Implementierung verschiedener Varianten des "Iterative Repeat Replacement"-Algorithmus zur Berechnung von kleinen Straight-Line-Programs zur Wort- und Textkompression

Nicolas Torchalla, 3854839

11. Juni 2012

Erklärung gemäß Bachelor-Ordnung Informatik 2007 §24 Abs. 11

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

Frankfurt am Main, 11. Juni 2012

Nicolas Torchalla

Inhaltsverzeichnis

1 Einführung

Das Smallest-Grammar-Problem sucht die kleinste kontextfreie Grammatik, die eindeutig einen bestimmten String erzeugt. Der Iterativ-Repeat-Replacment-Algorithmus findet zwar nicht die kleinste, aber eine kleine Grammatik und ist somit eine Zwischenlösung auf dem Weg zum Smallest-Grammar-Problem. Diese Zwischenlösung liefert für einige praktische Anwendungen zureichend gute Ergebnisse, während Sie für andere weniger praktikabel ist.

Ein Anwendungsfeld des Smallest-Grammar-Problems ist die Bestimmung der Kolmogorow-Komplexität [Wikc]. Dieses Maß für die Strukturiertheit eines Strings wird durch die Länge des kürzesten Programms, das dieses Zeichenkette erzeugen kann, bestimmt. Eine kleine, aber nicht kleinste Grammatik, kann nur eine Obergrenze für die Kolmogorow-Komplexität bestimmen, sie aber nicht berechnen.

Man kann ein kleinstes Programm, das eine Zeichenkette erzeugt, auch als beste Kompression dieser Zeichenkette verstehen. Hier ist die Zwischenlösungen, mittels des Iterativ-Repeat-Replacement-Algorithmus, sinnvoller. Es wird zwar keine maximale Kompressionsrate erreicht, aber die Berechenbarkeit in kubischer Zeit kann den Algorithmus für praktische Anwendungen trotzdem interessant machen.

Ein weiteres Anwendungsgebiet ist das finden von Mustern in einem Text. Dieses ist z.B. wichtig für die Arbeit mit DNA-Sequenzen. Das finden von Mustern, in einem mit dem IRR-Algorithmus komprimierten String, geht sehr viel schneller, da Wiederholungen von gleichen Sequenzen nicht mehrfach durchlaufen werden müssen. Leider stellt der Iterativ-Repeat-Replacment-Algorithmus auch hier nur eine Teillösung dar, da in der Gen-Forschung häufig keine exakten Übereinstimmungen gesucht werden. Algorithmen die teilweise Übereinstimmungen finden, verwenden allerdings das Auffinden von exakten Übereinstimmungen als Grundlage.

2 Haskell

Für die Implementierung des IRR-Algorithmus habe ich die Programmiersprache Haskell verwendet. Ich möchte hier eine Kurzeinführung geben.

Für die Arbeit mit Haskell empfehle ich WinGHCI, den Glasgow Haskell Compiler Interactive mit einer GUI für Windows und Notepad++ als Editor.

Haskell unterscheidet sich stark von imperativen Programmiersprachen. Ein wichtiger Unterschied ist, das bei imperativen Programmiersprachen der Speicher mittels Befehlen verändert wird. Bei funktionalen Programmiersprachen wie Haskell, werden Ausdrücke ausgewertet. Es existieren keine Variablen im Speicher, die geändert werden können. Somit ist das Ergebnis des Ausdruckes jederzeit unabhängig von Speicherzuständen oder dem Computer auf dem Haskell ausgeführt wird.
Dies ermöglicht z.B. eine automatische Parallelisierbarkeit.

2.1 Funktionen

Haskell besteht aus Funktionen, die sich gegenseitig oder sogar selbst aufrufen können.
Das zum Einstig in eine Programmiersprache übliche Hallo-Welt-Programm sieht in Haskell so aus:

```
1 f = "Hallo Welt"
```

Der Name der Funktion ist *f*. Wenn die Funktion f aufgerufen wird, gibt sie den String "Hallo Welt" aus.

Eine derartige Funktion ist noch langweilig. Deshalb schreiben wir jetzt eine Funktion die einen Parameter verwendet:

```
1 f a = a
```

Die neue Funktion *f* erwartet einen Wert. Diesen Wert gibt sie dann unverändert aus. Selbstverständlich kann der Wert vor der Ausgabe auch manipuliert werden.
Ein Beispiel:

```
1 f1 a = a^2
2 f2 a = a*a
3 f3 a b = a+b
```

Die Funktionen *f1* und *f2* berechnen beide das Quadrat einer beliebigen eingegeben Zahl. Die Funktion *f3* berechnet das Produkt aus zwei übergebenen Zahlen a und b.

Der Aufmerksame Leser wird sicher bemerkt haben, das die Funktion *f* noch beliebige Werte erwartet hat, während die Funktionen *f1*, *f2* und *f3* nur noch Zahlen akzeptieren.

Haskell berechnet automatisch ein Typsystem für seine Funktionen. Dieses versucht es so allgemein wie möglich zu halten. Durch das Anwenden der Multiplikation auf den Parameter a, haben wir uns festgelegt das a eine Zahl sein muss. Außerdem wissen wir schon, dass das Ergebnis einer Multiplikation immer eine Zahl ist, also auch das Ergebnis der Funktion eine Zahl sein muss.

Der Typ dieser Funktion wird in Haskel auf diese Art dargestellt:

```
1 f1 :: Int -> Int
2 f1 :: (Num a) => a -> a
```

Die erste Zeile stellt eine Vereinfachung, des eigentlich berechneten Typs dar. Wir geben eine Integer ein und bekommen eine Integer heraus. Da wir aber jede Art von Zahl eingeben können und nicht auf Integer beschränkt

sind, ist der von Haskell berechnete Typ in Zeile zwei. Wir legen uns fest, dass a ein numerischer Wert sein muss. Ansonsten ist alles erlaubt.

