Entwicklung eines Termersetzungssystems für assoziative und kommutative Ausdrücke zum vereinfachen arithmetischer Terme
Version 0.0.1

Bruno Borchardt

heute

Inhaltsverzeichnis

1 Einleitung

Hier kommt hin, was halt in eine Einleitung soll:

- was ist ein Termersetzungssystem?
- Einsatz von Termersetzungssystemen
  - Beweisprüfer /Beweisassistenten
  - Arithmetikausdrücke vereinfachen
  - Optimierender Compiler
  - bestimmt noch mehr
- was soll die in dieser Arbeit umgesetzte Variante (gut) können?

1.1 Definitionen

Eine Menge von Termen T ist in dieser Arbeit immer in Abhängigkeit der Mengen F und Σ definiert.

$$T(F, \Sigma) \coloneqq \Sigma \ \cup \ \{(f, t_1, \ldots, t_n) \mid f \in F, \ t_1, \ldots, t_n \in T(F, \Sigma)\}$$

Elemente $f \in F$ werden *Funktionssymbole* genannt, Elemente $s \in \Sigma$ werden *Konstantensymbole* genannt. Wichtig ist, dass vorerst nicht zwischen einem Konstantensymbol $s \in \Sigma$ unterschieden wird, welches bekannte Eigenschaften hat, etwa der Komplexen Zahl $s = -5i$ und einem Symbol, wessen Eigenschaften vom Kontext abhängen können, etwa dem Zeichen $s = \mathrm{x}$ (vorrausgesetzt $-5i, \ \mathrm{x} \in \Sigma$).
$(f, t_1, \ldots, t_n) \in T \setminus \Sigma$ wird auch als $f(t_1, \ldots, t_n)$ geschrieben und *Funktionsanwendung* von f auf die *Parameter* $t_1, \ldots, t_n$ genannt. $t_1, \ldots, t_n$ sind zudem die *Kinder* ihres Vaters f. Kinder sind allgemeiner *Nachkommen*. 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.
Ein *Teilterm* ist Nachkomme des äußersten Funktionssymbols oder der gesamte Term.
Häufig werden 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.
$f(t_1, \ldots, t_k, a, t_{k+2}, \ldots, t_n)$ kann also äquivalent $f(ts..., a, rs...)$ geschrieben werden, mit $(t_1, \ldots, t_k) = (ts...)$ und $(t_{k+2}, \ldots, t_n) = (rs...)$.

1.1.1 Beispiel

Wählt man $F = \{\mathrm{f}, \mathrm{g}\}$ und $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, ist $t = \mathrm{f}(\mathrm{b}, \mathrm{g}(\mathrm{f}(\mathrm{a}), \mathrm{b})) \in T(F, \Sigma)$ ein Term, dargestellt in Abb. 1.1.1. b ist ein Kind von g und ein Ahne von f. Damit ist g Vater von b. g(...) ist zudem Kind von f. b und f(a) sind Teilterme von t.

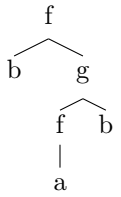

Abbildung 1: Baumdarstellung des Terms f(b, g(f(a), b))

1.1.2 Stelligkeit von Funktionssymbolen

Der Beispielterm aus Abb. 1.1.1 hat zwei Funktionsanwendungen von f, jedoch einmal mit den zwei Parametern b und einer Anwendung von g und das zweite Mal mit nur einem Parameter a. Die Funktionssymbole, für die eine Funktionsanwendung mit beliebig vielen Parametern sinvoll ist, nennen wir *variadisch*. Ergibt eine Funktionsanwendung für ein Funktionssymbol nur mit einer festen Anzahl von Parametern Sinn, wird im folgenden von einer *festen Stelligkeit* gesprochen. Um ausschließlich Terme zu erlauben, die in dieser Hinsicht sinvoll sind, wird jedem Funktionssymbol $f \in F$ von der Funktion $\text{arity}\colon F \to \mathbb{N} \cup \omega$ eine Stelligkeit zugewiesen. Ist $\text{arity}(f) = \omega$, so ist f ein variadisches Funktionssymbol. Im folgenden werden ausschließlich Terme betrachtet, die Funktionsanwendungen mit der passenden Anzahl von Parametern enthalten.

$$T(F, \Sigma) \coloneqq \Sigma \ \cup \ \{(f, ts...) \mid f \in F,\ \text{arity}(f) \in \{n, \omega\},\ ts... \in T(F, \Sigma)\}$$

1.1.3 Funktionsauswertung

Ein Funktionssymbol ist in der allgemeinen Definition eines Terms noch keine Funktion, da die Abbildungsvorschrift, sowie Definitionsmenge und Bildmenge nicht definiert sind. Eine Funktionsanwendung kann allerdings als Datenstruktur gesehen werden, etwa die des Funktionssymbols *pair* auf die zwei Parameter a und b als ein Paar: $pair(a, b)$.
Die erste Erweiterung der Funktionssymbolmenge zur Menge von Funktionen folgt mittels der eval Funktion.

