das Normalisieren mehrerer geschachtelter Anwendungen desselben assoziativen Funktionssymbols f entfernt nie die äußerste Funktionsanwendung. Die Struktur bleibt erhalten. Fall 2 und Fall 3 sind damit bewiesen.
Mit derselben Argumentation bleibt f auch in Fall 4 immer das äußerste Funktionssymbol erhalten. Da *normalize* in Fall 4 die Argumentreihenfolge nicht ändern darf, können Mustervariablen auch nach Auswertung der Musterinterpretation nicht den vorderen Bereich der Funktionsanwendungen beeinflussen. □

4.5 Termersetzungssystem

Das Ziel dieses Abschnittes ist erneut die Normalisierung eines Terms. Im Unterschied zu Kapitel 3 werden die Ersetzungsregeln hier nicht im Algorithmus festgelegt, sondern erst als Parameter übergeben. Die Thematik soll in dieser Arbeit allerdings nur angerissen werden.

Algorithmus 11: *applyRuleset*: $\mathit{Regelmenge} \times T \to T$

Input: $R \in \{M \times M\}$, $t \in T$
$t \leftarrow$ *normalize* t
while $\exists (p, p') \in R$, r Nachkomme von t, v_p Match von p und r
  ersetze r in t durch $r' = \mathit{lit}(p', v_p)$
  $t \leftarrow$ *normalize* t
return t

Ist eine Ersetzungsregel auf das übergebende Literal $t = t^{(0)}$ oder ein Teil dessen anwendbar, so wird das Ergebnis der Ersetzung $t^{(1)}$ genannt. Auf dem selben Weg kann aus $t^{(1)}$ der Term $t^{(2)}$ erzeugt werden oder allgemeiner aus $t^{(i)}$ der Term $t^{(i+1)}$. Ist auf keinen Teil von $t^{(n)}$ mehr eine Regel anwendbar, wird $t^{(n)}$ als Normalform von t zu den übergebenden Ersetzungsregeln bezeichnet.

4.5.1 Konfluenz

Es ist möglich, dass ein Literal t mit einer bestimmten Regelmenge mehr als nur eine Normalform besitzt. Eine einfache Regelmenge mit dieser Eigenschaft besteht aus zwei Regeln mit identischer linker Seite aber unterschiedlicher rechter Seite. Bestimmte Regeln können allerdings auch in Isolation mehrere Normalformen produzieren. Ein Beispiel ist

die folgende Regel, welche $\mathbf{x}$ ausklammert:

$$(\texttt{sum}, \mathbf{x}, (\texttt{prod}, \mathbf{x}, \mathbf{ys}...), \mathbf{zs}...) \mapsto (\texttt{sum}, (\texttt{prod}, \mathbf{x}, (\texttt{sum}, 1, (\texttt{prod}, \mathbf{ys}...)), \mathbf{zs}...)$$

Für das Literal $t = (\texttt{sum}, a, b, (\texttt{prod}, a, b))$ existiert sowohl die Normalform

$$t' = (\texttt{sum}, b, (\texttt{prod}, a, (\texttt{sum}, 1, b))),$$

als auch

$$t'' = (\texttt{sum}, a, (\texttt{prod}, b, (\texttt{sum}, 1, a))).$$

Das ist an dieser Stelle nicht neu: Algorithmus 7 (*rematch*) ist eine direkte Antwort auf Muster dieser Art. Die Fragestellung, welche Mengen von Ersetzungsregeln eindeutige Normalformen unabhängig von der Reihenfolge ihrer Anwendung produzieren (konfluent sind), ist im allgemeinen Fall nicht entscheidbar [BN98a]. Einschränkungen einer Regelmenge, die Konfluenz implizieren, werden etwa von Hoffmann und O'Donnell diskutiert [HO82]. Allerdings sind diese mit den hier beschriebenen Optionen für Muster nur schwer vereinbar. Zum einen findet der Matchbegriff dort ausschließlich in seiner strengen Form Anwendung, zum anderen beinhalten die Einschränkungen etwa die Restriktion, dass jede Mustervariable höchstens einmal in der linken Seite einer Ersetzungsregel vorkommen darf.
Ein Weg die Problematik teilweise zu umgehen ist, parallel alle möglichen Transformationen anzuwenden und erst am Ende mit einer entsprechenden Gewichtsfunktion die gewünschte Normalform zu wählen. Aufgrund der damit verbundenen hohen Laufzeitkosten wird dieser Ansatz in den meisten Fällen von Anfang an ausgeschlossen.

4.5.2 Ersetzungsreihenfolge

Wird eine gegebene Regelmenge als nicht konfluent angenommen, kann die Normalform eines Literals t nicht nur davon abhängen, welche Regel zuerst auf ihre Anwendbarkeit getestet wird, sondern auch an welcher Stelle eine Ersetzung priorisiert ist. Weiter ist es möglich, dass für bestimmte Literale manche Strategien eine Normalform erzeugen, während andere nicht konvergieren.

Beispiel 4.11. Die Fakultätsfunktion, repräsentiert durch das Funktionssymbol `fact`, kann durch die folgende Menge von Ersetzungsregeln beschrieben werden, wenn die Funktionssymbole `eq`, `prod` und `sub` mit *normalize* ausgewertet werden.

$$(\texttt{cond}, \texttt{true}, \mathbf{x}, \mathbf{y}) \mapsto \mathbf{x} \tag{1}$$
$$(\texttt{cond}, \texttt{false}, \mathbf{x}, \mathbf{y}) \mapsto \mathbf{y} \tag{2}$$
$$(\texttt{fact}, \mathbf{x}) \mapsto (\texttt{cond}, (\texttt{eq}, \mathbf{x}, 0), 1, (\texttt{prod}, \mathbf{x}, (\texttt{fact}, (\texttt{sub}, \mathbf{x}, 1)))) \tag{3}$$

Offensichtlich ist, dass die Normalisierung nicht konvergiert, wenn entweder die Ersetzung von Regel (3) an einer beliebigen Stelle Vorzug gegenüber Anwendung der anderen Regeln hat oder wenn immer der innerste transformierbare Teil eines Literals transformiert wird.

Die Problematik ist eng verwandt mit der Auswertungsstrategie für funktionale Programme, wo zwischen *strenger Auswertung* (engl. *eager evaluation*) und *fauler Auswertung* (engl. *lazy evaluation*) unterschieden wird [Hud89]. Die strenge Auswertung wählt immer die innerste Ersetzung zuerst aus. Der Vorteil ist hierbei die sehr einfache Umsetzung der Strategie: Ist ein Teilterm normalisiert, wird er es auch nach Anwendung von Ersetzungsregeln auf seine Ahnen bleiben. Ein Nachteil besteht darin, dass weniger Terme normalisiert werden können, siehe Beispiel 4.11. Die faule Auswertung transformiert dagegen immer den äußeren Teil des Literals zuerst.
Für die Umsetzung einer faulen Ersetzungsstrategie für ein Termersetzungssystem ist es im allgemeinen Fall schwierig, schnell den äußersten transformierbaren Teil zu finden, nachdem eine Ersetzung stattgefunden hat. Hoffman und O'Donnell haben aus diesem Grund einen Hybrid aus strenger und fauler Auswertung implementiert [HO82]. In dieser Arbeit fällt das Problem weniger ins Gewicht, da schon der einzelne Matchversuch je nach Muster exponentielle Laufzeitkosten aufweisen kann.

5 Umsetzung

Der gesamte Quelltext der Umsetzung in C++ ist auf GitHub einsehbar [Bor21]. Hier gezeigte Ausschnitte sind teilweise gekürzt. Dieses Kapitel hat keinen Anspruch auf Vollständigkeit. Der Fokus liegt darauf, die theoretisch ausführlich beschriebenen Konzepte in der Praxis zu zeigen und die wichtigsten bisher in dieser Arbeit unbehandelten Ideen vorzustellen. Aufgrund der gegenseitigen Abhängigkeit der verschiedenen Ideen sollten die Abschnitte 5.1 und 5.2 nicht direkt mit dem Anspruch eines lückenlosen Verständnisses gelesen werden, da die darauf folgenden Abschnitte noch weiter ins Detail gehen.

