Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Fachgruppe Algorithmen und Komplexität
Fürstenallee 11
33102 Paderborn

Studienarbeit

Zur Approximierbarkeit des Halteproblems in einer praktischen Gödelisierung

Sven Köhler
Matrikelnummer 6089897
1. Dezember 2004

Betreuer: Dr. Martin Ziegler

Zusammenfassung

In dieser Arbeit wird die Approximierbarkeit des Halteproblems untersucht. Dazu werden zwei Gödelisierungen für die simple aber dennoch turingvollständige Programmiersprache BF vorgestellt. Bei beiden Gödelisierungen ist die Konvertierung zwischen Quelltexten und Gödelnummern in Polynomialzeit möglich. Eine der beiden Gödelisierungen ist für die Praxis besonders relevant. Von ihr kann gezeigt werden, dass sie — im Gegensatz zur anderen — dicht ist. Daraus folgt, dass hier das Halteproblem nicht einmal approximativ gelöst werden kann.

Inhaltsverzeichnis

1 Einleitung

Eine zentrale Thematik der Informatik ist das Halteproblem. Gegenstand des Halteproblems ist es, zu entscheiden, ob das durch die Eingabe repräsentierte Programm hält oder nicht. Ein Algorithmus, der das Halteproblem löst, könnte die Qualität von Software steigern, indem keine Programme ausgeliefert werden, die potentielle Endlosschleifen enthalten. Es ist aber bekannt, dass es keinen Algorithmus gibt, der das Halteproblem für alle Eingaben entscheidet. Dennoch könnten Approximationen des Halteproblems existieren. Approximation ist hinsichtlich Entscheidungsproblemen so zu verstehen: ein Algorithmus, der das Halteproblem approximiert, darf bei einem gewissen Teil der möglichen Eingaben ein falsches Ergebnis liefern, d.h. eine Eingabe wird akzeptiert statt abgelehnt oder umgekehrt. (Tatsächlich lässt sich stets ein einseitiger Fehler erreichen.)
Die erreichbare Approximationsgüte, d.h. der prozentuale Anteil an Fehlern, hängt aber stark von der gewählten Gödelisierung ab. So kann eine Gödelisierung so gewählt sein, dass der Anteil an entscheidbaren Programmen überwiegt.
Schindelhauer und Jakoby führen 1999 in ihrer Arbeit „*The Non-Recursive Power of Erroneous Computation*“ [1] den Begriff der Dichtheit ein. Aus der Dichtheit einer Gödelisierung folgt eine konstante untere Schranke für die Fehlerrate der Approximation des Halteproblems. Schindelhauer und Jakoby weisen auch darauf hin, dass gängige Programmiersprachen dicht seien, geben allerdings kein Beispiel/Beweis für eine entsprechende konkrete Gödelisierung an.

In Kapitel 3 dieser Arbeit werden zwei Gödelisierungen für die turingvollständige Programmiersprache BF (siehe Kapitel 2) vorgestellt: eine recht naheliegende, naive Gödelisierung und eine sogenannte praktische Gödelisierung, die nur syntaktisch korrekten Programmtexten eindeutig eine Zahl zuordnet. Die letztere Gödelisierung ist deswegen besonders relevant für die Praxis, da sowieso nur syntaktisch korrekte Programme untersucht werden müssen.

Für beide Gödelisierungen werden Polynomialzeitalgorithmen zur Konvertierung zwischen Programmtexten und Gödelnummern vorgestellt. In Kapitel 4 wird sich herausstellen, dass die praktische Gödelisierung dicht ist und damit eine konstante untere Schranke für die Fehlerrate der Approximationen des Halteproblems existiert. Die naive Gödelisierung ist hingegen nicht dicht; dieser „Vorteil“ ist in der Praxis aber nicht von Nutzen. Somit ist das Halteproblem nicht approximierbar.

2 Die Programmiersprache BF

Die Programmiersprache BF (BrainF**k) wurde 1993 von Urban Müller entwickelt. Es gibt lediglich 8 Befehle, die jeweils durch ein Symbol aus dem Alphabet {`<`, `>`, `+`, `-`, `.`, `,`, `[`, `]`} repräsentiert werden. Die Befehle `[` und `]` sind zugleich Klammersymbole mit denen einfache Schleifen realisiert werden können.
An Speicher steht einem BF-Programm ein Arbeitsband zur Verfügung, welches aus Zellen besteht, die Werte zwischen 0 und 255 speichern können. Das Band ist dabei nach beiden Seiten unbeschränkt. Einem BF-Programm stehen also beliebig viele Zellen auf dem Band zur Verfügung. Allerdings kann immer nur die Zelle gelesen bzw. beschrieben werden, die sich unter dem Lese-/Schreibkopf befindet. Dieser Lese-/Schreibkopf kann von einem BF-Programm in beide Richtungen des Bandes verschoben werden.
Zwei der BF-Befehle sind explizit für die Eingabe und die Ausgabe zuständig. Der Wert der Zelle unter dem Kopf des Ausgabebandes kann entweder mit einem Eingabewert überschrieben oder ausgegeben werden.
Wird ein Programm gestartet, so ist in allen Zellen des Arbeitsbandes eine 0 gespeichert. Die Ausführung beginnt bei dem ersten Befehl des BF-Programms. In der Regel wird nach

der Ausführung jedes einzelnen Befehls zum nachfolgenden Befehl gesprungen. Ausnahmen bilden hier nur die Klammersymbole. Ein BF-Programm terminiert, sobald hinter den letzten Befehl des BF-Programms gesprungen wird.
Die Syntax von BF-Programmen ist einfach. Die einzigen Anforderungen an syntaktisch korrekte BF-Programme sind:

- das Programm darf nur die 8 Symbole des Alphabets enthalten
- die mit den beiden Klammersymbolen gebildete Klammerung muss korrekt sein

Mit einer korrekten Klammerung ist hier gemeint, was auch in der Mathematik oder anderen Programmiersprachen darunter verstanden wird. So sind Klammerungen wie `[][]` oder `[[]]` korrekt, Klammerungen wie `][[]` oder `[]]` sind es nicht. Formal kann eine korrekte Klammerung so definiert werden:

Definition 2.1. *Eine Klammerung ist genau dann korrekt, wenn folgendes gilt:*

- *es sind gleich viele öffnende und schließende Klammern vorhanden*
- *geht man vom ersten zum letzten Symbol, so muss die Anzahl der bisher gezählten öffnenden Klammern immer größer-gleich der Anzahl der bisher gezählten schließenden Klammern sein*

Die hier vorgestellte Definition von BF weist einige Besonderheiten auf:

1. Das Arbeitsband ist nicht wie in der gängigen Definition [6] auf 30000 Zellen beschränkt. BF-Programme mit einem endlichen Arbeitsband stellen gegenüber Turingmaschinen ein wesentlich einfacheres Modell dar. Für solche BF-Programme ist das Halteproblem entscheidbar, da es nur endlich viele Zustände bzw. Konfigurationen gibt. Damit existiert eine obere Schranke für die Laufzeit haltender Programme.
2. Die Eingabe wird nicht wie bei den meisten BF-Interpretern Symbol für Symbol vom Benutzer abgefragt. Die Modellvorstellung soll vielmehr sein, dass jeder Eingabebefehl ein Symbol aus einer Quelle liest. Die Quelle muss dabei vor dem Start des BF-Programms mit einer Folge von Eingabesymbolen, also einem Eingabewort, initialisiert werden. Damit ist die Eingabe vor dem Start fixiert und entscheidet sich nicht während der Laufzeit.

