Fachbereich Informatik und Mathematik
Institut für Informatik

Masterarbeit

Kompression von Matrizen als Multi-Terminal Decision Diagrams

Miso Starcevic

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

12. April 2016

Erklärung gemäß der Master-Ordnung Informatik von 2008 § 24 Abs. 12

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

Frankfurt am Main, den 12.04.2016

Miso Starcevic

Inhaltsverzeichnis

Kapitel 1

Motivation

In der Informatik beschäftigt man sich unter anderem mit (digitalen) Bildern und Graphen. Beide Objekte lassen sich auch als Matrizen auffassen: digitale Bilder als Matrizen über Farbwerte und Graphen als Adjazenzmatrizen.

Nun können auf den oben genannten Objekten auch Operationen ausgeführt werden wie das Invertieren von digitalen Bildern, also die Veränderung einer Farbe in die „gegenteilige Farbe des jeweiligen Farbspektrums“ (zitiert aus [wik]), oder der einfache Vergleich zwischen zwei Graphen, deren Laufzeiten natürlich von der Größe der jeweiligen Eingaben abhängen. Dabei scheint es auf den ersten Blick durchaus sinnvoll zu sein, die Objekte (digitale Bilder und Graphen) zu „verkleinern“, also zu komprimieren, um somit bessere Laufzeiten für Operationen zu erzielen. Im Allgemeinen ist das Grundkonzept beim Komprimieren das Finden von sich wiederholenden Strukturen oder Elementen in einem Objekt. Deshalb ist auch klar, dass sich nur Matrizen „gut“ komprimieren lassen, wenn sie aus vielen gleichen „Teilmatrizen“ aufgebaut sind.

Die meisten Kompressionsverfahren auf Matrizen arbeiten mit dünnbesetzten Matrizen (siehe [BBC+] und [EM03]. Das sind Matrizen, deren Einträge größtenteils aus Nullen bestehen. Zwar können mit diesen Kompressionsverfahren hohe Kompressionsraten erzielt werden, viele Matrixoperationen sind aber nicht effizient umsetzbar.

Es existieren auch Kompressionsverfahren auf vollbesetzten Matrizen, also Matrizen, wo die meisten Einträge nicht Null sind, wie die JPEG-Komprimierung für digitale Bilder. Diese ist jedoch nicht verlustfrei, dass heißt beim Dekomprimieren kann es durchaus vorkommen, dass die Ausgangsmatrix nicht wieder rekonstruiert werden kann, was für ein allgemeines Kompressionsverfahren für Matrizen nicht akzeptabel ist.

In [LS14] wird ein Ansatz verfolgt, mit dem hohe Kompressionsraten erzielt werden können und viele Matrixoperationen effizient ausführbar sind. Das Grundkonzept ist die Darstellung von Matrizen als *multi-terminal decision diagrams*, kurz *MTDDs*, die eine kontextfreie Grammatik repräsentieren. Es wird in [LS14] gezeigt, dass mit einer „kleinen“ Erweiterung von MTDDs Matrixoperationen, wie die Matrixmultiplikation sowie auch der Vergleich von zwei Matrizen, effizient ausführbar sind.

Diese Arbeit wird sich mit einem konkreten Kompressionsverfahren für MTDDs beschäftigen, einer Implementierung jenes in der funktionalen Programmiersprache Haskell präsentieren und durch konkrete Performance-Tests die „Tauglichkeit“ des Kompressionsverfahrens in der Praxis untersuchen.

Kapitel 2

Grundlagen

2.1 Funktionale Programmiersprachen

Programmiersprachen bzw. Programmierparadigmen lassen sich generell in zwei Klassen zusammenfassen: *imperative* und *deklarative* Programmiersprachen.

In imperativen Programmiersprachen besteht ein Programm aus Folgen von Anweisungen, die nacheinander abgearbeitet werden und dabei angeben *wie* ein Problem zu lösen ist, indem Variablen, also der Zustand des Rechners (Speicher), verändert werden. Programme in deklarativen Progamiersprachen hingegen beschreiben eher *was* berechnet werden soll und bedienen sich dabei meist des Auswertens von Ausdrücken.

Funktionale Programmiersprachen zählen zu den deklarativen Programmiersprachen, deren Programme – wie schon die Bezeichnung erahnen lässt – eine Menge von Funktionsdefinitionen darstellen, die beim Ausführen der Auswertung von Funktionen gleicht und am Ende als Resultat einen Wert liefern. Wird von einer funktionalen Programmiersprache dabei stets gewährleistet, dass die Anwendung einer gleichen Funktion auf gleiche Argumente immer das gleiche Ergebnis liefern, also das Prinzip der *referentiellen Transparenz* gilt, dann ist diese *rein*.

Diese Eigenschaft geht einher mit der Abwesenheit von Seiteneffekten (der Speicher wird nicht sichtbar manipuliert), die den schönen Vorteil mit sich bringt, dass beim Testen und Debuggen von Code, sowie bei dessen Korrektsbeweis keine (Rechner-)Zustände in Betracht gezogen werden müssen. Außerdem können Funktionen unabhängig voneinander getestet werden, sofern die referentielle Transparenz gilt.

Neben diesen Vorteilen bietet das funktionale Programmierparadigma eine andere Sichtweise auf Probleme, weil man eher das Problem beschreibt und versucht, es in einfachere Teilprobleme aufzuteilen, um diese und damit das Gesamtproblem zu lösen. Dies führt oft zu eleganteren Lösungen, da die Programme durch das „Beschreiben“ des Problems „mathematischer“ sind. Programmiersprachen, die (größtenteils) auf dem funktionalen Programmierparadigma beruhen sind neben Haskell F#, ML, Lisp (und viele mehr).

Gerade wegen einiger der obengenannten Vorteile des funktionalen Programmierparadigmas findet sich auch in vielen imperativen Programmiersprachen eben jenes Paradigma wieder: so bietet beispielsweise die Programmiersprache Python (siehe [pyt]) eine Möglichkeit, um funktional programmieren zu können.

Es benutzen auch weltbekannte Unternehmen wie Facebook (siehe [Mar15]) die funktionale Programmiersprache Haskell zur Spamerkennung in von Benutzern verfassten Beiträgen.

2.2 Haskell

Es wird in diesem Abschnitt ein Überlick über die wichtigsten (Sprach-)Elemente von Haskell gegeben, die auch in der Implementierung des Programms Verwendung fanden.

Haskell ist eine reine, nicht strikte funktionale Programmiersprache mit statischem polymorphem Typsystem und ist durch den „Haskell 2010 Report“ definiert (siehe [Mar10]).

Jedes Haskell-Programm besteht im Wesentlichen aus mindestens einer Funktion, die an sich einen Ausdruck darstellt. Dieser wird solange durch Einsetzen von (Funktions-)Definitionen umgeformt, bis er nicht weiter reduziert werden kann. Am Ende steht dann ein Wert da (oder aber wieder eine Funktion).

Anhand eines Beispiels werden nun einige Grundkonzepte vorgestellt:

```
add :: Int -> Int -> Int
add x y = x + y
```

Man sieht hier eine Definition der Funktion `add`, die zwei Zahlen erwartet und die Summe aus beiden Zahlen als Ergebnis zurückliefert. Die rechte Seite einer Funktionsdefinition nennt man auch *Funktionsrumpf*.

In der ersten Zeile wird der Typ von `add` deklariert: beide Argumente und das Resultat sind Ganzzahlen, also vom Typ `Int`. Da in Haskell der Typ eines Ausdrucks schon zur Compilezeit bekannt sein muss, ist Haskell *statisch* typisiert. Eine *Typdeklaration* wie oben ist dabei nur dann notwendig, wenn der Compiler den Typen eines Ausdrucks nicht eindeutig aus den Definitionen schließen kann.

Da die Addition nicht nur auf Ganzzahlen definiert ist, kann eine Typdeklaration auch allgemeiner gefasst werden:

```
add :: (Num a) => a -> a
add x y = x + y
```

Hier erwartet die Funktion `add` nun Zahlen, die Instanzen der *Typklasse* `Num` sind. Eine Typklasse fasst Typen mit selben Eigenschaften zusammen und besteht im Großen und Ganzen aus Funktionen, die gleich für eine Klasse von Typen definiert sind. So ist die Funktion `(+)`, also die Addition, für die Typklasse `Num` definiert, sodass jede Instanz von `Num` auch die Addition definiert.

Da `add` nun „allgemein“ Zahlen erwartet, spricht man an dieser Stelle von *Typpolymorphismus*: statt Ganzzahlen könnte `add` nun auch Gleitkommazahlen vom Typ `Float` addieren.

Es folgt ein Beispiel für einen Aufruf von `add`:

```
add (1+2) (1+2)
```

Die Funktion würde dann wie folgt ausgewertet:

```
add (1+2) (1+2) -> (1+2) + (1+2)
```

```
-> 3 + 3
-> 6
```

Der Compiler würde beim Einsetzen der Argumente in die rechte Seite der Funktionsdefinition erkennen, dass zwei Mal der gleiche Ausdruck ausgewertet werden müsste. Deshalb markiert der Compiler gleiche Ausdrücke, wertet nur einen aus und ersetzt die markierten durch diesen einen. In diesem Zusammenhang spricht man von der *verzögerten Auswertgung* (call-by-need), oder auch *nicht strikter* Auswertung.

Statt einen Bezeichner für eine Funktion anzugeben (wie oben mit `add`), können Funktionen auch *anonym* definiert werden:

```
\x y -> x + y
```

Diese anonyme Funktion ist eine *Lambda-Abstraktion* (siehe [SS15, Abschnitt 2.1]) und weil sie keinen Bezeichner hat, ist sie nur im aktuellen Kontext ansprechbar (man kann zwar der eben definierten Funktion einen Bezeichner verleihen, es ist jedoch nicht zwingend erforderlich). Ein Aufruf würde so aussehen:

```
(\x y -> x + y) (1+2) (1+2)
```

wobei die Auswertung die Gleiche wäre wie im obigen Beispiel der Funktion `add`.

2.2.1 Listen

Die wohl meist genutzte Datenstruktur in Haskell ist die *Liste* (siehe [Mar10, Abschnitt 3.7]), die rekursiv aufgebaut ist und deshalb dem funktionalen Programmierparadigma entspricht. Eine Liste ist demanch immer als Kopf einer Liste (ein einzelnes Element) und einer Restliste aufgebaut und bedient sich dabei des Listenkonstruktors `(:)` und der leeren Liste `[ ]`.

Eine Liste von Zahlen von 1 bis 4 sieht dementsprechend so aus:

```
(1:(2:(3:(4: [ ]))))
```

oder in verkürzter Schreibweise

```
[1,2,3,4]
```

2.2.2 Pattern Matching

Ein in Haskell häufig verwendetes Mittel – um zum einen die Auswertung des Funktionsrumpfes abhängig von den Werten der übergebenen Argumente zu machen und zum anderen bestimmte Daten aus den Argumenten zu filtern – ist das *Pattern Matching* (siehe [Mar10, Abschnitt 3.17]). Wie sich schon aus dem Namen schließen lässt, werden dabei Muster definiert, sodass die übergebenen Argumente gegen diese „gematcht“

werden können. Folgendes Beispiel soll dies verdeutlichen:

```
map :: (a -> b) -> [a] -> [b]
map _ [ ] = [ ]
map f (x:xs) = f x : map f xs
```

Die aus der Standard-Bibliothek von Haskell bekannte Funktion `map` wendet eine Funktion `f` auf alle Elemente einer Liste an und liefert als Ergebnis die manipulierte Liste zurück. Hierbei sieht man in der Defintion von `map` mehrere Pattern: in Zeile 2 steht die Wildcard `_` (siehe [Mar10, Abschnitt 3.17.1]) als erstes Argument von `map`, die dem Compiler sagt, dass dieses Argument für den Funktionsrumpf keinerlei Bedeutung hat und deshalb nicht weiter berücksichtigt wird. Das zweite Pattern ist die leere Liste `[ ]`. In Zeile 3 findet man das Pattern `(x:xs)`, das die übergebene Liste in den Kopfteil und den verbliebenen Restteil „aufspaltet", sodass dann im Funktionsrumpf auf beide Teile getrennt zugegriffen werden kann.

2.2.3 Weitere Kontrollstrukturen

Neben dem Kontrollieren des Programmflusses durch Pattern Matching gibt es noch weitere Möglichkeiten, um den Programmablauf zu beeinflussen:

case-Ausdruck: der `case`-Ausdruck (siehe [Mar10, Abschnitt 3.13]) bedient sich des Pattern Matching. Dabei werden auch Muster definiert, gegen die ein Argument, das ein Ausdruck sein kann, gematcht wird.

So kann eine Funktion `f` definiert werden, die als Argument eine Zahl entgegennimmt und den bool'schen Wert *wahr* (in Haskell repräsentiert durch `True`) zurückliefert, falls das Argument eine 0 war, dann den Wert *falsch* (in Haskell `False`) zurückgibt:

```
f x = case x of
  0 -> True
  _ -> False
```

Hierbei ist `x` das Argument und die Muster sind `0 -> True` und `_ -> False`.

if-then-else-Konstrukt: wie auch in den meisten Programmiersprachen gibt es auch in Haskell die *if*-Bedingung (siehe [Mar10, Abschnitt 3.6]): es wird ein Ausdruck definiert, der zu einem bool'schen Wert ausgewertet wird. Ist dieser Wert *wahr*, so wird der `then`-Zweig weiter ausgewertet, ist der Wert *falsch*, wird der Ausdruck im `else`-Zweig ausgewertet.

Die gleiche Funktion `f`, die als Beispiel für einen `case`-Ausdruck diente, kann mit dem *if-then-else*-Konstrukt so definiert werden:

```
f x =
  if x == 0
  then True
```

```
  else False
```

Es ist zu beachten, dass in Haskell zwangsläufig ein `else`-Zweig definiert werden muss.

guards: ein *guard* (siehe [Mar10, Abschnitt 3.13]) definiert eine Bedingung, bei dessen Erfüllung ein Funktionsrumpf ausgewertet wird, der jedem *guard* folgt. Der Funktionsrumpf des *guards*, dessen Bedingung zuerst erfüllt ist, wird auch zuerst ausgewertet.

Es folgt wieder ein Beispiel der bekannten Funktion `f`, nun mit *guards*:

```
f x
  | x == 0 = True
  | otherwise = False
```

Das Schlüsselwort `otherwise` ist ein Synonym für den Wert `True` in Haskell, dessen Angabe optional ist.

2.2.4 let und where

Um Funktionen auch innerhalb eines Funktionsrumpfes zu definieren und somit nur „lokal“ ansprechbar zu machen, bietet Haskell zum einen das `let...in`-Konstrukt, zum anderen das Schlüsselwort `where` an.

Beim `let...in`-Konstrukt (siehe [Mar10, Abschnitt 3.12]) werden innerhalb des `let`-Blocks, also des Bereichs zwischen dem `let` und des dazugehörigen `in`'s, Funktionen definiert, die innerhalb des `let`-Blocks und des Bereichs nach dem `in` ansprechbar sind. Verschachtelungen von `let...in`-Konstrukt sind möglich.

Im Gegensatz dazu befindet sich der Geltungbereich von Funktionen, die durch das Schlüsselwort `where` eingeleitet werden, im Funktionsrumpf.

Eine Funktion `g` beispielsweise, die als Argument eine Zahl erwartet und als Ergebnis 5 auf diese Zahl addiert, kann mit `let...in` so definiert werden:

```
g x = let
  y = 5
 in x + y
```

mit `where` dementsprechend

```
g x = x + y where y = 5
```

2.2.5 Eigene Datentypen

Neben den schon standardmäßig vorhandenen (primitiven) Datentypen wie zum Beispiel `Int` und `Char` können auch eigene Datentypen durch das Schlüsselwort `data` (siehe [Mar10, Abschnitt 4.2]) definiert werden.

Um einen Binärbaum zu beschreiben, der nur Markierungen an den Blättern hat, kann in Haskell folgender Datentyp definiert werden:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

Dabei sind `Node` und `Leaf` *Datenkonstruktoren*, also gerade die Elemente, aus denen der Binärbaum besteht. Der Typ ist durch `Tree a` gegeben, wobei `a` eine *Typvariable* ist, was in diesem Kontext bedeutet, dass die Blätter des Binärbaums mit Daten vom Typ `a` (wie `Int`, `Char` usw.) markiert sind. Somit ist der Typ *polymorph*. Außerdem sieht man an diesem Beispiel einen rekursiven Datentypen (wie die Liste in Haskell), da an jedem `Node` (also Knoten) zwei Teilbäume „hängen“, die natürlich auch wieder für sich alleine genommen Binärbäume darstellen.

Es existiert noch die Möglichkeit, *Typsynonyme* für vorhandene Datentypen zu definieren, mit dem Zweck die Lesbarkeit des Codes zu erhöhen. So ist standardmäßig in Haskell der Typ für Zeichenkeiten (String) als ein Typsynonym für eine Liste von Zeichen definiert:

```
type String = [Char]
```

2.2.6 Ein- und Ausgabe

Aufgrund der Tatsache, dass Haskell eine reine funktionale Programmiersprache ist und somit an sich keine Seiteneffekte erlaubt, bedient sich Haskell der *IO-Monade* (siehe [SS15, Abschnitt 5.1]).

Eine Monade ist in Haskell eine Typklasse, die „etwas verpackt und bestimmte Operationen auf dem Datentypen zulässt“ (zitiert aus [SS15, Abschnitt 5.1]).

Die IO-Monade kapselt die Seiteneffekte bei der Ein- und Ausgabe, indem Ein- und Ausgabeoperationen außerhalb der funktionalen Programmiersprache ausgeführt werden und deshalb in der funktionalen Programmiersprache zunächst *IO-Aktionen* sind.

Eine IO-Aktion ist eine Funktion, die als Eingabe einen Zustand erhält und den veränderten Zustand samt eines Ergebnisses ausgibt. Sie wird in Haskell durch den Typen `IO a` dargestellt. Genaueres zu Monaden kann man in [SS15, Kapitel 5] nachlesen.

Um nun zum Beispiel Daten aus einer Datei zu lesen, kann man dies wie folgt in Haskell implementieren:

```
readData :: IO String
readData = do
  data <- readFile "data.txt"
  return data
```

Die Funktion `readFile` liest Daten aus der Datei „data.txt“ und gibt sie als String zurück, der in dem IO-Typen „verpackt“ ist. Der Operator `<-` „entpackt“ den String aus dem IO-Typen und „weist“ der Variablen `data` den gelesenen String zu. Dieser String könnte nun weiter verarbeitet werden, hier aber wird der String wieder in den IO-Typen durch den Aufruf der Funktion `return` verpackt, da eine Funktion stets einen IO-Typen

zurückliefern muss, sobald irgendwo in der Funktion eine IO-Aktion auftaucht.

2.2.7 Modulsystem

Haskell besitzt ein Modulsystem, dass heißt zum einen, dass Funktionen in einem Modul zusammengefasst und somit exportiert werden können und zum anderen, dass Funktionen aus anderen Modulen importiert werden können.

Eine Datei wird zu einem Modul deklariert, indem das Schlüsselwort `module` (siehe [Mar10, Kapitel 5]) an den Dateianfang mit dem gewünschten Modulnamen und gefolgt von einem `where` gesetzt wird:

```
module MyModule where
```

Nach dem `where` können dann Module importiert, Datentypen und Funktionen definiert werden.

Ein Modulimport erfolgt dabei durch das Schlüsselwort `import` (siehe [Mar10, Abschnitt 5.3]), gefolgt vom Names des importierten Moduls. Um Namenskonflikte zu vermeiden, sollte man die Funktionen des importierten Moduls stets mit dem vollen qualifizerten Namen verwenden. Da diese Namen recht lang werden können, kann man den importierten Modulen (lokale) *Aliase* mit dem Schlüsselwort `as` (siehe [Mar10, Abschnitt 5.3.3]) vergeben.

Ein Beispiel zu einem Modulimport:

```
import HisModule as HM
```

Funktionen aus dem Modul `HisModule` können nun mit `HM.`*Funktion*, statt mit `HisModule.`*Funktion* angesprochen werden.

2.3 Matrizen

Als Eingabegröße für das Kompressionsverfahren, dass später vorgestellt wird, wird man sich in diesem Abschnitt näher mit Matrizen befassen. Die hier stehenden Definitionen basieren zum größten Teil auf [Sti16].

Eine *Matrix* kann man sich als eine rechteckige Tabelle vorstellen, die (meistens) Zahlen als Einträge enthalten. Sie wird damit selbst zum mathematischen Objekt, auf die dann Operationen wie die Addition oder Multiplikation definiert werden können. So lassen sich beispielweise digitale Bilder durch Matrizen beschreiben, indem jeder Bildpunkt (also Pixel) einem Eintrag in einer Matrix entspricht.

Formal besteht eine $n \times m$-Matrix A aus Einträgen bzw. Elementen $a_{ij} \in K$ mit $i = 1,..., n$, $j = 1,..., m$ und K ein *Körper* (siehe [Sti16, Abschnitt 3.4]) – im Kontext dieser Arbeit ist K meist die Menge der ganzen Zahlen (ohne Multiplikation) – welche in einem rechteckigen Schema angeordnet werden:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

Dabei bezeichnet n die Anzahl der *Zeilen* und m die Anzahl der *Spalten*.

2.3.1 Matrixaddition

Seien A und B beide $n \times m$-Matrizen und Einträgen aus dem Körper K. Dann ist die Matrixaddition von A und B, kurz $A + B$, definiert durch

$$A+B = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & \cdots & a_{nm}+b_{mn} \end{pmatrix},$$

also $a_{ij} + b_{ij}$, f.a $i = 1, ..., n$ und $j = 1, ..., m$.