Der Typ von *f3* sieht so aus:

```
1 (Num a) => a -> a -> a
```

Das Zeichen nach dem letzten (nicht in Klammern eingefassten) Pfeil steht immer für den Typ des Ergebnisses. Die Rest beschreibt den Typ der Eingabe.

Wir sehen das der Typ der ersten Eingabe mit dem Typ der zweiten Eingabe und dem Ergebnis übereinstimmen muss. Außerdem muss der Typ wieder ein numerischer sein.

Allgemein haben Funktionen in Haskell folgenden Typ und Aufbau:

```
1 NAME t1 -> t2 -> t3 ->.....-> tn = x
2 NAME p1 p2 ... pn = Funktionsdefinition
```

Der Ergebnistyp x lässt sich dabei meistens aus der Funktionsdefinition genauer berechnen. Der Name kann frei gewählt werden und dient dazu die Funktion später aufzurufen. Die Parameter p1 bis pn sind die Werte, die der Funktion übergeben werden. Diese stehen dann unter ihren Parameternamen in der Funktionsdefinition zur Verfügung. t1 bis tn sind die Typen der Parameter p1 bis pn. Die Funktionsdefinition ist eine Anweisung, wie die Parameter verarbeitet werden sollen, um den gewünschten Rückgabewert zu erreichen. Dabei können unter anderem mathematische Operationen, boolesche Operationen sowie Operationen auf Listen und anderen Datenstrukturen verwendet werden und andere Funktionen aufgerufen werden.

Es kann nur ein Wert zurückgegeben werden. Sollte es notwendig oder praktikabel sein, das die selbe Funktion mehrere Werte berechnet, müssen diese in einem gemeinsamen Datentyp verpackt werden. Zu diesem Zweck können Tupel, Listen oder eigene Datenstrukturen verwendet werden.

2.2 Rekursion

Eine Rekursion ist ein einfaches Mittel, die gleiche Funktion immer wieder auszuführen. Hierbei ruft sich die Funktion, in ihrem Rumpf, selbst wieder auf. Die aus imperativen Sprachen bekannten Schleifen stehen uns in Haskell, innerhalb von Funktionen, nicht zur Verfügung.

Gründe die gleiche Funktion wiederholt ausführen wären z.B. das sukzessive Verbessern einer Lösung, bei einer Funktion, die eine Lösung immer weiter annähert (Newtonverfahren zur Bestimmung von Nullstellen), die Anwendung einer Funktion auf sämtliche Elemente einer Liste (oder komplexeren Datenstruktur) oder das zurückführen einer Lösung auf ein kleineres Teilproblem.

Ein Beispiel für eine Rekursion ist diese Berechnung der Fakultät (Der Typ muss nicht mit angegeben werden, sondern kann von Haskell auch automatisch berechnet werden.):

```
1 fak :: (Num t) => t -> t
2 fak 2 = 2
3 fak a = a * fak (a-1)
```

Zuerst einmal lernen wir hier ein neues Konstrukt kennen. Die Funktion *fak* ist zwei mal definiert.

- Die Definition in Zeile 2 wird verwendet, wenn wir die Funktion *fak* mit dem Parameter 2 aufrufen.
- Die Definition in Zeile 3 wird für alle anderen Werte, die der Parameter annehmen kann, verwendet.

Diese Unterscheidung nennt sich Pattern-Matching und wird uns noch häufiger begegnen.

Zeile 2 ist unsere sogenannte Abbruchbedingung für die Rekursion. Da die Funktion *fak* sich in Zeile 3 immer wieder selbst aufruft, würde unser Programm nie terminieren. Bei Rekursionen muss immer darauf geachtet werden sinnvolle Abbruchbedingungen zu definieren.

Das Grundprinzip einer Rekursion ist, das Problem in Teilprobleme zu zerlegen. Ein Teilproblem sofort zu lösen und die Funktion für die das nächsten Teilproblem erneut zu referenzieren.

Unsere Teilproblem, für die Fakultät einer Zahl a, ist die Fakultät der Zahl $a-1$. Diese müssen wir schließlich nur noch mit a multiplizieren, um die Fakultät von a zu erhalten.
Die Fakultät von $a-1$ errechnen wir wiederum aus der Fakultät von $a-2$ usw. Die Fakultät von zwei haben wir fertig definiert.

In Zeile 3 sehen wir, wie die Funktion *fak* sich selbst für das nächst kleinere Teilproblem aufruft und das Ergebnis mit dem Faktor, den sie übergeben bekommen hat, multipliziert.

2.3 Listen

Wichtig um mit Haskell zu arbeiten ist auch das Verständnis von Listen. Eine Liste in Haskell wird folgendermaßen Dargestellt (hier ein Beispiel der Repräsentation einer Liste mit Zahlen (Int)):

```
1 [1,2,3,4,5,6,7,8,9,10,11]
```

Bei der Typdefinition werden um den Typ der Listenelemente noch eckige Klammern gesetzt:

```
1 f :: [Char]
2 f = ['d','c','b','a']
```

Der Typ Char enthält einen einzelnen Buchstaben.

Auch mit Listen kann Pattern-Matching verwendet werden:

```
1 f:: [a] -> String
2 f []     = "Leere Liste"
3 f [x]    = "Liste mit exakt einem Element"
4 f (x:xs) = "Liste mit einem ersten Element x und einer Restliste xs"
```

Auf diese Art kann, für verschiedene Listenzustände, eine unterschiedliche Funktionsdefinition angegeben werden. Dies ist vor allem sehr praktisch beim rekursiven Durchlaufen von Listen.