Tabelle 1: Befehle der Programmiersprache BF

Symbol	Bedeutung
`>`	der Lese-/Schreibkopf wird um eine Zelle nach rechts bewegt
`<`	der Lese-/Schreibkopf wird um eine Zelle nach links bewegt
`+`	der Wert der aktuellen Zelle wird um 1 inkrementiert, aus 255 wird dabei 0
`-`	der Wert der aktuellen Zelle wird um 1 dekrementiert, aus 0 wird dabei 255
`.`	der Wert der aktuellen Zelle wird zur Ausgabe hinzugefügt
`,`	der Wert der aktuellen Zelle wird mit dem nächsten Eingabewert überschrieben
`[`	Sprung zum Befehl nach der passenden `]`, falls die aktuelle Zelle den Wert 0 enthält
`]`	Sprung zur passenden `[`

2.1 BF ist turingvollständig

In diesem Abschnitt soll gezeigt werden, dass BF-Programme und Turingmaschinen gleich mächtig sind. Das würde bedeuten, dass BF-Programme alle Probleme lösen können, die auch von Turingmaschinen gelöst werden können. Zur Erinnerung hier nochmal die Definition einer Turingmaschine:

Definition 2.2. [2, KAPITEL 8.2.2, KAPITEL 8.4.1]
Eine k-Band Turingmaschine M ist ein 7-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$. *Es gilt:*

- Q *ist endliche Zustandsmenge*
- $\Sigma \subsetneq \Gamma$ *ist endliches Eingabealphabet*
- Γ *ist endliches Bandalphabet*
- $q_0 \in Q$ *ist Anfangszustand*
- $\sqcup \in \Gamma \setminus \Sigma$ *ist Symbol für leeres Feld*
- $\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, N, R\})^k$ *ist Übergangsfunktion*
- $F \subseteq Q$ *Menge der Endzustände*

Dazu muss gesagt werden, dass jede k-Band in eine 1-Band Turingmaschine umgewandelt werden kann. Durch Erhöhung der Anzahl der Bänder werden Turingmaschinen nicht mächtiger [2, THEOREM 8.9]. Es kann also ohne Bedenken eine Turingmaschine mit k Bändern konstruiert werden, selbst wenn diese gegenüber einer entsprechenden 1-Band Turingmaschine einfacher zu konstruieren ist. Ähnliches gilt auch für Turingmaschinen mit großem Bandalphabet. Diese können in Turingmaschinen mit kleinerem Bandalphabet umgewandelt werden.

Um im Folgenden komplexe BF-Programme besser beschreiben zu können, werden hier erst einmal ein paar einfache Bausteine für BF-Programme vorgestellt. Mit den recht einfachen Befehlen der Sprache BF wären komplexe Programme sonst nur schwer vorstellbar.

- Initialisierer: `[-]`
  Diese kurze Schleife dekrementiert den Wert der aktuellen Zelle solange, bis sie den Wert 0 enthält. Durch Anhängen von entsprechend vielen Inkrementier-Befehlen kann auch jede andere Konstante erzeugt werden.
- Addierer 1: `[->+<]`
  Diese Schleife addiert den Wert der aktuellen Zelle i auf die Zelle $i+1$. Dazu wird der Inhalt der Zelle i dekrementiert, bis diese 0 enthält. In der Schleife wird aber auch der Wert der Zelle $i+1$ inkrementiert, so dass diese im Endeffekt die Summe der beiden Zellen enthält.
  Nebeneffekt: die Zelle i hat nach dem Addieren den Wert 0.
- Kopierer 1: `>[-]<[->+<]`
  Dieser Kopierer initialisiert die Ziel-Zelle zuerst mit 0, um dann mit Addierer 1 den Wert aus Zelle i in die Zelle $i+1$ zu kopieren.
- Addierer 2: `>[-]<[->+>+<<]>[-<+>]<`
  Hier wird der Wert der aktuellen Zelle i auf die Zelle $i+2$ addiert. Der Wert der Zelle i bleibt allerdings erhalten. Dazu wird zuerst die Zelle $i+1$ mit 0 initialisiert. Danach wird der Wert der Zelle i auf die Zellen $i+1$ und $i+2$ addiert. Dann wird der Wert von Zelle $i+1$ zurück in die Zelle i kopiert.
  Nebeneffekt: Zelle $i+1$ enthält nach dem Addieren den Wert 0.

- Kopierer 2: `>>[-]<[-]<[->+>+<<]>[-<+>]<`
  Zuerst wird die Zelle $i+2$ mit 0 initialisiert, um dann mit Addierer 2 den Inhalt der Zelle *i* in die Zelle $i+2$ zu kopieren.

- Bedingte Anweisung der Form *if* $(Zelle \neq 0)$ *then* ...
  Mit Hilfe einer einfachen Schleife kann Code nur dann ausgeführt werden, wenn die aktuelle Zelle *i* einen Wert ungleich 0 enthält. Da der Code in der Schleife allerdings maximal einmal ausgeführt werden soll, initialisiert man einfach die Zelle *i* während des ersten Schleifendurchlaufs mit 0. Die Schleife wird dadurch abgebrochen. Da dabei die Zelle *i* eventuell gelöscht wird, sollte man diese vorher kopieren.

- Bedingte Anweisung der Form *if* $(Zelle = 0)$ *then* ...
  Dies lässt sich mit zwei bedingten Anweisungen der Form $Zelle \neq 0$ lösen, indem man eine Zelle *j* mit 1 initialisiert und diese genau dann auf 0 setzt, wenn die aktuelle Zelle *i* einen Wert ungleich 0 enthält. Die Zelle *j* enthält sozusagen die Invertierung der Zelle *i*. Setzt man nun Code in eine bedingte Anweisung, die nur dann ausgeführt wird, wenn die gerade modifizierte Zelle *j* ungleich 0 ist, so wird der Code wie gewünscht nur dann ausgeführt, wenn die Zelle *i* den Wert 0 enthält.

- Bedingte Anweisungen mit Überprüfung auf eine Konstante
  Man kann nicht nur testen, ob eine Zelle eine 0 enthält oder nicht. Man kann auch auf Konstanten testen, indem man die zu testende Zelle zuerst um die Konstante dekrementiert.

- Zahlen über mehrere Zellen speichern
  Man kann sich klar machen, dass es leicht möglich ist, nicht nur Zahlen zwischen 0 und 255 zu speichern, sondern auch Zahlen, die mehrere Zellen in Anspruch nehmen. Dazu muss lediglich nach einer Inkrementierung der Zelle einer Zahl lediglich überprüft werden, ob das Ergebnis eine 0 ist, denn dann hat ein Überlauf stattgefunden. Bei einem Überlauf inkrementiert man einfach die nächst höherwertige Zelle der Zahl. Beschränkt man dieses Verfahren z.B. auf 4 Zellen, so stehen 32-Bit Zahlen zur Verfügung.

Einige dieser BF-Bausteine benötigen temporären Speicher. So setzen Addierer 2 und Kopierer 2 den Inhalt einer zusätzlichen Zelle auf 0. Beim Reservieren von Speicherzellen für Variablen sollte man daher darauf achten, dass zwischen den Zellen etwas Platz ist, der temporär verwendet werden kann. Die obigen BF-Bausteine können außerdem durch mehr Verschiebe-Befehle (< und >) so angepasst werden, dass die temporär genutzen Zellen weiter entfernt sind.

Satz 2.3. *BF ist turingvollständig, das heißt:*

1. *für jedes BF-Programm gibt es eine äquivalente Turingmaschine*
2. *für jede Turingmaschine gibt es ein äquivalentes BF-Programm*

Beweisskizze.

1. BF-Programme haben starke Ähnlichkeit zu Turingmaschinen, denn sie arbeiten ebenfalls auf einem Arbeitsband. Turingmaschinen werden aber nicht durch einen Programmcode gesteuert, sondern durch die Übergangsfuntkion δ. Jedes syntaktisch korrekte BF-Programm lässt sich allerdings in die Übergangsfunktion einer 3-Band Turingmaschine mit Eingabealphabet $\Sigma = \{0 \dots 255\}$ und Bandalphabet $\Gamma = \Sigma \cup \{\sqcup\}$ umwandeln. Tabelle 2 zeigt wie.
   Band 1 soll dabei das Band sein, von dem die Turingmaschine das Eingabewort liest. Band 2 ist das Band auf dem die Turingmaschine ihre Ausgabe hinterlässt.

Tabelle 2: Übersetzung von BF-Befehl in Übergangsfunktion

Befehl	Übergangsfunktion für Zustand q_i
>	Der Kopf des Arbeitsbandes wird nach rechts bewegt.
<	Der Kopf des Arbeitsbandes wird nach links bewegt.
+	Der Wert v der aktuellen Zelle des Arbeitsbandes wird mit dem Wert $w = v + 1 \bmod 256$ überschrieben
-	Der Wert v der aktuellen Zelle des Arbeitsbandes wird mit dem Wert $w = v - 1 \bmod 256$ überschrieben
.	Der Wert in der aktuellen Zelle des Ausgabebandes wird mit dem Wert der aktuellen Zelle des Arbeitsbandes überschrieben. Der Kopf des Ausgabebandes wird danach nach rechts bewegt
,	Der Wert in der aktuellen Zelle des Arbeitsbandes wird mit dem Wert der aktuellen Zelle des Eingabebandes überschrieben. Der Kopf des Eingabebandes wird danach nach rechts bewegt
[	Ist der Wert der aktuellen Zelle $\neq 0$, so ist der Nachfolgezustand q_{i+1}, ansonsten ist der Nachfolgezustand der, der zu dem Symbol nach dem passenden]-Symbol gehört
]	Nachfolgezustand ist der Zustand, der zu dem passenden [-Symbol gehört

Band 3 wird als Arbeitsband genutzt.
Bei der Umwandlung entsteht aus jedem Symbol im BF-Programm ein neuer Zustand q_i. Der Nachfolgezustand des Zustands q_i ist dabei im Normalfall q_{i+1} und die Köpfe der Bänder werden nicht bewegt, falls nicht anders angegeben.
Das Symbol $\sqcup$ für das leere Feld sollte immer wie eine 0 behandelt werden, so dass das Verhalten, insbesondere bei noch leerem Band, dem von BF-Programmen entspricht. Darüber hinaus ist die Konstruktion so simpel, dass diese Umwandlung von einer Turingmaschine berechnet werden kann. Verkettet man diese mit einer entsprechenden universellen Turingmaschine, d.h. konstruiert man eine Turingmaschine, die aus einem BF-Programm die Übergangsfunktion berechnet und diese dann anschließend simuliert, so erhält man einen BF-Interpreter, der ein beliebiges BF-Programm simulieren kann.

