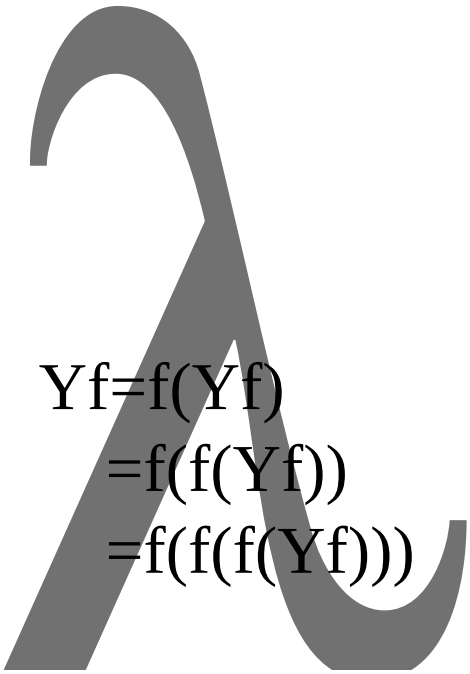
Der Lambda-Kalkül



Syntax des λ-Kalküls

(Tanja Nemetzade)

Der λ-Kalkül ist eine Anfang des 20. Jahrhunderts entstandene mathematische Sprache, die uns erlaubt alles Berechenbare nur mit Funktionen darzustellen, den sogenannten Kombinatoren, also Funktionen, die wiederum Funktionen als Werte haben.

Als Beispiel betrachten wir die Hintereinanderausführung f von zwei Funktionen g und h

$$f(x) = g(h(x)),$$

die Komposition genannt wird. Dabei wird jeder Stelle x genau der Wert der Funktion g an der Stelle $h(x)$ zugeordnet, wobei die zusammengesetzte Funktion f von g und h abhängig ist. In der Pfeilnotation, die Zuordnungen besonders deutlich macht, sieht die obige Funktionskomposition demnach foldendermaßen aus:

$$g \mapsto (h \mapsto (x \mapsto g(h(x))))$$

Auffällig sind nun hier die vielen Klammern und Pfeile, die umständlich sind und die Funktion unübersichtlich erscheinen lassen. Da Mathematiker zusätzlich bekanntermaßen faul sind und nie zu viel schreiben wollen, wird im λ-Kalkül eine neue Schreibweise verwendet, die keine Pfeile und kaum Klammern benötigt, um Kombinatoren darzustellen.

$$\lambda ghx.g(hx)$$

wäre die Funktionskomposition in λ-Schreibweise, was allerdings eine Kurzschreibweise für die offizielle Notation

$$\lambda g.\lambda h.\lambda x.g(hx)$$

ist. Zur Klammerkonvention im λ-Kalkül ist noch zu sagen, dass, wie bei $g(hx)$, linksgesetzte Klammern per Konvention weggelassen werden dürfen; abc ist also die Kurzschreibweise für $(ab)c$.

Was nun die Rechenregeln betrifft, kennt der λ-Kalkül kennt nur eine, nämlich die sogenannte β-Regel, die als „Einsetz-Regel“ verstanden werden kann. Um beispielsweise in die oben genannte Funktion $B = \lambda ghx.g(hx)$ für alle Vorkommen von g einen weiteren λ-Term $(\lambda u.u)$ einzusetzen, schreiben wir einfach:

$$(\lambda ghx.g(hx))(\lambda u.u)$$

was sich zu

$$(\lambda hx.(\lambda u.u)(hx))$$

reduzieren lässt.

Allgemein wird also ein Kombinator $(\lambda x.t)$ auf einen Term r angewendet, indem im Term t alle freien Vorkommen von x durch r ersetzt werden. Dafür schreiben wir

$$(\lambda x.t)r = t[x := r].$$

Im obigen Beispiel entspricht demnach x gerade g, t gerade $g(hx)$ und r gerade $(\lambda u.u)$.

Nun gibt es im λ-Kalkül einige Kombinatoren, die immer wieder benötigt werden, so dass sich einheitliche Bezeichnungen durchgesetzt haben, wie zum Beispiel

- die Identität $I = \lambda x.x$
- der K-Kombinator $K = \lambda xy.x$
- der S-Kombinator $S = \lambda xyz.xz(yz)$
- der Paarkombinator $P = \lambda xyz.zxy$
- der Bindekombinator $B = \lambda xyz.x(yz)$

In den folgenden Kapiteln werden wir nun einige besondere Eigenschaften der Kombinatoren und auch praktische Anwendungen des λ-Kalküls kennen lernen.

Die Kombinatoren S und K

(Andrea Betz)

Im λ-Kalkül ist es möglich, jeden beliebigen Kombinator durch die Kombinatoren S und K auszudrücken.

Das allgemein gültige Verfahren lässt sich bei jeder Rechnung dieser Art verwenden: Den Term duch S und K so umformen, dass die letzte Variable isoliert am Ende steht, also in der Form $\lambda abc...n.Qn$. Nach der Regel $\lambda x.fx = f$ wird die sogenannte η-Reduktion ausgeführt. Daraus erhalten wir $\lambda abc...m.Q$. Die Kunst besteht darin, die zu reduzierende Variable durch Umformung mit S und K ans Ende und nur ans Ende des Terms zu bringen.

Dies soll anhand eines Beipiels veranschaulicht werden. Gegeben sei

$$\lambda abc.ca(bc)\,.$$

Wir versuchen zuerst, c ans Ende des Terms zu bringen.

$$\lambda abc.Ic(Kac)(bc)$$

Hier erkennen wir die für die Benutzung von S notwendige Form $Sxyz = xz(yz)$, wir können also umformen.

$$\lambda abc.SI(Ka)c(bc)$$

Wir können erneut S anwenden:

$$\lambda abc.S(SI(Ka))bc$$

Wir erkennnen, dass man b und c gleichzeitig η-reduzieren kann, da die Variablen in der richtigen Reihenfolge nur noch am Ende stehen.

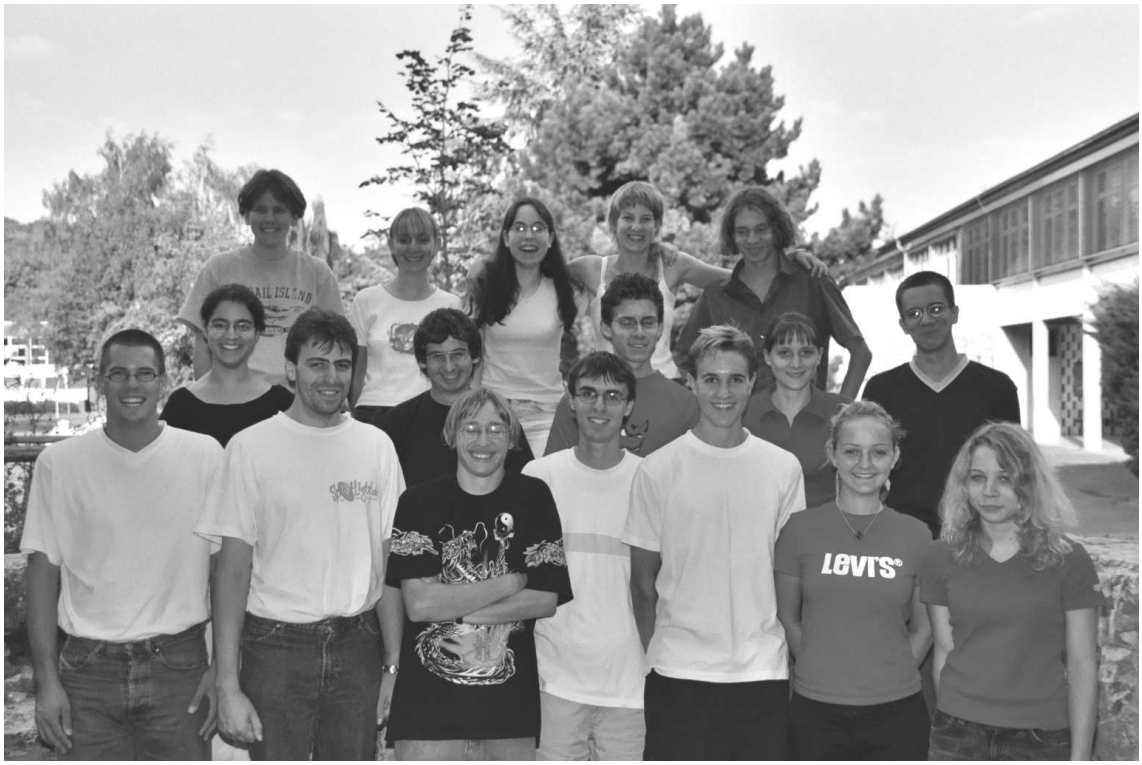
$$\lambda a.S(SI(Ka))$$

Durch die Anwendung von K bereiten wir uns auf eine weitere Umformung mit S vor.

$$\lambda a.S(K(SI)a(Ka))$$

Umformung mit S liefert

$$\lambda a.S(S(K(SI))Ka)$$



Kurs: Der Lambda-Kalkül

Erneute Anwendung von K, um später mit S umzuformen:

$$\lambda a.KSa(S(K(SI))Ka)$$

Umformung mit S liefert nun

$$\lambda a.S(KS)(S(K(SI))K)a$$

Wir können a η-reduzuieren. Außerdem ist bekannt, dass $I = SKK$, wir haben also folgendes Ergebnis:

$$\lambda abc.ca(bc) = S(KS)(S(K(S(SKK)))K)$$

Eine interessante Tatsache stellt sich bei der Benutzung des Kombinators $X = \lambda x.xKSK$ heraus: Durch die gewohnte Umformung mit Hilfe der β-Reduktion lassen sich S und K ausdrücken.

$$(XX)X = ... = K$$

$$X(XX) = ... = S$$

Es lassen sich also alle vollständig abgebundenen Terme im Lambda-Kalkül mit allein X und Klammern ausdrücken.

Church'sche Ziffern

(Juliane Claus)

Bisher haben wir gesehen, was der λ-Kalkül ist und dass alle λ-Terme sich aus den Kombinatoren S und K zusammensetzen lassen. Doch kann man in dem Kalkül auch rechnen beziehungsweise programmieren und dazu nötige Daten wie etwa Wahrheitswerte, `if-then-else`-Konstruktionen, Paare und natürliche Zahlen darstellen? Tatsächlich gibt es zahlreiche Möglichkeiten, diese Daten in λ-Ausdrücke umzusetzen. Einige elegante Varianten, die sich auch allgemein durchgesetzt haben, sollen nun beschrieben werden.

Wir beginnen mit den einfachsten Daten, den Wahrheitswerten. Dazu brauchen wir zwei verschiedene λ-Terme, die die Werte „wahr“ und „falsch“ darstellen. Es ist sinnvoll, „wahr“ durch den Kombinator $K = \lambda xy.x$ und „falsch“ durch den Kombinator $KI = \lambda xy.y$ auszudrücken, da der Kombinator K von zwei Argumenten genau das erste und KI das zweite ausgibt. Im Folgenden werden wir für „wahr“ anstelle von K einfach W und für „falsch“ F anstelle von KI schreiben, wenn wir sie konkret in der Rolle eines Wahrheitswertes meinen.