Eine Liste ist in Haskell als ein Datum plus eine Restliste definiert. Die Restliste kann nur Elemente vom gleichen Typ wie das vorangestellte Datum enthalten. Das Datum ist mit der Restliste durch den :-Operator verbunden, den wir auch im Pattern in Zeile 4 sehen.

Die genauere Darstellung von Listen sieht so aus:

```
1 'a':[]
```

In diesem Fall ist unser Datum wieder vom Typ Char (also ein einzelner Buchstabe). Unsere Restliste ist leer. Dies Repräsentieren wir durch eckige Klammern ohne einen Wert dazwischen.
Wir können unsere Liste nun sukzessive erweitern indem wir weitere Elemente voranstellen:

```
1 'b':('a':[])
2 'c':('b':('a':[]))
3 'd':('c':('b':('a':[])))
```

Haskell kann diese Liste auch noch interpretieren wenn wir die Klammern weglassen:

```
'd':'c':'b':'a':[]
```

Mit dem nun erworben Wissen können wir die beiden, Haskell schon bekannten, Funktionen *head* und *tail* definieren.

```
head :: [a] -> a
head (x:xs) = x
head []     = error "leere Liste"

tail :: [a] -> [a]
tail (x:xs) = xs
tail []     = []
```

Die Funktion *head* gibt das erste Element einer Liste zurück. Die Funktion *tail*, die gesamte Liste ohne das erste Element.

2.4 Arbeit mit Listen

Mit Pattern kann man auch einfach rekursiv über Listen laufen. Wir stellen uns eine Liste von Zahlen vor, die wir verdoppeln wollen:

```
1 f :: (Num a) => [a] -> [a]
2 f (x:xs) = x*2:f xs
3 f []     = []
```

In Zeile zwei verdoppeln wir das vorderste Element und führen danach die Funktion rekursiv auf die Restliste aus. Sobald unsere Restliste leer ist, hängen wir, durch die Abbruchbedingung in Zeile 3, eine leere Liste an und unsere Funktion ist fertig.

Um mehrere Listen zu einer zusammenzufügen, steht der ++-Operator zur Verfügung.

Außerdem wichtig für die Arbeit mit Listen, sind die Funktionen *map*, *filter* und *foldl* bzw. *foldr*.

2.4.1 map

Die Funktion *map* wendet eine Funktion auf eine komplette Liste an.
Um eine Liste von zahlen zu verdoppeln können wir also auch schreiben:

```
1 f :: (Num a) => [a] -> [a]
2 f xs = map verdoppeln xs
3
4 verdoppeln :: (Num a) => a -> a
5 verdoppeln a = a*2
```

Für die Eingabe f [1,2,3,4,5,6,7,8,9] erhalten wir [2,4,6,8,10,12,14,16,18].

Die Funktion *verdoppeln*, verdoppelt eine einzelne Zahl. Mit map wenden wir sie auf eine komplette Liste an. In diesem Beispiel können wir auch sehen, wie Funktionen sich gegenseitig referenzieren.

2.4.2 filter

Mit filter können wir einige Elemente einer Liste aussortieren. Wir definieren eine Funktion, die zu einem booleschen Wert ausgewertet werden kann und die Elemente unserer Liste als Eingabe akzeptiert. Alle Elemente einer Liste, für die diese Funktion True ergibt, werden behalten, der Rest wird verworfen.

```
1 f :: (Num a, Ord a) => [a] -> [a]
2 f xs = filter kleinerals5 xs
3
4 kleinerals5 :: (Num a, Ord a) => a -> Bool
5 kleinerals5 a = a<5
```

Die Funktion *kleinerals5* ergibt für die Eingaben 1 bis 4 True.
Für die Eingabe f [1,2,3,4,5,6,7,8,9] erhalten wir [1,2,3,4].

2.4.3 foldl und foldr

Die Funktionen *flodl* und *foldr* verrechnen alle Elemente einer Liste zu einem Ergebnis. Dazu verwenden sie eine Funktion, die ein Element in das Gesamtergebnis einrechnen kann.

```
1 f :: (Num a) => [a] -> a
2 f  xs = foldr (+) 0 xs
3
4 g :: (Num b, Ord b) => [b] -> Bool
5 g xs = foldl (alleskleiner5) True xs
6
7 alleskleiner5 :: (Num a, Ord a) => Bool -> a -> Bool
8 alleskleiner5 _ False = False
9 alleskleiner5 z True  = z<5
```

f addiert alle Zahlen einer Liste.
Für die Eingabe f [1,2,3,4] gibt Haskell 10 aus.

Die Funktion alleskeiner5 gibt für Zahlen kleiner fünf True aus. Sobald eine Zahl aber größer fünf war bleibt das Ergebnis für immer False.
Für die Eingabe g [1,2,3,4] bekommen wir True, für die Eingabe g [1,2,3,4,5] bekommen wir False.

Der Unterschied zwischen *foldl* und *foldr* ist die Richtung in der die Funktionen die Liste durchlaufen. Der zweite Parameter der Funktion ist der Startwert, mit dem die übergebene Funktion anfängt zu rechnen. Würde ich bei Funktion *f* statt 0 eine 4 eingeben, würde sich mein Ergebnis um 4 erhöhen.
Fange ich bei Funktion *g* mit einem False an, wird meine Funktion davon ausgehen, dass schon ein Element größer 5 gefunden wurde.

3 Kontextfreie Grammatiken

Um unsere IRRA implementieren zu können, müssen wir noch klären was Kontextfreie Grammatiken sind.

Kontextfreie Grammatiken können formal durch den Tupel (V, Σ, R, S) dargestellt werden.

