Einführung in den λ-Kalkül

Klaus Aehlig Thomas Fischbacher

Pfingsten 2001

Der vorliegende Text bildet eine Ausarbeitung der Unterlagen, die für den Kurs "λ-Kalkül" auf dem CdE Pfingsttreffen 2001 verwendet wurden. Außerdem enthalten diese Unterlagen einige Kapitel zu Themen, die aus Zeitgründen nicht behandelt werden konnten, sowie einige weiterführende Kapitel für das Selbststudium der Kursteilnehmer.

Die Autoren behaupten an keiner Stelle eigene Resultate zu präsentieren. Aussagen, die nicht explit einem Urheber zugeordnet sind wurden von den Autoren als "Folklore" betrachtet.

Die Autoren möchten sich an dieser Stelle noch mal bei den Organisatoren des Treffens bedanken ohne deren Einsatz es nie zu diesem Kurs gekommen wäre. Dank gilt natürlich auch insbesondere den Kursteilnehmern, die dem Kurs mit viel Interesse und Engagement gefolgt sind.

Inhaltsverzeichnis

1 Von der richtigen Schreibweise zur vollen Abstraktion: das Spiel beginnt

Wenn man *ganz* genau aufpaßt, stellt man fest, daß manche in der Mathematik weitverbreiteten Sprechweisen eigentlich ein wenig unbeholfen wirken und am eigentlichen Wesen der Sache vorbeigehen. Ein Beispiel: wenn wir den Sachverhalt ausdrücken wollen, daß eine Funktion f an jeder Stelle den Wert Null annimmt, sagen wir gerne "f ist identisch gleich null". Ganz genau betrachtet ist das eigentlich Käse. Haben wir es hier etwa mit zwei verschiedenen Arten von Gleichheit in der Mathematik zu tun, "gleich" und "identisch gleich"?[1] Wie läßt sich der Unterschied zwischen diesen beiden Begriffen fassen? Natürlich kann man sich auf den Standpunkt stellen, daß hier wohl jeder weiß, was gemeint sein soll, und wir uns deswegen über diese Dinge keine weiteren Gedanken machen müssen. Es lohnt sich aber bisweilen (und insbesondere auch hier), dann, wenn man festgestellt hat, daß irgend etwas bei genauer Betrachtung sonderbar aussieht, ein wenig mehr darüber nachzudenken, was das eigentliche Problem ist, ob da mehr dahinter stecken könnte, und ob es verwandte interessante Probleme gibt.

Des Pudels Kern ist wohl, daß wir hier irgendwie ungeschickt über Funktionen als solche sprechen. Sehen wir uns ein anderes Beispiel an: Stellen wir uns vor, wir hätten zwei Funktionen f und g. Das sollen ganz langweilige Funktionen von der Art sein, wie man sie in der Schule betrachtet, also etwa Abbildungen von den reellen Zahlen in die reellen Zahlen. Wir wissen, wie wir Zahlen addieren, aber *Funktionen* können wir zunächst eigentlich nicht addieren. Dennoch gibt es einen naheliegenden Begriff, den wir mal mit "punktweise Summe" bezeichnen wollen, nämlich diejenige Funktion, die jedes x auf $f(x) + g(x)$ schickt. Diese wird üblicherweise mit $f + g$ bezeichnet, was aber wie gesagt unbeholfen ist[2].

---

[1] Zur Ehrenrettung der Mathematik sei gesagt, daß dem natürlich nicht so ist. Zwei Funktionen sind gleich, wenn sie an jeder Stelle gleich sind, also meint "f ist identisch gleich Null", dasselbe wie "f ist die Nullfunktion", also $\forall x.f(x) = 0$. Der eigentliche Grund ist ein anderer. Aus algebraischen Gründen (auf die wir hier nicht eingehen) ist das Wort "Null" überladen. Es bezeichnet einerseits die reelle Zahl, andererseits die Funktion die konstant diesen Wert annimmt und noch so manches mehr (in gewissen Kontexten sogar die rationale Zahl 1).

[2] Der eigentliche Grund, warum es hier (und so oft) doch funktioniert ist, daß das überladene Zeichen $+$ an Hand der "Typen" von f und g disambiguiert werden kann: f und g sind "Funktionen" also vom Typ $\mathbb{R} \to \mathbb{R}$ und das $+$ von diesem Typ ist eben etwas anderes als das $+$ in $\mathbb{R}$. So was ähnliches wird uns in Form von Typklassen von `Haskell` wieder begegnen, aber unser vorrangiges Ziel ist erst mal, uns in die freie Welt des ungetypten λ-Kalküls zu begeben und da kann man so was nicht gebrauchen!

Außerdem sei noch angemerkt, daß jene (und weitergehende) Überladung(en) es einem der Kursleiter in forgeschrittenen Algebravorlesungen schwer bis unmöglich gemacht hat, dem Stoff zu folgen: Wer kann schon für drei bis sieben Zeichen (wechselnde) Typinformationen ständig parat haben, nur um den Tafelanschrieb zu disambiguieren?

Der andere Kursleiter möchte bei dieser Gelegenheit einen Seitenhieb auf die theoretische Physik loslassen: wenn die mathematischen Objekte, die man behandelt, irgendwelches "ganz abstruses Zeugs" sind, mit dem man sich zum ersten Mal beschäftigt, und die man noch überhaupt nicht richtig durchschaut, seien es nun Tensoren, Grassmann-Zahlen, oder andere Verrücktheiten, können solche ihr-wißt-schon-wie-das-gemeint-ist Konstruktionen, die der

Es muß nicht Addition sein, man sieht sofort ein, daß das bei Multiplikation, Potenzierung, eigentlich mit jeder "Rechenart" genau dasselbe ist und hier so was wie "punktweises irgendwie-Verknüpfen" zugrunde liegt. Aber einen eigentlichen Begriff davon, eine Sprache, um so was wirklich festzunageln, haben wir noch nicht. (Wir werden sie aber bald haben, und sie wird unglaublich einfach sein[3].)

Wie denn nun richtig? Bleiben wir dazu vorerst mal bei der "Funktion f, die identisch gleich Null ist". Eigentlich ist das ganz einfach: man muß nur den Mut aufbringen, die Dinge beim Namen zu nennen. Wir nennen die Funktion, die überall den Wert Null annimmt, die *Nullfunktion*, und können dann stattdessen sagen, "f ist die Nullfunktion". Wir können sogar noch viel direkter werden und einfach schlichtweg sagen: "f ist die Funktion, die jedes x auf 0 abbildet". Und wenn wir jetzt konsequent unsere übliche, noch aus der Schule bekannte, Schreibweise für Abbildungen einsetzen, können wir das auf folgende vielleicht auf den ersten Blick etwas sonderbar aussehende (weil nicht oft anzutreffende) Form bringen:

$$f = (x \mapsto 0)$$

Manche werden das jetzt banal finden, aber die werden wohl noch ins Schwitzen kommen. Andere werden es als unästhetisch empfinden. Das ist jedoch nur eine Frage der Gewöhnung und kommt daher, die Symbole selten in einer derartigen Kombination gesehen zu haben. Zumindest dürfte unbestreitbar sein, daß es konsequent ist, und die Dinge so wiedergibt, wie sie sind.

Kommen wir zu dem anderen oben kurz angesprochenen Fall: punktweise Addition von Funktionen. Die Summe zweier Funktionen, die wir üblicherweise etwa in der Art $s = f + g$ schreiben würden[4], ist in der direkten Notation gerade

$$s = (x \mapsto f(x) + g(x))$$

Notationen sind Schall und Rauch. Wichtig ist nur, daß wir unsere Gedanken in strenger Weise formalisieren. Wie wir die Symbole wählen, ist unwichtig. Wir könnten beispielsweise nach Lust und Laune auch vereinbaren, keinen Pfeil zu verwenden, und stattdessen $x \mapsto 0$ als $\lambda x.0$ zu schreiben. Dann wäre unsere Notation gerade die des λ-Kalküls. Mit Rücksicht auf den Leser verbleiben wir jedoch bei der vielleicht etwas gewohnteren Pfeil-Notation[5].

Erklärende verwendet ohne noch groß drüber nachdenken zu müssen, die Angst vor solchen Begriffen wesentlich steigern.

[3]Einer der Kursleiter kann sich noch gut erinnern, wie er selbst nach und nach gelernt hat, auf jene Weise zu denken, und er muß sagen, daß es für sein eigenes Verständnis wohl einer der bedeutendsten intellektuellen Durchbrüche bisher überhaupt war, eben weil es so fundamental ist.

Der andere Kursleiter kann sich zwar nicht mehr so genau erinnern, aber es wird ihm wohl ähnlich gegangen sein.

[4]Bei dieser Gelegenheit möchten sich die Kursleiter noch mal für das Abschiedsgeschenk der Teilnehmer bedanken, in dem sie auf ihr Zitat "$(f + g)(x)$ ist Müll" hingewiesen wurden.

[5]Hier zeigt sich, daß es nicht nur von Vorteil ist, das Skript vor dem Kurs (teilweise)

Ein Beispiel soll hier noch kurz gebracht werden: Hinternanderausführung von Funktionen, auch Komposition genannt, üblicherweise $c = f \circ g$ (oder von Kategorikern auch fg, oder aber auch gf) geschrieben. In unserer Notation:

$$c = (x \mapsto f(g(x)))$$

Was geschieht hier eigentlich? In einer Gleichung stehen auf beiden Seiten des Gleichheitszeichens irgendwelche Werte. Wir machen nichts anderes, als *Funktionen als Werte* wie alle anderen auch aufzufassen. Das ist sinnvoll und konsequent. Und es bringt uns den nächsten Schritt auf unserem "Weg zur Erleuchtung": Davon, was die *Summe* von zwei vorgegebenen Funktionen sein soll, haben wir jetzt halbwegs eine Vorstellung. Aber wir müssen die Funktionen, die wir addieren wollen, vorher festlegen, haben also die *Addition* von Funktionen als solche noch nicht richtig im Griff. Was macht die "Addition von Funktionen"? Wir nehmen zwei Funktionen und erhalten die punktweise Summe. Funktionen sind Werte! Die Funktions-Addition ist also eine *Abbildung* von zwei Funktionen auf eine neue Funktion (die die punktweise Summe darstellt). Abbildungen sind Funktionen! Wir können also die *Funktions-Addition als solche* auf folgende Weise schreiben:

$$\texttt{fun\_add} = ([f, g] \mapsto (x \mapsto f(x) + g(x)))$$

Hier ist eine Zwischenbemerkung angebracht, was die Klammern angeht: Wir schreiben Paare (Trippel,...) mit eckigen Klammern damit keine Gefahr besteht, sie mit ordinären gruppierenden Klammern zu verwechseln. Dennoch haben wir strenggenommen immer noch zwei Arten von runden Klammern: gruppierende Klammern, wie z.B. die die ganze rechte Seite umfassenden und Funktionsklammern, etwa bei $f(x)$ Wir wollen deswegen dazu übergehen, auch Funktionsklammern einfach nur als gruppierende Klammern aufzufassen. Das geht, ohne daß ein Problem mit Mehrdeutigkeiten entstehen kann, aber es bedeutet, daß wir eine Funktion von zwei (oder mehr) Argumenten $f(x, y)$ uminterpretieren müssen als eine Funktion eines Paares $f([x, y])$.

Nach dieser Bemerkung aber wieder zurück zu unserer Funktions-Addition. Was haben wir gerade gemacht? Wir sind von einem konkreten Begriff — der Summe zweier vorgegebener Funktionen — zum abstrakten Begriff der Addition von Funktionen übergegangen, indem wir das, was vorher konkret war (die Funktionen f und g) zum Input einer Funktion gemacht haben. Wir haben einen konkreten Begriff zu einem mächtigeren, allgemeineren *abstrahiert.*

Ob Addition von Funktionen, Multiplikation von Funktionen, Potenzierung, oder sonst eine andere komische Rechenart, die wir uns vielleicht aus lustig selbst definieren, das Spielchen von zuvor läßt sich auch für andere Fälle ganz

zu schreiben, aber erst nacher endgültig fertig zu stellen: dem intendierten Leser, der ja ein ehemaliger Kursteilnehmer ist, dürfte die λ-Notation inzwischen vielleicht vertrauter sein. Wenden wir uns mit dieser Bemerkung also an den Leser, wie er war, als er diese Lektion besuchte — oder auch einen geschätzten Leser, der den Kurs nicht besucht hat.

analog wiederholen. Das zeigt, daß hier wohl etwas Grundlegenderes dahintersteckt. Und in der Tat können wir das durch Abstraktion herausholen. Dafür müssen wir aber zuerst dazu übergehen, die Addition, die wir bisher mit $+$ zum Ausdruck gebracht haben, als Funktion zweier Argumente, oder, wie zuvor erklärt, als Funktion eines Paares aufzufassen. Nennen wir diese Funktion `add`.

$$\texttt{add} = ([a, b] \mapsto a + b)$$

Analog könnten wir Funktionen von Paaren definieren, die Multiplikation, Subtraktion, Potenzierung, sonstwas umsetzen. Damit können wir nun schreiben:

$$\texttt{fun\_add} = ([f, g] \mapsto (x \mapsto \texttt{add}([f(x), g(x)])))$$

Der Trick ist jetzt, *nach* `add` *zu abstrahieren* und damit das Wesen des punktweisen Verknüpfens zu einen eigenen Begriff herauszudestillieren[6]

$$\Phi = (v \mapsto ([f, g] \mapsto (x \mapsto v([f(x), g(x)]))))$$

Was ist das? Eine Funktion, die eine Funktion, die eine Verknüpfung darstellt (wie beispielsweise `add`) abbildet auf etwas, was den Begriff des punktweisen Verknüpfens von Funktionen mit dieser Verknüpfung implementiert.

Es mag vielleicht etwas verwirrend sein, wenn man so was das erste Mal sieht. Eigentlich ist bisher nichts wirklich böses passiert, wir sind in kleinen und einfachen Schritten vorgegangen, und inzwischen an einer Stelle gelandet, an der es beim besten Willen nicht mehr harmlos zugeht.

Wir sind noch nicht beim λ-Kalkül angelangt, aber wir nähern uns ihm in großen Schritten. Der herrlich abstrakte Begriff der punktweisen Verknüpfung von Funktionen wird dabei in unserem Zugang eine ganz zentrale Rolle einnehmen, allerdings nicht in der Form, wie er bisher vorgestellt wurde. Das liegt daran, daß es im Lambda-Kalkül nur Funktionen gibt, und wir diese Vorgabe mit unserer Formulierung noch nicht erreicht haben, weil noch ein Dorn zu entfernen ist: es taucht in unserer Beschreibung noch das komische ad hoc Konstrukt eines Paares auf, das es loszuwerden gilt.

Wofür verwenden wir hier Paare? Um Funktionen von zwei Argumenten zu erhalten. Es gibt nun einen sehr pfiffigen Trick, der von einem Mathematiker des 19. Jahrhunderts namens Schönfinkel gefunden wurde, und allgemein als “Currying” bekannt ist. (Nach einem Logiker des 20. Jahrhunderts, Haskell Curry, der intensiv davon gebraucht hat, und nach dem auch die Programmiersprache `Haskell` benannt ist. Es kommt öfters vor, daß eine Idee nicht mit den Namen ihres Erfinders in Verbindung gebracht wird, sondern mit den Namen dessen, der sie populär gemacht hat. Beispielsweise beim Gauß-Algorithmus oder bei der Polyakov-Wirkung war das auch nichts anderes.)

[6] Manch Kursteilnehmer wird sich beim Anblick dieser Funktion an die “punktweise **Ph**erknüpfung” erinnert fühlen...

Drolligerweise können wir diesen Trick verstehen, *ohne* ein neues Kaninchen aus dem Hut zaubern zu müssen (wie soll's auch anders gehen? wir wollen Dinge loswerden, und das schaffen wir bestimmt nicht dadurch, daß wir neune hinzunehmen!).

Wir müssen uns lediglich auf das berufen, was wir bisher kennen – Abstraktion nach Vorgaben von außen (man könnte auch sagen: nach Parametern, oder nach freien Variablen), und einsehen, daß wir dort, wo wir vorschnell auf Paare gekommen sind, zu hastig gegessen haben, und die Mahlzeit zwecks besserer Verdauung besser kauen hätten sollen.

Vergessen wir dies also für's erste wieder:

$$\texttt{add} = ([a, b] \mapsto a + b)$$

Wir machen jetzt in gewisser Weise dasselbe Spielchen von oben nochmal, aber etwas einfacher. "Addition" ist ein abstrakter Begriff. Da wollen wir hin. Wir tun erst mal so, als wüßten wir nichts von diesem Abstraktum, und würden nur wissen, wenn uns jemand zwei Zahlen a und b gibt, was die Summe s von a und b sein soll.

$$s = a + b$$

¿Wir abstrahieren jetzt, *aber nur nach* b! Das liefert uns den abstrakteren Begriff des "a-Hinzuzählens"

$$\texttt{add}_a = (b \mapsto a + b)$$

Wenn a als 1 vorgegeben war, ist das der Begriff "1 mehr", wenn a gerade 2 war, der Begriff "2 mehr" und für alle anderen a entsprechend. Wie wenden wir so was an? Ganz einfach:

$$\texttt{add}_a(5) = a + 5$$

Jetzt brennt es uns richtig auf den Nägeln, in einem *zweiten Schritt* nach a zu abstrahieren, und schon haben wir den abstrakten Begriff der Addition:

$$\texttt{add} = (a \mapsto \texttt{add}_a) = (a \mapsto (b \mapsto a + b))$$

Konkretes Arbeiten mit diesem Additions-Begriff mag auf den ersten Blick etwas gewöhnungsbedürftig aussehen, weil das Ergebnis von `add` selbst wieder eine Funktion ist, die ein Argument will:

$$\texttt{add}(2) = (b \mapsto 2 + b)$$
$$(\texttt{add}(2))(3) = 2 + 3$$

Aber eigentlich ist das so ganz konsequent und nicht weiter schwierig.

Informell gesprochen kann dieser Begriff alles, was der vorige, der ein Paar verwendete, auch konnte (sogar ein wenig mehr), aber er kommt ohne ein zusätzliches Konstrukt wie ein Paar aus.

Currying ist gerade der Trick, eine Funktion von mehreren — sagen wir zwei — Argumenten umzuschreiben in eine funktionswertige Funktion, gemäß dem Schema

$$[p, q] \mapsto \varphi_{p,q} \qquad \rightsquigarrow \qquad p \mapsto (q \mapsto \varphi_{p,q})$$

Dabei ist $\varphi_{p,q}$ ein beliebiger, von p und q vielleicht abhängiger Ausdruck.

Sehen wir uns nochmal das punktweise Verknüpfen in der bisherigen Formulierung an, mit der wir nicht ganz zufrieden waren:

$$\Phi = (v \mapsto ([f, g] \mapsto (x \mapsto v([f(x), g(x)]))))$$

Wir wollen hier ein wenig Unkraut jäten. Das innere Paar ist leicht aufzulösen: wir müssen einfach vereinbaren, daß die Verknüpfung v als gecurryte Funktion (also in der Art wie unser zweiter Anlauf bei der Additions-Funktion, der ohne Paar auskam) angegeben werden muß, und $v([f(x), g(x)])$ umschreiben zu $(v(f(x)))(g(x))$. Für das Paar $[f, g]$ in den Argumenten müssen wir nochmals curryen. Das liefert dann zusammen:

$$\Phi = (v \mapsto (f \mapsto (g \mapsto (x \mapsto (v(f(x)))(g(x))))))$$

Und *dieser* Begriff von punktweiser Verknüpfung ist jetzt wirklich wichtig, die anderen, vorigen, die noch Paare verwendet haben, wollen wir jetzt ganz schnell vergessen. (Es waren wirklich andere Begriffe, und so gesehen hätten wir vielleicht nicht dieselben Namen verwenden sollen.)

Das sonderbare ist: wir sind hier so abstrakt geworden, daß jetzt nur noch Funktionen auftauchen. Generell nennen "λ-Kalkül-Anhänger" wenn sie mit "Nichteingeweihten" sprechen Funktionen, die selbst Funktionen als Argumente übernehmen bisweilen *Funktionen höherer Ordnung*, aber weil es für uns so selbstverständlich ist, daß Funktionen auch nur Werte sind, macht so eine Unterscheidung für uns eigentlich erst mal wenig Sinn, es bedarf einfach keiner besonderen Erwähnung[7].

Solche total abstrakten Funktionen wie unsere hier, in denen nur noch andere Funktionen auftauchen, nennt man *Kombinatoren*. Einige haben Standardbezeichnungen, und wir werden noch etliche nützliche Kombinatoren kennenlernen.

[7] Und auch die *Sprechweise* macht für uns jetzt hoffentlich (bald) keinen Sinn mehr. Der Begriff Funktion höherer Ordnung setzt nämlich vorraus, daß die Funktion überhaupt eine Ordnung hat, daß man also sagen kann, wieviele Argumente sie der Reihe nach nimmt um etwas "von Grundtyp" zu liefern und wie die Argumente jeweils aufgebaut sind. Hier könnten wir Φ zwar noch den Typ

$$(\mathbb{R} \to \mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$$

zuweisen, aber davon wollen wir uns jetzt lösen.

Es bleibt jedoch zu erwähnen, daß der Begriff "Funktionen höherer Ordnung" auch gerade von mit dem λ-Kalkül vertrauten verwendet wird, nämlich dann, wenn es sich um solche Funktionen handelt, die gerade in obigen Sinne einen Typ besitzen. Dies ist zwar nur ein kleiner Teil aller Funktionen, aber das macht es ja erst recht zum gefragten `feature`. Etwas ähnliches wird uns bei der Besprechung von `Haskell` wieder begegnen.

Eine sehr wichtige aber ziemlich unscheinbare weitere Zutat ist die Identität. Diese wird konventionsgemäß mit I bezeichnet:

$$I = (x \mapsto x)$$

Natürlich ist auch das ein Kombinator. Außerdem werden wir sehen, daß konstantwertige Funktionen eine wesentliche Rolle spielen. Man könnte scherzhaft sagen, daß man den Begriff der konstantwertigen Funktion erhält, wenn man die Nullfunktion für beliebige Null betrachtet. Die zugehörige Abstraktion, die zu einer Funktion führt, die irgendwelche Werte x auf Funktionen abbildet, die immer den Wert x annehmen, sollte für uns jetzt keine Schwierigkeit mehr sein. Sie liefert:

$$K = (x \mapsto (y \mapsto x))$$

Auch das ist eine Standardbezeichnung. Zwei weitere wichtige Kombinatoren seien hier noch vorgestellt. Damit kennen wir dann insgesamt fünf Kombinatoren. Der erste davon stellt gerade den abstrakten Begriff der Komposition (Hintereinanderausführung) von zwei Funktionen dar; er wird B genannt.

$$B = (f \mapsto (g \mapsto (x \mapsto f(g(x)))))$$

Ein Beispiel ist wohl hier nicht fehl am Platze. Die Komposition von "3 mehr" und "4 mehr":

$$\begin{aligned}(B(\texttt{add}(3)))(\texttt{add}(4)) &= (x \mapsto (\texttt{add}(3))((\texttt{add}(4))(x))) \\ &= (x \mapsto (\texttt{add}(3))(4 + x)) \\ &= (x \mapsto (3 + (4 + x))\end{aligned}$$

Das erlaubt uns der Kalkül. Eine weitere Reduktion zu $(x \mapsto 7 + x)$ könnnen wir hier aber einfach so durch Reduktion durch Einsetzen und Auswerten nicht vornehmen, weil wir hier einfach gesagt noch überhaupt nichts über Rechengesetze für die Addition wissen. Aber das, was wir erhalten, sieht trotzdem schon mal nicht unvernünftig aus.

Außerdem soll hier noch kurz der P-Kombinator vorgestellt werden. Dieser macht etwas ganz sonderbares, und dahinter steckt ein kleines Geheimnis, das hier aber noch nicht preisgeben werden soll[8]. Er funktioniert so:

$$P = (a \mapsto (b \mapsto (x \mapsto (x(a))(b))))$$

Wahrscheinlich müßte man ohne den λ-Kalkül gemeistert zu haben ziemlich genial sein, um jemals auf den Gedanken zu kommen, daß sich der Begriff der

[8] Der intendierte Leser, also ein ehemaliger Kursteilnehmer wird die Bedeutung zwar schon kennen, aber er möge sich dadurch an diese Lektion erinnert fühlen. Die Kursleiter erinnern sich noch gut an den Ausruf eines Teilnehmer "Ah, *deswegen P*" und dieser Gewinn durch eigene Einsicht soll niemanden genommen werden, auch nicht demjenigen hochgeschätzen Leser, der den Kurs nicht besucht hat.

Funktionskomposition auf den Begriff der punktweisen Verknüpfung von Funktionen reduzieren läßt, wenn man zusätzlich die Identität und die Funktion, mit der sich konstantwertige Funktionen erzeugen lassen, verwendet. Grund[9]:

> Ich kann zwei Funktionen auch dadurch punktweise verknüpfen, daß ich den Wert der ersten an einem vorgegebenen Punkt als Funktion nehme, die ich auf den Wert der zweiten am vorgegebenen Punkt anwende. In meiner gecurryten Formulierung entspricht das gerade der Verwendung der Identität als Verknüpfung. Das ist dann schon fast die Hintereinanderausführung, nur will ich aber die erste Funktion eigentlich direkt auf den Wert der zweiten am vorgegebenen Punkt anwenden, und nicht erst ihren Wert am vorgegebenen Punkt als Funktion benutzen, die ich auf den Wert der zweiten Funktion anwende. Aber wenn es mir auf die Stelle nicht ankommt, kann ich mir hier schön mit der konstantwertige Funktionen erzeugenden Funktion weiterhelfen. Ich baue mir damit einfach eine Funktion, die überall die erste Funktion als Wert hat, und punktweise-verknüpfe das mit der Identität als Verknüpfung mit der zweiten Funktion. Das sieht schon mal sehr gut aus, aber am Ziel sind wir noch nicht! Ich bin jetzt angelangt bei der Komposition der punktweisen-Verknüpfung-mit-der-Identität mit der konstantwertige Abbildungen erzeugenden Funktion. (Denn wenn ich das auf die erste Funktion anwende, erhalte ich gerade was, was ich auf die zweite Funktion anwenden kann, um die Hintereinanderausführung aus erster und zweiter Funktion zu erhalten.) Es sieht so aus, als hätten wir uns im Kreis gedreht, und müßten versuchen, die Komposition über die Komposition zu erklären, doch ha! wenn wir denselben Trick nochmal anwenden, der beim ersten Mal nicht erfolgreich war, sind wir am Ziel: die Komposition der punktweisen-Verknüpfung-mit-der-Identität mit der konstantwertige Funktionen erzeugenden Abbildung ist doch nichts anderes als die punktweise-Verknüpfung-mit-der-Identität der Funktion, die konstant den Wert punktweise-Verknüpfung-mit-der-Identität hat (und über die konstantwertige Funktionen erzeugende Funktion leicht zu erhalten ist) mit der konstantwertige Funktionen erzeugenden Abbildung!

[9] Was nun folgt ist reine Angeberei. Auch der Co-Kursleiter des vortragenden hatte Probleme dem zu folgen. Trotzdem war es Bestandteil des Kurses und soll hier nicht verschwiegen werden. Außdem zeigt es schön den Unterschied zwischen natürlicher und symbolischer Sprache.

Oder kurz (in der weiter unten erklärten Notation):

$$\begin{aligned}
B &= \lambda xyz.x(yz) \\
&= \lambda xyz.I(Kxz)(yz) \\
&= \lambda xyz.\Phi I(Kx)yz \\
&= \lambda xyz.(K(\Phi I)x)(Kx)yz \\
&= \lambda xyz.\Phi I(K(\Phi I))Kxyz \\
&= \Phi I(K(\Phi I))K
\end{aligned}$$

Nachdem wir nun in etwa gesehen haben, was hier vor sich geht, und vielleicht schon in Ansätzen einen halbwegs intuitiven Begriff davon haben, was Kombinatoren sind, und was sie machen, wollen wir jetzt als Abschluß dieser Lektion uns noch ein wenig um Notationsfragen kümmern, damit wir in Zukunft ein effizientes Werkzeug haben, das uns einen sicheren Griff erlaubt.

Zuallererst wollen wir unnötige Klammern loswerden. Konkret wollen wir, daß abd ein wohlgeformter Ausdruck ist; wir könnten nun entweder vereinbaren, daß er für $a(b(c))$ stehen soll, oder daß er für $(a(b))(c)$ stehen soll. Beides ist im allgemeinen *nicht* dasselbe! Abbildungen sind assoziativ unter Komposition, aber hier setzen wir eine Abbildung als Wert in eine andere ein, wir verknüpfen sie nicht. Hier ein einfaches Beispiel, das das explizit zeigt: wir nehmen als a einfach den K-Kombinator und für b und c die Identität I. $a(b(c))$ ist dann $K(I(I)) = K(I)$ eine Funktion deren Wert (egal an welcher Stelle sie ausgewertet wird) stets die Identität ist. $(K(I))(I) = I$ hingegen ist die Identität selbst.

Die allgemeine Konvention ist nun, daß man vereinbart, daß Klammern *links* einzusetzen sind ("Linksklammerung"). Damit soll abc für $(a(b))(c)$ stehen. Im ersten Moment mag diese Konvention sonderbar erscheinen. Wir werden aber im Laufe der Zeit noch sehen, daß sie durchaus Sinn macht. Als Eselsbrücke kann man sich vielleicht die linkeste Funktion als gecurryte Funktion mehrer Argument b und c vorstellen. Man beachte, daß wir mit dieser Konvention sowohl die Funktions-Anwendungsklammern als auch einige der gruppierenden Klammern entfernt haben.

Mit dieser Konvention können wir nun bekannte Kombinatoren umschreiben; richtig viel bringt das allerdings noch nicht:

$$\begin{aligned}
B &= (g \mapsto (g \mapsto (x \mapsto f(gx)))) \\
P &= (a \mapsto (b \mapsto (x \mapsto xab)))
\end{aligned}$$

Das Problem ist, daß in solchen Ausdrücken als typische Struktur $(a \mapsto (b \mapsto (c \mapsto (\ldots$ auftaucht. Das ist weder sehr schön noch sehr nützlich, denn alle Klammern, die wir links nacheinander aufmachen, schließen rechts am Ende des Kombinators wieder genauso. Wir vereinbaren also, daß wir statt $(a \mapsto (b \mapsto (c \mapsto \varphi)))$ auch $\lambda abc.\varphi$ schreiben dürfen (mit φ einem beliebigen Term), für eine andere Zahl von Variablen entsprechend. Und weil das so schön geht, können wir die alte Pfeil-Notation auch gleich ganz vergessen. Damit sehen unsere Kombinatoren jetzt wirklich einigermaßen appetitlich aus:

$$\begin{aligned}
I &= \lambda x.x \\
K &= \lambda xy.x \\
B &= \lambda xyz.x(yz) \\
P &= \lambda abc.cab \\
\Phi &= \lambda vfgx.v(fx)(gx)
\end{aligned}$$

Man darf zu recht vermuten, daß - nach einiger Eingewöhnung - damit wesentlich streßfreier zu arbeiten ist als mit der ursprünglichen schwerfälligen Notation.