2. Nun wird eine Turingmaschine in ein BF-Programm umgewandelt. Diese Umwandlung wird Ansätze einer Simulation enthalten. Diese eignet sich nicht um beliebige Turingmaschinen zu simulieren. Durch die Umwandlung einer universellen Turingmaschine erhält man allerdings einen Simulator für beliebige Turingmaschinen. Weiterhin kann man sich auch auf 1-Band Turingmaschinen mit dem Bandalphabet $\{0 = \sqcup, 1\}$ beschränken. Diese Beschränkung macht die Umwandlung wesentlich einfacher.
   Zunächst teilt man das Arbeitsband in 3 Teile. In einem Teil wird die Übergangsfunktion der Turingmaschine gespeichert. Hinzu kommen noch einige temporäre Zellen, die für Addierer, Kopierer und Vergleiche gebraucht werden. Die restlichen beiden Teile des Bandes links und rechts von der Übergangsfunktion dienen zur Speicherung des Arbeitsbandes der Turingmaschine. Durch diese Aufteilung kann das ebenfalls nach beiden Seiten unendliche Band der Turingmaschine leicht simuliert werden. Es wird allerdings nur jede zweite Zelle des Arbeitsbandes für die Speicherung des Bandes verwendet. Die anderen Zellen werden dazu genutzt, um die Zelle zu markieren, auf der der Lese-/Schreibkopf der Turingmaschine steht. Start und Ende des Bereiches in dem die Übergangsfunktion gespeichert ist, werden außerdem durch

Markierungen in Form von Zellen mit speziellen Werten kenntlich gemacht. Somit lässt sich die Übergangsfunktion auf dem Band leicht wiederfinden.
Das resultierende BF-Programm selbst, lässt sich auch in 3 Teile aufteilen. Zuerst wird das Arbeitsband mit einer bestimmten Codierung der Übergangsfunktion gefüllt. Diese Codierung der Übergangsfunktion wird durch entsprechende Initialisierer fest in das BF-Programm integriert. Danach wird die Eingabe eingelesen und auf die Bandbereiche übertragen, die zur Speicherung des Bandes der Turingmaschine dienen. Die Eingabe sollte mit einem geeignetem Symbol abgeschlossen werden, damit das BF-Programm weiss, wann die Eingabe endet. Im dritten Teil wird nun in einer Schleife die Übergangsfunktion angewendet bis einer der Endzustände erreicht ist.
Um die Übergangsfunktion einfach anwenden zu können, wird diese speziell codiert. Die Zustände werden durchnummeriert und in dieser Reihenfolge auf das Band geschrieben. Dabei werden für jeden Zustand gleich viele Zellen in Anspruch genommen in denen gespeichert wird, ob es sich um den aktuellen Zustand handelt, ob es sich um einen Endzustand handelt und welche Nummer der Nachfolgezustand hat. Zusätzlich muss pro möglichem Symbol des Bandalphabets $0,1$ gespeichert werden, welcher Wert auf das Band zurückgeschrieben wird und in welche Richtung der Lese-/Schreibkopf verschoben wird.
Der Algorithmus, der die Übergangsfunktion anwendet, sucht als erstes die markierte Zelle über der der Lese-/Schreibkopf der Turingmaschine steht. Danach wird der markierte Zustand gesucht. Je nach Inhalt der markierten Zelle wird dieser anders ausgewertet: die markierte Zelle wird mit dem vorgegebenen Wert überschrieben und die Markierung wird in die angegebene Richtung verschoben. Jetzt wird die Markierung des Zustands entfernt und der entsprechende Nachfolgezustand markiert. Um zu wissen auf welcher Seite des Bandes die markierte Zelle zu suchen ist, muss dies zusätzlich abgespeichert werden.

2.2 Formale Sprache der BF-Programme

Im Folgenden wird die formale Sprache der syntaktisch korrekten BF-Programme betrachtet, d.h. es werden die Mengen von Wörtern betrachtet, die syntaktisch korrekten BF-Programmen entsprechen. Diese Mengen bzw. deren Größe werden in den folgenden Kapiteln benötigt.

Definition 2.4. *$\mathcal{BF}$ bezeichnet die Menge aller syntaktisch korrekten BF-Programme.*
$\mathcal{BF}_n$ bezeichnet die Menge aller syntaktisch korrekten BF-Programme der Länge n.
$\mathcal{BF}_{n,k}$ bezeichnet die Menge aller syntaktisch korrekten BF-Programme der Länge n mit k Klammerpaaren (für $k = 0 \dots \lfloor \frac{n}{2} \rfloor$). BF_n und $BF_{n,k}$ bezeichnen die Kardinalitäten der entsprechenden Mengen. Es gilt:

- $\mathcal{BF}_n := \dot{\bigcup}_{k=0}^{\lfloor \frac{n}{2} \rfloor} \mathcal{BF}_{n,k}$
- $\mathcal{BF} := \dot{\bigcup}_{n \in \mathbb{N}_0} \mathcal{BF}_n$
- $BF_{n,k} := \left|\mathcal{BF}_{n,k}\right|$
- $BF_n := |\mathcal{BF}_n|$

Hier ist intuitiv klar, dass die $\mathcal{BF}_n$ sowie die $\mathcal{BF}_{n,k}$ jeweils untereinander disjunkt sind. Daher darf hier die disjunkte Vereinigung $\dot{\bigcup}$ verwendet werden. Damit sind die Mengengleichungen direkt auf die Kardinalitäten übertragbar, indem die disjunkte Vereinigung durch Summierung ersetzt wird:

$$BF_n = |\mathcal{BF}_n| = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} BF_{n,k}$$

Um nun die $BF_{n,k}$ und damit die BF_n berechnen zu können, brauchen wir zunächst folgende Definition:

Definition 2.5. [4] *Die Catalansche Zahl* $C_k = \frac{1}{k+1}\binom{2k}{k} = \frac{(2k)!}{k!(k+1)!}$ *ist die Anzahl der Folgen von* 1 *und* -1*, die folgende Kriterien erfüllen:*

- *die Folge muss k-viele* 1 *und* -1 *enthalten*
- *geht man vom ersten Symbol zum letzten Symbol der Folge und summiert diese der Reihe nach auf, so darf keine der Zwischensummen negativ sein*

Diese Definition weist eine starke Ähnlichkeit zu Defintion 2.1 einer korrekten Klammerung auf, wenn man öffnende Klammern durch die 1 und schließende durch -1 ersetzt. In der Tat ist C_k genau die Anzahl der korrekten Klammerungen mit k-Klammerpaaren.

Hier einige Beispiele:

$C_0 = 1$
$C_1 = 1$ `[]`
$C_2 = 2$ `[][]`, `[[]]`
$C_3 = 5$ `[][][]`, `[][[]]`, `[[]][]`, `[[][]]`, `[[[]]]`
...