- V ist die Menge der Nichtterminale
- Σ ist die Menge der Terminale (muss disjunkt von V sein)
- R ist die Liste unsere Produktionen
- S ist das Startsymbol

Ein Nichtterminal ist ein Symbol, das wir uns zur Hilfe nehmen, um Stellen zu markieren, an der wir eine Produktionsregel anwenden können. Eine Produktionsregel einer kontextfreien Grammatik ersetzt immer ein Nichtterminal durch eine Sequenz von Terminalen und Nichtterminalen.

Die Produktionsregeln eine Kontextfreien Grammatik haben die Form:

$$\text{Nichterminal} \rightarrow \text{Liste von Terminalen und Nichtterminalen}$$

Wichtig ist, das bei kontextfreien Grammatiken, anders als bei Kontextsensitiven Grammatiken, auf der linken Seite immer genau ein Nichtterminal steht. Das heißt unabhängig von den Nachbarn eines Nichtterminals, kann dieses immer durch eine bestimmte Sequenz ersetzt werden. Die linke Seite der Produktionsregel werde ich auch als Rumpf bezeichnen.

Zum besseren Verständnis von Kontextfreien Grammatiken gebe ich als Beispiel die Grammatik der Sprache

$$\text{S} = \{ \text{ Wörter mit gleich vielen a's und b's} \}$$

über dem Alphabet $\{a, b\}$ an:

- Die Menge der Terminale ist somit leicht zu finden: $\Sigma = \{a, b\}$
- Als Startsymbol legen wir fest: $S = \{N\}$
- Jetzt brauchen wir noch ein Nichtterminal das b's und eins das a's erzeugt.
  Ich nenne die beiden: $V = \{A, B\}$
- Zuletzt unsere Produktionsregeln: $R = \{N \rightarrow NN|AB|BA, A \rightarrow a, B \rightarrow b\}$

Die Pipe's in der ersten Produktionsregel sind ein Oder. Wir könnten aus der einen Produktionsregel auch drei machen.

Es ist zu sehen das wir für jedes A das wir hinzufügen auch ein B hinzufügen müssen und umgekehrt. Durch die Möglichkeit der Verdopplung von N kann die Grammatik beliebig lange Wörter entwerfen. Die Produktionsregeln von A und B sind überflüssig. Sie können ersetzt werden wenn N direkt die Terminale a und b erzeugt.

Der Iterativ-Repeat-Replacement-Algorithmus hat allerdings noch eine speziellere Anforderung an die Grammatik, die er erzeugt. Diese muss zu genau einem String herleitbar sein, nämlich dem String aus dem sie erzeugt worden ist. Logischerweise taugt ein Kompressionsalgorithmus wenig, wenn wir die komprimierten Daten nicht eindeutig wieder herstellen können.

Um dieser Anforderung gerecht zu werden, müssen wir noch zwei zusätzliche Regeln einführen:

1. Jedes Nichtterminal muss in genau einer Produktionsregel, auf der rechten Seite, vorkommen.
2. Es sind keine Schleifen erlaubt. Das heißt wenn ein Nichtterminal N1, im Rumpf der Produktionsregel von N2 vorkommt, so darf N2 nicht im Rumpf der Produktionsregel von N1 vorkommen.

3.1 Repräsentation in Haskell

Um mit kontextfreien Grammatiken in Haskell zu arbeiten werde ich das Modul SCFG [SS] von Herrn Prof. Dr. Schmidt Schauß verwenden. Diese speichert Grammatiken als eine Liste von Produktionen ab. Um das Programm zu starten werde ich eine einzelne Produktion hinzufügen, in der mein Startsymbol auf den gesamten, zu verarbeitenden, String zeigt.

4 Suffixbaum

Ein Suffixbaum enthält alle Suffixe einer dazugehörigen Zeichenkette.
An jeder Kante eines Suffixbaumes stehen ein oder mehrere Zeichen. Jeder innere Knoten hat mindestens zwei Kinder.

Die Suffixe erhält man, wenn man von der Wurzel bis zum Blatt alle Zeichen der Kanten, an denen man vorbeikommt, aneinander reiht. Jedes Blatt stellt dabei einen anderen Suffix dar. Keiner der Suffixe kommt zwei mal vor. Für eine Zeichenkette der Länge n hat ein Suffixbaum also genau n Blätter.

Da jeder innere Knoten mindestens zwei Kinder hat, kann es nur genau so viele Knoten wie Blätter geben.

Eine weitere interessante Eigenschaft des Suffixbaumes ist, dass die Wege zu Knoten Wörter ergeben, die sich in dem erzeugenden String wiederholen. Dies kann man sich leicht veranschaulichen. Da jeder Knoten mindestens zwei Kinder haben muss, folgen nach dem Knoten noch mindestens zwei Blätter. Das bedeutet zwei verschiedene Suffixe die gleich beginnen.

Nehmen wir diese beiden Informationen zusammen haben wir n verschiedene Wörter die sich im String wiederholen.
Auf diese Art möchte ich die Anzahl an Wörtern, deren Vorkommen ich in der Zeichenkette überprüfen muss, beschränken.

Ich verwende einen fertig implementierten Suffixbaum [St]. Dieser lässt sich mit der Funktion *construct* aus einer Liste erzeugen. Er benötigt am Ende der eingegebenen Liste ein einzigartiges Zeichen.

5 Der Boyer-Moore-Algorithmus

Der Boyer-Moore-Algorithmus kann in linearer Zeit sämtliche Postionen, an denen ein Wort in einem String vorkommt, bestimmen.

Dies erreicht er dadurch, dass er nicht an jeder Position das komplette Wort mit dem Teil-String vergleichen muss. Stattdessen werden Wort und String solange verglichen bis ein erster Unterschied festgestellt wird. Danach springt der Algorithmus einige Zeichen weit nach vorne. Wie weit er springen darf hängt von dem Zeichen ab, mit dem der Missmatch auftritt und wird in einem Vorverarbeitungsschritt berechnet.
Zum Beispiel kann der Algorithmus, für Zeichen die nicht im Wort enthalten sind, immer um die volle Länge des Wortes springen.