$$\tilde{T}(F, \Sigma) \coloneqq \{(f, zs...) \mid f \in F,\ \text{arity}(f) \in \{n, \omega\},\ zs... \in \Sigma\}$$

$$\text{eval}\colon (\tilde{T} \to \Sigma) \times T \to \Sigma$$

$$\text{eval}(v, t) = \begin{cases} t & ,\ t \in \Sigma \\ v(s_1, \ldots, s_n) & ,\ t = (f, t_1, \ldots, t_n),\ s_i = \text{eval}(v, t_i) \end{cases}$$

Als Beispiel ist $\Sigma = \mathbb{N}_0$, $F = \{sum, product, power\}$ und v definiert als

$$v(f, s_1, \ldots, s_n) = \begin{cases} s_1 + s_2 & ,\ f = sum \\ s_1 \cdot s_2 & ,\ f = product \\ (s_1)^{s_2} & ,\ f = power \end{cases}$$

Der Term $t = sum(3, product(2, 4))$ kann dann ausgewertet werden zu

$$\begin{aligned}
\text{eval}(v, t) &= \text{eval}(v, sum(3, product(2, 4))) \\
&= v(sum, \text{eval}(v, 3), \text{eval}(v, product(2, 4))) \\
&= v(sum, 3, v(product, \text{eval}(v, 2), \text{eval}(v, 4))) \\
&= v(sum, 3, v(product, 2, 4)) \\
&= v(sum, 3, 8) \\
&= 11
\end{aligned} \tag{1}$$

Nicht mehr ausreichend ist eval, wenn auch Symbole unbekannten Wertes in Σ aufgenommen werden oder wenn manche Funktionen nicht von bzw. nach Σ abbilden.

2 Erste Normalform

In diesem Abschnitt werden erste Termumformungen beschrieben, die isolierte Eingenschaften einzelner Funktionen ausnutzen. Ziel ist es kommutative und assoziative Funktionsanwendungen eindeutig darzustellen.

2.1 Assoziative Funktionsanwendungen

Die geschachtelte Anwendung einer assoziativen Funktion führt je nach Klammersetzung zu verschiedenen mathematisch equivalenten Termen. Als Beispiel dient hier die Addition, dargestellt als Anwendung des Funktionssymbols $+$. Die folgenden Ausdrücke sind paarweise verschiedene Terme, jedoch alle mathematisch äquivalent.

$$\begin{aligned}
+(+(+(a, b), c), d) &= +(+(a, +(b, c)), d) \\
&= +(+(a, b), +(c, d)) \\
&= +(a, +(b, +(c, d))) \\
&= \dots
\end{aligned} \tag{2}$$

Das ist in den meisten Situationen kein Problem, da in Infixnotation Klammern nur zur Gruppierung notwendig sind und per Definition der Assoziativität in diesem Fall keinen Unterschied machen, also weggelassen werden können: $a + b + c + d$. Es gibt mehrere Optionen eine solche Schachtelung in einem Term zu normalisieren, also in eine eindeutige Form zu bringen. Die erste ist, festzulegen, dass höchstens einer der beiden Parameter der binären assoziativen Funktion wieder Ergebnis dieser Funktion sein darf. Wählt man den zweiten Parameter dafür aus, ist wird die Summe in der Normalform dargestellt als $+(a, +(b, +(c, d)))$. Diese Methode nennt sich Pivotisierung und wird in **Quellen** näher untersucht.

Alternativ kann man die Summe von zwei Parametern auch als Spezialfall einer Summe von $n \in \mathbb{N}$ Parametern auffassen, dann gewohnt geschrieben als $\Sigma_{x \in \{a,b,c,d\}} x$. Dieser Weg wird im folgenden gewählt, wobei die Darstellung als Term dann $+(a, b, c, d)$ ist.
Die Normalisierung assoziativer Funktionsanwendungen des Funktionssymbols f ersetzt dann alle Teilterme der Form $f(as..., f(bs...), cs...)$ zu $f(as..., bs..., cs...)$.

2.2 Kommutative Funktionsanwendungen

Eine Normalform für kommutative Funktionsanwendungen erfordert eine totale Ordnung auf der Menge aller Terme $T(F, \Sigma)$. Aufbauend auf einer totalen Ordnung von Σ und einer totalen Ordnung von F, kann eine lexikographische Ordnung wie folgt definiert werden.