Beispiel:
Seien $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ und $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$. Dann ist

$$A + B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}.$$

2.3.2 Matrixsubtraktion

Seien A und B beide $n \times m$-Matrizen und Einträgen aus dem Körper K. Dann ist die Matrixsubtraktion von A und B, kurz $A - B$, definiert durch $a_{ij} - b_{ij}$, f.a $i = 1, ..., n$ und $j = 1, .., m$.

Statt also die Einträge zu addieren wie bei der Matrixaddition, werden diese bei der Matrixsubtraktion subtrahiert.

2.3.3 Skalarmultiplikation von Matrizen

Sei A eine $n \times m$-Matrix mit Einträgen aus dem Körper K und $\lambda \in K$. Dann ist die Skalarmultiplikation von einem Skalar λ und einer Matrix A, kurz $\lambda \cdot A$, gegeben durch

$$\lambda \cdot A = \lambda \cdot \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} \lambda \cdot a_{11} & \cdots & \lambda \cdot a_{1n} \\ \vdots & \ddots & \vdots \\ \lambda \cdot a_{m1} & \cdots & \lambda \cdot a_{mn} \end{pmatrix},$$

also $\lambda \cdot a_{ij}$, f.a $i = 1, ..., n$ und $j = 1, ..., m$.

Beispiel:
Seien $\lambda = 4$ und $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. Dann ist

$$\lambda \cdot A = 4 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 4 \cdot 1 & 4 \cdot 2 \\ 4 \cdot 3 & 4 \cdot 4 \end{pmatrix} = \begin{pmatrix} 4 & 8 \\ 12 & 16 \end{pmatrix}.$$

Definition 2.1 *Seien A_1, A_2, A_3, A_4 $n \times m$-Matrizen und A eine $2n \times 2m$-Matrix. Dann ist A die* ***Konkatenation*** *von A_1, A_2, A_3, A_4, kurz $A =$ **concat**(A_1, A_2, A_3, A_4) wenn für Einträge a in A und a^k in A_k, mit $1 \leq k \leq 4$ gilt:*

1. *für $1 \leq i \leq n$ und $1 \leq j \leq m$ ist $a_{ij} = a^1_{ij}$*
2. *für $1 \leq i \leq n$, $1 \leq j \leq m$ und $m+1 \leq j' \leq 2m$ ist $a_{ij'} = a^2_{ij}$*
3. *für $1 \leq i \leq n$, $1 \leq j \leq m$ und $n+1 \leq i' \leq 2n$ ist $a_{i'j} = a^3_{ij}$*
4. *für $1 \leq i \leq n$, $1 \leq j \leq m$, $n+1 \leq i' \leq 2n$ und $m+1 \leq j' \leq 2m$ ist $a_{i'j'} = a^4_{ij}$*

Beispiel:
Seien $A_1 = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}$, $A_2 = \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}$, $A_3 = \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix}$, $A_4 = \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix}$. Dann ist

$$concat(A_1, A_2, A_3, A_4) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}.$$

In der kommenden Darstellung wird eine Beobachtung präsentiert, die die Methode liefern wird, mit der dann in Kapitel 3 Matrizen komprimiert werden.

Beobachtung 2.1 *Bei näherer Betrachtung der Definition, sowohl der Matrixaddition als auch der Skalarmultiplikation von Matrizen, kann festgestellt werden, dass die Addition bzw. Skalarmultiplikation von einzelnen Einträgen unabhängig von den anderen Einträgen in der Matrix ist. Dies führt zu folgenden zwei Aussagen:*

Seien A und B $n \times m$-Matrizen mit Einträgen aus K, $A = concat(A_1, A_2, A_3, A_4)$, $B = concat(B_1, B_2, B_3, B_4)$ und $\lambda \in K$. Dann gilt:

1. $A + B = concat(A_1 + B_1, A_2 + B_2, A_3 + B_3, A_4 + B_4)$
2. $\lambda \cdot A = concat(\lambda \cdot A_1, \lambda \cdot A_2, \lambda \cdot A_3, \lambda \cdot A_4)$

Abschließend zum Thema Matrix wird hier noch die Größe von Matrizen definiert, um die Kompressionsrate von Matrizen berechnen zu können.

Definition 2.2 *Sei A eine n x m-Matrix. Dann ist die Größe von A, kurz $|A|$, gegeben durch $|A| = n \cdot m$.*

2.4 Kontextfreie Grammatiken

Da die Kompression von Matrizen in dieser Arbeit eigentlich die Kompression von kontextfreien Grammatiken sein wird, wird in diesem Abschnitt eine kurze Einführung zu kontextfreien Grammatiken gegeben, die sich auf [Sch15] stützt.

Eine *formale Grammatik* ist eine Menge von Regeln, die eine *formale Sprache* beschreiben bzw. erzeugen.

Eine *kontextfreie* Grammatik ist ein spezielle formale Grammatik. Sie erzeugt eine zur Klasse der *formalen* Sprachen zugehörige kontextfreie Sprache (siehe [Sch15, Kapitel 4]).

Definition 2.3 *Sei G eine kontextfreie Grammatik, im Folgenden mit KFG abgekürzt, mit $G = (\Sigma, V, S, P)$. Dabei sind*

- *Σ das Alphabet oder die Menge der **Terminale**,*
- *V die Menge der Variablen oder der **Nichtterminale**,*
- *Σ und V sind disjunkt, also $\Sigma \cap V = \emptyset$,*
- *S das **Startsymbol** und*
- *P die Menge der Produktionsregeln oder einfach **Produktionen***

Die Produktionen von G sind von der Form

$$u \to v \text{ mit } u \in V \text{ und } v \in (V \cup \Sigma)$$

Die linke Seite von $\to$ nennt man *linke Seite* der Produktion, die rechte Seite von $\to$ die *rechte Seite* der Produktion.

Eine KFG K, die die Sprache $L = \{0^n 1^n : n > 0\}$ erzeugt, mit $K = (\Sigma, V, S, P)$ kann folgendermaßen formuliert werden:

- $\Sigma = \{0, 1\}$
- $V = S$
- S ist das Startsymbol und
- $P = \{S \to 0S1, S \to 01\}$

Definition 2.4 *Sei G eine KFG mit $G = (\Sigma, V, S, P)$. Dann ist die **Größe von G** definiert durch*

$$|G| = \sum_{p \in P} \textit{Anzahl der Terminale und Nichtterminale in der Produktion } p$$

Für die obige Grammatik K ist $|K| = 7$.

Die in dieser Arbeit betrachteten KFG's werden die Eigenschaft haben, dass sie genau ein Wort erzeugen, weshalb jedes Nichtterminal genau eine Produktion hat und die Grammatik *azyklisch* ist, also jede Produktion genau einmal beim Erzeugen des Wortes „durchlaufen“ wird.

Kapitel 3

Multi-terminal decision diagrams

In diesem Kapitel werden multi-terminal decision diagrams eingeführt, die dazu dienen, Matrizen komprimiert darzustellen. Anschließend wird dann das Kompressionsverfahren vorgestellt, welches multi-terminal-decision diagrams und somit auch (indirekt) Matrizen komprimieren wird.

Die Idee für multi-terminal-decision diagrams stammt aus [LS14], eine Implementierung jener in Haskell wurde in [RW13] vorgenommen.

3.1 MTDD

Ein *multi-terminal decision diagram*, kurz *MTDD*, ist eine spezielle kontextfreie Grammatik, die ausschließlich eine $2^h \times 2^h$-Matrix, mit $h \in \mathbb{N}_{>0}$, (im besten Fall komprimiert) darstellen kann. Man nennt dabei h die *Höhe* eines MTDDs. Dabei sind MTDDs azyklisch und es existieren keine zwei Produktionen, die die gleiche linke Seite haben. Diese Einschränkungen haben zur Folge, dass nur endliche Matrizen (Zeilen- und Spaltenanzahl sind fest) dargestellt werden können und jede MTDD genau eine Matrix repräsentiert.

Einer der Zwecke von MTDDs ist durch das Komprimieren von Matrizen Rechenoperationen zu sparen, indem diese auf den MTDDs schließlich ausgeführt werden.

Formal lässt sich ein MTDD $M = (\Sigma, N, S, P)$ der Höhe h definieren durch:

- $\Sigma = \mathbb{Z}$
- N = Menge von Nichtterminalen, wobei N unterteilt ist in Teilmengen N_i mit $0 \leq i \leq h$
- S das Startsymbol mit $S \in N_h$ und
- $P = \{ \; A \to \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ mit $A \in N_i$ und $A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2} \in N_{i-1}$ für $1 \leq i \leq h$, und
  $A \to z$ mit $A \in N_0$ und $z \in \Sigma \; \}$

Bezeichnet werden in dieser Arbeit analog zu [RW13] die Produktionen

- $A \to \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ als *DownStep-Produktion*
- $A \to z$ als *Terminalproduktion*

Die von einem Nichtterminal $A \in N_i$ repräsentierte Matrix bezeichnet man mit $val(A)$. Es ist dann eine $2^i \times 2^i$- Matrix, weshalb man auch hier von der *Höhe* i des Nichtterminals A spricht.

Die Regel $A \to \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ stellt gerade die Konkatenation von Matrizen dar,

hier also $val(A) = concat(val(A_{1,1}), val(A_{1,2}), val(A_{2,1}), val(A_{2,2}))$, wobei $A \in N_k$ und $A_{1,1}, A_{1,2}, A_{2,1}$ und $A_{2,2} \in N_{k-1}$ mit $k = 1, ..., h$.

Da Σ sich nie ändert, kann ein MTDD M auch als $M = (N, P, S)$ definiert werden. Im Tupel wurden darüberhinaus P und S getauscht, da in [LS14] MTDDs genauso definiert sind.

Beispiel:

Sei M ein MTDD mit $M = (N, P, S)$, wobei

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1\}$ und $N_0 = \{A_2\}$
- $S = A_0$ und
- $P = \{\ A_0 \to \begin{pmatrix} A_1 & A_1 \\ A_1 & A_1 \end{pmatrix}, A_1 \to \begin{pmatrix} A_2 & A_2 \\ A_2 & A_2 \end{pmatrix}, A_2 \to 1\ \}$

Die Matrix, die durch M repräsentiert wird, ist

$$val(A0) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Dieses Beispiel zeigt auch schon den besten Fall, also eine Matrix, die die höchste Kompressionsrate besitzt, da alle Einträge gleich sind.

Allgemein können $2^h \times 2^h$-Matrizen, bei denen alle Einträge gleich sind, mit einem MTDD M der Höhe h und Größe $|M| = O(h)$, also logarithmisch zur Eingabegröße, dargestellt werden, wie man auch im obigen Beispiel sieht.

Der schlimmste Fall tritt ein, wenn in einer Matrix alle Einträge unterschiedlich sind. Dann hat die dazugehörige MTDD M der Höhe h die Größe $|M| = \sum_{i=0}^{h} 2^i \cdot 2^i = \sum_{i=0}^{h} 4^i = \frac{1 - 4^{h+1}}{-3}$ (Summe einer geometrischen Reihe, siehe [gr]).

3.2 MTDD$_+$

Als Erweiterung von MTDDs wurden in [LS14] MTDD$_+$ (*multi-terminal decision diagrams with addition*) definiert, die eine zustäzliche Produktionsregel erhalten:

Sei M ein MTDD mit $M = (N, P, S)$. Dann ist ein MTDD$_+$ $M^{'} = (N, P', S)$ eine Erweiterung von M derart, dass

$$P^{'} = P \cup \{\ A \to A_1 + A_2 \text{ mit } A, A_1, A_2 \in N_i \text{ für } 1 \leq i \leq h\ \}.$$

Die obige Produktion, analog zu [RW13] als *Additionsproduktion* bezeichnet, bedeutet also, dass die Matrixaddition $val(A_1) + val(A_2)$ nun als Regel abgebildet werden kann, somit können auch Matrizen komprimiert werden, die als MTDD dargestellt kaum bis gar keine Kompression erfahren würden.

Beispiel:

Sei $M^{'}$ ein MTDD mit $M^{'} = (N, P, S)$, wobei

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A_4\}$ und $N_0 = \{T_1, T_2, T_3, T_4, T_{10}, T_{13}, T_{16}, T_{20}\}$
- $S = A_0$ und
- $P = \{$
  $A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$,
  $A_1 \to \begin{pmatrix} T_1 & T_2 \\ T_3 & T_4 \end{pmatrix}$,
  $A_2 \to \begin{pmatrix} T_{10} & T_{13} \\ T_{16} & T_{20} \end{pmatrix}$,
  $A_3 \to A_1 + A_2$,
  $A_4 \to A_1 + A_3$
  $\} \cup \{\ T_i \to i \mid T_i \in N_0 \text{ und } i \in \{1, 2, 3, 4, 10, 13, 16, 20\}\}$

Die Matrix, die durch $M^{'}$ repräsentiert wird, ist

$$val(A_0) = \begin{pmatrix} 1 & 2 & 10 & 13 \\ 3 & 4 & 16 & 20 \\ 11 & 15 & 12 & 17 \\ 19 & 24 & 21 & 28 \end{pmatrix}$$

Da $val(A_0)$ nur unterschiedliche Einträge hat, gebe es mit einem MTDD keine Kompression.

3.3 MTDD$_{(+,\cdot)}$

Damit noch größere Kompressionsraten bei MTDD$_+$ erzielt werden können, hat man jene in dieser Arbeit um eine zusätliche Regel erweitert:

Sei M ein MTDD$_+$ mit $M = (N, P, S)$. Dann ist ein MTDD$_{(+,\cdot)}$ (*multi-terminal decision diagrams with addition and scalar multiplication*) $M' = (N, P', S)$ eine Erweiterung von M derart, dass

$$P' = P \cup \{\ A \to c \cdot A_1 \text{ mit } c \in \mathbb{Z} \text{ und } A, A_1 \in N_i \text{ für } 1 \leq i \leq h\ \}.$$

Die obige Produktion, als *Skalarmultproduktion* bezeichnet, bedeutet also, dass die Skalarmultiplikation von Matrizen $c \cdot val(A_1)$ nun als Regel abgebildet werden kann.

Die Größe dieser Produktion legt man in dieser Arbeit dabei nicht auf 2 (für die beiden Nichtterminale), sondern auf 3 fest, da in der Implementierung später, der Skalar c und alle Nichtterminale den selben Typen (somit gleichen Speicherplatzverbrauch) haben werden.

Beispiel:

Sei M' ein MTDD mit $M' = (N, P, S)$, wobei

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A_4\}$ und $N_0 = \{T_1, T_2, T_3, T_4\}$
- $S = A_0$ und
- $P = \{$

  $A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$,

  $A_1 \to \begin{pmatrix} T_1 & T_2 \\ T_3 & T_4 \end{pmatrix}$,

  $A_2 \to (-1) \cdot A_1$,

  $A_3 \to 5 \cdot A_2$,

  $A_4 \to 5 \cdot A_3$

  $\} \cup \{\ T_i \to i \mid T_i \in N_0 \text{ und } i \in \{1, 2, 3, 4\}\}$

Die Matrix, die durch M' repräsentiert wird, ist

$$val(A_0) = \begin{pmatrix} 1 & 2 & -1 & -2 \\ 3 & 4 & -3 & -4 \\ -5 & -10 & -25 & -50 \\ -15 & -20 & -75 & -100 \end{pmatrix}$$

Man sieht am obigen Beispiel, dass durch das Einführen der Skalarmultproduktion ein weiterer Kompressionsschritt, nämlich $A_2 \to (-1) \cdot A_1$, möglich ist.

Zwar könnte man mit einer MTDD$_+$ Produktionen wie $A_3 \to 5 \cdot A_2$ durch Additionsproduktionen darstellen, beispielweise durch

$$A_3 \to A_5 + A_2$$

$$A_5 \to A_6 + A_6$$

$$A_6 \to A_2$$

doch dieses Ausdrücken von Multiplikationen durch Additionen hat zwei Nachteile:

1. Es können Multiplikationen mit einem negativen Skalar nicht direkt in Additionen überführt werden. Es wäre somit im schlechtesten Fall notwendig, bei einer Produktion wie $A \to c \cdot B$, wobei $c < 0$, eine neue Matrix B' mit $val(B') = c \cdot val(B)$ einzufügen, was die Kompression „zunichte“ machen kann.

2. Zwar kann eine Skalarmultproduktion $A \to c \cdot B$ (mit $c > 0$) durch $O(\log(c))$ viele Additionsproduktionen ausgedrückt werden (die Idee ist, einen Binärbaum über den c vielen Blättern mit Markierung B aufzuspannen, wo in jeder Ebene des Baumes maximal 2 verschiedene Knotenmarkierungen benötigt werden, genaueres wird man in Kapitel 4 sehen), bei hinreichend großem c kann auch hier ein Ersetzen der DownStep-Produktionen zu einer Vergrößerung der Grammatik führen: angenommen der Skalar in der Produktion $A_3 \to 5 \cdot A_2$ wäre 512 statt 5, dann bräuchte man $\log_2(512) = 9$ viele Additionsproduktion und da jede Additionsproduktion 3 Nichtterminale enthält, wären das insgesamt 27 zusätzliche Nichtterminale, womit dann auch hier eine Kompression des MTDD$_{(+,\cdot)}$ M' „zunichte“ gemacht werden würde.

Da ja der Zweck von MTDDs ist, Matrixoperationen durch das indirekte Komprimieren von Matrizen „schneller“ zu machen, wird eine Untersuchung der in [RW13] vorgestellten und implementierten Matrixoperationen auf MTDD$_+$s bei einer Erweiterung um die Skalarmultproduktionen im Anhang dieser Arbeit vorgenommen.

Der nächste Abschnitt wird sich der Kompression von MTDD$_{(+,\cdot)}$s widmen.

3.4 Komprimieren von MTDD$_{(+,\cdot)}$s

Der Ansatz beim Komprimieren ist ein *bottom-up* Anstatz, indem erstmal ein MTDD konstruiert wird und dieses dann beginnend mit den Terminalproduktionen bis zum Startsymbol abgearbeitet wird.

Das hier vorgestellte Kompressionsverfahren wird in die folgenden Schritte unterteilt, die dann in einzelnen Abschnitten genauer erklärt werden:

(1) Erzeuge aus der $2^h \times 2^h$-Matrix A das vollständige MTDD M der Höhe h mit $M = (N, P, S)$, dass heißt, es sind zunächst nur DownStep- und Terminalproduktionen in M enthalten.

(2) Beginnend in der Höhe 0, also mit den Nichtterminalen aus $N_0 \subset N$, sucht man alle Nichtterminale, die die gleiche Matrix produzieren, ersetzt alle gleichen Nichtterminale durch ein Ausgewähltes aus dieser Menge und löscht alle überflüssligen (gleichen) Nichtterminale und deren Produktionen.

(3) Beginnend in der Höhe 1, also mit den Nichterminalen aus $N_1 \subset N$, sucht man alle Nichtterminale, die durch

   1. Additionsproduktionen und/oder
   2. Skalarmultproduktionen

   ersetzt werden können und ersetzt dann die DownStep-Produktionen durch eine der neuen gefunden Produktionen, wobei die Additionsproduktion bevorzugt wird, falls ein Nichtterminal sowohl durch eine Additions- als auch Skalarmultproduktion dargestellt werden kann. Man merkt sich alle gefundenen Additions- und Skalarmultproduktionen der aktuellen Höhe, da diese zur Berechnung eben jener in der nächste Höhe notwendig sind!

(4) Die Schritte (2) und (3) werden nacheinander (von Höhe zu Höhe) solange ausgeführt, bis Höhe h erreicht ist.

(5) Ist Höhe h erreicht, so ist es sinnvoll, M nochmal zu durchlaufen, diesmal von Höhe h bis Höhe 0, um die wegen der Ersetzung durch Additions- bzw. Skalarmultproduktionen nicht mehr gebrauchten Nichtterminale zu löschen.

Das schrittweise Komprimieren findet dadurch statt, dass eben von den gleichen Nichtterminalen bzw. DownStep-Produktionen nur eines erhalten bleiben muss und die anderen deshalb gelöscht werden können. Beim Ersetzen von DownStep-Produktionen durch Additions- bzw. Skalarmultproduktionen spart man sich (mindestens) 2 Nichtterminale, da ja DownStep-Produktionen eine Größe von 5 und Additions- und Skalarmultproduktionen eine Größe von jeweils 3 haben.

Die in den folgenden Abschnitten angegebenen Laufzeiten für die einzelnen Verfahren sollen nur einen groben Anhaltspunkt über die Laufzeiten geben. Deshalb sind die in Kapitel 5 präsentierten Performance-Tests aussagekräftiger.

3.4.1 Vollständiges MTDD

Beim Erzeugen des vollständigen MTDD M der Höhe h mit $M = (N, P, S)$ aus einer $2^h \times 2^h$-Matrix A kann man zwischen zwei Teilen unterscheiden:

1. den Terminalproduktionen, deren rechten Seiten von den Einträgen in A bestimmt werden

2. den DownStep-Produktionen, die nur durch die Höhe h bestimmt werden und somit unabhängig von den konkreten Einträgen in A sind

Man erzeugt demnach zuerst die DownStep-Produktionen, von Höhe h bis Höhe 1, ohne die Einträge der Matrix zu „lesen“. Anschließend werden in M die Terminalproduktionen mit den Einträgen aus A hinzugefügt.

Laufzeit: die Laufzeit für das Erzeugen eines vollständigen MTDD hängt nur von der Höhe h ab. Da sich in jeder Höhe die Anzahl an Produktionen vervierfacht und es hier nur um das Erzeugen von Produktionen geht, ist die Laufzeit gegeben durch die Größe eines vollständigen MTDD: $\Omega(4^h) = \Omega(n^2)$ für $2^h = n$ (die Zeilen- bzw. Spaltenanzahl der zu komprimierenden Matrix). Hierbei wurde eine konstante Laufzeit für das Erzeugen der Produktionen angenommen (es könnte also noch ein log-Faktor hinzukommen). Der genaue Algorithmus ist in Kapitel 4 illustriert.