Abschließend fassen sollten wir noch mal zusammenfassen, welche Rechenregeln wir eigentlich verwenden wollen (und intuitiv auch schon immer verwendet haben). An erster und wichtigster Stelle ist die β-Regel[10] zu nennen, auch "Einsetzen in explizite Definitionen" genannt: wir setzen das Argument einfach in den Funktionskörper ein.

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

Dabei möge $t[x := r]$ bedeuten, daß wir t nehmen und alle Vorkommen von x durch r ersetzen[11]. Schließlich gibt es noch die η-Regel (wie "Extensionalität"). Wenn x nicht in t vorkommt, so ist $\lambda x.tx = t$. Diese Regel wird *in diesem Kurs* nur eine untergeordnete Rolle spielen.

[10] Es sei erwähnt, daß es auch eine α-Regel gibt, die besagt daß es auf die Namen (durch λ) gebundener Variablen nicht ankommt. Wir werden uns jedoch "α-ignorant" verhalten, d.h. uns darüber keine großen Gedanken machen, sondern diese Konvention einfach stillschweigend (richtig) verwenden. Höflicher formuliert (aber inhaltlich dasselbe) könnte man sagen, wir verwenden die "Barendregt-Konvention".

Eine technisch saubere Beschreibung dieses Problems sind de Bruijn Indices [2]: Variablen sind Nummern, die angeben wie viele (weiter unbeschriftete!) λ's übersprungen werden müssen um beim bindenden λ anzukommen. In diesem Sinne "meinen" wir natürlich $\lambda\lambda\lambda.021$ wenn wir $\lambda xyz.zxy$ als "naheliegende Abkürzung" verwenden.

[11] In der Literatur liest man oft auch "freie Vorkommen von x" und "capture free substitution". Dies ist natürlich auch hier gemeint. Auf Grund unserer Variablenkonvention können wir jedoch annehmen, daß alle Variablen "schon passend gewählt worden waren".

Konkret: es ist natürlich *nicht* der Fall, daß $(\lambda x.x)[x := y] = \lambda x.y$ oder $(\lambda y.x)[x := y] = \lambda y.y$. Vielmehr sind $\lambda x.x$ und $\lambda y.x$ natürlich "ungeschickte Schreibweisen" für $\lambda z.z$ und $\lambda z.x$ und es ist $(\lambda z.z)[x := y] = \lambda z.z$ und $(\lambda z.x)[x := y] = \lambda z.y$. Das soll's dann aber endgültig zu α gewesen sein.

2 $\Phi I = S$: Wie man die Welt mit ein bis zwei Zeichen aufbaut

Wie war das? Ein mathematisches Kapitel hat man in dem Moment verstanden, in dem man von sich behaupten kann, die Überschrift vollkommen verstanden zu haben. Also machen wir uns mal ans Werk und lüften den ersten Teil des in der Überschrift formulierten Geheimnisses: aus unseren bereits bekannten Kombinatoren

$$\begin{aligned}
\Phi &= \lambda vfgx.v(fx)(gx) \\
I &= \lambda x.x \\
K &= \lambda xy.x \\
B &= \lambda fgx.f(gx) \\
P &= \lambda xyz.zxy
\end{aligned}$$

definieren wir uns einen neuen, den wir S nennen wollen.

$$\begin{aligned}
S = \Phi I &= (\lambda vfgx.v(fx)(gx))I \\
&= \lambda fgx.I(fx)(gx) \\
&= \lambda fgx.fx(gx)
\end{aligned}$$

Die Rechenregel für diesen Kombinator erinnert uns an das “Verteilen einer Substitution über eine Applikation”. In der Tat, ist σ eine Substitution, so ist $(rs)\sigma = r\sigma(s\sigma)$ und $Srst = rt(st)$. Deswegen sollten wir uns S auch “moralisch” als “Applikation” vorstellen: Srs sagt “moralisch”, daß wir eigentlich r auf s anwenden wollen, aber beide vielleicht noch freie Variablen enthalten und damit vom Kontext abhängen. Erst wenn dieser (als Term t kodiert) zur Verfügung gestellt wird können wir die Applikation wirklich ausführen.

Damit sind wir auch schon dem Verständnis der Überschrift ein wenig näher gekommen: wir wollen uns nur mit den Rechenregeln für S und einige weitere der oben erwähnten Kombinatoren einen Ersatz für die λ-Abstraktion schaffen[12].

Fallen wir doch einfach gleich mit der Tür ins Haus und bringen erst das allgemeine Rezept und motivieren es anschließend anhand eines Beispiels[13].

[12] Dies hat auch einen historisch/philosophischen Hintergrund: die α-Regel ist so kompliziert/böse, daß es besser scheint, ohne Variablen auszukommen — aber über α wollten wir ja nicht reden.

[13] An dieser Stelle waren sich die beiden Kursleiter über das weitere Vorgehen nicht einig. Einer meinte, es sei didaktisch besser zunächst ein Beispiel zu bringen. Der andere Kursleiter war der Auffassung, daß das Vorführen eines Algorithmuses an einem Beispiel Angeberei sein, wenn der Algorithmus nicht vorher verraten wird und daher solle zunächst der Algorithmus und dann das Beispiel gebracht werden (außerdem ist es natürlich leichter, einen Algorithmus anzuwenden, der an der Tafel steht, als einen den man nur im Kopf hat). Da dieser Kursleiter gerade vortrug kam es dann auch so.

In dieser Hinsicht (also was Beispiele angeht) dürften die Kursunterlagen (im Gegensatz zum Kurs selbst!) zu diesem Kapitel wohl ein didaktischer Tiefschlag sein; aber auch das muß man mal erlebt haben.

Wir sind also in folgender Situation: wir haben einen Term t in dem x vielleicht vorkommt und sollen einen Term $\lambda^* x.t$ *ohne* explizite λ-Abstraktion finden für den gilt $(\lambda^* x.t)r = t[x := r]$. Nun, der aller einfachste Fall ist der, daß x in t gar nicht vorkommt; dies ist etwa der Fall wenn t eine von x verschiedene Variable ist, oder auch wenn t einer der Kombinatoren ist, durch die wir alles ausdrücken wollen. In diesem ist die Antwort einfach Kt, denn in der Tat ist $Ktx = t$.

Was wenn t nun eben doch die Variable x ist? Nun, dafür haben wir den Kombinator I. In der Tat, $Ir = r = x[x := r]$.

Damit bleibt nur noch ein einziger Fall, nämlich der der Applikation. Denn beim übersetzen eines λ-terms wie $\lambda xyz.t$ gehen wir selbstverständlich *von innen* nach außen vor und bestimmen erst $\lambda^* z.t$, was ein Term der nur aus Variablen, Kombinatoren und Applikationen aufgebaut ist, anschließend berechnen wir $\lambda^* y.(\lambda^* z.t)$ und so weiter. Nun, was tun wir also, falls t eine Applikation ist? Dann sind wir aber gerade in dem Fall, daß wir "moralisch" eine Applikation haben — aber eben noch die Abhängigkeit von x beachten müssen. Also definieren wir

$$\lambda^* x.rs = S(\lambda^* x.r)(\lambda^* x.s)$$

wobei wir $\lambda^* x.r$ und $\lambda^* x.s$ (nach Induktionsvoraussetzung) schon definiert haben, da es sich um kleinere Terme handelt. Überzeugen wir uns noch, daß wir es richtig definiert haben: $S(\lambda^* x.r)(\lambda^* x.s)t = (\lambda^* x.r)t((\lambda^* x.s)t) = r[x := t](s[x := t]) = (rs)[x := t]$

Wir sind unserem Ziel schon recht nahe gekommen. Über η machen wir uns keine Gedanken und wollen es diesbezüglich nicht so genau nehmen. Trotzdem bleiben noch ein paar Fragen offen — doch die sind Übung!

- Wir hatten bisher alles durch S, K und I ausgedrückt. Das sind 3 Kombinatoren. In der Überschrift war von "ein bis zwei" die Rede. Nun, I ist überflüssig, siehe Übung 6.1.
- Jetzt haben wir es mit 2 Zeichen geschafft, aber wie war das mit einem? Nun, siehe Übung 6.6.
- Wie war das mit dem versprochenen Beispiel? Nun, siehe Übung 6.7[14].

[14] Es bleibt anzumerken, daß dieses Beispiel im Kurs tatsächlich vorgerechnet wurde. Wen dies stört, der drücke eben $P = \lambda xyz.zxy$ allein durch S und K aus.

3 Daten animieren: wie man im λ-Kalkül programmiert

So langsam wird es Zeit in unseren Kalkül etwas "Daten" reinzubringen. Funktionen sind ja ganz nett, aber Fallunterscheidung und natürliche Zahlen wären zum Programmieren doch auch ganz nett...

Keine Angst, dazu müssen wir unseren Kalkül nicht erweitern; wir werden lediglich lernen anders über gewisse Funktionen zu denken.

Zunächst zu booleschen Werten, also "wahr" und "falsch". Was ist Wahrheit?[15] Diese Frage wollen wir hier nicht beantworten und uns vielmehr fragen, was wir von den Wahrheitswerten überhaupt wollen. Sprich, warum können wir nicht einfach zwei beliebige (verschiedene!) Kombinatoren als "wahr" und "falsch" auszeichnen?

Nun, die Hauptverwendung von booleschen Werten dürfte sicher die Fallunterscheidung sein, also ein Konstrukt der Form

> If b then r else s

wobei wir natürlich fordern, daß die Fallunterscheidung jeweils korrekt aufgelöst werden, also daß

$$\begin{aligned}(\text{If } \texttt{true} \text{ then } r \text{ else } s) &= r\\ (\text{If } \texttt{false} \text{ then } r \text{ else } s) &= s\end{aligned}$$

Schön, aber eine "If... then... else..." Syntax haben wir ohnehin nicht, vielleicht sollten wir uns zuerst überlegen, wie wir dieses Konstrukt dastellen? Ein jetzt nur vielleicht naheliegender, aber auf jeden Fall zentraler Gedanke ist

> Zum Glück haben wir uns noch nicht festgelegt, wie die Wahrheitswerte aussehen sollen. Dann können wir ihnen doch einfach die Arbeit übertragen.

Konkret heißt das, wir übergeben einem Wahrheitswert einen "if"-Zweig und einen "else"-Zweig und er soll sich das richtige heraussuchen! "If b then r else s" wird damit einfach zu brs — und außerdem gewinnen wir Gleichungen zur Definition der Wahrheitswerte. Es muß nämlich gelten

$$\texttt{true}\, rs = r$$
$$\texttt{false}\, rs = s$$

Dies legt die Definition $\texttt{true} = \lambda rs.r = K$ und $\texttt{false} = \lambda rs.s = KI$ nahe.

[15] Joh 18,38. Man beachte, daß diese Frage unbeantwortet blieb!

Zum Rechnen mit booleschen Werten sei nur erwähnt, daß der Definitionsbereich stets endlich ist, so daß mit vollständiger Fallunterscheidung (und einer Wertetabelle) stets zum Ziel kommen kann[16] So können wir etwa `xor` oder `nand` definieren als $\lambda ab.a(b\ \texttt{false}\ \texttt{true})(b\ \texttt{true}\ \texttt{false})$ beziehungsweise als $\lambda ab.a(b\ \texttt{false}\ \texttt{true})(b\ \texttt{true}\ \texttt{false})$. Wer's gerne umständlich mag, für den sei angemerkt, daß das obige Beispiel `nand` ja ohnehin schon das generische ist, so daß wir eigentlich nicht weiter hätten argumentieren müssen.

Übung 3.1. *Zeige, daß sich alle booleschen Funktionen aus* `nand` *explizit definieren lassen.*

Hinweis. Zeige zunächst, unter der Verwendung der Idee von Wahrheitstafeln, daß sich jede boolesche Funktion durch `and`, `or` und `not` ausdrücken läßt. Drücke sodann diese Funktionen durch `nand` aus. □

Als nächstes nehmen wir die natürlichen Zahlen in Angriff. Wie war das doch gleich? Sie sind aus "Null" und "Nachfolger" aufgebaut. Denken wir uns einfach mal, daß wir die schon haben (hier als `Zero` und `Suc` bezeichnet), so gilt[17]

$$
\begin{aligned}
0 &= \texttt{Zero} \\
1 &= \texttt{Suc Zero} \\
2 &= \texttt{Suc}\ (\ \texttt{Suc Zero}\ ) \\
&\vdots \\
n &= \texttt{Suc}\ (n-1) = \underbrace{\texttt{Suc}\ (\ \texttt{Suc}\ (\ldots(\ \texttt{Suc}}_{n \text{ mal}}\ \texttt{Zero})))
\end{aligned}
$$

Haben wir jetzt ein Problem weil wir `Zero` und `Suc` nicht haben? Abstrahieren wir dieses Problem doch einfach weg! Der Satz "Denken wir uns einfach mal,..." gibt schon den richtigen Hinweis. Wir definieren also ganz frech nach dem Motto

[16] Das dies nicht optimal ist, ist eine andere Geschichte. So weiß man etwa bereits, daß "a oder b" wahr ist, falls a wahr ist, unabhängig, welchen Wert b hat. Diese Information kann man etwa nutzen, um der nichtterminierenden Berechnung von b zu entkommen. Doch diese nicht-strikten Funktionen sind ein anderes Thema, das uns im Kaptiel 7 über `haskell` wieder begegnen wird.

[17] Jede Struktur, für das diese Regeln gelten, kann als Realisierung der natürlichen Zahlen angesehen werden. Es macht in manchen Situationen durchaus auch Sinn, nicht-treue Realisierungen der natürlichen Zahlen zu betrachten, d.h. Strukturen , für die wir einen "Anfang" und einen "Nachfolger" definieren können, in denen aber verschiedene natürliche Zahlen durch denselben Wert dargestellt werden können. Beispielsweise kann der Nulltest als nicht-treue Realisierung der natürlichen Zahlen angesehen werden, in der das Anfangselement durch "wahr" dargestellt wird, und jede andere Zahl durch "falsch".

"Gib mir den Nachfolger und die Null und ich geb Dir die Zahl"

$$\begin{aligned}
\underline{0} &= \lambda sz.z \\
\underline{1} &= \lambda sz.sz \\
\underline{2} &= \lambda sz.s(sz) \\
\underline{3} &= \lambda sz.s(s(sz)) \\
&\vdots \\
\underline{n} &= \lambda sz.s^n(z) = \lambda sz.\underbrace{s(s(\ldots(s}_{m \text{ mal}}\,z)))
\end{aligned}$$

Damit haben wir dann auch eine Null, nämlich $\underline{0}$ und ein Nachfolger ist auch schnell gefunden,[18] nämlich $\lambda nsz.s(nsz)$.

Übung 3.2. *Zeige daß stets* $(\lambda nsz.s(nsz))\underline{n} = \underline{n+1}$ *gilt.*

Schön und gut, aber können wir mit diesen Codes für natürliche Zahlen denn auch etwas anfangen? Im Sinne des "Animierens von Daten": Welche Aufgaben erfüllen diese natürlichen Zahlen (die üblicherweise "Churche Numerale" genannt werden)? Nun, es ist

$$\underline{n}fa = f^n(a)$$

Dies können wir uns als `for`-Schleife vorstellen, wenn wir a als den Anfangswert, also den Zustand vor Eintritt in die Schleife und f als Schritt, also als die Funktion vorstellen, die die Zustandsänderung innerhalb eines Schleifendurchlaufs beschreibt. Als "Rechenregel" zusammengefasst

$$\begin{aligned}
\underline{0}fa &= a \\
\underline{n+1}fa &= f(\underline{n}fa)
\end{aligned}$$

Als erstes Beispiel definieren wir uns einen "Nulltest", also ein Prädikat, daß angibt, ob das Argument gerade $\underline{0}$ ist. Wenn wir also ein Zahl haben und so oft eine Schleife durchlaufen dürfen, wie würden wir uns also anstellen? Naheliegend ist es, den Anfangswert auf `true` zu setzen und, falls wir den Schleifenrumpf überhaupt durchlaufen, im Schrittfall den vorherigen Wert einfach wegzuwerfen und auf `false` zu setzen: wenn wir in den Schleifenrumpf reinkommen, dann wurde die Schleife mindestens einmal durchlaufen! Formal aufgeschrieben ist das gerade

$$\lambda n.n(K\ \texttt{false}\ )\ \texttt{true}$$

und in der Tat ist

$$\begin{aligned}
\underline{0}(K\ \texttt{false}\ )\ \texttt{true}\ &=\ \texttt{true} \\
\underline{n+1}(K\ \texttt{false}\ )\ \texttt{true}\ &= K\ \texttt{false}\ (\underline{n}(K\ \texttt{false}\ )\ \texttt{true}\ ) =\ \texttt{false}
\end{aligned}$$

[18] Man beachte, dass wir hier auf sehr trickreiche Weise Wunschdenken als Konstruktionsprinzip verwenden.

Ein wenig Arithmetik wär schon ganz gut. Fangen wir doch einfach mal mit der Addition an. n und m könnten wir vermöge `for`-Schleifen so implementieren, daß wir mit m Anfangen und n-mal den Nachfolger bilden. Wir können das ganze auch etwas mehr "internalisieren". Nehmen wir uns also zwei Zahlen "$\lambda nm.\ldots$" und liefern eine Zahl ab; dazu nehmen wir unsere hypothetischen Nachfolger und Null "$\ldots\lambda sz.\ldots$". Wir iterieren n mal den hypothetischen Nachfolger "$\ldots ns\ldots$" und zwar beginnend mit der Darstellung von m aus s und z: "$\ldots(msz)$". Macht zusammen

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

Für die Multiplikation stellen wir uns genauso an "$\lambda nmsz.\ldots$" und iterieren n mal das "m-fache Nachfolger bilden", also "$\ldots n(ms)\ldots$" und zwar beginnend mit Null "$\ldots z$". Macht zusammen

$$\lambda nmsz.n(ms)z = \lambda nms.n(ms) = B$$

die Komposition!

Wie exponentieren wir? Da wir gesehen haben, daß Multiplikation gerade Komposition ist müssen wir uns also fragen, wie wir eine Zahl gerade k mal mit sich selbst komponieren. Erinnern wir uns, daß $\underline{k}fa = \underbrace{f(f(\ldots(f}_{k\text{ mal}}a)))$, so sehen wir $\underline{k}$ dies gerade leistet! Es ist $\underline{k}f = \underbrace{f\circ\ldots\circ f}_{k\text{ mal}}$, also ist

$$\underline{k}\underline{n} = \underline{n^k}$$

Das heißt unsere Zahlen übernehmen gerade die Aufgabe des exponentierens. Insbesondere ist

$$\mathtt{sqr} = \underline{2}$$

Wie in der Überschrift versprochen: unsere Zahlen sind ganz schön selbständig!

Übung 3.3. *Man beweise (formal), daß die angegebenen Rechenoperationen, also Addition, Multiplikation und Exponentation die gewünschten Eigenschaften haben. Außerdem berechne man "von Hand"* $1\cdot 1$ *und* $\underline{3}\,\underline{2}$.

Damit haben wir schon einen schönen Satz an grundlegenden Daten. Aber irgendwie müssen wir die noch zu komplexeren Datenstrukturen zusammenfassen können. Der zentrale Begriff dazu ist der des (geordneten) Paares. Damit können wir dann alles darstellen: Listen sind Paare aus dem ersten Element der Liste und dem Rest, Bäume sind Paare aus linkem und rechtem Teilbaum, Tripel sind sind Paare aus einem Element und einem Paar,... [19]

[19] Pedantischer Weise müßte man noch sagen wie man die leere Liste, den leeren Baum und so weiter darstellt. Aber der Leser dürfte inzwischen genug Erfahrung haben um sich das selbst zusammen zu stricken. Wenn einem überhaupt nichts dümmeres einfällt codiert man es eben als ein Paar aus einem booleschen Wert, der angibt ob es sich um die leere Liste oder nicht handelt und den sonst noch benötigten Daten.

Um ein Paar zu definieren überlegen wir uns wieder, was wir von ihm wollen. Das mindeste dürfte sein, daß man daraus die beiden Komponenten ablesen können. Setzen wir doch ganz naiv an und definieren das Paar von a und b als

$$\lambda z.\ \texttt{if}\ z\ \texttt{then}\ a\ \texttt{else}\ b$$

Das sieht doch ganz vielversprechend aus: Wenn wir dem Paar den Wert "wahr" übergeben erhalten wir die linke Komponente und bei "falsch" die rechte. Und der Leser dürfte sich inzwischen auch überlegt haben, welcher Kombinator zur Bildung des Paares verwendet wird... ja, richtig... es ist... der Kombinator $P = \lambda xyz.zxy$.

Bevor wir uns jetzt entspannt zurücklegen und sagen, daß wir alle Probleme gelöst haben sollten wir noch feststellen, daß unsere Paare eigentlich viel besser sind, als es auf den ersten Blick scheint! Wer legt uns denn darauf fest, daß wir einem Paar nur einen Wahrheitswert übergeben? Niemand. Und die "übliche" und "richtige" Art mit einem Paar zu arbeiten ist auch eine andere. Ein Paar kann man nämlich aus lesen als "Ich hab' die beiden Komponenten, sag mir, was ich damit tun soll!". Dementsprechend nehmen wir einfach an, wir hätten die beiden die beiden Komponenten des Paares schon ($\lambda ab.\ldots$) und konstruieren daraus dann ein Ergebnis. Diese Konstruktion ist dann also von der Form $p(\lambda ab.\ldots)$, wobei p das Paar ist. Das mag vielleicht etwas gewöhnungsbedürftig sein, scheinbar "die Kontrolle aus der Hand zu geben" und das Paar agieren zu lassen. Wenn man aber mal gelernt hat seinen Paaren (und sonstigen Datenstrukturen) zu vertrauen ist dies die natürlichste Sache der Welt.

Abschließend noch eine Übungsaufgabe. Ihr Auftreten im Text (denn inhaltlich geht es noch mal um natürliche Zahlen!) mag schon einen Hinweis geben, welches Konstrukt vielleicht hilfreich sein könnte...

Übung 3.4. *Man definiere einen Vorgänger, das heißt einen Kombinator* `pred` *für den für alle natürlichen Zahlen* n *gilt* `pred` $\underline{n+1} = \underline{n}$.

4 Konfluenz: Warum die Welt nicht in einem Punkt kollabiert

Diese Thematik wurde im Kurs nur kurz angedeutet. Ziel ist zu zeigen, daß die Theorie nicht trivial ist, in dem Sinne, daß nicht alle Terme gleich sind. Für eine genauere Analyse, welche Terme noch alle gleich gemacht werden können ohne, daß die Theorie zusammenbricht sei auf die Literatur verwiesen, siehe etwa den Handbuchartikel von Barendregt [1].

Dieses Kapitel basiert auf Resultaten von Takahashi [4] und von Joachimski und Matthes [3]. Die Darstellung lehnt sich an diejenige von Ralph Matthes' Skript über den λ-Kalkül aus dem Sommersemester 2000 an.

Erinnern wir uns, wie wir (informell) Gleichheit definiert hatten.

Definition 4.1 ($=_\beta$)**.** *Es sei $=_\beta$ die kleinste[20] Äquivalenzrelation[21] auf den λ-Termen die folgende Eigenschaften besitzt:*

- $(\lambda x.r)s =_\beta r[x := s]$
- $r =_\beta r' \Rightarrow \lambda x.r =_\beta \lambda x.r'$
- $r =_\beta r',\ s =_\beta s' \Rightarrow rs =_\beta r's'$

Damit hatten wir eine Menge Gleichungen herleiten können. So hatten wir etwa (in Kapitel 3) gesehe, wie man ein Paar bildet und wie man daraus die Komponenten zurückgewinnt. Eine Frage, die wir bisher noch überhaupt nicht behandelt hatten ist, ob es denn auch Dinge gibt, die verschieden sind. Oder ist es vielleicht so, daß wir aus obigen drei Eigenschaften für beliebige Terme herleiten können, daß sie gleich sind? Daß dies vielleicht doch nicht so absurd ist, wie es zunächst klingt zeigt folgendes

Beispiel 4.2. *Ist $\underline{0} =_\beta \underline{1}$, so sind alle Terme gleich.*

Beweis. Mit $\underline{0} =_\beta \underline{1}$ muß auch $\underline{0}(\lambda x.r)s =_\beta \underline{1}(\lambda x.r)s$ für beliebige r und s gelten (mit x "neu"). Wegen $\underline{0}(\lambda x.r)s =_\beta s$ und $\underline{1}(\lambda x.r)s =_\beta (\lambda x.r)s =_\beta s$ ergibt sich die Behauptung. □

[20] "Kleinste" ist im Sinne einer *induktiven* Definition zu verstehen. $=_\beta$ möge also nur gelten, wenn es aus den angegebenen Eigenschaften herleitbar ist.

[21] Eine binäre Relation heißt *Äquivalenzrelation*, wenn sie symmetrisch, reflexiv und transitiv ist. Zusätzlich zu den angegebenen Eigenschaften fordern wir also noch daß

- $r =_\beta s \Rightarrow s =_\beta r$
- $r =_\beta r$
- $r =_\beta r',\ r' =_\beta r'' \Rightarrow r =_\beta r''$

Man überlege sich ein Beispiel das zeigt, daß zumindest Reflexivität notwendig sein kann, um überhaupt "vom Fleck" zu kommen!

Übung 4.3. *Zeige daß bereits dann alle Terme gleich sind, wenn irgend zwei (verschiedene) Churche Numerale gleich sind.*

Hinweis. Obiges Argument greift offenbar (warum?) für alle Gleichungen der Form $\underline{0} =_\beta \underline{n+1}$. Außerdem hatten wir (in Übung 3.4) gesehen, wie man einen Vorgänger definiert. □

Betrachten wir Beispiel 4.2 etwas genauer, so stellen wir fest, daß das, was gegen die Intuition läuft gerade das "Expandieren" von Termen ist, bei denen in unkontrollierte Weise neue Terme auftauchen. Unser "offizielles" Argument begann ja mit $s =_\beta \underline{0}(\lambda x.r)s =_\beta \underline{1} \ldots =_\beta r$. Wäre es nicht schön, könnten wir auf derartige Expansionen verzichten? Gefühlsmäßig würde man ja eine Gleichung dadurch überprüfen, daß man beide Seiten "so weit wie möglich" vereinfacht und anschließend nachsieht ob die beiden Seiten nun (syntaktisch) gleich aussehen.

Wir werden sehen, daß dieses Vorgehen tatsächlich auch immer möglich ist, doch zunächst wollen wir genauer fassen, was wir unter "Vereinfachen" verstehen wollen. Das Einsetzen eines Arguments in eine explizite Funktionsdefinition war die Vereinfachung von der wir ursprünglich ausgegangen waren, also ist die rechte Seite in der ersten Gleichung von Definition 4.1 sicher "einfacher" als die linke. Die Umformung von $r[x := s]$ zu $(\lambda x.r)s$ im Falle, daß x in r gar nicht vorkommt, war ja gerade die Ursache für das Auftreten unbekannter Terme. Wir "richten" diese Gleichung also von links nach rechts. Außerdem sind wir vorsichtig und wollen nur eine Reduktion auf einmal vornehmen. Also definieren wir:

Definition 4.4 ($\to_\beta$). *Es sei $\to_\beta$ die kleinste Relation auf den λ-Termen die folgende Eigenschaften besitzt:*

- $(\lambda x.r)s \to_\beta r[x := s]$
- $r \to_\beta r' \;\Rightarrow \lambda x.r \to_\beta \lambda x.r'$
- $r \to_\beta r' \;\Rightarrow rs \to_\beta r's,\; sr \to_\beta sr'$

Da eine einzelne Umformung oft nicht reicht erweist sich folgende Schreibweise als nützlich

Definition 4.5. *Für eine Relation $\rhd$ bezeichne $\rhd^*$ die reflexiv-transitive Hülle von $\rhd$. $r \rhd^* s$ gilt also genau dann, wenn es eine endliche (möglicherweise leere) Folge $r_1, \ldots, r_n$ gibt mit $r = r_1 \rhd r_2 \rhd \ldots \rhd r_n = s$.*

Außerdem schreiben wir $s\,{}_\beta\!\leftarrow r$ bzw. $s\,{}^*_\beta\!\leftarrow r$, falls $r \to_\beta s$ bzw. $r \to^*_\beta s$.

Da $\to_\beta$ aus $=_\beta$ "im wesentlichen" durch richten einer Gleichung entstanden ist, müssen wir die "andere Richtung" noch hinzunehmen, um auf den ursprünglichen Gleichheitsbegriff zu kommen. In der Tat gilt

Proposition 4.6. $=_\beta$ *ist die äquivalente Hülle von* $\to_\beta$ *; mit anderen Worten, es gilt* $({}_\beta\!\leftarrow \cup \to_\beta)^* \quad = \quad =_\beta$ *.*

Beweisskizze. Wir müssen die Gleichheit von zwei Relationen zeigen, zeigen also beide Inklusionen: Daß $({}_\beta\!\leftarrow \cup \to_\beta)^*$ in $=_\beta$ enthalten ist zeigen wir durch "Induktion über $({}_\beta\!\leftarrow \cup \to_\beta)^*$"; genauer zeigen wir zunächst durch Induktion über $r \to_\beta s$, daß aus $r \to_\beta s$ folgt $r =_\beta s$, anschließend, daß dies schon aus $r\ ({}_\beta\!\leftarrow \cup \to_\beta)\ s$ folgt und schließlich die Behauptung. Um zu sehen $=_\beta$ in $({}_\beta\!\leftarrow \cup \to_\beta)^*$ enthalten ist, erinnern wir uns, daß $=_\beta$ die kleinste Äquivalenzrelation mit gewissen Eigenschaften ist. Wir müssen also zeigen, daß $({}_\beta\!\leftarrow \cup \to_\beta)^*$ eine Äquivalenzrelation mit den geforderten Eigenschaften ist, was nicht schwer ist. □

Übung 4.7. *Führe den Bewis im Detail durch!*

Sind wir unserem Ziel damit näher gekommen? Wir wollten erreichen, daß Gleichheit schon durch "Vereinfachen beider Seiten" getestet werden kann. Das mindeste, was dazu nötig ist, ist wir Terme r' und r'' wieder zusammenführen können, die verschiedene "Vereinfachungen" des selben Terms r sind. Diese Eigenschaft, die man "Konfluenz" nennt genügt auch.

Definition 4.8 ($\rhd^*\Diamond$). *Eine Relation* $\rhd$ *heißt konfluent (in Zeichen* $\rhd^*\Diamond$*), wenn es für* r, r', r'' *mit* $r' \, {}^*\!\lhd \; r \; \rhd^* \; r''$ *stets ein* r''' *mit* $r' \; \rhd^* \; r''' \, {}^*\!\lhd \; r''$ *gibt.*

Lemma 4.9 (Church-Rosser Eigenschaft). *Wenn* $\to_\beta$ *konfluent ist, so haben* $=_\beta$*-gleiche Terme ein gemeinsames* $\to_\beta^*$*-Redukt.*

Übung 4.10. *Beweise dies!*

Hinweis. Man verwende die Charakterisierung von Proposition 4.6. □

Beispiel 4.11. *Ist* $\to_\beta$ *konfluent, so ist* $\underline{0} \neq_\beta \underline{1}$*.*

Beweis. Nach Lemma 4.9 müßten $\underline{0}$ und $\underline{1}$ ein gemeinsames Redukt haben. Beide Terme sind aber normal, d.h. sie haben keine (echten) Redukte. Außerdem sind sie offenbar syntaktisch verschieden. □

Man beachte, daß Konfluenz eine Eigenschaft der transitiven Hülle ist[22]. Da bei der β-Reduktion Teilterme (und damit möglicherweise auch Redexe) verdoppelt werden, kann man im Allgemeinen nicht mehr erwarten.