- sind $s, \tilde{s} \in T$ Konstantensymbole, so ist die Ordnung identisch zu der Ordnung in Σ
- sind $s, a, \in T$ und s ein Konstantensymbol und a eine Funktionsanwendung ist $s < a$
- sind $a = f(ts...), b = g(rs...) \in T$ Funktionsanwendungen und ist $f \neq g$ gilt $a < b \iff f < g$
- sind $a = f(t_1, \ldots, t_n), b = f(r_1, \ldots, r_m) \in T$ Funktionsanwendungen, und $\tilde{t_1}, \ldots, \tilde{t_n}$, $\tilde{r_1}, \ldots, \tilde{r_m}$ die normalisierten Parameter von a und b, ist die Ordnung wiefolgt
  - wenn $\exists k \leq \min(n, m) \colon \forall i < k \ \tilde{t_i} = \tilde{r_i}, \ \tilde{t_k} \neq \tilde{r_k}$ sind a und b so geordnet wie $\tilde{t_k}$ und $\tilde{r_k}$
  - gilt $n < m$ und $\forall i < n \colon \tilde{t_i} = \tilde{r_i}$ ist $a < b$
  - gilt $n = m$ und $\forall i \leq n \colon \tilde{t_i} = \tilde{r_i}$ ist $a = b$

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

2.3 Teilweise Auswertung

Mit der Darstellung assoziativer Funktionen als variadische Funktionen, ist es möglich, dass eine Funktion mittels eval teilweise ausgewertet werden kann, also gilt für assoziative Funktionssymbole $f \in F$

$$f(a, b) = c \implies f(xs..., a, b, ys...) = f(xs..., c, ys...)$$

Ist f zudem kommutativ gilt

$$f(a, b) = c \implies f(xs..., a, ys..., b, zs...) = f(xs..., c, ys..., zs...)$$

Eine normalisierte Funktionsanwendung ist damit so weit wie möglich ausgewertet. Sollte sie ganz ausgewertet werden können, ist die normalisierte Funktionsanwendung das Ergebnis der Auswertung.

3 Datenstruktur

Hier wird erläutert, wie meine konkrete Implementierung Terme speichert und verwaltet. Vielleicht werden auch ein paar grundlegene Algorithmen auf der Datenstruktur erklärt, sofern ich mein exaktes Vorgehen in keinem Paper wiederfinde.
Denkbare Algorithmen zum erklären:

- Falten eines Terms
- Sortieren eines Terms / ordnen von Teiltermen
- exaktes Ausrechen von Teilbäumen ohne Variablen (und warum ich das mit dem Zusammenfassen von geschachtelten Summen und weiteren Sachen in einen Algorithmus zusammenwursten musste)
- Frage: aktuell parse ich einen String zum bauen eines Terms noch in $O(n^2)$. Das ist warscheinlich nicht optimal. Ergo würde ich diesen Teil wenn, dann eher am Rande erwähnen. Ist es für die Arbeit wichtig, das noch zu optimieren (unter der Vorraussetzung, dass die von mir gewählte Grammatik kontextfrei ist, was sie aber glaube ich ist)?

Mögliche Tangente: Meine Idee (und Umsetzung) einer Art Aufzählung (in C++ und Co. as enum in der Sprache enthalten), die Hirarchien erlaubt und damit Basis eines (wie ich finde) relativ eleganten Typsystems für die einzelnen Arten von Termknoten darstellt. (Tangente der Tangente: Das native C++ Typsystem mit Umsetzung von Polymorphie als Vererbung erschwert einem nicht nur das Speichern eines polymorphen Baums in einem Array, sondern verteilt auch den Code jedes Algorithmus über die verschiedenen Klassen, die die einzelnen Baumknotenarten modellieren. Deswegen habe ich mir was eingendes gebastelt)

Sowohl der zu vereinfachende Term, als auch die Vereinfachungsregeln liegen im Programm als Baumstruktur vor. Die Knoten des Baumes haben jeweils einen der folgenden Typen:

1. Komlexe Zahl
   Jeder bekannte Wert wird intern als komplexe Zahl mit Realteil und Imaginärteil als Fließkommazahl doppelter Präzesion aus der C++ Standardbibliothek gespeichert.

2. Variable
   Eine Variable besteht nur aus ihrem Name, welcher als Array von ASCII Buchstaben vorliegt.

3. Bekannte Funktion
   Im Speicher des einzelnen Terms ist eine dem Programm bekannte Funktion als Array von Referenzen auf ihre Parameter gespeichert. Auf der Datenebene ist dabei nicht direkt erkennbar, ob die Funktion eine feste Anzahl von Parametern hat, wie etwa die Exponentialfunktion, oder variadisch ist, wie zum Beispiel die Summe. Dies erlaubt es, oft nicht zwischen beiden Arten differenzieren zu müssen.

4. Unbekannte Funktion
   Funktionen, deren Name dem Programm nicht bekannt sind, speichern diesen pro Instanz, analog zur Variable. Vor dem Namensarray befindet sich ein Parameterarray, identisch zu dem der bekannten Funktion. Damit muss in vielen Fällen auch nicht zwischen bekannten Funktionen und unbekannten Funktionen unterschieden werden.