3.4.2 Finden von gleichen Nichtterminalen

Das Finden von gleichen Nichtterminalen in einem MTDD M der Höhe h mit $M = (N, P, S)$, die die gleichen Matrizen erzeugen, gestaltet sich durch den *bottom-up* Ansatz relativ einfach:

Fall 1: man befindet sich bei den Terminalproduktionen. Es genügt ein Vergleich der Terminale. Dabei merkt man sich alle gleichen Nichtterminale, die dann in der nächsten Höhe durch genau ein Nichtterminal aus dieser Menge von Gleichen ersetzt werden.

Fall 2: man befindet sich bei den DownStep-Produktionen. Für den Vergleich, ob zwei Nichtterminale bzw. deren DownStep-Produktionen die selbe Matrix erzeugen, genügt es sich nur die Nichtterminale in den zwei DownStep-Produktionen anzuschauen: befinden sich in beiden DownStep-Produktionen nur gleiche Nichtterminale und auch an gleicher Stelle, so sind auch beide Produktionen gleich!

Dabei merkt man sich auch wieder alle gleichen Nichtterminale, die dann in der nächsten Höhe durch genau ein Nichtterminal aus dieser Menge von Gleichen ersetzt werden.

Nun macht es nur Sinn in einer Höhe k mit $0 \leq k \leq h$ nach gleichen Nichtterminalen zu suchen, wenn in dieser Höhe auch „ausreichend“ viele Nichtterminale ersetzt werden. Da man mindestens zwei gleiche Nichtterminale dazu finden muss und zu jedem Nichtterminal jeweils eine DownStep-Produktion kommt, müssen also insgesamt mindestens vier Nichtterminale ersetzt werden, damit es überhaupt eine Möglichkeit gibt, zwei gleiche Nichtterminale zu finden. Dieser Fall tritt beispielsweise ein, wenn man die zwei Produktionen

$$A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \text{ und } B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} \text{ hat.}$$

Damit nun A und B gleich sind, müssten zum Beispiel B_1 durch A_1, B_2 durch A_2,

B_3 durch A_3 und B_4 durch A_4 ersetzt, somit also genau vier Ersetzungen gemacht werden.

Laufzeit: die Laufzeit für das Finden von gleichen Nichtterminalen in der Höhe k, mit $0 \leq k \leq h$, hängt nur von der Anzahl der Nichtterminale der Produktionsmenge $P_k := \{\ p : p \text{ erzeugt eine Matrix der Höhe } k\ \}$. Da man ja im schlimmsten Fall alle Kombinationen von $A = B$ betrachtet, wobei A und B Nichtterminale von Produktionen der Produktionsmenge P_k sind, ergibt sich eine Laufzeit von $\Omega(|P_k|^2)$.

Wendet man dieses Verfahren auf alle Produktionsmengen P_k, mit $0 \leq k \leq h$, so ergibt sich die Lauftzeit $\Omega(h \cdot |P_{max}|^2) = \Omega(\log n \cdot n^4)$, wenn die zu komprimierende Matrix I die Größe $2^h \times 2^h = n^2$ hatte und man $P_{max} := P_0$ (in P_0 gibt es für jeden Eintrag in I eine Terminalproduktion, dies ist das Maximum) wählt.

3.4.3 Finden von Additionsprdouktionen

Zum Finden von Additionsproduktionen in einem MTDD M der Höhe h mit $M = (N, P, S)$ wird man sich die Beobachtung 2.1 zunutze machen.
Es geht also bei der Überprüfung von $A = B + C$ mit $A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, und $C \to \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$ darum zu prüfen, ob $A_1 = B_1 + C_1$, $A_2 = B_2 + C_2$, $A_3 = B_3 + C_3$ und $A_4 = B_4 + C_4$ gilt.

Dabei unterscheidet man zwischen zwei Fällen:

Fall 1: man befindet sich in Höhe 1. Beim Vergleich, ob $A = B + C$ gilt, mit $A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, und $C \to \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$, wobei $A, B, C \in N_1$, $A_i, B_i, C_i \in N_0$ für $i = 1, ..., 4$, muss man die Terminale konkret einsetzen und die Matrixaddition berechnen. Es genügt nicht, nur eine Additionsproduktion zu finden, da die Ausgangsmatrix mehrere Möglichkeiten anbieten kann, das Nichtterminal A durch Additionsproduktionen darzustellen.

Beispiel:

Angenommen man kann A_1 durch $A_1 \to B_1 + C_2$ und $A_1 \to C_1 + D_1$ darstellen und in der nächsten Höhe ist A gegeben durch $A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, und $C \to \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$ und $D \to \begin{pmatrix} D_1 & D_2 \\ D_3 & D_4 \end{pmatrix}$, wobei A, B, C und D Höhe k haben, mit $k = 1, ..., h$. Dann ist bei der ersten Additionsproduktion klar, dass diese bei einer

Überprüfung von $A = B + C$ fehlschlägt, da statt C_2 das Nichtterminal C_1 dort stehen müsste.

Dass diese Additionsproduktion einem aber nicht weiterhilft, weiß man aber erst dann, wenn man sich die entsprechenden Nichtterminale in Höhe k anschaut, weshalb immer alle möglichen Kombinationen von $X_1 = X_2 + X_3$ mit $X_1, X_2, X_3 \in N_k$, $X_1 \neq X_2$ und $X_1 \neq X_3$, berechnet und zur nächsten Höhe „weitergereicht" werden müssen.

Aufgrund des Kommutativgesetzes bei der Additon reicht es aus, nur $X_1 = X_2 + X_3$ und nicht noch $X_1 = X_3 + X_2$ zu berechnen. Außerdem ist klar, dass es keine weiteren Additionsproduktionen $A \to B + X$ mit $X \neq C$ geben kann, wenn schon die Additionsproduktion $A \to B + C$ gefunden wurde.

Fall 2: man befindet sich in Höhe $k > 1$. Beim Vergleich, ob $A = B + C$ gilt, mit

$$A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}, \text{ und } C \to \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}, \text{ wobei } A, B, C \in N_k,$$

$A_i, B_i, C_i \in N_{k-1}$ für $i = 1, ..., 4$, muss man wie auch in Fall 1 prüfen, ob aus Höhe $k - 1$ Additionsproduktionen für A_1, A_2, A_3 und A_4 gibt und die zu $B + C$ führen.

Es müssen auch hier alle möglichen Additionsproduktionen, die A repräsentieren, berechnet und zur nächsten Höhe $k + 1$ weitergereicht werden.

Laufzeit: die Laufzeit für das Finden von Additionsproduktionen in der Höhe k, mit $0 \leq k \leq h$, hängt nur von der Anzahl der Nichtterminale der Produktionsmenge $P_k := \{\ p : p$ erzeugt eine Matrix der Höhe $k\ \}$. Da man ja im schlimmsten Fall alle Kombinationen von $A = B + C$ betrachtet (man hat zwar die Anzahl der betrachteten Kombinationen beschränkt, asymptotisch ändert diese Beschränkung jedoch nichts an der Laufzeit), wobei A, B und C Nichtterminale von Produktionen der Produktionsmenge P_k sind, ergibt sich eine Laufzeit von $\Omega(|P_k|^3)$ für den worst-case.

Wendet man dieses Verfahren auf alle Produktionsmengen P_k, mit $0 \leq k \leq h$, so ergibt sich die Lauftzeit $\Omega(h \cdot |P_{max}|^3) = \Omega(\log n \cdot n^6)$, wenn die zu komprimierende Matrix I die Größe $2^h \times 2^h = n^2$ hatte und man $P_{max} := P_0$ (in P_0 gibt es für jeden Eintrag in I eine Terminalproduktion. Davon gibt es im worse-case eben n^2 viele, wenn alle Einträge der Terminale verschieden waren) wählt.

3.4.4 Finden von Skalarmultproduktionen

Das Finden von Skalarmulproduktionen gestaltet sich sehr ähnlich zu dem Finden von Additionsproduktionen. Auch hier müssen stets alle Skalarmultproduktionen in einer MTDD M der Höhe h mit $M = (N, P, S)$ berechnet und zur nächsten Höhe weitergereicht werden.

Es gibt auch hier wieder zwei Fälle:

Fall 1: man befindet sich in Höhe 1. Beim Vergleich, ob $A = c \cdot B$ gilt, mit $A \to$

$\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, $A, B \in N_0$, und $c \in \mathbb{Z}$, wobei $c \neq 0$ und $c \neq 1$, müssen alle Terminale konkret eingesetzt werden und es muss gelten: $\frac{A_1}{B_1} = \frac{A_2}{B_2} = \frac{A_3}{B_3} = \frac{A_4}{B_4} = c$.

Fall 2: man befindet sich in Höhe $k > 1$: Beim Vergleich, ob $A = c \cdot B$ gilt, mit

$A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, $A, B \in N_k$, und $c \in \mathbb{Z}$, müssen $A_1 \to c \cdot B_1$, $A_2 \to c \cdot B_2$, $A_3 \to c \cdot B_3$ und $A_4 \to c \cdot B_4$ in Höhe $k-1$ berechnet worden sein. Fehlt nur eines der aufgeführten Skalarmultproduktionen, so ist $val(A) \neq c \cdot val(B)$.

Laufzeit: die Laufzeit für das Finden von Skalarmultproduktionen in der Höhe k, mit $0 \leq k \leq h$, hängt nur von der Anzahl der Nichtterminale der Produktionsmenge $P_k := \{\ p : p \text{ erzeugt eine Matrix der Höhe } k\ \}$. Da man ja immer alle Kombinationen von $A = c \cdot B$ betrachtet, wobei A und B Nichtterminale von Produktionen der Produktionsmenge P_k und $c \in \mathbb{Z}$ sind, ergibt sich eine Laufzeit von $\Omega(|P_k|^2)$.

Wendet man dieses Verfahren auf alle Produktionsmengen P_k, mit $0 \leq k \leq h$, so ergibt sich die Lauftzeit $\Omega(h \cdot |P_{max}|^2) = \Omega(\log n \cdot n^4)$, wenn die zu komprimierende Matrix I die Größe $2^h \times 2^h = n^2$ hatte und man $P_{max} := P_0$ (in P_0 gibt es für jeden Eintrag in I eine Terminalproduktion. Davon gibt es im worse-case eben n^2 viele, wenn alle Einträge der Terminale verschieden waren) wählt.

3.4.5 Differenzmatrix

Es gibt Matrizen, die nicht mit Hilfe von Additionsproduktionen, die durch die gezeigte Methode zum Finden von Additionsproduktionen, und Skalarmultproduktionen komprimiert werden können, aber durch das Berechnen von Differenzmatrizen kann eine Kompression möglich sein.

Beispiel:

Sei A eine 4×4-Matrix mit $A = \begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{pmatrix}$ und $A = concat(A_1, A_2, A_3, A_4)$.

Da alle Einträge verschieden sind, gibt es keine Gleichheiten. Man kann auch weder Additions- noch Skalarmultproduktionen finden.

Schaut man sich die Matrix A aber genauer an, so erkennt man zum Beispiel, dass

$A_2 - A_1 = \begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix}$ ergibt. Genauso auch $A_3 - A_2 = A_4 - A_3 = \begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix}$.

Das könnte zu einem MTDD$_+$ M der Höhe 2 mit $M = (N, P, S)$ und

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A_4, C\}$ und $N_0 = \{T_1, T_2, T_3, T_4\}$
- $S = A_0$ und
- $P = \{$

$$A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix},$$
$$A_1 \to \begin{pmatrix} T_1 & T_2 \\ T_3 & T_4 \end{pmatrix},$$
$$A_2 \to A_1 + C,$$
$$A_3 \to A_2 + C,$$
$$A_4 \to A_3 + C,$$
$$C \to \begin{pmatrix} T_4 & T_4 \\ T_4 & T_4 \end{pmatrix}$$
$$\} \cup \{\ T_i \to i \mid T_i \in N_0 \text{ und } i \in \{1, 2, 3, 4\}\}$$

führen. Deshalb scheint die Berechnung einer Differenzmatrix recht sinnvoll, wobei man sich in dieser Arbeit auf *konstante Matrizen* als Ergebnis der Differenzmatrix beschränkt, also einer Matrix, wo alle Einträge gleich sind, da diese als MTDD in logarithmischer Größe dargestellt werden können und es deshalb nicht so sehr ins Gewicht fallen kann, wenn konstante Matrizen in die $\text{MTDD}_{(+,\cdot)}$ hinzugefügt werden.

Die Differenzmatrix wird auch nur dann hinzugefügt, sollte die Additionsproduktionen, die die Differenzmatrix beinhaltet, eine DownStep-Produktion wirklich ersetzen. Die Additionsproduktionen, die Differenzmatrizen beinhalten, müssen somit gesondert im Schritt (3) des Kompressionverfahrens betrachtet werden.

Wie in den letzten beiden Abschnitten zuvor müssen stets alle Differenzmatrizen in einem MTDD M der Höhe h mit $M = (N, P, S)$ berechnet und weitergereicht werden. Darüberhinaus unterscheidet man auch wie zuvor zwischen zwei Fällen:

Fall 1: man befindet sich in Höhe 1. Beim Vergleich, ob $A - B = C$ gilt, mit

$A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, $A, B \in N_0$, und C eine konstante Matrix der Höhe 1, müssen alle Terminale konkret eingesetzt werden und es muss gelten: $A_1 - B_1 = A_2 - B_2 = A_3 - B_3 = A_4 - B_4 = c$ mit $c \in \mathbb{Z}$.

Somit wäre $C \to \begin{pmatrix} T_c & T_c \\ T_c & T_c \end{pmatrix}$ mit $T_c \to c$ und es würde $A = B + C$ gelten, wobei man diese Produktion nicht als Additionsproduktion weiterreichen würde. Man muss diese Produktion als Sonderfall kenntlich machen (beispielsweise als Tupel $(B,\ C)$) , da ja die konstante Matrix C erst wirklich „existiert“, wenn die Produktion auch eine

DownStep-Produktion ersetzt (siehe für Genaueres in der Implementierung des Verfahrens nach).

Fall 2: man befindet sich in Höhe $k > 1$: Beim Vergleich, ob $A - B = C$ gilt, mit

$A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B \to \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, $A, B \in N_k$ und $C \to \begin{pmatrix} C' & C' \\ C' & C' \end{pmatrix}$

eine konstante Matrix der Höhe k, müssen $A_1 - B_1 = C'$, $A_2 - B_2 = C'$, $A_3 - B_3 = C'$ und $A_4 - B_4 = C'$ in Höhe $k-1$ berechnet worden sein. Fehlt nur eines der aufgeführten Differenzen, so ist $A - B \neq C$.

Kann ein Nichtterminal A sowohl durch eine „normale“ Additions- oder Skalarmultproduktion als auch durch eine Additionsproduktion, die durch die Differenzmatrix erzeugt wird, dargestellt werden, so sind die beiden erstgenannten Produktionen vorzuziehen, wenn es um die Ersetzung einer DownStep-Produktion geht.

Laufzeit: die Laufzeit für das Finden von Additionsproduktionen durch Berechnen von Differenzmatrizen in der Höhe k, mit $0 \leq k \leq h$, hängt nur von der Anzahl der Nichtterminalen der Produktionsmenge $P_k := \{\ p : p \text{ erzeugt eine Matrix der Höhe } k\ \}$. Da man ja immer alle Kombinationen von $A - B = \textit{konstantes MTDD}$ betrachtet, wobei A und B Nichtterminale von Produktionen der Produktionsmenge P_k sind, ergibt sich eine Laufzeit von $\Omega(|P_k|^2)$. Das Einfügen von konstanten MTDDs wird in die Betrachtung nicht miteinbezogen.

Wendet man dieses Verfahren auf alle Produktionsmengen P_k, mit $0 \leq k \leq h$, so ergibt sich die Lauftzeit $\Omega(h \cdot |P_{max}|^2) = \Omega(logn \cdot n^4)$, wenn die zu komprimierende Matrix I die Größe $2^h \times 2^h = n^2$ hatte und man $P_{max} := P_0$ (in P_0 gibt es für jeden Eintrag in I eine Terminalproduktion. Davon gibt es im worse-case eben n^2 viele, wenn alle Einträge der Terminale verschieden waren) wählt.

3.4.6 Zyklen im MTDD

Ein Problem, welches Entstehen kann, wenn man „blind“ die DownStep-Produktionen durch die gefundenen Additions- und Skalarmultproduktion ersetzt, ist das Erzeugen von Zyklen innerhalb eines MTDDs. Dies ist wichtig zu berücksichtigen, da zum einen eine Dekompression des MTDD nicht mehr möglich ist (die Dekompression terminiert dann nicht mehr), zum anderen arbeiten viele Matrixoperation auf MTDDs mit dem „Auflösen“ von Nichtterminalen.

Man betrachte zu diesem Problem folgendes Beispiel:

Angenommen es gebe Nichtterminale A_1, A_2, A_3 und A_4 der Höhe 1 mit

$val(A_1) = \begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix}$, $val(A_2) = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$, $val(A_3) = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ und

$val(A_4) = \begin{pmatrix} -3 & -3 \\ -3 & -3 \end{pmatrix}$.

Mit diesen Nichtterminalen ergeben sich unter anderem die Produktionen

- $A_1 \to A_2 + A_2$
- $A_2 \to A_3 + A_3$
- $A_3 \to A_1 + A_4$

Wenn man nun diese Produktionen in das MTDD einsetzen würde, so hätte man den Zyklus $A_1 \to A_2 \to A_3 \to A_1$.

Erwähnenswert ist bei den Skalarmultproduktionen, dass es bei der ausschließlichen Betrachtung jener nur einen Fall gibt, der zu einem Zyklus führen kann: hat man eine Produktion $A \to (-1) \cdot B$ gefunden, so hat man auch automatisch die Produktion $B \to (-1) \cdot A$, denn $B \to (-1) \cdot A$ entspricht beim Einsetzen von $A => B \to (-1) \cdot (-1) \cdot B$ $=> B \to B$.

Es kann unter Umständen auch der Fall vorkommen, dass dann keine Produktion ersetzt werden darf, da sonst ein Zyklus im MTDD enstehen würde!

Deshalb muss bei jedem Ersetzen geprüft werden, ob die einzusetzende Produktion einen Zyklus erzeugt.

3.4.7 Löschen von nicht-referenzierten Nichtterminalen

In den bis jetzt vorgestellten Beispielen wurde der Schritt des Löschens von nicht-referenzierten (gebrauchten) Nichtterminalen nicht explizit erwähnt, er ist aber ein wichtiger (letzter) Schritt beim Komprimieren von MTDDs. Dass es solch einen Fall gegeben kann, soll folgendes Beispiel zeigen.

Beispiel:

Sei A eine 4×4-Matrix mit $A = \begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix}$ und $A = concat(A_1, A_2, A_3, A_4)$.

Nach dem bisherigen Verfahren ohne Schritt (5) würde die entsprechende $\text{MTDD}_{(+,\cdot)}$ M der Höhe 2 mit $M = (N, P, S)$ so ausehen:

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3\}$ und $N_0 = \{T_1, T_2, T_3, T_4\}$
- $S = A_0$ und
- $P = \{$

$$
\begin{aligned}
&A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_3 \end{pmatrix},\\
&A_1 \to \begin{pmatrix} T_1 & T_1 \\ T_1 & T_1 \end{pmatrix},\\
&A_2 \to 2 \cdot A_1,\\
&A_3 \to 3 \cdot A_1,\\
&A_4 \to 4 \cdot A_1,\\
&\} \cup \{\ T_i \to i \mid T_i \in N_0 \text{ und } i \in \{1,2,3,4\}\}
\end{aligned}
$$

Vor der Ersetzung der DownStep-Produktionen von A_2, A_3 und A_4 wurden die Nichtterminale T_2, T_3, T_4 gebraucht, die nun alleine durch das Nichtterminal T_1 repräsentiert werden und somit nutzlos sind. Deshalb kann man die Produktionen $T_2 \to 2$, $T_3 \to 3$ und $T_4 \to 4$ löschen.

Laufzeit: Ist $|M|$ die Größe eines MTDD$_{(+,\cdot)}$, so beträgt die Laufzeit $\Omega(|M|)$, da man sich jede Produktion einmal anschauen muss. Löschoperationen sind hier nicht mitberücksichtigt worden.

3.4.8 Beispiel

Es folgt nun ein (kleines) Beispiel für das vorgestellte Kompressionsverfahren.

Sei A eine 4×4-Matrix mit

$$
A = \begin{pmatrix} 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 5 & 5 & 6 \\ 4 & 5 & 4 & 5 \end{pmatrix}.
$$

Schritt (1): Erzeugen des vollständigen MTDDs

Das vollständige MTDD M mit $M = (N, P, S)$ ist

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A_4\}$ und $N_0 = \{T_i \mid 1 \leq i \leq 16\}$
- $S = A_0$ und
- $P = \{$

$$
\begin{aligned}
&A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix},\\
&A_1 \to \begin{pmatrix} T_1 & T_2 \\ T_3 & T_4 \end{pmatrix},
\end{aligned}
$$

$$A_2 \to \begin{pmatrix} T_5 & T_6 \\ T_7 & T_8 \end{pmatrix},$$
$$A_3 \to \begin{pmatrix} T_9 & T_{10} \\ T_{11} & T_{12} \end{pmatrix},$$
$$A_4 \to \begin{pmatrix} T_{13} & T_{14} \\ T_{15} & T_{16} \end{pmatrix},$$
$T_1 \to 1,$

...,

$T_{16} \to 5$ }

Schritt (2): Finden von gleichen Nichtterminalen - in Höhe 0

Eine Suche in Höhe 0 ergibt, dass

- die Menge von Nichtterminalen $\{T_1, T_2\}$ das Terminal 1,
- die Menge von Nichtterminalen $\{T_3, T_4, T_7, T_8\}$ das Terminal 0,
- die Menge von Nichtterminalen $\{T_5, T_6\}$ das Terminal 2,
- die Menge von Nichtterminalen $\{T_9, T_{11}, T_{15}\}$ das Terminal 4,
- die Menge von Nichtterminalen $\{T_{10}, T_{12}, T_{13}, T_{16}\}$ das Terminal 5 erzeugen,

Ersetze in der nächsten Höhe alle Nichtterminale der oben gefundenen Mengen jeweils durch das erste Element der Mengen. Lösche alle Produktionen von gleichen Nichtterminalen.