Mit Hilfe dieser Wahrheitswerte kann man nun beispielsweise die Funktion `and` definieren. Diese soll, wenn ihr die Argumente $a =$ „wahr“ und $b =$ „wahr“ übergeben werden, zu „wahr“ reduzieren und für alle anderen Kombinationen von Wahrheitswerten zu „falsch“. Eine solche Funktion könnte folgendermaßen aussehen:

$$\lambda ab.abF$$

Tatsächlich liefert die Funktion für $a = F$ und $b = W$

$$(\lambda ab.abF)FW = FWF = F$$

und für $a = W$ und $b = W$:

$$(\lambda ab.abF)WW = WWF = W$$

Gleichermaßen lässt sich die Funktion `not`, die jedem Wahrheitswert seine Negation zuordnet, als $\lambda a.aFW$ implementieren, oder die Funktion `nand`, die Negation von `and`, als $(\lambda a.aFW)\lambda ab.abF = \lambda ab.(abF)FW$. Auf diese Weise und mit Hilfe weiterer Kombinatoren, z.B. `or` können wir jetzt schon Boole'sche Algebra betreiben.

Vergleichsweise komplizierter darzustellen sind die natürlichen Zahlen und die auf ihnen möglichen Berechnungen wie Addition oder Multiplikation. Doch auch die lassen sich im λ-Kalkül als Kombinatoren ausdrücken, und zwar folgendermaßen.

Man kann sich die Zahl 1 als den Nachfolger der 0 vorstellen, die 2 als den Nachfolger der 1 und damit als den Nachfolger des Nachfolgers der 0, die 3 als Nachfolger der 2 beziehungsweise Nachfolger des Nachfolgers der 1 beziehungsweise Nachfolger des Nachfolgers des Nachfolgers der 0. Somit kann jede natürliche Zahl als n-ter Nachfolger der 0 definiert werden. Um das nun im λ-Kalkül umzusetzen, brauchen wir zur Darstellung einer natürlichen Zahl n einen Kombinator, der zwei Argumente, nämlich gerade die Basis- oder Nullfunktion z (für zero) und eine Funtion s (für successor), die uns sagt, wie ein Nachfolger zu bilden ist, nimmt und die Nachfolgerfunktion n-mal auf die „Null“ anwendet. So wäre der Kombinator, der die 0 ausdrückt $\underline{0} = \lambda sz.z$. Für die folgenden Zahlen ergibt sich

$$\begin{aligned} \underline{1} &= \lambda sz.sz \\ \underline{2} &= \lambda sz.s(sz) \\ \underline{3} &= \lambda sz.s(s(sz)) \quad \text{und so weiter.} \end{aligned}$$

Im Prinzip sind also die Zahlen nur Funktionen, die ihr erstes Argument n-mal auf das zweite anwenden. Diese Zahlen bezeichnet man als Church'sche Ziffern.

Obwohl diese Schreibweise etwas ungewohnt und kompliziert erscheint, lassen sich auf den natürlichen Zahlen nun relativ einfach Funktionen wie beispielsweise die Addition herleiten. Dazu gehen wir quasi rückwärts vor und betrachten zunächst die Zahl $\underline{5} = \lambda sz.s(s(s(s(sz))))$ als Summe der Zahlen $\underline{2}$ und $\underline{3}$. Bei näherem Betrachten dieses Ausdrucks stellen wir fest, dass wir ihn genauso gut als $\underline{5}sz = s(s(\underline{2}sz))$ schreiben können, die 5 also als dritten Nachfolger der 2 betrachten. Dies wiederum ist dasselbe wie $\lambda sz.\underline{3}s(\underline{2}sz)$, womit wir unsere beiden Summanden in den Ausdruck gebracht haben. Verallgemeinernd lässt sich aus dem Beispiel die Additionsfunktion für beliebige Zahlen n und m ableiten, die dann folgendermaßen aussieht:

$$A = \lambda nmsz.ns(msz)$$

An einem weiteren Beispiels wollen wir uns die Funktionsweise von `add` klar machen.

$$(\lambda nmsz.ns(msz))\underline{2}\,\underline{1} = \lambda sz.\underline{2}s(\underline{1}sz) = \lambda sz.\underline{2}s(sz) = \lambda sz.s(s(sz))) = \underline{3}$$

Ähnlich funktioniert die Multiplikation. Als Beispiel betrachten wir $2 \cdot 3 = 6$. Im λ-Kalkül ist die 6 gerade durch $\underline{6}sz = s(s(s(s(s(sz)))))$ gegeben. Dies kann man auch als 2-fache Ausführung der Funktion $\lambda sz.s(s(sz)) = \underline{3}$ auf z sehen, also: $\underline{6}sz = \underline{3}s(\underline{3}sz) = \underline{2}(\underline{3}s)z$. Allgemein ist der Kombinator für die Multiplikation folglich

$$M = \lambda mnsz.m(ns)z = \lambda mns.m(ns)$$

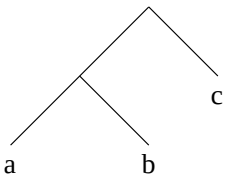


Abb. 1: Ein Baum

Diesen Kombinator kennen wir bereits unter dem üblicheren Namen B. Auf ähnliche Weise können wir nun auch alle übrigen Rechenoperationen herleiten. Als letztes Beispiel betrachten wir dazu noch das Potenzieren. Wieder beginnen wir mit einem konkreten Beispiel, nämlich $2^3 = 8$. In λ-Schreibweise ist das $\underline{8}sz = s(s(s(s(s(s(s(sz)))))))) = \underline{2}(\underline{2}(\underline{2}s))z = \underline{3}\,\underline{2}sz$. Allgemein bedeutet das für die Potenzierungsfunktion:

$$E = \lambda mnsz.nmsz = \lambda mn.nm$$

Dieser Kombinator wird gelegentlich auch mit D bezeichnet.

Schließlich brauchen wir für unsere natürlichen Zahlen noch einen Nulltester. Diese Funktion soll herausfinden, ob eine gegebene Church'sche Ziffer 0 ist, das heißt sie soll für den Fall, dass es sich um die 0 handelt, „wahr" ausgeben und für alle anderen Fälle „falsch". Vorher haben wir gesehen, dass der Kombinator für die Church'sche Ziffer 0 gerade $\lambda sz.z = KI = F$ ist. Da das n-fache Anwenden der konstant falschen Funktion auf den Wert „wahr" gerade zwischen 0 und den anderen Zahlen diskriminiert, lässt sich ein Nulltester so formulieren:

$$N = \lambda n.n(KF)W$$

Angewandt auf die Ziffer $\underline{3}$ liefert die Funktion:

$$N\underline{3} = \underline{3}(KF)W = KF(KF(KFW)) = F$$

Das gleiche gilt, wie man sich leicht vorstellen kann, für alle Zahlen größer als 0. Wird der Nulltester jedoch auf die $\underline{0}$ angewandt, so ergibt sich:

$$N\underline{0} = \underline{0}(KF)W = KI(KF)W = IW = W.$$

Zum Schluss ist noch hinzuzufügen, dass sich aus den Kombinatoren A (Addition), M (Multiplikation), E (Potenzfunktion) und der $\underline{0}$ jeder beliebige λ-Term darstellen lässt. Konkret heißt das, dass S und K definiert werden können als

$$K = M(M(E\underline{0})M)E$$

und

$$S = MME(M(M(MME))A)E.$$

Nachdem wir Wahrheitswerte und natürliche Zahlen jetzt bereits sinnvoll im λ-Kalkül definiert haben, brauchen wir zum Programmieren auch noch Möglichkeiten, Daten zu strukturieren und beispielsweise in Listen oder Bäumen zusammenzufassen. Zunächst wollen wir dazu einen Kombinator einführen, der zwei beliebige Werte a und b zu einem Paar $[a, b]$ zusammenfügt. Merkmal dieses Paares soll sein, dass man mit geeigneten Funktionen auf das linke oder rechte Element zugreifen kann. Der Kombinator, der dies ermöglicht, ist der bereits bekannte Standardkombinator $P = \lambda xyz.zxy$. Dieser fügt die zwei Werte x und y zu dem Paar $[x, y]$ zusammen und verlangt als drittes Argument eine Funktion, die auf das Paar angewendet werden soll. Wollen wir nun auf das linke Element dieses Paares zugreifen, so setzen wir K ein und gelangen damit zu $PxyK = Kxy = x$. Gleichermaßen können wir mit Hilfe von KI das rechte Element des Paares herausfinden: $Pxy(KI) = KIxy = Iy = y$. Daneben gibt es weitere Kombinatoren, die in irgendeiner Weise mit dem Paar arbeiten. Als letztes Beispiel dazu sei noch das Argument genannt, der die beiden Elemente vertauscht, also aus dem Paar $[a, b]$ das Paar $[b, a]$ macht: $\lambda xyz.zyx$.

Mit dieser Art, zwei beliebige Werte zu paaren, können wir nun auch ganze Bäume zusammensetzen. Dabei betrachten wir jede Verzweigung in dem Baum als Paar, welches wiederum aus Paaren zusammengesetzt sein kann.

Der Baum in Abbildung 1 lässt sich so als Paar $[a, [b, c]]$ schreiben. Gleichermaßen können natürlich an Stelle der Werte a, b und c weitere Paare stehen, so dass ein beliebig großer und beliebig kompliziert strukturierter Baum entwickelt werden kann. Eine spezielle Form dieser Bäume sind Kamm-Bäume (s. Abb. 2).

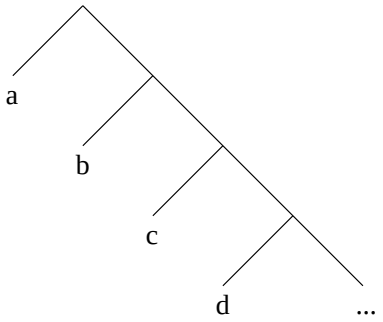


Abb. 2: Ein Kamm-artiger Baum

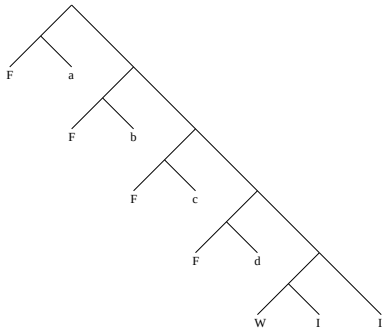


Abb. 3: Eine Liste

Sie haben die Form $[a,[b,[c,[d,...]]]]$ und entsprechen damit gerade Listen von Daten. Um jedoch besser in den Listen arbeiten zu können, brauchen wir zusätzlich zu jedem Element noch die Information, ob die Liste dort zu Ende ist oder noch fortgesetzt wird. Dies wird allgemein dadurch erreicht, dass an Stelle eines Wertes a ein Paar $[F,a]$ eingesetzt wird und das Ende der Liste durch $[W,I]$ gekennzeichnet ist, die Liste also die Form $[[F,a],[[F,b],[[F,c],[[F,d],[[W,I],I]]]]]$ annimmt, wobei anstelle von I hier ein beliebiger Kombinator auftauchen könnte.