Ich werden den Boyer-Moore-Algorithmus verwenden um herauszufinden wie oft ein bestimmtes Wort sich in einem String wiederholt. Dafür verwendet ich die Implementierung aus dem Buch "Pearls of Functional Algorithm Design" [Bir].

6 Iterativ-Repeat-Replacement-Algorithmus

Der Iterativ-Repeat-Replacement-Algorithmus, im nachfolgenden IRRA genannt, führt die Kompression in zwei Schritten durch. Erst findet er das nächste, zu ersetzende, Wort, danach fügt er es hinzu.

6.1 Einige Hilfsfunktionen

Zuerst definiere ich einige Hilfsfunktionen die ich später brauchen werde.

Create Long String

Die Funktion *cls* erzeugt aus einer Grammatik einen langen String, indem sie den Rumpf aller Produktionen aneinanderreiht. Die einzelnen Produktionen werden durch einzigartige Nichtterminale getrennt. Dadurch ist sichergestellt, das der String später wieder gespalten werden kann.
Außerdem ist damit garantiert, das alle gefundenen Repeats in einem Rumpf liegen und nicht von einem in den anderen Rumpf überlappen. Würde das doch der Fall sein, müsste der Repeat das Trennzeichen zwischen den beiden Rümpfen enthalten. Die Trennzeichen sind aber alle einmalig, wiederholen sich also nicht.

```
1 cls (g:gs) = hlp list 1 where
2                     list = map (\(Production _ x)->x) (g:gs)
3                     hlp [] c          = []
4                     hlp (x:xs) c      = hlp1 x xs c
5                     hlp1 [] r c       = (N 1 ('$':show c)):hlp r (c+1)
6                     hlp1 (x:xs) r c   = x:hlp1 xs r c
```

Die Funktion *list* macht aus einer Grammatik eine Liste von Rümpfen. Diese werden dann von der Funktion *hlp* zu einem langen String verbunden. Nach dem einfügen jedes Rumpfes wird die Funktion *hlp1* gerufen die ein einzigartiges Trennzeichen erzeugt und einfügt, sodass die Grammatikstruktur leicht wieder hergestellt werden kann. Das Trennzeichen am Ende ist notwendig zum erstellen des Suffixbaumes.

Split

Die Funktion Split trennt einen langen String an Trennzeichen (einzigartigen Nichterminalen) auf und macht eine Liste von Strings darauß. Sie ist sozusagen die erste Stufe der Umkehrfunktion von *cls*.

```
1 split [] _ = []
2 split str c = fst a : split (tail (snd a)) (c+1) where a = break ((N 1 ('$':show
      c))==) str
```

In jedem Schritt wird die Restliste, mittels des Befehls *break*, in zwei Teillisten geteilt. Die erste Teilliste ist ein Rumpf einer Produktion. Die zweite Teilliste enthält alle verbleibenden Rümpfe und wird so lange weiter geteilt bis keine Trennzeichen mehr übrig sind.

Generate All Sub-Strings

Um alle Wörter zu finden, die als Repeat in Frage kommen, verwende ich die Funktion gass2. Diese verwendet einen Suffixbaum.

```
1 initgass2 ls = hlp (T.construct (ls ++ [T '#']))
2                           where hlp (T.Node x) = gass2 x []
3
4 gass2 []                  way=[]
5 gass2 ((w,T.Leaf):ts)     way=gass2 ts way
```

```
6 gass2 ((w,T.Node t):ts) way=if (length we)>1 then we:(gass2 t we)++gass2 ts way
7                               else (gass2 t we) ++ gass2 ts way
8                               where we = way ++ (T.prefix w)
```

Die Funktion initgass2 initialisiert unsere gass2-Funktion. Sie erwartet die Grammatik als einen einzelnen langen String, so wie er von der Funktion *cls* erzeugt wird.

Die Funktion gass2 (generate all sub-strings) erzeugt alle in Frage kommenden Teilworte. Sie unterscheidet zwischen 3 Fällen:

1. Untersuchte Teilliste leer (Zeile 4): Wir haben alle Kinder eines Knotens besucht. Das heißt es sind keine weiteren rekursiven Aufrufe nötig.
2. Blatt gefunden (Zeile 5): Wenn wir ein Blatt finden kommt dieser Teilstring nur einmal vor und wir brauchen den Substring nicht näher ansehen. Wir machen mit der Restliste weiter.
   Die anderen Knoten und Blätter in der Restliste haben den gleichen Vaterknoten. Die Variable way, die alle Präfixe bis zum Vaterknoten zusammenfasst, verändert sich bei einer Seitwärtsbewegung nicht.
3. Knoten gefunden (Zeile 6): Wenn wir einen Knoten finden, wissen wir, dass das Wort, das von der Wurzel aus hier endet, mehr als einmal im String vorkommt. Dieses Wort müssen wir, aus allen Präfixen an denen wir schon vorbeigekommen sind und dem Präfix das zu diesem Knoten geführt hat, zusammenfügen.
   Der Präfix bis zum Vaterknoten ist in way gespeichert. Der String vom Vaterknoten bis zum derzeitigen Knoten ist in w enthalten. Die Funktion *we* (in Zeile 8) fasst beide Variablen zusammen.

   Zuletzt muss noch überprüft werden ob unser, im Suffixbaum, zurückgelegter Weg mehr als ein Zeichen enthält. Wenn dies zutrifft wird er als Lösung akzeptiert.