5. Baum-Mustervariable
   Während die normale Variable vor allem im zu vereinfachenden Term erwartet wird, ist die Baum-Mustervariable das Symbol, welches in einem Muster für einen Teilbaum des zu vereinfachenden Terms steht.

6. Wert-Mustervariable
   Ähnlich wie die Baum-Mustervariable steht auch diese representativ für einen Teil des zu vereinfachenden Baums, hier allerdings nicht für einen beliebigen Teilbaum, sondern einen bekannten Zahlenwert. Die Separierung von der Baum- Mustervariable ist notwendig, um mehr Arten von Werten erkennen zu können, ohne extra ein System für das Überprüfen von Logikbedingungen einbauen zu müssen (wenn auch dieses mächtiger wäre).

4 Patternmatching

Hier stelle ich vor, was für Matchingalgorithmen (für Bäume mit Knoten beliebigen Grades) ich in der Literatur finde. (Hoffentlich finde ich nicht den von mir genutzten, aber den als erster entdeckt zu haben halte ich für unwarscheinlich.) Den von mir genutzen Algorithmus würde ich hier im Detail erklären und die Laufzeit analysieren. Spoiler: ich meine gelesen zu haben, dass das matchen von Mustern mit möglichen Mehrfachnennungen von Mustervariablen in variadischen Ausdrücken NP-schwer ist, aber vielleicht kann ich für praxisrelevante Teilmengen aller Musterterme bessere Worst-Case Laufzeiten erzielen. (Das optimieren meines Algorithmus ist noch im Gange).

einfacher Matchingalgorithmus

5 Vereinfachen von Arithmetischen Termen

Ich habe begonnen das Termersetzungssystem zu entwickeln, um Arithmetische Ausdrücke zu vereinfachen (etwa $a + 2a \to 3a$). Wie genau ich das umsetze, wird in diesem Abschnitt erläutert.
Während die Datenstruktur und der Matchingalgorithmus schon benutzbar sind, ist dieser Teil von mir bisher so gut wie gar nicht implementiert worden. Der grobe Plan ist aber folgender:

1. Funktionen höherer Ordnung anwenden:
   - ableiten (Prototyp dafür steht schon)
   - vielleicht integrieren (soll für den allgemeinen Fall wohl schwer sein)
   - vielleicht fouriertransformieren
   - vielleicht laplacetransformieren
   - ganz ganz ganz ganz vielleicht Differentialgleichungen lösen
2. Normalform herstellen:
   - alles ausmultiplizieren ($a \cdot (b + c) \to a \cdot b + a \cdot c$)
   - Vorzeichen aus ungeraden Funktionen herausziehen ($\sin(-x) \to -\sin(x)$)
   - Vorzeichen in geraden Funktionen auf plus setzen ($\cos(-x) \to \cos(x)$)
   - Überlegen, wie man das selbe für Fälle mit Summen im Argument definiert ($\cos(a - b)$ vs. $\cos(b - a)$)
   - bekannte Faktoren aus Potenz ziehen ($(3x)^2 \to 9x^2$)
   - ...
3. Vereinfachen:

- manche Transformationen sollten immer angewendet werden (etwa $\sin^2(x) + \cos^2(x) \to 1$)
- andere Transformationen nur ausprobieren und mit einer passenden Metrik gucken, wie gut ein Term nach Anwendung noch weiter vereinfacht werden kann (etwa, wenn man aus verschiedenen Optionen des Ausklammerns wählen kann)
- vielleicht Linearfaktorzerlegung von Polynomen (schätze ich für den allgemeinen Fall schwierig ein, solange ich nur exakte Operationen zulasse)
- vielleicht Polynomdivision (schätze ich genau so schwierig ein, zumindest wenn ich nicht vorher schon Linearfaktoren habe)
- ...

Anmerkung 1: Die Normalform ist notwendig, um zu garantieren, dass mehrfaches Auftreten eines Teilbaums / Teilterms auch erkannt wird.
Anmerkung 2: es kann sein, dass ich manche Eigenschaften der Normalformauch während des Vereinfachungsschrittes immer wieder wiederherstellen muss.

5.1 Vergleich meiner Features mit anderen Computeralgebrasystemen

Ich bin ja nicht der erste, der auf die Idee kommt, Terme zusammenzufassen. Wolphram Alpha und Maple sind zwar nicht Open Source, aber andere Optionen, wie etwa SymPy aus der Python Standardbibliothek soweit ich weiß schon. Da lässt sich bestimmt ein bisschen vergleichen, wie andere Leute die selben Probleme lösen.

6 Zusammenfassung

Was halt in eine Zusammenfassung kommt