5.1 Konzeptionelle Unterschiede

Die Umsetzung implementiert nicht exakt die bisher beschriebenen Strukturen. Der erste Unterschied ist, dass die Mengen der Funktionssymbole F und der Konstantensymbole C hier nicht unterschieden werden. Da der Zweck der Implementierung alleine in der Vereinfachung von Ausdrücken über den komplexen Zahlen $\mathbb{C}$ liegt, ist die Menge an Symbolen zudem im Code nicht generisch gehalten. Die beiden wichtigsten Arten von Symbolen für Literale sind komplexe Zahlen $z \in \mathbb{C}$, sowie Zeichenketten $c_1 c_2 \dots c_n$ beliebiger Länge n. Die einzelnen Zeichen c_i stammen dabei aus dem Alphabet Σ, welches aus Klein-und Großbuchstaben des lateinischen Alphabetes, Ziffern von 0 bis 9, dem Apostroph `'` und dem Unterstrich `_` besteht. Die Ausnahme bildet das erste Zeichen c_1, welches ein Klein- oder Großbuchstabe des Lateinischen Alphabetes sein muss. Zu den Zeichenketten gehören insbesondere auch die Funktionssymbole. Mit der dadurch entstehenden Möglichkeit, Funktionssymbole als Werte zu behandeln, ergibt es Sinn, auch Funktionsanwendungen von dynamisch bestimmten Funktionssymbolen zuzulassen. Das erste Element f des Funktionsanwendungstupels $(f, t_0, \dots, t_{n-1})$ ist in der Umsetzung deshalb kein Symbol, sondern ein Term. Mit den bis hier diskutierten Änderungen und Σ^+ als Bezeichnung für die Menge von Zeichenketten über dem beschriebenen Alphabet Σ, sähe die idealisierte Menge aller Literale in der Umsetzung im Gegensatz zu Definition 2.1 so aus:

$$T \coloneqq \Sigma^+ \cup \mathbb{C} \cup \{(t_0, \dots, t_n) \mid t_0, \dots, t_n \in T\}.$$

Die wichtigste Erweiterung der Implementierung ist die der anonymen Funktion, bekannt als Lambda. Das Lambdakalkül von Church [Chu36] definiert Terme verwandt mit den in dieser Arbeit diskutierten. Der wichtigste Unterschied zum bisherigen Funktionssymbol ist, dass keine externe Interpretation für einen gegebenen Lambdaausdruck notwendig ist. Anstatt Funktionssymbole über Namen zu identifizieren und die Abbildungsvorschrift getrennt anzugeben, ist eine Lambdafunktion $f \in \Lambda$ ausschließlich durch ihre Abbildungsvorschrift identifiziert. Church erlaubt neben der Funktionsanwendung als Term lediglich Variablensymbole $v \in V$ und Lambdafunktionen. Soll T also eine Obermenge aller Ausdrücke im Lambdakalkül werden, müsste prinzipiell nur

die Lambdafunktion selbst hinzugefügt werden. Die Menge der Variablen V, hier als Lambdaparameter bezeichnet, wird allerdings von den bisher erlaubten Zeichenketten in Σ^+ getrennt.
Muster sind auch in der Umsetzung eine Obermenge der Literale. Neben den weiteren dadurch entstehenden Formulierungsmöglichkeiten, kommt die Fähigkeit, Bedingungen an Mustervariablen zu stellen, um mögliche Matches weiter einzuschränken. Eine spezielle Form dieser Einschränkung ist dabei die der Wert-Mustervariable $w \in W$, welche versucht, Matches zu finden, die in Abschnitt 3.3 erlaubt werden, also komplexe Zahlen wieder in Rechenausdrücke zu dekonstruieren. Näher behandelt wird die Wert-Mustervariable noch in Abschnitt 5.4. Zuletzt kann die Multi-Mustervariable aus Abschnitt 4.2 von hier an nicht mehr nur als abstrakte Idee gehandelt werden. Die Menge der Multi-Mustervariablen wird X^* genannt.

Definition 5.1. Die Menge der Terme T ist in diesem Kapitel definiert als

$$T \coloneqq \Sigma^+ \cup \mathbb{C} \cup X \cup X' \cup X^* \cup W \cup V \cup \Lambda \cup \{(t_0, \ldots, t_n) \mid t_0, \ldots, t_n \in T\}$$

mit

$$\begin{aligned} \Sigma^+ &\coloneqq \text{Zeichenketten} \\ \mathbb{C} &\coloneqq \text{Komplexe Zahlen} \\ X &\coloneqq \text{Mustervariablen} \\ X^* &\coloneqq \text{Multi-Mustervariablen} \\ W &\coloneqq \text{Wert-Mustervariablen} \\ V &\coloneqq \text{Lambdaparameter} \\ \Lambda &\coloneqq \text{Lambdafunktionen.} \end{aligned}$$

5.2 Syntax

In der Umsetzung wird der Termbaum immer aus einer ASCII Zeichenkette gebaut. Solche Zeichenketten sind von hier an in `monospace` gesetzt. Auf eine formale Definition der genutzten Grammatik wird verzichtet.

5.2.1 Funktionsanwendungen

Die bisher genutzte Schreibweise (f, x, y, z) für die Funktionsanwendung des Funktionssymbols f auf die Parameter x, y, z wurde in Abschnitt 2.1 eingeführt, um den Term syntaktisch von anderen Ideen zu differenzieren. Dies ist für die Umsetzung in C++ nicht notwendig, da hier durch den Kontext klar ist, ob eine Zeichenkette in einen Term übersetzt wird. Aus dem Grund werden Funktionsanwendungen in der Syntax `f(x, y, z)` geparst. Weiter können Funktionsanwendungen bestimmter Funktionen auch mit fest definierten Infixoperatoren geschrieben werden, siehe Tabelle 5.2.1. Die Bindekraft der Operatoren nimmt in der Tabelle von oben nach unten ab. Aus der Tabelle hervorzuheben sind zwei Dinge: Zum einen wird ein Leerzeichen zwischen zwei Termen als

Normale Schreibweise	Alternative Syntax
`not(x)`	`!x`
`pow(x, y)`	`x^y`
`prod(-1, x)`	`-x`
`prod(x, y)`	`x * y`
`prod(x, y)`	`x y`
`prod(x, pow(y, -1))`	`x / y`
`sum(x, y)`	`x + y`
`sum(x, prod(-1, y))`	`x - y`
`cons(x, y)`	`x :: y`
`eq(x, y)`	`x == y`
`neq(x, y)`	`x != y`
`greater(x, y)`	`x > y`
`smaller(x, y)`	`x < y`
`greater_eq(x, y)`	`x >= y`
`smaller_eq(x, y)`	`x <= y`
`and(x, y)`	`x && y`
`or(x, y)`	`x || y`
`of_type__(x, y)`	`x :y`

Abbildung 5.2.1: alternative Syntax für bestimmte Funktionssymbole

Multiplikation interpretiert, Funktionsanwendungen dürfen also kein Leerzeichen zwischen Funktion und Parametertupel setzen. Der Ausdruck `f (x)` ist dementsprechend gleichbedeutend zu `prod(f, x)`, während `f (x, y)` einen Syntaxfehler darstellt. Zum anderen fehlen Funktionssymbole für die Darstellung des additiven Inversen und des multiplikativen Inversen. Dies reduziert die Anzahl der Ersetzungsregeln, die benötigt werden, um eine Gesetzmäßigkeit, die Summen oder Produkte involviert, abzubilden.

5.2.2 Lambdafunktionen

Da der ASCII Zeichensatz keine griechischen Buchstaben enthält, wird anstelle des kleinen Lambdas λ der umgekehrte Schrägstrich `\` genutzt. Die Identitätsfunktion $\lambda x.x$ wird dementsprechend `\x .x` geschrieben. Mehrere Parameter werden durch Leerzeichen getrennt: `\x y .pow(x, y)` ist eine Lambdafunktion mit identischem Verhalten zum Funktionssymbol `pow`.

5.2.3 Symbole

komplexe Zahlen auf der reellen Achse sind in der Syntax vergleichbar mit Darstellungen für Integer und Fließkommazahlen erlaubt in C. Möglich sind etwa `3.1415`, `42`, `1.337e3` oder `1.602e-19`. komplexe Zahlen auf der imaginären Achse sind in der Struktur identisch, allerdings immer direkt gefolgt vom Zeichen `i`.

```
3.5 + 7i

sin(3 pi / 2 t) + 5 cos(pi / 8 t)

map(tup, \x .x^2, tup(1, 2, 3, 10i))
```

Abbildung 5.2.2: Beispiele für Literale

Die verschiedenen Mustervariablen werden auf unterschiedliche Weise identifiziert. Eine normale Mustervariable muss mit einem Unterstrich beginnen (`_x`), eine Wert-Mustervariable mit einem Dollar (`$x`) und eine Multi-Mustervariable endet in drei Punkten (`xs...`). Besitzt ein Name keines dieser besonderen Merkmale, wird daraus kontextabhängig ein Lambdaparameter oder ein Symbol aus Σ^+. Ist der umgebende Name bereits in einem umschließenden Lambda gebunden, wird die innerste Bindung gewählt. Als Beispiel ist im Ausdruck `\x .x + y` das Symbol `x` als Lambdaparameter interpretiert, während `y` als Zeichenkette erhalten bleibt. Im Ausdruck `\x .\x .2 - x` kann der Parameter `x` des äußeren Lambdas in der Definition des inneren Lambdas nicht mehr referenziert werden. Der Gesamtausdruck ist damit identisch zu `(\x .(\y .2 - y))`.