   Von einem Knoten aus muss jetzt in zwei Richtungen weiter gesucht werden. Benachbarte Knoten auf gleicher Höhe und Kinderknoten. Die Rekursionen in die beiden Richtungen unterscheiden sich nur durch den bisher zurückgelegten Weg, der übergeben wird. Bei Nachbarknoten bleibt dieser identisch. Bei Kinderknoten muss er, um das Wegstück zum derzeitigen Knoten, erweitert werden, wie wir es bereits in *we* getan haben.

Wir haben also mit der Funktion *gass2* eine Möglichkeit alle sich wiederholenden Teilwörter zu finden und lazy zu erzeugen.
Das bedeutet, dass sobald ein Wort erzeugt wird, es sofort zur Weiterverarbeitung zur Verfügung steht. Auf diese Art muss die Liste mit allen Wörtern nicht abgespeichert werden.

Im nächsten Schritt wollen wir uns für das Wort, das wir ersetzen wollen, entscheiden.

6.2 Wörter ersetzen

Für das finden, des zu ersetzenden Wortes, habe ich alle drei Alternativen aus "The Smallest Grammar Problem as Constituents Choice and Minimal Grammar Parsing" [Car+10] implementiert.
Diese sind:

- Wort das am häufigsten vorkommt (mindestens einmal)
- Wort das am längsten ist und sich mindestens einmal wiederholt
- Wort das die stärkste Kompression ermöglicht

Wiederholungen dürfen dabei nicht überlappend sein.

Alle drei Methoden haben gemeinsam, das wir für eine Reihe von Worten testen müssen, wie oft es sich in unserem Text wiederholt. Mit dieser Information können wir dann eine Score für jedes Wort berechnen. Das Wort mit der besten Score wird ersetzt und wir fangen von vorne an.

6.3 Welche Ersetzung ist die nächste?

Für jede der Angesprochenen drei Methoden schreiben wir eine Funktion.

Alle drei Funktionen haben eine Initialisierungsfunktion, die eine Grammatik als Eingabe erwartet.

Die Hauptfunktionen fbm[nummer] (Find Best Match) haben folgende Parameter:

- cls - langer String, der den Rumpf aller Regeln enthält
- (x:xs) - Liste mit in Frage kommenden Repeats
- max - bisher bester Score
- erg - Positionen an denen der bisher beste Repeat gefunden werden kann
- rep - der bisher beste Repeat

Das Ergebnis der Funktionen ist jeweils ein Tupel dieser Form:

(mit der Funktion *cls* erzeugter String, (bestes Wort, Positionen an denen dieses Wort endet))

Compress

Der Compress-Algorithmus soll das Wort finden, das die höchste Kompressionsrate erreicht.

Um das Wort, das die stärkste Kompression ermöglicht, bestimmen zu können, gehen wir davon aus, das ein ersetztes Wort w uns $|w|-1$ Zeichen erspart. Das gesamte Wort wird durch ein Nichtterminal ersetzt. Außerdem muss eine zusätzliche Regel der Grammatik hinzugefügt werden. Diese braucht $|w|+1$ Zeichen.
Bezeichnen wir die Häufigkeit des Vorkommens w in unserem Gesamtstring s als op(w), dann können wir die Kompressionsrate, durch das ersetzen eines bestimmten Wortes, mit folgender Rechnung bestimmen:

$$(|w|-1)*(op(w)-1)-2$$

Zuerst betrachten wir die initialisierende Funktion:

```
1 initfbm1 gr = fbm1 ls (initgass2 ls) 0 [] [] where ls = cls gr
```

Sie verwendet die bereits angesprochenen Funktionen *initgass2* und *cls* um die zu testenden Wörter und den String, auf dem die Anzahl der Repeats berechnet werden, zu erzeugen.
Außerdem initialisiert sie die Ergebnisse.

Die eigentliche Funktion *fbm1* sieht so aus:

```
1 fbm1 cls      []     _   erg rep =(cls,(rep,erg))
2 fbm1 cls      (x:xs) max erg rep =if erg2 > max then fbm1 cls xs erg2 erg1 x
3                                  else fbm1 cls xs max erg rep where
4                                         erg1 = check (matches x cls) len
5                                         erg2 = (len -1)*((length erg1)-1)-2
6                                         len = length x
```

Mit jedem Rekursionsschritt errechnen wir die Score von einem Wort und merken und uns die bisher beste, für den nächsten Schritt.

Zuerst sehen wir uns die Hilfsfunktionen nach dem where-Statement an. In Zeile 6 wird die Länge des aktuell überprüften Wortes berechnet, da diese mehr als einmal gebraucht wird. In Zeile 4 berechnen wir eine

Liste, mit Positionen, an denen unser Wort, in unserem String, endet. Dafür verwenden ich den Booyer-Moore-Algorithmus. Die Funktion *check* entfernt überlappende Repeats. Die Länge dieser Liste ist die Anzahl an nicht überlappenden Repeats. In Zeile 5 berechnen wir die Score.

Die Funktion fbm1 unterscheidet zwischen 3 Fällen.

1. Das in diesem Schritt berechnete Ergebnis ist besser als das gemerkte: Wir merken uns das neue Wort, seinen Score und die Liste mit den Positionen an denen es vorkommt und starten den nächsten Rekursionsschritt mit diesen Werten.
2. Das berechnete Ergebnis ist schlechter als das gemerkte: Wir starten den nächsten Rekursionsschritt mit identischen Werten wie den derzeitigen.
3. Die Liste mit zu testenden Wörtern ist leer: Wir sind fertig und geben das beste Wort aus.

Repair

Der Repair-Alogrithmus findet den Repeat der sich am Häufigsten im String wiederholt.

```
1 initfbm2 gr = fbm2 ls (initgass2 ls) 1 [] [] where ls = cls gr
```

Einziger Unterschied der Initfunktion *initfbm2* zu der Initfunktion aus dem Compress-Algorithmus ist das die Score mit 1 initialisiert wird. Dies ist wichtig, da sonst die Score nicht leer bleibt, falls es keine Wörter gibt die sich mehr als einmal wiederholen.