$BF_{n,k}$ kann nun wie folgt berechnet werden:

Satz 2.6. *Für die Anzahl der BF-Programme der Länge n mit k Klammerpaaren gilt:*

$$\begin{aligned} BF_{n,k} &= 6^{n-2k} \cdot C_k \cdot \binom{n}{2k} \\ &= 6^{n-2k} \cdot \frac{n!}{k!(k+1)!(n-2k)!} \end{aligned}$$

Beweis. Man stelle sich das BF-Programm in Form von n leeren Feldern vor, die es nun zu füllen gilt. Die Anzahl der Möglichkeiten dafür hängt von 3 Faktoren ab:

1. welche der n Felder werden mit den $2k$-vielen Klammersymbolen gefüllt
2. wie viele Möglichkeiten gibt es, die $n-2k$ Felder zu füllen, die nicht mit Klammersymbolen gefüllt werden
3. wie viele Möglichkeiten gibt es, die $2k$ Felder mit Klammersymbolen zu füllen, ohne dass die Syntax verletzt wird

Es werden nun $2k$-viele der n Felder ausgewählt. Dafür gibt es $\binom{n}{2k}$-viele verschiedene Möglichkeiten. Die Restlichen $n-2k$ Felder werden mit den normalen 6 Befehlen gefüllt. Dafür gibt es genau 6^{n-2k} verschiedene Möglichkeiten. Die $2k$ übrigen Felder im Programm müssen nun mit einer syntaktisch korrekten Klammerung gefüllt werden. Dafür gibt es nach Definition 2.5 genau C_k viele Möglichkeiten.
Jedes so erzeugte BF-Programm ist syntaktisch korrekt und jedes Programm in $\mathcal{BF}_{n,k}$ kann eindeutig auf diese Weise erzeugt werden. Multipliziert man also die 3 Faktoren, so ergibt sich wie behauptet:

$$BF_{n,k} = 6^{n-2k} \cdot C_k \cdot \binom{n}{2k}$$

□

Mit der Formel für $BF_{n,k}$ können nun auch die BF_n berechnet werden.
Ein paar Beispielwerte:

$BF_0 = 1$	Es gibt nur ein Programm der Länge 0: das leere Wort
$BF_1 = 6$	Noch keine Klammern möglich, deswegen nur 6 Möglichkeiten
$BF_2 = 37$	6^2 Möglichkeiten mit den normalen Befehlen + 1 Klammerung
$BF_3 = 234$	
$\dots$	

Intuitiv ist klar, dass $6^n \leq BF_n \leq 8^n$ gelten muss. Schließlich gibt es schon allein 6^n BF-Programme ohne Klammern. 8^n BF-Programme kann es nicht geben, da nicht alle Programme eine von der Syntax geforderte korrekte Klammerung aufweisen.
Für Kapitel 3 sind die Beziehungen der Mengen $\mathcal{BF}_n$ untereinander wichtig. Hier werden einige wichtige Ansätze vorgestellt, die es erlauben ein Wort aus $\mathcal{BF}_n$ in Worte kleinerer Länge zu zerteilen. Daraus ergibt sich ebenfalls eine Rekursionsgleichung für die BF_n. Diese lässt sich dazu nutzen, viele BF_n für aufeinander folgende n zu berechnen. Die Rekursionsgleichung zu nutzen, wird sich als effizienter erweisen, als jedes BF_n mit der in Definition 2.4 vorgestellten Summe zu berechnen. Die Summanden, also die $BF_{n,k}$, enthalten viele Fakultäten, die es sehr teuer machen, die Summe auszurechnen.
Damit beim Zerteilen von BF-Programmen die Syntax nicht verletzt wird, muss man zunächst besonders auf die Klammerungen eingehen.

Beobachtung 2.7. *Jede korrekte Klammerung x mit mindestens einem Klammerpaar lässt sich in die korrekten Klammerungen u und v zerlegen bzw. aus ihnen zusammensetzen, so dass $x = [\circ u \circ] \circ v$ gilt. Diese Zerlegung ist darüber hinaus eindeutig.*

Der Operator $\circ$ meint hier das symbolweise Konkatenieren von Wörtern und Symbolen. Da die Zerlegung eindeutig ist, gibt es nur eine schließende Klammer, an der das Wort x in u und v getrennt werden kann. Es muss die schließende Klammer sein, bei der das erste mal gleich viele öffnende und schließende Klammern gezählt werden, während man vom ersten zum letzten Symbol von x geht.
Diese Zerlegung lässt sich im übrigen auch auf BF-Programme übertragen, was zu folgendem Lemma führt:

Lemma 2.8. *Es gelten folgende Rekursionsgleichungen für $\mathcal{BF}_n$ bzw. BF_n:*

1. $\mathcal{BF}_{n+1} = \left(\dot{\bigcup}_{s\in\{<,>,+,-,.,,\}} s \circ \mathcal{BF}_n\right) \dot{\cup} \left(\dot{\bigcup}_{i=1}^{n} [\circ \mathcal{BF}_{i-1} \circ] \circ \mathcal{BF}_{n-i}\right)$
2. $BF_{n+1} = 6 \cdot BF_n + \sum_{i=1}^{n} BF_{i-1} \cdot BF_{n-i}$
3. $(n+3) \cdot BF_{n+1} = (12n+18) \cdot BF_n - 32n \cdot BF_{n-1}$

Gleichungen 1 und 2 gelten jeweils für $n \geq 1$, $\mathcal{BF}_0 = \{\varepsilon\}$ und $BF_0 = 1$.
Gleichung 3 gilt für $n \geq 2$ und $BF_0 = 1, BF_1 = 6$.

Beweis.

1. Diese Gleichung beschreibt eine Zerlegung eines syntaktisch korrekten BF-Programms der Länge $n+1$ in syntaktisch korrekte BF-Programme kleinerer Länge. Diese Zerlegung entsteht durch Abschneiden des ersten Symbols. Der Mengenäquivalentsbeweis wird nun in den typischen zwei Schritten durchgeführt:

   - Zunächst wird gezeigt, dass jedes BF-Programm auf die beschriebene Weise zerlegt werden kann. Schneidet man das erste Symbol eines BF-Programms ab, so kann dies gemäß der Syntax nicht] sein, sondern nur einer der übrigen 7 Befehle. Ist das erste Symbol nicht [, so ist die Sache einfach: es bleibt wie behauptet ein syntaktisch korrektes Programm der Länge n über. Ist das

erste Symbol allerdings [, so würde nach dem Abschneiden ein BF-Programm entstehen, welches ein]-Symbol zu viel enthält. Analog zu Beobachtung 2.7 teilt man nun das BF-Programm zusätzlich an der Position i des passenden]-Symbols. Zwischen dem ersten und dem i-ten Symbol liegt nun ein syntaktisch korrektes BF-Programm. Ein weiteres syntaktisch korrektes BF-Programm erstreckt sich vom $i+1$-ten Symbol bis zum letzten Symbol. Man erhält also zwei syntaktisch korrekte Programme der Längen $i-1$ und $n-i$.

- Nun muss gezeigt werden, dass jedes gemäß diesen Mengenvereinigungen zusammengesetzte Programm wieder ein syntaktisch korrektes Programm der Länge $n+1$ darstellt. Vor jedes beliebige BF-Programm der Länge n kann man einen der 6 normalen Befehle setzen, ohne die Syntax zu verletzen. Somit ist also die erste Häfte der Gleichung abgedeckt. Seien nun w_1 und w_2 zwei BF-Programme der Länge $i-1$ und $n-i$ (für $i = 1 \dots n$). Das Programm $[\circ w_1 \circ] \circ w_2$ hat die Länge $n+1$ und ist gemäß Beobachtung 2.7 wieder syntaktisch korrekt, da die einzige Anforderung der Syntax eine korrekte Klammerung ist.

2. Da die im Beweis von Gleichung 1 beschriebene Zerlegung eindeutig ist, sind die Mengenvereinigungen in Gleichung 1 wieder disjunkt. Daher kann die Mengengleichung wieder direkt auf die Kardinalitäten übertragen werden.

3. Diese Gleichung kann per vollständiger Induktion über n bewiesen werden.

□

3 Die Gödelisierung

In diesem Kapitel werden 2 Gödelisierungen für BF-Programme vorgestellt. Gödelisierungen sind Grundvoraussetzung für eine ganze Klasse von Algorithmen und Problemen: jeder Algorithmus bzw. jedes Problem, welches einen anderen Algorithmus als Eingabe fordert, wird in der theoretischen Informatik formal so definiert, dass die Eingabe eine natürliche Zahl ist, welche den Algorithmus repräsentiert. Diese natürliche Zahl ist das Ergebnis einer Gödelisierung. Der Begriff Gödelisierung ist formal wie folgt definiert:

Definition 3.1. [3, THEOREM II.1.5, PROPOSITION II.1.7]
Eine Gödelisierung $\varphi = (\varphi_e)_{e \in \mathbb{N}}$ ist eine Folge von Funktionen so dass gilt:

- *für jede Gödelnummer e ist φ_e eine partielle turingberechenbare Funktion*
- *ist ψ eine partielle turingberechenbare Funktion, dann existiert eine Gödelnummer e, so dass $\varphi_e = \psi$ gilt*
- *es gibt eine partielle turingberechenbare Funktion φ_u, so dass für alle x gilt:* $\varphi_u(e,x) = \varphi_e(x)$
- *die Abbildung $(e,x) \mapsto e'$ ist turingberechenbar und für alle e,x und y gilt:* $\varphi_{e'}(y) = \varphi_e(x,y)$

Unter einer partiellen turingberechbaren Funktion versteht man die Funktion, die von einer Turingmaschine oder einem Programm einer turingvollständigen Programmiersprache berechnet wird. Die Definition fordert also, dass es eine universelle Turingmaschine geben muss, die bei Eingabe einer Gödelnummer das gödelisierte Programm simulieren kann. Darüberhinaus muss es auch eine Turingmaschine geben, die aus einer Konstanten x und der Gödelnummer e eine neue Gödelnummer e' berechnet, wobei in das Programm $\varphi_{e'}$ gegenüber dem Programm φ_e die Konstante x bereits einkodiert ist. Dies kann realisiert werden, indem die Gödelnummer e in die Übergangsfunktion einer Turingmaschine zurückgerechnet wird und diese dann durch Zustände ergänzt wird, die die Konstante x auf das

Eingabeband schreiben bevor die Teile der ursprünglichen Übergangsfunktion ausgeführt werden. Danach wird die neue Übergangsfunktion in die Gödelnummer e' umgerechnet.