5.2.4 Literale

Ein Literal kann Lambdafunktionen, Funktionsanwendungen, komplexe Zahlen und Symbole aus Σ^+ enthalten. Abbildung 5.2.2 enthält Beispiele.

5.2.5 Ersetzungsregeln

Es gibt zwei Varianten Ersetzungsregeln zu schreiben.

<linke Seite> `=` *<rechte Seite>*
<linke Seite> `|` *<Bedingungen>* `=` *<rechte Seite>*

In der zweiten Variante ist *<Bedingungen>* eine kommaseparierte Liste von Bedingungen an die vorkommenden Mustervariablen, wie in Abschnitt 5.4.2 diskutiert. Beispiele sind in Abbildung 5.2.3 aufgeführt.
Die beiden letzten Regeln haben den exakt gleichen Effekt, wie noch in Abschnitten 5.4.1 und 5.4.2 erörtert. Während die vorletzte die Wert-Mustervariable `$a` einsetzt, bringt die letzte Regel die Mustervariablen `_aPow2` und `_2a` über die angestellte Bedingung in Beziehung zueinander.

5.3 Lambdafunktionen

Die umgesetzten Funktionen erweitern die Definition von Church, indem die selbe Lambdaabstraktion auch mehrere Parameter erlaubt. Während der Ausdruck $\lambda xy.x(y)$ für

```
id = \x .x

0 xs... = 0

cos(($k + 1/2) pi) | $k :int = 0

change(_f, _g, _f(xs...)) = _g(xs...)

$a^2 + 2 $a _b + _b^2 = ($a + _b)^2

_aPow2 + _2a _b + _b^2 | 4 _aPow2 == _2a^2 = (1/2 _2a + _b)^2
```

Abbildung 5.2.3: Beispiele für Ersetzungsregeln

Church also nur eine vereinfachte Schreibweise[1] für den Ausdruck $\lambda x.\lambda y.x(y)$ darstellt, handelt es sich für die hier beschriebene Umsetzung um zwei verschiedene Funktionen. Ein Lambdaparameter ist in der Syntax nicht von anderen Symbolen unterscheidbar[2], wird intern jedoch durch einen Index dargestellt. Die Darstellung unterscheidet sich von De Bruijn Indizierung [de 72] darin, dass jede Instanz derselben Variable immer denselben Index hat[3]. Jeder neu gebundene Parameter bekommt als Index die Anzahl der weiter außen bzw. in derselben Abstraktion vor ihm gebundenen Parameter. Zur Veranschaulichung wird hier der Index im Tiefsatz mitgeschrieben. Wichtig ist hervorzuheben, dass die hier nach wie vor dargestellten Namen ausschließlich der Übersichtlichkeit dienen und in der Umsetzung nicht gespeichert sind. Als Beispiel dient $\lambda x_0 y_1.\lambda z_2.x_0 + y_1 + z_2$. Das Zeichen x wird in der äußersten Abstraktion zuerst gebunden, hat also keine Vorgänger und dementsprechend Index 0. In derselben Abstraktion wird als zweiter Parameter weiter y gebunden. Als Nachfolger von x wird y Index 1 zugewiesen. Die Bindung von z liegt innerhalb der Abbildungsvorschrift des äußeren Lambdas, x und y können also referenziert werden. Dementsprechend hat z zwei Vorgänger und Index 2.
Die Auswertung der Funktionsanwendung einer Lambdafunktion f entspricht der β-Reduktion (von Church [Chu36] als *operation II* bezeichnet). Parameter von f werden dafür durch die übergebenen Argumente ersetzt. Enthält die Definition von f selbst weitere Lambdaabstraktionen, so werden die Indices derer Parameter um die Stelligkeit von f erniedrigt. Der Ausdruck $(\lambda x_0 y_1.\lambda z_2.x_0 + y_1 + z_2)(3, 6)$ wird damit zum Ausdruck $\lambda z_0.3 + 6 + z_0$ ausgewertet.
Die gewählte Indizierung hat ein Problem, welches die De Bruijn Indizierung nicht besitzt. Der Index eines Lambdaparameters ist abhängig vom Kontext der bindenden Ab-

[1] Die Klammern um y werden in der Literatur oft weggelassen, für diese Arbeit sind sie allerdings notwendig.
[2] siehe 5.2.2
[3] Sollte eine Ersetzungsregel auch innerhalb eines Lambdas anwendbar sein, ist eine solche Herangehensweise vorteilhaft, da der Matchalgorithmus bisher keinen Unterschied zwischen syntaktischer Gleichheit und semantischer Gleichheit macht.

straktion. Ändert sich dieser Kontext, stimmt die bisherige Indizierung möglicherweise nicht mehr: Enthält ein Argument a der Funktionsanwendung einer Lambdafunktion f selbst eine Lambdafunktion g, so reicht es für die Auswertung von f mit bisher diskutierten Konzepten nicht aus, den entsprechenden Lambdaparameter von f einfach durch a zu ersetzen. Sollte diese Ersetzung innerhalb einer Lambdaabstraktion f' stattfinden, müssen alle Lambdaparameter von g im Index um die Anzahl der weiter außen gebundenen Parameter erhöht werden. Als Beispiel ist $a = g = \lambda x_0.x_0$ die Identität, welche der Funktion $f = \lambda x_0.\lambda y_1.x_0(y_1)$ übergeben wird. Teil der Definition von f ist $f' = \lambda y_1.x_0(y_1)$.

$$(\lambda x_0.\lambda y_1.x_0(y_1))(\lambda x_0.x_0)$$

An dieser Stelle wird zudem eine alternative Notation eingeführt, die nur Informationen benutzt, die der Implementierung tatsächlich bekannt sind. Ehrlicher ist die Darstellung einer Lambdaabstraktion mit n Parametern als $\lambda[n].$<*Definition*>. Mit den Parametern geschrieben als Prozentzeichen gefolgt von ihrem Index, ist der problematische Ausdruck geschrieben als

$$(\lambda[1].\lambda[1].\%0(\%1))(\lambda[1].\%0).$$

Die naive Ersetzung von x_0 in f durch g würde im folgenden Ausdruck resultieren.

$$\lambda y_0.(\lambda x_0.x_0)(y_0)$$

Ohne Namen ist jetzt klar, dass in der Abbildungsvorschrift von g nun fälschlicherweise auf den Parameter von f' abgebildet wird.

$$\lambda[1].(\lambda[1].\%0)(\%0)$$

Es gibt verschiedene Möglichkeiten den Fehler zu beheben. Eine einfache Lösung ist, Indices von eingesetzten Argumenten in der Auswertung einer Lambdafunktion entsprechend anzupassen. Eine Konsequenz wäre aber, dass die Komplexität der Auswertung von der Größe der Argumente abhängig ist. Ein weiterer Nachteil ist, dass veränderte Argumente kopiert werden müssen. Das macht *lazy evaluation* [Hud89], also die Idee, jede Funktionsanwendung höchstens einmal auswerten zu müssen, unmöglich.
Eine Alternative ist die Unterscheidung zwischen sogenannten transparenten Lambdas und nicht-transparenten Lambdas, zweitere dargestellt durch umschließende geschweifte Klammern: {λ<*Parameter*>.<*Definition*>} bzw. {$\lambda[n].$<*Definition*>}. Kommt ein transparentes Lambda f' in der Definition eines Lambdas f vor, wird bei der Auswertung von f auch in f' nach Lambdaparametern gesucht und diese ersetzt bzw. deren Index erniedrigt. Transparente Lambdas verhalten sich damit nicht anders als die Lambdas bisher. Ist f' allerdings nicht-transparent, lässt die Auswertung von f die Definition von f' unberührt. Mit zwei weiteren Bedingungen wird die Überprüfung von Argumenten bei der Auswertung Lambdas dann obsolet: Zum einen darf ein außenliegendes Lambda f nicht transparent sein. Ein Lambda ist außenliegend, wenn es nicht Teil der Definition eines weiteren Lambdas ist, also zu seinen Ahnen keine Lambdaabstraktionen gehören. Zum anderen darf die Funktionsanwendung eines Lambdas nur ausgewertet werden,

wenn das Lambda außenliegend ist. Damit wird garantiert, dass während der Auswertung des Lambdas f kein Argument a direkt transparente Lambdas enthält, höchstens als Teil g' (im Beispiel nicht enthalten) eines nicht-transparenten Lambdas g.
Im bereits behandelten Problemfall sind f und g damit nicht-transparent. Nur f' ist zu Beginn transparent. Nach Auswertung von f ist f' dann jedoch ebenfalls außenliegend.

$$\begin{aligned} & \{\lambda x_0.\lambda y_1.x_0(y_1)\}(\{\lambda x_0.x_0\}) \\ \rightarrow\ & \{\lambda y_0.\{\lambda x_0.x_0\}(y_0)\} \end{aligned}$$

Alternativ ohne Namen:

$$\begin{aligned} & \{\lambda[1].\lambda[1].\%0(\%1)\}(\{\lambda[1].\%0\}) \\ \rightarrow\ & \{\lambda[1].(\{\lambda[1].\%0\})(\%0)\} \end{aligned}$$

Mit den soeben formulierten Bedingungen darf der Ausdruck nicht weiter transformiert werden. Das liegt daran, dass die Funktionsanwendung von g selbst noch vom Lambda f' umschlossen ist. Es ist also nicht ausgeschlossen, dass Argumente der Funktionsanwendung von g (hier nur %0) direkt transparente Lambdas enthalten[4].