Beispiel 4.12 (Diagonalabbildung). *Betrachte die Diagonalabbildung* $\lambda xz.zxx$*, die ihr Argument in die beiden Komponenten eines Paares einsetzt. Gelte nun* $s \to_\beta s'$*. Dann ist zwar* $(\lambda xz.zxx)s' \;{}_\beta\!\leftarrow\; (\lambda xz.zxx)s \;\to_\beta\; (\lambda z.ss)$*, aber es sind* zwei *Reduktionen nötig um* $(\lambda z.ss)$ *auf das gemeinsame Redukt* $(\lambda z.s's')$ *zu bringen*

[22] Dies wird auch durch die Schreibweise suggeriert. Allgemeiner sagt man, daß eine Relation $\rhd$ die $\Diamond$-Eigenschaft hat (in Zeichen $\rhd\Diamond$), wenn es für r, r', r'' mit $r' \lhd r \rhd r''$ stets ein r''' mit $r' \rhd r''' \lhd r''$ gibt.

Die entscheidende Technik um Konfluenz zu zeigen ist es nun, einen "parallelen" Reduktionsbegriff zu definieren, der zwischen der Einschrittreduktion und deren transitiven Hülle liegt und "alle möglichen Redexe" parallel umdreht, insbesondere etwa auch in den beiden Komponenten eines Paares gleichzeitig arbeiten kann. Damit kann dann in einem parallelen Schritt zusammengeführt werden, was in einem (parallen) Schritt auseinandergeführt wurde.

Definition 4.13 ($\twoheadrightarrow$). *Die Relation $\twoheadrightarrow$ wird induktiv definiert durch*

- $x \twoheadrightarrow x$
- $r \twoheadrightarrow r' \;\Rightarrow \lambda x.r \twoheadrightarrow \lambda x.r'$
- $r \twoheadrightarrow r',\ s \twoheadrightarrow s' \;\Rightarrow rs \twoheadrightarrow r's'$
- $r \twoheadrightarrow \lambda x.t,\ s \twoheadrightarrow s' \;\Rightarrow rs \twoheadrightarrow t[x := s']$

Plakativ ausgedrückt läßt sich also sagen, daß parallele Reduktion nichts anderes ist als Reduktion in Teiltermen möglicherweise gefolgt von einer Konversion. Die parallele Reduktion hat gewisse offensichtliche Eigenschaften:

Proposition 4.14. $\twoheadrightarrow$ *ist reflexiv und* $\rightarrow_\beta \subset \twoheadrightarrow \subset \rightarrow_\beta^*$.

Beweis(skizze). Zeige $r \twoheadrightarrow r$ durch Induktion nach r. Zeige $\rightarrow_\beta \subset \twoheadrightarrow$ durch Induktion nach $\rightarrow_\beta$ und $\twoheadrightarrow \subset \rightarrow_\beta^*$ durch Induktion nach $\twoheadrightarrow$. □

Darüberhinaus ist die parallele Reduktion wirklich parallel, in dem Sinne, daß sie die Verdoppelungen durch eine Substitution schluckt. Mehr noch: es darf sogar noch der Term reduziert werden, in dem reduziert wird.

Lemma 4.15 (Substitutionseigenschaft). $r \twoheadrightarrow r',\ s \twoheadrightarrow s' \;\Rightarrow r[x := s] \twoheadrightarrow r'[x := s']$

Beweis. Wir argumentieren durch Induktion nach $r \twoheadrightarrow r'$. Falls $y \twoheadrightarrow y$ so folgt die Behauptung aus $s \twoheadrightarrow s'$ oder $y \twoheadrightarrow y$, je nach dem x und y gleich oder verschieden sind. Die Fälle $rt \twoheadrightarrow r't'$ und $\lambda x.r \twoheadrightarrow \lambda x.r'$ ergeben sich jeweils aus den entsprechenden Induktionsvoraussetzungen.

Sei also $rt \twoheadrightarrow r'[y := t']$ dank $r \twoheadrightarrow \lambda y.r'$ und $t \twoheadrightarrow t'$. Nach Induktionsvoraussetzung haben wir $r[x := s] \twoheadrightarrow \lambda y.r'[x := s']$ und $t[x := s] \twoheadrightarrow t'[x := s']$. Also $r[x := s]t[x := s] \twoheadrightarrow r'[x := s'][y := t'[x := s']]$. Wegen $r[x := s]t[x := s] = (rt)[x := s]$ und $r'[x := s'][y := t'[x := s']] = r'[y := t'][x := s']$ ergibt sich die Behauptung. □

Wie weit kommen wir also mit so einer parallelen Reduktion? Nun, wir können in Teiltermen reduzieren und eventuell noch *anschließend* eine Konversion durchführen. Um alle Eventualitäten vorzusehen sollten wir alle diese auch Möglichkeiten nutzen:

Definition 4.16 (Superentwicklung t^β). *Definiere t^β durch Induktion über t.*

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

Theorem 4.17 (Maximalität von $\cdot^\beta$ bezüglich $\twoheadrightarrow$). $r \twoheadrightarrow r' \Rightarrow r' \twoheadrightarrow r^\beta$

Beweis. Wir argumentieren durch Induktion über $r \twoheadrightarrow r'$. Der entscheidende Fall ist der, daß $rs \twoheadrightarrow t[x := s']$ auf Grund von $r \twoheadrightarrow \lambda x.t$ und $s \twoheadrightarrow s'$. Nach Induktionsvoraussetzung ist $s' \twoheadrightarrow s^\beta$ und $\lambda x.t \twoheadrightarrow r^\beta$. Inspektion der Definition von $\twoheadrightarrow$ ergibt, daß r^β von der Form $r^\beta = \lambda x.t'$ mit $t \twoheadrightarrow t'$. Mit Lemma 4.15 ergibt sich $t[x := s'] \twoheadrightarrow t'[x := s^\beta]$. Da die rechte Seite nach Definiton(!) gerade $(rs)^\beta$ ist, ergibt sich die Behauptung. □

Übung 4.18. *Führe den Rest des Beweises aus!*

Damit haben wir erreicht, was wir wollten, denn hieraus folgt leicht die Konfluenz.

Übung 4.19 ($\rightarrow_\beta^* \diamond$). *Wie?*

Wir haben aber sogar noch mehr gezeigt. Wir wissen nicht nur, daß es einen Term gibt, zu dem wir auseinanderlaufende Terme wieder zusammenführen können, wir können ihn sogar angeben: Wenn wir uns von einem Term höchstens n (sogar parallele) Schritte wegbewegen, dann müßen wir ihn nur n-mal entwickeln um ein gemeinsames Redukt zu haben:

Lemma 4.20 (Simulation). $r \twoheadrightarrow s \Rightarrow r^\beta \twoheadrightarrow s^\beta$

Beweis. Wegen der Maximalitätseigenschaft folgt aus $r \twoheadrightarrow s$ daß $s \twoheadrightarrow r^\beta$, weiter folgt, wieder aus der Maximalitätseigenschaft daß $r^\beta \twoheadrightarrow s^\beta$. □

Lemma 4.21. $r \twoheadrightarrow^n s \Rightarrow s \twoheadrightarrow^n r^{\overbrace{\beta \ldots \beta}^{n}}$

Beweis. Induktion über n. Falls $r \twoheadrightarrow^n r' \twoheadrightarrow s$, dann $s \twoheadrightarrow r'^\beta$ nach Induktionsvoraussetzung. Außerdem $r^\beta \twoheadrightarrow^n r'^\beta$ nach Simulation. Also $r'^\beta \twoheadrightarrow^n r'^{\beta\overbrace{\beta \ldots \beta}^{n}}$ nach Induktionsvoraussetzung. □

5 Θ: Das Fixpunktprinzip und sein Zeuge

Offenbar scheint die Theorie der Kombinatoren, die ja selbst Funktionen sind, die Kombinatoren auf Kombinatoren abbilden, Sinn zu machen. Weil die Kombinatoren die ganzen Zahlen umfassen, gibt es sicher unendlich viele echt verschiedene davon[23] Weil wir aber jeden Kombinator, wie wir in Kapitel 2 gesehen haben, als SK-Term schreiben können, sind sie abzählbar. Die Menge der Kombinatoren ist damit genauso mächtig wie die natürlichen Zahlen. Es gibt nun Abbildungen zwischen Kombinatoren, die selbst nicht als Kombinator geschrieben werden können. Der Grund ist ganz einfach: wenn wir allein nur die Abbildungen betrachten, die jeden Kombinator auf entweder S oder K abbilden, so gibt es davon genauso viele wie Teilmengen der natürlichen Zahlen. (Jede solche Abbildung kann ja gerade durch Angabe der Teilmenge der Kombinatoren, die auf K abgebildet werden, charakterisiert werden.) Das sind aber überabzählbar viele. Von den denkbaren Abbildungen von Kombinatoren auf Kombinatoren stellen die Kombinatoren selber nur einen verschwindend kleinen Bruchteil dar. Das legt die Vermutung nahe, daß sie einige sehr spezielle Eigenschaften haben, die man von allgemeinen Kombinator-Abbildungen nicht erwarten würde. Das ist in der Tat der Fall. Eine erstaunliche Tatsache ist beispielsweise, daß es für jeden Kombinator F einen Kombinator X gibt mit $FX = X$.

Hierfür wollen wir nun einen Beweis konstruieren und zwar langsam, Schritt für Schritt.

- Wenn wir zu gegebenem F ein X finden wollen, so daß $FX = X$, ist es vielleicht ein hoffnungsvoller Ansatz, anzunehmen, daß es einen 'Orakelkombinator' O gibt, der F auf ein solches X abbildet. Anders gesagt: $F(OF) = OF$.

- Der Versuch, an FX herumzureduzieren, um X zu erhalten, führt in eine Sackgasse, weil wir über den allgemein gehaltenen Kombinator F nichts wissen. Aber vielleicht können wir ja X abhängig von F so bauen, daß es sich zu FX reduzieren läßt. Anders ausgedrückt, die Idee ist, daß FX "einfacher" ist als X; soviel zu der intuitiven Idee, daß vereinfachen was mit "einfach" zu tun hat...

- Für einen beliebigen Kombinator Q können wir einen Kombinator R finden, der $RF = F(QF)$ erfüllt: $R = \lambda f.f(Qf)$. Das Problem ist nur, daß wir gerne $Q = R = O$ hätten. Irgendwie muß also O seinen eigenen Bauplan enthalten.

- Bisweilen hört man die argwöhnische Vermutung, daß das gar nicht gehen könne. Allerdings: jedes Lebewesen belehrt uns hier eines Besseren. Jeder von uns enthält seinen kompletten Bauplan, in Form des Genotyps. Wir

[23] Im Gegensatz zum Kurs, wo die Teilnehmer dies glauben mußten ist dies hier sogar gesichert. Dies ist das Hauptergebnis von Kapitel 4.

versuchen also uns hier was von der Natur abzugucken und einen Kombinator zu konstruieren, der aus einem gegebenen "Genotyp" (in Form eines Kombinators) einen Kombinator baut, der gerade die Umsetzung des Genotyps in ein Lebewesen ausdrückt. Da Kombinatoren herrlich abstrakt sind, können wir als Bauplan für die Maschine gleich die Maschine selber verwenden. Die Abbildung (Bauplan $\mapsto$ Maschine Bauplan) können wir dann mit "Bauplan = Maschine" schreiben als $\lambda x.xx$. In der Tat: wenn wir diesen Kombinator als Maschine auffassen und auf sich selbst als Bauplan anwenden, erhalten wir $(\lambda x.xx)(\lambda x.xx)$. Wenn wir das ausreduzieren, gibt das gerade wieder $(\lambda x.xx)(\lambda x.xx)$. Wir haben also in der Tat hier einen Kombinator, der seinen eigenen Bauplan enthält. Die üblichen Bezeichnungen sind

$$\begin{aligned}\omega &= \lambda x.xx \\ \Omega &= \omega\omega \;\to_\beta\; \omega\omega \;\to_\beta\; \dots\end{aligned}$$

- Sehen wir noch einmal genau hin: $OF = F(OF)$. Wir brauchen eine Maschine m, die mit einem Bauplan b für eine Maschine gefüttert werden muß, sowie einem Kombinator f, und als Ergebnis den Wert der Anwendung von f auf mbf liefert: $mbf = f(mbf)$. Die Maschine auf der rechten Seite soll gerade gleich dem Bauplan sein: $mbf = f(bbf)$, also $m = $ $.\mathbf{f}(\mathbf{bbf})$. Nun ist O wie beschrieben gerade das Ergebnis der Anwendung des Bauplans auf die Maschine $O = mb$, aber Bauplan=Maschine! Also: $O = mm = \lambda bf.f(bbf)(\lambda bf.f(bbf)) = \lambda f.f((\lambda bf.f(bbf))(\lambda bf.f(bbf))f)$. In der Tat haben wir damit $OF = F(OF)$.

Der wesentliche Trick war, Bauplan=Maschine stufenweise zweimal hintereinander anzuwenden. Es lohnt sich, länger darüber zu meditieren.

Ein "Orakelkombinator" wie der, den wir zuvor konstruiert haben, wird Fixpunktkombinator genannt. Die Standardbezeichnungen für obige Kombinatoren sind:

$$\begin{aligned}T &= \lambda xy.y(xxy) \\ \Theta &= TT\end{aligned}$$

Damit haben wir sogar mehr erreicht, als wir ursprünglich wollten! Wir wissen jetzt nicht nur daß es zu jedem Kombinator einen Fixpunkt gibt, sondern haben auch ein uniformes Verfahren, den Fixpunkt zu finden. Mehr noch, das Verfahren ist sogar so uniform, daß es innerhalb des Systems bezeugt wird, etwa durch Θ. Dies ermöglicht auch die knappe Darstellung die man in den meisten Mathematikbüchern findet; gemäß dem Motto "kurz, präzise und unverständlich"[24] heißt es dort typischerweise

[24] Man mag darüber streiten, ob das jetzt ein Vor- oder ein Nachteil ist. Verständnis wird sicher weniger geweckt, aber man muß deutlich weniger Text lesen, um dieses Prinzip nachzuvollziehen. In diesem Sinne ist es effizienter und durch die Kürze vielleicht auch weniger abschreckend. Die Autoren würden sich über Kommentare hierzu freuen.

Fixpunktprinzip: Sei $T = \lambda xy.y(xxy)$ und $\Theta = TT$. Dann gilt für beliebiges f gerade $\Theta f = TTf = f(TTf) = f(\Theta f)$. Insbesondere gilt $\forall f \exists x. fx = x$, da Θf ein mögliches x ist.

Es bleibt noch anzumerken, daß Θ der weniger bekannte Fixpunktkombinator ist. Viel beliebter ist

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Übung 5.1. *Zeige, daß Y ein Fixpunktkombinator ist, daß also $Yf = f(Yf)$ für beliebiges f gilt.*

Als erste Anwendung überlegen wir uns, daß wir nun eine Funktion von Kombinatoren in die Kombinatoren explizit angeben können, die *nicht* durch einen Kombinator dargestellt werden kann.

Übung 5.2. *Zeige, daß die Abbildung, die jeden Kombinator außer K auf K abbildet, K selbst jedoch auf S abbildet im λ-Kalkül nicht definiert werden kann.*

Hinweis:. Angenommen R ist so ein Kombinator. Dann ist YR ein Fixpunkt. Gilt $YR = K$? (Stichwort "**R**ussel") $\square$

6 Zwischenspiel: Ein Übungsblatt

Da man den Umgang mit dem λ-Kalkül nur durch praktisches Üben erlernt (und um zu unterstreichen, daß es sich um einen Mathematikkurs handelte) wurde ein Teil der zweiten Kurseinheit verwendet, um den Teilnehmern zu ermöglichen, in kleinen Gruppen, unter Anleitung der Kursleiter sich mit folgendem Übungsblatt zu beschäftigen.

Das beeindruckende dabei war, daß nicht wenige Teilnehmer einen Teil ihrer (auf diesem Wochenende sehr knapp bemessenen) Freizeit darauf verwandten um sich freiwillig *intensiver mit den Üngen und dem Kurs auseinanderzusetzen. Unsere Hochachtung!*

Selbstverständlich sind alle Teilnehmer eingeladen auch nach Abschluß des Kurses noch Lösungsvorschläge zu Übungen (per email) abzugeben, die selbstverständlich ausführlich korrigiert werden.

Übung 6.1. *Man zeige, da $SKK = I$.*

Diese Aufgabe kann auf zwei Weisen verstanden werden. Einerseits kann man unsere Definitionen von S, K und I einsetzen und nur mit der β-Regel rechnen. Lehrreicher ist jedoch folgende Variante: Man zeige unter verwendung der "Rechenregeln" für die betreffenden Kombinatoren $SKKx = Ix$. (Welche Rolle spielt dabei x und welche das zweite K?)

Übung 6.2. *Man betrachte folgende Rechnung:*

$$\begin{aligned} BB &= (\lambda xyz.x(yz))(\lambda abc.a(bc)) \\ &= (\lambda yz.(\lambda abc.a(bc))(yz)))) \\ &= (\lambda yz.(\lambda bc.(yz)(bc))) \\ &= (\lambda pqrs.pq(rs)) \end{aligned}$$

Man berechne analog BBB In die andere Richtung versuche man, den Kombinator $\lambda abcde.ab(cde)$ allein durch B auszudrcken.

Übung 6.3. *Man drücke B durch S und K aus.*

Übung 6.4. *Sei $B = \lambda xyz.x(yz)$, $D = \lambda xy.yx$. Man leite (nur) unter Zufilfename von B und D einen Ausdruck fur $\lambda abc.a(cb)$ her.*

Übung 6.5. *Gegeben sei eine unter Funktionsanwendung abgeschlossene Menge von Kombinatoren, die $\omega = \lambda x.xx$ und $B = \lambda xyz.x(yz)$ enthält. Man zeige, daß es zu jedem Kombinator F ein X gibt derart, daß $FX = X$.*

Übung 6.6. *Es sei $X = \lambda x.xKSK$. Man berechne XXX und $X(XX)$. Welche Konsequenzen hat dieses Ergebnis?*

Übung 6.7. *Man drücke $\lambda xy.yx$ allein durch S und K aus.*

Übung 6.8. *Man finde zwei* beweisbar verschiedene *Kombinatoren M und N derart, daß $MK = N$ und $NK = M$.*

Übung 6.9. *Man konstruiere einen Vorgängerkombinator fr die Church-Zahlen nach der im Text beschriebenen Methode.*

Und zum Abschluß noch etwas für diejenigen, die sich die Zähne ausbeißen wollen:

Übung 6.10. *Sei $L = \lambda xy.x(yy)$ gegeben. Man zeige, daß sich allein unter Verwendung von L ein Kombinator Q konstruieren läßt, für den $Q = QQ$ gilt.*

7 Realexistierende Approximationen an den λ-Kalkül: Haskell und LISP

In dieser Lektion wollen wir über konkrete praktische Anwendungen sprechen. Der λ-Kalkül ist nämlich gleich nochmal so toll, wenn man damit abstrakte Gedanken zum Leben erwecken kann. Dies geschieht anhand der Programmiersprachen Haskell und LISP.

LISP und Haskell sind nicht die einzigen Sprachen, die mehr oder weniger stark vom Lambda-Kalkül beeinflußt wurden. Sogar die Sprache, die gewissermaßen Inbegriff der Objektorientierung ist - Smalltalk - enthält Features, die eine direkte Einbettung des Lambda-Kalküls ermöglichen. Darauf soll hier nicht eingegangen werden; es sei nur kurz für die Smalltalker demonstriert, was gemeint ist:

```
^(([:x | [:y | x]] value: 1) value: 2)!
```

Warum nun gerade Haskell und LISP? Haskell deswegen, weil es eine sehr direkte und zudem die heute wohl technisch am weitesten fortgeschrittene und kompromißloseste Realisierung des Lambda-Kalküls im Hinblick auf Programmierung darstellt. Zudem herrscht hier in der Community ein gewisser Konsens was die Rolle von Haskell angeht. Benannt ist Haskell nach dem Logiker Haskell Curry, der unter anderem ein sehr lesenwertes zweibändiges Werk über Kombinatoren verfaßt hat. LISP hingegen ist deswegen interessant, weil es die älteste und die am weitesten verbreitete funktionale Sprache ist. Zudem ist LISP etwas besonderes, weil es eigentlich keine profane Programmiersprache ist, sondern ursprünglich als abstrakte Notation zur Formalisierung von allgemeinen Berechnungen gedacht war. Was das bedeutet und welche Konzepte sich daraus ergeben (als Stichworte seien mal "programmierbare Programmiersprache" und "Metazirkularität" in den Raum geworfen), werden wir noch sehen.

Natürlich kann und soll dies kein Schnellkurs sein, der perfektes Programmieren in Haskell und LISP in ein paar Minuten vermittelt. Viel mehr als einen kleinen Einblick kann hier nicht gegeben werden. Wir werden in der nächsten Lektion einige Code-Beispiele sehen, die zum Teil Syntax verwenden, die nicht in allen Details erklärt wird. Man sollte sich aber dennoch an den entsprechenden Stellen halbwegs zusammenreimen können, was was bedeutet und den Code zumindest passiv verstehen können.

Fangen wir mit Haskell an - weil es sich enger am Lambda-Kalkül orientiert. (McCarthy hatte übrigens mal zugegeben, zu der Zeit, als LISP entstand, das Papier von Church über den Lambda-Kalkül noch gar nicht richtig durch und durch verstanden zu haben.) Als erstes müssen wir uns mit einer sehr allgemeinen und etwas merkwürdigen Frage beschäftigen: man kann sich verschiedene Realisierungen der Idee, mit dem Lambda-Kalkül zu programmieren, vorstellen. Eine wäre, einen Interpreter direkt mit Lambda-Ausdrücken als Zeichenketten arbeiten zu lassen und damit letztendlich so zu programmieren wie wir es zuvor schon mal gesehen haben. Eine andere Realisierung wäre ein System, in dem wir Funktionen definieren können, die sich wie Kombinatoren verhalten. (Das bedeutet insbesondere, daß Funktionen auch als Werte anderer Funktionen auftauchen können. In Sprachen wie `C` ist das im allgemeinen ohne weiteres nicht[25] [26] möglich.) Letztere Idee liegt den funktionalen Programmiersprachen zugrunde.

Damit haben wir aber auch sofort ein Problem:

wenn wir Kombinatoren als Lambda-Ausdrücke gegeben haben und "in sie reinsehen können", haben wir zumindest in manchen Fällen die Möglichkeit, durch Hinsehen oder einen kleinen Beweis die Gleichheit oder Verschiedenheit von Kombinatoren zu beweisen. (Letzteres über einen Widerspruchsbeweis, der aus der Annahme der Gleichheit zweier Kombinatoren etwa $S = K$ ableitet (was nicht möglich ist, falls es überhaupt zwei verschiedene Kombinatoren gibt.))

Wenn alle Kombinatoren einfach nur Funktionen sind, und nur als solche in Erscheinung treten, sprich: Kombinatoren auf andere Kombinatoren abbilden, haben wir das Problem, daß wir keine Möglichkeit haben, auch wirklich irgendwelche Werte zu identifizieren. Wir brauchen also in funktionalen Sprachen zwingend irgendwelche Basis-Größen, die zumindest eine eigene, überprüfbare Identität haben - meist haben sie auch noch etwas mehr Struktur, aber mindestens diese Eigenschaft muß gegeben sein. Obwohl es denkbar wäre, diese Basis-Größen durch irgendwelche Konventionen auch als Funktionen in Erscheinung treten zu lassen, macht dies wohl keine Sprache so. Das bedeutet konkret, daß es in funktionalen Sprachen Situationen gibt, in denen ein Wert keine Funktion ist, d.h. ein Wert nicht an der Stelle eines zweiten Werts ausgewertet werden kann. In der Tat ist es sogar so, daß es in Haskell *nicht* möglich ist, eine Funktion zu definieren, für die die Prozedur "Wert einsetzen und in das Ergebnis wieder einen Wert einsetzen" beliebig oft iteriert werden kann[27]. Das ist eine Beschränkung, die letztendlich auf das Typsystem von Haskell zurückgeht, das in mancherlei Hinsicht ein zweischneidiges Schwert ist.

---

[25] Dieses "nicht" ist etwas mit Vorsicht zu genießen. Natürlich ist es in `C` möglich, mit Variablendeklaration wie `struct {void *((*fun)(void *,void *)); void *context} *closure;` zu arbeiten. Aber bereits diese Deklaration zeigt, daß dies eine etwas umständliche Art funktionaler Programmierung ist und es ist auch (leider) *nicht* die Art, wie man "typischerweise" in `C` programmiert.

Trotzdem möchte einer der Kursleiter (wohl zum Mißfallen des anderen) betonen, daß funktionale Programmierung eine Frage der inneren Einstellung und nicht der Implementierungssprache ist.

[26] Anmerkung des anderen Kursleiters: der natürliche Funktionsbegriff, den eine Sprache wie C von sich aus mitbringt, ist zum einen nicht geeignet, um damit wirklich funktional zu programmieren, und zum anderen starr: er lässt sich nicht so verallgemeinern, dass beispielsweise schon vorhandene Bibliotheksfunktionen wie z.B. `qsort(3)`), die Funktionszeigerargumente erwarten, mit diesem neuen Funktionsbegriff zusammenarbeiten können. So gesehen, sind dem naheliegenden Wunsch, auch in einer Sprache wie C funktional zu programmieren, erhebliche technische Hindernisse in den Weg gelegt.

[27] Dies ist etwa anders als in `LISP`, wo es zwar auch Werte gibt, die keine Funktionen sind, man aber dem System keine Rechenschaft schuldig ist, was denn nun das Ergebnis dieser Funktion sein soll.

Eigentlich müßten bei so einem Statement sofort die Alarmglocken läuten: ein Kombinator, der mit Sicherheit die Eigenschaft hat, daß man Wert-einsetzen beliebig oft iterieren kann, ist YK. Da aber K sicher ein zahmer Kombinator ist, und Anwendung auch nichts böses machen darf, scheint das darauf hinzudeuten, daß es in Haskell nicht möglich ist, einen Fixpunktkombinator zu definieren. . . [28]

Wir können also (obwohl dem nicht so ist[29]) vorstellen, daß wir in Haskell Funktionen nach der maximalen Zahl, wie oft Auswertung iteriert werden kann, (zumindest "moralisch") graduieren können. "nulläre Funktionen" (also solche, in die sich nichts einsetzen läßt) sollten dann konzeptionell gerade so was wie Basisgrößen darstellen können. Wir wollen sie künftig einfach nur als "Daten" ansehen.

Zeit für ein erstes Beispiel:

```
data Bases = A | T | C | G deriving (Eq, Show)

complement A = T
complement T = A
complement C = G
complement G = C
```

Was macht man nun damit? Hintergrund: Ein guter erster Anlaufpunkt für Haskell ist `http://www.haskell.org`. Es gibt verschiedene Haskell-Systeme, Compiler wie Interpreter. Für die ersten Schritte sollte `hugs` eine gute Wahl sein. (In einigen freien Unix-Distributionen auch schon als Paket enthalten.) Um mit diesem Code was anfangen zu können, sollte man ihn in ein File (etwa `example1.hs`) schreiben und dann `hugs example1.hs` aufrufen. Nach der Startmeldung begrüßt uns der Haskell-Interpreter mit einem Eingabeprompt, an dem wir verschiedene Dinge zur Auswertung eingeben können, und dabei auf eingebaute wie auch von uns definierte Funktionen zurückgreifen können.

```
tf@ouija: ~/cde/lambda/examples$ hugs example-1.hs
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-1999
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: November 1999  _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions
```

[28] Ganz so böse ist die Welt dann doch wieder nicht. Zwar wird uns das Typsystem eine Definition wie `y = \f->(\ x -> f(xx))(\ x -> f(xx))` um die Ohren hauen, aber mit `y f = f (y f)` definieren wir vollkommen einfach einen Fixpunktoperator. Nur auf `k` anwenden werden wir ihn dann nicht können.

[29] Der Grund sind "Alltypen", die zumindest ganz außen erlaubt sind. Die Identität wird etwa den Typ $\forall\alpha.\alpha \to \alpha$ bekommen. "Moralisch" heißt das, daß sie eine einstellige Funktion ist. Wenn das Argument aber eine Funktion 27-stellige Funktion ist, so können natürlich diese 27 Applikationen weiterhin kommen. . .

```
Reading file "/usr/share/hugs98/lib/Prelude.hs":
Reading file "example-1.hs":

Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
example-1.hs
Type :? for help
Main> (+) 2 3
5
Main> (\x->(*) x x) 3
9
Main> A
A
Main> complement T
A
Main> complement C
G
Main> (+) A C
ERROR: Illegal Haskell 98 class constraint in inferred type
*** Expression : A + C
*** Type       : Num Bases => Bases

Main> (+) 2 C
ERROR: Illegal Haskell 98 class constraint in inferred type
*** Expression : 2 + C
*** Type       : Num Bases => Bases

Main> complement 2
ERROR: Illegal Haskell 98 class constraint in inferred type
*** Expression : complement 2
*** Type       : Num Bases => Bases

Main> complement
ERROR: Cannot find "show" function for:
*** Expression : complement
*** Of type    : Bases -> Bases

Main> A == T
False
Main> A == A
True
Main>
```

Den Haskell-Interpreter verläßt man am schnellsten mit Control-D. Hier sind absichtlich etliche Fehler produziert, weil sie einige sehr lehrreiche Dinge über das System verraten.

Wie man sieht, läßt sich mit Haskell ganz einfach rechnen. Es mag seltsam erscheinen, daß die Addition hier `(+)` geschrieben wird, der Grund ist aber einfach nur der, daß die eingebauten arithmetischen Funktionen sind normalerweise Infixfunktionen, d.h. man kann $2+3$ einfach `2+3` schreiben, allerdings gibt es sowohl syntaktische Möglichkeiten, Infixfunktionen in Präfixnotation wie wir sie bisher kennen zu verwenden als auch umgekehrt binäre nicht-Infixfunktionen in Infixnotation. Die runden Klammern machen aus `+` gerade eine "normale" Funktion, aber wie gesagt: das ist reiner syntaktischer Zauber. Außerdem haben wir auch noch kurz an einem Beispiel vorgeführt bekommen, wie man von der Kommandozeile aus mit anonymen Funktionen - hier die Quadratabbildung - rechnet. Die Notation ist natürlich an den Lambda-Kalkül angelehnt. Es ist allerdings bei `hugs` (anders etwa als z.B. bei vielen Implementierungen der in vielen Dingen Haskell nahe verwandten aber rein funktional gesehen nicht ganz so tollen Sprache SML) nicht möglich, am Prompt interaktiv neue Definitionen einzugeben[30].