Wiederhole Schritt (2): Finden von gleichen Nichtterminalen - in Höhe 1

Es werden zunächst die nötigen Ersetzungen gemacht. Eine Suche in Höhe 1 ergibt, dass

- es in dieser Höhe keine gleichen Nichtterminale gibt.

Schritt (3): Finden von Additionsproduktionen - in Höhe 1

Eine Suche in Höhe 1 ergibt, dass

- A_2 durch $A_1 + A_1$,
- A_4 durch $A_1 + A_3$ dargestellt werden können.

Anmerkung: Additionen wie $B + C$, wobei C eine *Nullmatrix* (alle Einträge sind 0) ist, müssen nicht gemerkt werden, da man sich stattdessen für jede Höhe merken kann, ob und welches Nichtterminal eine Nullmatrix erzeugt.

Schritt (3): Finden von Skalarmultproduktionen - in Höhe 1

Eine Suche in Höhe 1 ergibt, dass

- A_2 durch $2 \cdot A_1$ dargestellt werden kann.

Schritt (3): Berechnung von Differenzmatrizen - in Höhe 1

Eine Suche in Höhe 1 ergibt, dass

- es in dieser Höhe keine Differenzmatrizen gibt, die konstante Matrizen darstellen.

Aus den gefundenen Additions- und Skalarproduktionen ersetzt man die DownStep-Produktionen

- von A_2 durch $A_1 + A_1$,
- von A_4 durch $A_1 + A_3$.

Schritt (2): Finden von gleichen Nichtterminalen - in Höhe 2

Dieser und alle anderen Schritte entfallen, dass es in Höhe 1 keine gleichen Nichtterminale gab und es somit keine Ersetzungen gibt.

Das vorläufige Endergebnis ist dann M mit

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A4\}$ und $N_0 = \{T_i \mid i \in \{1, 3, 5, 9, 10, 14\}\}$
- $S = A_0$ und
- $P = \{$

$$A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_3 \end{pmatrix},$$
$$A_1 \to \begin{pmatrix} T_1 & T_1 \\ T_3 & T_3 \end{pmatrix},$$
$$A_2 \to A_1 + A_1,$$
$$A_3 \to \begin{pmatrix} T_9 & T_{10} \\ T_9 & T_{10} \end{pmatrix},$$
$$A_4 \to A_1 + A_3,$$

$T_1 \to 1,$

$T_3 \to 0,$

$T_5 \to 2,$

$T_9 \to 4,$

$T_{10} \to 5,$

$T_{14} \to 6\ \}$

Schritt (5): Löschen von nicht-referenzierten Nichtterminalen

Beim Durchlaufen wird festgestellt, dass die Nichtterminale T_5 und T_{14} nicht mehr gebraucht werden.

Daraus ergibt sich für M als Endergebnis:

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A4\}$ und $N_0 = \{T_i \mid i \in \{1, 3, 9, 10\}\}$
- $S = A_0$ und
- $P = \{$

  $A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_3 \end{pmatrix},$

  $A_1 \to \begin{pmatrix} T_1 & T_1 \\ T_3 & T_3 \end{pmatrix},$

  $A_2 \to A_1 + A_1,$

  $A_3 \to \begin{pmatrix} T_9 & T_{10} \\ T_9 & T_{10} \end{pmatrix},$

  $A_4 \to A_1 + A_3,$

  $T_1 \to 1,$

  $T_3 \to 0,$

  $T_9 \to 4,$

  $T_{10} \to 5\ \}$

Laufzeit: die Laufzeit für das Gesamtverfahren wird dominiert von der Laufzeit für das Finden von Additionsproduktionen, somit ist die Laufzeit $\Omega(logn \cdot n^6)$, für $n =$ Zeilen- bzw. Spaltenanzahl der zu komprimierenden Matrix.

3.5 Dekomprimieren von MTDD$_{(+,\cdot)}$s

Ein Dekomprimieren von MTDD$_{(+,\cdot)}$s ist nichts anderes als das Anwenden von Matrixoperationen (mit der Konkatenation von Matrizen) auf schon dekomprimierte Matrizen.

Sei das MTDD$_{(+,\cdot)}$ M der Höhe h mit $M = (N, P, S)$ gegeben. Dann wird ein Nichtterminal $A \in N_k$ mit $k = 1, ..., h$ und $A \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$ dekomprimiert, also $val(A)$ errechnet, indem

1. A_1, A_2, A_3, A_4 dekomprimiert, also $val(A_1), val(A_2), val(A_3)$ und $val(A_4)$ errechnet, und
2. $concat(val(A_1), val(A_2), val(A_3), val(A_4))$ berechnet werden.

Beispiel:

Sei das MTDD$_{(+,\cdot)}$ M der Höhe h mit $M = (N, P, S)$ gegeben durch

- $N = N_2 \cup N_1 \cup N_0$, mit $N_2 = \{A_0\}$, $N_1 = \{A_1, A_2, A_3, A_4\}$ und $N_0 = \{T_1, T_3, T_4\}$
- $S = A_0$ und
- $P = \{$
  $A_0 \to \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$,
  $A_1 \to \begin{pmatrix} T_1 & T_1 \\ T_1 & T_1 \end{pmatrix}$,
  $A_2 \to \begin{pmatrix} T_3 & T_4 \\ T_4 & T_3 \end{pmatrix}$,
  $A_3 \to A_1 + A_2$,
  $A_4 \to 2 \cdot A_1$,
  $\} \cup \{\ T_i \to i \mid T_i \in N_0 \text{ und } i \in \{1, 3, 4\}\}$

Um A_0 zu dekomprimieren, müssen zunächst A_1, A_2, A_3 und A_4 dekomprimiert werden:

- Dekomprimiere A_1: wir können hier direkt die Terminale (T_1) einsetzen und die Matrix $concat(T_1, T_1, T_1, T_1)$ erzeugen. Somit ist $val(A1) = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$.
- Dekomprimiere A_2: analog zu A_1 und deshalb ist $val(A_2) = \begin{pmatrix} 3 & 4 \\ 4 & 3 \end{pmatrix}$.

- Dekomprimiere A_3: da A_3 die Produktion $A_1 + A_2$ erzeugt und somit $val(A_3) = val(A_1) + val(A_2)$, ist $val(A_3) = \begin{pmatrix} 4 & 5 \\ 5 & 4 \end{pmatrix}$.
- Dekomprimiere A_4: da A_4 die Produktion $2 \cdot A_1$ erzeugt und somit $val(A_4) = 2 \cdot val(A_1)$, ist $val(A_3) = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$.
- Dekomprimiere A_0: da A_0 die Produktion $\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$ erzeugt und somit $val(A_0) = concat(val(A_1), val(A_2), val(A_3), val(A_4))$, ist $val(A_0) = \begin{pmatrix} 1 & 1 & 3 & 4 \\ 1 & 1 & 4 & 3 \\ 4 & 5 & 2 & 2 \\ 5 & 4 & 2 & 2 \end{pmatrix}$.

Laufzeit: man muss für jedes Nichtterminal, die Matrix berechnen. Dies hängt aber von der Höhe des Nichtterminals ab.

In einer Höhe k, mit $0 \leq k \leq h$, und der Produktionsmenge $P_k := \{ p : p$ erzeugt eine Matrix der Höhe $k \}$ ergibt sich eine Laufzeit für diese Höhe von $|P_k| \cdot 4^k$ (4^k ist gerade die Größe der Matrizen, die in dieser Höhe von den Produktionen erzeugt werden und man geht von einer Laufzeit von $\Omega(|I|)$ für die Konkatenation, Addition, Subtraktion und Skalarmultiplikation aus, wenn $|I|$ die Größe der resultierenden Matrix ist).

Wendet man dieses Verfahren auf alle Produktionsmengen P_k, mit $0 \leq k \leq h$, so ergibt sich die Lauftzeit $\Omega(h \cdot |P_{max}| \cdot 4^h) = \Omega(\log n \cdot n^4)$, wenn die zu komprimierende Matrix I die Größe $2^h \times 2^h = n^2$ hatte und man $P_{max} := P_0$ (in P_0 gibt es für jeden Eintrag in I eine Terminalproduktion. Davon gibt es im worse-case eben n^2 viele, wenn alle Einträge der Terminale verschieden waren) wählt.

Kapitel 4

Implementierung

4.1 Das Modul `MTDDPlusScal`

In diesem Modul wird der Datentyp für ein $\text{MTDD}_{(+,\cdot)}$ und einigen Hilfsfunktionen implementiert.

Da dem Modul `Data.Map` der alias Map beim Import vergeben wurde, werden in diesem Modul alle Datentypen und Funktionen dieses Moduls ein `Map.` vorangestellt.

4.1.1 Repräsentierung eines $\text{MTDD}_{(+,\cdot)}$ in Haskell

Die Implementierung des Datentyps, der ein $\text{MTDD}_{(+,\cdot)}$ in Haskell repräsentiert, wurde im Wesentlichen von [RW13] übernommen:

```
type NonTerminal = Int

data RightHandSide =
  DownStep NonTerminal NonTerminal NonTerminal NonTerminal
  | Addition NonTerminal NonTerminal
  | ScalMult Int NonTerminal
  | Terminal Int
  deriving Show

type ProdMap = Map.Map NonTerminal RightHandSide

data MTDDPlusScal = MTDDPlusScal {
  startSymbol :: NonTerminal,
  productions :: [ProdMap]
  } deriving Show
```

Ein $\text{MTDD}_{(+,\cdot)}$ M der Höhe h wird demnach durch den Typen `MTDDPlusScal s pms` dargestellt, wobei `s` das Startsymbol und der Typ `[ProdMap]` die Produktionen von M ist.

Eine Produktion wird durch die linke Seite als Nichtterminal, hier `NonTerminal`, und eine rechte Seite, hier `RightHandSide`, dargestellt. Produktionen gleicher Höhe werden in einer `Map` (siehe [map]) gespeichert, wobei als Schlüssel die linke Seite und als Wert die rechte Seite einer Produktion dient. Die Gesamtmenge der Produktionen wird durch eine Liste von Maps repräsentiert, wobei das erste Element der Liste die Map von Produktionen der Höhe h, das zweite Element die Map von Produktionen der Höhe $h-1$, ..., und das letzte Element der Liste die Map von Produktionen der Höhe 0 darstellen. Der Konstruktor `ScalMult Int NonTerminal` des Datentypen `RightHandSide` steht für die rechte Seite einer Skalarmultproduktion $A \to c \cdot B$, wobei `Int` den Skalar c und

`NonTerminal` das Nichtterminal B abbilden.

Im Gegensatz zur Implementierung in [RW13] ist der Datentyp für MTDDs nicht mehr polymorph, da es für das Kompressionsverfahren nicht notwendig war. Außerdem haben Tests ergeben, dass es speichertechnisch und deswegen auch laufzeittechnisch „günstiger“ ist, Nichtterminale durch Integer vom Typ `Int` darzustellen. Man hat damit zwar eine obere Schranke für die Größe von den zu komprimierenden Matrizen, in der Praxis wird diese Schranke aber kaum eine Rolle spielen, da sie aufgrund der Laufzeit des Kompressionsverfahrens nie ausgereizt werden sollte.

4.1.2 Größe eines MTDD$_{(+,\cdot)}$

Die Funktion `size` berechnet die Größe eines MTDD$_{(+,\cdot)}$ bei Eingabe jenes:

```
size :: MTDDPlusScal -> Int
size (MTDDPlusScal _ pms) = foldl (\s pm -> s +
  Map.foldl (\s' rhs -> case rhs of
  (DownStep _ _ _ _) -> 5 + s'
  (Addition _ _) -> 3 + s'
  (ScalMult _ _) -> 3 + s'
  (Terminal _) -> 2 + s') 0 pm) 0 pms
```

Man geht demnach alle Produktionen durch und summiert dabei die Größen der einzelnen Produktionen auf.

4.1.3 Kompressionsrate zwischen MTDD$_{(+,\cdot)}$ und Matrix

Die Funktion `compressionRateMat` berechnet die Kompressionsrate eines (komprimierten) MTDD$_{(+,\cdot)}$ gegenüber der entsprechenden Matrix bei Eingabe des T MTDD$_{(+,\cdot)}$:

```
compressionRateMat :: (Fractional a) => MTDDPlusScal -> a
compressionRateMat mtdd@(MTDDPlusScal _ pms) = let
  h = length pms - 1
  sizeMat = 4^h
 in 1 - (fromInteger (toInteger (size mtdd)) / sizeMat)
```

Das Ergebnis ist eine Zahl i, die aussagt, dass die Eingabematrix um i komprimiert wurde.

4.1.4 Kompressionsrate zwischen MTDD$_{(+,\cdot)}$ und vollständigem MTDD

Die Funktion `compressionRateCompMTDD` berechnet die Kompressionsrate eines (komprimierten) MTDD$_{(+,\cdot)}$ gegenüber dem entsprechenden vollständigen MTDD bei Eingabe des MTDD$_{(+,\cdot)}$:

```
compressionRateCompMTDD :: (Fractional a) => MTDDPlusScal -> a
```

```
compressionRateCompMTDD mtdd@(MTDDPlusScal _ pms) = let
  h = length pms - 1
  dim = 2^h
  sizeCompMTDD = (fromInteger (2 * dim * dim)) +
    (fromInteger (foldl (\s i -> s + (4^i) * 5) 0 $ (take h $ [0..])))
 in 1 - (fromInteger (toInteger (size mtdd)) / sizeCompMTDD)
```

Da die Größe eines vollständigen MTDD nur von dessen Höhe h abhängt, kann man diese direkt mit Hilfe von h berechnen:

- Für jeden Eintrag in der Matrix gibt es eine entsprechende Terminalproduktion (der Größe 2), deshalb 2· Anzahl der Größe der Matrix
- In jeder Höhe vervierfacht sich die Anzahl der Produktionen, wobei jede Produktion eine DownStep-Produktion ist (der Größe 5), somit ist die Gesamtanzahl der Produktionen von Höhe h bis Höhe 1 gegeben durch $\sum_{i=0}^{h-1} 5 \cdot 4^i$

Das Ergebnis ist eine Zahl i, die aussagt, dass das vollständige MTDD um i komprimiert werden kann.

4.1.5 Erzeugen eines konstanten MTDD

Die Funktion `createConstMTDD` erzeugt ein MTDD, die eine konstante Matrix repräsentiert, bei Eingabe der Höhe der Matrix, des konstanten Eintrags und eines noch nicht vorkommenden Nichtterminals:

```
createConstMTDD :: Int -> Int -> NonTerminal
                -> (MTDDPlusScal, NonTerminal)
createConstMTDD height entry curr_nt = let
  pms = createConstMTDD' height curr_nt
 in (MTDDPlusScal curr_nt pms, curr_nt + height + 1) where
  createConstMTDD' 0 nt = [Map.singleton nt (Terminal entry)]
  createConstMTDD' h nt = let
    new_nt = nt + 1
    new_map = Map.singleton nt (DownStep new_nt new_nt new_nt new_nt)
   in new_map : createConstMTDD' (h - 1) new_nt
```

Die lokale Funktion `createConstMTDD‘` erzeugt dabei die Produktionen der MTDD, indem für jede Höhe eine Map mit nur einer DownStep-Produktion kreiert wird, in deren rechten Seite alle Nichtterminale gleich sind. Dabei werden die linken Seiten der Produktionen einfach durchnummeriert, beginnend mit `curr_nt`. Somit wird die minimalste Darstellung einer konstanten Matrix durch ein MTDD erzeugt. Das Ergebnis ist ein Tupel bestehend aus MTDD als `MTDDPlusScal` und des nächsten, frei verfügbaren Nichtterminals.

4.1.6 Hilfsfunktionen

Die Funktion `getNTDSRule` holt sich aus der rechten Seite einer Produktion DownStep-Produktion das i-te Nichtterminal bei Eingabe der rechten Seite einer DownStep-Produktion und einer Zahl i:

```
getNTDSRule :: RightHandSide -> Int -> NonTerminal
getNTDSRule (DownStep nt1 nt2 nt3 nt4) i
  | i == 1 = nt1
  | i == 2 = nt2
  | i == 3 = nt3
  | i == 4 = nt4
  | otherwise = error ("Error in 'getNTDSRule': index out of bounds!")
```

Es wird ein Fehler ausgegeben, falls $i < 1$ oder $i > 4$ sein sollte.

Die Funktion `getValFromNTOne` holt sich das Terminal aus der rechten Seite einer Terminalproduktion bei Eingabe der linken Seite (des Nichtterminals) und der Map von Produktionen der Höhe 0:

```
getValFromNTOne :: NonTerminal -> ProdMap -> Int
getValFromNTOne nt pm0 = case Map.lookup nt pm0 of
      Just (Terminal t) -> t
      Nothing -> error ("Error in 'getValFromNTOne': non-terminal "
       ++ show nt ++ " not found!")
```

Es wird ein Fehler ausgegeben, falls die linke Seite nicht in der Map von Produktionen auftauchen sollte.

4.2 Das Modul `Data.Matrix`

Das Modul `Data.Matrix` (siehe [mat]) wurde in dieser Implementierung zur Darstellung von Matrizen genutzt.

Es ist zwingend notwendig dieses zu installieren, da es nicht standardmäßig mit der Haskell-Plattform ausgeliefert wird. Dazu muss man über *cabal* (siehe [cab]) das Cabal-Package zu `Data.Matrix` installieren: unter Windows öffnet man die Betriebssystem-Shell und gibt dort „`cabal install matrix`“ (ohne Anführungszeichen) ein.

Dieses Modul bietet eine einfache Möglichkeit um

- Matrizen zu generieren
- Matrixeinträge abzufragen (in konstanter Laufzeit)
- Matrizen zu konkatenieren
- und vieles mehr

Der Typ einer Matrix ist in dieser Implementierung stets durch `Matrix Int` gegeben.

4.3 Das Modul `MatrixCompression`

Da dem Modul `Data.Map` der Alias `Map` beim Import vergeben wurde, werden somit in diesem Modul alle Datentypen und Funktionen dieses Moduls über `Map.` angesprochen.
Dem Modul `Data.Matrix` wurde der Alias `Mat` vergeben.

In diesem Modul befinden sich die Funktionen, mit denen man eine Matrix bzw. ein MTDD komprimieren kann. Dazu werden folgende Funktionen bereitgestellt:

- `compressMat0` und `compressMTDD0`
- `compressMat01` und `compressMTDD01`
- `compressMat02` und `compressMTDD02`
- `compressMat03` und `compressMTDD03`
- `compressMat012` und `compressMTDD012`
- `compressMat0123` und `compressMTDD0123`

Funktionen, die `Mat` im Namen haben, erwarten eine Matrix vom Typ `Mat.Matrix Int` und Funkionen, die MTDD im Namen haben, erwarten einen **vollständigen MTDD** vom Typ `MTDDPlusScal`. Alle Funktionen liefern als Ergebnis ein $\text{MTDD}_{(+,\cdot)}$ vom Typ `MTDDPlusScal`.

Die Ziffern in den Funktionsnamen stehen für:

- 0 für das Finden von gleichen Nichtterminalen (siehe 3.4.2)
- 1 für das Finden von Additionsproduktionen (siehe 3.4.3)
- 2 für das Finden von Skalarmultproduktionen (siehe 3.4.4)
- 3 für das Finden von Additionsproduktionen durch Berechnen von Differenzmatrizen (siehe 3.4.5)

Demnach wird beispielsweise eine Matrix durch die Funktion `compressMat012` komprimiert, indem alle gleichen Nichtterminale beseitigt, alle möglichen Additions- und Skalarmultproduktionen gefunden und ersetzt werden.

4.3.1 Erzeugen eines vollständigen MTDD

Die Funktion `createCompMTDD` erzeugt aus einer Matrix die Produktionsmenge P eines vollständigen MTDD $M = (N, S, P)$ bei Eingabe einer $2^h \times 2^h$-Matrix und deren Höhe h:

```
createCompMTDD :: Mat.Matrix Int -> Int -> [ProdMap]
createCompMTDD mat height = createCompMTDD' height 0 1 where
  createCompMTDD' 0 nt_start _ =
    getMatEntries mat height nt_start Map.empty : [ ]
  createCompMTDD' h nt_start range = let
    new_map = addAllProds nt_start range Map.empty
   in new_map : createCompMTDD' (h-1) (nt_start + range) (range*4)
  addAllProds nt i cmap = let
    nt_rule = nt * 4 + 1
    new_map = Map.insert nt (DownStep nt_rule (nt_rule+1) (nt_rule+2)
      (nt_rule+3)) cmap
   in if i == 0 then cmap else addAllProds (nt+1) (i-1) new_map
```

In der Unterfunktion `createCompMTDD'` wird rekursiv für jede Höhe - also von Höhe h bis Höhe 0 - die Map von Produktionen erzeugt. Dabei geschieht dies wie in 3.4.1 beschrieben in 2 Teilen:

Fall 1: die DownStep-Produktionen der Höhe k mit $1 \leq k \leq h$ werden durch die Unterfunktion `addAllProds` erzeugt. Dabei steht das Argument `nt` für die linke Seite der Produktion, die im aktuellen Aufruf in die Map von Produktionen eingesetzt werden. Das Argument i ist eine Laufvariable, sie geht von `range` bis 0, es werden somit immer genau `range`-viele Produktionen für die aktuelle Höhe erzeugt.