3.1 Naiver Ansatz

Für BF-Programme bietet sich ein naiver Ansatz zur Gödelisierung an. Allen 8 möglichen BF-Befehlen ordnet man eine Zahl zwischen 0 und 7 zu. Ein BF-Programm der Länge n kann dann als Oktalzahl mit n Ziffern betrachtet werden. Diese Oktalzahl liegt dann in dem Intervalle $[0, 8^n - 1]$. Die Intervalle überschneiden sich allerdings für verschiedene n. Um dies zu korrigieren, addiert man auf die Oktalzahl die Größe aller Intervalle für BF-Programme kleinerer Länge auf. Die Größe eines Intervalls für die BF-Programme der Länge n ist dabei 8^n. Die folgende Tabelle zeigt die resultierenden Intervalle für BF-Programme bestimmter Länge:

Länge		Oktalzahl		Gödelisierung
0	$\mapsto$	$[0,0]$	$\mapsto$	$[0,0]$
1	$\mapsto$	$[0,7]$	$\mapsto$	$[1,8]$
2	$\mapsto$	$[0,63]$	$\mapsto$	$[9,72]$
3	$\mapsto$	$[0,511]$	$\mapsto$	$[73,584]$
				$\dots$

Diese Gödelisierung $\tilde{\varphi}$ entspricht also einer Aufzählung aller BF-Programme sortiert nach Länge und lexikographischer Ordnung. Dass dabei auch syntaktisch inkorrekte Programme in der Gödelisierung vorkommen ist kein Problem. Diese werden üblicherweise als jene partielle Funktion aufgefasst, die für keine Eingabe definiert ist.
Dass diese Gödelisierung allerdings den syntaktisch inkorrekten Programmen eine Zahl zuordnet ist unnötig. Die entsprechende partielle Funktion, die für keine Eingabe definiert ist, kann auch durch das syntaktisch korrekte Programm repräsentiert werden, welches eine simple Endlosschleife enthält. In Kapitel 4 kann aber gezeigt werden, dass diese Gödelisierung nicht dicht ist. Daher wird nun eine weitere Gödelisierung vorgestellt.

3.2 Praktischer Ansatz

Die nun vorgestellte Gödelisierung φ ist praktisch besonders relevant, da die Tatsache berücksichtigt wird, dass in der Praxis viele Probleme nur für syntaktisch korrekte Programme betrachtet werden müssen. Die Gödelisierung ordnet jedem dieser Programme eindeutig eine Zahl zu, so dass sie eine Bijektion zwischen $\mathcal{BF}$ und $\mathbb{N}_0$ darstellt.
Um diese Gödelisierung besser beschreiben zu können, werden hier kurz zwei Ergänzungen zu der Definition von BF_n vorgestellt:

Definition 3.2. *Die Anzahl $BF_{<n}$ der BF-Programme der Länge kleiner n und die Anzahl $BF_{\leq n}$ der BF-Programme der Länge kleiner-gleich n seien wie folgt definiert:*

- $BF_{<n} := \sum_{i=0}^{n-1} BF_n$
- $BF_{\leq n} := \sum_{i=0}^{n} BF_n$

Insbesondere gilt $BF_{<0} = 0$ und $BF_{\leq 0} = 1$.

Es soll auch wie schon beim naiven Ansatz gelten: je länger ein Programm, desto größer die Zahl, die ihm zugeordnet wird. Damit sind die Programme in $\mathbb{N}_0$ der Länge nach sortiert und es folgt sofort, dass ein Programm der Länge n einer Zahl im Intervall $[BF_{<n}, BF_{\leq n} - 1]$ zugeordnet wird. Diese Intervalle haben erwartungsgemäß die Größe BF_n, wie man leicht nachrechnen kann.

3.3 Konvertierung

Bislang wurden die Gödelisierungen φ und $\tilde{\varphi}$ nur formal definiert. Hier sollen nun Algorithmen vorgestellt werden, die diese Gödelisierungen in die Praxis umsetzen, d.h. zwischen Quelltext und zugehöriger Gödelnummer hin- und herkonvertieren.

Ein Algorithmus für den naiven Ansatz wurde bereits in Abschnitt 3.1 skizziert. Im Folgenden wird daher nur der praktische Ansatz aus Abschnitt 3.2 betrachtet.

Die Umsetzung des praktischen Ansatzes basiert auf zwei Algorithmen: *BFtoID* und *IDtoBF*. Der Algorithmus *BFtoID* wandelt ein BF-Programm der Länge n in eine Zahl aus dem Intervall $[0, BF_n - 1]$ um. Der Algorithmus *IDtoBF* kann diese Zahlen wieder zurück in das entsprechende BF-Programm umrechnen. Diese Zahlen liegen allerdings noch nicht im geforderten Intervall $[BF_{<n}, BF_{\leq n} - 1]$, vielmehr geben diese nur das Offset innerhalb des Intervals an. Die Gödelnummer $g(p)$ des Programmes p errechnet sich daher wie folgt:

$$g(p) = BFtoID(p,0,n) + BF_{<n} = BFtoID(p,0,n) + \sum_{i=0}^{n-1} BF_n$$

Um die Umkehrung $g^{-1}(x)$ der Gödelisierung mit Hilfe des Algorithmus *IDtoBF* zu berechnen ist es nötig, die Länge des Programms zu errechnen, welches zur Zahl x gehört. Da die Intervalle bekannt sind, in denen Programme bestimmter Länge liegen, kann man die Länge n leicht berechnen, indem man das kleinste n sucht, welches die Bedingung $x < BF_{\leq n} = \sum_{i=0}^{n} BF_n$ erfüllt. Das n wird einfach durch ausprobieren der $BF_{\leq n}$ gefunden, erst $n = 0$, dann $n = 1$, $n = 2, \ldots$. Danach kann $g^{-1}(x) = p$ berechnet werden, indem $IDtoBF(x - BF_{<n}, p, 0, n)$ aufgerufen wird. Der Paramter p wird von dem Algorithmus *IDtoBF* verändert und enthält nach dem Aufruf das dekodierte BF-Programm.