Die Vorteile dieser zusätzlichen Markierung liegen darin, dass sich nun auf einfache Weise Funktionen definieren lassen, die mit der Liste arbeiten, beispielsweise feststellen, wie lang die Liste ist, bestimmte Elemente herausfinden oder neue Elemente hinzufügen. Um zu testen, ob die Liste leer ist, kann man einfach den Kombinator $\lambda\ell.\ell WW$ nutzen. Dieser wird für jede nicht leere Liste zu F reduziert und nur genau dann zu W, wenn die Liste leer ist, also die Form $[[W,I],I]$ hat. Weiterhin findet der Kombinator $\lambda\ell.\ell WF$ das erste Element einer Liste. Ein neues Element a, beziehungsweise ein neues Paar $[F,a]$ fügt der Kombinator $\texttt{cons} = \lambda a\ell.P(PFa)\ell = \lambda a\ell.P[F,a]\ell = \lambda a\ell.[[F,a],\ell]$ einer schon vorhandenen Liste hinzu.

Damit haben wir die grundlegendsten Arten der Implementierung von Daten und Datenstrukturen kennen gelernt, so dass wir im λ-Kalkül nun auch rechnen und programmieren können.

Konfluenz

(Michael Walter)

β-Gleichheit. Bei der Betrachtung des λ-Kalküls ergibt sich nicht zuletzt die Frage der Gleichheit zweier λ-Terme. Wir formalisieren daher den Gleichheitsbegriff, den wir bereits verwendet haben. Die β-Gleichheit $=_\beta$ ist die kleinste Kongruenzrelation mit folgender Eigenschaft:

$$(\lambda x.r)s =_\beta r[x := s]$$

Eine Kongruenzrelation ist eine Äquivalenzrelation (das heißt sie ist symmetrisch, reflexiv und transitiv), die mit dem Termaufbau verträglich ist.

Wir wollen nun beweisen, dass im λ-Kalkül nicht alle Terme β-gleich sind. Falls etwa $\underline{0} = \underline{1}$ wäre, könnten wir für beliebige Terme r und s wie folgt argumentieren:

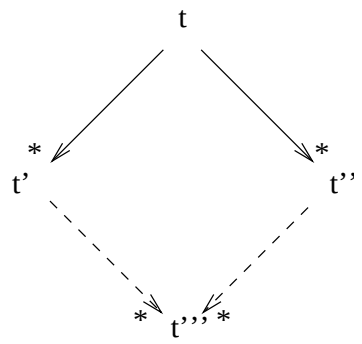


Abb. 4: Konfluenz

$$\begin{array}{rcl} \underline{0} & =_\beta & \underline{1} \\ \underline{0}(Kr)s & =_\beta & \underline{1}(Kr)s \\ s & =_\beta & r \end{array}$$

Es sind also genau dann alle λ-Terme β-gleich, wenn die Church'schen Ziffern $\underline{0}$ und $\underline{1}$ β-gleich sind. Im Folgenden werden wir jedoch zeigen, dass $\underline{1}$ und $\underline{0}$ tatsächlich verschieden sind.

Grundidee. Wollten wir den Beweis direkt auf der β-Gleichheit aufbauen, so hätten wir das Problem, dass wir bei der Umformung von $r[x := s]$ nach $(\lambda x.r)s$ den Term s raten müssten bzw. dass dieser nicht einmal sicher in r vorkäme. Wir suchen demnach eine andere Relation, die uns erlaubt, für alle β-gleichen Terme zu einem gemeinsamen Redukt hin *vorwärts* zu reduzieren.

Einschrittreduktion. Zunächst definieren wir die *Einschrittreduktion* $\rightarrow_\beta$ als die kleinste Relation auf den λ-Termen mit folgenden Eigenschaften:

$$\begin{array}{rcl} (\lambda x.r)s & \rightarrow_\beta & r[x := s] \\ r \rightarrow_\beta r' & \Rightarrow & \lambda x.r \rightarrow_\beta \lambda x.r' \\ r \rightarrow_\beta r' & \Rightarrow & rs \rightarrow_\beta r's, sr \rightarrow_\beta sr' \end{array}$$

β-Reduktion. Für eine Relation $\rhd$ bezeichnet $\rhd^*$ die *reflexiv-transitive Hülle* von $\rhd$. Es gilt $r \rhd^* s$ also genau dann, wenn es eine endliche Folge $r_0 \ldots r_n$ gibt mit $r = r_0 \rhd r_1 \rhd \ldots \rhd r_n = s$. Die *$\beta$-Reduktion* $\rightarrow_\beta^*$ bezeichnet also die reflexiv-transitive Hülle von $\rightarrow_\beta$.

Die β-Gleichheit ist also gerade die *äquivalente* Hülle von $\rightarrow_\beta$ und kann als $(\ {}_\beta\!\leftarrow \cup \rightarrow_\beta\ )^*$ dargestellt werden.

Konfluenz. Eine Relation $\rhd$ heißt *stark konfluent*, wenn es für t, t', t'' mit $t' \lhd t \rhd t''$ stets ein t''' gibt mit $t' \rhd t''' \lhd t''$. Eine Relation heißt *konfluent*, wenn ihre reflexiv-transitive Hülle stark konfluent ist (siehe Abbildung 4).

Die Church-Rosser-Eigenschaft besagt nun, dass β-gleiche Terme ein *gemeinsames Redukt* haben. Man überlegt sich leicht, dass die Church-Rosser-Eigenschaft bereits aus der Konfluenz von $\rightarrow_\beta$ folgt.

Die parallele Reduktion $\twoheadrightarrow$ wird induktiv definiert durch:

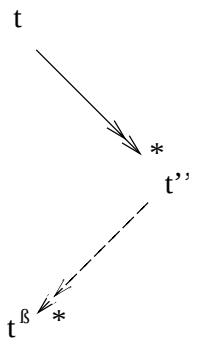


Abb. 5: Maximalität

$$
\begin{array}{rcll}
x & \twoheadrightarrow & x & x \text{ eine Variable} \\
r \twoheadrightarrow r' & \Rightarrow & \lambda x.r \twoheadrightarrow \lambda x.r' & \\
r \twoheadrightarrow r' \wedge s \twoheadrightarrow s' & \Rightarrow & rs \twoheadrightarrow r's' & \\
r \twoheadrightarrow \lambda x.t \wedge s \twoheadrightarrow s' & \Rightarrow & rs \twoheadrightarrow t[x := s'] &
\end{array}
$$

Durch Induktion zeigt man, dass diese Relation reflexiv ist, sowie dass $\rightarrow_\beta \subset \twoheadrightarrow \subset \rightarrow_\beta^*$ gilt.

Unter der *Substitutionseigenschaft* verstehen wir folgende Eigenschaft der parallelen Reduktion, die durch Induktion nach $s \twoheadrightarrow s'$ gezeigt wird.

$$r \twoheadrightarrow r' \wedge s \twoheadrightarrow s' \Rightarrow r[x := s] \twoheadrightarrow r'[x := s']$$

Superentwicklung. Wir wollen nun die starke Konfluenz der parallelen Reduktion beweisen. Dazu suchen wir eine Relation, die das gemeinsame Redukt der durch parallele Reduktion aus einem Term t hervorgegangenen Terme abhängig nur von t darstellt. Die Superentwicklung t^β, für die wir diese als *Maximalität* bezeichnete Eigenschaft beweisen wollen, ist rein syntaktisch über den Aufbau von t wie folgt definiert:

$$
\begin{array}{rcll}
x^\beta & = & x & x \text{ eine Variable} \\
(\lambda x.r)^\beta & = & \lambda x.r^\beta & \\
(rs)^\beta & = & \begin{cases} t[s := s^\beta] & \text{falls } r^\beta = \lambda x.t \\ r^\beta t^\beta & \text{sonst} \end{cases} &
\end{array}
$$

Maximalität. Es gilt: $t \twoheadrightarrow t' \Rightarrow t' \twoheadrightarrow t^\beta$, was durch Induktion über $t \twoheadrightarrow t'$ gezeigt werden kann (siehe Abbildung 5).

Mit der Maximalität zeigen wir aber gleichzeitig auch die *starke Konfluenz* der parallelen Reduktion: Denn unabhängig von der Richtung, in die wir von t aus parallel reduzieren, gelangen wir mit genau einer weiteren parallelen Reduktion zu dem gemeinsamen t^β.

Abschluss des Beweises. Betrachten wir nun die β-Reduktion als eine endliche Kette von Einschrittreduktionen. Wenn wir von einem Term t aus n_1 mal in eine Richtung und n_2 mal in die andere Richtung reduzieren, so können wir wegen $\rightarrow_\beta \subset \twoheadrightarrow$ jede dieser Einschrittreduktionen auch als parallele Reduktion schreiben. Da die parallele Reduktion außerdem reflexiv ist, lässt sich obiger Fall auch durch $n = \max(n_1, n_2)$ parallele Reduktionen in beiden Richtungen simulieren.

Wegen der Maximalität gilt für eine Kette $t \twoheadrightarrow t_1 \twoheadrightarrow t_2 \twoheadrightarrow \ldots \twoheadrightarrow t_n$, dass $t_1 \twoheadrightarrow t^\beta$. Ferner $t_2 \twoheadrightarrow t_1^\beta$ sowie $t^\beta \twoheadrightarrow t_1^\beta$, also $t_1^\beta \twoheadrightarrow t^{\beta\beta}$, mithin $t_2 \twoheadrightarrow^2 t^{\beta\beta}$. Durch Fortsetzen dieses Arguments ergibt sich $t_n \twoheadrightarrow^n t^{\beta\ldots\beta}$ mit $t^{\beta\ldots\beta}$ der n-fachen Superentwicklung. Wir folgern, dass wir von n Schritten in eine Richtung (ausgehend von t) nach wiederum n Schritten zum gemeinsamen Redukt $t^{\beta\ldots\beta}$. Mehrere hintereinander ausgeführte parallele Reduktionen sind folglich auch als Ganzes konfluent.

Wegen $\twoheadrightarrow \subset \rightarrow_\beta^*$ können wir mehrere parallele Reduktionen in eine Richtung auch als β-Reduktion schreiben, damit ist die Konfluenz der β-Reduktion gezeigt, was wiederum die Church-Rosser-Eigenschaft beweist.

Aus der Church-Rosser-Eigenschaft wiederum folgt zwingend, dass $\underline{0}$ und $\underline{1}$ genau dann verschieden sind, wenn sie kein gemeinsames Redukt haben, was offensichtlich aus den Regeln der β-Reduktion hervorgeht (weder $\underline{0}$ noch $\underline{1}$ können weiter reduziert werden). Damit ist bewiesen, dass nicht alle λ-Terme gleich sind.

Das Fixpunktprinzip

(Paul-Fiete Hartmann)

Was ist ein Fixpunkt? Wenn eine Funktion f an einer Stelle x ihr Argument x wieder zurückliefert, so hat sie an dieser Stelle einen Fixpunkt.

$$fx = x$$

Bei Funktionen, die reelle Zahlen auf reelle Zahlen abbilden, liegen alle möglichen Fixpunkte auf dem Graphen der Funktion $f(x) = x$. Wenn ein Funktionsgraph diese Gerade nicht schneidet (z. B. bei $g(x) = x + 1$), dann hat die zugehörige Funktion keinen Fixpunkt.