Die Benennung der Nichtterminale hat das folgende Muster:

Höhe h:

0 → DownStep 1 2 3 4

Höhe $h - 1$:

1 → DownStep 5 6 7 8
2 → DownStep 9 10 11 12
3 → DownStep 13 14 15 16
4 → DownStep 17 18 19 20

Höhe $h - 2$:

5 → DownStep 21 22 23 24

... usw.

Teil 2: die Terminalproduktionen werden durch die Funktion `getMatEntries` gebildet:

```
getMatEntries :: Mat.Matrix Int -> Int -> Int -> ProdMap-> ProdMap
getMatEntries mat 1 curr_nt curr_map = let
  getMatEnt mat nt (i, j) counter cmap = let
    new_map =
     Map.insert nt (Terminal $ Mat.getElem (i + 1) (j + 1) mat) cmap
   in if counter == 0 then cmap else
     getMatEnt mat (nt + 1) (i + j, (j + 1) `mod` 2) (counter-1) new_map
 in getMatEnt mat curr_nt (0,0) 4 curr_map
getMatEntries mat h curr_nt curr_map = let
  pivot_mat = 2 ^ (h - 1)
  (tl_mat, tr_mat, bl_mat, br_mat) =
    Mat.splitBlocks pivot_mat pivot_mat mat
  nt_range = pivot_mat * pivot_mat
  tr_nt = curr_nt + nt_range
  bl_nt = tr_nt + nt_range
  br_nt = bl_nt + nt_range
  curr_map1 = getMatEntries tl_mat (h - 1) curr_nt curr_map
  curr_map2 = getMatEntries tr_mat (h - 1) tr_nt curr_map1
  curr_map3 = getMatEntries bl_mat (h - 1) bl_nt curr_map2
 in getMatEntries br_mat (h - 1) br_nt curr_map3
```

Die Argumente der Funktion sind die Matrix, die komprimiert werden soll, die Höhe der Matrix, das Nichtterminal, welches als linke Seite einer Produktion dienen soll und die aktuelle Map von (Terminal-)Produktionen.

Die Funktion teilt in jedem Schritt die Matrix durch die Funktion `splitBlocks` in 4 gleich große (Teil-)Matrizen auf (beide Argumente sagen der Funktion, an welcher Stelle horizontal und an welcher Stelle vertikal die Matrix aufgeteilt werden soll). Das Argument `pivot_mat` steht gerade für die Stelle der Teilung: wenn die Ausgangsmatrix eine $2^h \times 2^h$- Matrix ist, dann sind deren 4 gleich großen Matrizen $2^{h-1} \times 2^{h-1}$-Matrizen. Die Menge der benötigten Nichtterminale für eines der Teilmatrizen entspricht gerade deren Größe.

Die obige Funktion teilt die Ausgangsmatrix solange auf, bis man nur noch 2×2-Teilmatrizen hat. Diese werden dann mit der Hilfsfunktion `getMatEnt` durchgegangen.

Wenn also $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ die betrachtete 2×2-Matrix ist, so wird zuallererst für 1,

dann für 2, anschließend für 3 und schließlich für 4 jeweils die Terminalproduktion erzeugt.

Das Berechnen der aktuell betrachteten Zeile i und Spalte j der 2×2-Matrix durch `(i + j, (j + 1) ‘mod‘ 2)` erzeugt gerade das oben am Beispiel beschriebene „Abfrageverhalten“: man fängt mit `(0, 0)`. Das nächste Argument ist `(0+0, 0+1 ‘mod‘ 2)` = `(0,1)`. Dann `(0+1, 1+1 ‘mod‘ 2)` = `(1,0)` und schließlich `(1+0, 0+1 ‘mod‘ 2)` = `(1,1)`.

Man beachte, dass die Indizierung von Matrizen des Typs `Mat.Matrix Int` bzw. `Data.Matrix` mit 1 beginnt, weshalb immer auf die aktuell betrachtete Zeile und Spalte (mit der Funktion `getElement`) eine 1 addiert wird.

4.3.2 Finden von gleichen Nichtterminalen

Die Funktion `eliminateEqualNTs` geht bei Eingabe einer Map von Produktionen der Höhe k mit $0 \leq k \leq h$ alle Produktionen dieser Höhe durch, löscht alle Nichtterminale, die die gleichen Matrizen repräsentieren und merkt sich dabei alle gelöschten Nichtterminale:

```
eliminateEqualNTs :: ProdMap
                  -> (ProdMap, Map.Map NonTerminal NonTerminal)
eliminateEqualNTs pm =
  eliminateEqualNTs' pm 0 (Map.size pm) Map.empty where
  eliminateEqualNTs' pm cindex cmap_size cequal_map
    | cindex >= cmap_size = (pm, cequal_map)
    | otherwise = let
      (nt, rhs) = Map.elemAt cindex pm
     in case rhs of
       (Terminal t) -> let
         (new_pm, new_equal_map) =
           Map.foldlWithKey (\cv@(cpm, cequal_map') nt' (Terminal t') ->
           if t == t' && nt /= nt'
           then (Map.delete nt' cpm, Map.insert nt' nt cequal_map')
           else cv) (pm, cequal_map) pm
        in eliminateEqualNTs' new_pm (cindex + 1) (Map.size new_pm)
         new_equal_map
       (DownStep nt1 nt2 nt3 nt4) -> let
         (new_pm, new_equal_map) =
         Map.foldlWithKey (\cv@(cpm, cequal_map') nt'
         (DownStep nt1' nt2' nt3' nt4') ->
           if nt1 == nt1' && nt2 == nt2' && nt3 == nt3'
           && nt4 == nt4' && nt /= nt'
           then (Map.delete nt' cpm, Map.insert nt' nt cequal_map')
           else cv) (pm, cequal_map) pm
        in eliminateEqualNTs' new_pm (cindex + 1) (Map.size new_pm)
          new_equal_map
```

Die eigentlichen Berechnungen finden in der Funktion `eliminateEqualNTs'` statt. Sie hat als Argumente die akutell betrachtete Map von Produktionen der Höhe k (Argument `pm`), den Index (die Indizierung der Map beginnt immer mit 0) des aktuell betrachteten Nichtterminals in der Map (Argument `cindex`), die aktuelle Anzahl der Produktionen (Argument `cmap_size`) und eine Map, deren Schlüssel die zu ersetzenden Nichtterminale in der Höhe $k+1$ sind und deren Werte die Schlüssel in der Höhe $k+1$ dann ersetzen (Argument `cequal_map`).

Die Ausgabe ist ein Tupel bestehend aus der Map von Produktionen der Höhe k, in der alle Nichtterminale verschiedene Matrizen abbilden, und der Map für die Ersetzungen in Höhe $k+1$.

Wie in 3.4.2 beschrieben gibt es zwei Fälle:

Fall 1: ist die rechte Seite `rhs` des akutell betrachteten Nichtterminals `nt` (also die linke Seite) der Produktion ein Terminal `t`, so ist man in Höhe 0 und es wird nur `t` mit allen anderen Terminalen `t'` dieser Höhe verglichen. Sind zwei Terminale `t` und `t'` gleich, so wird aus der aktuell betrachteten Map von Produktionen die Produktion gelöscht, wo `t'` als rechte Seite vorkam und das Nichtterminal `nt'`, welches die linke Seite der Produktion `nt'` $\rightarrow$ `t'` ist, in die Map `cequal_map` samt dem aktuell betrachteten Nichtterminal `nt` hinzugefügt.

Man beachte, dass man Terminale nur löscht, wenn die Nichtterminale `nt` und `nt'` ungleich sind. Dies ist nötig, da man mit der Funktion `Map.foldlWithKey` immer alle Elemente der Map durchgeht.

Der nächste Aufruf wird mit dem Erhöhen des Indexes des betrachteten Nichtterminals eingeleitet. Überschreitet oder ist dieser Index irgendwann gleich der Anzahl der Produktionen in der Map, so ist man hier fertig.

Fall 2: ist die rechte Seite `rhs` des akutell betrachteten Nichtterminals `nt` der Produktion eine DownStep-Produktion, so müssen stets nur die Nichtterminale (siehe Fall 2 in 3.4.2) der rechten Seiten verglichen werden. Analog zu Fall 1 werden aus der aktuell betrachteten Map von Produktionen die Produktionen gelöscht, die gleich der aktuell betrachteten Produktion sind, und deren Nichtterminale (`nt'`) werden in die Map `cequal_map` samt dem aktuell betrachteten Nichtterminal (`nt`) hinzugefügt.

Der nächste Aufruf wird analog zu Fall 1 mit dem Erhöhen des Indexes des betrachteten Nichtterminals eingeleitet.

4.3.3 Ersetzen von gleichen Nichtterminalen

Die Funktion `subEqualNTs` ersetzt in der Map von Produktionen `pm` der Höhe k die durch `eliminateEqualNTs` gefunden gleichen Nichtterminale der Höhe $k-1$, die in der Map `equal_map` zusammengefasst sind und liefert als Ergebnis die neue Map von Produktionen:

```
subEqualNTs :: ProdMap -> Map.Map NonTerminal NonTerminal
```

```
          -> ProdMap
subEqualNTs pm equal_map =
  Map.mapWithKey (\nt rhs -> checkDS rhs 1) pm where
  checkDS def@(DownStep nt1 nt2 nt3 nt4) i = let
    result
      | i == 1 = (\ntx -> (DownStep ntx nt2 nt3 nt4))
        (Map.findWithDefault nt1 nt1 equal_map)
      | i == 2 = (\ntx -> (DownStep nt1 ntx nt3 nt4))
        (Map.findWithDefault nt2 nt2 equal_map)
      | i == 3 = (\ntx -> (DownStep nt1 nt2 ntx nt4))
        (Map.findWithDefault nt3 nt3 equal_map)
      | i == 4 = (\ntx -> (DownStep nt1 nt2 nt3 ntx))
        (Map.findWithDefault nt4 nt4 equal_map)
   in if i > 4 then def else checkDS result (i + 1)
```

Es werden einfach alle Produktionen von `pm` durchlaufen und dabei wird für jedes Nichtterminal einer DownStep-Produktion geschaut, ob dieses Nichtterminal in `equal_map` auftaucht. Wenn dem so ist, so wird es durch den gefunden Wert in `equal_map` ersetzt, wenn es nicht der Fall ist, wird mit dem nächsten Nichtterminal fortgefahren.

4.3.4 Finden von Additionsproduktionen

Wie in 3.4.3 beschrieben, betrachtet man zwei Fälle beim Finden von Additionsproduktionen. Dazu wurden zwei von einander unabhängige Funktionen implementiert.

Fall 1: Finden von Additionsproduktionen in Produktionen der Höhe 1.

Die Funktion `checkAddOne` berechnet alle möglichen Additionsproduktionen der Höhe 1 bei Eingabe der Map von Produktionen der Höhe 1 und Höhe 0, die in einer Liste zusammengefasst sind:

```
checkAddOne :: [ProdMap] -> Map.Map NonTerminal [RightHandSide]
checkAddOne pms@(pm1:pm0:[ ]) =
  Map.foldlWithKey (\cmap nt_a rhs_a ->
    Map.foldlWithKey (\cmap2 nt_b rhs_b ->
      Map.foldlWithKey (\cmap3 nt_c rhs_c ->
        if nt_b <= nt_c
        then if nt_a == nt_b && nt_a == nt_c then cmap3 else
          if checkAddOne' rhs_a rhs_b rhs_c 1 pm0
          then Map.insertWith (++) nt_a [Addition nt_b nt_c] cmap3
          else cmap3
        else cmap3) cmap2 pm1) cmap pm1) Map.empty pm1 where
```

Die Map von Produktionen der Höhe 1, hier `pm1`, wird so durchlaufen, dass alle möglichen Kombinationen aus drei Nichtterminalen A, B und C betrachtet werden. Dabei macht

man die Einschränkung, dass zum einen nur die Additionen $A = B + C$ - und nicht $A = C + B$ - überprüft werden, zum anderen die Addition $A = A + A$ nicht in Betracht gezogen wird, die alle durch `checkAddOne'` überprüft werden.

Die Ausgabe von `checkAddOne` ist schließlich eine Map, in der die Schlüssel Nichtterminale sind, die auf alle möglichen Additionsproduktionen „zeigen", die in einer Liste zusammengefasst sind.

Die Unterfunktion `checkAddOne'` bekommt als Argumente drei rechte Seiten von DownStep-Produktionen `a`, `b` und `c`, eine Laufvariable `i` und die Map von Produktionen der Höhe 0 übergeben:

```
checkAddOne' a b c i pm0
    | (i < 1) || (i > 5) =
      error ("Error in 'checkAddOne': index out of range!")
    | i == 5 = True
    | otherwise = let
      a_i = getNTDSRule a i
      b_i = getNTDSRule b i
      c_i = getNTDSRule c i
      val_a_i = getValFromNTOne a_i pm0
      val_b_i = getValFromNTOne b_i pm0
      val_c_i = getValFromNTOne c_i pm0
      cond = val_a_i == (val_b_i + val_c_i)
     in if cond then checkAddOne' a b c (i + 1) pm0 else False
```

Es wird mit jedem Aufruf von `checkAddOne'` verglichen, ob sich die `i`. Terminale von `a`, `b` und `c` durch die Addition ausdrücken lassen (in der Variable `cond` geschieht das). Ist dies der Fall, so können mit den `i`+1. Terminalen fortgefahren werden. Es müssen sich demnach alle Terminale von `a`, `b` und `c` als Addition ausdrücken lassen, damit die Regel $A = B + C$ (siehe oben) zur Ausgabe-Map hinzugefügt werden kann.

Fall 2: Finden von Additionsproduktionen in Produktionen der Höhe $k > 1$.

Die Funktion `checkAdd` berechnet alle möglichen Additionsproduktionen der Höhe $k > 1$ bei Eingabe der Map von Produktionen der Höhe k und der Map von gefundenen Additionsproduktionen in der Höhe $k - 1$, die in einer Liste zusammengefasst sind:

```
checkAdd :: ProdMap -> Map.Map NonTerminal [RightHandSide]
        -> Map.Map NonTerminal [RightHandSide]
checkAdd pm_k rule_map =
  Map.foldlWithKey (\cmap nt_a rhs_a ->
    Map.foldlWithKey (\cmap2 nt_b rhs_b ->
      Map.foldlWithKey (\cmap3 nt_c rhs_c ->
        if nt_b <= nt_c
```

```
      then if checkAdd' rhs_a rhs_b rhs_c rule_map 1
        then Map.insertWith (++) nt_a [Addition nt_b nt_c] cmap3
        else cmap3
      else cmap3) cmap2 pm_k) cmap pm_k) Map.empty pm_k where
```

Die Unterfunktion `checkAdd'` bekommt als Argumente die rechten Seiten `a`,`b` und `c` von DownStep-Produktionen, die gefundenen Additionsproduktionen in der Höhe $k-1$ und die Laufvariable `i`.

Es wird mit jedem Aufruf von `checkAdd'` verglichen, ob sich die `i`. Nichtterminale von `a`, `b` und `c` durch die Addition ausdrücken lassen. Dazu schaut man sich an, ob es Additionsproduktionen für das `i` Nichtterminal von `a` und ob es eine Additionsproduktion `a_i` = `b_i` + `c_i` gibt (durch die Funktion `checkRuleList`), wobei diese Variablen jeweils den `i`. Nichtterminalen von `a`, `b` und `c` entsprechen. Kann für jedes `i` (von 1 bis 4) eine Additionsproduktion gefunden werden, sodass `a_i` = `b_i` + `c_i` gilt, so gilt auch `nt_a` = `nt_b` + `nt_c` (siehe `checkAdd`), wobei `nt_a`, `nt_b` und `nt_c` die linken Seiten von `a`, `b` und `c` sind.

4.3.5 Finden von Skalarmultproduktionen

Wie in 3.4.4 beschrieben, betrachtet man auch hier zwei Fälle beim Finden von Skalarmultproduktionen. Dazu wurden zwei von einander unabhängige Funktionen implementiert.

Fall 1: Finden von Skalarmultproduktionen in Produktionen der Höhe 1.

Die Funktion `checkSMOne` berechnet alle möglichen Skalarmultproduktionen der Höhe 1 bei Eingabe der Map von Produktionen der Höhe 1 und Höhe 0, die in einer Liste zusammengefasst sind:

```
checkAdd' a b c rule_map i
    | (i < 1) || (i > 5) =
      error ("Error in 'checkAdd': index out of range!")
    | i == 5 = True
    | otherwise = let
      a_i = getNTDSRule a i
      b_i = getNTDSRule b i
      c_i = getNTDSRule c i
      rule_list = case Map.lookup a_i rule_map of
        Just l -> l
        Nothing -> [ ]
     in if (checkRuleList rule_list b_i c_i)
       then checkAdd' a b c rule_map (i + 1)
       else False
```

```
checkRuleList [ ] _ _ = False
checkRuleList ((Addition nt_b nt_c):rls) b_i c_i =
  if nt_b == b_i && nt_c == c_i
  then True
  else checkRuleList rls b_i c_i
```

Die Map von Produktionen der Höhe 1, hier `pm1`, wird so durchlaufen, dass alle möglichen Kombinationen aus zwei Nichtterminalen A und B betrachtet werden. Somit werden alle möglichen Regeln $A = c \cdot B$ durch `checkSMOne'` überprüft. Diese gibt als Ergebnis das c in der Produktion (im Code `scal`) zurück. Das c darf dabei nicht 0 oder 1 sein, da diese Produktionen zu keiner Kompression führen!

Die Ausgabe von `checkSMOne` ist schließlich eine Map, in der die Schlüssel Nichtterminale sind, die auf alle möglichen Skalarmultproduktionen „zeigen“, die in einer Liste zusammengefasst sind.

Die Unterfunktion `checkSMOne'` bekommt als Argumente zwei rechte Seiten von DownStep-Produktionen `a` und `b`, den aktuell berechneten Skalar `scal`, eine Laufvariable `i`, einen Flag `flag` (wird weiter unten genau erklärt) und die Map von Produktionen der Höhe 0 übergeben:

```
checkSMOne' a b scal i flag pm0
  | (i < 1) || (i > 5) =
    error ("Error in 'checkSMOne': index out of range!")
  | i == 5 = scal
  | otherwise = let
    a_i = getNTDSRule a i
    b_i = getNTDSRule b i
    val_a_i = getValFromNTOne a_i pm0
    val_b_i = getValFromNTOne b_i pm0
    (scal', rest, flag') = if val_b_i == 0
      then if val_a_i == val_b_i
        then if i == 1 || flag == 1 then (scal, 0, 1) else (scal, 0, 0)
        else (0, 1, 0)
      else (\(x, y) -> (x, y, 0)) (divMod val_a_i val_b_i)
   in if rest == 0
      then if i < 2 then checkSMOne' a b scal' (i + 1) flag' pm0
        else
          if scal == scal' then checkSMOne' a b scal (i + 1) 0 pm0 else
            if flag == 1
            then checkSMOne' a b scal' (i + 1) flag' pm0 else 0
      else 0
```

In jedem Aufruf von `checkSMOne'` wird der ganzzahlige Quotient (mit Rest) des `i`. Terminals von a (im Code `val_a_i`) und von b (im Code `val_b_i`) berechnet (im Code

das Tupel (`scal', rest, flag'`)). Mit der Variable `flag` bzw. `flag'` merkt man sich, ob in allen vorigen Aufrufen nur Regeln $0 = c \cdot 0$ vorkamen, da es ja in diesem Fall unendlich viele Lösungen für c gibt und somit im nächsten Aufruf der berechnete Skalar keine Rolle spielt, man sich dies aber merken muss. Der `flag` wird in diesem Fall auf 1 gesetzt. Der Flag `flag` ist und bleibt dann 0, wenn eine eindeutige Lösung für c in einem Aufruf berechnet wurde.

Es gibt insgesamt folgende Fälle für eine Skalarmultproduktion $A \to c \cdot B$ im `i`. Aufruf von `checkSMOne'`, die berücksichtigt werden müssen:

- Ist $A = B = 0$, so gibt es für c unendlich viele Lösungen und es wird der Skalar aus dem `i-1`. Aufruf genommen, sonst merkt man sich mit dem `flag'`, dass es für c keine eindeutige Lösung gibt.
- Ist nur $B = 0$, so gibt es keine Lösung für c und man ist fertig mit der Überprüfung.
- Ansonsten wird der ganzzahlige Quotient aus A und B berechnet, wobei es diesen nur geben kann, wenn der Rest 0 ist. Sonst ist man auch hier mit der Überprüfung fertig.

Insgesamt muss in jedem Aufruf von `checkSMOne'` der selbe Skalar rauskommen, damit es eine Skalarmultproduktion geben kann.

Fall 2: Finden von Skalarmultproduktionen in Produktionen der Höhe $k > 1$.

Die Funktion `checkSM` berechnet alle möglichen Skalarmultproduktionen der Höhe $k > 1$ bei Eingabe der Map von Produktionen der Höhe k und der Map von gefundenen Skalarmultproduktionen in der Höhe $k - 1$, die in einer Liste zusammengefasst sind:

```
checkSM :: ProdMap -> Map.Map NonTerminal [RightHandSide]
        -> Map.Map NonTerminal [RightHandSide]
checkSM pm_k rule_map =
  Map.foldlWithKey (\cmap nt_a rhs_a ->
    Map.foldlWithKey (\cmap2 nt_b rhs_b -> let
      scal = checkSM' rhs_a rhs_b 1 rule_map 1
     in if scal /= 0 && scal /= 1
       then Map.insertWith (++) nt_a [ScalMult scal nt_b] cmap2
       else cmap2) cmap pm_k) Map.empty pm_k where
```

Die Map von Produktionen `pm_k` der Höhe k werden genauso wie in `checkSMOne` durchlaufen. Die Ausgabe ist dann die Map von gefundenen Skalarmultproduktionen der Höhe k.