3.3.1 Programm $\mapsto$ Zahl

Der Algorithmus *BFtoID* lässt sich über die beiden Parameter s und e auf Ausschnitte von Programmen anwenden. Für Programmausschnitte der Länge $n = e - s$ liefert er immer eine Zahl im Intervall $[0, BF_n - 1]$.
Der Algorithmus ist rekursiv. Die Rekursion wird abgebrochen, sobald ein Programmausschnitt der Länge $n \leq 0$ behandelt werden soll. Dies wird in Zeile 2 überprüft. Da es nur ein Programm der Länge 0 gibt, wird diesem in Zeile 19 die Zahl 0 zugeordnet. Hat der Programmabschnitt allerdings eine Länge größer 0, so wird anhand der Rekursionsgleichung für $\mathcal{BF}_n$ und BF_n aus Lemma 2.8 gearbeitet. Die Menge $\mathcal{BF}_n$ wird in 7 disjunkte Teilmengen unterteilt: für jede Möglichkeit des ersten Symbols eine. Analog dazu wird das Intervall $[0, BF_n - 1]$ in 7 disjunkte Teilintervalle aufgeteilt. Die Programme, die einen der 6 normalen Befehle als erstes Symbol enthalten, werden in Zeile 16 behandelt. Dort wird zuerst die untere Grenze eines der 6 Intervalle berechnet, nämlich $p[s] \cdot BF_n$. Dazu wird das Offset innerhalb des Intervalls addiert: $BFtoID(p, s+1, e)$. Die Programmausschnitte, deren erstes Symbol eine öffnende Klammer ist, finden im 7-ten Intervall Platz. Dieses muss allerdings nochmal in kleinere disjunkte Teilintervalle zerlegt werden: für jede der $n-1$ in Zeile 4 möglichen Positionen der schließenden Klammer eines. Die $n-1$ Teilintervalle sind leider nicht gleich gross. Daher berechnen die Zeilen 5 bis 10 in einer Schleife gemäß der Rekursionsgleichung die untere Grenze eines der $n-1$ Teilintervalle. In Zeile 14 wird nun wieder das Offset innerhalb des Teilintervalls addiert. Allerdings setzt sich dieses diesmal aus 2 Werten zusammen: der Nummer *idl* für das Teilprogramm links der Position k und der Nummer *idr* für das Teilprogramm rechts der Position k. Der Wert $idl \cdot cr + idr$ ist so zusammengesetzt, dass er leicht via Division mit Rest durch cr in seine Bestandteile *idl* und *idr* zerlegt werden kann. Darüber hinaus übersteigt der Wert die Größe $BF_{k-s-1} \cdot BF_{e-k-1}$ des Intervalls nicht, so dass keine Überschneidungen mit den anderen Teilintervallen auf-

Eingabe : BF-Programm p als Array gefüllt mit Werten von 0 bis 7,
wobei 6 = öffnende Klammer und 7 = schließende Klammer
Eingabe : Position s des ersten Symbols (inklusive)
Eingabe : Position e des letzten Symbols (exklusive)
Ausgabe : Nummer id

```
1  n := e − s
2  if n > 0 then
3      if p[s] = 6 then
4          k := Index in p von passender schließender Klammer
5          id := 6 · BF_{n−1}
6          for i = (s+1)...(k−1) do
7              cl := BF_{i−s−1}
8              cr := BF_{e−i−1}
9              id := id + cl · cr
10         end
11         cr := BF_{e−k−1}
12         idl := BFtoID(p, s+1, k)
13         idr := BFtoID(p, k+1, e)
14         id := id + idl · cr + idr
15     else
16         id := p[s] · BF_{n−1} + BFtoID(p, s+1, e)
17     end
18 else
19     id := 0
20 end
```

Algorithmus 1 : $BFtoID(p,s,e)$

treten können.

3.3.2 Zahl ↦ Programm

Dieser Algorithmus lässt sich ebenfalls durch die Paramter s und e auf Ausschnitte des Arrays p eingrenzen. Er füllt den Ausschnitt des Arrays dann mit dem zu der Zahl id gehörenden BF-Programms. Sobald ein Auschnitt der Länge $n \leq 0$ behandelt werden soll, wird die Rekursion abgebrochen. Dies wird in Zeile 2 sichergestellt. In Zeile 3 wird errechnet, welcher der 7 Teilintervalle des Intervalls $[0, BF_n - 1]$ vom Algorithmus *BFtoID* gewählt wurde. Da die ersten 6 dieser Teileintervalle gleichgroß gewählt wurden, kann die Berechnung durch eine simple Division erfolgen. Lag die Zahl in einem der ersten 6 Teilintervalle, so wird in Zeile 21 das erste Symbol des Programmausschnitts geschrieben und der Rest des Programms durch einen rekursiven Aufruf in Zeile 22 dekodiert.
Liegt die Zahl allerdings im 7-ten Intervall, dann war das erste Symbol eine öffnende Klammer und es wird zunächst in den Zeilen 5 bis 12 ermittelt, in welchem der $n-1$ Teilintervalle die Zahl liegt. Nach Zeile 5 gilt $id \geq 0$. Dadurch wird die Schleife mindestens einmal durchlaufen, so dass die Variablen cl und cr in jedem Fall initialisiert werden. In Zeile 13 muss allerdings kompensiert werden, dass id negativ ist, wenn die Schleife verlassen wird. Das durch die Schleife ermittelte k entspricht der Position der schließenden Klammer wie sie im Algorithmus *BFtoID* ermittelt wurde. In den Zeilen 14 und 15 werden nun die Symbole für die Klammern an die entsprechenden Stellen im Programm geschrieben. Der Wert cr, der in Zeile 10 berechnet wurde, ermöglicht es nun, dass in den Zeilen 16 und 17 die beiden Werte idl und idr wieder berechnet werden können. Mit diesen Werten werden nun

Eingabe : Nummer id des zu dekodierenden Programms
Eingabe : Array p
Eingabe : Position s des ersten Symbols (inklusive)
Eingabe : Position e des letzten Symbols (exklusive)
Ausgabe : Array p gefüllt mit Werten von 0 bis 7,
wobei 6 = öffnende Klammer und 7 = schließende Klammer

```
1  n := e − s
2  if n > 0 then
3      t := id div BF_{n−1}
4      if t ≥ 6 then
5          id := id − 6 · BF_{n−1}
6          k := s
7          while id ≥ 0 do
8              k := k + 1
9              cl := BF_{k−s−1}
10             cr := BF_{e−k−1}
11             id := id − cl · cr
12         end
13         id := id + cl · cr
14         p[s] := 6
15         p[k] := 7
16         idl := id div cr
17         idr := id mod cr
18         IDtoBF(idl, p, s + 1, k)
19         IDtoBF(idr, p, k + 1, e)
20     else
21         p[s] := t
22         IDtoBF(id − t · BF_{n−1}, p, s + 1, e)
23     end
24 end
```

Algorithmus 2 : $IDtoBF(id, p, s, e)$

rekursiv die beiden Teile links und rechts der schließenden Klammer dekodiert.

3.4 Effizienz

Es ist wünschenswert, dass die Gödelnummern der beiden Gödelisierungen effizient — d.h. in polynomieller Zeit — berechenet werden können. Daher widmet sich dieser Abschnitt der Laufzeit- und Platzbedarfsanalyse.

Gödelnummern der naiven Gödelisierung können mit dem in Abschnitt 3.1 geschilderten Algorithmus in Laufzeit $O(n)$ berechnet und dekodiert werden. Es wird ebenfalls nur $O(n)$-viel Platz benötigt.

Die Algorithmen für die praktische Gödelisierung wurden bereits in Java implementiert. Da die Zahlen recht gross werden und 32-Bit bei weitem übersteigen, muss hier mit ganzen Zahlen beliebiger Länge gerechnet werden. Dafür gibt es in Java und in anderen Sprachen spezielle Pakete, allerdings liegen dann die Laufzeiten für Additionen und Multiplikationen nicht mehr bei $O(1)$, was man für 32-Bit Zahlen annehmen würde. Die Addition zweier n-Bit Zahlen benötigt Laufzeit $O(n)$ und die Multiplikation $O\left(n^2\right)$. Es gibt auch schnellere Algorithmen für Multiplikationen, die diese in Laufzeit $O\left(n^{\log 3}\right)$ [5, THEOREM 8.4] oder

in $O(n \log n \log\log n)$ [5, THEOREM 8.24] berechnen. Allerdings würden diese Laufzeiten keinen großen Einfluss auf das Endergebnisses haben, so dass weiterhin von $O(n^2)$ ausgegangen wird.
Ist bei einer Multiplikation eine der beiden multiplizierten Zahlen von konstanter Länge, so lohnt es sich, die Laufzeit nocheinmal genauer zu betrachten. Die Multiplikation einer n-Bit und einer m-Bit Zahl kostet nämlich nur $O(nm)$-viel Zeit. Ist m konstant, so geht also die Multiplikation in Laufzeit $O(n)$ vonstatten.
Der Platzbedarf von Addition, Multiplikation und Integer-Division ist $O(n)$. Die Laufzeit- und Platzbedarfsklassen lassen sich alle mit den aus der Schule bekannten Methoden erreichen. Subtraktion und Integer-Division mit Rest benötigen identische Laufzeit und Platzbedarf wie Addition und Multiplikation.

Die beiden Algorithmen *BFtoID* und *IDtoBF* sind einander sehr ähnlich. Daher wird hier darauf verzichtet, beide Algorithmen getrennt zu betrachten. Es wird nur die Laufzeit des Algorithmus *BFtoID* betrachtet, welche auf den Algorithmus *IDtoBF* übertragbar ist. Eine Besonderheit hat *IDtoBF* allerdings: in Zeile 3 wird eine Integer-Division durchgeführt. Da von dem Ergebnis nur die Fälle 0 bis 5 und ≥ 6 relevant sind, kann diese Division auch durch maximal 6-maliges Subtrahieren ersetzt werden. Dies kann in derselben Laufzeit wie die entsprechenden Multiplikationen in *BFtoID* berechnet werden.