5.4 Mustervariablen

In der Umsetzung wird zwischen drei Arten von Mustervariablen unterschieden. Die Sonderform der Multi-Mustervariable ist bereits aus Kapitel 4.2 bekannt, die Wert-Mustervariable ist allerdings neu. Als drittes ist die normale Mustervariable nach wie vor vorhanden. Wert-Mustervariable und normale Mustervariable können zudem durch Bedingungen eingeschränkt werden.

5.4.1 Wert-Mustervariable

Nach Definition 2.7 ist das Match v_p für ein Paar (p, t) aus Muster p und Literal t dann gültig, wenn $lit(p, v_p) = t$ gilt. In Kapitel 4 nicht diskutiert wurde die Möglichkeit, Zahlen mit Rechenausdrücken zu matchen. Ein Muster $p = (\texttt{prod}, 2, \mathbf{k})$ würde für das Literal $t = 12$ etwa ein Match v_p mit $v_p\ \mathbf{k} = 6$ besitzen. Um die Matchsuche nicht zu kompliziert zu gestalten, besitzt die Umsetzung zwei Einschränkungen. Zum einen ist eine solche Dekonstruktion des Musters zu einem Wert nur dann möglich, wenn das entsprechende Teilmuster exakt eine Instanz w einer Wert-Mustervariable enthält. Zum anderen werden nur bijektive Funktionen[5] dekonstruiert, wie im Beispiel die Multiplikation mit zwei. Die Umsetzung ist damit eine Verallgemeinerung von n+k-Mustern aus Haskell [21b].

[4] Eng verwandt ist der Grund, warum auch bei der Verwendung von De Bruijn Indices ohne weiteres nur die äußerste Funktionsanwendung abgebildet werden darf. In dem Fall könnte sich sonst die Anzahl der Abstraktionen zwischen einer Variable in einem Argument und ihrer eigenen Abstraktion ändern. Der alte De Bruijn Index würde die Variable dann der falschen Abstraktion zuordnen.

[5] bzw. sonst Mehrdeutigkeiten ignoriert

```
value_match__(_i, _dom, _inv) + _a + cs... | _a :complex
    = value_match__(_i, _dom, \x ._inv(x - _a)) + cs...

value_match__(_i, _dom, _inv) * _a * cs... | _a :complex
    = value_match__(_i, _dom, \x ._inv(x / _a)) * cs...

value_match__(_i, _dom, _inv) ^ 2
    = value_match__(_i, _dom, \x ._inv(sqrt(x)))

sqrt(value_match__(_i, _dom, _inv))
    = value_match__(_i, _dom, \x ._inv(x^2))
```

Abbildung 5.4.1: Transformationsregeln für Muster mit Wert-Mustervariable

Vor einem Matchversuch muss ein Muster mit Wert-Mustervariablen transformiert werden. Ziel der Transformation ist, dass für den Matchalgorithmus direkt klar ist, ab wo das Muster einen Wert repräsentieren soll. Der Teilbaum des Musters, der eine Wert-Mustervariable w enthält und als ganzes einen Wert matchen soll, ist idealerweise also direkt gekennzeichnet. In der Umsetzung ist das realisiert, indem eine Wert-Mustervariable im zu matchenden Muster kein Blatt darstellt, sondern einen Teilbaum markiert. Dieser entspricht nicht der ursprünglichen Umgebung um die Wert-Mustervariable, sondern derer Inversen.
Die Transformation in diese Form erfolgt über Anwendung von Ersetzungsregeln. Die Wert-Mustervariable wird dazu temporär als Funktionsanwendung des Symbols `value_match__` gespeichert. Die Regeln sind jeweils zweizeilig in Abbildung 5.4.1 aufgeführt. Das Funktionssymbol erwartet drei Argumente, hier die Mustervariablen `_i` für den Identifikationsindex, `_dom` für den erlauben Raum (kurz für *domain*) und `_inv` für den Teilterm, der für Inverse des bereits umgewandelten Teils der Umgebung steht. Zu Beginn steht die Identitätsfunktion an Stelle von `_inv`. Die finale Repräsentation einer Wert-Mustervariablen entspricht in den gespeicherten Informationen fast exakt der Zwischenform als Funktionsanwendung von `value_match__`. Ergänzt wird einzig, ob die jeweilige Instanz bindend oder gebunden ist. Die genaue Form von `_inv` ändert sich zudem auch leicht, die Idee allerdings nicht.
Im Match der Wert-Mustervariablen w mit einer komplexen Zahl z als Literal wird der in Abbildung 5.4.1 `_inv` genannte Teilbaum genutzt, um den Wert, den die Wert-Mustervariable an ihrem ursprünglichen Ort im Muster für ein Match besitzen müsste, zu bestimmen. Berechnet werden kann dieser als Anwendung von `_inv` auf das Literal.

$$w = \texttt{inv}(z)$$

Je nachdem, ob w eine bindende oder eine gebundene Instanz ist, wird der berechnete Wert im Matchzustand[6] gespeichert oder mit dem bereits gespeicherten Wert verglichen.

[6] siehe Abschnitt 5.6.2

Beispiel 5.2. Das Muster $p =$ `cos(2 $k pi) | $k :int` mit der Wert-Mustervariable $w =$ `$k` wird in der Zwischenrepräsentation dargestellt als

```
cos(2 value_match__(0, int, \y .y) pi).
```

Die zweite Regel aus Abbildung 5.4.1 lässt sich mit v_p `_a` $=$ `2` auf den gegebenen Term anwenden. Die Normalform von p ist das Ergebnis der Regelanwendung:

```
cos(value_match__(0, int, \x .(\y .y)(x / 2)) pi)
```

Nach Umwandlung der Funktionsanwendung von `value_match__` in die interne Darstellung, genannt w', hat das finale Muster die Form $p' =$ `cos(`w' `pi)`. Die interne Darstellung vereinfacht zudem den Ausdruck der Inversen. Ein Lambda näher an dieser Form ist `(\x .x / 2)`.
Das Literal `cos(8 pi)` kann mit p' gematcht werden. Dazu muss `8` mit w' matchen. Das funktioniert, denn `(\x .x / 2)(8)` wird zu v_p $w =$ `4` ausgewertet. Damit erfüllt das Match die für w zu Beginn geforderte Bedingung der ganzen Zahl.

Starke Ordnung

In Kapitel 4.4 wurden Wert-Mustervariablen ausgeschlossen, da sie die starke Ordnung vieler Muster zueinander zerstören. In der hier beschriebenen Umsetzung ist das allerdings nicht der Fall. Nach außen verhält sich eine Wert-Mustervariable in matchbereiter Konfiguration exakt wie eine normale Mustervariable, die ausschließlich bestimmte Teilmengen der komplexen Zahlen matchen kann. Die Frage, welche Funktionsanwendungen eines Musters auch Konstantensymbole matchen können, stellt sich damit nicht mehr während der Ausführung des Matchalgorithmus. Die entsprechenden Teile des Musters wurden bereits vorab erkannt und mit den Regeln aus Abbildung 5.4.1 so transformiert, dass die Wert-Mustervariable direkt den gesamten Teilbaum repräsentiert.
Nachteil dieser Vorgehensweise ist, dass an manchen Stellen mehrere fast identische Muster benötigt werden, da Wert-Mustervariablen jetzt ausschließlich Werte erkennen, nicht mehr die ursprünglichen Teilbäume.

5.4.2 Bedingungen

Muster können durch Bedingungen an normale Mustervariablen in möglichen Matches weiter eingeschränkt werden. Diese Bedingungen übernehmen teilweise eine ähnliche Rolle wie die Wert-Mustervariablen[7], sind dabei aber etwas flexibler.