Im λ-Kalkül bildet man nicht reelle Zahlen auf reelle Zahlen ab, sondern Kombinatoren auf Kombinatoren. Wobei Funktionen, die das tun, wieder Kombinatoren sind (gegebenenfalls in einem erweiterten Sinne, wie im Abschnitt über die Jump-Hierarchie). Durch die besonderen Eigenschaften der Kombinatoren, kommt es zu einem erstaunlichen Satz, dem

1. Fixpunktprinzip: Zu jedem Kombinator f gibt es einen Kombinator x, so dass gilt: $fx = x$.

Um einen Beweis herzuleiten, beginnen wir mit einigen Annahmen. Günstig wäre es, wenn wir einen Orakel-Kombinator Θ hätten, der für jede Funktion f den Fixpunkt findet. Wir nehmen erst mal an, es gäbe einen solchen Kombinator. Also

$$\Theta f = x$$

folglich

$$\Theta f = f(\Theta f)$$

man könnte mutmaßen, dass Θ von der Form

$$\Theta = \lambda f.f(\Theta f)$$

ist.

Es sieht so aus, als wäre der Orakel-Kombinator gefunden – problematisch ist nur, dass Θ sich selbst verwendet. Solche Rekursion kann man nicht einfach so stehenlassen – das ist noch kein gültiger λ-Ausdruck!

Θ muss praktisch eine komplette Kopie von sich selbst erstellen. Das hört sich vielleicht erst mal ziemlich komisch an, aber man braucht nur daran zu denken, dass jedes Lebewesen in der Lage ist sich selbst zu reproduzieren. Dabei wird der komplette Bauplan (Genotyp) weitergegeben. Wir können hier probieren, Θ als $\Theta = mb$ darzustellen. Dabei steht m für Maschine und b für Bauplan. Die Maschine soll schließlich anhand des Bauplans sich selbst und einen weiteren Bauplan erzeugen. Zurück zu $\Theta = mb$.

$$mb = \lambda f.f(mbf)$$

Wieder können wir vermuten, dass

$$m = \lambda b.(\lambda f.f(mbf)) = \lambda bf.f(mbf)$$

Da im λ-Kalkül Baupläne selbst durch lebendige Kombinatoren ausgedrückt werden können, versuchen wir den Ansatz, $m = b$ zu setzen. Wir übergeben der Maschine den kompletten Bauplan von sich selbst.

$$m = \lambda bf.f(mbf) = \lambda bf.f(bbf)$$

Das ist schließlich das, was wir gesucht haben. In mathematisch präziser Form sieht das Ganze so aus:

Sei	$T = \lambda bf.f(bbf)$
und	$\Theta = TT$
Dann ist	$\Theta f = TTf = (\lambda bf.f(bbf))Tf = f(TTf) = f(\Theta f)$ $\square$

Für jeden Kombinator f ist also Θf ein Fixpunkt.

Um das zweite Fixpunktprinzip beschreiben zu können, muss die Schreibweise $\ulcorner t \urcorner$ eingeführt werden: Kennt man einen Kombinator t nur als black-box, so kann man im Allgemeinen nicht auf seine Struktur schließen. Damit dies doch möglich ist, kann man den Bauplan eines Kombinators t in einem binären Baum kodieren. Um diesen Bauplan verwenden zu können schreibt man in die Blätter des Baumes nur die Kombinatoren S und K. Wie bereits besprochen, kann ja jeder Kombinator allein mit S und K ausgedrückt werden. Den λ-Term, der in der im Abschnitt über Churche Ziffern beschriebenen Weise für diesen Baum steht, der t beschreibt nennen wir $\ulcorner t \urcorner$.

2. Fixpunktprinzip: Zu jedem Kombinator f gibt es einen Kombinator x, so dass gilt: $f\ulcorner x \urcorner = x$.

Zum Nachweis benutzen wir wieder einige heuristische Ansätze. Die erste Annahme besteht darin, dass sich x als $x = z\ulcorner z \urcorner$ darstellen lässt

$$z\ulcorner z \urcorner = f\ulcorner z\ulcorner z \urcorner\urcorner = f(PF(P\ulcorner z \urcorner\ulcorner z \urcorner\urcorner))$$

Es wäre wünschenswert, den Bauplan des Bauplans ($\ulcorner\ulcorner z \urcorner\urcorner$) berechnen zu können. Da praktisch alle Informationen über einen Kombinator im Bauplan enthalten sind, scheint es natürlich, dass es einen Kombinator U gibt, der den Bauplan von einem Bauplan herausfindet: $U\ulcorner x \urcorner = \ulcorner\ulcorner x \urcorner\urcorner$ für jedes x. Den Kombinator U gibt es tatsächlich, er soll an dieser Stelle aber nicht hergeleitet werden.

Nun kann man weiter umformen zu:

$$\begin{aligned} & z\ulcorner z \urcorner \\ & = f(PF(P\ulcorner z \urcorner(U\ulcorner z \urcorner)) \\ & = f(PF(SPU\ulcorner z \urcorner)) \\ & = f(B(PF)(SPU)\ulcorner z \urcorner) \\ & = Bf(B(PF)(SPU))\ulcorner z \urcorner \end{aligned}$$

Nun können wir

$$z = Bf(B(PF)(SPU))$$

versuchen.

Der folgende Test bestätigt, dass die Annahmen zum Erfolg geführt haben.

$$\begin{aligned} x &= z\ulcorner z \urcorner \\ &= Bf(B(PF)(SPU))\ulcorner z \urcorner \\ &= f(B(PF)(SPU)\ulcorner z \urcorner) \\ &= f(PF(SPU\ulcorner z \urcorner)) \\ &= f(PF(P\ulcorner z \urcorner(U\ulcorner z \urcorner)) \\ &= f(PF(P\ulcorner z \urcorner\ulcorner z \urcorner\urcorner)) \\ &= f\ulcorner z\ulcorner z \urcorner\urcorner = f\ulcorner x \urcorner \end{aligned}$$

Haskell und Scheme

(Franziska Glassmeier)

Praktische Anwendung findet der λ-Kalkül unter anderem in den verschiedenen funktionalen Programmiersprachen, von denen Haskell und der Lisp-Dialekt Scheme im Folgenden kurz vergleichend vorgestellt werden sollen.

Als funktionale Programmiersprachen bezeichnet man im Allgemeinen alle Sprachen, die eine auf Funktionsauswertung basierte Programmierweise mit dem λ-Kalkül als Grundlage unterstützen. Rein funktionale

Sprachen wie Haskell sind dabei sehr eng am λ-Kalkül orientiert, während Sprachen wie Scheme zusätzlich imperative Elemente enthalten. Im Gegensatz zu Haskell, wo ausschließlich die im Quelltext definierten Funktionen ausgewertet werden, können in Scheme so beispielsweise Definitionen bzw. Variablenwerte während des Programmablaufs verändert werden.

Konkret äußert sich der funktionale Stil in Scheme und Haskell z. B. in der Darstellung der Rechenoperationen. Diese werden als Funktionen aufgefasst und stellen sich in Präfixnotation wie folgt dar:

Scheme	Haskell
`(+ 2 3)`	`(+) 2 3`

Hierbei ist zu beachten, dass in Scheme eine Funktion und ihre Argumente generell durch Klammern zusammengefasst werden. Da Haskell auch die Möglichkeit bietet, die gewohntere infixe Schreibweise ($2 + 3$) zu verwenden, muss das präfixe $(+)$ hier eingeklammert werden.

Allgemein werden Funktionen/Werte wie folgt definiert:

Scheme	Haskell
`(define x 10)`	`x = 10`
`(define f (lambda (x) (+ x 5)))`	`f = \x -> x + 5`
oder kürzer	
`(define (f x) (+ x 5))`	`f x = x + 5`

Anzumerken ist, dass in Scheme Currying nicht automatisch vorgesehen ist, so dass alle geforderten Argumente auf einmal übergeben werden müssen. In Haskell dagegen werden durch die strengere Bindung an den Lambda-Kalkül nacheinander so viele Argumente eingesetzt, wie gegeben sind. Dennoch kann Currying in Scheme durch folgende Schreibweise ermöglicht werden:

```
(lambda (a) (lambda (b) (+ a b )))
```

Wie bereits erwähnt, können Variablenwerte in Scheme auf imperative Art während des Pogrammablaufs verändert werden. Um einer durch `(define x 10)` definierten Variable zwischendurch einen neuen Wert zu zuweisen, wird `(set! x 30)` verwendet.

Neben der Schreibweise unterscheiden sich Haskell und Scheme grundsätzlich in ihren Auswertungsstrategien. Haskell basiert auf so genannter *fauler Auswertung* („lazy evaluation“), bei der die Funktionsdefinitionen zunächst nur als Versprechen einen Wert zu liefern angesehen werden. Die tatsächliche Auswertung erfolgt erst während des Programmablaufs, sobald ein Wert wirklich gefragt ist. Für $K = \lambda ab.a$ würde beispielsweise der b entsprechende Ausdruck nie betrachtet. Im Gegensatz dazu werden in Scheme alle Funktionsargumente schon vor der Anwendung vorsorglich ausgewertet. Dies hat u. a. zur Folge, dass undefinierte Ausdrücke als Parameter auf jeden Fall zum Programmabbruch führen, unabhängig davon, ob ihr Wert im Endeffekt wirklich verwendet wird. Um dieses Problem zu umgehen, gibt es in Scheme für bestimmte Fälle so genannte 'special forms', wie z. B. die `if`-Anweisung im folgenden Beispiel zur Berechnung einer überall definierten Erweiterung des Kehrwertbegriffes:

```
(define kehrwert
  (lambda (n)
    (if (= n 0)
        0
        (/ 1 n))))
```

Wäre `if` eine Funktion, würde im Fall $n = 0$ der undefinierte Ausdruck `(/ 1 0)` ausgewertet, was zum Programmabbruch führen würde. Da `if` nach Definition aber zuerst das erste Argument, und je nach dessen Wert das zweite bzw. dritte auswertet, tritt dieses Problem nicht auf.

Eine weitere *special form* in Scheme ist `let`. Es erlaubt die Vereinfachung komplizierter Ausdrücke und erhöht somit die Übersichtlichkeit:

```
  (let ((a 10) (b 20)) (+ a b))
≡ ((lambda (a b) (+ a b)) 10 20)
⇒ 30
```

Zu beachten ist hierbei, dass a und b im Scheme-Beispiel logisch unabhängig voneinander gebunden werden und sich daher nicht aufeinander beziehen können. Im entsprechenden Haskell-Ausdruck `let...in` ist grundsätzlich Rück- und auch Selbstbezüglichkeit möglich:

```
let
    a = 10
    b = 2 * a
in
    a + b
```

$\Rightarrow 30$

Um auch in Scheme Rekursion ausdrücken zu können ohne umständlich auf Fixpunktkombinatoren zurückgreifen zu müssen, steht die *special form* `letrec` zur Verfügung.

Darüber hinaus sei bemerkt, dass die unterschiedlichen Auswertungsstrategien von Haskell und Scheme zu subtilen Unterschieden in den bevorzugten Programmierstilen führen. Die `map`-Funktion, die eine Funktion f auf eine Funktion abbildet, die eine Liste `as` = $\langle a_1, a_2, \ldots \rangle$ auf eine Liste $\langle fa_1, fa_2, \ldots \rangle$ abbildet, ist ein kurzes Beispiel hierfür:

- Haskell:

```
my_map f [] = []
my_map f (head:tail) = (f head):(my_map f tail)
```

  In diesem Beispiel zeigt sich eine weitere Besonderheit von Haskell, das so genannte *pattern matching*: Während der Ausführung wird von oben nach unten vorgehend überprüft, in welches Funktionsmuster die gegebenen Argumente passen. Im Beispiel wird für eine leere Liste als Eingabe eine leere zurückgegeben und für alle anderen Fälle die allgemeine Funktion in der zweiten Zeile ausgewertet.