Um nun die Laufzeit des Algorithmus *BFtoID* für ein Programm der Länger n abzuschätzen, muss man sich klar machen, dass der Algorithmus während der gesamtem Rekursion maximal n mal durchlaufen wird. Die Startposition s zeigt während der Rekursion nämlich maximal einmal auf jedes Symbol des Programms. Um nun die Laufzeit eines Durchgangs des Algorithmus abzuschätzen, nehmen wir außerdem für die Länge des Programmausschnitts den worst-case an. Das in Zeile 1 des Algorithmus berechnete n soll also immer der Länge des kompletten Programms entsprechen. Mit diesem n als Maß haben alle Variablen mit einem einbuchstabigen Namen eine Länge von $O(\log(n))$ Bit. Alle anderen Variablen haben eine Länge von $O(n)$ Bit, da deren Werte nicht größer als $BF_n \leq 8^n$ werden können. Weiterhin soll davon ausgegangen werden, dass die BF_i für $0 \leq i \leq n$ bereits vorberechnet sind. Ein Zugriff auf ein solches BF_i benötigt daher Laufzeit $O(n)$. Warum dem so ist, wird später erklärt. Weitere BF_i mit $i > n$ werden vom Algorithmus nicht benötigt.
Die Laufzeit eines Durchgangs setzt sich wie folgt zusammen:

Zeilen	*Laufzeit*	*Ursache*
1,2,3	$O(\log(n))$	Addition, Vergleiche
4	$O(n)$	maximal wird der ganze Programmausschnitt durchsucht
5	$O(n)$	Multiplikation mit 6
6	$O(\log(n))$	Addition und Vergleich bei jedem Schleifendurchlauf
7,8	$O(n)$	Kopieren des Werte nach *cl* und *cr*
9	$O(n^2)$	Multiplikation
11	$O(n)$	Kopieren des Wertes nach *cr*
12,13	$O(n)$	Kopieren der Rückgabewerte nach *idl* und *idr*
14	$O(n^2)$	Multiplikation
16	$O(n)$	Multiplikation mit 6, Addition
19	$O(1)$	Zuweisung

Die Laufzeiten für die Zeilen 6 bis 9 müssen allerdings noch mit n multipliziert werden, da die Schleife im schlimmsten Fall n-mal durchlaufen wird. Für den Fall das ein Programm Klammern enthält, benötigt ein Durchlauf des Algorithmus also maximal Laufzeit $O(n^3)$, da Zeile 9 in der Schleife Laufzeit $O(n^2)$ benötigt. Für den Fall das ein Programm keine Klammern enthält, benötigt der Algorithmus nur Laufzeit $O(n)$, da die Zeilen 4 bis 14 nie ausgeführt werden.

Insgesamt beträgt die Laufzeit für ein Programm der Länge n also $O(n^4)$, für ein Programm ohne Klammern sogar lediglich $O(n^2)$.

Der Platzbedarf der der beiden Algorithmen beläuft sich auf $O(n^2)$, da sich die Algorithmen maximal n-mal rekursiv aufrufen und pro Aufruf nur konstant viele Variablen mit $O(n)$ Bit benötigt werden.

Zu dem Platzbedarf und der Laufzeit kommen aber noch die in Abschnitt 3.3 angesprochenen zusätzlichen Arbeitsschritte. So muss auf das Ergebnis des Algorithmus *BFtoID* noch $BF_{<n}$ addiert werden. Bislang wurde davon ausgegangen, dass die BF_i bereits vorberechnet sind. Der Algorithmus *BFtoID* kann davon ausgehen, dass die BF_i vorberechnet sind, denn diese werden bereits benötigt, um das $BF_{<n}$ zu berechnen. Speichert man die BF_i in einem Array, und benutzt man die dritte Formel aus Lemma 2.8 um die BF_i für $i = 2 \dots n$ auszurechnen, so liegt die Laufzeit bei $O(n^2 \log n)$. Das berechnen von $BF_{<n}$ durch simples Aufsummieren der BF_i kostet nun Laufzeit $O(n^2)$. Der Platzbedarf für das Array beläuft sich ebenfalls auf $O(n^2)$. Diese Überlegungen sind ebenfalls wieder auf den Algorithmus *IDtoBF* übertragbar.

Zusammengefasst ergibt sich folgenden Theorem:

Theorem 3.3. *Die Algorithmen für die praktische Gödelisierung φ benötigen Laufzeit $O(n^2 \log n)$ für Programme ohne Klammern, $O(n^4)$ für Programme mit Klammern und $O(n^2)$ an Platz um ein Programm in eine Gödelnummer umzurechnen oder eine Gödelnummer zu dekodieren.*
Die Algorithmen für die naive Gödelisierung benötigen Laufzeit und Platzbedarf $O(n)$.

4 Approximierbarkeit und Dichtheit

Der gängige Begriff eines Approximationsalgorithmus kann nicht auf das Halteproblem übertragen werden. Normale Approximationsalgorithmen, wie sie z.B. für das Traveling Salesman Problem existieren, liefern Ergebisse, bei denen eine gewisse prozentuale Abweichung gegenüber dem Optimum in Kauf genommen wird.
Bei dem Halteproblem handelt es sich allerdings um ein Entscheidungsproblem. Die Ausgabe muss entweder 1 lauten, falls die Eingabe akzeptiert wurde, oder 0 falls nicht. Der Begriff Approximation muss hier neu definiert werden: eine Approximation des Halteproblems darf bei einem gewissen Anteil der Eingaben ein falsches Ergebnis liefern.
Diese Art der Aproximation des Halteproblems hängt unmittelbar mit der verwendeten Gödelisierung zusammen. Das Halteproblem ist so definiert, dass die Eingabe die Gödelnummer eines Programms ist. Das Speichern eines Programmquelltextes in einer Datei ist ein Beispiel einer Gödelisierung, denn die Datei ist eine Folge von Bytes, die wiederum als Zahl interpretiert werden können. Allerdings können in einer Datei viele syntaktisch ungültige Programme gespeichert werden. Es ist leicht, das Halteproblem für syntaktisch nich korrekte Programme zu entscheiden: je nach Vereinbarung werden diese akzeptiert oder abgelehnt. Über die Syntax eines Programms zu entscheiden ist ein einfaches Problem. Solche Programme sind meist Teil des Compilers oder Interpreters der jeweiligen Programmiersprache. Das Halteproblem kann für andere Gruppen von Programmen ebenfalls sehr leicht entschieden werden. Ein Beispiel dafür sind Programme ohne Schleifen.
Kommen nun in einer Gödelisierung die Programme, für die das Halteproblem entschieden werden kann, besonders häufig vor, so entsteht der Eindruck, dass das Halteproblem gut approximiert werden kann.
Es sind auch Gödelisierungen denkbar, bei denen der Anteil dieser Programme gegen 1 geht, wenn immer größere Intervalle betrachtet werden. Eine solche Gödelsierung lässt sich konstruieren, indem man den Programmen nur Zweierpotenzen zuordnet. Allen ande-

ren Zahlen wird ein syntaktisch inkorrektes Programm zugeordnet. Solche Gödelisierungen kann man auch als „dünn“ bezeichnen.

Hier wird nun ein Kriterium vorgestellt, welches eine Aussage über die Verteilung von Programmen innerhalb einer Gödelisierung macht.

Definition 4.1. [1, DEFINITION 4] *Die Wiederholrate der partiellen Funktionen mit äquivalentem Definitionsbereich $RD_{\varphi,i}$ ist wie folgt definiert:*

$$RD_{\varphi,i}(n) = \text{Prob}[\text{dom}(\varphi_x) = \text{dom}(\varphi_i) | \varphi_x \textit{ ist Programm der Länge } n]$$

Eine Gödelisierung φ ist dicht genau dann wenn gilt:

$$\forall i \exists c > 0 \exists n_0 \forall n > n_0 : RD_{\varphi,i}(n) > c$$

Ist eine Gödelisierung also dicht, so wiederholt sich ein beliebiges Programm φ_i bei den Programmen großer Länge mit einer Rate, die den Schwellenwert c nicht unterschreiten darf. Ist eine Gödelisierung dicht, so kann z.B. der obene beschriebene Fall einer „dünnen“ Gödelisierung nicht auftreten.
Darüber hinaus kann eine Aussage über die Approximierbarkeit des Halteproblems getroffen werden:

Theorem 4.2. [1, THEOREM 4] *Schindelhauer und Jakoby zeigen, dass für jede dichte Gödelisierung φ gilt:*

$$\forall M \exists \alpha > 0 \exists n_0 \forall n > n_0 : \text{Prob}[M(x) \neq H_\varphi(x) | \varphi_x \textit{ ist Programm der Länge } n] \geq \alpha$$

Das heisst, dass jedes Programm M welches das Halteproblem zu approximieren versucht, mindestens einen konstanten Anteil an Fehlern macht.