Der Unterschied zu `checkSMOne` besteht in der Unterfunktion `checkSM'`:

```
checkSM' a b scal rule_map i
  | (i < 1) || (i > 5) =
    error ("Error in 'checkSM': index out of range!")
  | i == 5 = scal
  | otherwise = let
    a_i = getNTDSRule a i
    b_i = getNTDSRule b i
    rule_list = case Map.lookup a_i rule_map of
      Just l -> l
      Nothing -> [ ]
    (isScal, scal') = checkRuleList rule_list b_i
   in if isScal
     then if i < 2 then checkSM' a b scal' rule_map (i + 1)
       else
         if scal == scal'
         then checkSM' a b scal rule_map (i + 1) else 0
     else 0
checkRuleList [ ] _ = (False, 0)
checkRuleList ((ScalMult c nt_b):rls) b_i =
  if nt_b == b_i
  then (True, c)
  else checkRuleList rls b_i
```

In jedem Aufruf von `checkSM'` wird einfach überprüft, ob für die `i`. Nichtterminale der rechten Seiten von DownStep-Produktionen `a` und `b` Skalarmultproduktionen schon gefunden wurden. Ist dies der Fall, so muss sich immer der gleiche Skalar c bei jedem Aufruf von `checkSM'` für `a` und `b` ergeben damit dann `nt_a` $=$ `c` $\cdot$ `nt_b` (siehe `checkSM`) gilt, wobei `nt_a` und `nt_b` die linken Seiten von `a` und `b` sind.

4.3.6 Finden von Additionsproduktionen mit Hilfe von Differenzmatrizen

Wie in 3.4.5 beschrieben, betrachtet man auch hier schließlich zwei Fälle beim Finden von Additionsproduktionen mit Hilfe von Differenzmatrizen. Dazu wurden zwei von einander unabhängige Funktionen implementiert.

Fall 1: Finden von Additionsproduktionen mit Hilfe von Differenzmatrizen in Produktionen der Höhe 1.

Die Funktion `checkDiffMatOne` berechnet alle Additionsproduktionen mit Hilfe von Differenzmatrizen der Höhe 1 bei Eingabe der Map von Produktionen der Höhe 1 und Höhe 0, die in einer Liste zusammengefasst sind:

```
checkDiffMatOne :: [ProdMap] -> Map.Map NonTerminal [(NonTerminal, Int)]
checkDiffMatOne pms@(pm1:pm0:[ ]) =
```

```
  Map.foldlWithKey (\cmap nt_a rhs_a ->
    Map.foldlWithKey (\cmap2 nt_b rhs_b -> let
      (isConst, const) =
        checkDiffMatOne' rhs_a rhs_b 1 (False, False) 1 pm0
     in
      if isConst && const /= 0
      then let
       in Map.insertWith (++) nt_a [(nt_b, const)] cmap2
      else cmap2) cmap pm1) Map.empty pm1 where
```

Die Map von Produktionen der Höhe 1, hier `pm1`, wird so durchlaufen, dass alle möglichen Kombinationen aus zwei Nichtterminalen A und B betrachtet werden. Somit werden alle möglichen Regeln $A = B + \textit{konstante Matrix}$ durch `checkDiffMatOne'` überprüft. Diese gibt als Ergebnis ein Tupel `(isConst, const)` aus, wobei `isConst` angibt, ob die Überprüfung erfolgreich war und `const` angibt, aus welchem (einzigen) Eintrag dann die konstante Matrix besteht.

Die Ausgabe von `checkSMOne` ist schließlich eine Map, in der die Schlüssel Nichtterminale sind, die statt auf die rechten Seiten von Additionsproduktionen nun lediglich auf Tupel (B, Eintrag der konstanten Matrix) zeigen, die in einer Liste zusammengefasst sind.

Die konstanten Matrizen, die für die Additionsproduktionen nötig sind, werden erst beim tatsächlichen Ersetzen der gefundenen Produktion auch in das MTDD eingefügt (siehe 4.3.8).

Die Unterfunktion `checkDiffMatOne'` bekommt als Argumente zwei rechte Seiten von DownStep-Produktionen `a` und `b`, den aktuell berechneten konstanten Eintrag `const` einer möglichen konstanten Matrix, ein Tupel `(const_a, const_b)`, welches am Ende der Berechnung aussagt, ob `a` und/oder `b` Nullmatrizen sind, eine Laufvariable `i`, und die Map von Produktionen der Höhe 0 übergeben:

```
checkDiffMatOne' a b const (const_a, const_b) i pm0
  | (i < 1) || (i > 5) =
    error ("Error in 'checkDiffMatOne': index out of range!")
  | i == 5 = if const_a || const_b then (False, 0) else (True, const)
  | otherwise = let
    a_i = getNTDSRule a i
    b_i = getNTDSRule b i
    val_a_i = getValFromNTOne a_i pm0
    val_b_i = getValFromNTOne b_i pm0
    const' = val_a_i - val_b_i
    const_a' = const' == val_a_i
    const_b' = const' == val_b_i
   in if i < 2
     then checkDiffMatOne' a b const' (const_a', const_b') (i + 1) pm0
```

```
    else if const == const'
      then (checkDiffMatOne' a b const
        (const_a' && const_a, const_b' && const_b) (i + 1) pm0)
      else (False, 0)
```

In jedem Aufruf von `checkDiffMatOne'` wird die Differenz vom `i`. Nichtterminal von `a` und `b` berechnet. Damit die Differenz von `a` und `b` eine konstante Matrix ist, muss in jedem Aufruf von `checkDiffMatOne'` mit `a` und `b` stets der gleiche Wert `const` als Ergebnis geliefert worden sein.

Es wird in jedem Aufruf von `checkDiffMatOne'` auch berechnet, ob das `i`. Nichtterminal von `a` und `b` 0 ist:

- wenn $b_i = 0$, so ist $a_i =$ der berechnete konstante Wert
- wenn $a_i = 0$, so ist $b_i =$ der berechnete konstante Wert

Wie in dem Argument `(const_a' && const_a, const_b' && const_b)` zu sehen ist, wird die obere Überprüfung verkettet (es werden die bool'schen Werte einfach verkettet), um am Ende einer Berechnung durch `checkDiffMatOne'` den Fall abzufangen, dass `a` oder `b` eine Nullmatrix darstellen, denn eine Nullmatrix als Summe zweier Matrizen oder eine Nullmatrix auf eine konstante Matrix zu addieren, führt zu keiner Kompression!

Fall 2: Finden von Additionsproduktionen mit Hilfe von Differenzmatrizen in Produktionen der Höhe $k > 1$.

Die Funktion `checkDiffMat` berechnet alle möglichen Additionsproduktionen mit Hilfe von Differenzmatrizen der Höhe $k > 1$ bei Eingabe der Map von Produktionen der Höhe k und der Map von gefundenen „Additionsproduktionen“ mit Hilfe von Differenzmatrizen in der Höhe k-1, die in einer Liste zusammengefasst sind:

```
checkDiffMat :: ProdMap -> Map.Map NonTerminal [(NonTerminal, Int)]
             -> Map.Map NonTerminal [(NonTerminal, Int)]
checkDiffMat pm_k rule_map =
  Map.foldlWithKey (\cmap nt_a rhs_a ->
    Map.foldlWithKey (\cmap2 nt_b rhs_b -> let
      (isConst, const) = checkDiffMat' rhs_a rhs_b 1 rule_map 1
     in
      if isConst && const /= 0
      then let
       in Map.insertWith (++) nt_a [(nt_b, const)] cmap2
      else cmap2) cmap pm_k) Map.empty pm_k where
```

Die Map von Produktionen `pm_k` der Höhe k werden genauso wie in `checkDiffMatOne` durchlaufen.

Die Ausgabe ist dann die Map, in der die Schlüssel Nichtterminale sind, die auf Tupel (B, Eintrag der konstanten Matrix) zeigen, die in einer Liste zusammengefasst sind.

Der Unterschied zu `checkDiffMatOne` besteht in der Unterfunktion `checkDiffMat'`:

```
checkDiffMat' a b const rule_map i
  | (i < 1) || (i > 5) =
    error ("Error in 'checkDiffMat': index out of range!")
  | i == 5 = (True, const)
  | otherwise = let
    a_i = getNTDSRule a i
    b_i = getNTDSRule b i
    rule_list = case Map.lookup a_i rule_map of
      Just l -> l
      Nothing -> [ ]
    (isConst, const') = checkRuleList rule_list b_i
   in if i < 2 then checkDiffMat' a b const' rule_map (i + 1)
      else if const == const'
        then checkDiffMat' a b const rule_map (i + 1) else (False, 0)
checkRuleList [ ] _ = (False, 0)
checkRuleList ((nt_b, const):rls) b_i =
  if nt_b == b_i
  then (True, const)
  else checkRuleList rls b_i
```

In jedem Aufruf von `checkDiffMat'` wird einfach überprüft, ob für die `i`. Nichtterminale der rechten Seiten von DownStep-Produktionen `a` und `b` „Additionsproduktionen“ (also Tupel, siehe Eingabe von `checkDiffMat`) schon gefunden wurden. Ist dies der Fall, so muss sich immer der konstante Wert `const` bei jedem Aufruf von `checkDiffMat'` für `a` und `b` ergeben, damit dann `nt_a` = `nt_b` + *konstante Matrix* mit `const` als Eintrag (siehe `checkDiffMat`) gilt, wobei `nt_a` und `nt_b` die linken Seiten von `a` und `b` sind.

4.3.7 Ersetzen von DownStep-Produktionen durch Additions- und Skalarmultproduktionen

Die Funktion `subDownStep` ersetzt DownStep-Produktionen einer Map von Produktionen der Höhe k durch die gefundenen Additions- und Skalarmultproduktionen bei Eingabe von

- `pm`: Map von Produktionen der Höhe $k > 0$
- `rule_map`: Map mit gefundenen Additions- oder Skalarmultproduktionen der Höhe k (jeweils nur eine Art von Produktionen ist in der Map vertreten)
- `cyc_add_map`: Map von bisher eingefügten Additionsproduktionen zur Prüfung von Zyklen in `pm`

Die Ausgabe von `subDownStep` ist ein Tupel (a, b, c), wobei

- a: die neue Map von Produktionen mit den ersetzten DownStep-Produktionen
- b: die Map von allen eingesetzten Additionsproduktionen
- c : die Map von allen eingesetzten Skalarmultproduktionen

Nun zur Funktion selbst:

```
subDownStep pm rule_map cyc_add_map =
  Map.foldlWithKey (\(c_pm, cyc_add_map', cyc_sm_map') nt rhs_list
  -> case Map.lookup nt c_pm of
    Just (DownStep _ _ _ _) -> case head rhs_list of
      (Addition nt1 nt2) ->
      case checkRhsForCycle nt rhs_list cyc_add_map' of
        Just rhs -> (Map.adjust (\x -> rhs) nt c_pm,(\(Addition nt1' nt2')
          -> Map.insert nt (nt1', nt2') cyc_add_map') rhs, cyc_sm_map')
        Nothing -> (c_pm, cyc_add_map', cyc_sm_map')
      (ScalMult c nt') ->
      case checkRhsForCycle nt rhs_list cyc_add_map' of
        Just rhs -> (Map.adjust (\x -> rhs) nt c_pm, cyc_add_map',
          (\(ScalMult _ nt1') -> Map.insert nt nt1' cyc_sm_map') rhs)
        Nothing -> (c_pm, cyc_add_map', cyc_sm_map')
    Just _ -> (c_pm, cyc_add_map', cyc_sm_map')
    Nothing -> error ("Error in 'substituteDownStep': non-terminal "
     ++ show nt ++ " not found!")) (pm, cyc_add_map, Map.empty) rule_map
   where
```

Es wird die gesamte Map `rule_map` durchlaufen. Dabei betrachtet man in jeder Iteration die gefundenen Additions- oder Skalarmultproduktionen `nt` $\rightarrow$ `rhs_list`, wobei `rhs_list` eine Liste von rechten Seiten ist.

Es werden folgende Schritte unternommen:

- wurde `nt` noch nicht in `pm` ersetzt, die rechte Seite also immernoch die einer DownStep-Produktion ist, so wird zunächst mit `head rhs_list` geschaut, ob es sich um Additions- oder Skalarmultproduktionen handelt
- in beiden Fällen wird mit Hilfe von `checkRhsForCycle` überprüft, ob es eine Produktion `nt` $\rightarrow$ `rhs` (`rhs` ist eine rechte Seite aus `rhs_list`) gibt, die beim Einsetzen in die betrachtete Map von Produktionen keinen Zyklus erzeugt
- wird eine solche rechte Seite gefunden, so wird die DownStep-Produktion durch die Produktion `nt` $\rightarrow$ `rhs` ersetzt und man merkt sich die Ersetzung in der Map `cyc_add_map'` (was am Ende der Komponente b der Ausgabe entspricht), falls es sich um eine Additionsproduktion handelte bzw. in der Map `cyc_sm_map'` (was am Ende der Komponente c der Ausgabe entspricht), falls es sich um eine Skalarmultproduktion handelte. Man merkt sich die ersetzten Produktionen in verschiedenen

Maps, da ja die rechten Seiten von Additionsproduktionen 2, die von Skalarmultproduktionen 1 Nichtterminal enthalten und man beim Durchsuchen eben jener nach Zyklen diesen Unterschied irgendwie kenntlich machen sollte.

Die Unterfunktion `checkRhsForCycle` bekommt als Argument ein Nichtterminal `nt'`, dessen rechten Seiten und die Map von bisher eingesetzten Additionsproduktionen in `pm`:

```
checkRhsForCycle _ [ ] _ = Nothing
  checkRhsForCycle nt' (rhs@(ScalMult c nt1'):rhss) cycle_map' =
    if c == (-1) && nt' > nt1'
    then checkRhsForCycle nt' rhss cycle_map'
    else if checkCycle nt' nt1' cycle_map'
      then checkRhsForCycle nt' rhss cycle_map'
      else Just rhs
  checkRhsForCycle nt' (rhs@(Addition nt1' nt2'):rhss) cycle_map' =
    if nt' == nt1' || nt' == nt2' || checkCycle nt' nt1' cycle_map'
    then checkRhsForCycle nt' rhss cycle_map'
    else
      if checkCycle nt' nt2' cycle_map'
      then checkRhsForCycle nt' rhss cycle_map'
      else Just rhs
```

In dieser Funktion sucht man nach einer Additons- bzw. Skalarmultproduktion, die beim Einfügen keinen Zyklus erzeugen würde.

Es werden hier vor dem „richtigen“ Prüfen von Zyklen Fälle überprüft, wie

- $A \to (-1) \cdot B$ bzw. $B \to (-1) \cdot A$. Hierbei ist anzumerken, dass es sein kann, dass keiner der beiden Produktionen eingesetzt wird, wenn vorher die Produktion $A \to (-1) \cdot B$ einen Zyklus erzeugen und somit verworfen wird, da die Produktion $B \to (-1) \cdot A$ nie eingesetzt wird.
- $A \to A + B$ oder $A \to A + A$, da damit Nullmatrizen addiert werden und wie vorher schon erwähnt keine Kompression erzielt wird.

Tritt keiner dieser Fälle auf, wird mit Hilfe von `checkCycle` nach möglichen Zyklen gesucht:

```
checkCycle nt' curr_nt cycle_map' = case Map.lookup curr_nt cycle_map' of
  Just (nt1', nt2') -> if nt1' == nt' || nt2' == nt' then True
    else (checkCycle nt' nt1' cycle_map')
      || (checkCycle nt' nt2' cycle_map')
  Nothing -> False
```

Die Funktion `checkCycle` betreibt eine Tiefensuche derart, dass bei Eingabe eines Nichtterminals `nt'` und der Map von schon eingesetzten Additions- bzw. Skalarmultproduktionen einfach der durch die Produktionen entstehende Graph (die Knoten sind die Nichtterminale, die Kanten die Verbindung von linker und rechter Seite einer Produktion) traversiert wird. Tritt `nt'` dabei in einer rechten Seite von Produktionen auf, so gebe es hier einen Zyklus.

4.3.8 Ersetzen von DownStep-Produktionen durch Additionsproduktionen, die mit Hilfe von Differenzmatrizen gefunden wurden

Die Funktion `subDownStepConstMTDD` ersetzt DownStep-Produktionen einer Map von Produktionen der Höhe k durch die gefundenen Additionsproduktionen (welche durch die Berechnung von Differenzmatrizen ermittelt wurden) bei Eingabe von

- `prods`: Liste von Maps von Produktionen der Höhe bis $k > 0$
- `rule_map`: Map mit gefundenen Additionsproduktionen (mit konstanten MTDDs) der Höhe k
- `cf_map`: die Map mit den Häufigkeiten von konstanten MTDDs in `rule_map`
- `height`: die aktuelle Höhe
- `curr_nt`: das nächste freie Nichtterminal, welches in die bisherigen Produktionen eingesetzt werden kann
- `curr_const_mtdds`: die Map der in der Höhe $k-1$ eingesetzten konstanten MTDDs
- `(cyc_add_map, cyc_sm_map)`: Tupel aus der Map von bisher eingefügten Additions- und Skalarmultproduktionen zur Prüfung von Zyklen

Man braucht hier die Liste von allen Maps bis Höhe k, da man beim Einfügen eines konstanten MTDDs in alle Maps etwas einfügen muss.

Die Ausgabe von `subDownStepConstMTDD` ist ein Tupel(a, b, c), wobei

- a: die neue Map von Produktionen mit den ersetzten DownStep-Produktionen
- b: das nächste freie Nichtterminal, welches in der nächsten Höhe eingesetzt werden könnte
- c: die Map von allen eingesetzten konstanten MTDDs

Nun zur Funktion selbst, die nach den behandelten Fälle erklärt wird (den Typen der Funktion spart man sich hier):

```
subDownStepConstMTDD prods@(pm:pms) rule_map cf_map height curr_nt
  curr_const_mtdds (cyc_add_map, cyc_sm_map) =
    (\(d, e, f, _) -> (d, e, f))
```

Es wird zuallererst geschaut, ob die rechte Seite des aktuell betrachteten Nichtterminals `nt` schon ersetzt wurde:

```
(Map.foldlWithKey (\(c_pms, c_nt, c_const_mtdds, cyc_sm_map')
nt rhs_list -> case Map.lookup nt (head c_pms) of
  Just (DownStep _ _ _ _) ->
    case checkRhsList rhs_list c_const_mtdds of
```

Wenn dem so ist, wird mit Hilfe der Unterfunktion `checkRhsList` überprüft, ob `nt` mit Hilfe einer schon hinzugefügten konstanten MTDD dargestellt werden kann.

```
(Just nt_const, (nt_b, _)) ->
```

Ist auch dieser Fall eingetreten, wird im nächsten Schritt überprüft, ob das Einfügen solch einer Produktion zu einem Zyklus im $\text{MTDD}_{(+,\cdot)}$ führen würde:

```
if checkCycleAdd nt nt_b cyc_add_map || checkCycleSM nt nt_b cyc_sm_map'
then case checkRhsForCycle nt rhs_list cyc_add_map cyc_sm_map' of
```

Würde das Hinzufügen einer solchen Produktion zu einem Zyklus führen, so wird einfach eine Produktion (aus `rhs_list`) ausgewählt, die keinen Zyklus erzeugt (falls es überhaupt solch eine gibt). Dabei wird einfach das dafür notwendige konstante MTDD dem aktuellen $\text{MTDD}_{(+,\cdot)}$ hinzugefügt, auch wenn dieses schon in der $\text{MTDD}_{(+,\cdot)}$ sein könnte (die Nichtterminale in zwei gleichen konstanten MTDDs sind immer unterschiedlich):

```
Just (nt_b', const') -> let
  (MTDDPlusScal s constPMs, next_nt) =
    createConstMTDD height const' c_nt
  new_prods = reverse $ foldl (\c_pms' (pm1, pm2) ->
    (Map.union pm1 pm2) : c_pms') [ ] (zip c_pms constPMs)
 in (Map.adjust (\x -> (Addition nt_b' s)) nt (head new_prods) :
  (tail new_prods), next_nt, Map.insert const' s c_const_mtdds,
  Map.insert nt nt_b' cyc_sm_map')
```

Man fügt noch die eingefügte Produktion in die aktuelle Map von eingefügten Skalarmultproduktionen, da bei den rechten Seiten beider Produktionsarten nur ein Nichtterminal betrachtet werden muss (bei der Additionsproduktion, wo ein Summand ein konstantes MTDD repräsentiert, fällt die Betrachtung dieses Nichtterminals weg, da es mit diesem keine Zyklen geben kann, denn die Nichtterminale in einem konstanten MTDD tauchen nur in diesem auf).