- Scheme:

```
(define my-map
  (lambda (f as)
    (letrec
     ((reverse
       (lambda (todo done)
         (if (null? todo)
             done
           (reverse (cdr todo) (cons (car todo) done)))))
      (map
       (lambda (todo done)
         (if (null? todo)
             (reverse done '())
           (map (cdr todo) (cons (f (car todo)) done))))))
     (map as '()))))
```

  Wichtig bei dieser Implementation der `map`-Funktion ist die endrekursive Form. Hierbei wird die Funktion `f` zunächst nacheinander auf das jeweils aktuell erste Element von `todo` angewendet und das Ergebnis in `done` abgelegt. Anschließend wird die nun rückwärts aufgebaute Liste mit `reverse` umsortiert. Gegenüber der normal-rekursiven Darstellung wie im Haskell-Beispiel, die dort auf Grund der faulen Auswertung auch die sinnvollere ist, hat Endrekursion in Scheme den Vorteil, dass das Zwischenergebnis als eine Liste gespeichert werden kann und sich nicht viele Einzelwerte bis zum Zusammenfügen am Rekursionsschluss ansammeln.

Das $3n + 1$ Problem

(Anita Müller)

Mit Hilfe eines einfachen Programmes soll das Arbeiten mit unendlichen Bäumen in Haskell erläutert werden. Unser Programm basiert auf dem $(3n+1)$ Problem. Dies beinhaltet die Frage, ob sich ein gegebener Startwert mit Hilfe der wiederholt ausgeführten Funktion

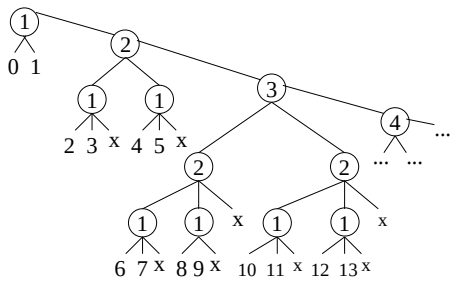


Abb. 6: Der Baum für das $3n + 1$-Problem

$$f(n) = \begin{cases} 3n+1 & \text{wenn } n \text{ ungerade} \\ n/2 & \text{wenn } n \text{ gerade} \end{cases}$$

in endlich vielen Schritten zu 1 reduzieren lässt. In unserem Programm gehen wir davon aus, dass dies möglich ist.

Es kann also von jeglicher Startzahl ausgegangen werden, die sich dann mit einer endlichen Anzahl von Schritten zu 1 reduziert. Ein Beispiel dafür wäre: $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Somit ist es möglich jeder Zahl die Anzahl von Reduktionschritten bis zur 1 zuzuweisen. Im Beispiel der 3 wären dies 8 Schritte. Jedoch kann von der 1 noch immer periodisch weiter gerechnet werden: $1 \to 4 \to 2 \to 1$, das heißt das Programm soll immer bis zur ersten 1 rechnen und den periodischen Rest weglassen. Das Programm soll nun die Anzahl der Reduktionschritte aller Zahlen innerhalb eines gegebenen Intervalles berechnen und das Schrittzahlmaximum heraussuchen. Hierfür wird für jede Zahl im Intervall die Reduktionskette und die Anzahl der Reduktionen berechnet und dies in einer Liste abgelegt, aus der dann das Maximum ermittelt werden kann. Normalerweise würde es reichen die Reduktionen des Intervalls zu berechnen und die maximale Anzahl der Reduktionsschritte auszugeben. Hier wird das Ganze aber an eine Baumstruktur übegeben, denn die Struktur des Baumes ist in diesem Falle weitaus effizienter und besser zu bedienen als die einer Liste, da bei einer Liste häufig zwischen den Daten hin- und hergesprungen werden muss. Die Idee dieser Ablage ist, dass Speicherplatz gegen Rechenzeit getauscht wird. Sobald eine Reduktion zu einer schon bekannten Zahl führt, wird der schon gespeicherte Wert genutzt. Betrachten wir als Beispiel den Startwert 10. Im ersten Schritt wird $10 \to 5$, von der man die Anzahl der Reduktionen bereits zuvor berechnet hat. Deswegen braucht man nun, um die Anzahl der Reduktionen von 10 auszurechnen, zum Wert der Reduktionszahl von 5 nur noch 1 hinzuzuzählen. Man nutzt also das schon Berechnete und spart somit wiederholtes Ausrechnen.

Wie man in der Skizze 6 des Baumes erkennen kann, ist der Baum auf Zweierpotenzen aufgebaut. Das heißt, alles geht von einem Rückgrat aus, an dem sich unendlich viele Knoten befinden können. Im Beispiel sind dies die Zahlen in den Kreisen, die diagonal nach unten verbunden sind. Von diesen Knoten gehen Blätter ab. Diese tragen die Anzahl von Reduktionen für die jeweilige Zahl als Wert. Je größer die Zahl im Knoten ist, desto größer ist auch die weitere Verzweigung daran. Die Stellen, an denen x stehen sind leer, das heißt, es ist nicht von Interesse, was dort steht.

Nur durch faule Auswertung ist es in Haskell möglich, mit unendlich großen Bäumen zu arbeiten, da Haskell immer nur das, was zur Berechnung benötigt wird, nicht aber den restlichen Teil des Programmes berechnet. In anderen Programmiersprachen, die nicht mit fauler Auswertung arbeiten sind unendlich goße Datenstrukturen eigentlich nicht möglich, da dies zu viel für diese Programme wäre.

Kettenbrüche

(Ulrike Wenig)

Neben den bekannten Darstellungen von reellen Zahlen als Dezimalbrüche beziehungsweise Folgen von echten Brüchen gibt es eine weitere weniger bekannte Zahlendarstellung, die beim Rechnen ihre eigenen Vor- und Nachteile hat: die Kettenbruchdarstellung.

Was also sind Kettenbrüche? Dies soll hier am Beispiel der Zahl π dargestellt werden. Als Dezimalzahl ist $\pi = 3.1415\ldots$, dies entspricht $3 + 1/7$ aber nicht genau $1/7$, sondern eher wie 1 durch $7 + 1/15$, aber nicht genau $1/15$ sondern eher wie 1 durch $15 + 1$, aber nicht genau 1 sondern eher wie 1 durch $1 + 1/292$, aber nicht genau $1/292$, sondern ...

In allgemeiner Form heißt das:

$$x = p_0 + q_0/(p_1 + q_1/(p_2 + q_2/(p_3 + q_3/\ldots)))$$

dabei bezeichnet man $(p_n; q_n)$ als Paar. Durch die Festlegung, wie viele Paare $(p_n; q_n)$ man ausgeben will, legt man die Genauigkeit eines Kettenbruches fest. Bei einem regelmässigem Kettenbruch ist jedes $q = 1$.

Zum Addieren, Subtrahieren, Multiplizieren und Dividieren benötigt man folgende Hilfsfunktion:

$$z(x, y) = (axy + bx + cy + d)/(exy + fx + gy + h)$$

mit den Tupeln (a, b, c, d) und (e, f, g, h). Durch das Ersetzen von a bis h durch die Zahlen ± 1 und 0 lassen sich nun die Grundrechenarten erreichen. Bei der Addition wird aus (a, b, c, d) und (e, f, g, h) $(0, 1, 1, 0)(0, 0, 0, 1)$. Bei der Subtraktion: $(0, 1, -1, 0)(0, 0, 0, 1)$. Multiplikation: $(1, 0, 0, 0)(0, 0, 0, 1)$. Und die Division wird durch das Einsetzen von $(0, 1, 0, 0)(0, 0, 1, 0)$ erlangt. Wenn man für x und y Kettenbrüche einsetzt, lässt sich nun auch mit diesen problemlos rechnen.

Der Zusammenhang zwischen diesem Rechnen mit Kettenbrüchen und dem λ-Kalkül ist die Faulheit: Faulheit bedeutet, nur das auszurechnen, was man benötigt und kein bisschen mehr. Will man mit Kettenbrüchen rechnen, legt man meist zuvor fest, welche Genauigkeit man von seinem Ergebnis erwartet. Das Programm nimmt dann immer nur soviele Paare (p_n, q_n) wie es braucht um bis zu dieser Stelle genau zu sein. Bei dieser Art der Rechnung braucht man jedoch einige Regeln, vor allem die Intervall-Arithmetik. Hierzu ein Beispiel: Man hat festgestellt, dass der Wert des Kettenbruchs x zwischen 2.25 und $2.\bar{3}$ liegt, da bisher lediglich die Paare $(2, 1)$ und $(3, 1)$ bekannt sind. Das Intervalll zwischen diesen beiden Werten ist der sogenannte Vertrauensbereich. Der des Kettenbruchs y sei zwischen 6.1 und 6.9. Somit liegt der Vertrauensbereich für die Addition von x und y zwischen 8.3 und 10. Wenn man nun für x und y weitere Paare berücksichtigt, kann auch der Vertrauensbereich genauer werden. Entweder wird der Vertrauensbereich nach einem Paar so genau, dass eine weitere Stelle festgelegt werden kann oder man nimmt das nächste Paar. Das Programm versucht hierbei immer das jeweils günstigere Paar aus x oder y zu nehmen (also das mit dem bisher ungenaueren Vertrauensbereich) damit es weniger rechnen muss (Faulheit!).

Desweiteren muss berücksichtigt werden, dass als Intervallgrenze auch die Unendlichkeit in Frage kommen kann. Es muss also auch das Rechnen mit der Unendlichkeit als Wert definiert werden. Um sich den Umgang mit den Vertrauensbereichen zu erleichtern ist es nützlich sie zu grösseren zusammenzufassen. Beispiel: Bereich 1 sei $6 - 8$ während Bereich 2 von $10 - 12$ sei; also werden wir die Intervalle zusammenfassen zu $6 - 12$.

Ist ein Kettenbruch z so genau eingegrenzt, dass, wenn wir ihn in der Form $z = p + 1/z'$ mit ganzzahligem p und $z' \geq 1$ schreiben, die Zahl p festgelegt ist, kennen wir das nächste Paar der Kettenbruchentwicklung, das wir zurück liefern können, und versuchen im folgenden nach einer einfachen Transformation der Koeffizienten a bis h mit derselben Methode die Kettenbruchentwicklung von z' zu bestimmen.

Wenn man sich an diese und einige weitere Rechenregeln (die den Rahmen dieser Doku sprengen würden) hält, ist es angenehmer mit Kettenbrüchen als mit Dezimalen zu rechnen, da sehr einfach beliebig genaue Ergebnisse erhalten werden können.

Rechnen mit unendlichen Potenzreihen

(Felicitas Thorne)

Wir haben bereits gesehen, wie wir mit der Programmiersprache Haskell unendlich lange Kettenbrüche darstellen können. Genauso haben wir die Möglichkeit, unendlich lange Potenzreihen der Form

$$f(x) = c_1 \cdot x^p + c_2 \cdot x^{p+1} + c_3 \cdot x^{p+2} + \ldots$$

darzustellen. Dazu werden im Folgenden die wichtigsten Definitionen und Elemente kurz erklärt, die ein Programmcode enthalten muss, um solche Potenzreihen darzustellen. Zunächst einmal muss man sich überlegen, wie man Potenzreihen überhaupt darstellen kann. Wir fassen Potenzreihen als (sortierte) Listen auf, die aus Koeffizienten und ganzzahligen Exponenten bestehen.