Mögliche Ausdrücke

Bedingungen, die nicht mit Wert-Mustervariablen dargestellt werden könnten, nutzen primär die Funktionssymbole `contains`, `neq` und `of_type__`. Ersteres ist binär und prüft, ob das erste Argument Teilbaum des zweiten Argumentes ist[8]. Zweiteres (normal

[7] siehe Abschnitt 5.7.2
[8] siehe Abschnitt 5.7.1

in Infixform `!=` geschrieben) ist ebenfalls binär und prüft, ob zwei Teilbäume ungleich sind. Beide dürfen dabei ausschließlich direkt Mustervariablen als Argumente übergeben bekommen.
Genauso ist das Symbol `of_type__` binär und wird normal nicht ausgeschrieben hingeschrieben, sondern tritt als Infixoperator `:` auf. Die Bedeutung variiert dann nach dem rechten Argument. Ist dies ein Knotentyp, etwa `complex` oder `f_app`, prüft `of_type__`, ob das linke Argument von diesem Typ ist. Eine spezifischere Variante der Bedingung `_x :f_app` ist `_x :`f, mit einem Symbol f. In dem Fall wird geprüft, ob die linke Seite eine Funktionsanwendung von f ist. Beispielsweise muss mit der Bedingung `_x :sum` die Mustervariable `_x` mit einer Summe, also einer Funktionsanwendung des Symbols `sum`, gematcht sein, damit das Match gültig ist. Der dritte Spezialfall tritt auf, wenn die rechte Seite eine fest definierte Teilmenge der Komplexen Zahlen ist, etwa `_x :int`. Die linke Seite muss dann nicht direkt eine Mustervariable sein, sondern lediglich zu einer komplexen Zahl ausgewertet werden können[9].

Umsetzung

Eine Bedingung, die mehrere Mustervariablen enthält, wird der Mustervariablen hinzugefügt, deren bindende Instanz auf der linken Regelseite als letztes vorkommt. Ist eine solche Mustervariable dann von mehreren Bedingungen abhängig, werden diese als Argumente des variadischen Funktionssymbols `and` zusammengefasst. Die Überprüfung einer Bedingung wird durchgeführt, wenn Algorithmus *findMatch* die entsprechende Mustervariableninstanz als Argument erhält.

5.5 Datenstruktur

Die Knoten des Termbaums liegen in einem Array, dem sogenannten `Store`. Während eine komplexe Zahl oder ein Lambdaparameter immer ein Arrayelement besetzen, kann die Funktionsanwendung auch mehrere direkt hintereinanderliegende Arrayelemente nutzen, je nachdem, wie viele Argumente referenziert werden müssen. Referenziert wird ein Knoten aus einem Paar aus Arrayindex und Knotentyp, beide dargestellt als natürliche Zahl. Dieses Paar wird `NodeIndex` genannt. Der Knotentyp wird als `NodeType` bezeichnet und kann die in Tabelle 5.5.1 gelisteten 12 verschiedene Werte annehmen, eng verwandt mit Definition 5.1.
Manche der hier aufgezählten Knoten müssen neben ihrem Typen nur eine einzige natürliche Zahl kennen. Bereits diskutiert wurde das für Knoten von Typ 3, den Lambdaparametern. Für einen Knoten dieser Art ist es nicht notwendig, die Information als eigenes Arrayelement des `Store` zu speichern; der Index des Paares aus Index und Typ von `NodeType` enthält direkt den erforderlichen Wert. Damit werden die entsprechenden

[9] siehe Abschnitt 5.7.2

[9] Die Nutzung von Fließkommazahlen ist für Computeralgebra nur bedingt geeignet, erlaubt aber die Verwendung der im C++ Standard geforderten Rechenoperationen für komplexe Zahlen. Über Auslesen der entsprechenden CPU-Flags kann dennoch garantiert werden, dass ausschließlich exakte Vereinfachungen durchgeführt werden.

Wert	repräsentiert
0	Σ^+
1	komplexe Fließkommazahl[10]
2	Λ
3	V
4	Funktionsanwendung in einem Literal
5	Funktionsanwendung in einem Muster
6	Hilfstyp, aktuell genutzt in der Transformation eines Musters mit Wert-Mustervariablen
7	bindende Instanz einer Mustervariable $\mathbf{x} \in X$ an die im Muster Bedingungen geknüpft sind
8	bindende Instanz einer Mustervariable $\mathbf{x} \in X$ ohne weitere Bedingungen
9	gebundene Instanz einer Mustervariable $\mathbf{x} \in X$
10	X^*
11	W

Abbildung 5.5.1: Werte von `NodeType`

Argumente ohne Indirektion direkt in einer Funktionsanwendung gespeichert. Knoten dieser Art sind Typ 0, 3, 6, 8 und 9. Für Symbole aus Σ^+ wird nicht der Name selbst gespeichert, sondern der Index in einem zentralen Namensspeicher[10]. Der Zugriff auf den Namensspeicher wird so auf die Eingabe und Ausgabe eines Terms beschränkt.

Alle im `Store` mögliche Belegungen eines Feldes sind der `union` `TermNode` aus Quelltext 1 zusammengefasst. Der Typ `Complex` ist dabei nur ein Alias für `std::complex<double>`. Interessant ist der Typ `FApp`, welcher die Funktionsanwendung repräsentiert. Intern ist dieser Typ erneut polymorph, da das erste Arrayfeld einer Funktionsanwendung nicht nur die ersten `NodeIndex` Referenzen auf Funktion und Argumente enthält, sondern auch die Information, wie viele Arrayfelder von der Funktionsanwendung besetzt sind und wie viele Argumente insgesamt gehalten werden. Folgende Felder im `Store` halten ausschließlich weitere Argumente.

[10] Nicht optimal ist damit, dass die Ordnung zweier Symbole aus Σ^+ zueinander von der Reihenfolge der Auflistung im Namensspeicher und damit von der Eingabereihenfolge abhängen, was aber nur ästhetische Nachteile birgt.

```
1  union TermNode
2  {
3      //nodes valid everywhere:
4      Complex complex;
5      FApp f_app;
6      Lambda lambda;
7
8      //nodes only expected in a pattern:
9      RestrictedSingleMatch single_match;
10     MultiMatch multi_match; //only expected in right hand side
11     ValueMatch value_match;
12     FAppInfo f_app_info; //metadata for f_app in pattern (directly preceeding)
13 };
```

Quelltext 1: mögliche Einträge eines Feldes im Speicher

Beispiel 5.3. Das Literal `tup(f(a, b)(c), 1, 2, 3)` wird in der folgenden Debugausgabe genutzt, um den Speicheraufbau zu veranschaulichen.

```
head at index: 5
 0 | application:     { 121,  92,  93 }    f(a, b)
 1 | application:     {   0,  94 }         f(a, b)(c)
 2 | value      :                          1
 3 | value      :                          2
 4 | value      :                          3
 5 | application:     {  78,   1,   2,     tup(f(a, b)(c), 1, 2, 3)
 6 |   ...           3,   4 }
```

Die linke Spalte gibt den Arrayindex an. Nach der Art des gespeicherten Knotens folgt dann für Funktionsanwendungen die Auflistung der Funktion und Argumente. Ganz rechts ist der Teilbaum beginnend an dem entsprechenden Arrayindex angegeben. Die Symbole `f`, `tup`, `a`, `b` und `c` sind nicht als Elemente im Array des `Store` vertreten. Die Funktionsanwendung von `f` gespeichert an Index 0 verrät, dass das Programm vor dem Parsen von Symbol `f` bereits 121 andere unbekannte Symbole eingelesen hat. Analog ist `a` als Index 92, `b` als Index 93 und `c` als Index 94 gespeichert. Interessant ist weiter, dass die Funktionsanwendung des Symbols `tup` beginnend an Arrayindex 5 auch das darauffolgende Arrayelement belegt. Bis auf den Index von `tup`, sind die restlichen Indices dieser Funktionsanwendung Arraypositionen.

Beispiel 5.4. Die in Abschnitt 5.3 genutzte Notation für Lambdas mit namenlosen Lambdaparametern wird auch in der Umsetzung zur Darstellung von Lambdafunktionen genutzt. Als Beispiel wird die Speichernutzung des Literals `(\f x .f(x, x))(sum, 3)` gezeigt.

```
head at index: 3
 0 | application:     {   0,   1,   1 }    %0(%1, %1)
 1 | lambda      :                          {\[2].%0(%1, %1)}
 2 | value       :                          3
 3 | application:     {   1,  81,   2 }    {\[2].%0(%1, %1)}(sum, 3)
```

Nach Auswertung von *normalize* ändert sich die Speicherbelegung.

```
head at index: 0
 0 | value       :                          6
 1 | -----free slot-----
 2 | -----free slot-----
 3 | -----free slot-----
 4 | -----free slot-----
```

Zuerst wurden die Argumente `sum` und `3` in eine Kopie der Lambdadefinition eingesetzt[11], dann wurde die Funktionsanwendung von `sum` zur komplexen Zahl `6` ausgewertet. Diese kann in einem einzelnen Arrayelement gespeichert werden, der restliche Platz ist jetzt ungenutzt.