Gibt es keine Produktionen, die beim Einfügen keinen Zyklus erzeugt, so wird einfach

das Nächste Nichtterminal betrachtet (indem, hier nun nichts verändert wird):

```
Nothing -> (c_pms, c_nt, c_const_mtdds, cyc_sm_map')
```

Hier wird der Fall betrachtet, wenn man die Produktionen keinen Zyklus erzeugt und somit problemlos den akutellen Maps von Produktionen hinzugefügt werden kann:

```
else (Map.adjust (\x -> (Addition nt_b nt_const)) nt (head c_pms) :
  (tail c_pms), c_nt, c_const_mtdds, Map.insert nt nt_b cyc_sm_map')
```

Wurde kein konstantes MTDD der Höhe k gefunden, das man für das aktuell betrachtete Nichtterminal `nt` hätte verwenden können, so wird nach konstanten MTDDs der Höhe $k-1$ geschaut, die in der aktuellen Maps von Produktionen hinzugefügt wurden:

```
(Nothing, _) -> case checkRhsList rhs_list curr_const_mtdds of
  (Just nt_const, (nt_b, const)) ->
```

Würde das benutzten solch eines MTDDs der Höhe $k-1$ zu einem Zyklus führen, so wird wieder einfach eine Produktion (aus `rhs_list`) ausgewählt, die keinen Zyklus erzeugt (falls es überhaupt solch eine gibt). Dabei wird einfach das dafür notwendige konstante MTDD dem aktuellen $\text{MTDD}_{(+,\cdot)}$ hinzugefügt, auch wenn dieses schon in der $\text{MTDD}_{(+,\cdot)}$ sein könnte:

```
if checkCycleAdd nt nt_b cyc_add_map || checkCycleSM nt nt_b cyc_sm_map'
then case checkRhsForCycle nt rhs_list cyc_add_map cyc_sm_map' of
  Just (nt_b', const') -> let
    (MTDDPlusScal s constPMs, next_nt) =
      createConstMTDD height const' c_nt
    new_prods = reverse $ foldl (\c_pms' (pm1, pm2) ->
      (Map.union pm1 pm2) : c_pms') [ ] (zip c_pms constPMs)
   in (Map.adjust (\x -> (Addition nt_b' s)) nt (head new_prods) :
     (tail new_prods), next_nt, Map.insert const' s c_const_mtdds,
     Map.insert nt nt_b' cyc_sm_map')
```

Gibt es keine Produktionen, die beim Einfügen keinen Zyklus erzeugt, so wird einfach das nächste Nichtterminal betrachtet (indem, hier nun nichts verändert wird):

```
Nothing -> (c_pms, c_nt, c_const_mtdds, cyc_sm_map')
```

Hat man ein konstantes MTDD der Höhe $k-1$ gefunden, welches aus dem gleichen (einzigen) Eintrag aufgebaut ist, der sich auch in einer Produktion von `nt` in `rhs_list` findet, so genügt es, nur eine DownStep-Produktion statt eines ganzen MTDDs hinzuzufügen:

```
else let
  new_prod = Map.insert c_nt
    (DownStep nt_const nt_const nt_const nt_const) (head c_pms)
```

```
 in (Map.adjust (\x -> (Addition nt_b c_nt)) nt new_prod : (tail c_pms),
   c_nt + 1, Map.insert const c_nt c_const_mtdds,
   Map.insert nt nt_b cyc_sm_map')
```

Wurde auch hier nichts gefunden, so wird ein komplett neues konstantes MTDD erzeugt, welches nicht in den aktuell betrachtenen Maps von Produktionen auftaucht:

```
(Nothing, _) -> let
  (nt_b, const, _) = foldl (\c_val@(_, _, c_best) (nt_b', const') -> let
  new_best = case Map.lookup const' cf_map of
    Just i -> i
    Nothing -> 0
 in if new_best > c_best then (nt_b', const', new_best) else c_val)
   (0, 0, 0) rhs_list
```

Dabei wird mit Hilfe von `cf_map`, das konstante MTDD erzeugt, welches auch am häufigsten beim Finden dieser Additionsproduktionen verwendet wurde.

Tritt auch hier wieder der Fall ein, dass solch eine Produktion zu einem Zyklus führen würde, so wird einfach die nächst beste Produktion aus `rhs_list` ausgewählt, die keinen Zyklus erzeugt und dem aktuellen $\text{MTDD}_{(+,\cdot)}$ hinzugefügt:

```
in if checkCycleAdd nt nt_b cyc_add_map
   || checkCycleSM nt nt_b cyc_sm_map'
  then case checkRhsForCycle nt rhs_list cyc_add_map cyc_sm_map' of
    Just (nt_b', const') -> let
      (MTDDPlusScal s constPMs, next_nt) =
        createConstMTDD height const' c_nt
       new_prods = reverse $ foldl (\c_pms' (pm1, pm2) ->
         (Map.union pm1 pm2) : c_pms') [ ] (zip c_pms constPMs)
      in (Map.adjust (\x -> (Addition nt_b' s)) nt (head new_prods) :
        (tail new_prods), next_nt, Map.insert const' s c_const_mtdds,
        Map.insert nt nt_b' cyc_sm_map')
```

Gibt es wieder keine Produktionen, die beim Einfügen keinen Zyklus erzeugt, so wird einfach das nächste Nichtterminal betrachtet (indem, hier nun nichts verändert wird):

```
Nothing -> (c_pms, c_nt, c_const_mtdds, cyc_sm_map')
```

Gab es beim obigen Fall keinen Zyklus, so kann die neue Produktion bedenkenlos eingefügt werden:

```
else let
  (MTDDPlusScal s constPMs, next_nt) =
    createConstMTDD height const c_nt
```

```
  new_prods = reverse $ foldl (\c_pms' (pm1, pm2) ->
    (Map.union pm1 pm2) : c_pms') [ ] (zip c_pms constPMs)
 in (Map.adjust (\x -> (Addition nt_b s)) nt (head new_prods) :
   (tail new_prods), next_nt, Map.insert const s c_const_mtdds,
   Map.insert nt nt_b cyc_sm_map')
```

Der nächste Fall tritt ein, wenn die Produktion von `nt` schon ersetzt wurde:

```
Just _ -> (c_pms, c_nt, c_const_mtdds, cyc_sm_map')
```

Dies ist nur eine Fehlerausgabe, für den Fall, dass das aktuelle betrachtete Nichtterminal `nt` nicht im $\text{MTDD}_{(+,\cdot)}$ vertreten sein sollte (aus welchen Gründen auch immer):

```
Nothing -> error ("Error in 'substituteDownStep': non-terminal "
  ++ show nt ++ " not found!"))
```

Die nachfolgenden Variablen sind die „Startwerte“ für das `Map.foldlWithKey` ganz am Anfang der Funktion:

```
(prods, curr_nt, Map.empty, cyc_sm_map) rule_map) where
```

Wie auch in der Funktion `subDownStep` gibt es auch hier eine Unterfunktion `checkRhsForCycle`, die die Maps von eingesetzten Additions- und Skalarmultproduktionen nach Zyklen durchsucht (die nur beim Einfügen einer Produktion, wo `nt` die linke Seite dieser darstellt, entstehen könnte):

```
checkRhsForCycle _ [ ] _ _ = Nothing
checkRhsForCycle nt' (rhs@(nt_b', _):rhss) c_add_map c_sm_map =
  if checkCycleAdd nt' nt_b' c_add_map
  then checkRhsForCycle nt' rhss c_add_map c_sm_map
  else if checkCycleSM nt' nt_b' c_sm_map
    then checkRhsForCycle nt' rhss c_add_map c_sm_map
    else Just rhs
checkCycleAdd nt' curr_nt c_add_map' =
case Map.lookup curr_nt c_add_map' of
  Just (nt1', nt2') -> if nt1' == nt' || nt2' == nt' then True
    else (checkCycleAdd nt' nt1' c_add_map')
        || (checkCycleAdd nt' nt2' c_add_map')
  Nothing -> False
checkCycleSM nt' curr_nt c_sm_map' =
case Map.lookup curr_nt c_sm_map' of
  Just nt1' -> if nt1' == nt' then True
    else checkCycleSM nt' nt1' c_sm_map'
  Nothing -> False
```

Die Unterfunktion `checkCycleAdd` sucht dabei nach (möglichen) Zyklen in der Map von bisher eingesetzten Additionsproduktionen, `checkCycleSM` tut dies bei der Map von bisher eingesetzten Skalarmultproduktionen.

Die Unterfunktion `checkRhsList` dient dazu zu prüfen, ob in der Liste von rechten Seiten `rhs_list` die benötigten konstanten MTDDs schon in einer Map `const_mtdds` von bisher eingesetzten konstanten MTDDs vorhanden sind und gibt die notwendigen Informationen aus:

```
checkRhsList [ ] _ = (Nothing, (0, 0))
checkRhsList (rhs:rhss) const_mtdds = let
  (nt_b, const) = rhs
 in case Map.lookup const const_mtdds of
   Just nt_const -> (Just nt_const, (nt_b, const))
   Nothing -> checkRhsList rhss const_mtdds
```

4.3.9 Löschen von nicht-referenzierten Nichtterminalen

Die Funktion `checkNonRefNTs` löscht alle nicht-referenzierten (oder benötigten) Nichtterminale eines $\text{MTDD}_{(+,\cdot)}$ bei Eingabe der Liste von Maps von Produktionen und des ersten Nichtterminals, welches durch die Hinzunahme von „konstanten MTDDs“ in die Maps von Produktionen hinzugefügt wurde.

Die Ausgabe ist dann die neue Liste von Maps von Produktionen:

```
checkNonRefNTs :: [ProdMap] -> NonTerminal -> [ProdMap]
checkNonRefNTs prods curr_nt =
  checkNonRefNTs' prods (Map.singleton 0 1) where
  checkNonRefNTs' [ ] _ = [ ]
  checkNonRefNTs' (pm:pms) curr_nt_map = let
    new_pm = Map.foldlWithKey (\c_pm nt _ ->
    case Map.lookup nt curr_nt_map of
      Just _ -> c_pm
      Nothing -> if nt < curr_nt && nt > 0
        then Map.delete nt c_pm else c_pm) pm pm
    new_nt_map = Map.foldlWithKey (\c_map _ rhs -> case rhs of
      (DownStep nt1 nt2 nt3 nt4) -> Map.insert nt4 1
        $ Map.insert nt3 1 $ Map.insert nt2 1 $ Map.insert nt1 1 c_map
      _ -> c_map) Map.empty new_pm
   in new_pm : (checkNonRefNTs' pms new_nt_map)
```

Man geht die Maps von Produktionen eines $\text{MTDD}_{(+,\cdot)}$ der Höhe h von Höhe h bis Höhe 0 durch. Dabei merkt man sich in Höhe $k > 0$ in der Map `curr_nt_map`, welche Nichtterminale sich in dieser befinden - genauer: alle Nichtterminale der DownStep-Produktionen der Höhe k - und schaut dann in Höhe $k-1$ nach, ob alle Nichtterminale

auch in `curr_nt_map` vorkommen. Wenn dem nicht so ist, kann das entsprechende Nichtterminal gelöscht werden.

Damit nun Nichtterminale, die durch das Ersetzen von Additionsproduktionen, die mit Hilfe von Differenzmatrizen gefunden und hinzugefügt wurden, nicht gelöscht werden (da diese in den DownStep-Produktionen nicht vorkommen), werden genau diese Nichtterminale nicht berücksichtigt. Dies geschieht, indem man nur Nichtterminale `nt` $<$ `curr_nt` betrachtet, wobei `curr_nt`, das erste Nichtterminale ist, welches durch Hinzufügen von konstanten MTDDs seinen Weg in ein $\text{MTDD}_{(+,\cdot)}$ fand.

4.3.10 Beispiel: Komprimieren mit der Funktion `compressMat0123`

Es folgt nun eine schrittweise Erklärung der Funktion `compressMat0123`, die eine Matrix komprimiert, indem

- alle gleichen Nichtterminale eliminiert werden
- Additionsproduktion (beide „Arten“) und
- Skalarmultproduktionen

ersetzt werden.

Die Funktion `compressMat0123` komprimiert eine Matrix vom Typ `Mat.Matrix Int` als $\text{MTDD}_{(+,\cdot)}$:

```
compressMat0123 :: Mat.Matrix Int -> MTDDPlusScal
compressMat0123 mat = let
  prods = prepareMat mat
  height = length prods - 1
  next_nt = (foldl (\s i -> s + (4^i) * 5) 0 $ (take height $ [0..])) + 1
  compressed_prods = compressMTDD0123 0 prods Map.empty
    (Map.empty, Map.empty, Map.empty) next_nt Map.empty [ ]
  final_prods = checkNonRefNTs compressed_prods next_nt
 in MTDDPlusScal 0 final_prods
```

Es wird zunächst das vollständige MTDD aus der Matrix erzeugt, dieses anschließend komprimiert und dann werden schließlich alle überflüssigen Nichtterminale des $\text{MTDD}_{(+,\cdot)}$ gelöscht.

Die Funktion `prepareMat` erwartet eine Matrix und gibt als Resultat das entsprechende vollständige MTDD in Form der Liste von Maps von Produktionen zurück:

```
prepareMat :: Mat.Matrix Int -> [ProdMap]
prepareMat mat = let
  height = case testMatrix mat of
    Just h -> h
```

```
    Nothing -> error "The input matrix has wrong format!"
  prods = createCompMTDD mat height
 in reverse prods
```

Mit Hilfe der Funktion `testMatrix` (siehe im Code in der Datei „MatrixCompression.hs“) wird überprüft, ob die eingegebene Matrix eine quadratische Matrix ist, deren Dimensionen Zweierpotenzen sein müssen.

Die Funktion `compressMTDD0123` hat als Argumente

- die aktuell betrachtete Höhe k des $\text{MTDD}_{(+,\cdot)}$
- die aktuell betrachteten Maps von Produktionen bis (einschließlich) Höhe k
- die Map, die zum Ersetzen von gleichen Nichtterminalen gebraucht wird
- das Tupel (a, b, c), wobei
  - a: Map der gefundenen Additionsproduktionen der Höhe $k-1$
  - b: Map der gefundenen Skalarmultproduktionen der Höhe $k-1$
  - c: Map der gefundenen Additionsproduktionen der Höhe $k-1$, die mit Hilfe von Differenzmatrizen gefunden wurden
- das freie Nichtterminal, welches zum Einsetzen von konstanten MTDDs zur Verfügung steht
- die bisher eingefügten konstanten MTDDs der Höhe $k-1$

und als Ausgabe das komprimierte $\text{MTDD}_{(+,\cdot)}$ als Liste von Maps von Produktionen (den Typen der Funktion spart man sich hier):

```
compressMTDD0123 h [ ] _ _ _ _ pms_c = pms_c
compressMTDD0123 h (pm:pms) eq_map rule_maps curr_nt
curr_const_mtdds pms_c = let
  pm_subst = if eq_map == Map.empty then pm
    else subEqualNTs pm eq_map
  (pm_c, new_eq_map) = if h > 0
    then if Map.size eq_map > 3
      then eliminateEqualNTs pm_subst
      else (pm_subst, Map.empty)
    else eliminateEqualNTs pm_subst
```

Hier werden zunächst in der aktuellen Map von Produktionen `pm` der Höhe k alle gleichen Nichtterminale ersetzt. Anschließend wird dann in `pm` nach Gleichheiten gesucht, aber auch nur dann, wenn es vorher mindestens 4 Ersetzungen gab (siehe 3.4.2).

```
(add_rule_map, sm_rule_map, diff_rule_map) = rule_maps
new_add_rule_map = if h == 1
  then checkAddOne (pm_c : (head pms_c) : [ ])
  else if h < 1 then Map.empty else checkAdd pm_c add_rule_map
new_sm_rule_map = if h == 1
  then checkSMOne (pm_c : (head pms_c) : [ ])
  else if h < 1 then Map.empty else checkSM pm_c sm_rule_map
new_diff_rule_map = if h == 1
  then checkDiffMatOne (pm_c : (head pms_c) : [ ])
  else if h < 1 then Map.empty else checkDiffMat pm_c diff_rule_map
```

Es wird dann nacheinander nach

- Additionsproduktionen (beider „Arten“) und
- Skalarmultproduktionen

gesucht und die Gefundenen werden in den entsprechenden Maps gespeichert.

```
(new_pm_c, cyc_add_map, _) = if h > 0
  then subDownStep pm_c new_add_rule_map Map.empty
  else (pm_c, Map.empty, Map.empty)
(new_pm_c2, _, cyc_sm_map)= if h > 0
  then subDownStep new_pm_c new_sm_rule_map cyc_add_map
  else (new_pm_c, Map.empty, Map.empty)
(new_pms_c, next_nt, next_const_mtdds) = if h > 0
  then let
    cf_map = constFrequency new_diff_rule_map
   in subDownStepConstMTDD (new_pm_c2 : pms_c) new_diff_rule_map
     cf_map h curr_nt curr_const_mtdds (cyc_add_map, cyc_sm_map)
  else (new_pm_c2 : pms_c, curr_nt, curr_const_mtdds)
```

Es werden hier die gefundenen Additionsproduktionen (beide „Arten“) und Skalarmultproduktionen eingesetzt. Beim Ersetzen von Additionsproduktionen, die mit Hilfe von Differenzmatrizen gefunden wurden, wird die Map `cf_map` berechnet, die aussagt, wie oft eine konstante MTDD in diesen gefundenen Produktionen vorkam:

```
constFrequency :: Map.Map NonTerminal [(NonTerminal, Int)]
              -> Map.Map Int NonTerminal
constFrequency const_map =
  Map.foldl (\c_cf_map rhs_list ->
    foldl (\c_cf_map' (_, const) ->
    case Map.lookup const c_cf_map' of
      Just _ -> Map.adjust (\x -> x + 1) const c_cf_map'
      Nothing -> Map.insert const 1 c_cf_map') c_cf_map rhs_list)
        Map.empty const_map
```

Man berechnet aus den gefunden Additionsproduktionen (mit konstanten MTDDs) die Häufigkeit von konstanten MTDDs in diesen. Dies hat den Sinn, dass man nur die konstanten MTDDs in das $\text{MTDD}_{(+,\cdot)}$ einfügt, die auch oft in den Additionsproduktionen vorkamen, um damit die größtmögliche Kompression zu erzielen.

Man hat am Ende eine Map, in der die Schlüssel dem konstanten Eintrag und die Werte dem Startsymbol des konstanten MTDDs entspricht.

Zum Schluss wird vor jedem erneuten Aufruf mit der nächsten Höhe überprüft, ob es überhaupt gleiche Nichtterminale in der Map von Produktionen, mögliche Additions- oder Skalarmultproduktionen oder Additionsproduktionen (die durch das Berechnen von Differenzmatrizen gefunden wurden) gab. Wenn absolut nichts ersetzt/gefunden wurde, kann das ganze Verfahren abgebrochen werden. Im anderen Fall macht es noch Sinn zu komprimieren:

```
in if h > 1
   then if Map.size new_eq_map > 0 || Map.size new_add_rule_map > 0
     || Map.size new_sm_rule_map > 0 || Map.size new_diff_rule_map > 0
     then compressMTDD0123 (h + 1) pms new_eq_map
       (new_add_rule_map, new_sm_rule_map, new_diff_rule_map)
       next_nt next_const_mtdds new_pms_c
     else (reverse pms) ++ new_pms_c
   else compressMTDD0123 (h + 1) pms new_eq_map
     (new_add_rule_map, new_sm_rule_map, new_diff_rule_map)
     next_nt next_const_mtdds new_pms_c
```

4.3.11 Dekomprimieren eines $\text{MTDD}_{(+,\cdot)}$

Die Funktion `decompressMTDDPlusScal` dekomprimiert ein $\text{MTDD}_{(+,\cdot)}$ bei dessen Eingabe und gibt die entsprechende Matrix als Resultat aus:

```
decompressMTDDPlusScal :: MTDDPlusScal -> Mat.Matrix Int
decompressMTDDPlusScal (MTDDPlusScal s pms) =
  decompressMTDDPlusScal' s pms where
  decompressMTDDPlusScal' nt prods@(pm:pms) =
  case Map.lookup nt pm of
    Just rhs -> case rhs of
      DownStep nt1 nt2 nt3 nt4 ->
        let
          nt1_mat = decompressMTDDPlusScal' nt1 pms
          nt2_mat = decompressMTDDPlusScal' nt2 pms
          nt3_mat = decompressMTDDPlusScal' nt3 pms
          nt4_mat = decompressMTDDPlusScal' nt4 pms
        in Mat.joinBlocks (nt1_mat, nt2_mat, nt3_mat, nt4_mat)
```

```
    Addition nt1 nt2 ->
      let
        nt1_mat = decompressMTDDPlusScal' nt1 prods
        nt2_mat = decompressMTDDPlusScal' nt2 prods
      in Mat.elementwise (+) nt1_mat nt2_mat
    ScalMult c nt1 ->
      let
        nt1_mat = decompressMTDDPlusScal' nt1 prods
      in Mat.scaleMatrix c nt1_mat
    Terminal t -> Mat.matrix 1 1 (\(i,j) -> t)
  Nothing -> error ("Error while decompressing: non-terminal "
    ++ show nt ++ " not found!")
```

Es wird hier der Einfachheit halber jedes Nichtterminal gemäß 3.5 dekomprimiert, aber es wird ein und das selbe Nichtterminal mehrmals dekomprimiert wird, weil man sich hier die schon berechneten Matrizen nicht merkt, was zwar zu Lasten der Laufzeit ist, der Code dadurch aber viel enfacher und „intuitiver“ ist.

Außerdem wird ein Fehler ausgegeben, sollte zu einem Nichtterminal die Produktion fehlen – aus welchen Gründen auch immer.

4.4 Das Modul `IO_Datatypes`

Das Modul `IO_Datatypes` wurde mit dem Zweck implementiert, sowohl Matrizen als auch $\text{MTDD}_{(+,\cdot)}$s in Dateien zu schreiben und aus Dateien zu lesen.