Nun stellt sich die Frage, ob die Gödelisierungen $\tilde{\varphi}$ und φ dicht sind. Dazu brauchen wir zunächst ein paar weitere Abschätzungen für BF_n.

Lemma 4.3. *Für alle $n \geq 0$ gilt:*

$$BF_{n+1} \leq 8 \cdot BF_n \cdot \frac{n + \frac{9}{4}}{n+3} \leq 8 \cdot BF_n$$

Daraus folgt direkt:

$$BF_{n+1} \leq 8^n \cdot \prod_{i=0}^{n} \frac{i + \frac{9}{4}}{i+3}$$

Beweis. Nach Lemma 2.8 gilt:

$$\begin{aligned} BF_{n+1} &= \frac{(12n+18) \cdot BF_n - 32n \cdot BF_{n-1}}{n+3} \\ &\leq \frac{12n+18}{n+3} \cdot BF_n \end{aligned}$$

Der Quotient $\frac{12n+18}{n+3}$ steigt streng monoton, und konvergiert für $n \to \infty$ gegen den Wert $c_0 = 12$. Diese Abschätzung $BF_{n+1} \leq c_0 \cdot BF_n$ lässt sich nun benutzen, um eine bessere Abschätzung zu finden:

$$\begin{aligned} BF_{n+1} &= \frac{(12n+18) \cdot BF_n - 32n \cdot BF_{n-1}}{n+3} \\ &\leq \frac{(12n+18) \cdot BF_n - \frac{32n}{c_0} BF_n}{n+3} \\ &= \frac{(12 - \frac{32}{c_0})n + 18}{n+3} \cdot BF_n \end{aligned}$$

Der Faktor vor dem BF_n steigt wieder streng monoton und konvergiert für $n \to \infty$ gegen den Wert $c_1 = 12 - \frac{32}{c_0} \approx 9,3$. Nun kann man diese Abschätzung wieder einsetzen und erhält ein neue Abschätzung c_2. Durch mehrmaliges Wiederholen erhält man die Folge $(c_i)_{i \in \mathbb{N}_0}$ mit $c_{i+1} = 12 - \frac{32}{c_i}$. Diese Folge fällt streng monoton und konvergiert für $i \to \infty$ gegen 8. Also gilt wie behauptet $BF_{n+1} \leq 8 \cdot BF_n$. Setzt man dies erneut in die Rekursionsformel ein, so erhält man:

$$\begin{aligned} BF_{n+1} &= \frac{(12n+18) \cdot BF_n - 32n \cdot BF_{n-1}}{n+3} \\ &\leq \frac{(12n+18) \cdot BF_n - 4n \cdot BF_n}{n+3} \\ &= \frac{8n+18}{n+3} \cdot BF_n \\ &= 8 \cdot \frac{n+\frac{9}{4}}{n+3} \cdot BF_n \end{aligned}$$

□

Satz 4.4. *Die hier vorgestellte naive Gödelisierung $\tilde{\varphi}$ ist **nicht** dicht.*

Beweis. Um die Dichtheit der Gödelisierung $\tilde{\varphi}$ zu wiederlegen, muss eine obere Schranke für die Wiederholrate gefunden werden. Sei nun $\tilde{\varphi}_i$ ein syntaktisch korrektes Programm mit $\mathrm{dom}(\tilde{\varphi}_i) \neq \varnothing$. Nun wird angenommen, dass alle syntaktisch korrekten Programme der Länge n den selben Definitionsbereich wie das Programm $\tilde{\varphi}_i$ aufweisen. Dies ist zwar nicht korrekt, stellt aber sicherlich eine obere Schranke dar. Die syntaktisch inkorrekten Programme kommen nicht in Betracht, da deren Definitionsbereich der leeren Menge entspricht. Es ergibt sich daher folgendes:

$$\begin{aligned} RD_{\tilde{\varphi},i}(n) &\leq \frac{BF_n}{8^n} \\ &\leq \frac{8^n \cdot \prod_{i=0}^{n-1} \frac{i+\frac{9}{4}}{i+3}}{8^n} \\ &= \prod_{i=0}^{n-1} \frac{i+\frac{9}{4}}{i+3} \end{aligned}$$

Dieser Ausdruck geht gegen 0. Die Gleichung aus Definition 4.1 kann also nicht erfüllt werden, da jedes noch so kleine $c > 0$ für große n unterschritten wird.

□

Satz 4.5. *Die hier vorgestellte praktische Gödelisierung φ ist dicht.*

Beweis. Es muss geziegt werden, das die Gleichung aus Definition 4.1 gilt. Um die Wiederholrate für ein gegebenes Programm φ_i abzuschätzen, müssen andere Programme gefunden werden, die denselben Definitionsbereich aufweisen. Das können Programme sein, die komplett andere Algorithmen darstellen, als das Programm φ_i. Diese sind daher sehr schwer zu finden. Einfacher ist es, sich auf Programme zu beschränken, die zu φ_i äquivalent sind. Diese sind einfacher zu finden, z.B. indem in das Programm Befehle eingefügt werden, die entweder nicht ausgeführt werden oder keine Auswirkungen auf das Ergebnis haben.
Es bietet es sich an, vor das Programm eine Schleife zu setzen, die ein beliebiges BF-Programm beinhalten darf. Der Inhalt der Schleife wird nicht ausgeführt sondern übersprungen, da alle Zellen des Arbeitsbandes beim Start eines BF-Programms mit 0 gefüllt sind.
Sei nun m die Länge des Programms φ_i und sei $n \geq m+2$. Dann stehen BF_{n-m-2}-viele

mögliche BF-Programme zur Verfügung, die in Klammern vor das Programm gesetzt werden können. Es gilt also:

$$\begin{aligned} RD_{\varphi,i}(n) &\geq \frac{BF_{n-m-2}}{BF_n} \\ &\geq \frac{BF_{n-m-2}}{8^{m+2} \cdot BF_{n-m-2}} \\ &= \left(\frac{1}{8}\right)^{m+2} \end{aligned}$$

Für jedes Programm φ_i existieren also wie gefordert c und n_0.

□

5 Schlussfolgerungen

Die Gödelisierung φ ist gemäß Satz 4.5 dicht. Laut Schindelhauer und Jakoby kann hier das Halteproblem also nicht besser als bis auf einen konstanten Faktor approximiert werden. Es stellen sich folgende weiterführende Fragen:

- Gibt es einen Algorithmus, der das Halteproblem für φ tatsächlich bis auf einen konstanten Faktor approximiert?
- Falls ja, welche konstanten Approxmationsfaktoren sind erreichbar?

Die Gödelisierung $\tilde{\varphi}$ ist nicht dicht und tatsächlich folgt aus dem Beweis von Satz 4.4, dass hier bereits das Identifizieren syntaktisch inkorrekter BF-Programme eine Fehlerrate von $O\left(\frac{1}{\sqrt{n}}\right)$ für die Approximation des Halteproblems liefert. Diese ist für die Praxis natürlich wenig hilfreich.

Literatur

[1] C. SCHINDELHAUER, A. JAKOBY: „The Non-Recursive Power of Erroneous Computation“, Seiten 394–406 in *Proc. 19th Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS'99), Springer LNCS Band **1738**.

[2] J. E. HOPCROFT, R. MOTWANI, J. D. ULLMAN: „*Introduction to Automata Theory, Languages, and Computation*“, zweite Auflage, Addison Wesley (2001).

[3] P. ODIFREDDI: „*Classical Recursion Theory*“, Elsevier (1989).

[4] D. F. BAILEY: „Counting Arrangements of 1's and −1's“ Seiten 128–131 in *Mathematics Magazine* **69** (1996).

[5] J. VON ZUR GATHEN, J. GERHARD: „*Modern Computer Algebra*“, zweite Auflage, Cambridge Press (2003).

[6] B. RAITER: `http://www.muppetlabs.com/~breadbox/bf/`

Danksagung

Bei Dr. Martin Ziegler möchte ich mich besonders für die ausgezeichnete Betreuung und die aufschlussreichen Diskussion bedanken. Meiner Mutter und meinen Freunden danke ich für die Unterstützung beim Korrekturlesen.

Versicherung

Ich versichere hiermit, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Diese Arbeit wurde in gleicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 1. Dezember 2004,

Sven Köhler