5.6 Algorithmen

Die beschriebene Datenstruktur wird in den meisten Fällen rekursiv abgelaufen. Der einzelne `NodeIndex` bietet aber nicht genug Information, um zu wissen, wo der referenzierte Knoten gespeichert ist. Das Array des `Store` muss adressierbar sein. Für diesen Zweck gibt es Hilfstypen, die eine Referenz auf den `Store` bzw. das unterliegende Array mit dem `NodeIndex` bündeln. Die mutierbare Variante wird als `MutRef` bezeichnet. Soll nur Lesen des Terms erlaubt sein, gibt es die Hilfstypen `Ref` und `UnsaveRef`.
Illustriert wird die Vorgehensweise des Ablaufens des Terms als Baum mit polymorphen Knoten anhand der Funktion `free_tree`, dargestellt als Quelltext 2, welche im `Store` die von dem übergebenden Teilbaum `ref` belegten Arrayfelder freigibt. Diese Freigabe ist aber nur möglich, wenn in Zeile 3 festgestellt wird, dass zu dem Term auch Arrayelemente im `Store` gehören und dass diese von nirgendwo sonst referenziert werden. Die Felder `ref.type` und `ref.index` von `MutRef` sind als Daten von `NodeIndex` zu erkennen. Die Methode `ref.at(`x`)` konstruiert eine Referenz zu einem neuen Teilterm x, übergeben als `NodeIndex`. Die Werte, die die einzelnen `case`-Statements differenzieren, sind die Zahlen aus Tabelle 5.5.1, jedoch zur Kompilierzeit berechnet aus korrespondierenden `enum`-Werten.

[11] Deswegen sind in der zweiten Version fünf, statt der ursprünglichen vier Slots zu sehen.

```
1  void free_tree(const MutRef ref)
2  {
3      if (!is_stored_node(ref.type) || ref.store->decr_at(ref.index) != 0) {
4          return;
5      }
6      switch (ref.type) {
7      case NodeType(Literal::lambda):
8          free_tree(ref.at(ref->lambda.definition));
9          break;
10     case NodeType(Literal::f_app):
11         for (const NodeIndex subtree : ref) {
12             free_tree(ref.at(subtree));
13         }
14         FApp::free(*ref.store, ref.index);
15         return;
16     case NodeType(SingleMatch::restricted_):
17         free_tree(ref.at(ref->single_match.condition));
18         break;
19     case NodeType(SpecialMatch::value):
20         free_tree(ref.at(ref->value_match.inverse));
21         break;
22     }
23     ref.store->free_one(ref.index);
24 } //free_tree
```

Quelltext 2: Speicherfreigabe eines Teilterms

5.6.1 Eingebaute Auswertung und Normalisierung

Die „natürliche“ Auswertung der Funktionsanwendungen eingebauter Funktionssymbole, etwa `sum`, `prod`, `sin`, etc. zu komplexen Zahlen ist in der Umsetzung fest eingebaut und ausschließlich als Teil des ebenfalls fest eingebauten *normalize* Algorithmus vorhanden. Dieser unterscheidet sich insofern von Kapitel 3, als dass alle beschriebenen Normalisierungsschritte in der nicht rekursiven Funktion `normalize::outermost` implementiert sind, welche ausschließlich eine außenliegende Funktionsanwendung ändert. Das erlaubt, in der Ersetzung eines Teilterms durch ein Muster, ausschließlich die neuen Teile des Literals auf mögliche Normalisierungen zu prüfen.

Erst die Funktion `normalize::recursive`, abgebildet als Quelltext 3, implementiert die Auswertungsstrategie für den Gesamtterm: Zuerst wird in Zeile 6 das Funktionssymbol berechnet, was in der aktuellen Funktionsanwendung angewendet wird. Möglich ist, dass ein Funktionssymbol faul ist[12], also vor seinen Argumenten normalisiert wird. Sonst erfolgt in Zeile 13 der Rekursionsaufruf für jedes Argument, bevor zum Schluss in Zeile 17 der gesamte Term normalisiert wird.

[12] Das ist aktuell ausschließlich für `true` und `false` als binäre Funktionen der Fall: `true` bildet auf das erste Argument ab, `false` auf das zweite.

```
NodeIndex normalize::recursive(const MutRef ref, const Options options)
{
    assert(ref.type != PatternFApp{});
    if (ref.type == Literal::f_app) {
        auto iter = begin(ref);
        *iter = normalize::recursive(ref.at(*iter), options);
        if (nv::is_lazy(*iter)) [[unlikely]] {
            const NodeIndex result = normalize::outermost(ref, options).res;
            return normalize::recursive(ref.at(result), options);
        }
        else {
            for (++iter; !iter.at_end(); ++iter) {
                *iter = normalize::recursive(ref.at(*iter), options);
            }
        }
    }
    return normalize::outermost(ref, options).res;
} //recursive
```

Quelltext 3: rekursive Normalisierung eines Literals

```
bool find_match(const UnsaveRef pn_ref, const UnsaveRef ref, State& match_state);

bool rematch(const UnsaveRef pn_ref, const UnsaveRef ref, State& match_state);

bool find_identic(const UnsaveRef pn_ref, const UnsaveRef hay_ref, State&
↪ match_state, const bool find_rematch);

bool find_permutation(const UnsaveRef pn_ref, const UnsaveRef hay_ref, State&
↪ match_state, const bool find_rematch);

bool find_dilation(const UnsaveRef pn_ref, const UnsaveRef hay_ref, State&
↪ match_state, const bool find_rematch);
```

Quelltext 4: Funktionsdelarationen der Algorithmen 6, 7, 8, 9 und 10

5.6.2 Matchalgorithmen

Die große Lücke bei der Diskussion des Matchalgorithmus in Kapitel 4 ist die Datenstruktur, die nicht nur das finale Match enthalten muss, sondern auch, welches Musterargument gerade mit welchem Literal gematcht ist - und das für jede Funktionsanwendung des Musters. Diese Datenstruktur wird als C++ Referenz[13] mit dem Namen `match_state` übergeben, wie in Quelltext 4 gezeigt.
Der Typ `State` enthält primär drei Arrays. Im Array `.single_vars` sind die Referenzen der Teilterme des Literals, mit dem die bindenden Instanzen der normalen Mustervariablen aktuell assoziiert sind, als `NodeIndex` gespeichert. Im Array `.value_vars` ist für jede Wert-Mustervariable der aktuell assoziierte Wert gespeichert. Diese beiden Arrays implementieren damit fast exakt das Match v_p. Die Indices der Mustervariablen werden

[13] äquivalent zu einem Zeiger, nur bereits dereferenziert

in den Arrays auf die entsprechend assoziierten Literale abgebildet. Nicht ausschließlich als Teil v_p zu verstehen ist der Teil von `match::State`, der die Matchkonfiguration jeder Funktionsanwendung des Musters mit kommutativem Funktionssymbol oder mit Multi-Mustervariablen unter den Argumenten festhält. Das Array `.f_app_entries` hält diese Konfigurationen mit einem Element für jede darauf zurückgreifende Funktionsanwendung. Ein solches Element besteht selbst aus einem Array mit einem Eintrag für jedes Argument der Funktionsanwendung. Wenn Musterargument p_i aktuell mit Argument t_k des Literals gematcht ist, steht in diesem Array an Index i der Eintrag k. Eine sekundäre Funktion besitzt `.f_app_entries` für Multi-Mustervariablen. Die Argumente des Literals, die einer entsprechenden Mustervariablen zugeordnet werden, können hieraus rekonstruiert werden, da die Multi-Mustervariablen genau die Stellen besetzen, die mit keinem normalen Argument gematcht sind. Das erlaubt der Implementierung, die Multi-Mustervariablen auf der linken Seite eines Musters so zu nutzen, wie im Pseudocode angedeutet, also als einfache Logikwerte in einem Array bei jeder Funktionsanwendung.