Dazu wurden folgende Funktionen implementiert:

- `writeMatrix :: FilePath -> Mat.Matrix Int -> IO()`: die Funktion `writeMatrix` bekommt als Argument einen Pfad (inklusive des Dateinamens und der Dateiendung) als String, eine Matrix vom Typ `Mat.Matrix Int` und schreibt die Matrix in die entsprechende Datei.
- `readMatrix :: FilePath -> IO (Mat.Matrix Int)`: die Funktion `readMatrix` bekommt als Argument einen Pfad (inklusive des Dateinamens und der Dateiendung) als String und liefert die Matrix des Typs `Mat.Matrix Int`.
- `writeMTDD :: FilePath -> MTDDPlusScal -> IO ()`: die Funktion `writeMTDD` bekommt als Argument einen Pfad (inklusive des Dateinamens und der Dateiendung) als String, ein $\text{MTDD}_{(+,\cdot)}$ vom Typ `MTDDPlusScal` und schreibt die $\text{MTDD}_{(+,\cdot)}$ in die entsprechende Datei.
- `readMTDD :: FilePath -> IO MTDDPlusScal`: die Funktion `readMTDD` bekommt als Argument einen Pfad (inklusive des Dateinamens und der Dateiendung) als String und liefert das $\text{MTDD}_{(+,\cdot)}$ des Typs `MTDDPlusScal`.

Wie die Ausgaben genau aussehen, findet man im Unterordner „out“. Dabei enthalten Dateien mit der Endung „.mat“ Matrizen und Dateien mit der Endung „.mtdd“ $\text{MTDD}_{(+,\cdot)}$s.

Anzumerken ist noch, dass es keine Überprüfung beim Lesen und Schreiben der Dateien gibt, die den Inhalt und Form der Datei als korrekt bestätigt.

4.5 Das Modul `TestMatrixCompression`

Im Modul `TestMatrixCompression` befinden sich zum einen Funktionen, die bestimmte Matrizen erzeugen und zum anderen Funktionen, mit denen man dann die Performance und Kompressionsraten getestet hat.

Es werden hier nur die Funktionen zum Erzeugen spezieller Matrizen vorgestellt. Der Rest wird im nächsten Kapitel näher beleuchtet.

4.5.1 Test-Matrizen

Die folgenden Funktionen erstellen die zu testenden Matrizen:

- die Funktion `matConst h entry` erzeugt eine konstante Matrix der Höhe `h` und den Einträgen `entry`
- die Funktion `walsh h` erzeugt die *Walsh-Matrix* (siehe [YMSS]) der Höhe `h`. Sie ist definiert durch

$$W(1) = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \text{ und } W(h) = \begin{pmatrix} W(h-1) & W(h-1) \\ W(h-1) & -W(h-1) \end{pmatrix}.$$

- die Funktion `matEntryEqRow h` erzeugt eine Matrix der Höhe `h`, in der die Einträge dem entsprechenden Zeilenindex entspricht. Beispiel:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix}$$

- die Funktion `matLinear h` erzeugt eine Matrix der Höhe `h`, die lineare Folgen in ihren Teilmatrizen enthält. Beispiel:

$$A = \begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{pmatrix}$$

- die Funktion `matTriangular h entry` erzeugt eine Dreiecks-Matrix der Höhe `h` und den Einträgen `entry` derart, dass sie unterhalb der Diagonalen und der Diagonalen selbst die Einträge `entry` hat und sonst überall 0 ist

- die Funktion `matTest1 h` erzeugt eine Matrix M der Höhe `h` mit $M = concat(M_1, M_1, M_2, M_3)$ und M_1 = eine durch `matLinear`, M_2 = eine durch `matTriangluar` erzeugte Matrix und M_3 = die Walsh-Matrix ist.

Kapitel 5

Tests und Ergebnisse

Dieses Kapitel widmet sich nun den Perforamance-Tests, die auch kurz kommentiert werden.

5.1 Das Testsystem

Getestet wurde der Code auf einem HP EliteBook 840 G1 (Notebook) mit folgender Hardware

- CPU: Intel Core i5 4200U mit 1.6 GHz (2.6 GHz) und 3 MB Cache
- RAM: 4GB DDR3L SDRAM 1600 Mhz
- OS: Windows 7 Pro 64bit

und dem GHCi 7.10.3 (siehe [ghc]).

5.2 Tests

Die Laufzeiten der Kompressionsfunktionen wurden ermittelt, indem der GHCi mit den Parametern (siehe [par])

- `+RTS`: aktiviert die Parametereingabe (über die Konsole) für das GHCi Laufzeitsystem
- `-H1G`: setzt die maximale Größe des verwendeten Heaps auf 1 GB
- `-A10m`: erhöht die Größe des Allozierungsbereichs, der vom Garbage Collector benutzt wird

aufgerufen wurde und die Interpreter interne Zeitmessung mit dem Befehl „`:set +s`“ aktiviert wurde.

Die Begrenzung des Heaps ist notwendig, da die Kompressionsfunktionen einen hohen Speicherverbrauch ab einer bestimmten Größe (Höhe) von Matrizen zeigten. Somit ist schon jetzt klar, dass die Funktionen eine bessere Laufzeit hätten, würde immer ausreichend RAM zur Verfügung gestellt werden.

Um bei der Messung von Laufzeiten der Kompressionsfunktionen die Ausgabe auf der Konsole zu unterdrücken, wurden einfach die Größen der einzelnen Maps von Produktionen des komprimierten $\mathrm{MTDD}_{(+,\cdot)}$ aufsummiert (die Größe einer Map wird in konstanter Laufzeit ermittelt), damit auch jedes Element der Liste von Maps auch tatsächlich berechnet wird (bei der Berechnung der Länge einer Liste beispielsweise würden die Elemente der Liste nicht weiter ausgewertet).

Ein Beispiel für solch eine „Testfunktion“ sieht man hier:

```
test1 h = let
  mat = matConst h 1
  (MTDDPlusScal _ prods) = compressMat0 mat
 in foldl (\i pm -> i + Map.size pm) 0 prods
```

Die Kompressionsraten zwischen dem komprimierten $\text{MTDD}_{(+,\cdot)}$ und vollständigen MTDD wurden bespielsweise durch

```
testRMTDD1 h = let
  mat = matConst h 1
 in compressionRateCompMTDD $ compressMat0 mat
```

zwischen dem komprimierten $\text{MTDD}_{(+,\cdot)}$ und der Matrix durch

```
testRM1 h = let
  mat = matConst h 1
 in compressionRateMat $ compressMat0 mat
```

ermittelt.

5.3 Ergebnisse

Es folgen nun tabellarisch die Laufzeit- und Kompressionsergebnisse für die in 4.5.1 erwähnten Testmatrizen. Die Laufzeiten wurden immer drei Mal gemessen und dann der Durchschnitt als Endergebnis genommen.

Die Tabellen erfassen

- die Höhe der zu komprimierenden Matrix
- die dafür benötigte Zeit in Sekunden
- die Kompressionsrate zwischen komprimiertem $\text{MTDD}_{(+,\cdot)}$ und vollständigem MTDD (unter c-rate-mtdd) in Prozent
- die Kompressionsrate zwischen komprimiertem $\text{MTDD}_{(+,\cdot)}$ und der entsprechenden Matrix (unter c-rate-mat) in Prozent

Ist die Prozentzahl dabei negativ, so bedeutet dies nur, dass das $\text{MTDD}_{+,\cdot)}$ um den Betrag der Prozentzahl gewachsen ist.

Dauerte die Laufzeit der Komprimierungen länger als 5 min, so wurden diese vorzeitig abgebrochen und sind somit hier nicht enthalten.

Test 1 - `matConst` komprimiert mit `compressMat0`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	78,95	25
3	0,01	92,7	73,44
4	0,02	97,65	91,41
5	0,04	99,28	97,36
6	0,08	99,79	99,22
7	0,3	99,94	99,77
8	1,2	99,98	99,94
9	6,6	99,99	99,98
10	29	99,99	99,99

Test 2 - `walsh` komprimiert mit `compressMat0`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	66,67	-18,75
3	0,01	87,55	54,69
4	0,02	95,84	84,77
5	0,04	98,69	95,22
6	0,08	99,61	98,56
7	0,3	99,89	99,58
8	1,5	99,97	99,88
9	6,8	99,99	99,97
10	30	99,99	99,99

Test 3 - `walsh` komprimiert mit `compressMat02`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	70,18	-6,3
3	0,01	89,27	60,94
4	0,02	96,48	87,11
5	0,04	98,91	96,00
6	0,09	99,63	98,80
7	0,3	99,91	99,65
8	1,3	99,97	99,91
9	7,3	99,99	99,97
10	32	99,99	99,99

Im Unterschied zu Test 2 ergeben sich die besseren Kompressionsraten in Test 3 durch das Finden von Skalarmultproduktionen.

Test 4 - `matEntryEqRow` komprimiert mit `compressMat0`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	59,65	-43,75
3	0,02	78,11	20,31
4	0,04	88,58	58,20
5	0,06	94,16	78,61
6	0,25	97,05	89,18
7	1,3	98,52	94,56
8	9,5	99,26	97,27
9	73	99,63	98,63

Hier steigt die Laufzeit, was darauf zurückzuführen ist, dass zum einen der Speicherplatzbedarf erhöht ist, da man hier nicht so viele Gleichheiten findet, zum anderen gibt es mehr Vergleiche beim Finden von gleichen Nichtterminalen, da nicht so viele gelöscht werden aufgrund der Ungleichheiten.

Test 5 - `matEntryEqRow` komprimiert mit `compressMat0123`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	57,89	-50
3	0,01	81,11	31,25
4	0,04	92,64	73,05
5	0,1	97,36	90,33
6	0,4	99,11	96,73
7	3	99,71	98,94
8	21	99,91	99,67
9	165	99,99	99,90

Dadurch, dass man nun auch nach Additions- und Skalarmultproduktionen sucht, steigt die Laufzeit enorm. Die erhöhte Kompressionsrate kommt dadurch zustande, dass man mit Hilfe von konstanten MTDDs viele Additionsproduktionen finden bzw. erzeugen kann.

Test 6 - `matLinear` komprimiert mit `compressMat03`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	40,35	-112,5
3	0,05	74,25	6,3
4	0,24	90,92	64,45
5	4,6	96,62	87,60
6	133	98,88	95,90

Man würde hier eine bessere Laufzeit erzielen, wenn nicht nach gleichen Nichtterminalen gesucht werden würde (die Matrix besitzt nämlich gar keine).

Test 7 - `matTest1` komprimiert mit `compressMat0`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	43,86	-100
3	0,01	56,65	-57,81
4	0,04	67,77	-17,97
5	0,2	72,53	-0,68
6	2,3	74,22	5,47
7	38	74,76	7,46

Im Vergleich zu den anderen Tests sind die Kompressionsraten hier schlechter, was wohl einfach an einem Mangel an gleichen Nichtterminalen liegt.

Test 8 - `matTest1` komprimiert mit `compressMat01`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	43,86	-100
3	0,01	57,51	-54,69
4	0,08	67,98	-17,19
5	1,3	72,58	-0,49
6	75,2	74,23	5,52

Es gibt kaum eine Verbesserung zu Test 9, die Laufzeiten sind sogar viel schlechter.

Test 9 - `matTest1` komprimiert mit `compressMat0123`

Höhe	Zeit	c-rate-mtdd	c-rate-mat
2	0,01	43,86	-100
3	0,03	67,38	-18,75
4	0,09	85,59	47,27
5	1,62	94,88	81,25
6	77,5	98,31	93,80

Und auch hier sieht man wieder, dass das Ausnutzen von konstanten MTDDs zu Additionsproduktionsbildung zu einer sehr guten Kompression führt.

Kapitel 6

Fazit

Es liegt in der „Natur der Sache“, dass es bei Kompressionsverfahren immer um einen Trade-off zwischen Laufzeit und Kompressionsrate geht, was auch die im vorigen Kapitel präsentierten Tests gezeigt haben. Hinzu kommt noch, dass beide Parameter auch stark von dem Aufbau von Matrizen abhängen, also, ob und wie sich Strukturen in Matrizen wiederholen. Dabei kann man zum Schluss kommen, dass sich das Komprimieren nur lohnt, wenn möglichst viele Gleichheiten in den Matrizen zu finden sind, da dadurch natürlich die Laufzeit für das Finden von Additionsproduktionen (beide „Arten“) und Skalarmultproduktionen deutlich gesenkt wird.

Ob die gewählte Datenstruktur für das Darstellen von $\mathrm{MTDD}_{(+,\cdot)}$ in Haskell „gut“ ist, könnte man wohl nur durch einen Vergleich mit anderen Datenstrukturen sicher beantworten. Die Frage stellt sich deshalb, da der Speicherverbrauch doch enorm für große Matrizen ist (weshalb ja der Verbrauch eingeschränkt werden musste). Ein Grund für den hohen Speicherverbrauch wird sicherlich an der nicht vorhandenen *lazyiness* liegen, dass heißt, dass die Maps von Produktionen immer komplett ausgewertet vorliegen und somit im RAM verweilen, auch wenn die gesamten Maps vielleicht nicht gebraucht werden.

Es wäre deshalb vielleicht besser, wenn man statt Maps einfach Listen nimmt, obwohl die Zugriffzeiten auf die Elemente der Listen natürlich steigen würden.

Die Kompressionsalgorithmen bieten darüberhinaus eine Möglichkeit, Parallelisierungen einzuführen, in dem parallel nach möglichen Additions- und Skalarmultproduktionen gesucht wird.

Inwieweit mit dem in dieser Arbeit vorgestellten Verfahren sich Strings komprimieren lassen, ist unbeantwortet geblieben. Man kann sich aber leicht vorstellen, dass sich nur Strings gut komprimieren lassen, die man auch in die Struktur einer Matrix „zwängen“ kann und es zusätzlich viele gleiche Teilstrings gibt.

Anhang A

Matrixoperationen auf MTDD$_{(+,\cdot)}$s

Es werden hier nun die Matrixoperationen auf MTDD$_{(+,\cdot)}$s grob untersucht (es sind an sich Ideen wie die Lösung aussehen könnte), die in [RW13] für MTDD$_+$s definiert waren. Dabei wird auch nur grob auf die in [RW13] gezeigten Algorithmen eingegangen.

Es soll an sich nur gezeigt werden, dass bei Matrixoperationen auf MTDD$_{(+,\cdot)}$s im Vergleich zu MTDD$_+$s keine Abstriche bezüglich der Laufzeit der Algorithmen gemacht werden müssen.

A.1 Skalarmultiplikation

Sei M ein MTDD$_{(+,\cdot)}$ der Höhe h mit $M = (N, P, S)$ und $n \in \mathbb{Z}$. Betrachte $n \cdot val(S)$:

Mit der Einführung der Produktion $A \to c \cdot B$ ändert sich am Algorithmus aus [RW13, Abschnitt 3.2.1] nichts, denn auch hier müssen nur die Terminalproduktionen dahingehend geändert werden, dass bei Terminalproduktionen $A' \to a$ nur das Terminal zu $n \cdot a$ geändert werden muss.

A.2 Transposition

Wie in [RW13, Abschnitt 3.2.2] zu sehen ist, betrachtet man bei der Transposition auf MTDD$_+$ nur DownStep-Produktionen, die man mit der Einführung von Skalarmultproduktionen unverändert lässt. Deshalb bleibt der Algorithmus auf MTDD$_{(+,\cdot)}$s gleich.

A.3 Summe aller Matrixelemente

Da beim Berechnen der Summe aller Matrixelemente ein *+-Circuit* (ein MTDD$_+$, wo keine DownStep-Produktionen enthalten sind und somit keine Matrix, sondern ein Skalar dadurch repräsentiert wird, (siehe [LS14]) erzeugt und ausgewertet wird, bietet sich hier an, den +-Circuit um die Skalarmultproduktion zu erweitern, also einen $(+, \cdot) - Circuit$ zu kreieren.

Es ist lediglich anzumerken, dass es beim Auswerten des $(+, \cdot)$-Circuit's zur Einsetzung von B in einer Skalarmultproduktion $A \to c \cdot B$ kommen kann und es dabei drei Fälle gibt:

Fall 1: B erzeugt eine DownStep-Produktion $\begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$. Dann wird aus $c \cdot B$: $c \cdot B_1 + c \cdot B_2 + c \cdot B_3 + c \cdot B_4$

Fall 2: B erzeugt eine Additionsproduktion $B_1 + B_2$. Dann wird aus $c \cdot B$: $c \cdot B_1 + c \cdot B_2$

Fall 3: B erzeugt eine Skalarmultproduktion $c_1 \cdot B_1$. Dann wird aus $c \cdot B$: $c_1' \cdot B_1$,

mit $c_1^{'} = c \cdot c_1$

An der Laufzeit des Algorithmus ändert sich nichts.

A.4 Spur einer Matrix

Auch im Algorithmus aus [RW13, Abschnitt 3.2.4] wird an sich nur ein $+$-Circuit erzeugt und ausgewertet. Mit dem Erzeugen eines $(+, \cdot)$-Circuit's wie in A.3 ändert sich auch hier an der Laufzeit nichts.

A.5 Matrixelement berechnen

Es wird auch im Algorithmus aus [RW13, Abschnitt 3.2.5] an sich nur ein $+$-Circuit erzeugt und ausgewertet. Mit dem Erzeugen eines $(+, \cdot)$-Circuit's wie in A.3 ändert sich auch hier an der Laufzeit nichts.

A.6 Matrixmultiplikation

Soll $A \cdot B$, mit A, B sind Nichtterminale, berechnet werden, so wird gemäß [RW13, Abschnitt 3.2.6] eine neue Matrix (A, B) erzeugt, wobei $val((A, B)) = val(A) \cdot val(B)$ gilt.

Es muss lediglich ein weiterer Fall, nämlich $A \to c \cdot A^{'}$ (analog $B \to d \cdot B^{'}$) betrachtet werden:

Aus $(A, B) = A \cdot B$ wird dann $(A, B) = c \cdot A^{'} \cdot B$, was zur Produktion $(A, B) \to c \cdot (A^{'}, B)$ führt. Dies ist korrekt, da $val((A, B)) = val(c \cdot A^{'}) \cdot val(B) = c \cdot val(A^{'}) \cdot val(B) = c \cdot val((A^{'}, B))$.

Die Laufzeit bleibt gleich.

A.7 Gleichheitstest

Ob $A = B$ gilt, mit A, B sind Nichtterminale, wird in [RW13, Abschnitt 3.2.7] dadurch beantwortet, dass ein lineares Gleichungssystem aus $A - B = 0$ erzeugt wird, indem man A und B solange auflöst, bis nur noch Nichtterminale der Höhe 0 übrig sind. Die Terminale werden nicht „angerührt“.

In diesem Algorithmus werden Summen wie $A_1 + A_1$ zu $2 \cdot A_1$ zusammengefasst – die „Standardisierung“ (siehe Schritt 1 in [RW13, Abschnitt 3.2.7]) – weshalb das Einführen einer Skalarmultproduktion nur einen weiteren Fall bei der „Standardisierung"darstellt und sich deshalb an der Laufzeit nichts ändert.

Literaturverzeichnis

[BBC+] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J.M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van Der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Seite 57, 1994.

[cab] `https://www.haskell.org/cabal/`, abgerufen am 10. April 2016.

[EM03] Anand Ekambaram and Eurípides Montagne. *Computer and Information Sciences - ISCIS 2003: 18th International Symposium, Antalya, Turkey, November 3-5, 2003. Proceedings*, chapter An Alternative Compressed Storage Format for Sparse Matrices, pages 196–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[ghc] `https://www.haskell.org/ghc/download_ghc_7_10_3`, abgerufen am 10. April 2016.

[gr] `https://people.richland.edu/james/lecture/m116/sequences/geometric.html`, abgerufen am 10. April 2016.

[LS14] M. Lohrey and M. Schmidt-Schauß. *Processing Succinct Matrices and Vectors*. *CoRR*, abs/1402.3452, 2014. `http://dblp.uni-trier.de/rec/bib/journals/corr/LohreyS14`, abgerufen am 10. April 2016.

[map] `https://hackage.haskell.org/package/containers-0.5.7.1/docs/Data-Map-Lazy.html`, abgerufen am 10. April 2016.

[Mar10] S. Marlow. *Haskell 2010 Language Report*, 2010. `https://www.haskell.org/definition/haskell2010.pdf`, abgerufen am 10. April 2016.

[Mar15] S. Marlow. *Fighting spam with Haskell*, 2015. `https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/`, abgerufen am 10. April 2016.

[mat] `https://hackage.haskell.org/package/matrix-0.3.4.4/docs/Data-Matrix.html`, abgerufen am 10. April 2016.

[par] `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/runtime-control.html`, abgerufen am 10. April 2016.

[pyt] `https://docs.python.org/3.5/howto/functional.html`, abgerufen am 10. April 2016.

[RW13] D. Reichelt and R. Willhauk. *Implementierung von Algorithmen für Grammatik-komprimierte Matrizen in der funktionalen Programmiersprache Haskell*, 2013.

[Sch15] G. Schnitger. *Skript zu Vorlesung „Theoretische Informatik 2“*, 2015. http://www.thi.informatik.uni-frankfurt.de/lehre/gl2/sose15/gl2_sose15_skript.pdf, abgerufen am 10. April 2016.

[SS15] M. Schmidt-Schauß and D. Sabel. *Einführung in die Funktionale Programmierung*, 2015. http://www.ki.informatik.uni-frankfurt.de/lehre/WS2015/EFP/skript/skript-23-Nov-15.pdf, abgerufen am 10. April 2016.

[Sti16] J. Stix. *Lineare Algebra*, 2016. https://www.uni-frankfurt.de/58527609/Stix-LineareAlgebra-Skript.pdf, abgerufen am 10. April 2016.

[wik] https://de.wikipedia.org/wiki/Invertieren_%28Bildbearbeitung%29, abgerufen am 10. April 2016.

[YMSS] S.N. Yanushkevich, D.M. Miller, V.P. Shmerko, and R.S. Stankovic. *Decision Diagram Techniques for Micro- and Nanoelectronic Design Handbook*. Seite 907, 2005. https://books.google.de/books?id=QjLZVzULaiYC, abgerufen am 10. April 2016.