Kommen wir zum Programm. Die erste Zeile definiert einen neuen Datentyp `Bases` `A`, `T`, `C`, `G` sind sogenannte "Konstruktoren". Das sind (hier nulläre) Funktionen, die Werte von diesem Typ liefern. (Gemeint sind natürlich hier die Nucleobasen Adenin, Thymin, Cytosin und Guanin). Der Nachsatz `deriving (Eq, Show)` hat was mit dem Typklassensystem von Haskell zu tun und sorgt dafür, daß das System in der Lage ist, nicht nur mit diesen Werten zu rechnen, sondern gewisse naheliegende Definitionen (in diesem Fall "Gleichheit" und "Darstellung als Zeichenkette") für diesen Typ gelten. (Wir "überladen" gewisse Funktionen wie die Vergleichsfunktion `==` für diesen neuen Typ.) Es soll an dieser Stelle explizit darauf hingewiesen sein, daß dieser Datentyp `Bases`, den wir gerade definiert haben, sich *aus Sicht des Programmierers*[31] durch nichts von Typen unterscheidet, die das System bereits mitbringt (wie z.B. `Bool` mit den Konstruktoren `True` und `False`, die Wahrheit und Falschheit repräsentieren, oder auch `Integer` mit den Konstruktoren `0`, `1`, `-1`, `2`, `-2`, ...). Die "eingebauten" Typen sind in einigen Fällen unter der Haube aus verschiedenen Gründen (beispielsweise Effizienz) anders realisiert, aber das braucht uns nicht groß zu kümmern.

Darunter definieren wir die Funktion `complement` durch eine explizite Fallunterscheidung. (Wichtig ist bei Haskell, daß jede Definition schön mathematisch nur auf vorangehende Definitionen aufbauen kann, hier baut `complement` auf `Bases` auf[32].) Fallunterscheidungen sind immer so zu lesen, daß der nachfolgende Fall

[30] Dies scheint zunächst wie ein Nachteil, sollte aber nicht als solcher angesehen werden. Würden wir so etwas erlauben, so bräuchte man einen (funktionalen Programmieren verhaßten) Zustandsbegriff. Anders ausgedrückt: im Laufe unserer Sitzung könnte es uns passieren, daß wir auf die gleiche Eingabe ein verschiedenes Verhalten beobachten können, allein schon deshalb, weil eine Funktion die vorher noch nicht definiert war (was dann wohl zu einem Fehler führte) inzwischen definiert ist...

[31] Und nicht nur aus dessen Sicht. Wer sich mit `hugs` mal etwas intensiver auseinander gesetzt hat, der wird feststellen, daß doch erstaunlich viele Datentypen erst in der "Standard-prelude" definiert werden...

[32] Es sollte angemerkt werden, daß Haskell auch etwas großzügiger ist, wenn klar ist, in welcher reihenfolge die Dinge "offiziell" definiert sind. Aber nachdem dies ohnehin kein guter Programmierstil ist, gehen wir hier nicht auf Details ein...

nur angesehen wird, wenn keiner der vorangehenden zutrifft[33]. Das Schöne ist, daß sich Konstruktoren auf der linken Seite einer Definition im Pattern Matching einsetzen lassen. Dazu gleich noch was. Der Mitschnitt zeigt, daß sich die Funktion wie erwartet verhält.

Zu den oben absichtlich produzierten Fehlern: Basen lassen sich nicht addieren, Basen lassen sich auch nicht zu Zahlen addieren, und `complement` läßt sich nicht auf Zahlen anwenden. In jedem dieser Fälle sorgt das Typsystem dafür, daß solche Fehler abgefangen werden - nicht nur interaktiv, sondern auch in Programmcode. Das Typsystem ist sogar noch sehr viel pfiffiger, aber hierauf soll an dieser Stelle nicht eingegangen werden[34] [35]. Der letzte Fehler geht darauf zurück, daß das System nicht weiß, wie es eine Abbildung von Basen auf Basen ausdrucken soll. (Der einzige Grund, warum z.B. bei `A` nicht genau dasselbe passiert, ist das `deriving Show` am Anfang.)

Nach diesem Vorgeplänkel wollen wir nun mit Haskell ein erstes kleines Problem lösen: die "Türme von Hanoi".

> Einer Legende nach gibt es in einem sehr abgelegenen Winkel der Welt ein Bergkloster, in dem Mönche seit Anbeginn der Zeit rund um die Uhr damit beschäftigt sind, einen Turm aus 64 goldenen gelochten Metallscheiben, der auf einer von drei Stangen aufgebaut ist, auf eine andere Stange umzuschichten. Die Regel ist, daß immer nur eine Scheibe auf einmal umgesetzt werden darf und während dieser Prozedur niemals eine größere Scheibe auf einer kleineren zu liegen kommen darf. Die Mönche machen jede Sekunde einen Zug, wenn sie ihre Aufgabe vollendet haben, endet die Welt...

Zur besseren Illustration in Zeichnung 1 die ersten paar Züge, die einen Dreierturm auf der ersten Stange auf die zweite Stange transportieren.

Da die Mönche mit dieser Aufgabe 18446744073709551615 Sekunden (rund $5.8 \cdot 10^{11}$ Jahre) beschäftigt sein werden, sollten wir uns hoffentlich keine allzu großen Sorgen über den baldigen Weltuntergang machen müssen...

Wir wollen wissen, welche Züge zur Lösung des Problems im einzelnen durchzuführen sind. Mit dem richtigen Ansatz läßt sich das in Haskell sehr bequem ausrechnen. Das Problem ist intrinsisch rekursiv. Um einen Turm aus $N \geq 1$

[33]... und dies durch ein endliches Scheitern erkannt wurde. Aber hier begeben wir uns in etwas unschöne Gefilde, auf die wir nachher noch mal zurückkommen sollen...

[34]Einer der Kursleiter möchte anmerken, dass er beim praktischen Arbeiten mit diesen Systemen die Erfahrung gemacht hat, dass das Typsystem mehr als 95% aller Programmierfehler, die man in Sprachen wie C oder LISP macht, zur Compilierzeit erkennen und damit verhindern kann.

[35]Der andere Kursleiter ist sich über den Wert von Typsystemen nicht so sicher, auch aus eigener Erfahrung. Etwas spöttisch pflegt er manchmal zu sagen, daß ein erwachsener Mensch ja wohl weiß, was er tut und einer Maschine darüber nicht (in Form des Typsystems) Rechenschaft schuldig ist. Andererseits erkennt er an, daß gewisse (triviale) Fehler tatsächlich abgefangen werden können.

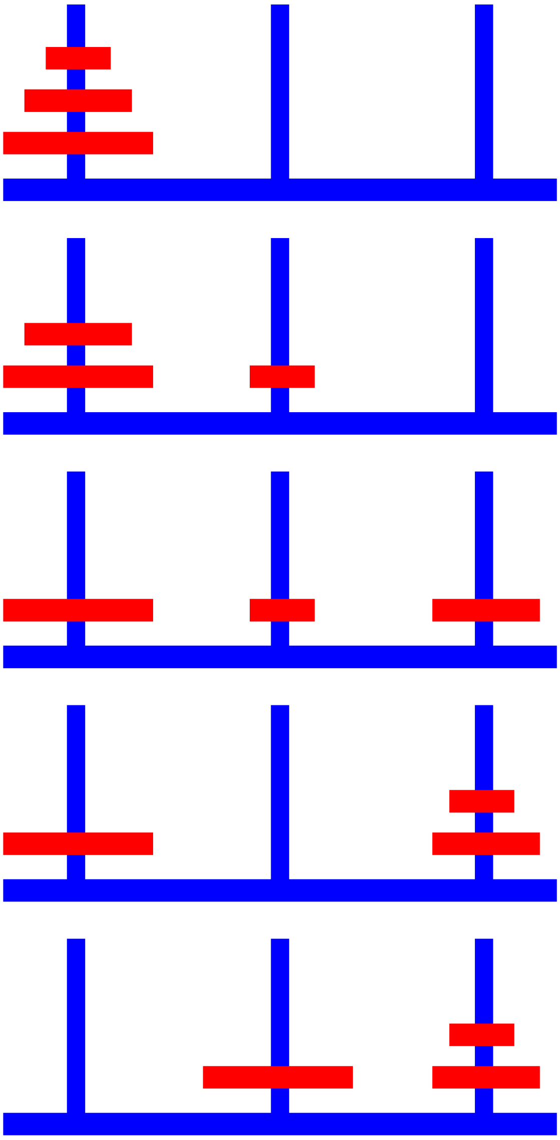

Abbildung 1: Die ersten Züge zum Bewegen eines Dreierturms.

Scheiben von Stange A nach Stange B zu befördern, befördern wir erst den Turm von der kleinsten (obersten) Scheibe bis zur $N-1$-ten Scheibe von A nach C, dann die N-te Scheibe von A nach B und schließlich den $N-1$-Turm auf Scheibe C nach B. Einen Turm aus 0 Scheiben zu befördern ist trivial. In Haskell läßt sich das sehr einfach und direkt formulieren:

```
{-

  Towers of Hanoi

-}

data Peg = Peg1 | Peg2 | Peg3 deriving (Eq, Show)
data Moves = End | Then Peg Peg Moves deriving (Eq, Show)

append End moves                     = moves
append (Then peg_a peg_b rest) moves = Then peg_a peg_b
                                            (append rest moves)

move_tower::Integer -> Peg -> Peg -> Peg -> Moves
move_tower 0 a b c = End
move_tower n a b c = append (move_tower (n-1) a c b)
                            (Then a b (move_tower (n-1) c b a))
```

Das ist das komplette Programm. Ohne auf die letzten sprachlichen Details einzugehen, sollen einige Dinge erwähnt werden: Als erstes fällt auf, daß auf der linken Seite einer Definition nicht nur Werte, sondern auch Variablen auftauchen können, die beim Pattern Matching die entsprechenden Werte erhalten. Weiterhin fällt auf, daß wir auf der rechten Seite einer Definition auf die Größe zurückgreifen können, die wir gerade definieren. Sowohl in Datendefinitionen als auch in Funktionsdefinitionen. In gewisser Weise ist das sehr angenehm, denn auf diese Weise muß man nicht erst das Fixpunktprinzip verstanden haben, um rekursive Funktionen zu definieren[36].

Man hätte hier auch auf eingebaute Datentypen wie Listen und Paare zurückgreifen können um das Problem zu lösen, aber für's erste wurde hierauf verzichtet um noch mal zu unterstreichen, daß wir auch ganz ohne und völlig mit eigenen Definitionen auskommen können und um nicht zu viel von der Syntax von Haskell erklären zu müssen, die in gewisser Weise nur zusätzlicher syntaktischer Zucker ist. Alle Werte in Haskell haben einen Typ. Wenn man den Typ einer Funktion explizit im Programmcode unterstreichen möchte, kann man ihn explizit angeben (wie oben bei `move_tower`: dies ist eine Funktion, die eine ganze Zahl abbildet auf eine Funktion, die eine Stange abbildet auf eine Funktion, die eine Stange abbildet auf eine Funktion, die eine Stange abbildet auf Züge). Wenn man das nicht macht, wird der Typ automatisch abgeleitet (wie bei `hanoi`).

[36] Und dies ist die angedeutete "Hintertür" durch die der Fixpunktoperator wieder Einzug hält...

Spielen wir mal ein wenig damit herum:

```
Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
hanoi.hs
Type :? for help
Main> hanoi 0
End
Main> hanoi 1
Then Peg1 Peg2 End
Main> hanoi 2
Then Peg1 Peg3 (Then Peg1 Peg2 (Then Peg3 Peg2 End))
Main> hanoi 3
Then Peg1 Peg2 (Then Peg1 Peg3 (Then Peg2 Peg3 (Then Peg1 Peg2 (Then Peg3
Peg1 (Then Peg3 Peg2 (Then Peg1 Peg2 End))))))
```

Das sieht ganz vielversprechend aus.

Der `hugs`-Interpreter hat einige nette Features, die wenig mit der eigentlichen Sprache zu tun haben, die aber trotzdem an der einen oder anderen Stelle kurz erwähnt seien: beispielsweise kann man am Kommandoprompt gewisse "Metakommandos" eingeben, die mit einem Doppelpunkt beginnen (`:?` liefert eine Liste). Interessant ist z.B. `:type ausdruck`. Damit läßt sich der inferierte Typ eines Ausdrucks ausgeben:

```
Main> :type Peg1
Peg1 :: Peg
Main> :type End
End :: Moves
Main> :type Then Peg1 Peg2 End
Then Peg1 Peg2 End :: Moves
Main> :type Then Peg1 Peg2
Then Peg1 Peg2 :: Moves -> Moves
Main> :type Then Peg1
Then Peg1 :: Peg -> Moves -> Moves
Main> :type Then
Then :: Peg -> Peg -> Moves -> Moves
Main> :type hanoi
hanoi :: Integer -> Moves
Main> :type move_tower
move_tower :: Integer -> Peg -> Peg -> Peg -> Moves
Main> :type hanoi 2
hanoi 2 :: Moves
```

Daß muß wohl nicht groß kommentiert werden. Mit ein wenig Nachdenken stellt man fest, daß das gerade die Typen sind, die man auch erwarten würde. Ein anderes witziges Feature von Hugs ist, daß der Interpreter mit der Kommandozeilenoption `+s` veranlaßt werden kann, bei jeder Berechnung die Zahl der Reduktionsschritte mit anzuzeigen.

Schön und gut, all das riecht auch ein kleinwenig nach λ-Kalkül, aber können wir ganz explizit sehen, wie der λ-Kalkül in Haskell eigebettet ist? Können wir. Etwa, indem wir uns ein paar Kombinatoren definieren und mit ihnen spielen:

```
s x y z = x z (y z)
k = \x y -> x

genesis = \x -> x k s k

i x = x

i2 x y = x y

b1 x y z = x (y z)
b2 = s(k s)k

p x y z = z x y

true = k
false = k i
```

Wir spielen wieder ein wenig damit rum. (Die Kombinatoren mußten klein geschreiben werden, weil in Haskell der Name einer Funktion, die kein Konstruktor ist, nicht mit einem Großbuchstaben beginnen darf.)

```
Main> k 2 3
2
Main> k 5 8
5
Main> k 3 s
3
Main> k (\x->x*x) 3 5
25
Main> s (\offset -> \x -> x+offset) (\offset -> 1000*offset) 2
2002
Main> s k k 3
3
Main> i 3
3
Main> p 2 3 true
2
Main> p 2 3 false
3
Main> b1 (\x->x*x) (\x->x+100) 2
10404
Main> b2 (\x->x*x) (\x->x+100) 2
10404
Main> k "Hallo" 5
"Hallo"
```

```
Main> k "Hallo" "Welt"
"Hallo"
```

Da lacht das Herz.

Wenn man genau hinsieht, kann man hier allerdings ein wenig stutzig werden: was mag der Typ von `k` hier wohl sein, wenn `k wert wert` einmal eine Zahl und einmal ein String sein kann? Wenn wir schon Typen haben, nimmt `k` als nackter Kombinator, als rein abstrakte Größe, in die keine konkreten Werte eingehen, eigentlich keinen Bezug drauf. Das heißt: nicht ganz, denn zumindest wissen wir, daß sich, wenn der Wert `X` vom Typ `TX` in `k` eingesetzt wird, eine Funktion ergibt, die einen Wert vom Typ `TX` liefert (eben gerade `X`). So gesehen kann `k`, was das Typverhalten angeht, ganz allgemein gehalten als Funktion aufgefaßt werden, die Werte vom Typ α abbildet auf Funktionen, die Werte vom Typ β abbildet auf Werte vom Typ α — und das für *beliebige* Typen α und β. Das Schöne ist nun, daß das Typsystem von Haskell eine verlustfreie Umsetzung dieser Idee erlaubt. In der Tat, sehen wir's uns an:

```
Main> :type k
k :: a -> b -> a
Main> :type i
i :: a -> a
Main> :type b1
b1 :: (a -> b) -> (c -> a) -> c -> b
Main> :type b2
b2 :: (a -> b) -> (c -> a) -> c -> b
Main> :type p
p :: a -> b -> (a -> b -> c) -> c
Main> :type p True
p True :: a -> (Bool -> a -> b) -> b
Main> :type p True False
p True False :: (Bool -> Bool -> a) -> a
Main> :type p True k
p True k :: (Bool -> (a -> b -> a) -> c) -> c
Main> :type p p p
p p p :: ((a -> b -> (a -> b -> c) -> c) -> (d -> e -> (d -> e -> f)
-> f) -> g) -> g
Main> :type s k k
s k k :: a -> a
Main> :type s s
s s :: ((a -> b -> c) -> a -> b) -> (a -> b -> c) -> a -> c
Main> :type (\x->not x)
\x -> not x :: Bool -> Bool
Main> :type (\x y -> not y)
\x y -> not y :: a -> Bool -> Bool
```

Ebenso wie Funktionen in dem Sinne "abstrakt" sein können, daß der Typ ihres Arguments nicht festgenagelt zu sein braucht, gilt dies auch für Konstruktoren.

Damit können wir so was wie abstrakte "Containertypen" - etwa allgemeine Listen, Bäume, etc. definieren. Für Listen sei das explizit demonstriert. Wir erinnern uns: eine Liste ist entweder ein spezieller Wert, der die leere Liste repräsentiert, oder ein geordnetes Paar aus einem Wert und einer Liste. Wenn wir (was zumeist Sinn macht) fordern wollen, daß alle Werte in der Liste gleichen Typ haben, lautet eine mögliche Definition in Haskell so:

```
data List a = Nil | Pair a (List a) deriving (Eq, Show)
```

In der Tat:

```
Main> :type Nil
Nil :: List a
Main> Pair True Nil
Pair True Nil
Main> :type Pair True Nil
Pair True Nil :: List Bool
Main> :type Pair False (Pair True Nil)
Pair False (Pair True Nil) :: List Bool
Main> :type Pair False (Pair 5 Nil)
ERROR: Illegal Haskell 98 class constraint in inferred type
*** Expression : Pair False (Pair 5 Nil)
*** Type       : Num Bool => List Bool


Main> Pair (Pair True (Pair False Nil)) (Pair (Pair True Nil) Nil)
Pair (Pair True (Pair False Nil)) (Pair (Pair True Nil) Nil)
Main> :type Pair (Pair True (Pair False Nil)) (Pair (Pair True Nil) Nil)
Pair (Pair True (Pair False Nil)) (Pair (Pair True Nil) Nil) :: List (List
Bool)
Main> Pair (Pair True Nil) (Pair True Nil)
ERROR: Type error in application
*** Expression     : Pair (Pair True Nil) (Pair True Nil)
*** Term           : Pair True Nil
*** Type           : List Bool
*** Does not match : Bool
```

Wie wir sehen, können wir auch z.B. Listen von Listen (hier von Bool) definieren. Das Typsystem sorgt dafür, daß wir z.B. nicht versehentlich einen Bool in eine Liste von Listen von Bool packen können. Auch wenn ein obligatorisches Typsystem in manchen Situationen eine Einschränkung darstellen kann - in anderer Hinsicht macht es schier unglaubliche Dinge möglich. Beispielsweise wäre es vielleicht denkbar, ein Programm zu schreiben, das einen Datentyp `Theorem` implementiert, wobei die nackten Konstruktoren allerdings so abgekapselt sind, daß die einzige Möglichkeit, einen Wert vom Typ `Theorem` zu generieren die ist, eine mitgegebene `Theorem`-wertige Funktion zu verwenden, die eine logische Schlußregel implementiert. Man kann das getrost so sehen, daß dieser Ansatz das alte philosophische Problem löst, eine “Sprache” (natürlich im verallgemeinerten Sinn) zu schaffen, in der es unmöglich ist, zu lügen. Wenn irgend ein Wert den Typ `Theorem` hat, weiß man (wenn das System ordentlich gebaut ist und “wahre” Basis-Theoreme -Axiome- mitbringt), daß dieser Wert eine wahre Aussage darstellt. Es gibt in der Tat formale Beweisverifikationssysteme, die *so ähnlich* funktionieren. Wer sich dafür interessiert, sollte sich mal *Isabelle* anschauen.

Wohlgemerkt: *so ähnlich.* Bevor jetzt alle anfangen wie wild Programme nur ihres Typs wegen zu schreiben, ein Wort der Warnung. Es ist nämlich in der Tat so, daß in Haskell alle Typen bewohnt sind. Ein Bewohner von $\perp = \forall\alpha.\alpha$ wird zwar rechnerisch nicht viel sinnvolles zu Stande bringen, aber immerhin er existiert!

```
i = \x->x
y f = f (y f)
bot = y i
```

Und der zugehörige Dialog (zusammen mit dem vom System bereitgestellten Bewohner eines jeden Typs):

```
Main> :type bot
bot :: a
Main> :type error "bot"
error "bot" :: a
Main>
```

Listen in der Art, wie wir sie gerade gesehen haben, sind eine sehr praktische Sache und tauchen praktisch überall auf. Beispielsweise ist der rekursiv definierte Hanoi-Zugabfolge-Datentyp eigentlich auch nichts anderes als eine sehr spezielle Form einer Liste. Natürlich ist auch hier die abstrakte Definition eigentlich die wertvollere, weil es möglich ist, auf ihr Funktionen zu definieren, die allgemein nur mit der Tatsache zu tun haben, daß wir mit Listen arbeiten, und immer wieder gebraucht werden. Beispielsweise eine Funktion, die die Länge einer Liste liefert, oder eine Funktion, die eine Liste umdreht, oder... Hier ein paar Beispiele:

```
data List a = Nil | Pair a (List a) deriving (Eq, Show)

-- compute the length of a list

my_length Nil = 0
my_length (Pair a b) = 1 + my_length b


-- reverse a list
```

```
my_reverse list =
 let
  traverse so_far Nil = so_far
  traverse so_far (Pair a rest) = traverse (Pair a so_far) rest
 in
  traverse Nil list

-- apply a function to every element of a list
-- (Strictly speaking, this is even better:
-- given a function a->b, produce a function List a -> List b
-- defined by sequentially applying function to every list element
-- and collecting the values in a list)

my_map f Nil = Nil
my_map f (Pair h t) = Pair (f h) (my_map f t)

my_take 0 _   = Nil
my_take n Nil = Nil
my_take n (Pair h t) = Pair h (my_take (n-1) t)
```

Und der Dialog...

```
Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
list.hs
Type :? for help
Main> my_reverse (Pair 1 (Pair 2 (Pair 3 Nil)))
Pair 3 (Pair 2 (Pair 1 Nil))
Main> my_length (Pair 1 (Pair 2 (Pair 3 Nil)))
3
Main> my_map (\x->x*x) (Pair 1 (Pair 2 (Pair 3 (Pair 4 Nil))))
Pair 1 (Pair 4 (Pair 9 (Pair 16 Nil)))
```

Wie man sich unschwer vorstellen kann, sind Listen so nützlich, daß Haskell sie bereits mitbringt - und auch spezielle Syntax dafür besitzt. (Das ist auch gut, denn unsere Notation ist *sehr* schwerfällig. Für Tupel gilt ähnliches. All die Funktionen, die hier vorgestellt wurden, gibt es in der Tat für die eingebauten Listen (Ihre Namen tragen naheliegenderweise einfach kein `my_` am Anfang) und sie sind ziemlich genau so wie hier vorgestellt implementiert. Beim einführenden Hanoi-Beispiel gab es gute Gründe, alle Datenstrukturen selbst zu definieren. Das andere Extrem wäre, ausschließlich die eingebauten Datentypen zu verwenden und z.B. Stäbe durch die Zahlen 1,2,3 darzustellen, Züge durch 2-Tupel (Paare) und Zugfolgen durch Listen. Zwischenformen sind natürlich auch möglich. Das sähe dann beispielsweise so aus:

```
move 0 a b c = []
move n a b c = (move (n-1) a c b)++((a,b):(move (n-1) c b a))
hanoi n = move n 1 2 3
```

Und der Dialog...

```
Main> hanoi 5
[(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2),(1,3),(2,3),(2,1),
 (3,1),(2,3),(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2),(3,1),
 (2,3),(2,1),(3,1),(3,2),(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),
 (1,2)]
```

Diese Zugfolge ist eine Haskell-Liste und wie erwähnt können die von Haskell bereits mitgebrachten Listenfunktionen darauf angewandt werden:

```
Main> length (hanoi 5)
31
Main> length (hanoi 6)
63
Main> length (hanoi 7)
127
Main> take 5 (hanoi 7)
[(1,2),(1,3),(2,3),(1,2),(3,1)]
```

Wer hier einmal auf den Geschmack gekommen ist, was echte Polymorphie leisten kann, will Ansätze wie z.B. das Template-System von `C++`, das konzeptionell in eine ähnliche Richtung geht, am liebsten des Mitternachts in einem tiefen Grab versenken.

Stellen wir uns mal ganz frech an (und erhalten trotzdem nach Sekundenbruchteilen eine Antwort):

```
Main> take 5 (hanoi 64)

[(1,3),(1,2),(3,2),(1,3),(2,1)]
```

In der Tat wirft dieses Ergebnis - oder vielmehr: die Art, wie es auf dem Bildschirm erscheint, hier einige Fragen auf. Sogar *sehr* beunruhigende Fragen, wenn man an die anfangs erwähnte Apokalypse denkt.

Nüchtern betrachtet kann es nicht sein, daß Haskell zuerst ganz eifrig `hanoi 64` ausrechnet und dann hiervon die ersten 5 Elemente nimmt. Selbst wenn Zeit nicht das Problem wäre, würde uns der Platzbedarf ganz sicher das Genick brechen. Trotzdem ist es geschehen. Doch wie sagt Douglas Adams: wenn es auf keine mögliche Art und Weise passiert sein kann, muß es eben auf unmögliche Weise geschehen sein. Das kommt hier in etwa ihn.

Anders als eigentlich alle anderen herkömmlichen Programmiersprachen werden in Haskell Funktionswerte *nicht* so berechnet, daß zuerst die Funktionsargumente (hier: `hanoi 64`) ausgerechnet werden, dann eine Funktion mit diesen Werten aufgerufen und ihr Ergebnis zurückgeliefert wird. Vielmehr macht Haskell das, was im Lambda-Kalkül links-außen-Reduktion entspricht: es betrachtet `hanoi`

64 als eine Aufforderung, einen Wert zu berechnen (kann man auch als nicht ausreduzierten Kombinator ansehen) in die Definition der Funktion `take 5` ein. Reduziert wird jeweils nur so weit, wie es wirklich nötig ist[37].

In etwas vertrauterer Sprache kann man das vielleicht so ausdrücken: Haskell macht aus `hanoi 5` nicht sofort einen entsprechenden Wert, sondern erst einmal ein Versprechen, diesen Wert auszurechnen, wenn er wirklich gebraucht wird. Wenn jemand mit diesem Wert weiterrechnen will, kriegt er zuerst nur dieses Versprechen, den Wert bei Bedarf auszurechnen. Erst, wenn es sich wirklich nicht mehr vermeiden läßt, wird das Versprechen eingelöst. Eine Liste ist so gesehen ein Versprechen, das, wenn eingelöst, ein Paar liefert aus einem Versprechen, den Kopf auszurechnen, und einem Versprechen, den Schwanz auszurechnen[38]. Wenn wir die fünf ersten Züge wollen, werden die Versprechen genau so weit eingelöst, wie es dafür notwendig ist. Unnötige Arbeit macht die Maschine keine.

Man kann sich jetzt fragen, was in Situationen wie dieser passiert:

```
Prelude> (\x->x*x) (5+2)
49
```

Hat hier Haskell das wirklich erst zu `(5+2)*(5+2)` reduziert, und dann zweimal unabhängig `5+2` ausgerechnet? Die Antwort ist Nein: implementiert ist das vielmehr so, daß dieses eingesetzte `x` intern zweimal derselbe Wert, bzw. dasselbe Versprechen ist. Ist es einmal eingelöst worden, steht es auch das zweite Mal zur Verfügung. Es ist vielleicht sinnvoller, sich Berechnungen in Haskell als eine Art sukzessive Graphenreduktion vorzustellen als auf die von anderen Programmiersprachen bekannte Weise.

Dieser Ansatz ist auch als 'faule Auswertung' bekannt - er ermöglicht einige abartige Tricks; wir werden hiervon in der nächsten Lektion noch mehr sehen.

Auch wenn wir damit was Haskell anbelangt eigentlich nur an der Oberfläche gekratzt haben, wollen wir es dabei belassen und uns im folgenden noch kurz mit LISP befassen.

Lisp wurde in den 50ern von John McCarthy erfunden. Von der Idee her ist es eigentlich gar nicht mal unbedingt eine Programmiersprache, mit der man eine Maschine füttern sollte, sondern vielmehr eine abstrakte und strenge Notation für die Formalisierung von Berechnungen aller Art. Man könnte jetzt den Standpunkt vertreten, daß alle Algorithmen doch an sich ebensogut — um in er Zeit zu bleiben — in FORTRAN formalisiert werden können. Die Frage ist nur, ob man das will. Wenn wir eine Sprache wollen, um über komplexe, teils auch schon mal ganz arg verwickelte Abläufe zu sprechen, so sollte diese Sprache alle wesentlichen Einzelschritte akkurat beschreiben können, und selbst

[37] Das dies tatsächlich immer zum Ziel führt, wenn es überhaupt eines gibt, werden wir in Kapitel 9 beweisen.

[38] Damit ist aber schon arbeit geleistet! Immerhin wissen wir dann, daß die Liste nicht leer ist. Vielleicht reicht der einen oder anderen Funktion diese Information ja schon. Und wenn nicht? Naja, erst mal abwarten, ob es überhaupt so weit kommt...

nicht durch irgendwelche hinderlichen oder gar mißverständlichen Konstrukte zusätzliche Probleme schaffen. Das ist auch der eigentliche Grund, warum eine natürliche Sprache hierfür nicht gut geeignet ist, und in der Tat Bedarf für eine eigene Sprache besteht.

Um bei Formalisierungen von Abläufen zu bleiben: in fast allen bekannten herkömmlichen Programmiersprachen, die mit der Maschine im Hinterkopf entwickelt wurden (und oft eigentlich nur ein "verzuckerter Assembler"[39] sind) ist es *nicht* möglich, ganz einfache Grundideen auf einfache Weise auszudrücken. Das einfachste Beispiel dürfte wohl die Addition von zwei ganzen Zahlen sein. Versuchen wir's in `C`:

```
int add(int a, int b)
{
return a+b;
}
```

Der Overhead, um wirklich damit arbeiten zu können, ist allerdings nicht ganz ohne:

```
#include <stdlib.h>
#include <stdio.h>

int add(int a, int b)
{
return a+b;
}


int main(int argc, char *argv[])
{
int a,b;

if(argc!=3)
{
fprintf(stderr, "Usage: %s <number> <number>\n", argv[0]);
return 1;
}

a=atoi(argv[1]);
b=atoi(argv[2]);

printf("%d + %d = %d\n", a,b,add(a,b));

return 0;
}
```

[39] Um diese Formulierung zu verstehen denke man an den berühmten "sytactical sugar", der in Beschreibungen von Programmiersprachen immer wieder auftaucht...

Damit:

```
tf@ouija: ~/cde/lambda/examples$ gcc -o add add.c
tf@ouija: ~/cde/lambda/examples$ ./add 1 1
1 + 1 = 2
```

Hier scheint alles in Ordnung zu sein — was soll also die Behauptung, man könne in C die Addition von zwei ganzen Zahlen nicht gut formalisieren? Nun:

```
tf@ouija: ~/cde/lambda/examples$ ./add 100000000001 122222222221
2147483647 + 2147483647 = -2
```

(Es könnte interessant sein, hier die `atoi(3)` und `strtol(3)`-Manpages zu konsultieren, um dieses Verhalten zu erklären.)