Die Hauptfunktion sieht so aus:

```
1 fbm2 cls     []       _    erg rep =(cls,(rep,erg))
2 fbm2 cls     (x:xs)       max erg rep =if erg2 > max then fbm2 cls xs erg2 erg1 x
3                                       else fbm2 cls xs max erg rep where
4                                            erg1 = check (matches x cls) (length x)
5                                            erg2 = length erg1
```

Die Funktion *fmb2* ist identisch zu der Funktion *fbm1*, abgesehen von der Berechnung der Score. Wir erinnern uns das erg1 eine Liste ist, die die Postionen an denen das Wort im String endet, speichert. Sie enthält keine Überlappungen mehr. Unsere Score ist die Länge dieser Liste, d.h. die Anzahl an nicht überlappenden Wiederholungen.

Longest

Der Longest-Algorithmus findet den längsten Repeat.

```
1 initfbm3 gr = fbm3 ls (initgass2 ls) 1 [] [] where ls = cls gr
2
3 fbm3 cls         []       _    erg rep =(cls,(rep,erg))
4 fbm3 cls         (x:xs)   max erg rep =if erg2 > max then fbm2 cls xs erg2 erg1 x
5                                        else fbm2 cls xs max erg rep where
6                                             erg1 = check (matches x cls) erg2
7                                             erg2 = length x
```

Die Longest-Funktionen sind abgesehen von der Scoreberechnung identisch zu *fbm1*. Hier wird als Score die Länge des aktuellen Wortes verwendet (Zeile 7).

Jetzt haben wir Funktionen, die das nächste zu ersetzende Wort errechnen. Als nächstes müssen wir diese Ersetzungen durchführen.

6.4 Replace one match

Zuerst sehen wir uns die Initialisierungsfunktion an:

```
1 initrmis cls m n rep = split (rmis cls m rep (length rep) 0 n) 1
```

Die Funktion *initrmis* (replace matches in string) bekommt als Parameter:

- cls, einen String wie er von der Funktion *cls* erzeugt wird
- m, eine Liste mit den Positionen an denen der Repeat endet
- n, das Nichtterminal welches den Repeat ersetzen soll
- rep, den Repeat

Die Funktion initialisiert die Funktion *rmis* und splitet das Ergebnis in eine Liste von Produktionsrümpfen mittels der schon vorgestellten Funktion *split*.

Die Funktion *rmis* ersetzt in unserem cls-String alle Vorkommen des Repeats.

```
1 rmis xs      []      _   _   _   _ = xs
2 rmis []      _       _   _   _   _ = []
3 rmis (x:xs) (m:ms) rep len pos n =
4         if (m-len) == pos then n:(rmis (drop len (x:xs)) ms rep len (pos+len) n)
5                         else x:rmis xs (m:ms) rep len (pos+1) n
```

Sie hat folgende Parameter:

- (x:xs), aktuelles Zeichen im cls-String und Restliste
- (m:ms), nächste Position an der ein Repeat endet und Restliste
- rep, den Repeat
- len, Länge des Repeats
- pos, aktuelle Position im cls-String
- n, Nichtterminal welches den Repeat ersetzen soll

Die Funktion hat vier Fälle:

1. Es gibt keine weiteren Vorkommen des Repeats im String (Zeile 1). Wir hängen den restlichen cls-String hinten an.
2. Der String ist zu Ende (Zeile 2). (Sollte nicht vorkommen, da spätestens mit dem Ende des Strings auch alle Repeats eingesetzt sein sollten.)
3. An der derzeitigen Position fängt ein Repeat an (Zeile 3): Wir setzen unser Nichtterminal und entfernen den Repeat vom Anfang des cls-Strings. Danach machen wir rekursiv, mit dem nächsten Listenelement der Repeat-Liste, weiter.
4. An der derzeitigen Position fängt kein Repeat an (Zeile 4): Wir machen rekursiv mit dem nächsten Zeichen weiter.

Jetzt müssen wir noch die Funktionen *rmis* und *fbm* miteinander komponieren und aus der Liste mit Produktionsrümpfen wieder eine Grammatik machen.

Das zweite Problem lässt sich leicht lösen wenn die Grammatik, aus der der cls-String entstanden ist, noch bekannt ist:

```
1 createGr []                    []  = []
2 createGr ((Production a b):xs) (l:ls) = (Production a l): createGr  xs ls
```

Die Produktionsrümpfe werden der Reihe nach den linken Seiten der Produktionen zugewiesen. Dies ist kein Problem, da wir die Reihenfolge der Rümpfe nicht ändern.

Zuletzt die alles verbindende Funktion *minigr* (minimize grammar) die gleichzeitig die Grammatik solange reduziert bis der gewählte Algorithmus sie nicht weiter komprimieren kann.

6.5 Die alles verbindende Funktion

Wie immer eine Initialisierungsfunktion:

```
1 initminigr g fu = minigr g fu 1
```

Es wird eine Grammatik sowie einer der drei Algorithmen: Longest, Repair und Compress, übergeben.
Die Funktion *minigr* hat noch einen Parameter mehr. Diese Zählvariable wird dazu verwendet eindeutige Namen für die Nichtterminale zu erzeugen.