Es ist sinnvoll, mögliche Lücken zwischen den Potenzen füllen, weil sich einige Algorithmen (insbesondere Multiplikation) in solcher Darstellung besser formulieren lassen.

Beispiel:

$$\sin = \quad x^1 \qquad -\tfrac{1}{6} \cdot x^3 \qquad +\tfrac{1}{120} \cdot x^5 + \ldots$$
$$+0 \cdot x^2 \qquad +0 \cdot x^4$$

Außerdem ist ein wichtiges und grundlegendes Element in unserem Programmcode eine definierte Funktion, mit der wir festlegen können, bis zu welcher Potenz der Computer das Ergebnis ausgeben soll und verhindern somit, dass wir eine unendliche Potenzreihe, also Liste, als Ausgabe erhalten. Schon hier lässt sich das Prinzip der Faulheit erkennen: Wir berechnen nur das, was wir brauchen.

Nachdem wir jetzt die wichtigsten Elemente, die für diesen Programmcode benötigt werden, kennengelernt haben, wollen wir uns jetzt ansehen, wie wir mit solchen Potenzreihen rechnen können.

Wir beginnen mit der Addition und der Subtraktion. Die Grundidee hierbei ist, dass wir uns die Koeffizienten der beiden Potenzreihen der Reihe nach ansehen. Wir erinnern uns: Potenzreihen sind Listen. Damit können wir die Potenzen als Paar ausdrücken. Wir schreiben also ein Programm, das sich zunächst die Exponenten der von den ersten Paaren der Liste beschriebenen Monome ansieht. Von diesen Paaren schreibt es jetzt das mit dem kleinsten Exponenten hin und geht anschließend zum Paar mit dem nächstgrößeren Exponenten über. Wenn nun die Exponenten der ersten Paare der Listen gleich sind, kombiniert das Programm diese beiden Koeffizienten entspreched und erhält so den neuen Exponenten für x^p. Dieses Verfahren wendet das Programm auf alle Koeffizienten der beiden Potenzreihen an. Falls eine zu Ende ist trifft unser Programm auf einen leere Liste. Dann werden einfach nur noch die Paare der anderen Liste aufgeschrieben. Wenn beide Listen leer sind, wird als Ergebnis 0 zurückgegeben.

Eine andere Möglichkeit, mit solch einer Potenzreihe zu rechnen, ist sie zu differenzieren. Man betrachtet wieder jedes Element der Potenzreihe einzeln. Das Programm, das wir schreiben müssen, um die Potenzreihe zu differenzieren, arbeitet also folgendermaßen: Es sieht sich das erste Paar der Liste, also das Monom, an. Falls in dem Koeffizienten $c \cdot x^p$ gerade $c = 0$ oder $p = 0$ ist, fällt er weg und das Programm geht zum nächsten Koeffizienten über. In allen anderen Fällen geht das Programm nach der üblichen Ableitungsregel vor: $p \cdot c \cdot x^{p-1}$.Allerdings dürfen wir keinen x^0-Term wegfallen lassen, weil sonst eine Lücke in der Potenzreihe entsteht, was wir aber, wie wir am Anfang gesagt haben, gerade vermeiden wollen.

Wir können jetzt für jede Rechenart eine Funktion definieren, die entsprechend mit unendlich langen Potenzreihen rechnet. Entscheidend bei jeder Rechnung mit diesen Potenzreihen ist, dass wir jeden Koeffizienten berechnen können, den wir brauchen. Wir geben an, bis zu welcher Potenz die Liste ausgegeben werden soll und der Computer rechnet dann so weit, bis er uns die gewünschte Ausgabe liefern kann.

Das Halteproblem

(Hanne Hardering)

Mit Hilfe des λ-Kalküls können wir Computerprogramme darstellen. Eine große Hilfe zum Arbeiten mit Programmen wäre ein Programm, welches, wenn man ihm den Quelltext eines Programmes als Eingabe gibt, sagen kann, ob dieses Programm nach einer endlichen Zahl von Rechenschritten ein Ergebnis liefert. Dieses Programm, welches über ein anderes Programm sagen kann, ob es jemals anhält oder nicht, wird im Folgenden als „Halte-Erkenner“ kurz auch H bezeichnet. So ein Halte-Erkenner würde z.B. einen einfachen Beweis des Fermat'schen Satzes liefern. Dieser Satz besagt, dass $a^n + b^n = c^n$ für ganzzahlige $a, b, c > 1$ und $n > 2$ keine Lösung hat. Mit dem Halte-Erkenner könnte man erkennen, ob ein Programm, das systematisch alle Möglichkeiten für a, b, c und $n > 2$ testet bis es eine Lösung findet, jemals anhält oder nicht. Hielte es an, wäre der Satz widerlegt. Liefe es endlos weiter, wäre er bewiesen. Doch obwohl der Halte-Erkenner so nützlich wäre, kann es ein solches Programm nicht geben. Im Folgenden wird diese Behauptung bewiesen.

Vor dem eigentlichen Beweis muss jedoch zunächst ein anderes Problem geklärt werden. Jedes Programm, das man schreiben kann, lässt sich als Kombinator im λ-Kalkül darstellen. Das Problem dabei liegt im „Black-Box-Prinzip“ der Kombinatoren. Das heißt, dass der Halte-Erkenner nicht in den Kombinator hineinschauen kann, um z.B. nicht terminierende Prozesse zu finden. Die Lösung für dieses Problem liegt darin, dass man den Halte-Erkenner nicht auf den zu testenden Kombinator selbst, sondern auf seinen „Bauplan“ (siehe unter „Fixpunktprinzip“), also auf seine Baumdarstellung, anwendet.

Wir wollen also einen Halte-Erkenner H konstruieren, in den man die Baumdarstellung eines Kombinators x also $\ulcorner x \urcorner$ eingibt und welcher „wahr“ zurückgibt und anhält, falls der eingegebene Kombinator terminiert, und „falsch“ zurückgibt und anhält, falls der eingegebene Kombinator nicht terminiert.

Wir argumentieren über einen indirekten Beweis. Angenommen der Halte-Erkenner existierte, dann könnten wir auch einen Kombinator konstruieren, welcher ein Argument x erhält, auf welches er H anwendet. Er stoppt, wenn $H(x) = F$, und hält niemals an, wenn $H(x) = W$. Ein nicht terminierender Prozess lässt sich im λ-Kalkül durch $\Omega = (\lambda x.xx)(\lambda x.xx)$ ausdrücken, da dieser Term immer wieder zu sich selbst reduzierbar ist. Der Kombinator seltsam entspricht also der Verknüpfung von H mit Z durch B, kurz BZH, wobei Z dem

λ-Term $Z = P\Omega F$ entspricht. Wendet man Z auf „wahr“ also den Kombinator K an, so reduziert Z zu Ω, terminiert also nicht. Wendet man Z auf „falsch“ also den Kombinator KI an, reduziert Z zu einer Form, die nicht weiter reduzierbar ist. Für BZH gilt nun also: $BZH^{\ulcorner}x^{\urcorner} = Z(H^{\ulcorner}x^{\urcorner}) = P\Omega F(H^{\ulcorner}x^{\urcorner}) = (H^{\ulcorner}x^{\urcorner})\Omega F$. Wir erhalten also einen Kombinator, der einen ewig rechnenden Kombinator liefert, wenn eine Baumdarstellung eines terminierenden Kombinators eingegeben wird, und umgekehrt. Dies widerspricht jedoch dem zweiten Fixpunktprinzip, welches besagt, dass es zu jedem Kombinator f einen Kombinator x gibt, so dass gilt $f^{\ulcorner}x^{\urcorner} = x$. Wir wissen, dass die Kombinatoren B und Z existieren. Damit ist bewiesen, dass H nicht existieren kann. Das Halteproblem ist also nicht lösbar.

Außer dem Halteproblem gibt es noch andere unberechenbare Probleme. Zwei Beispiele dafür sind das Totalitätsproblem und das Äquivalenzproblem. Das Totalitätsproblem beschäftigt sich mit der Frage, ob P total ist, das heißt, ob ein Programm P bei allen Eingaben anhält. Wenn man jedoch erkennen kann, ob ein Programm bei allen Eingaben anhält, kann man dies nutzen um auch das Halteproblem zu lösen. (Man betrachte ein Programm, das seine Eingabe ignoriert.) Da dieses unlösbar ist, kann auch das Totalitätsproblem nicht gelöst werden. Das Äquivalenzproblem beschäftigt sich mit der Frage, ob zwei Programme P und E die gleichen Aufgaben erfüllen.

Die Jump-Hierarchie

(Robert Bamler)

Wie wir in Abschnitt über das Fixpunktprinzip gesehen haben, ist das Halteproblem im λ-Kalkül nicht lösbar. Trotzdem können wir darüber nachdenken, was wäre, wenn wir auf eine höhere Instanz zugreifen könnten, die in der Lage wäre, das Halteproblem zu lösen. In diesem Kapitel werden wir sehen, dass immer wieder neue unlösbare Probleme auftreten, sobald man ein Orakel einführt, das die Antwort auf ein bekanntes unlösbares Problem gibt. Es lässt sich somit eine Hierarchie von unlösbaren Problemen nach dem Grad ihrer Unlösbarkeit definieren.

Um das Lösen eines möglicherweise eigentlich nicht lösbaren Problems mathematisch präzise auszudrücken, definieren wir ein Orakel, das die Lösung zu diesem unter Umständen nicht lösbaren Problem gibt. Ein solches Orakel verhält sich zwar ähnlich wie ein Kombinator, es lässt sich jedoch im Allgemeinen nicht durch einen λ-Kombinator ausdrücken.

Da die Menge aller λ-Kombinatoren abzählbar ist, kann jeder Kombinator eindeutig durch eine natürliche Zahl benannt werden. Es gibt also eine Menge M, die Teilmenge der natürlichen Zahlen ist und genau die Zahlen enthält, die einen terminierenden λ-Kombinator darstellen, auch wenn sich diese Menge auf Grund der Unlösbarkeit des Halteproblms nicht algorithmisch ermitteln lässt. Unter einem ‘terminierenden’ Kombinator verstehen wir hier einen Kombinator, der eine Normalform hat.

Das Orakel $\underline{M}$ soll genau dann ‘wahr’ liefern, wenn es auf eine Church’sche Ziffer angewendet wird, die in der Menge M enthalten ist, andernfalls ‘falsch’. $\underline{M}$ gibt also an, ob ein durch eine natürliche Zahl beschriebener Kombinator terminiert und liefert damit zu der Beschreibung jedes Kombinators die Antwort auf das Halteproblem.

Ein solches Orakel lässt sich für alle Teilmengen der natürlichen Zahlen und damit auch für alle Teilmengen der λ-Terme definieren. Zwei unterschiedliche Teilmengen der natürlichen Zahlen A und B haben also ihre eigenen Orakel $\underline{A}$ und $\underline{B}$ für die jeweils durch sie beschriebenen Probleme. Lässt sich durch Kombination aus allen λ-Kombinatoren und dem Orakel $\underline{B}$ ein Orakel bauen, das die gleichen Rechenregeln wie das Orakel $\underline{A}$ erfüllt, so kann das Problem B als “schwieriger” als das Problem A angesehen werden: Allein aus dem Orakel für die Menge B und den λ-Kombinatoren kann ein Orakel für die Menge A gebaut werden. In diesem Fall nennen wir A ‘Turing reduzierbar’ zu B und schreiben $A \leq_{\mathrm{T}} B$. Diese Relation ist transitiv, das heißt wenn für drei Teilmengen der natürlichen Zahlen A, B und C gilt: $A \leq_{\mathrm{T}} B$ und $B \leq_{\mathrm{T}} C$, dann gilt auch $A \leq_{\mathrm{T}} C$, wie sich leicht beweisen lässt.