Man kann nun drei Standpunkte vertreten: zum einen den blind pragmatischen, daß derartige Zahlen in der Praxis praktisch nie auftreten, bzw. dann nicht zwingend wirklich als Ganzzahlen behandelt werden müssen, und man sich deswegen über solche Dinge keine großen Gedanken machen muß. (Letztendlich steckt hinter allem die begrenzte Bittigkeit der auf der CPU festverdrahteten Elementaroperationen. (Wer so denkt, schreibt wahrscheinlich auch Code mit vielen Buffer Overflows, die Stacksmash-Angriffe eine wahre Freude werden lassen.)) Zum anderen den aufgeklärt pragmatischen, daß dies zwar eine durchaus ernstzunehmende Beschränkung ist, man aber mit etwas Vorsicht in seinen Programmen dafür sorgen kann, daß diese Beschränkung sich nicht negativ auswirkt. Schließlich gibt es auch noch den mathematischen Standpunkt, daß die natürlichen Zahlen heilig sind, und eine derartige Implementierung unsinnig und gefährlich ist, weil sie nicht einmal im Ansatz versucht, korrekt zu bleiben, oder aber mit dem Hinweis, daß sie dazu nicht in der Lage ist, abzubrechen. Nein, stattdessen kriegen wir stillschweigend ein falsches Ergebnis.

`LISP` ist in der Tat nicht nur formale Notation, sondern auch Programmiersprache, und eine auch nur halbwegs anständige `LISP`-Implementierung (wie etwa `CMU CL` oder `CLISP` oder auch `Guile` oder `Gambit`, nicht aber z.B. der `Emacs`[40]) hat sich bitte so zu verhalten:

```
[1]> (+  100000000001 122222222221)

222222222222

[2]> (expt 2 1024)

17976931348623159077293051907890247336179
76978942306572734300811577326758055009631
32708477322407536021120113879871393357658
78976881441662249284743063947412437776789
```

[40] auch wenn wir jenem speziellem Freund ein eigenes Kapitel widmen…

```
342486548527630221960124609411945308295 20
850057688381506823424628814739131105408 27
237163350510684586298239947245938479716 30
4835356329624224137216

[3]> (time (expt 2 1024))

Real time: 9.0E-5 sec.
Run time: 0.0 sec.
Space: 324 Bytes

179769313486231590772930519078902473361 79
769789423065727343008115773267580550096 31
327084773224075360211201138798713933576 58
789768814416622492847430639474124377767 89
342486548527630221960124609411945308295 20
850057688381506823424628814739131105408 27
237163350510684586298239947245938479716 30
4835356329624224137216
```

Was ausgesagt werden soll: mathematische Sauberkeit, Korrektheit, und die Möglichkeit, über die Dinge zu sprechen, über die man wirklich sprechen will, ohne sich zusätzlich mit der Sprache herumärgern zu müssen, sind gerade die Features, auf die es den `LISP`-Hackern ankommt. Haskell erbt das natürlich. In einer naheliegenden aber ziemlich ineffizienten Implementierung der Potenzierung:

```
pow b 0   = 1
pow b exp = b*(pow b (exp-1))

Main> pow 2 1024
17976931348623159077293051907890247336
17976978942306572734300811577326758055
00963132708477322407536021120113879871
39335765878976881441662249284743063947
41243777678934248654852763022196012460
94119453082952085005768838150682342462
88147391311054082723716335051068458629
82399472459384797163048353563296242241
37216
```

Halten wir also nochmal fest: LISP ist dazu gedacht, um irgendwelche Abläufe formalisieren zu können, und es sieht Korrektheit und Eleganz als ein höheres Gut an als die Nähe zur Maschine.

Natürlich drängt sich jetzt die Frage auf, *wie* Berechnungen in LISP formalisiert werden. LISP enthält den Lambda-Kalkül, allerdings nicht auf ganz so unbittelbare Weise wie Haskell. Aber gehen wir noch einmal zurück zu der Stelle, an der wir gelernt haben, mit Kombinatoren zu programmieren. *Was* genau haben wir dort benutzt? Wir hatten Wahrheitswerte, Paare, Bedingungen, und die Möglichkeit, Rekurrenz zu erzeugen. Zahlen wären auch noch ganz angenehm. In gewisser Weise definieren diese Elemente für sich genommen schon so was wie eine kleine Sprache, die ausreicht, um Rechenabläufe zu formalisieren, die hier direkt in den Lambda-Kalkül eingebettet ist, aber von der wir uns durchaus auch vorstellen können, daß es interessant sein könnte, sie herauszulösen und allein für sich genommen zu studieren. Von der Idee her ähnliche Konstruktionen, in denen eine mathematische Struktur allein für sich genommen studiert werden kann, oder eingebettet in eine andere, fundamentalere Struktur untersucht werden kann, findet man quer durch die gesamte Mathematik. Beispielsweise werden in etlichen Anfängervorlesungen die reellen Zahlen als ein Gebilde definiert, das den Körpergesetzen, dem Anordnungsaxiom und dem Trennungsaxiom genügt. (Manche Dozenten weisen auch noch darauf hin, daß dies die Struktur (bis auf Isomorphie) eindeutig festlegt.) Trotzdem drängt sich hier die Frage auf: gibt es in der Tat reelle Zahlen, d.h. können wir aus fundamentaleren Größen eine Struktur konstruieren, die sich wie die reellen Zahlen verhält? (In manchen Vorlesungen — oft sogar schon in der Schule — wird das in der Tat vorgeführt. Möglichkeiten gibt es mehrere. Äquivalenzklassenbildung ist (fast) immer das Werkzeug schlechthin, um solche Strukturen zu erschaffen. In der Schule werden reelle Zahlen gerne als Äquivalenzklassen von (rationalen) Intervallschachtelungen betrachtet; eine Konstruktion, auf die auch wir noch mal kurz zu sprechen kommen werden, definiert reelle Zahlen als Äquivalenzklassen von Cauchy-Folgen. Man kann aber z.B. auch über Dedekindsche Schnitte gehen, und diese Konstruktion ist deswegen interessant, weil sie sich ganz witzig verallgemeinern läßt[41].

Wir wollen also allein mit diesen Grundelementen eine Sprache definieren, und wir wollen versuchen, uns dabei (soweit als möglich) halbwegs schlau anzustellen. Wenn wir schon Zahlen haben, sollten wir auch keine allzu große Hemmungen haben, gewisse naheliegende Funktionen wie die Grundrechenarten Addition, Multiplikation, etc. direkt in unsere Sprache hineinzunehmen, ohne uns zu große Gedanken darüber machen zu müssen. Funktionen werden wir auf die eine oder andere Weise sowieso haben, und spezielle Funktionen können wir bei Bedarf so viele reinnehmen, wie wir grade wollen. Zahlen sind unteilbare Basisgrößen, die in LISP deswegen naheliegenderweise “Atome” genannt werden. Als nächstes brauchen wir Wahrheitswerte. Die Konvention in LISP (*nicht* aber beispielsweise in manchen Dialekten wie Scheme[42]) ist, daß der spezielle Wert `NIL` für “falsch” steht, und jeder andere Wert für “wahr” steht. Von der Idee

[41] Anmerkung für die Kirchheim-Teilnehmer: aus der Kursankündigung geht hervor, daß dies auch in einem der Kurse besprochen wird. Allerdings hat es der Kursleiter etwas besser verstanden als wir, in der Ankündigung geschickt zu verstecken, daß es nicht nur um Spiel und Spaß, sondern um ganz ordentliche, harte Mathematik geht. Bei uns ist es vielleicht sogar eher andersherum: die Ankündigung klingt nach harter Mathematik, und eigentlich geht es um Spiel und Spaß.

[42] Auch wenn das in `Scheme` da noch mal so eine eigene Geschichte ist: uns ist garantiert, daß `#t` für “wahr” und `#f` für “falsch” steht; ansonsten wissen wir nichts über die Bedeutung anderer Werte als Wahrheitswerte (noch nicht einmal, ob sie als solche aufgefaßt werden

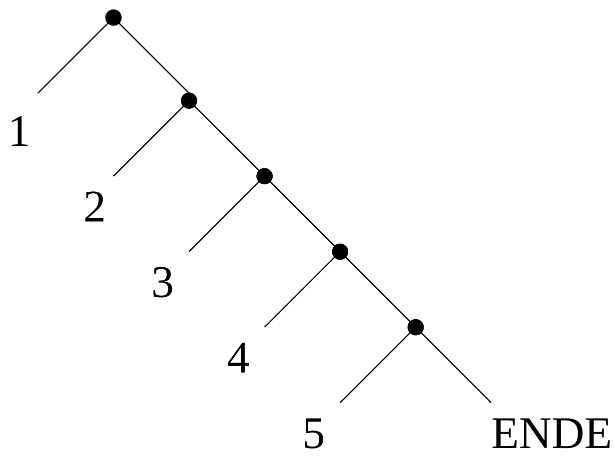

Abbildung 2: Listen als Binärbäume

her ist das ähnlich wie in C: dort steht die Zahl 0 (bzw. der Nullpointer) für "falsch", alles andere für "wahr". (Witzig ist in diesem Zusammenhang auch das Wahrheitskonzept von Perl. Aber wer darüber was wissen will, sollte z.B. in den Perl-Kurs[43]von einem der Kursleiter (T.F.) schauen.) Es gibt noch einen weiteren speziellen Wert, der in LISP verwendet werden kann, um "einfach nur Wahrheit" auszudrücken, nämlich `T`. Paare drücken wir mit runden Klammern aus, wobei das Trennzeichen ein Punkt ist: das Paar aus den Zahlen 5 und 3 wäre z.B. `(5 . 3)`. Wenn wir rekursiv Paare paaren, können wir auch beliebige binäre Bäume bauen. Eine Liste ist nun nichts anderes als ein Kamm-artiger Baum:

Als End-Wert dient in LISP `NIL`. `NIL` drückt damit in gleicher Weise Falschheit wie auch die leere Liste aus. (`NIL` soll übrigens für "nihil" stehen.) Obige Liste wäre in Paar-Notation

```
(1 . (2 . (3 . (4 . (5 . NIL)))))
```

Weil das aber schnell schwerfällig wird, führen wir eine abkürzende Schreibweise ein: wenn auf einen Punkt eine öffnende Klammer folgt, dürfen wir den Punkt und das entsprechende Klammerpaar weglassen:

```
 (1 . (2 . (3 . (4 . (5 . NIL)))))
```

können). In vielen Implementierungen ist es jedoch so, daß `#t` eine Konstante ist, `#f` zu `()` auswertet und die `LISP`-Wahrheitswertkonvention gilt.

Man beachte, daß dies zwar eine korrekte Implementierung der `Scheme`-Spezifikation ist, nicht die einzige und daß deshalb Programme, die sich darauf verlassen zwar wunderbar funktionieren, aber *nicht* korrekt sind. (Wer sich für mehr von dem Kaliber "funktioniert, aber nicht korrekt" interessiert, der betrachte lokale Variablen und die calling-convention in `Fortran` und überlege sich, welchen Unfug man damit anstellen kann. . .)

[43] `http://www.cip.physik.uni-muenchen.de/~tf/perl/`

```
=  (1 2 . (3 . (4 . (5 . NIL))))

= (1 2 3 4 5 . NIL)

= (1 . (2 3 4 5 . NIL))

= {...}
```

Zusätzlich vereinbaren wir, daß an jeder Stelle, an der "`. NIL)`" auftaucht, der Punkt und das NIL weggelassen werden dürfen. Damit haben wir:

```
(1 . (2 . (3 . (4 . (5 . NIL)))))

=  (1 2 3 4 5)
```

Und das ist die übliche Notation, in der LISP-Systeme Listen auch ausgeben. Wir können natürlich in die andere Richtung an jeder Stelle mitten in der Liste einen Punkt einsetzen und den hinteren Teil der Liste klammern, wenn wir wollen. Oder nach dem letzten Element der Liste "`. NIL`" vor der schließenden Klammer einführen. (Weil das aus einer Liste in üblicher Listennotation aber eine Liste in unüblicher Notation macht, und wir `. NIL` nur am Ende von Listen in üblicher Listennotation einsetzen dürfen, können wir das nicht rekursiv machen. Folgendes ist Quatsch: `(1 . NIL . NIL)`. Diese Syntaxregeln sind etwas gewöhnungsbedürftig, aber wenn man ein wenig Übung damit hat, durchaus nicht schwer zu handhaben. Letztendlich: es sind nur Syntaxregeln!

Kommen wir nun zur ersten genialen Idee: wir *könnten* jetzt eine Syntax definieren, die zu Ablaufbeschreibungen z.B. in der folgenden Art führt:

```
length NIL = NIL
length (a . b) = 1 + length b
```

Ein Maschinen-LISP-System würde dann diese Syntax erst intern in einen Syntaxbaum parsen und daraus ein lauffähiges Programm erzeugen. Wenn wir ganz genau hinsehen, ist das eigentlich ein Umweg, den wir uns im Prinzip auch sparen könnten: eigentlich könnten wir den Syntxbaum auch direkt eingeben. Eine Notation für Bäume haben wir ja inzwischen, weil wir eine rekursiv verwendbare Notation für Paare haben. Die Idee ist allein schon deswegen bemerkenswert, weil viele Programmiersprachen, mit denen manche Leute meinen uns von Zeit zu Zeit immer mal wieder beglücken zu müssen, sich durch nicht viel mehr als die Syntax unterscheiden. Wenn wir auf diese Weise die Syntax komplett töten, haben wir sehr viel Willkür aus der Definition einer Sprache herausgeworfen. Willkür und "komische Konstrukte" erfüllen allerdings in zwischenmenschlicher Kommunikation eine durchaus wichtige Rolle, unter anderem dadurch, daß sie Redundanz und Abwechslung in die Beschreibung bringen. (Es gibt auch eine

Programmiersprache, die sich diesem Ansatz verschrieben hat: Perl.) Wir sollten erwarten, daß unsere Programme wie monotone Wüsten aussehen, wenn wir uns auf eine derartige Konvention einlassen. Das Geheimnis ist hier aber ein ganz Einfaches (man muß es nur mal gesagt bekommen): LISP-Programme werden anhand der Einrückung gelesen und unter Zuhilfenahme eines Editors geschrieben, der Parentheses-Matching und Subexpression Highlighting beherrscht. Der Trick ist also, daß man weder beim Schreiben noch beim Lesen auf die Klammern schaut, und nur dran denkt, daß "die Klammern wohl schon so gesetzt sind, daß die Einrückung paßt"[44].

Im Lambda-Kalkül haben wir Funktionen mit mehreren Argumenten durch Currying erhalten, und Haskell erbt das auf sehr direkte Weise. In LISP ist das anders. Currying ist hier zwar auch möglich, hat aber längst nicht den Stellenwert, der vielleicht angemessen wäre. Dafür gibt es Funktionen, die echt mehrere Parameter besitzen, manche sogar eine variable Zahl davon. Die LISP-Notation für die Auswertung einer Funktion sieht so aus:

```
(atom 2)
```

Wenn diese Liste ausgewertet wird, liefert sie den Wert `T`. (`atom` ist eine LISP-Funktion, die prüft, ob ihr Argument ein Atom ist.)

LISP funktioniert nun so: *Wenn eine Liste ausgewertet werden soll, wird der Kopf als Liste als Funktion betrachtet, in die der Reihe nach die Ergebnisse der Auswertung der restlichen Listenelemente eingesetzt werden.*

Wenn wir uns obige Liste scharf ansehen, die wir offenbar auch genausogut `(atom . (2 . NIL))` schreiben könnten, fallen einige Dinge auf. Erstens: was ist `atom` eigentlich? Es ist ein Element einer Liste, und es scheint keine offensichtliche Unterstruktur zu haben, also wird `atom` wohl ein Atom sein. Aber es ist nicht String, nicht Zahl, weder Fisch noch Fleisch. `atom` ist der Name der Funktion, die prüft, ob ihr Argument ein Atom ist. Weil `atom` für einen Wert - hier eine Funktion - steht, nennen wir es ein *Symbol*. An dieser Stelle ist es wohl angebracht, zu erwähnen, daß LISP normalerweise, wenn man nichts anderes vereinbart, bei Symbolen nicht auf Groß- und Kleinschreibung achtet. Weiterhin erlaubt LISP bei Symbolnamen so gut wie alles, was andere Programmiersprachen nie und nimmer durchgehen lassen würden. Beispielsweise darf ein Symbolname auch "`+`" lauten. Naheliegenderweise steht das Symbol "`+`" übrigens für die Funktion, die Zahlen addiert. Wir haben hier das Beispiel einer Funktion, die eine beliebige Zahl von Argumenten erlaubt:

```
(+ 1 2) => 3
```

[44] Hier ist etwas Vorsicht angebracht: Zumindest beim Schreiben ist man erstaunlich (erschreckend?) schnell in der Lage, über fünf bis sieben Klammerebenen hinweg zu wissen welche Klammer man denn nun gerade schließt und wie es weitergeht. Wenn man dabei mit dem "falschen" Texteditor arbeitet dann ist zwar das Programm korrekt, aber nicht die Einrückung! (Denn (zumindest aus der Sicht der Kursleiter: glücklicherweise) sind in `LISP` Einrückungen irrelevant.)

```
(+ 1 2 3) => 6

(+ 2) => 2

(+) => 0
```

Ein noch wesentlich subtilerer Punkt: wenn wir den Ausdruck `(atom 2)` auswerten, so ergibt sich sein Wert zu `T`. Wir haben aber gesagt, daß für die Auswertung einer Funktion die Funktionsargumente ausgewertet werden müssen. Funktionsargument ist hier der Wert 2. Zur Auswertung muß der Wert des Werts 2 bestimmt werden. Das ist gerade 2. Was soll das ganze? Nun,

```
(atom (+ 2 3)) => T
```

Das Argument von `atom` ist die Liste `(+ 2 3)`. Der *Wert* dieser Liste ist nach unseren Auswertungsregeln `5`. Von diesem Wert des Funktionsarguments wird geprüft, ob es ein Atom ist, *nicht* von der Liste `(+ 2 3)`. Wenn wir folgendes versuchen:

```
(atom (1 2 3))
```

meckert das System, `1` sei keine Funktion. Es versucht nämlich, den Wert der Liste `(1 2 3)` zu bestimmen, und dafür muß es die Funktion `1` auf die Elemente im Listenschwanz anwenden. Wenn wir's uns überlegen: jede Liste, die wir eingeben, wird hier irgendwann ausgewertet, d.h. letztendlich als Programmängesehen. Wie können wir nun aber Listen als Daten verwenden? Eine Funktion, die aus ihren Argumenten eine Liste baut, wäre sicher sehr hilfreich. Um bei den Elementarbausteinen zu bleiben, interessieren wir uns aber erst mal mehr für eine Funktion, die aus zwei Werten ein Paar macht. In LISP nennt man ein Paar ein *CONS* (bzw. wenn man die Darstellung im Speicher meint, eine CONS-Zelle). Das linke Element eines Paares heißt in LISP *CAR*, das rechte Element *CDR*.[45] Die Funktionen, um Paare zu erzeugen, bzw. das linke oder rechte Element zu extrahieren, heißen dementsprechend in LISP `CONS`, `CAR`, `CDR`. Das, was wir ursprünglich wollten, können wir nun z.B. so schreiben:

```
(atom (cons 1 (cons 2 (cons 3 nil)))) => NIL
```

Natürlich steht *CONS* auf unterster Stufe der Nahrungskette. Ein erfahrener LISP-Programmierer würde ganz andere Funktionen, bzw. eine ganz andere Notation verwenden, aber das würde hier zu weit führen. Wir möchten nur die wesentlichen Ideen präsentieren.

[45] Diese Namen kommen von den IBM 704 Assembler-Anweisungen "CAR" = "Contents of Address Register", "CDR" = "Contents of Decrement Register". Die 704 hatte 36-Bit-Register, die als 2×18-Bit angesprochen werden konnten, eben CAR und CDR.

Wer selbst ein wenig programmiert, dürfte vielleicht ein wenig mit der Idee herumspielen, wie sich ein LISP-System implementieren lassen könnte und zu dem Schluß kommen, daß nach allem was ich hier über die Funktionsweise erzähle LISP notwendigerweise ganz hoffnungslos ineffizient sein muß. Hierzu sei gesagt, daß LISP unter der Haube in einigen Punkten *ganz* anders arbeitet als man zunächst meinen könnte. Trotzdem verhält es sich für den Programmierer so, wie es hier erklärt wurde. Leider können wir hier nur nicht zu tief in teils sehr involvierte technische Details einsteigen. Vielleicht nur soviel: einige Features wie z.B. LISPs Kompromißlosigkeit in Puncto Korrektheit machen die Sprache viel langsamer als z.B. `C`, etwa weil bei der Addition von zwei Zahlen immer nachgesehen werden muß, ob die festverdrahtete Additionslogik ausreicht, oder mehr Arbeit nötig ist. Es wäre denkbar, spezielle LISP-Hardware zu bauen, die derartige Overheads praktisch völlig verschwinden läßt, aber auf herkömmlichen Prozessoren wird dies zu einem Problem. Trotzdem erlauben es sehr viele LISP-Systeme dem Programmierer, explizit Information darüber mitzuliefern, wo z.B. einfache Hardware-Addition mit Sicherheit immer ausreicht, und mit solchen Zusatzinformationen können manche LISP-Systeme Code generieren, der sich nicht hinter compiliertem `C` verstecken muß.

Weiterhin können wir in LISP auch Funktionen definieren. An dieser Stelle soll allerdings auf den Dialekt `Scheme` gewechselt werden. Das meiste, was bisher erzählt wurde, gilt auch für Scheme, mit wenigen Ausnahmen: beispielsweise heisst die `atom`-Funktion in Scheme `atom?`, `NIL` steht in Scheme nicht für 'falsch'. Weiterhin ist Scheme case-sensitive, und `NIL` muß man explizit als `'()` schreiben. Man könnte sagen, daß Scheme eine modernere, entschlackte, vereinfachte Variante von LISP darstellt. Ein inzwischen sehr weit verbreitetes Scheme-System ist Guile von der FSF; leider ist das "nur" ein reines Interpreter-System, aber eins mit Potential.[46] Von den Common LISP-Systemen sind CLISP und CMU CL ganz sicher einen Blick wert.

Um einen kleinen Eindruck von Scheme zu vermitteln, seien hier nur kurz die oben erwähnten Listenfunktionen in Scheme umgesetzt[47]

```
(define (my-length li)
  (letrec
      ((traverse
        (lambda (count rest)
          (if (null? rest)
              count
              (traverse (+ 1 count) (cdr rest))))))
    (traverse 0 li)))
```

[46]Einer der Autoren ist ein grosser Fan von Marc Feelys Compiler/Interpretersystem 'gambit'.

[47]Dem aufmerksamen Leser wird auffallen, daß diese Umsetzung alles andere als "eins-zu-eins" ist. Das liegt daran, daß Scheme — im Gegensatz zu Haskell eben *nicht* faul ist, sondern alle ihm aufgetragenen Aufgaben sofort und bis zum bitteren Ende (oder der Divergenz) erledigt. Damit heißt das Zauberwort zunächst eben nicht "guarded by constructor" sondern "tail recursive"...

```
(define (my-reverse li)
  (letrec
      ((traverse
        (lambda  (so-far rest)
          (if (null? rest)
              so-far
              (traverse (cons (car rest) so-far) (cdr rest))))))
    (traverse '() li)))

(define (my-map f li)
  (letrec
      ((traverse
        (lambda (so-far rest)
          (if (null? rest)
              (my-reverse so-far)
              (traverse (cons (f (car rest)) so-far) (cdr rest))))))
    (traverse '() li)))

(define (my-take n li)
  (letrec
      ((traverse
        (lambda (n-todo so-far rest)
          (if (or (null? rest) (= 0 n-todo))
              (my-reverse so-far)
              (traverse (- n-todo 1) (cons (car rest) so-far) (cdr rest))))))
    (traverse n '() li)))
```

Wenn man genau hinguckt, merkt man, daß die oben erklärten Evaluationsregeln offenbar noch nicht alles sein können, denn es gibt anscheinend Sonderformen, die sich nicht wie Funktionen verhalten, wie etwa `define`. Das ist in der Tat richtig.

Das obige Beispiel läßt uns auch schon ein wenig erahnen, wie der Lambda-Kalkül in Scheme (bzw. LISP) eingebettet ist: es gibt eine spezielle Sonderform `lambda`, mit der wir anonyme Funktionen erzeugen können. Leider haben wir hier das Problem des fehlenden automatischen Curryings, weswegen wir z.B. S explizit schreiben müssen als `(lambda (x) (lambda (y) (lambda (z) ((x z) (y z)))))`. Damit gilt beispielsweise:

```
 (((lambda (x) (lambda (y) x)) 42) 100) => 42
```

Bis jetzt haben wir LISP bzw. Scheme nur als abstrakte Notation zur Formalisierung von allgemeinen Berechnungen kennengelernt. Ulkigerweise ist die Auswertung eines LISP-Ausdrucks aber gerade nichts anderes als eine streng formalisierbare Berechnung. Wir können also für den Auswertungsprozeß in LISP eine Definition in LISP geben. Diese Idee, einen Evaluator in der Sprache zu verfassen, die er auswerten soll, ist das, was man *Metazirkularität* nennt.

Schön und gut - bis jetzt sieht das ganze allerdings noch nicht sonderlich berauschend aus. Scheme präsentiert sich uns als eine Art minimales System, das gerade mal ausreicht, um damit zu programmieren, und einige nette Features wie z.B. anonyme Funktionen mitbringt. Auch in Scheme können Funktionen als Werte wie alle anderen auch aufgefaßt werden. (Für Common LISP gilt im Prinzip dasselbe, allerdings bringt Common LISP *wesentlich* mehr Infrastruktur mit was eingebaute Datentypen angeht; dafür stecken in LISP noch einige antiquierte konzeptionell eigentlich falsche, aber wenn man sie mal gemeistert hat nicht allzusehr störende Ansätze drin. Wie beispielsweise die Unterscheidung zwischen `symbol-value` und `symbol-function`.) Was soll daran nun so toll sein? Nun, Scheme (beziehungsweise LISP) ist eine *programmierbare Programmiersprache*. Der Programmierer kann dieses Basissystem nach Belieben erweitern und beispielsweise problemlos neue Sprachsyntax definieren, falls er das für notwendig erachtet. Während C ein Meißel ist, mit dem ich aus einem Marmorblock in mühevoller Kleinarbeit ein Kunstwerk herausschlagen kann, verhält sich LISP mehr wie ein Batzen Knetmasse, den man formen, aber auch sehr leicht wieder umformen kann, und dem man die gewünschte Gestalt geben kann. LISP ist zunächst ein Embryo, der nur die notwendigen Grundfunktionen mitbringt, und der den Programmierer auffordert, ihn in die gewünschte Richtung weiterzuentwickeln.

Das Interessante ist, daß die Features, die ein LISP-System mitbringt, in gewisser Weise minimal, orthogonal und vollständig sind. Jedes größere Programm wird, wenn es eine gewisse Komplexität erreicht hat, beispielsweise die Möglichkeit zur flexiblen Erweiterung benötigen. Oder aber eine dynamische Speicherverwaltung. (Wer jemals in C oder C++ in die Verlegenheit gekommen ist, Speicherlecks zu suchen und zu debuggen, sollte sich definitiv schlau machen, ob er nicht was Grundsätzliches falsch macht, und ob ein LISP-System nicht eine wesentlich günstigere Ausgangsbasis wäre.) Es gibt deswegen den etwas zynischen Spruch, daß jedes typische große Programm ein undokumentiertes, schlecht implementiertes, stellenweise fehlerhaftes, von Grund auf ad-hoc neu geschaffenes LISP-System enthält. So unwahr ist das nicht.

Auch wenn hierzu kein Beispiel gebracht werden soll, so sei dennoch etwas näher erklärt, was damit gemeint ist, die Sprache selbst sei "programmierbar". Man kann in C – in sehr eingeschränktem Maße allerdings – die Syntax der Sprache durch Verwendung des Präprozessors erweitern. Der Präprozessor baut vor der eigentlichen Compilationsstufe den Quelltext nochmal um, bevor ihn der Compiler sieht. Man kann sich nun auf den Standpunkt stellen, den Bjarne Stroustrup in "The C++ Programming Language" vertritt und behaupten, daß eine Sprache, die erst noch umgebaut werden muß, bevor sie der Compiler sieht, eigentlich ganz grundlegende Probleme hat. (Solche Transkriptionsschritte interagieren erfahrungsgemäß auch immer sehr schlecht mit Debuggern, weil man beim Debuggen z.B. schnell Code zu Gesicht bekommt, den man so niemals geschrieben hat.) Konsequenterweise gibt es in C++ für alle Anwendungen, für die man `#define` in C einsetzt (mit der Ausnahme, andere Abschnitte wie insbesondere `#include`-Direktiven zu steuern) sinnvolle Alternativen. Solche Code-

Ersetzungsregeln sind keine Funktionen, es sind Makros. (In Haskell allerdings verwischen die Grenzen durch die faule Auswertung ein wenig.) Man kann die Makro-Programmierung an sich zu einer Kunstform erheben, wenn man das wirklich will. TeX ist ein gutes Beispiel für ein rein makrogesteuertes System. Makros haben etwas Böses an sich, und es gibt sehr gute Gründe, sie äußerst sparsam einzusetzen und beispielsweise etwas, was konzeptionell eine Funktion ist, niemals als Makro zu realisieren. Funktionen geben unserer Sprache neue Wörter, Makros geben ihr neue Syntax.[48]

Wenn LISP eine programmierbare Programmiersprache ist, werden wir wohl davon ausgehen müssen, daß es auch hier so was wie einen "Präprozessor" gibt, der Code erst einmal umbaut, bevor er ausgewertet wird. Wenn eine Funktion compiliert wird (viele LISP-Systeme enthalten Compiler, entweder nur Bytecode-Compiler, oder aber richtige Maschinensprache generierende Compiler), findet Makroexpansion komplett zur Compilierzeit statt. Wird eine Funktion nur interpretiert, können Makros auch "on the fly expandiert" werden. Das ist begrifflich deswegen wichtig, weil wir auch mit Makros noch lange nicht bei der Grausamkeit, vor der sich jeder Programmierer hüten sollte - selbstmodifizierendem Code - angelangt sind.

Der Präprozessor von LISP ist natürlich - wie könnte es auch anders sein - LISP. Programmcode besteht aus Listen (bzw. Bäumen), und LISP ist das ideale Werkzeug, um auf solchen Bäumen zu arbeiten, bzw. sie zu erzeugen. Ein LISP-Makro ist also nichts anderes als LISP-Code, der LISP-Code schreibt.