5.7 Anwendung

5.7.1 Ableiten

Wenn auch Ableitungsregeln nicht die Möglichkeiten des assoziativen und kommutativen Matches ausnutzen, bieten sie doch eine gute erste Übersicht der Nutzung eines Termersetzungssystems.

```
diff(_x, _x)                      = 1
diff(_a, _x) | !contains(_a, _x) = 0
diff(_g^_h, _x) = (diff(_h, _x) ln(_g) + _h diff(_g, _x) / _g) _g^_h
diff(_a, _x) | _a :sum = map(sum, \f .diff(f, _x), _a)
diff(_u vs..., _x) = diff(_u, _x) vs... + _u diff(prod(vs...), _x)
diff(_f(_y), _x) = diff(_y, _x) fdiff(_f)(_y)

fdiff(\x ._y) = \x .diff(_y, x)
fdiff(sin)     = cos
fdiff(cos)     = \x .-sin(x)
fdiff(exp)     = exp
fdiff(ln)      = \x .x^(-1)
fdiff(tan)     = \x .cos(x)^(-2)
...
```

Die linken Seiten sind entweder Funktionsanwendungen des binären Funktionssymbols `diff` oder des unären Funktionssymbols `fdiff`. Während `diff` einen beliebigen Term als erstes Argument erwartet, gefolgt von dem Symbol, nach dem abgeleitet werden soll, leitet `fdiff` unäre Funktionen ab. Das vereinfacht die Syntax zur Darstellung der Kettenregel, welche als letzte Regel für `diff` vor der Leerzeile umgesetzt ist. Interessant ist

bei dem entsprechenden Muster `diff(_f(_y), _x)`, dass die Funktion, deren Anwendung abgeleitet wird, repräsentiert durch die Mustervariable `_f`, selbst eine Unbekannte ist. Die rechte Seite `diff(_y, _x) fdiff(_f)(_y)` implementiert dann das „innere mal äußere Ableitung“-Muster. Da `fdiff(_f)` eine Funktion zurückgibt, kann auf diese Funktion das Argument `_y` angewendet werden.
Im Folgenden wird die rechte Seite der vierten Regel diskutiert.

```
diff(_a, _x) | _a :sum = map(sum, \f .diff(f, _x), _a)
```

Das Funktionssymbol `map` hat eine fest in *normalize* eingebaute Auswertung: Sollte das dritte Argument eine Funktionsanwendung des ersten Argumentes g sein, wird wieder eine Funktionsanwendung von g zurückgegeben, allerdings mit dem zweiten Argument von `map` angewendet auf jedes Argument von g. In Ersetzungsregeln geschrieben sähe die Abbildungsvorschrift von `map` beispielsweise wie folgt aus.

```
map(_g, _f, _g(xs...))
    = map_helper(_g(), _g(xs...), _f)

map_helper(_g(xs...), _g(_y, ys...), _f)
    = map_helper(_g(xs..., _f(_y)), _g(ys...), _f)

map_helper(_g_app, _g(), _f)
    = _g_app
```

In der konkreten Anwendung im Muster wird also die Ableitung einer Summe umgeschrieben als die Summe der Ableitungen der Argumente. Das Muster

```
diff(_u + vs..., _x) = diff(_u, _x) + diff(sum(vs...), _x)
```

hätte in wiederholter Anwendung denselben Effekt. Die genutzte Version benötigt allerdings nur eine einzige Ersetzung[14] für eine Summe mit beliebig vielen Argumenten.

Beispiel 5.5. Angewendet auf das Literal `diff(sin(pi t) + 3, t)` erzeugen die Ableitungsregeln folgende Zwischenergebnisse bzw. folgende Normalform (Leerzeichen sind zur verbesserten Lesbarkeit ergänzt).

```
diff(3, t) + diff(sin(pi t), t)
0          + diff(sin(pi t), t)
             diff(pi t, t)                    fdiff(sin)(pi t)
             (pi diff(t, t) + t diff(pi, t)) fdiff(sin)(pi t)
             (pi diff(t, t) + t diff(pi, t))        cos(pi t)
             (pi 1          + t diff(pi, t))        cos(pi t)
             (pi            + t 0          )        cos(pi t)

pi cos(pi t)
```

[14] sofern `map` in *normalize* ausgewertet wird

5.7.2 Exakte Auswertung trigonometrischer Funktionen

Die Regeln zur Erkennung von Nullstellen und Extrempunkte der Kosinus-Funktion lassen sich sehr kompakt mit Wert-Mustervariablen beschreiben:

```
cos(           pi)           = -1
cos(($k + 0.5) pi) | $k :int =  0
cos((2 $k)     pi) | $k :int =  1
cos((2 $k + 1) pi) | $k :int = -1
```

Eine alternative Regelmenge mit dem selbem Effekt nutzt Bedingungen.

```
cos(   pi)                      = -1
cos(_a pi) | _a + 1/2     :int =  0
cos(_a pi) | _a / 2       :int =  1
cos(_a pi) | (_a - 1) / 2 :int = -1
```

Bemerkenswert ist, dass in beiden Regelmengen dem System kein numerischer Wert für das Symbol `pi` genannt wird.

5.7.3 Faktorisieren

Schwieriger in der Frage, ob jede Regelanwendung zu einer Vereinfachung führt, ist das Faktorisieren einer Summe aus Produkten. Wer dieser Frage unkritisch gegenübersteht, kann die folgenden Ersetzungsregeln nutzen.

```
_a^2 + 2 _a _b + _b^2 = (_a + _b)^2
_a^2 - 2 _a _b + _b^2 = (_a - _b)^2
$a^2 + 2 $a _b + _b^2 = ($a + _b)^2
$a^2 - 2 $a _b + _b^2 = ($a - _b)^2

_a bs... + _a cs... = _a (prod(bs...) + prod(cs...))
_a bs... + _a       = _a (prod(bs...) + 1)
_a       + _a       = 2 _a
```

Leider nicht direkt durch eine endliche Menge von Mustern abbildbar ist der binomische Lehrsatz. Die ersten beiden binomischen Formeln sind durch die vier Regeln im ersten Block vollständig beschrieben. Die unteren beiden Regeln des Blocks erkennen die Formeln auch dann, wenn `$a` nicht direkt vorliegt, sondern als Wert bereits mit seiner Umgebung kombiniert wurde. Anwendbar ist die dritte Regel etwa auf das Literal `81 + 18 sin(x) + sin(x)^2` mit v_p `$a` $= 9$.
Die dargestellten Regeln müssen deshalb kritisch betrachtet werden, weil sie hochgradig nicht-konfluent sind, wie bereits in Abschnitt 4.5.1 diskutiert. Eine einfache Ersetzung des ersten gefundenen Musters führt deshalb möglicherweise nicht immer zum bestmöglichen Ergebnis.

6 Fazit

Terme können als Baumstrukturen verstanden werden. Die Transformation eines Terms wird dementsprechend als Transformation eines Baumes umgesetzt. In dieser Arbeit erfolgt die Transformation über Ersetzungsregeln, die selbst als Paare von Bäumen dargestellt werden; die Bäume werden dabei als Muster bezeichnet. Mustervariablen fungieren als Platzhalter in einer Ersetzungsregel.
Es ist schwierig, effiziente Algorithmen zu finden, die eine große Teilmenge aller Muster mit assoziativen und kommutativen Funktionssymbolen erkennen. Die allgemeine Lösung bietet sich nur für wenige Probleme an, da kein deterministischer Algorithmus mit polynomieller Laufzeit bekannt ist. Die beschriebenen Algorithmen nutzen Backtracking, um unterschiedliche Matchmöglichkeiten der Teilmuster zu testen.
Als teilweise Alternative zur Berücksichtigung von Assoziativität bei der Matchsuche bieten sich Konstrukte an, die für mehrere Argumente einer Funktionsanwendung gleichzeitig stehen können, hier als Multi-Mustervariablen bezeichnet. Einen vollständigen Ersatz bieten sie jedoch nicht. Bestimmte kommutative Muster können in linearer Zeit in einem Literal erkannt werden, auch wenn gleiche Mustervariablen mehrfach auftreten.
Die Zielsetzung, Ausdrücke über den komplexen Zahlen zu vereinfachen, hat sich als sehr ambitioniert herausgestellt. Das implementierte Termersetzungssystem erlaubt die einfache und schnelle Anpassung einer Normalisierungsstrategie. Vor allem aber die häufig fehlende Konfluenz der gewählten Regelmengen machen es schwierig, die Vereinfachung exakt zu kontrollieren. Die Implementierung von Wert-Mustervariablen erlaubt in einigen Fällen, Muster zu definieren, die sehr nah an gewohnten mathematischen Schreibweisen liegen. In ihren Möglichkeiten sind sie gegenüber dem implementierten Bedingungssystem jedoch stärker eingeschränkt. In beiden Fällen können aber fast ausschließlich Bedingungen an die Wertebereiche komplexer Zahlen gestellt werden.

6.1 Ausblick

Drei Gebiete sind für den Autor besonders interessant. Zum einen sind die Möglichkeiten, bestimmte Muster schneller zu finden, noch nicht ausgeschöpft. Weiter wird die fehlende Konfluenz der genutzten Regelmengen in der Umsetzung noch wenig berücksichtigt. Denkbar wäre für die Vereinfachung recht kleiner Terme etwa die gleichzeitige Anwendung aller möglichen Ersetzungsregeln mit anschließender Auswahl der besten Normalform. Um in diesem Fall die Größe des Literals bzw. der Literale und damit auch den Ersetzungsaufwand kontrollieren zu können, müssten möglichst viele gleiche Abschnitte von mehreren Versionen des Literals geteilt werden.
In dieser Arbeit nicht besprochen ist die Kompilierung von Mustern zu entsprechenden musterspezifischen Matchalgorithmen. Die Schwierigkeit steigt hierbei allerdings mit der Ausdruckskraft der Muster.