Interessant wird diese Überlegung nun, wenn wir zwei Mengen zu einer zusammenführen. Dazu definieren wir für zwei Teilmengen der natürlichen Zahlen X und Y:

$$X \oplus Y := \{2n | n \in X\} \cup \{2m+1 | m \in Y\}$$

Mit dieser Definition gibt es in $X \oplus Y$ für jedes Element aus X und Y genau ein Element. Außerdem lässt sich die “Herkunft” für jedes Element aus $X \oplus Y$ eindeutig bestimmen: Die Anwesenheit von geraden Zahlen in $X \oplus Y$ wird durch die Elemente von X, die von ungeraden Zahlen durch die Elemente von Y verursacht.

Für eine solche zusammengeführte Menge $X \oplus Y$ gelten nun die folgenden leicht zu zeigenden Eigenschaften:

$$X \leq_T X \oplus Y \qquad \text{und} \qquad Y \leq_T X \oplus Y$$

$X \oplus Y$ ist also sowohl schwieriger als X, als auch schwieriger als Y. Es ist außerdem das leichteste Problem, dass diese beiden Eigenschaften erfüllt, wie einfach zu beweisen ist.

Wir wollen nun zeigen, dass es zu jedem unlösbaren Problem A ein schwierigeres unlösbares Problem $j(A)$ gibt, das heißt, dass A Turing-reduzierbar ist zu $j(A)$, aber $j(A)$ nicht Turing-reduzierbar ist zu A. Dieses schwierigere Problem $j(A)$ nennen wir den Jump von A. Wir werden sehen, dass das Halteproblem für die Menge Λ^A, die alle λ-Terme mit möglicherweise dem Orakel $\underline{A}$ enthält, gerade diese beiden Bedingungen erfüllt. Die Menge $j(A)$ ist also die Teilmenge der natürlichen Zahlen, die genau die Zahlenrepräsentationen aller terminierenden λ-Kombinatoren enthält, die zusätzlich zu den bekannten Regeln zum Aufbau von λ-Kombinatoren auch auf das Orakel $\underline{A}$ zurückgreifen dürfen.

$$j(A) := \{n \mid n = \ulcorner t \urcorner, t \in \Lambda^A, t \text{ hat eine Normalform}\}$$

Wie wir bereits bei der Einführung in die Baupläne gesehen haben, lässt sich jeder Bauplan in den durch ihn beschriebenen Kombinator übersetzen. Da $j(A)$ die Zahlenrepräsentation des Orakels $\underline{A}$ enthält, lässt sich also mit Hilfe der λ-Kombinatoren das Orakel $\underline{A}$ aus diesem Bauplan erstellen. A ist also Turing-reduzierbar zu $j(A)$.

Angenommen $j(A)$ wäre Turing-reduzierbar zu A, so gäbe es einen Term $q \in \Lambda^A$, der das Problem $j(A)$ löst. Das bedeutet, dass dieser Term q genau dann 'wahr' zurück gibt, wenn er auf eine Zahl angewendet wird, die in der Menge $j(A)$ enthalten ist. Da $j(A)$ genau die Beschreibungen der terminierenden λ-Terme mit dem offensichtlich terminierenden Orakel $\underline{A}$ enthält, gibt q genau dann 'wahr' zurück, wenn es auf die Beschreibung eines terminierenden Kombinators aus Λ^A angewendet wird, andernfalls 'falsch'.

Wenn wir diesen Wahrheitswert qn analog zum Beweis des Halteproblems zuerst auf einen nicht terminierenden und dann auf einen terminierenden Kombinator aus A anwenden, erhalten wir beispielsweise $qn\Omega I = (\lambda n.qn\Omega I)n$, wobei Ω ein beliebiger nicht terminierender und I ein terminierender λ-Kombinator ist. Ist qn gleich 'wahr', so ist $qn\Omega I$ gleich Ω, andernfalls I. Man erhält also genau dann einen terminierenden Kombinator, wenn der durch n beschriebene Ausgangskombinator nicht terminiert und umgekehrt reduziert der Term genau dann zu dem nicht terminierenden Kombinator Ω, wenn der Ausgangskombinator terminiert. Das 2. Fixpunktprinzip sagt jedoch aus, dass es ein Argument für $\lambda n.qn\Omega I$ gibt, sodass $\lambda n.qn\Omega I$ angewandt auf dieses Argument zu dem Kombinator terminiert, der durch das Argument beschrieben wird. Dieser Satz wurde zwar nur für den λ-Kalkül ohne Orakel bewiesen, er lässt sich jedoch auch auf den λ-Kalkül mit Orakeln erweitern. Wir sind also auf einen Widerspruch zum 2. Fixpunktprinzip gestoßen und unsere Annahme $j(A) \leq_T A$ war falsch.

Busy Beaver

(Marco Schreiber)

1962 entwickelte der amerikanische Mathematiker Tibor Rado das BusyBeaver Problem: Er wollte wissen, wie viele Striche eine Turing-Maschine auf ihrem Magnetband schreiben kann, die danach anhält. Die Turing-Maschine ist ein in den 30er Jahren von Alan Turing entwickeltes Gerät, das aus einem endlosen Magnetband, einem Lese-/Schreibkopf und mehreren Zuständen besteht. Das Magnetband ist in Kästchen unterteilt, auf denen ein Strich oder ein Leerzeichen stehen kann. Der Lese-/Schreibkopf liest ein Zeichen von dem Magnetband und die Turing-Maschine führt gemäß des aktuellen Zustandes drei Operationen aus: Sie schreibt ein neues Zeichen an den Platz des gelesenen, bewegt den Kopf nach links, rechts oder gar nicht und wechselt in einen neuen Zustand. Die hierfür nötigen Informationen sind für jeden Zustand in einer Tabelle festgehalten.

Da man mit einer Turing-Maschine auch durch Codierung den λ-Kalkül ausdrücken kann, ist es möglich, mit ihr jede berechenbare Funktion darzustellen. So wird eine Turing-Maschine, die die Funktion $f(x) = x^2$ berechnet, bei der Eingabe von drei Strichen neun Striche ausgeben. Die BusyBeaver Funktion $BB(n)$ gibt nun, wie gesagt, die maximale Anzahl von Strichen an, die eine Turing-Maschine mit n Zuständen nach dem Anhalten auf dem Band hinterlassen kann. Diese Anzahl liegt für einen Zustand bei 1, für zwei Zustände bei 4, für drei bei 6, für vier bei 13 und für fünf Zustände bei mindestens 4098. Die Werte der Zustände eins bis vier sind durch Ausprobieren aller möglichen Turing-Maschinen mit n Zuständen ermittelt worden, der fünfte Wert ist nicht sicher. Die BusyBeaver Funktion ist nämlich nicht berechenbar:

Beweis(indirekt): Annahme: Es gibt eine Funktion BB, die von einer Turing-Maschine mit m Zuständen berechenbar ist. Ohne Einschränkung können wir annehmen, dass diese Maschine mehr als 20 Zustände hat.

1. Konstruiere Maschine, die m Zustände hat und m Einsen schreibt, wobei $m > 20$.
2. Konstruiere Maschine, die n Einsen zu $3n$ Einsen macht mit weniger als 20 Zuständen.
3. Verbinde dies zu einer Maschine mit $m + 20$ Zuständen, die auf dem leeren Band $3m$ Einsen schreibt.
4. Konstruiere Turing-Maschine mit m Zuständen, welche $3)$ ausführt und dann die Funktion BB anwendet:

Zustände: $m + 20 + m = 2m + 20$
Ausgabe: $BB(3m)$

Daraus folgt

$$BB(3m) < BB(2m + 20).$$

Da die Funktion BB streng monoton wachsend ist, gilt wegen $m > 20$

$$BB(3m) > BB(2m + 20).$$

Dies steht im Widerspruch. Unsere Annahme war demnach falsch. □

Meta Loopless Sort

(Christian Hercher)

Kann man in einer Programmiersprache nur mit Vergleichen und Ausgaben sortieren? Dieser Frage wollen wir uns in diesem Abschnitt widmen, indem es um ein Programm in Haskell gehen soll, welches uns den Quelltext eines anderen, speziellen Programmes liefert. Dabei soll das zu erstellende Programm ohne Verwendung von Schleifen eine gegebene Liste sortieren können. Ein Programm, welches ein anderes schreibt (und nichts anderes haben wir hier vor), wollen wir hier auch Meta-Programm nennen. Weiterhin heißt dieser Abschnitt „Loopless Sort“, da unser Meta-Programm ein anderes erstellen soll, welches ohne Verwendung von Schleifen eine Liste gegebener Länge sortieren kann.

Das Sortieren der Elemente wird dabei durch „Merge-Sort“ erreicht – ein Verfahren, welches nach dem „Divide & Conquer“-Prinzip arbeitet. Das Problem wird dabei auf das Lösen von Teilproblemen zurückgeführt. Hier bedeutet dies, dass erst kleinere Teil-Listen sortiert werden, welche dann zu einer sortierten Gesamtliste zusammengefügt werden.

Der zu schreibende Quelltext, welcher dieses Merge-Sort umsetzen soll, darf aber nach Aufgabenstellung nur aus `If`-Verzweigungen und Ausgabe-Elementen bestehen; rekursiver Selbstaufruf – wie in der „normalen“ Umsetzung von Merge-Sort – ist uns nach Voraussetzung nicht gestattet. Wegen diesen Einschränkungen ist unser Meta-Programm leider nicht in der Lage ein einziges Programm zu schreiben, welches jede mögliche Liste sortieren kann. Doch für jede vorgegebene Länge schreibt unser Metaprogramm einen korrekten Quelltext.

Eine elegante Variante unser Meta-Programm zu schreiben besteht darin, sich zu „merken“, was noch „zu tun“ ist. Dies erreicht man dadurch, dass man den Rest des Programmes, der noch abzuarbeiten ist, einem Parameter übergibt. Dieser Parameter wird meist Fortsetzung (oder auch engl. “continuation”) genannt, eben da in ihm alles noch Abzuarbeitende festgehalten wird.

Auch das schon verarbeitete Programm können wir einer Variablen übergeben. Dies ist nützlich, wenn man die ganze Ausgabe auf einmal erledigen möchte.

Nun betrachten wir unser Meta-Programm.

```
symbmerge sofar cont [] as         = cont (sofar++as)
symbmerge sofar cont as []         = cont (sofar++as)
symbmerge sofar cont (a:as) (b:bs) =
           If a b (symbmerge (sofar++[a]) cont as (b:bs))
                  (symbmerge (sofar++[b]) cont (a:as) bs)
```

Hier wird die Prozedur `symbmerge` definiert, welche symbolisch das Zusammenfügen zweier Teillisten zu einer einzigen via Merge-Sort nachstellt. Die Variable `sofar` stellt die bisher schon erstellte (und sortierte) Gesamtliste dar. Die ersten zwei Zeilen zeigen, wie `symberge` eine Liste und eine leere zusammenfügt: Das Ergebnis ist die andere Liste. Da diese übrig bleibende Liste schon vorsortiert ist, kann sie einfach an den „Rest“ angefügt werden. Dies geschieht durch Anhängen der entsprechenden Liste an `sofar` (das „++“).