Die Hauptfunktion sieht so aus:

```
1 minigr g fu z = if spec == ([],[]) then g else minigr ng fu (z+1) where
2                     m  = fu g
3                     ls = fst m
4                     spec = snd m
5                     n = (N 1 ("N" ++ (show z)))
6                     ng = (Production n (fst spec)):
7                               createGr g (initRmis ls (snd spec) n (fst spec))
```

Zuerst die Hilfsfunktionen nach dem where-Statement:

- m (Zeile 2), berechnet das beste zu ersetzende Wort, mittels des ausgesuchten Algorithmus
- ls (Zeile 3), "befreit" den cls-String aus dem Ergebnis-Tupel von m
- spec (Zeile 3), enthält den Tupel: (Repeat, Liste von Postionen an denen der Repeat im cls-String vorkommt)
- n (Zeile 4), erzeugt das Symbol für ein neues Nichtterminal
- ng (Zeile 5), fügt die neue Grammatik zusammen aus:
  - der neuen Produktionsregel
  - der Grammatik in der alle Vorkommnisse des Repeats ersetzt wurden.

Wenn der Ergebnistupel spec leer ist, beendet die Funktion. Solange er ein bestes zu ersetzendes Wort enthält, wird dieses in der Grammatik ersetzt und dann mit der neuen Grammatik rekursiv weiter gerechnet.

7 Tests

Für das testen habe ich noch vier weitere Funktionen entworfen. Zuerst eine Funktion die aus Strings eine Grammatik erzeugt:

```
strtogram str = [Production (N (fromIntegral(length(str))) "S") (map (\x -> T x)
    str)]
```

Und dann drei Funktionen, die jeweils eine der IRRA-Varianten auf einem String ausführt

```
compress str    = initminigr (strtogram str) initfbm1
repair str      = initminigr (strtogram str) initfbm2
longest str     = initminigr (strtogram str) initfbm3
```

Ich habe die drei IRAA-Varianten mit einigen Texten aus dem Canterbury-Corpus [Cc] getestet. Die Dateien waren dabei nicht kompiliert.

7.1 Compress

Textdatei	Anzahl der Zeichen	Bearbeitungszeit	Verwendeter Speicher
grammar.lsp	3721	94.35 secs	57460666848 bytes
xargs.1	4227	197.89 secs	116841235400 bytes
fields.c	11150	4489.66 secs	2235435795188 bytes
cp.html	24603	14670.15 secs	6810808423392 bytes

7.2 Repair

Textdatei	Anzahl der Zeichen	Bearbeitungszeit	Verwendeter Speicher
grammar.lsp	3721	260.60 secs	155383879272 bytes
xargs.1	4227	331.74 secs	193482969976 bytes
fields.c	11150	7535.44 secs	3758903053444 bytes
cp.html	24603	29143.30 secs	5448646738713 bytes

7.3 Longest

Textdatei	Anzahl der Zeichen	Bearbeitungszeit	Verwendeter Speicher
grammar.lsp	3721	260.86 secs	155380555300 bytes
xargs.1	4227	337.09 secs	194575055936 bytes
fields.c	11150	7714.34 secs	3758903850275 bytes
cp.html	24603	29726.16 secs	5448592252245

8 Zusammenfassung

Angenommen meine Eingabe hat n Zeichen. Mittels des Suffixbaums, kann ich in linearer Zeit sämtliche Worte, die als Repeat in Frage kommen, erzeugen.
Mittels des Boyer-Moore-Algorithmus kann ich für jedes dieser Worte, in linearer Zeit eine Score, berechnen.
Somit kann ich in $O(n^2)$ das nächste zu ersetzende Wort finden.

Da ich nur n Zeichen habe, werde ich auf jeden Fall weniger als n Ersetzungen durchführen. Eine obere Schranke für die Laufzeit meines Programms ist also $O(n^3)$. Dies ist auch die, im Paper "The Smallest Grammar Problem as Constituents Choice and Minimal Grammar Parsing"[Car+10] genannte Schranke.

Eine derartige asymptotische Laufzeit ist für kleine n aktzeptabel. Für große n aber schon zu viel.
Beispiel:
Anzahl Rechenschritte: $25000^3 = 15625000000000$
Angenommen wir machen eine Milliarde Rechenschritte pro Sekunde dann brauchen wir
$15625000000000/1000000000/3600 \approx 4,3408$ Stunden um die Grammatik zu berechnen.

Dies deckt sich grob mit den Zeiten die mein Computer benötigt. Das Programm ist somit nicht zur Kompression geeignet.

Auch das kompilieren der .hs-Dateien bringt keine nennenswerte Verbesserung. Die konstanten Faktoren der Berechnung können, durch eine andere Programmiersprache oder Verbesserungen am Code, sicher noch stark gedrückt werden.

Literatur

[Bir] Richard Bird. *Pearls of Functional Algorithm Design*. Kapitel 16;

[Car+10] Rafael Carrascosaa u. a. *The Smallest Grammar Problem as Constituents Choice and Minimal Grammar Parsing*. Website. `http://www.irisa.fr/symbiose/images/stories/mgalle/papers/sgp_ccmpg.pdf`; 2010.

[Cc] *Canterbury Corpus*.

[SS] Prof. Schmidt-Schauß. *SCFG*. CTAN: `http://www.ki.informatik.uni-frankfurt.de/research/gbc/dist/doc/htmlsrc/Data/GBC/SCFG/SCFG.html`.

[St] *Data.SuffixTree*. CTAN: `http://hackage.haskell.org/packages/archive/suffixtree/0.2.2/doc/html/Data-SuffixTree.html`.

[Wika] *Boyer-Moore-Algorithmus*.

[Wikb] *Context-free grammar*. CTAN: `http://en.wikipedia.org/wiki/Context-free_grammar`.

[Wikc] *Kolmogorow-Komplexität*. CTAN: `http://de.wikipedia.org/wiki/Kolmogorow-Komplexitt`.

[Wikd] *Suffixbaum*. CTAN: `http://de.wikipedia.org/wiki/Suffixbaum`.