Danksagung

Ich bedanke mich bei allen, die mich bei der Fertigstellung der Arbeit unterstützt haben. Am wichtigsten war für mich die Betonung meines Betreuers Dr. Prashant Batra, keine Doktorarbeit anfertigen zu müssen. Für Druck und Bindung danke ich Robert Reiter. Vielen Dank auch an die Korrekturleser Carl Egge, Josias Martens, Max Neuendorf, Jannik Rank Christoph Ueker, Jann Warnecke, und meine Eltern. Für das Catering bedanke ich mich bei Laura Beck und Paul Borchardt.

Glossar

Ahne Vater des aktuellen Terms oder Ahne des Vaters des aktuellen Terms 7

Argument Koordinate des Definitionsbereiches einer Funktion 6

Auswertung Funktionswert einer Interpretation 7

Ersetzungsregel Paar von Mustern. Das linke Muster soll in einem Literal durch das rechte Muster ersetzt werden 9

Funktionsanwendung Art von Term: Tupel $(f, t_0, \ldots, t_{n-1})$ aus Funktionssymbol $f \in F$ und Argumenten $t_0, \ldots, t_{n-1} \in T$. die Stelligkeit von f beträgt n oder f ist variadisch. 6

Funktionssymbol repräsentiert Funktion in einer Funktionsanwendung, Element in F 6

Interpretation Funktion, die Termen oder Teilen von Termen eine Bedeutung zuweist 7

Kind Argument der aktuellen Funktionsanwendung 7

konfluent keine zwei linke Seiten unterschiedlicher Regeln können das selbe Literal matchen, wenn die rechten Seiten ungleich sind 31

Konstantensymbol (nicht-rekursiver) Term, Element in C 6

Konstruktor Interpretation u_c eines Funktionssymbols als Tag des Argumenttupels 8

Lambda auch Lambdafunktion oder Lambdaabstraktion, Funktion mit lokal definierter Abbildungsvorschrift 33

Lambdaparameter Parameter einer Lambdafunktion, Element in V 34

Literal zu transformierender Term oder Teil davon 9

Match Interpretation $v_p \colon X \to T$ von Mustervariablen als Literale mit $lit(p, v_p) = t$ für Muster p und Literal t 10

Matchalgorithmus Verfahren um ein Match zu finden 10

Multi-Mustervariable spezielle Mustervariable, die mehrere Argumente einer Funktionsanwendung matchen kann 19

Muster Term in $M(F, C)$ als Teil einer Ersetzungsregel 9

Musterinterpretation Interpretation $lit \colon (X \to T) \times M \to T$ von Mustern als Literale via Match v_p 10

Mustervariable Konstantensymbol in X mit Platzhalterrolle in einem Muster 9

Nachkomme Kind des aktuellen Terms oder Kind eines Nachkommen des aktuellen Terms 7

Normalform Element in Bild von *normalize* oder Literal, auf das keine Ersetzungsregel mehr angewendet werden kann 30

Parameter Platzhalter für eine Koordinate des Definitionsbereiches einer Funktion 8

Stelligkeitsfunktion $arity\colon F \to \mathbb{N} \cup \omega$ bildet ein Funktionssymbol f auf die erwartete Anzahl seiner Argmente ab 6

Term Baumstruktur mit Funktionsanwendungen als innere Knoten und Konstantensymbolen als Blätter 6

transparent Lambdafunktion hat möglicherweise nicht-eigene Lambdaparameter in Abbildungsvorschrift 38

variadisch Ist das Funktionssymbol f variadisch, kann es auf beliebig viele Argumente angewendet werden. Schreibe dann $arity\ f = \omega$. 6

Vater Funktionsanwendung, die den aktuellen Term als Argument enthält 7

Wert-Mustervariable spezielle Mustervariable, die erlaubt Funktionsanwendungen als Muster mit Konstantensymbolen zu matchen 34

Literatur

[19] *GHC/Using rules - HaskellWiki*. Apr. 2019. URL: https://wiki.haskell.org/GHC/Using_rules (besucht am 08. 07. 2021).

[21a] *Constructor - HaskellWiki*. Juni 2021. URL: http://wiki.haskell.org/Constructor#Data_constructor (besucht am 11. 05. 2021).

[21b] *n+k patterns - Haskell*. Juli 2021. URL: https://sites.google.com/site/haskell/notes/nkpatterns (besucht am 10. 07. 2021).

[21c] *Symbolic Calculations—Wolfram Language Documentation*. Aug. 2021. URL: https://reference.wolfram.com/language/tutorial/SymbolicCalculations.html (besucht am 08. 07. 2021).

[21d] *Symbolic Computations in MATLAB - MATLAB & Simulink - MathWorks Deutschland*. Aug. 2021. URL: https://de.mathworks.com/help/symbolic/symbolic-computations-in-matlab.html (besucht am 08. 07. 2021).

[21e] *Symbolics - Maple Help*. Aug. 2021. URL: https://de.maplesoft.com/support/help/maple/view.aspx?path=updates/Maple10/symbolic#simplify (besucht am 08. 07. 2021).

[BKN87] Dan Benanav, Deepak Kapur und Paliath Narendran. „Complexity of matching problems“. In: *Journal of Symbolic Computation* 3.1 (1987), S. 203–216. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(87)80027-5.

[BN98a] Franz Baader und Tobias Nipkow. „Confluence“. In: *Term Rewriting and All That*. Cambridge University Press, 1998, S. 134–157. DOI: 10.1017/CBO9781139172752.007.

[BN98b] Franz Baader und Tobias Nipkow. „Motivating Examples“. In: *Term Rewriting and All That*. Cambridge University Press, 1998, S. 1–6. DOI: 10.1017/CBO9781139172752.002.

[Bor21] Bruno Borchardt. *TermV2*. Juli 2021. URL: https://github.com/brunizzl/TermV2 (besucht am 08. 07. 2021).

[Chu36] Alonzo Church. „An Unsolvable Problem of Elementary Number Theory“. In: *American Journal of Mathematics* 58.2 (1936), S. 345–363. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2371045.

[CM 82] M.J. O'Donnell C.M. Hoffmann. „Pattern Matching in Trees“. In: *Journal of the ACM (JACM)* (1982).

[de 72] N.G de Bruijn. „Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem“. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), S. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[Eke95] S. M. Eker. „Associative-commutative matching via bipartite graph matching“. In: *The Computer Journal* 38.5 (Jan. 1995), S. 381–399. ISSN: 0010-4620. DOI: 10.1093/comjnl/38.5.381. eprint: https://academic.oup.com/comjnl/article-pdf/38/5/381/1235222/380381.pdf. URL: https://doi.org/10.1093/comjnl/38.5.381.

[HO82] Christoph M Hoffmann und Michael J O'Donnell. „Programming with equations“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.1 (1982), S. 83–112.

[Hud89] Paul Hudak. „Conception, Evolution, and Application of Functional Programming Languages“. In: *ACM Comput. Surv.* 21.3 (Sep. 1989), S. 382–386. ISSN: 0360-0300. DOI: 10.1145/72551.72554. URL: https://doi.org/10.1145/72551.72554.

[Jon87] Simon Peyton Jones. „13.2.1 Supercombinators“. In: *The Implementation of Functional Programming Languages*. Prentice Hall, Jan. 1987, S. 223–228. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages.

[Jon96] Simon L. Peyton Jones. „Compiling Haskell by program transformation: A report from the trenches“. In: *Programming Languages and Systems — ESOP '96*. Hrsg. von Hanne Riis Nielson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, S. 18–44. ISBN: 978-3-540-49942-8.

[KL91] E. Kounalis und D. Lugiez. „Compilation of pattern matching with associative-commutative functions“. In: *TAPSOFT '91*. Hrsg. von S. Abramsky und T. S. E. Maibaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, S. 57–73. ISBN: 978-3-540-46563-8.

[Lau03] Niels Lauritzen. „Gröbner bases“. In: *Concrete Abstract Algebra: From Numbers to Gröbner Bases*. Cambridge University Press, 2003, S. 186–222. DOI: 10.1017/CBO9780511804229.006.

[McC60] John McCarthy. „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1“. In: *Communications of the ACM* (1960).

[ODo77] M.J. O'Donnell. *Computing in Systems Described by Equations*. Springer-Verlag Berlin Heidelberg, 1977.