In der dritten Definitionsvorschrift von `symbmerge` wird erläutert, was die Funktion bei der Anwendung auf zwei nicht-leere Listen machen soll. Als erstes soll dabei nämlich „`If a b`“ als Teil des zu schreibende Sortierprogramm ausgegeben werden. Dies soll in unserem Quelltext „`If a<b`“ ausdrücken. Danach folgt in Klammern, was der Quelltext für den „Then“- und den „Else“-Zweig als Anweisungen enthalten soll. Da a und b jeweils die Listenanfänge der Teillisten waren, ist eines der beiden das kleinste Element aus beiden Listen. Dieses wird nun an `sofar` – an die bisherige Gesamtliste – angefügt. Die continuation (hier das, was mit der fertigen Liste gemacht werden soll) bleibt dabei unverändert. Nun müssen nur noch die beiden Listen (eine ohne ihr erstes Element) zusammen gefasst werden. Dies geschieht eben durch den rekursiven Selbstaufruf von `symbmerge`.

Für zwei gegebene, vorsortierte Listen fügt uns die Prozedur `symbmerge` diese zwei zu einer zusammen. Dabei muss beachtet werden, dass man diese Prozedur dann immer mit der leeren Liste für `sofar` aufrufen müsste, da ja zu Beginn noch nichts in die Gesamtliste aus beiden Teillisten aufgenommen wurde.

Um nun Merge-Sort vollständig zu implementieren, müssen wir nicht nur vorsortierte Listen zusammensetzen können, sondern auch eine unsortierte in zwei zerlegen. Erhalten wir dabei Listen der Länge Null oder Eins, so sind die übergebenen Listen nach Definition sortiert. Dann kann ich diese wieder (mit der Prozedur `symbmerge`) zusammen fassen. Die hier beschriebene, noch fehlende Funktion (welche sozusagen das Sortieren leitet) erhalten wir durch die Prozedur `ssort`:

```
ssort cont []  = cont []
ssort cont [a] = cont [a]
ssort cont as  = ssort (\l-> (\r-> ssort symbmerge [] cont l r) ar) al
                   where (al,ar) = split as.
```

Erhält `ssort` die leere Liste, so hat es selbst nichts mehr zu tun und übergibt cont die Liste als Ergebnis. Analoges trifft auch für den Fall zu, dass die an `ssort` übergebene Liste genau ein Element enthält. In diesen beiden Fällen sind die übergebenen Listen ja schon sortiert. Die dritte Definition bearbeitet nun Listen mit mindestens zwei Elementen. Zuerst wird dabei die Liste `as` in `al` und `ar` (linke und rechte Teilliste) gesplittet. Dies wird durch die Funktion `split` geleistet, auf welche hier aber nicht näher eingegangen wird. Nun folgt der rekursive Selbstaufruf der Prozedur `ssort`. Dabei wird als zu sortierende Liste `al` übergeben. Ist diese Liste dann sortiert worden, so wird auf sie die continuation angewendet, welche sich hier nun zu „`(\r->ssort symbmerge [] cont al r) ar`“ vereinfacht. Also muss als nächstes die Teilliste `ar` sortiert werden. Dann sagt uns die continuation, dass diese zwei sortierten Teillisten mit Hilfe von `symbmerge` zu einer sortierten Gesamtliste vereinigt werden. Abschließend wird mit der ursprünglichen continuation, mit der `ssort` anfänglich aufgerufen wurde, fortgefahren.

Unser nun geschriebenes Meta-Programm erzeugt für jede Eingabe einer beliebig langen Liste ein Programm, welches diese Liste sortiert. Es folgt ein Beispiel:

```
Main> ssort Result ["a","b","c"]
If "a" "c" ("a" "b" (If "c" "b" (Result ["a","c","b"])(Result ["a","b",c"])) (Result ["b"
,"a","c"]))(If "c" "b" (If "a" "b" (Result ["c","a","b"])(Result ["c","b","a"])) (Result
["b","c","a"])).
```

Fortsetzungen

(Achim Hildenbrandt)

Der Continuation-Passing-Style (CPS) ist eine sehr interessante Programmiertechnik, die insbesondere in funktionalen Sprachen eine wichtige Rolle spielt. Man übergibt nicht wie in den meisten Sprachen üblich, nur Werte an Prozeduren, sondern man übergibt das ganze noch abzuarbeitende Programm. Als Beispiel wollen wir die Addition im CPS implementieren.

```
(define (c+ a b cont)
              (cont (+ (a b)))
```

Zur Erklärung: Die Prozedur `c+` erwartet drei Argumente. Sie benötigt mit `a` und `b` die beiden Zahlen welche wir addieren möchten. Als drittes Argument wird noch die Fortsetzung `cont` gebraucht. `c+` macht jetzt Folgendes: Sie führt die Fortsetzung aus und übergibt ihr als Argument die Summe von `a` und `b`. Wenn also beispielsweise `a` den Wert 4 und `b` den Wert 3 sowie `cont` gerade `(lambda (n) (* 4 n))` ist, so erhalten wir nach dem Ausführen von `c+` gerade `(* 4 (+ 4 3))`.

Allgemein gesagt arbeitet eine CPS-Funktion wie folgt: Sie rechnet ein Ergebnis aus und reicht dieses an die Fortsetzung weiter. Anders formuliert kann man auch sagen, daß die Fortsetzung uns sagt, was wir mit einem Ergebnis tun sollen sobald wir es ausgerechnet haben.

Call/CC. Eine ganz besonders trickreiche Anwendung von Fortsetzungen ist die *Special Form* `call-with-current-continuation`. Das Besondere dabei ist, dass wir damit die aktuelle Fortsetzung bekommen können. Wir können diese dann später verwenden um mit der gleichen Forsetzung fortzufahren. Es ist sogar möglich die gleiche Fortsetzung mehrmals und mit unterschiedlichen Argumenten aufzurufen.

Übergeben wir etwa dem Scheme-Interpreter folgendes:

```
(define sa-cont 0)

(string-append "The text is: "
   (call-with-current-continuation
     (lambda (c)
       (set! sa-cont c)
       ( c"[dummy]"))))
```

so bindet `(lamdba(c)...)` an die aktuelle Fortsetzung, in unserem Fall `(string-append "The text is: " □)`. Dabei steht □ für ein Argument vom Typ String.

Als nächstes setzt `set!` die Variable `sa-cont` auf die aktuelle Fortsetzung. Die letzte Zeile ruft nun die aktuelle Fortsetzung mit `"[dummy]"` auf. Wir können jetzt aber auch mit `sa-cont` die Fortsetzung von außen aufrufen und ihr jedes beliebiges Argument übergeben, und damit direkt auf die oberste Ebene springen.

Die IO-Monade

(Vincent Dombrowski)

In reinen funktionalen Sprachen wie Haskell ist jede Funktion eine wohldefinierte eindeutige Abbildung, d.h. dass sie immer denselben Wert liefert, wenn sie mit den denselben Parametern aufgerufen wird. Das ist ein wesentlicher Unterschied zu anderen, *imperativen*, Programmiersprachen, in denen “Funktionen” nur syntaktisch aussehen wie Funktionen, sich aber nicht so verhalten.

Seien a, b natürliche Zahlen, dann müssten

```
a = random(10)
b = a
```

und

```
a = random(10)
b = random(10)
```

das gleiche bedeuten.

Da die „Pseudofunktion“ `random` eine Zufallszahl liefert, muss die Schlussfolgerung nicht zwingend gelten. In Haskell sind Umformungen dieser Art jederzeit möglich (Gleiches kann durch Gleiches ersetzt werden). Man spricht bei dem zuvor gezeigten von *fehlender referentieller Transparenz*. Haskell dagegen ist *referenztransparent*.

Das obige Beispiel lässt sich auch auf Input bzw. Output übertragen und führt uns zu einem schwerwiegenden Problem: Funktionale Sprachen sind sehr gut geeignet irgendwelche Berechnungen durchzuführen, aber es ist dem Benutzer bisher nicht möglich mit dem Programm zu kommunizieren („Wie kann rechnen die Welt verändern?“).

In der Vergangenheit gab es verschiedene Ansätze mit diesem Problem umzugehen, aber erst in den späten Neunziger Jahren hat sich ein brauchbarer Ansatz herauskristallisiert: Der *monadische I/O*. Wie die I/O Monade funktionniert soll im Folgenden beschrieben werden.

Wenn wir die Welt um uns herum aktiv verändern wollen, aber nur geistige Berechnungen durchzuführen in der Lage sind, können wir von unserem Vorhaben einen Plan entwickeln, den wir dann nur noch irgendwie zur

Ausführung bringen müssen. Das war eigentlich auch schon die Idee, die sich hinter dem monadischen I/O verbirgt. Haskell erstellt einen Plan eine Aktion auszuführen und der Interpreter *HUGS* führt diesen Plan dann aus.

Ein Beispiel: Wir wollen ein Programm schreiben, das den String `"Hallo!"` ausgibt. Dazu gibt es in der Haskellbibliothek eine Funktion `putStr`, die gerade einen String abbildet auf einen Plan, diesen String auszugeben:

```
main = putStr "Hallo!"
```

Um zwei Pläne zu verknüpfen kennt Haskell den Kombinator `(>>)`. Dieses Werkzeug ermöglicht es uns, zwei Pläne hintereinander auszuführen.

Zum Beispiel:

```
  main = (>>) (putStr "Hallo!") (putStr "Bye!")
```

In diesem Fall ist ist `main` ein Plan, erst den String `"Hallo!"` und dann `"Bye!"` auszugeben.

Um vollständig mit dem Programm zu kommunizieren, fehlt jetzt nur noch die Implementierung des *Input*. Der Plan, ein Zeichen einzulesen heißt in Haskell `getChar`. Unser Ziel ist es nun, zwei Pläne derart zu verknüpfen, dass der zweite Plan auf den ersten Plan „zugreifen“ kann. Für eine derartige Implementierung existiert in Haskell der Operator `(>>=’’)`. Auch hier wieder ein Beispiel:

```
 main = (>>=) getChar (\c -> putStr ("You entered: " ++ [c]))
```

Ruft man `main` im Hugs auf, so erwartet das Programm als Eingabe einen Char. Daraufhin gibt das Programm dieses Zeichen durch die `putStr`-Funktion zurück.

Mit den beiden Kombinatoren `(>>)` und `(>>=)` haben wir nun eine Möglichkeit gefunden, Input und Output in Haskell zu implementieren und somit vollwertig mit einem Programm zu kommunizieren.

Literatur

[1] S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[2] Henk Barendregt. The type free lambda calculus. In Barwise [4], chapter D.7, pages 1091–1132.

[3] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, second revised edition, 1984.

[4] Jon Barwise, editor. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1977.

[5] John C. Martin. *Introduction to languages and the theory of computation*. McGraw-Hill, Inc., 1991.

[6] Piergiorgio Odifreddi. *Classical Recursion Theory. The Theory of Functions and Sets of Natural Numbers*, volume 125 of *Studies in Logic and the Foundation of Mathematics*. North Holland, 1989.

[7] Masako Takahashi. Parallel reductions in λ–calculus. *Information and Computation*, 118:120–127, 1995.