Als Beispiel sei kurz ein sehr nützliches Makro vorgestellt, das einer der Autoren häufig verwendet; es erweitert LISP um einen sehr nützlichen Begriff, der eine Denkweise ermöglicht, die in der Perl-Community weitverbreitet ist. Wie Perl auch kennt LISP Hash-Tabellen. Hashes werden unter anderem verwendet, um Dinge systematisch und effizient zu klassifizieren oder zu akkumulieren. Beispielsweise gibt der folgende Mini-Perl-Hack eine Wortstatistik über einen Text aus:

```
tf@ouija: ~/cde/lambda$ perl -e 'while(<>){$w{$_}++ for /\b([a-zA-Z]+)\b/g};
printf "%-20s => %10d\n", $_, $w{$_} for sort {$w{$b}&lt;=>$w{$a}} keys %w;'
 not_quite_lambda.html
```

Hier ein LISP-Makro, das Hash-Inkrementierung implementiert, wobei 'nicht-vorhanden' gleichbedeutend zu null interpretiert wird:

```
(defmacro hv-inc* (hash pos &optional (increment 1)
                              &key addition is-zero)
```

[48] Hier sei aber das Makro-Konzept von TeX explizit in Schutz genommen. Makros haben nämlich nur dann etwas böses, wenn man ohnehin schon aus dem Paradies vertrieben ist und einen Zustandsbegriff hat (den hat man in TeX zwar, aber sehr behutsam). Wenn man also keine Angst vor der Verdopplung von Seiteneffekten haben muß und die *richtige* (meint: faule, also links-außen) *Auswertestrategie* hat, dann kann man mit Makros durchaus funktional Programmieren. Wir erinnern uns an den Abschnitt über Kombinatoren.

```
  (let ((sym-old-entry (gensym "hv-inc*-old-entry"))
        (sym-new-val (gensym "hv-inc*-new-val"))
        (add-call (if addition `(funcall ,addition) `(+)))
        (is-zero-call (if is-zero `(funcall ,is-zero) `(eql 0))))
    `(let* ((,sym-old-entry (gethash ,pos ,hash))
            (,sym-new-val
             (,@add-call
               (if ,sym-old-entry ,sym-old-entry 0)
               ,increment)))
       (if (,@is-zero-call ,sym-new-val)
           (progn
             (remhash ,pos ,hash)
             0)
         (setf (gethash ,pos ,hash) ,sym-new-val)))))
```

Zum Abschluß noch eine Bemerkung zu dem Vorurteil, LISP sei langsam. Wir wollen auf einfache, direkte Weise eine Approximation an die Mandelbrotmenge ausrechnen. Beginnen wir mit `C`. Ein wenig Komfortabilität will natürlich schon sein, und so ist dieses Beispiel vielleicht auch ganz lehrreich was die Verwendung von `getopt(3)` angeht.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <getopt.h>
/* Note: getopt.h is a GNU extension */

extern char *optarg;

/* defaults */

static int iter=100;
static int width=70;
static int height=20;
static double lefttop_r=-2.0;
static double lefttop_i=-1.5;
static double rightbottom_r=1.0;
static double rightbottom_i=1.5;

static char *periodic_densities=" .oOxX";
static char max_density='#';


static struct option opts[]=
{
{"width", 1, 0, 'w'},
{"height", 1, 0, 'h'},
```

```
{"iter", 1, 0, 'i'},
{"lefttop-real", 1, 0, 'l'},
{"lefttop-imag", 1, 0, 'L'},
{"rightbottom-real", 1, 0, 'r'},
{"rightbottom-imag", 1, 0, 'R'}
};


static void bailout_show_help(void)
{
        fprintf(stderr,
" mandelbrot - simple program to compute mandelbrot set\n\
\n\
Usage: mandelbrot [--width 70] [--height 20]\n\
[--lefttop-real -2.0] [--lefttop-imag -2.0]\n\
[--rightbottom-real +2.0] [--rightbottom-imag +2.0]\n\
[--iter 100]\n\n"
                        );
        exit(1);
}

void do_mandel(void)
{
 int i, n, x,y;
 double zstep_r, zstep_i;
 double z_r, z_i, next_z_r, next_z_i;
 double pos_r, pos_i;
 char *field=alloca((1+width)*height+1);
 char **fpointers=alloca(height*sizeof(char*));
 int nr_densities;

 nr_densities=strlen(periodic_densities);

 /* init field for fast printing */
 for(i=0;i&lt;height;i++)
    {
     field[width+(1+width)*i]='\n';
     fpointers[i]=&field[i*(1+width)];
     }
 field[(1+width)*height]=0;


 zstep_r=(rightbottom_r-lefttop_r)/(width-1);
 zstep_i=(rightbottom_i-lefttop_i)/(height-1);

 for(x=0;x&lt;width;x++)
    {
     pos_r=lefttop_r+x*zstep_r;

     for(y=0;y&lt;height;y++)
```

```
        {
         pos_i=lefttop_i+y*zstep_i;

         z_r=z_i=0;

         for(i=0;i&lt;iter;i++)
            {
             next_z_r=z_r*z_r-z_i*z_i+pos_r;
             next_z_i=2*z_r*z_i+pos_i;

             z_r=next_z_r;
             z_i=next_z_i;
             if(z_r*z_r+z_i*z_i>4)
               {
                fpointers[y][x]=periodic_densities[i%nr_densities];
                i=iter+1;
               }
             }
         if(i==iter)
           {
            fpointers[y][x]=max_density;
           }
        }
    }
 printf("%s", field);
}

int main(int argc, char *argv[])
{
 int opt_ret;

 while(-1!=(opt_ret=getopt_long(argc, argv, "", opts, 0)))
      {
       if(opt_ret==':' || opt_ret=='?')
         {
          bailout_show_help();
         }
       switch(opt_ret)
       {
        case 'i':
         iter=atoi(optarg);
         break;
        case 'w':
         width=atoi(optarg);
         break;
        case 'h':
         height=atoi(optarg);
         break;
        case 'l':
         lefttop_r=atof(optarg);
```

```
          break;
         case 'L':
          lefttop_i=atof(optarg);
          break;
         case 'r':
          rightbottom_r=atof(optarg);
          break;
         case 'R':
          rightbottom_r=atof(optarg);
          break;
        }

        if(width<=1 || height<=1) /* other insane parameters are harmless */
          {
           fprintf(stderr,
 "Please specify valid width and height parameters.\n");
           exit(1);
          }
       }
 do_mandel();
 return 0;
}
```

LISP hat es hier sehr schwer: es handelt sich expliziz um ein stark numeriklastiges Beispiel, das noch dazu in einer engen, einfachen Schleife rechnet. Der Evolutionsdruck auf C-Compilern sollte so groß sein, daß sie hier fast das Optimum herausholen können. Numerik war nie LISPs wirkliche Stärke, und so werden wir von all dem, was LISP so cool macht, praktisch nichts wirklich anwenden können. In optimiertem Common LISP sieht unser Code so aus:

```
(defun mandelbrot (&key
                   (iter 100)
                   (width 70)
                   (height 20)
                   (lefttop-real -2.0d0)
                   (lefttop-imag -1.5d0)
                   (rightbottom-real 1.0d0)
                   (rightbottom-imag 1.5d0)
                   (densities " .oOxX")
                   (max-density #\#))
  (declare (fixnum iter width height)
           (double-float lefttop-real lefttop-imag
                         rightbottom-real rightbottom-imag)
           (string densities)
           (base-char max-density)
           (optimize (safety 0) (speed 3)))
  (macrolet
      ((i+ (&rest args)
           `(the fixnum
```

```
          (+ ,@(mapcar #'(lambda (x) `(the fixnum ,x)) args))))
   (f+ (&rest args)
       `(the double-float
          (+ ,@(mapcar #'(lambda (x) `(the double-float ,x)) args))))
   (f* (&rest args)
       `(the double-float
          (* ,@(mapcar #'(lambda (x) `(the double-float ,x)) args))))
   (f- (&rest args)
       `(the double-float
          (- ,@(mapcar #'(lambda (x) `(the double-float ,x)) args))))
   (f> (&rest args)
       `(> ,@(mapcar #'(lambda (x) `(the double-float ,x)) args))))
;;
(let ((field (make-array height))
      (zstep-r (/ (f- rightbottom-real lefttop-real) (1+ width)))
      (zstep-i (/ (f- rightbottom-imag lefttop-imag) (1+ height)))
      (pos-r 0.0d0)
      (pos-i 0.0d0)
      (nr-densities (length densities)))
  (declare
   (fixnum nr-densities)
   (double-float zstep-r zstep-i pos-r pos-i))
  (dotimes (j height)
    (declare (fixnum j))
    (setf (svref field j) (make-string width)))
  (dotimes (x width)
    (declare (fixnum x))
    (setf pos-r (f+ lefttop-real
                    (f* (coerce x 'double-float) zstep-r)))
    (dotimes (y height)
      (declare (fixnum y))
      (setf pos-i (f+ lefttop-imag
                      (f* (coerce y 'double-float) zstep-i)))
      (labels
          ((iter (z-r z-i n-iter)
                 (declare (fixnum n-iter) (double-float z-r z-i))
                 (if (= n-iter iter)
                     (setf (schar (aref field y) x) max-density)
                   (if (f> (f+ (f* z-r z-r) (f* z-i z-i)) 4.0d0)
                       (setf (schar (aref field y) x)
                             (schar densities (mod n-iter nr-densities)))
                     (iter (f+ pos-r (f* z-r z-r) (f- (f* z-i z-i)))
                           (f+ pos-i (f* 2.0d0 z-r z-i))
                           (i+ 1 n-iter))))))
        (iter 0.0d0 0.0d0 0))))
  field)))
```

Werfen wir vielleicht auch mal einen Blick auf den Output (hier des C-Compilats. Der LISP-Code berechnet ein Array von Strings. Gerechnet wurde mit maximal 1 000 000 Iterationen.)

```
......................................................
..........................................................
..........oooooooooooooooooooooooooooooooo...................
......oooooooooooooooooooOOOOOOxxxXXx   OOOOOooooo.............
....ooooooooooooooooooOOOOOOOOxxxX OX.#Xxo xxxOOOOOooooo.........
..oooooooooooooooooOOOOOOOxx  .  .oOXX#### o. XXXX xOOoooooo......
..ooooooooooooooOOOxxxxxxxXX .o.#################X##oOXxOOoooooo....
.oooooooOOOOOxx X.   ..    ..XX######################XxOXxOOooooooo..
ooOOOOOOOxxxxX  ooo#O### o.XO######################### oXxOOOooooooo.
OxxOOXXXX   oo...####################################X.XxxOOOooooooo.
OxxOOXXXX   oo...####################################X.XxxOOOooooooo.
ooOOOOOOOxxxxX  ooo#O### o.XO######################### oXxOOOooooooo.
.oooooooOOOOOxx X.   ..    ..XX######################XxOXxOOooooooo..
..ooooooooooooooOOOxxxxxxxXX .o.#################X##oOXxOOoooooo....
..oooooooooooooooooOOOOOOOxx  .  .oOXX#### o. XXXX xOOoooooo......
....ooooooooooooooooooOOOOOOOOxxxX OX.#Xxo xxxOOOOOooooo.........
......oooooooooooooooooooOOOOOOxxxXXx   OOOOOooooo.............
..........oooooooooooooooooooooooooooooooo...................
..........................................................
......................................................
```

Let's do statistics[49]:

Quellcode-Größe	C	LISP
Codezeilen	147	56
Zeichen	2857	2102

Sehen wir uns die Performance verschiedener Systeme an. Die Hardware ist übrigens ein ASUS F7400 Notebook mit 366 MHz Intel Celeron (Mendocino-Kern).

System	Iterationen	Parameter	Laufzeit (real)
CLISP 2000-03-06	1 000	uncompiliert	60.06 s
CLISP 2000-03-06	1 000	compiliert (bytecode), optimize (speed 0)	3.57 s
CLISP 2000-03-06	1 000	compiliert (bytecode), optimize (speed 1)	3.51 s
CLISP 2000-03-06	1 000	compiliert (bytecode), optimize (speed 2)	3.49 s
CLISP 2000-03-06	1 000	compiliert (bytecode), optimize (speed 3)	3.51 s
CMU Common LISP x86-Linux	1 000	uncompiliert, gc size 40 000 000 bytes	28.81 s
CMU Common LISP x86-Linux 2.4.19	1 000 000	compiliert, optimize (speed 0)	95.64 s
CMU Common LISP x86-Linux 2.4.19	1 000 000	compiliert, optimize (speed 1)	92.67 s
CMU Common LISP x86-Linux 2.4.19	1 000 000	compiliert, optimize (speed 2)	92.64 s
CMU Common LISP x86-Linux 2.4.19	1 000 000	compiliert, optimize (speed 3)	25.78 s
GCC 2.95.2	1 000 000		86.735 s
GCC 2.95.2	1 000 000	-O1 Optimierung	17.064 s
GCC 2.95.2	1 000 000	-O2 Optimierung	16.393 s
GCC 2.95.2	1 000 000	-O3 Optimierung	16.840 s

Natürlich sind solche Vergleiche immer sehr mit Vorsicht zu genießen. Dennoch zeigt das schon ungefähr, wie die verschiedenen Systeme einzustufen sind, wenn es allein um rohe Performance und sonst nichts geht. CMU CL kann hier Code erzeugen, der gerade mal 60% weniger effizient ist als (gcc) C-Compilat, und das, obwohl wir es hier mit einer Anwendung zu tun haben, die wie gesagt sicher

[49] Da der Quelltext für die gedruckte Darstellung umformatiert werden mußte, stimmen die Quelltextstatistiken nicht genau mit dem angegebenen Code überein.

nicht zu den großen Stärken von LISP zählt. Wir sprechen hier also nicht mal über einen Faktor 2. Und das sollte wirklich in den allermeisten Fällen zu verschmerzen sein. Als letzte Maßnahme ist es immer noch möglich, C-Funktionen aus LISP heraus aufzurufen.

Eine letzte Anmerkung noch hierzu: COMMON LISP kennt gemäs̈s Sprachstandard auch komplexe Zahlen, und in der Tat kann CMU CL mit den richtigen Typinformationen dafür auch effizienten Maschinencode generieren. Dennoch wurde hier stattdessen der C-Algorithmus beinahe 1:1 übertragen. Dieses Beispiel soll *lediglich* demonstrieren, das̈s eine getreue Übertragung eines Algorithmus in C nach LISP nicht notwendigerweise mit unakzeptablen Performanceverlusten einhergeht.

8 “More Magic”: Faule Tricks

Nachdem wir in der vorletzten Lektion einige Einblicke erhalten haben, was für unglaubliche Dinge mit Kombinatoren möglich sind, und uns in der letzten Lektion mit konkreten Realisierungen beschäftigt haben, wollen wir in dieser Lektion noch einmal zu den irrsinnigen Tricks zurückkehren. Das Schöne hierbei ist, daß wir jetzt auch in der Tat Werkzeuge haben, um diese und ähnliche Ideen nicht nur mit Lambda-Termen, sondern auch ganz konkret in nützlichen Programmen umzusetzen.

Obwohl wir eigentlich nur Funktionen kennen – Kombinatoren – treten einige von ihnen moralisch als Träger von Information auf, die von anderen Funktionen verarbeitet wird. Für die Konstruktion der letzteren haben wir das Fixpunktprinzip als cleveren Trick kennengelernt, um Rekurrenz in jeder beliebigen Form zu erzeugen. (Wir haben nicht explizit etwa über verschränkt-rekursive Funktionen gesprochen, aber die Umformung einer verschränkt-rekursiven in eine normale rekursive Funktion sollte mit ein wenig Nachdenken keine Schwierigkeit darstellen.) Der Lambda-Kalkül an sich führt uns immer wieder vor Augen, daß eine derartige Trennung zwischen Programm und Daten eigentlich ein rein künstliches Konstrukt ist, ein Denkschema, das man verwenden kann, oder auch nicht. In manchen Situationen lassen sich Gedanken klarsten ausdrücken, wenn man sich fest an dieses Prinzip hält, in anderen Situationen dadurch, daß man es mißachtet.

Wenn uns das Fixpunktprinzip erlaubt, rekurrente Funktionen zu erzeugen, können wir es dann nicht auch benutzen, um rekurrente Datenstrukturen zu erzeugen? Ein einfaches (wenn auch nicht sehr nützliches) erstes Beispiel wäre eine unendlich lange Liste, die an jeder Stelle den Wert $\underline{1}$ enthält. Sehen wir uns das etws genauer an: wir suchen einen Kombinator L für den (mit K und KI als üblichen Paar-Selektoren) gilt:

$$LK = \underline{1} \qquad\qquad L(KI) = L$$

Solange wir nur mit K und KI auf einen Kombinator zugreifen der diese Bedingungen erfüllt, und keine anderen speziellen Eigenschaften verwenden, wird er sich wie eine unendlich lange Liste aus “eins”-Einträgen verhalten. Aber wieso sind wir nicht so dreist und versuchen gleich, mit dem P-Kombinator eine derartige Liste zu bauen. Alles, was wir dafür zu tun haben, ist, die Gleichung

$$L = P\underline{1}L$$

zu lösen. Mit unserem jetztigen Wissen über das Fixpunktprinzip (siehe Kapitel 5) ist das nicht das geringste Problem. Wir verwenden einfach

$$L = Y(P\underline{1})$$

Versuchen wir doch einmal, das erste Element dieser Liste zu selektieren:

$$\begin{aligned}
LK &= Y(P\underline{1})K \\
&= P\underline{1}(Y(P\underline{1}))K \\
&= (\lambda abc.cab)\underline{1}(Y(P\underline{1}))K \\
&= K\underline{1}(Y(PK)) \\
&= \underline{1}
\end{aligned}$$

oder das dritte:

$$\begin{aligned}
L(KI)(KI)K &= Y(P\underline{1})(KI)(KI)K \\
&= P\underline{1}L(KI)(KI)K \\
&= (KI)\underline{1}L(KI)K \\
&= L(KI)K \\
&= Y(P\underline{1})(KI)K \\
&= P\underline{1}L(KI)K \\
&= (KI)\underline{1}LK \\
&= LK \\
&= \ldots = \underline{1}
\end{aligned}$$

Wir haben hier also wirklich eine unendlich lange Liste gebaut.

Das mag nun auf den ersten Blick nicht sonderlich aufregend erscheinen - analoges Verhalten kann man auch in einer Programmiersprache wie `C` haben, wenn man sich Paare als geeignete Struktur definiert, und den CDR-Zeigereines Paares auf sich selbst zurückbiegt. Also versuchen wir vielleicht etwas interessanteres: wie wäre es etwa mit einer Liste A aller natürlichen Zahlen? Der naheliegendste Weg dürfte sein, einen "Leiter"-Kombinator L zu definieren, der $\underline{n}$ auf die Liste der natürlichen Zahlen beginnend mit $\underline{n}$ abbildet. Mit dem Nachfolgerkombinator $N = \lambda nsz.s(nsz)$ aus Übung 3.2 sieht das beispielsweise so aus:

$$\begin{aligned}
&L\underline{n} = P\underline{n}(L(N\underline{n})) = SP(BLN)\underline{n} \\
&L = (SP(BLN)) \\
&L = Y(\lambda\ell.SP(B\ell N))
\end{aligned}$$

Damit haben wir also

$$A = Y(\lambda\ell.SP(B\ell N))\underline{0}$$

und das findet ganz sicher keine Entsprechung in einer Programmiersprache wie C. (Nun, in C++ möchte man vielleicht an so Grausamkeiten denken wie das überladen von Zugriffs-Operatoren, womit man auch so was simulieren könnte. Weil dieser ganze Hokus Pokus in C++ aber statisch, d.h. zur Compilierzeit erfolgen muß, ist dem Autor das allenfalls ein müdes Lächeln wert.) Aber in Haskell geht das ohne weiteres - allerdings können wir hier nicht direkt einen Fixpunktkombinator hinschreiben, oder die Rekurrenz manuell auflösen, weil uns das Typsystem hier einen Strich durch die Rechnung macht und sich Haskell um derartige Dinge ausschließlich selbst kümmern will. Aber wir können folgendes machen:

```
numbers_from 0 where numbers_from a = a : numbers_from (a+1)
```

Das sieht nun wirklich merkwürdig aus! Wir haben auf einer endlichen Maschine in endlicher Zeit eine Datenstruktur erschaffen, die gemäß ihrer Definition unendlich groß ist. Der Trick hierbei ist sogenannte "faule Auswertung". (Mit diesem Begriff lassen sich verschiedene Dinge verbinden – hier geht es uns speziell um die mechanistischen Aspekte.) Informell kann man die Idee dahinter wie folgt beschreiben: wenn die Maschine einen Wert X berechnen soll, rechnet sie nicht wirklich X aus, sondern sie liefert stattdessen ein Versprechen, X auszurechnen, wenn der Zeitpunkt gekommen ist, an dem dies absolut nicht mehr zu vermeiden ist. Falls ein- und dasselbe Versprechen an mehreren Stellen auftaucht, reicht es, wenn es einmal eingelöst wurde, um den Wert auch an anderen Stellen sofort zu haben. Wenn man etwas nur lange genug hinausschiebt, muß man es irgendwann vielleicht überhaupt nicht mehr machen – das ist das Grundprinzip hinter allen faulen Methoden. Wie wir speziell hier sehen: Faulheit kann manchmal eine ziemlich geniale Strategie sein, weil sie Dinge möglich macht, die ansonsten undenkbar wären. (Aber auch wenn Faulheit nur unnötige endliche Arbeit spart, kann sie sehr praktisch sein. Alle halbwegs sinnvollen Betriebssysteme enthalten heute an vielen zentralen Stellen faule Ansätze. Beispielsweise wird beim Start eines Programms nur der Teil überhaupt in den Speicher geladen, auf den die CPU unbedingt zugreifen muß – die anderen Teile werden ebenfalls nur bei Bedarf nachgeladen. Auch beim Kopieren von Speicherseiten ist die Kopie anfangs nur ein Verweis auf die ursprüngliche Page. Erst, wenn schreibend drauf zugegriffen wird, wird wirklich kopiert. (Copy-on-write.) Beim Reservieren von Speicher sieht es ganz ähnlich aus.)

Um das ganze noch ein wenig näher auszumalen möchte ich ein sehr cleveres Beispiel eines sehr cleveren Verfahrens aus dem *Wizard Book*[50] vorstellen. Wir werden allerdings für die konkrete Implementierung Haskell verwenden, weil das die Magie hier noch betont.

Wir wollen π berechnen. (Diese spezielle Zahl scheint einen irrationalen(!) Zauber auf viele Menschen auszuüben. (Zumindest einer der Autoren müßte lügen, würde er behaupten, dagegen immun zu sein…)) Viele explizite Verfahren, Näherungswerte für π zu berechnen, sind sehr interessant, weil sie nicht allzu kompliziert sind, aber doch irgendwie einen ganz speziellen, oft sehr lehrreichen Kniff haben. (Eigentlich ist es andersherum: allgemein anwendbare trickreiche Ansätze werden gerne in Form eines Verfahrens, Näherungswerte für π zu berechnen, dargestellt - so auch hier.) Allerdings: wir werden nicht nur Näherungswerte für π berechnen, sondern direkt π! Wie diese dreist klingenden Aussage zu verstehen ist, werden wir noch sehen…

Wie gehen wir vor? Wir verwenden eine Funktion, die an einer rationalen Stelle den Wert π (oder ein rationales Vielfaches) annimmt, und rechnen diese

[50] Abelson/Sussman, "Structure and Interpretation of Computer Programs"; nicht nur deswegen, weil auf dem Cover zwei Zauberer abgebildet sind, in der Community als *Wizard Book* bekannt. Wenn wir nur ein Buch empfehlen sollten, um die Hackerkunst zu lernen, wäre es dieses. Auch online verfügbar unter `http://mitpress.mit.edu/sicp/full-text/book/book.html`

Funktion einfach an der richtigen Stelle aus. Als erstes mag man hier wohl an $\arctan(1) = \pi/4$ denken, und genau das werden wir auch heranziehen. Der Arcustangens (die Umkehrfunktion des Tangens) ist in einer Umgebung der Null analytisch, kann also als Taylorreihe dargestellt werden.

$$\arctan = x \mapsto \sum_{j=0}^{\infty} \frac{(-1)^j}{2j+1} x^{2j+1}$$

Man überlegt sich schnell, daß diese Reihe für $|x| > 1$ nicht konvergieren kann. Wir brauchen hier gerade den Rand des Konvergenzbereichs:

$$\pi/4 = \sum_{j=0}^{\infty} \frac{(-1)^j}{2j+1}$$

Diese Reihe multiplizieren wir noch mit 4. Wir dürfen sie einfach so hinschreiben, weil Konvergenz dadurch garantiert wird, daß die Absolutwerte der alternierenden Reihenglieder eine Nullfolge bilden. Der Versuch, diese Reihe direkt zu benutzen, um π auszurechnen, scheint allerdings ziemlich aberwitzig, denn um π auf 6 Stellen hinter dem Komma genau zu haben, müssen wir größenordnungsmäßig eine Million Terme mitnehmen. Wenn wir einfach mal von Hand losrechnen, um ein wenig Gespür zu entwickeln:

n	Summenglied	Partialsumme exakt	Dezimale Näherung
0	$+4$	4	4
1	$-4/3$	$8/3$	2.6667
2	$+4/5$	$52/15$	3.4667
3	$-4/7$	$304/105$	2.8952
4	$+4/9$	$1052/315$	3.3397

Sonderlich berauschend ist das nicht. (Wir könnten jetzt eine Identität der Art $\arctan(1) = \arctan(1/2) + \arctan(1/5) + \arctan(1/8)$ oder noch ausgefeilter aus dem Hut zaubern und zeigen, daß die entsprechenden hier auftretenden Reihen angenehm schnell konvergieren, aber das wäre irgendwie unsportlich.) Es sieht so aus als ob die Folge der Partialsummen irgendwie im Kern wesentlich gutmütiger ist als sie hier erscheint, nur leider in jedem Schritt viel zu weit übers Ziel hinausschießt. Es scheint plausibel, daß sich bereits aus den ersten Elementen mehr Information über den Grenzwert der Folge rauslesen läßt, wenn es gelingt, die Oszillation zu dämpfen. ('Herauslesen' ist hier allerdings eher im Sinn von Kaffeesatzlesen zu sehen – streng mathematisch betrachtet lassen endliche Partialsummen alternierender Reihen, deren Glieder eine streng monoton fallende Nullfolge bilden, keine Schlüsse zu, die nicht auch allein aus Betrachtung der letzten beiden 'am engsten einschränkenden' Partialsummen gezogen werden könnten.) Einen Versuch ist das sicher wert.

Wir spielen einfach mal drauflos mit der Folge der Partialsummen herum. An jeder Stelle können wir so etwas wie eine aktuelle Schrittweite ermitteln. (Dies sind gerade die Glieder der ursprünglichen Folge, die wir summiert haben.) Wenn wir zwei aufeinanderfolgende Schrittweiten vergleichen, können wir zudem feststellen, wie groß die relative Abnahme der Schrittlängen zwischen aufeinanderfolgenden Schritten ist. Nehmen wir den Quotienten aufeinanderfolgender Schrittlängen als Dämpfungsfaktor, so können wir uns an jeder Stelle in der Partialsummenfolge fragen, welchen Wert wir erhalten würden, wenn alle weiteren Schritte jeweils mit genau diesem Faktor geometrisch gedämpft wären.

So, wie ich das hier formuliert habe, müssen wir an der Stelle n die Elemente S_n, S_n+1 und S_n+2 der Partialsummenfolge ansehen; die für jede Stelle n auf die beschriebene Weise ermittelten "daumengepeilten" Werte bilden eine Folge D_n:

$$D_n = S_n + \frac{(S_{n+1} - S_n)^2}{2S_{n+1} - S_n - S_{n+2}}$$

(Man schreibt diesen Ansatz Euler zu, aber es ist nicht unvorstellbar, daß bereits vorher jemand auf diese Idee gekommen ist. Die Formel, die als Eulers Formel kommentarlos im Abelson/Sussman angegeben ist, sieht etwas anders aus, verhält sich aber genauso. Unsere Version entspricht direkt unseren vorigen überlegungen.)

Eulers Formel liefert eine Abbildung von Folgen auf Folgen. Das schreit gerade danach, mit faulen funktionalen Tricks in Code umgesetzt zu werden. Deswegen geben wir hierzu ein wenig Haskell-Code an:

```
map2 f []  _    = []
map2 f _   []   = []
map2 f (h1:t1) (h2:t2) =(f h1 h2):(map2 f t1 t2)


pi_summands=map (\x->(if odd x then -4.0 else 4.0)
                        /(2.0*((toEnum x)::Double)+1.0))
             [0,1..]

partialsums s = (head s):(map2 (+) (tail s) (partialsums s))

euler (s0:(rest@(s1:s2:s))) =
 (s0+(s1-s0)*(s1-s0)/(2*s1-s0-s2)):(euler_accel rest)
```

Ist das nicht beeindruckend, wie viel Ausdruckskraft sich in wie wenig Code pressen läßt? Mathematiker mögen sich daran stören, daß hier explizit mit häßlichen Gleitkommazahlen gerechnet wird, wo doch exakte Brüche viel schöner wären. Darauf kommen wir später noch zurück.

Es ist ganz interessant, die neue Folge der ursprünglichen gegenüberzustellen:

n	Gleitkommanäherung S_n	Gleitkommanäherung D_n
0	4	3.16667
1	2.66667	3.13333
2	3.46667	3.14524
3	2.89524	3.13968
4	3.33968	3.14271

Das sieht schon mal wesentlich vielversprechender aus. Wenn ein Trick einmal funktioniert hat, funktioniert er vielleicht auch zweimal. Was passiert nun, wenn wir unseren "Konvergenzbeschleuniger" auf die Ergebnisfolge nochmal anwenden? Nennen wir das Bild der Folge X unter dieser Abbildung mal $E(X)$.

n	Gleitkommanäherung S_n	Gleitk. $D_n = E(S)_n$	Gleitk. $E(E(S))_n$
0	4	3.16667	3.14211
1	2.66667	3.13333	3.14145
2	3.46667	3.14524	3.14164
3	2.89524	3.13968	3.14157
4	3.33968	3.14271	3.14160

Es ist vielleicht allerhöchste Zeit mal zu erwähnen, daß die ersten Stellen von π 3.1415926535897932384626 lauten.

Bevor wir jetzt lange weiter in den Niederungen von expliziten Zahlenfolgen rumkriechen, ist es vielleicht mal wieder an der Zeit für einen genialen abstrakten funktionalen Trick. Wir können das obige Schema, das hier aus drei beliebig nach unten fortsetzbaren Zahlenkolonnen besteht, immer weiter um eine neue Kolonne erweitern, in dem wir das Bild der rechtesten unter der Abbildung E berechnen. Dieses Schema läßt sich also auch horizontal beliebig erweitern. Wir tun mal so als wäre es in beide Richtungen unendlich groß. Wenn wir nun die erste Zeile betrachten, so stehen in ihr von links nach rechts immer besser werdende Näherungswerte von π. Und in der Tat werden diese Werte viel schneller besser als in jeder Spalte.

Was haben wir gemacht? Wir haben mit einem offenbar (zumindest hier) nicht-idempotenten Beschleunigungsverfahren ein Tableau aufgebaut. Die Konstruktion des Tableaus und Extraktion der ersten Zeile liefert so was wie ein Super-Beschleunigungsverfahren. Wenn wir nach der ursprünglichen Folge abstrahieren, haben wir hier eine Abbildung eines Beschleunigungsverfahrens auf ein besseres Beschleunigungs-Verfahren. Also einen Beschleunigungsverfahrens-Verbesserer. Das ist mit Sicherheit wert, in Haskell festgehalten zu werden. Wir erweitern also obigen Haskell-Code um *eine* Zeile:

```
map2 f []  _   = []
map2 f _   []  = []
map2 f (h1:t1) (h2:t2) =(f h1 h2):(map2 f t1 t2)


pi_summands=map (\x->(if odd x then -4.0 else 4.0)
                       /(2.0*((toEnum x)::Double)+1.0))
```

```
              [0,1..]

partialsums s = (head s):(map2 (+) (tail s) (partialsums s))

euler (s0:(rest@(s1:s2:s))) =
 (s0+(s1-s0)*(s1-s0)/(2*s1-s0-s2)):(euler_accel rest)

boost transform =
 \series -> map head (iter transform series)
  where iter f x = x:(map f (iter f x))
```

Das sehen wir uns jetzt mal im Detail an:

```
tf@ouija: ~/cde/lambda$ hugs euler.hs

[Hugs Banner]

Main>  take 5 (euler (partialsums pi_summands))
[3.16667,3.13333,3.14524,3.13968,3.14271]
Main>  take 5 (boost euler (partialsums pi_summands))
[4.0,3.16667,3.14211,3.1416,3.14159]
```

Vermessenheit wird allerdings bestraft. Man kann sich hier auch ‘verboosten’:

```
Main> take 5 (boost (boost euler) (partialsums pi_summands))
[4.0,4.0,4.0,4.0,4.0]
```

Was hier passiert, ist intuitiv relativ klar (wenn auch nicht ganz einfach in Worte zu fassen) – wir können es jedenfalls dadurch reparieren, daß wir aus den für das Boosten zu bildenden Tableaus nicht die erste Zeile extrahieren, sondern die zweite, sprich:

```
boost transform =
 \series -> map (head . tail) (iter transform series)
  where iter f x = x:(map f (iter f x))
```

(Wir könnten vielleicht sogar an ein echtes Diagonalverfahren denken...) Damit haben wir dann:

```
Main> take 5 (euler (partialsums pi_summands))
[3.16667,3.13333,3.14524,3.13968,3.14271]
Main> take 5 (boost euler (partialsums pi_summands))
[2.66667,3.13333,3.14145,3.14159,3.14159]
Main> take 5 (boost (boost euler) (partialsums pi_summands))
[2.66667,3.13333,3.14159,
Program error: {primDivDouble 0.0 0.0}
```

Im dritten Schritt wird hier die Gleitkommagenauigkeit der Maschine erreicht, was zu einer Division durch Null führt bei der Bildung des Quotienten der Länge aufeinanderfolgender Schritte.

Wir könnten noch eine Stufe höher steigen und dieses iterative Boosten selbst wieder als Beschleuniger für einen Konvergenzbeschleuniger sehen. (Wir verkneifen uns ein "und-so-weiter", weil... aeh... hm... ach, vergesst es.) Interessant sind hier nur zwei Fragen: (1) wie weit können wir's treiben, bis dieses Verfahren, das sehr wenige Folgenwerte immer komplizierter mixt und einen vermeintlichen Grenzwert berechnet, anfängt, Quatsch zu liefern, und (2) wie viele Elementar-Rechenschritte sind denn nun wirklich nötig? Unsere Verfahren werden immer schneller, aber jeder einzelne Schritt wird immer komplizierter. Natürlich hängt die Antwort von der gewünschten Genauigkeit ab.

Probieren wir nun noch was anderes: $\ln 2$ aus der Taylorentwicklung von $x \mapsto ln(1+x)$:

```
Main> take 5 z
 where z=partialsums (map (\x->(if odd x then 1.0 else -1.0)/(toEnum x))
                          [1,2..])
[1.0,0.5,0.833333,0.583333,0.783333]
Main> take 5 (euler z)
 where z=partialsums (map (\x->(if odd x then 1.0 else -1.0)/(toEnum x))
                          [1,2..])
[0.7,0.690476,0.694444,0.692424,0.69359]
Main> take 5 (boost euler z)
 where z=partialsums (map (\x->(if odd x then 1.0 else -1.0)/(toEnum x))
                          [1,2..])
[0.5,0.690476,0.693106,0.693147,0.693147]
```

So toll dieses Verfahren ist: es stellt sich hier die Frage, ob es mehr als rein heuristische Schlüsse zuläßt. Eine konvergierende Folge zu haben ist eine Sache. (Es läßt sich ohne allzu großen Aufwand zeigen, daß die beschleunigte und ursprüngliche Folge denselben Grenzwert haben müssen.) Allerdings wäre es wünschenswert, auch eine Garantie für die Konvergenzgeschwindigkeit zu haben, konkret eine Formel, die uns sagt, wie weit wir in der beschleunigten Folge gehen müssen um ein vorgegebenes Maß an Genauigkeit zu erfüllen. Rein gefühlsmäßig drauflosgesponnen möchte ich meinen, daß dieser Trick in sehr vielen Fällen, in denen die Koeffizienten der ursprünglichen Folge durch eine analytische Funktion der Koeffizientenposition gegeben sind, sehr nützliche Dienste leisten kann, aber wohl auch sehr leicht überlistet werden kann, wenn man's wirklich darauf anlegt. Stellen wir uns etwa vor, wir nehmen die ersten 100 Glieder der obigen gegen π konvergierenden alternierenden Reihe. Alle folgenden Glieder ersetzen wir durch die Glieder der gegen $\ln 2$ konvergierenden Reihe $1-1/2+1/3-1/4\ldots$, wobei wir hier endlich viele Anfangsglieder wegschneiden, so daß auch an der Nahtstelle der Absolutbetrag der Folgenglieder monoton abnimmt. Diese Folge konvergiert gegen $\ln 2+$ rationale Zahl, doch wenn wir unser Verfahren nur den Anfang der Folge erschnüffeln und daraus Schlüsse ziehen lassen, wird sie (so hoffen wir) einen Näherungswert für π liefern. Die rationale Zahl oben wird natürlich gerade so groß sein, um $\ln 2$ in die Nähe von π zu verschieben, aber weil beim 100. Glied Böses passiert, sollten wir Abweichungen im Prozent-Bereich erwarten. Unsere beschleunigte Folge sollte aber ganz naiv betrachtet den Eindruck erwecken, schon nach wenigen Gliedern den falschen vermeintlichen Grenzwert π auf wesentlich mehr als zwei Nachkommastellen zu liefern. Jedenfalls scheint es allein schon deswegen wenig sinnvoll, zu versuchen, hier beim 'Kaffeesatzlesen' eine nicht gegebene Exaktheit durch Verwendung von exakten Brüchen anstelle von Gleitkommazahlen betonen zu wollen (es sei denn vielleicht, um die Grenzen der Hardwaregenauigkeit ohne großen Aufwand ein wenig rauszuschieben.)

Man kann sich jetzt fragen: wo genau steckt denn eigentlich hier der Trick, der all dies möglich macht? Funktionen höherer Ordnung erlauben uns, treue Darstellungen abstrakter Ideen in Form von Code zu finden, faule Auswertung erlaubt uns dasselbe für die 'Objekte', auf die sich unsere Ideen beziehen. Je mehr man darüber nachdenkt, umso verwirrender scheint es zu werden, eine intuitiv richtige Erklärung zu finden. Vielleicht ist die beste Art, darüber nachzudenken, folgende: *Am Anfang war der Kalkül.*

Es scheint beängstigend, daß Anhänger einer Programmiersprache wie C zum einen überhaupt nicht über solche Ansätze nachdenken können, zum anderen das nicht mal als Einschränkung empfinden[51].

Zu Anfang wurde versprochen, wir würden direkt π berechnen und nicht nur Näherungswerte. Nun, das ist einfach: eine Möglichkeit, reelle Zahlen zu definieren ist über Äquivalenzklassen von Cauchy-Folgen. `partialsums pi-summands` ist sicher eine treue Darstellung einer Cauchy-Folge, wenn wir von Dezimalbrüchen übergehen zu rationalen Zahlen[52]. Und das wird von Haskell unterstützt:

```
pi_summands=map (\x->(if odd x then -4 else 4)%(2*x+1)) [0,1..]
```

Zum Schluß möchten wir noch ein ganz besonderes "Schmankerl" vorführen: *die Weltformel.* Nachdem wir uns in der letzten Lektion schon mit der Apokalypse beschäftigt haben, dürfte das vielleicht ein angemessener Abschluß sein.

Man kann auf die Idee kommen, daß es mit faulen Tricks möglich sein könnte, einen Kombinator zu basteln, in dem jeder andere nur irgendwie erdenkliche Kombinator als Unter-Ausdruck enthalten ist. (Für Informatiker: die Menge al-

[51] "Imperprog plusgut denk".

[52] Allerdings sollte angemerkt werden, daß die Eigenschaft Cauchy-Folge zu sein "rechnerischen Gehalt" hat. Erinnern wir uns an die Definition: "Für alle $\varepsilon > 0$ *gibt es* einen Index N, so daß...", wobei der Rest eine reine $\forall$-Formel ist. Für diese Eigenschaft gibt es einen "Zeugen", auch "Cauchy-Modul" genannt, der nicht weggeworfen sollte. Zur vollständigen Darstellung einer reellen Zahl gehört also nicht nur eine Cauchy-Folge, sondern auch eine Funktion die zu gegebener Genauigkeit $\varepsilon > 0$ einen möglichen Index N angibt, ab dem die Folgenglieder entsprechend dicht zusammenrücken. Der mathematisch interessierte Leser mag sich einen möglichen Zeugen der Cauchy-Eigenschaft überlegen.

ler Kombinatoren ist rekursiv aufzählbar[53].) Dieser Kombinator würde dann beweisbar alle Programme, alle Texte, alle Gedichte, alle Formeln, einfach die ganze Welt enthalten. Natürlich auch sich selbst. Es gibt viele Wege, so was explizit zu konstruieren. Wir verwenden hier folgende Methode, die auf der Idee basiert, daß jeder Term rekursiv konstruiert werden kann aus dem Genesis-Kombinator `X` und Termen, die durch Anwendung eines Terms auf einen anderen Term entstehen. Leider funkt uns hier das Typsystem ein wenig dazwischen (wäre auch ein Wunder, wenn nicht. Wenn es irgendwas gibt, was nicht typisierbar ist, dann sicher so ein Kombinator, der alle denkbaren Kombinatoren aufzählt. . .), so daß wir dahingehend nur eine Approximation erhalten können, daß wir letztendlich nur die Struktur dieser Terme berechnen, aber sie nicht als echte Kombinatoren in Erscheinung treten lassen[54]. Das stört aber nicht weiter, weil wir unser Haskell-Programm eh als Kurznotation eines entsprechenden Lambda-Kalkül-Ausdrucks ansehen wollen (wer will kann das als Übung gerne umformen – oder sogar in Form eines Ausdrucks bringen, der nur X enthält. Falls es jemand gibt, der wirklich so verrückt ist, das auch zu machen, hätten wir gern eine Kopie.)

Letztendlich sieht das dann beispielsweise so aus:

```
data LAM = X | App LAM LAM deriving (Eq, Show)

-- Take two streams and merge then into a single one.

merge (r:rs) s = r : merge s rs

-- take two streams and create the list of all combinations. This in a
-- sense is the computational content of the proof that N x N and N have
-- the same cardinality:

square (a:as) l = merge (map (\ b-> (a,b)) l) (square as l)

-- and now define our world formula

combs = X: (map (\ (a,b)->App a b) (square combs combs))
```

Und das kommt raus, wenn man's auswertet:

```
[X,App X X,App (App X X) X,App X (App X X),App (App (App X X) X) X,App
X (App (App X X) X),App (App X X) (App X X),App X (App X (App X
X)),App (App X (App X X)) X, {Interrupted!}
```

[53]Da ein Informatiker auch mit Rekursionstheorie vertraut sein sollte und daher mit dem Begriff “Kleene-Index” etwas anfangen können sollte dürfte ihm das nur ein müdes Lächeln wert sein. Natürlich können wir die natürlichen Zahlen aufzählen. . .

[54]Es sei angemerkt, daß wir diese “toten” Terme auch recht einfach “animieren” können; aber “Normalisierung durch Auswertung” ist ein anderes Thema und soll hier nicht behandelt werden.

In unserer herkömmlichen Notation ist das gerade die Liste der Ausdrücke

$$X, XX, XXX, X(XX), XXXX, X(XXX), XX(XX), \ldots$$

Das ist zwar schön und irgendwie vielleicht auch ein wenig verblüffend, aber natürlich nicht so recht nützlich, denn eine Bibliothek, in der jeder nur denkbare Buch steht (also auch alle Bücher mit allen erdenklichen Fehlern), ist genauso nutzlos wie eine Bibliotkek, in der gar kein Buch steht. Dennoch ist es irgendwie vielleicht doch gerechtfertigt, einen solchen Ausdruck eine "Weltformel" zu nennen, auch wenn das ganz sicher die Leute, die wirklich im Weltformel-Busineß arbeiten[55], nicht arbeitslos machen wird. Dennoch, eine nette und recht plakative Spielerei mit dem Lambda-Kalkül.

[55] Wie beispielsweise der Chef eines der beiden Autoren.

9 Standardisierung: Der Beweis, daß Faulheit siegt

In diesem Kapitel soll ein theoretischer Hintergrund zur faulen Auswertung geliefert werden. Auch dies wurde im Kurs nicht besprochen, sollte jedoch jedem bekannt sein — allein um sehen, wie einfach man verstehen kann, daß Faulheit (sogar die dumme in Form von links-außen) immer zum Ziel führt, falls es überhaupt eines gibt. (Einem der Kursleiter erschien dieses Resultat immer als wahnsinning schwer und tiefliegend Resultat (zumal es in Einführungsveranstaltungen in Informatik immer nur zitiert aber nie bewiesen wird), solange bis ihm jemand diesen Beweis zeigte.)

Die Darstellung dieses Kapitels orientiert sich im wesentlichen an derjenigen von Joachimski und Matthes [3]. Und ist bewußt etwas knapp gehalten, nicht nur weil nicht behandelte Dinge nicht die Kursunterlagen überschwemmen sollen, sondern auch, weil es sehr erhellend für den Leser ist, sich die Detail selber zu überlegen; außerdem ist man oft eher bereit ein kurzes und kompaktes Kapitel zu lesen, statt einem leicht verdaulichen, aber langatmigen. (Man beachte, wie auch hier Faulheit (in dem Fall die der Kursleiter) vom `bug` zum `feature` wird.) Selbstverständlich gilt auch hier das Angebot an unsre Kursteilnehmer*, daß die Kursleiter bei Fragen, Problemen und Lösungsvorschlägen für Übungsaufgaben stets zur Verfügung stehen (email-Adressen sind ja bekannt).*

Die intuitive Vorstellung von (dummer) Faulheit war, daß man im Falle einer Funktionsanwendung $(\lambda x.r)s$ das Argument s lieber gar nicht ausrechnet, wer weiß, vielleicht braucht man es ja gar nicht; also vorsichtshalber erst mal $r[x := s]$ bilden und weitersehen. Wenn sich dann nacher herausstellt, daß man das s doch braucht und man es hätte gefahrlos ausrechnen können, kann man das ja immer noch tun (wenn man allerdings im Gegensatz zu `Haskell` die Substitution symbolisch ausführt und nicht Zeiger teiltx, muß man zwar vielfache Arbeit machen, aber endlich oft etwas endliches bleibt endlich; wichtiger war es erst einmal zu "überleben", sprich das Risiko zu vermeiden, einen nicht-terminierenden Term ausrechnen zu wollen, den man am Ende gar nicht braucht, denn das hätte ja bedeutet vollkommen grundlos nicht zu terminieren).

Wie machen wir diese Überlegungen nun formal? Nun, wir wollen zeigen, daß unsere Reduktionsstrategie immer eine Normalform findet, wenn es eine gibt. Dazu müßten wir sie mit allen auf Normalformen führenden Reduktionsfolgen vergleichen. Da letztere etwas schwer in den Griff zu bekommen sind, vergleichen wir unsere Reduktionsstrategie eben mit allen anderen Reduktionsfolgen. Dann können wir aber nicht mehr erwarten, daß unsere Reduktionsstrategie das alles nachfahren kann: sie ist darauf angelegt, alle Redexe zu beseitigen; beliebige Reduktionsfolgen dürfen aber im allgemeinen Redexe stehen lassen!

Aus diesem Grund weichen wir den Begriff links-außen etwas auf (und nennen ihn dann Standardreduktion): statt den links-äußersten Redex zu reduzieren dürfen wir ihn auch stehen lassen — dann aber für immer. Anders ausgedrückt:

wir gehen die Redexe in der "kanonischen" Reihenfolge durch und entscheiden uns, ob wir sie umdrehen wollen oder nicht.

Bevor wir zur eigentlichen Definition kommen benötigen wir jedoch noch etwas Notation: Mit $\vec{t}$ ("t Vektor") bezeichnen wir endliche Listen $t_1, \ldots, t_n$ von Termen. Diese können auch leer sein. t und $\vec{t}$ stehen in nicht notwendig in Beziehung zueinander (den letzters sind ja alles Terme mit einem Index) und weiterhin sei Links-Klammerung vereinbart; es ist also insbesondere $t\vec{t}$ kurz für $(((tt_1)t_2)\ldots)t_n$. Offenbar sind alle λ-Terme von einer der Formen $x\vec{t}$ oder $(\lambda x.r)\vec{t}$ (denn Variablen und λ-Abstraktionen sind von der Form und im Falle einer Applikation rs ist r schon von einer dieser Formen und dann wird der "Vektor" halt eins Länger).

Definition 9.1 (Standardreduktion $\rightsquigarrow$). *Wir definieren induktiv den Begriff $\rightsquigarrow$ einer Standardreduktionsfolge. Dabei sei $\vec{t} \rightsquigarrow \vec{t'}$ eine Abkürzung für $t_1 \rightsquigarrow t'_1, \ldots, t_n \rightsquigarrow t'_n$. Man beachte, daß diese Forderung trivial ist, falls $\vec{t}$ die leere Folge von Termen ist.*

- *Falls $\vec{t} \rightsquigarrow \vec{t'}$, so $x\vec{t} \rightsquigarrow x\vec{t'}$.*
- *Falls $\vec{t} \rightsquigarrow \vec{t'}$ und $r \rightsquigarrow r'$ so $(\lambda x.r)\vec{t} \rightsquigarrow (\lambda x.r')\vec{t'}$.*
- *Falls $r[x := s]\vec{t} \rightsquigarrow t'$ so $(\lambda x.r)s\vec{t} \rightsquigarrow t'$.*

Zunächst mal sieht man, daß Standardisieren nichts anderes ist als Reduktionen in einer bestimmten Reihenfolge, auszuführen (genau diesen Begriff wollten wir ja auch definieren!); genauer zeigt man leicht durch Induktion über die Definition von $\rightsquigarrow$, daß

Lemma 9.2 ($\rightsquigarrow \subset \rightarrow_\beta$). *Falls $t \rightsquigarrow t'$, so auch $t \rightarrow^*_\beta t'$.*

Unser Ziel wird sein, die "Umkehrung" zu zeigen, das heißt, daß jede Folge von Reduktionen auch durch eine Standardreduktionsfolge erreicht werden kann. Dazu brauchen wir zunächst einige Lemmata.

Lemma 9.3. *Für alle Terme t gilt $t \rightsquigarrow t$.*

Beweis. Einfache Induktion über den Aufbau von t □

Lemma 9.4. *Gilt $r \rightsquigarrow r'$ und $t \rightsquigarrow t'$, so auch $rt \rightsquigarrow r't'$.*

Beweis. Induktion darüber, daß $r \rightsquigarrow r'$ (meint: wenn $r \rightsquigarrow r'$ gilt, so muß es hierfür eine Herleitung aus obigen Regeln geben; wir argumentieren durch Induktion über diese Beweisfigur). Man beachte, daß in den "Anfangsfällen" (also den ersten beiden Fällen) die $\vec{t}$ stets verlängert werden können. □

Übung 9.5. *Führe die Beweise der beiden Lemmata im Detail aus!*

Lemma 9.6. *Falls $r \rightsquigarrow r'$ und $s \rightsquigarrow s'$ so auch $r[x := s] \rightsquigarrow r'[x := s']$.*

Beweis. Induktion über $r \rightsquigarrow r'$. Falls $y\vec{t} \rightsquigarrow y\vec{t'}$ dank $\vec{t} \rightsquigarrow \vec{t'}$, so wissen wir $t[s \vec{:=} x] \rightsquigarrow t'[x \vec{:=} s']$ nach Induktionsvorraussetzung. Falls x und y verschiedene Variablen sind, so sind wir fertig; andernfalls wenden wir wiederholt Lemma 9.4 an (für jedes Element der Liste $\vec{t}$ einmal).

Die andern beiden Fälle folgen unmittelbar aus der Induktionsvoraussetzung. □

Theorem 9.7. *Falls $r \rightsquigarrow r' \to_\beta r''$, so auch $r \rightsquigarrow r''$*

Beweis. Induktion über $r \rightsquigarrow r'$.

Der einzig nicht-triviale Fall (meint: der einzige Fall in die Induktionsvoraussetzung nicht sofort die Behauptung liefert) ist der, daß ein Kopfredex zunächst stehen gelassen wude und nun reduziert wird, also $(\lambda x.r)s\vec{t} \rightsquigarrow (\lambda x.r')s'\vec{t'} \to_\beta r'[x := s']\vec{t'}$ dank $r \rightsquigarrow r'$, $s \rightsquigarrow s'$ und $\vec{t} \rightsquigarrow \vec{t'}$.

Die Idee der Standardisierung war gerade, daß wir diesen Redex als erstes ausführen (nach man sich jetzt offenbar doch entschieden hat, den Redex nicht stehen lassen zu wollen) und dann mit Standardfolgen weiter machen können. Formal argumentieren wir so: Wir schließen die Behauptung aus der dritten Regel. Dazu müssen wir zeigen $r[x := s]\vec{t} \rightsquigarrow r'[x := s']\vec{t'}$. Aus Lemma 9.6 wissen wir, daß $r[x := s] \rightsquigarrow r'[x := s']$. Das Anhängen der $\vec{t}$ geschieht durch wiederholtes Anwenden von Lemma 9.4. □

Damit haben wir es gschafft: jede Reduktionsfolge kann durch eine Standardreduktionsfolge ersetzt werden. Durch Induktion über die Länge der Reduktionsfolge erhalten wir nämlich aus Lemma 9.3 und Theorem 9.7

Korollar 9.8. *Falls $r \to_\beta^* r'$, so auch $r \rightsquigarrow r'$.*

Insbesondere wissen wir nun, daß links-außen Reduktion stets die (wegen Konfluenz eindeutige) Normalform findet, falls sie existiert: sei r' die Normalform von r. Dann $r \to_\beta^* r'$, also $r \rightsquigarrow r'$. Da r' normal ist, kann die zweite Regel (in der Herleitung von $r \rightsquigarrow r'$) nur mit leerem $\vec{t}$ verwendet worden sein. Dies liefert dann aber eine deterministische Reduktionsstrategie, nämlich gerade links-außen-Reduktion(!).

10 Monadischer IO: Ein-/Ausgabe für solche die nicht glauben, daß sich die Welt verändern kann

Dieses Kapitel wurde nicht im Kurs behandelt. Es beschreibt wie Ein-/Ausgabe in voll funktionalen Programmiersprachen (wie etwa `Haskell`*) realisiert ist. Auf die kategorischen Hintergründe (Stichwort "monadische Funktoren") wird* nicht *eingegangen.*

In reinen funktionalen Sprachen wie *Haskell* ist jede Funktion eine Funktion im streng mathematischen Sinne, d.h. eine wohldefinierte eindeutige Abbildung. Dies ist ein wesentlicher Unterschied zu anderen Programmiersprachen, in denen "Funktionen" mehr oder weniger ad hoc Approximationen an dieses mathematische Konzept darstellen und in erster Linie eher von den Gegebenheiten einer Rechenmaschine auf niedriger Ebene geprägt sind.

Dieses "Funktionskonzept *imperativer* Sprachen", an das wir uns beim Programmieren über Jahre hinweg schon so sehr gewöhnt haben, daß wir es als Selbstverständlichkeit hinnehmen (und bisweilen Schwierigkeiten haben, uns überhaupt vorstellen zu können, wie es anders gehen könnte), hat einige fundamentale Schwächen. Auch an diese haben wir uns so sehr gewöhnt, daß wir eigentlich nicht groß darüber nachdenken, aber insbesondere unbedarfte Anfänger haben erfahrungsgemäß hier oft Probleme. Der "Zuweisungsbegriff" nimmt hier eine zentrale Rolle ein. In einer imperativen Programmiersprache (für alle folgenden Beispiele sei die Programmiersrpache `C` verwendet) bedeutet ein Konstrukt wie

```
k=2;
```

daß der Variable `k` den Wert 2 zugewiesen wird. Die Variable `k` ist hierbei nicht etwa eine mathematische Variable, sondern ein Behälter, der einen *Zustand*[56] enthalten kann.

Was nun Anfängern in aller Regel große Probleme bereitet, ist, daß hier mathematische Notation und Terminologie verwendet wird, sich diese Objekte aber ganz und gar nicht wie bekannte mathematische Objekte verhalten. Das zeigt sich beispielsweise hier:

```
int c,d;

c=fgetc(stdin);
d=c;
```

versus

[56] Als "Zustand" bezeichnet man allgemein die Teile eines Programms, die sich ändern können. Das ist zwar etwas ungenau formuliert, trifft aber ganz gut den Kern der Sache.

```
int c,d;

c=fgetc(stdin);
d=fgetc(stdin);
```

Im ersten Fall wird ein Zeichen gelesen und sein Wert den Variablen `c` und `d` zugewiesen. Im weiten Fall werden zwei Zeichen gelesen, das erste der Variable `c` zugewiesen, das zweite der Variable `d`. Das ist etwas wesentlich anderes, obwohl es aus rein mathematischer Sicht sehr wünschenswert wäre, wenn diese Transformation erlaubt wäre. (Gerade daß sie es nicht ist, ist für viele Einsteiger ein Grund für Verständnisprobleme.)

Vorneweg: in funktionalen Sprachen, wie `Haskell` sind solche rein mathematischen Umformungen an jeder Stelle zulässig!

Probleme sind oft besser in den Griff zu kriegen, indem man ihnen einen Namen gibt. Bei dem zuvor Untersuchten spricht man von *fehlender referentieller Transparenz*. Eine Sprache wie *Haskell* ist im Gegensatz hierzu *referenztransparent*.

Die Wurzel des Übels ist, daß wir hier mit Zuweisungen und Zuständen arbeiten, und I/O über "Pseudofunktionen" implementieren. Diese sehen syntaktisch aus wie Funktionen, aber sie vermitteln keine wohldefinierten eindeutigen Abbildungen[57]. Mit dem "normalen" Ansatz kann das auch gar nicht gehen; am deutlichsten wird das bei einer Funktion wie `rand()`. Eine Zufallsfunktion, die (in imperativer Sprechweise) "bei Aufruf mit denselben Parametern immer denselben Wert liefern muß, und null Parameter hat, also gar nicht mit verschiedenen Parametern aufgerufen werden kann", kann nicht sonderlich zufällig sein. Für Input gilt praktisch entsprechendes. Für Output darf sich der Leser überlegen, daß das Problem der fehlenden Referenztransparenz auch hier besteht, und sich das Beispiel von oben sinngemäß übertragen läßt.

Funktionale Sprachen sind prima geeignet, um irgendwelche Berechnungen durchzuführen. Sobald wir aber versuchen, "die Welt um uns herum zu verändern", d.h. I/O zu tätigen, laufen wir in ein *Körper-Geist-Problem.* "Wie kann Rechnen die Welt verändern?" Die Antwort ist: gar nicht. Damit haben wir — was I/O in

[57] An dieser Stelle sei angemerkt, daß solche, als Funktionen getarnte Zustände genau den Kern objekt-orientierter Programmierung in solchen Sprachen ausmachen, in denen so etwas erlaubt ist, wie etwa `Scheme`. Auch wenn wir wissen, daß `let` nur syntaktischer Zucker für einen zugrundeliegenden β-Redex ist (Übungsaufgabe: für welchen?), so eignet es sich doch prima, um (selbstverständlich anonyme!) Objekte zu definieren:

```
(let ((⟨Objektvariable1⟩ ⟨Anfangswert1⟩)
      (⟨Objektvariable2⟩ ⟨Anfangswert2⟩)
      ...)
  (lambda (methodenname)
    (case methodenname
        ((⟨Methode1⟩) (lambda ⟨Arglist1⟩ ⟨Body1⟩))
        ((⟨Methode2⟩) (lambda ⟨Arglist2⟩ ⟨Body2⟩))
        ...)))
```

funktionalen Sprachen angeht — ein fundamentales schwerwiegendes Problem. In der Vergangenheit gab es verschiedene mehr oder weniger pfiffige Ansätze, mit diesem Problem umzugehen, sei es nun die Verwendung von sehr trickreichen Streams, Continuation Coding, Tricks mit "linearer Logik", oder noch ausgefallenere Ideen. Alles davon "funktioniert zwar irgendwie", ist aber sowohl ästhetisch als auch rein pragmatisch in mehrerlei Hinsicht nicht vollkommen befriedigend.

Erst in den späten Neunzigern hat sich ein neuer Ansatz herauskristallisiert, von dem man, wenn man ihn mal verstanden hat, wohl am ehesten behaupten kann, daß er die richtige Weise darstellt, über derartige Dinge nachzudenken und mit ihnen umzugehen. Es ist der *monadische I/O*. Hinter dem Begriff steckt der Kategorien-theoretische Begriff einer Monade[58], auf den hier aber nicht eingegangen werden soll. Statt dessen suchen wir einen mehr oder minder anschaulichen Zugang zur I/O-Monade.

Wenn man sich das erste Mal mit Haskell beschäftigt, mag es sehr erstaunlich klingen, daß es Ansätze gibt, diese Sprache für Chipdesign einzusetzen.[59] Oberflächlich betrachtet liegen zwischen den formalen rein mathematischen Ansätzen einer funktionalen Sprache, die sich in Sphären fernab aller profanen Hardware-Beschränkungen bewegt und der nackten Welt der Gatter und Flip-Flops Welten. Genauer betrachtet macht das allerdings durchaus einigen Sinn; man darf nur nicht den Fehler begehen, Objektsprache (Gatter-Logik) und Meta-Sprache (formale Beschreibungssprache) durcheinanderzuwerfen. Haskell kann seine Stärken überall dort ausspielen, wo korrektes Arbeiten und die Vermeidung von Spezifikationsfehlern sehr hohe Priorität hat, und bringt einige sehr abstrakte Werkzeuge mit. Wieso sollen wie "Werte", mit denen man arbeitet, nicht Hardwarespezifikationen sein?

Dieser Objektsprache-Metasprache Ansatz findet sich auch an anderen Stellen. Die Sprache SML beispielsweise ist mehr oder weniger als Metasprache eines formalen Beweisverifikationssystems entstanden und hat sich zu einer eigenständigen Programmiersprache gemausert. Die Objektsprache ist hier die

---

[58] Der Begriff Monade ist in gewisser Weise die kategorische Fassung der Monoid-Begriffs. Es handelt sich um einen Funktor T mit "Multiplikation" (natürlicher Transformation $T^2 \to T$) und "Einselement" (natürlicher Transformation $\mathrm{Id} \to T$) und "naheliegenden Rechenregeln".

Anschaulich bezeugt die Multiplikation, daß es sich um eine Hüllenbildung handelt und die Eins ist die Einbettung in die Hülle. Für die anderen beiden `Haskell` Monaden `[]` und `Maybe` heißt das konkret:

- Listen von Listen von Dingen "sind" wieder Listen von Dingen via `flatten`, also `[[a,b],[],[c,d,e],[f]]` $\rightsquigarrow$ `[a,b,c,d,e,f]`.
- Sich zweimal fragen, ob man etwas hat, "ist" das gleiche wie sich nur einmal fragen, via `Just Nothing` $\rightsquigarrow$ `Nothing`.
- Ein Ding kann als Einerliste von Dingen aufgefaßt werden: `a` $\rightsquigarrow$ `[a]`
- Ein konkret gegebenes Ding "ist" auch ein nur möglicherweise vorhandenes vermöge des Wissens, daß es doch vorhanden ist, also via `Just`.

[59] Die `haskell.org` Site verweist hier auf Hawk (`http://www.cse.ogi.edu/PacSoft/projects/Hawk/`) und Lava (`http://www.cs.chalmers.se/~koen/Lava/`).

mathematische Sprache, in der Terme und Theoreme formuliert sind, auf denen mit Werkzeugen der Metasprache gearbeitet wird.

Gewissermaßen stellen I/O-Monaden auch nichts anderes als eine analoge Trennung zwischen Objektsprache und Metasprache dar. Zwar können wir in voll funktionalen Sprachen nicht auf spezielle I/O-Funktionen zurückgreifen, aber bei genauer Betrachtung wollen wir so was eigentlich auch gar *nicht*(!). Ein wenig Haarspalterei bringt uns hier weiter: funktionale Programmierung ist das Reich des Geistes. Wenn wir auf irgend eine Weise die Welt aktiv gestalten wollen, machen wir uns zuerst im Kopf einen *Plan* dieser Aktion, und dieser wird dann weitergereicht an die bewußt kontrollierbare Muskulatur. Eine zentrale Rolle spielt dabei das motorische Zentrum im Gehirn, das dafür zuständig ist, die Muskelbewegungen zu koordinieren.

Die Welt beeinflussen können wir in der funktionalen Welt des Geistes nicht. *Aber* wir können allein mit den Werkzeugen des Geistes Pläne schmieden und insbesondere große Pläne aus kleineren zusammensetzen. Metasprache ist die funktionale Sprache, hier Haskell, Objektsprache ist die Sprache, in der I/O-Pläne formuliert werden. Wenn wir dann noch einen magischen Platz haben, dem wir diesen Plan übergeben können, so daß er auch wirklich zur Ausführung gelangt, haben wir bereits alles, was wir brauchen. Fragen nach der Effizienz wollen wir zunächst mal vergessen; wer dennoch darüber nachdenken will, sollte das Zauberwort "faule Auswertung" im Hinterkopf behalten.

Recht viel mehr verbirgt sich hinter monadischem I/O in Haskell auch schon gar nicht. Die "magische Stelle" heißt konventionsgemäß in Haskell `main` (und wenn man's genau nimmt, kann man die `main()`-Funktion in `C` ganz analog in dem hier vorgestellten Bild verstehen.) Wenn in einem Haskell-Programm `main` definiert ist als ein Plan für einen I/O-Ablauf, so wird dieser bei Start des Programms abgearbeitet.

Ein vollständiger I/O-Plan hat Typ[60] `IO()`. Wir wollen uns von einfachen zu komplizierteren Beispielen vorwärts tasten. Deswegen beschäftigen wir uns als allererstes nur mit *Output*.

Um einen String auszugeben, müssen wir `main` definieren als einen Plan, nach dem ein String ausgegeben werden soll. Nun gibt es in Haskell eine Funktion `putStr :: String -> IO ()`, die gerade einen String auf einen Plan abbildet, diesen String auszugeben. (Man kann `putStr` auch aus `putChar` aufbauen, aber damit wollen wir uns jetzt nicht beschäftigen.) Wir wollen als erstes nur ganz einfach "`Hello world!`" ausgeben:

```
main = putStr "Hello world!\n"
```

Das schreiben wir in ein File "`hello.hs`". Wenn wir den Glasgow Haskell Compiler installiert haben, ist der schnellste Weg, um "quick&dirty" zu einem Resultat

[60] Mit jedem Typ `a` ist auch `IO a` ein Typ. Außerdem gibt es die Abbildung `fmap :: (a -> b) -> IO a -> IO b`. In diesem Sinne ist `IO` ein Funktor.

`()` ist der "Unittyp", also derjenige Typ, der nur vom leeren Tupel (ebenfalls mit `()` bezeichnet) bewohnt wird. Der entsprechende Typ in `C` ist `void`.

zu kommen, hieraus ein Binary zu compilieren: `ghc -o hello hello.hs`. (Für Debian gibt es die Pakete `ghc4`, `ghc4-doc`, `ghc4-libsrc`, `ghc4-prof`.) `hugs` verhält sich hier anscheinend etwas speziell: alles, was interaktiv ausgewertet wird und ein Ergebnis vom Typ I/O-Plan (also `IO ()`) liefert, wird auch gleich als Plan ausgeführt. Demzufolge rufen wir hier "`hugs hello.hs`" auf und geben dann "`main`" ein.

So weit, so gut. Doch wie kombinieren wir nun Pläne für I/O? Solange wir nur Output machen wollen, reduziert sich das auf die Frage, wie wir Output sequenzieren. Wir brauchen ein Werkzeug der Metasprache, um aus zwei Plänen einen zu machen, der die Hintereinanderausführung der beiden Pläne beinhaltet. In Haskell gibt es hierfür den (für verschiedene Monaden-Realisierungen "überladenen") Infix-Operator "`>>`". Wir wollen hier die sektionierte Schreibweise verwenden, um einheitlich nur mit der üblichen Präfix-Notation arbeiten zu können. "`(>>)`" ist dann ein Kombinator, der einen I/O-Plan `a` abbildet auf eine Funktion, die einen I/O-Plan `b` abbildet auf einen I/O-Plan, der daraus besteht, erst den Plan `a` und gleich danach den Plan

Konkrete Anwendung:

```
main = (>>) (putStr "Hello world!\n")(putStr "again!")
```

Damit können wir jetzt Output schön sauber sequenzieren, aber ohne eine Möglichkeit, Input zu lesen und zu verarbeiten, sind unsere Möglichkeiten damit noch ein wenig arg beschränkt. Bevor wir dazu kommen ist es aber vielleicht an der Zeit für eine interessante Bemerkung, die wir jetzt schon verstehen können, und an der das zusätliche Feature "Input" auch nichts ändert. Wir können diesen Ansatz nämlich auch von einer rein technischen Seite betrachten. Faule Auswertung in einer reinen funktionalen Sprache bedeutet, daß ein Ausdruck nicht ausgewertet wird, wenn es nicht unbedingt sein muß. Vom mathematischen Standpunkt aus betrachtet ist es allerdings sinnvoller, dies einwenig anders zu sehen: der unumstößliche Grundsatz ist, daß Rechnen die Welt nicht verändern darf. So gesehen könnten wir uns durchaus eine Implementierung einer funktionalen Sprache vorstellen, in der, sagen wir auf einer massiv-parallelen Maschine in manchen Situationen bei einem `if`-Konstrukt sowohl die Bedingung als auch der "erfüllt"-Zweig als auch der "nicht erfüllt"-Zweig gleichzeitig (auf verschiedenen Prozessoren) ausgewertet werden, um effektiv schneller fertig sein zu können. Sobald das Ergebnis der Bedingung vorliegt, kann die Berechnung des uninteressanten Werts abgebrochen werden; dieser Zweig wird verworfen. (Dieser Ansatz nennt sich "spekulative Auswertung"; fast alle modernen Mikroprozessoren beherrschen diesen Trick auf niedrigster Ebene, oft in mehreren Varianten, allerdings ist der Buchhaltungsaufwand in der Hardware, der dafür nötig ist, um die uninteressanten Berechnungen wirder zu verwerfen nicht ganz ohne.) Während es in einer imperativen Sprache fatal wäre, beide Zweige einer Bedingung auszuwerten und einen erst am Ende zu verwerfen (was - in imperativer Sprechweise - einer Implementierung von `if` als Funktion anstelle eines speziellen Sprachkonstrukts gleichkäme), taucht dieses Problem bei einem rein funktionalen Ansatz grundsätzlich nicht auf. Ob die Werte, mit denen wir rechnen, I/O-Pläne sind oder nicht, spielt dabei keinerlei Rolle. Rein technisch betrachtet muß eine rein funktionale Implementierung von I/O zwingende Sequenzierung beinhalten. Es darf nicht möglich sein, sagen wir, erst einen String auszugeben, sich dann den "Wert der den Zustand der Welt nach der Ausgabe des Strings kapselt" zu merken, danach noch einen String auszugeben, und dann zurückzugehen zum gemerkten Zustand vor Ausgabe des zweiten Strings und einen anderen String auszugeben. Ein alternativer Ansatz für I/O, der von der Sprache `Clean` verfolgt wird, basiert darauf, zu verhindern, daß Werte, die den Zustand der Welt darstellen, dupliziert werden. Lineare Logik wird hier eingesetzt, um Sequenzierung zu erzwingen. Verbreiteter war aber wohl als alternative Strategie Continuation Coding. Aber auch das sehen wir uns aber hier nicht an.

Nachdem wir nun Output verstanden haben, müssen wir uns etwas mehr Gedanken über Input machen. Es hilft hier ein wenig, sich vorneweg Gedanken über Typisierung zu machen. Wenn wir ein Zeichen einlesen wollen, brauchen wir eine Funktion, die einen "Baustein" I/O-Plan liefert, der daraus besteht, ein Zeichen zu lesen. In der Tat: `getChar` ist vom Typ `IO Char`. Damit können wir uns auch vorstellen, warum `IO` ein parametrisierter Datentyp ist; der Typ-Parameter gibt Auskunft darüber, was am Ende des Plans gelesen vorliegt. Für einfachen Output war das `IO ()`.

Wir brauchen nun ganz dringend ein Werkzeug, um elementaren Input in größere I/O-Pläne hineinzuverweben. Ein Werkzeug für Sequenzierung. Man könnte sich hier mehrere ad hoc-Ansätze vorstellen, wie etwa folgenden: es könnte nützlich sein, einen Kombinator zu haben, der einen Plan `a`, an dessen Ende `A` gelesen vorliegt, und einen Plan `b`, an dessen Ende `B` gelesen vorliegt, nacheinander ausführt und somit einen Plan liefert, an dessen Ende `(A,B)` vorliegt. Das wäre denkbar, und vielleicht in manchen Fällen durchaus brauchbar, bei genauerer Betrachtung allerdings nicht sehr nützlich, denn im Normalfall wollen wir, daß Information von `a` nach `b` fließen kann. (Nicht, daß wir nicht, wenn wir ein allgemeineres Werkzeug haben, ein Werkzeug, das sich auf die geschilderte Weise verhält, bauen könnten.)

Es mag mehrere Möglichkeiten für einen allgemein brauchbaren Ansatz geben, aber der natürlichste orientiert sich sicher an der intuitiven Denkweise, wie Pläne geschmiedet werden: wenn wir einen Plan haben, sagen wir, um ein Paßwort zu lesen, ist beim Programmieren der nächste Schritt der, einfach anzunehmen, wir hätten das Paßwort gerade gelesen und als Wert vorliegen, und mit einer Funktion weiterzumachen, die dieses Paßwort abbildet auf einen Plan, wie es verarbeitet werden soll. Was wir brauchen, ist ein Werkzeug, das den *Plan, `X` zu bekommen*, und die *Funktion*, die uns zu beliebigem `X` einen Plan liefert, wie wir es verarbeiten wollen, verbindet zu einem größeren Plan, erst `X` zu erhalten und es dann zu verarbeiten[61].

[61] An dieser Stelle sei dann doch ganz kurz (und (zu) grob vereinfacht) die Idee des Continuation Codings erwähnt: Man verabschiedet sich von der Welt mit der endgültigen Entscheidung

Bei Lichte betrachtet ist das eigentlich nichts anderes als ein verallgemeinertes `let`, und so arbeitet man auch damit, aber das werden wir noch genauer sehen. Das entsprechende Konstrukt ist der Infix-Operator “`>>=`”. Wir werden anfangs wieder die sektionierte Schreibweise benutzen.

Ein einfaches Beispiel wäre wohl auch hier angebracht: wir lesen ein Zeichen und geben es 60mal aus, gefolgt von einem “\ n”.

```
print_n_times 1 c = putChar c
print_n_times n c = (>>) (putChar c) (print_n_times (n-1) c)


main = (>>=) getChar (\c -> (>>) (print_n_times 60 c) (putChar '\n'))
```

Hier werden wir auch sofort mit einem Problem konfrontiert: uns fehlt noch ein Begriff eines ”leeren I/O-Plans”. Die Rekursion in der Funktion `print_n_times` sollte eigentlich sinnvollerweise bei 0 beginnen, nicht bei 1. Das geht nur, wenn wir einen I/O-Plan haben, “gar nichts zu tun”. So was können wir (für beliebige Monaden) mit `return` basteln. (Hier liefert `return X` einen I/O-Plan, um genau den Wert `X` zu bekommen, wofür nichts getan werden muß.) Außerdem gehen wir jetzt von der sektionierten Notation weg. Damit sieht unser Programm — wenn wir zudem überflüssige Klammern streichen — so aus:

```
print_n_times 0 c = return ()
print_n_times n c = putChar c >> print_n_times (n-1) c

main = getChar >>= \c -> print_n_times 60 c >> putChar '\n'
```

Spätestens jetzt sollte erwähnt werden, daß `a >> b` dasselbe ist wie `a >>= \ x-> b`, wenn `x` in `b` nicht auftaucht. (Und das wäre in Haskell gerade `a >>= \_->b`.)

Mancher (`LISP`-Programmierer) könnte sich daran stören, daß das `print_n_times` nicht endrekursiv ist, *aber* wir sind ja in einer faulen Umwelt und dort ist das Hauptziel sich möglichst schnell unter einem Konstruktor zu verstecken[62]. In unserem Fall heißt, als erstes mal sagen mit welcher Ausgabe wir anfangen wollen (`putChar c`), dann können wir schon mal loslegen. Wenn wir das getan haben können wir die erste Aktion auch wieder vergessen und uns dann die zweite

was als erstes zu tun ist, also etwa ein Zeichen einzulesen. Da das ein Abschied für immer ist (denn die Welt verändert sich dadurch, aber das Weltbild des funktionalen ist ein statisches in dem immer gleiches durch gleiches ersetzt werden kann), das Problem mit dieser Einzelaktion in der Regel aber noch nicht gelöst ist hinterläßt man (gleichsam als Testament) eine “Fortsetzung”, also eine Funktion, die angibt, was mit dem Zeichen zu tun ist, wenn man es denn mal erhalten hat.

Das Problem von Rechnen und die Welt verändern wird also im wesentlichen dadurch gelöst, daß man sich in dieser Welt ausrechnet, wie man die Welt verändern möchte und was man in der dann veränderten Welt zu tun gedenkt.

[62]Nein, den etwas makaberen Vergleich vom “Verstecken spielen in einem Minenfeld” bringen wir jetzt nicht, auch wenn er sehr passend wäre. . .

überlegen; aber wer weiß, vielleicht hat der Benutzer ja bis dahin genug von dem Programm...

Besonders deutlich wird dieses Prinzip, wenn man mit unendlichen I/O-Plänen, also mit nicht-terminierenden Programmen arbeitet. Beispiele für nicht-terminierende Programme sind (hoffentlich) das Betriebssystem, aber auch so harmlose Dinge wie `yes`. Schreiben wir also ein Programm, das ein Zeichen einliest und solange ausgibt, bis es unterbrochen wird. Es ist klar, das wir sofort loslegen wollen, *ohne* uns vorher genau zu überlegen, was wir eigentlich (alles) tun[63]:

```
yes c = putChar c >> yes c

main = getChar >>= \c -> yes c
```

Bisher war der I/O-Ablauf absolut starr. Fangen wir nun langsam in kleinen Schritten an, dynamischer zu werden. Erstes Beispiel: wir lesen eine Zeile und verwandeln sie in eine Zahl – wobei wir uns so ähnlich wie `atoi` verhalten wollen: (dieses Beispiel ist absichtlich nicht sehr geschickt - siehe das darauf folgende.)

```
is_digit :: Char -> Bool
is_digit x = (cx >= (fromEnum '0')) && (cx <= (fromEnum '9'))
 where cx = fromEnum x

digit_val :: Char -> Integer
digit_val x =  fromInt(fromEnum x - fromEnum '0')


read_num :: IO Integer
read_num = traverse 0 True
  where traverse so_far still_ws =
getChar >>= \c -> if still_ws && (c==' ' || c=='\t')
                              then traverse so_far still_ws
                              else if is_digit c
                                   then traverse (10*so_far+digit_val c)
                                        False
   else return so_far

print_num 0 = putChar '0'
print_num n = if n<10 then putChar (toEnum ((fromEnum '0')+(toInt n)))
                      else (print_num (n `div` 10))
   >> putChar (toEnum ((fromEnum '0')
   +((toInt n) `mod` 10)))


main = read_num
   >>= \n -> putStr "The number is: "
   >> print_num n >> putChar '\n'
```

[63] Auch das ist ein Aspekt von Faulheit! Im "wirklichen Leben" nennt man ihn wohl "Pragmatismus": erst mal die Arbeiten erledigen, die ohnehin erledigt werden müssen und dann weitersehen.

Hier ist `main` ein Plan, zuerst eine Zahl zu lesen (wobei vorangehender horizontaler Whitespace bestehend aus Leerzeichen und Tabulatoren ignoriert wird), und diese Zahl dann (mit etwas Verzierung drumherum) wieder auszugeben. Nicht sehr sinnvoll, und in der Tat ließe sich der Code auch noch verschönern, aber immerhin ein Beispiel.

In diesem Beispiel haben wir den imperativen Teil sehr eng mit dem Code verwoben. In aller Regel ist es sinnvoller, den I/O-Plan so klein und einfach wie möglich zu halten und die eigentliche Intelligenz in möglichst abstrakte, allgemein verwendbare Funktionen zu verpacken. Speziell hier könnte das bedeuten, daß es statt des oben gewählten Ansatzes mehr Sinn macht, eine Funktion zu basteln, die einen Plan liefert, eine Zeile zu lesen (bzw. auf eine derartige Bibliotheksfunktion zurückzugreifen), und eine allgemeine Funktion zu verwenden, die `atoi()` als `String -> Integer`-Abildung implementiert. Das könnte dann etwa so aussehen:

```
is_digit :: Char -> Bool
is_digit x = (cx >= (fromEnum '0')) && (cx <= (fromEnum '9'))
    where cx = fromEnum x

digit_val :: Char -> Integer
digit_val x =  fromInt(fromEnum x - fromEnum '0')

read_num :: String -> Integer
read_num (' ':xs) = read_num xs
read_num xs = traverse xs 0
 where
  traverse "" n = n
  traverse (x:xs) n = if is_digit x then traverse xs (10*n+digit_val x)
                                    else n

print_num 0 = putChar '0'
print_num n = if n<10 then putChar (toEnum ((fromEnum '0')+(toInt n)))
                      else (print_num (n `div` 10))
   >> putChar (toEnum ((fromEnum '0')
   +((toInt n) `mod` 10)))


read_line :: IO String
read_line = getChar
>>= \c -> if c == '\n'
  then return ""
                      else (read_line >>= \r -> return (c:r))

main = read_line
   >>= \s -> putStr "The number is: "
   >> print_num (read_num s)
   >> putChar '\n'
```

Das war schon etwas dynamischer, aber ein weiteres Beispiel kann wohl nicht schaden. Eine der ersten Programmieraufgaben, mit denen man sich in imperativen Sprachen gerne beschäftigt, ist “Zahlen raten”. Es geht darum, eine (vorgegebene) Zahl zwischen 1 und 100 zu erraten, wobei der Rechner nach jedem Rateversuch “zu klein” oder “zu groß” oder “Treffer” meldet.

```
-- guess-a-number; primitive "game" which requires basic understanding of I/O.
-- as Knuth would say, this program is dedicated to Io, the greek goddess of
-- input and output.

print_string :: String -> IO ()
print_string "" = return ()
print_string (h:t) = putChar h >> print_string t

read_string :: IO String
read_string = getChar
              >>= \c -> if c == '\n'
                        then return ""
                        else read_string >>= \rest -> return (c:rest)


-- primitive atoi function. Iterative process.

atoi :: String -> Int
atoi s =
 let atoi_iter so_far "" = so_far
     atoi_iter so_far (h:t) =
      let hcode = ord h - ord '0'
      in if or [(hcode < 0), (hcode > 9)]
         then 0
         else atoi_iter (10*so_far+hcode) t
 in atoi_iter 0 s


game_step :: Int -> Int -> IO ()
game_step nr_step secret_nr =
 print_string ("Your guess (Step "++(show nr_step)++"): ")
 >> read_string
 >>= \s_guess ->
     let guess = atoi s_guess
     in
      if guess == secret_nr
       then print_string
            ("You won (in "
             ++(show nr_step)
             ++" steps).\n")
       else (print_string
                ("Too "
                 ++(if guess < secret_nr then "low" else "high")
                 ++", try again.\n"))
            >> game_step (nr_step+1) secret_nr
```

```
game_step 0 0 = return ()

game :: Int -> IO ()
game nr = print_string
  "I have chosen a number between 1 and 100. Guess which!\n"
  >> game_step 1 nr

main = game 23
```

11 Noch nicht einmal Lisp: Der emacs

In diesem Kapitel führen in die Programmierung mit emacs-Lisp ein. Auch wenn dies mit dem λ-Kalkül außer der Notation nichts mehr zu tun hat, so kommt doch nicht herum, sich damit näher zu beschäftigen. Außerdem wird damit der Wunsch eines Kursteilnehmers erfüllt näheres über jenen angeblichen "Texteditor" zu erfahren (trotzdem gilt natürlich weiterhin das Angebot des betreffenden Kursleiters an jenen Teilnehmer (und der Gerechtigkeit halber auch an die anderen Teilnehmer) es ihm persönlich zu erklären).

Als Beispiel implementieren wir einen "Taschenrechner" für den λ-Kalkül, was sicher für manche Übungsaufgaben hilfreich ist. Der Code ist (unter der GNU-Lizenz) auch im Netz erhältlich unter `http://www.cip.physik.uni-muenchen.de/~tf/lambda/lambda.el`

Wir haben in Kapitel 7 gesehen, daß Lisp eine (grobe) Approximation an den λ-Kalkül ist. Unschön war die Unterscheidung zwischen `symbol-value` und `symbol-function` und außerdem gab es so etwas häßliches wie "Seiteneffekte", so daß man über die Auswertereihenfolge bescheid wissen mußte. Nicht gerade das Paradies also. Und wir lassen uns hieraus noch weiter vertreiben, denn nun beschäftigen wir uns mit `emacs-lisp`. Dazu ist zunächst folgendes klarzustellen:

- Der `emacs` ist *kein* Texteditor.
- `emacs-lisp` ist *keine* korrekte Implementierung von Lisp.

Zwar steht `emacs` eigentlich für "editor macros" und man kann damit auch Texte editieren, aber allein die Größe[64] zeigt schon, daß dies wohl nicht die Hauptanwendung sein kann.[65] So gibt es etwa einen (eigentlich mehrere) Mailreader, einen News-reader, einen Web-browser, einen Kalender, ein Symbolische-Algebra-Paket, einen Oberfäche für die Shell, einen (eigentlich externen) Spellchecker,... Zu großartigen Anbindung auf wenige, einfach zu merkende Tastenkombinationen sei noch angemerkt "Escape-Meta-Alt-Control-Shift", aber dazu später mehr.

Zur zweiten Behauptung versuche man nur mal `(((lambda (x) (lambda (y) x)) 3) 4)` auszuwerten und freue sich über die kryptische Fehlermeldung.

Bleibt nur noch die Frage: warum beschäftigen wir uns überhaupt damit? Nun,

- Der `emacs` ist ein einfaches Werkzeug zur Erstellung von (graphischen) Benutzeroberflächen.

[64] Spötter behaupten, EMACS sei ein Eintrag in `errno.h` für "Editor too big". Gelegentlich hört man auch: "eight megabytes and constantly swapping".

[65] Um es mit Tom Christiansen zu sagen: "Emacs is a nice Operating System, but I still prefer Unix."

und das ist doch schon was (denn damit kann man immer Punkten)[66]. Um uns nicht allzusehr vom λ-Kalkül zu trennen nehmen wir ein naheliegendes Beispiel: wir programmieren uns einen "Taschenrechner" für den λ-Kalkül; das sollte wohl auch bei der ein oder anderen Übungsaufgabe helfen.

Also, was wollen wir? Zunächsteinmal über λ-Terme reden. Also stellen wir eine geeignete "Datenstruktur" zur Verfügung:

```
;; The term structure (mainly for documentation purpurse).
;; ----------------------------------------------------

(defun lc-app (term1 term2)
  "Return the internal representation of an application of two terms."
  (list 'app term1 term2))

(defun lc-app->l (term) (cadr term))
(defun lc-app->r (term) (caddr term))

(defun lc-var (internalvar-or-string &optional comment)
  "Return the internal representation of a variable, maybe with some comment.
If the first argument is a number that the variable might be bound, if
it is a string, the variable is a globally free one."
  (list 'var internalvar-or-string comment))

(defun lc-var->name (term) (cadr term))
(defun lc-var->comment (term) (caddr term))

(defun lc-abs (internalvar intended-var-name body &optional comment)
  "Return the internal representation of an abstraction.
The first argument is the number of the variable to be bound, the second
is the prefered name for the bound variable the third argument is the
body of the abstraction and the optional fourth argument is a comment."
  (list 'abs internalvar intended-var-name body comment))

(defun lc-abs->var     (term) (cadr term))
(defun lc-abs->varname (term) (caddr term))
(defun lc-abs->body    (term) (cadddr term))
(defun lc-abs->comment (term) (car (cddddr term)))

(defun lc-tag (term)
```

[66]Wer sich an den Abendvortrag von Thomas erinnert, dem wird das merkwürdige Präsentationstool aufgefallen sein: vor einem schwarzen Hintergrund erschienen immer wieder entsprechende Beispiele mit korrekter Syntax-Hervorhebung, bei shell-Beispielen auch schön schrittweise. Zusätzlich sollte man wissen, daß der Vorführer meinte, er würde durcheinanderkommen, wenn er mit mehr als einer Datei zu tun hat. Also ist klar, was man will: der ganze Vortrag in einer Datei, aus der immer wieder Ausschnitte auf einem anderen Display erscheinen sollen (der Vorführer will natürlich weiterhin Überblick über den ganzen Vortrag haben!). Eine typisches Beispiel für den Wunsch nach einer Maßgeschneiderten Oberfäche — eine typische Anwendung für den `emacs`. Allerdings sollte man zugeben, daß hier auch ein geeigneter Window-manager von entscheidender Hilfe war.

```
  "Get the tag describing the term. Legal tags are 'app 'abs 'var."
  (car term))
```

Hierzu sollte einiges angemerkt werden. Zunächst einmal verrät schon die "Überschrift" worum es geht. Wir hätten uns auch merken können wie wir λ-Terme als Listen implementieren; aber wenn wir uns nach einiger Zeit wieder mit dem Code beschäftigen (müssen), so haben wir dies sicher vergessen. In so fern ist es praktischer, ein passendes Interface zu haben, das uns sagt, wie wir Abstraktionen/Applikationen/Variablen erzeugen und wie wir daraus die entsprechenden Informationen wieder extrahieren.

Zur Syntax ist noch anzumerken, daß `(defun f (x)...)` das ist, was man in `Scheme` als `(define f (lambda (x)...))` schreiben würde: die Variable `f` wird an eine explizit definierte Funktion gebunden. Der Köper besteht aus einer (endlichen) Liste von Ausdrücken; diese werden der Reihe nach ausgewertet. Wert der Funktion ist der zuletzt ausgewertete Ausdruck. Zu erwähnen ist noch, warum der erste Ausdruck eine Zeichenkette ist, wo doch ihr Auswerten keinen Seiteneffekt hat und der Wert ohnehin weggeschmißen wird: nun, wir handeles sich wieder um einen (`emacs`-spezifischen) magischen Ort. Dieser String wird als Dokumentation der betreffenden Funktion verwendet, die sich mit wenigen Tasten (default-mäßig `C-h f`) erreichen läßt; dies macht das ganze zu einer unheimlich bequemen Umgebung zur Erstellung von `emacs-lisp` Code.

Zur Wahl der Funktionsnamen sei erwähnt, daß es *keine* Möglichkeit gibt, interne Funktionen zu kapseln. Wir leben also in einem vollkommen flachen Namensraum! Um dadurch keine Konflikte zu erzeugen wähle man sich ein Präfix (in unserem Fall `lc` wie "lambda calculus") das es noch nicht gibt lasse alle eigenen Definitionen mit diesem Präfix anfangen und hoffe, daß niemand das selbe Präfix für ein Programm wählt, das man selber verwenden möchte.

Zur Implementierung ist anzumerken, daß wir Variablen intern als Nummern (aber nicht als deBruijn-Indices, sondern ganz primitiv `1 2 3...` statt der Buchstaben `x y z...`) implementieren. Da wir bei Substitutionen vorsichtshalber alles umbenennen werden, werden diese Nummern schnell nichts mehr mit der Eingabe zu tun haben. Um lesbarere Ausgaben zu produzieren merken wir uns an jeder Bindungsstelle (also bei jeder Abstraktion) einen String als "intendierten Namen" für die hier gebundene Variable. ' Diesen werden wir verwenden, wenn dadurch keine Konflikte entstehen.

Um mit dieser Notation vertraut zu werden definieren wir uns ein paar einfache Kombinatoren. `setq` weist einer Variable einen Wert zu.

```
;;; Examples
;;; --------

(setq lc-comb-K (lc-abs 1 "x"
                        (lc-abs 2 "y"
                                (lc-var 1)
```

```
                                "leere Abstraktion")
                        "Kombinator K"))

(setq lc-comb-S (lc-abs 1 "a"
                        (lc-abs 2 "b"
                                (lc-abs 3 "c"
                                        (lc-app (lc-app (lc-var 1) (lc-var 3))
                                                (lc-app (lc-var 2) (lc-var 3)
                                                        ))))
                        "Kombinator S"))
```

Als nächstes definieren wir uns einige grundlegende Funktionen, etwa die Menge der freien Variablen. Da wir bei der Substitution "neue" Variablen brauchen, definieren wir uns auch gleich eine Funktion die uns solche (in funktionaler Weise!) liefert: man nehme einfach die kleinste Variable (sprich: natürliche Zahl) die in der Liste der "verbotenen" Variablen *nicht* vorkommt. Etwas ähnliches brauchen wir bei der Ausgabe auch für Strings.

Zur Implementierung sollte man noch erwähnen, daß der Autor dieses Codes excessiv von der Lisp-Wahrheitswert-Konvention ("es ist alles wahr, was nicht die leere Liste ist") und dem Defaultwert für optionale Variablen (nämlich der leeren Liste!) gebrauch macht. Mit `cond` lassen sich in LISP sequentielle Fallunterscheidungen durchführen.

```
;; Basic operations:
;; -----------------

(defun lc-free-var (term)
  "Return a list containing all the free variables; more precisely only
internal number or string (for gloabally free vars)."
  (cond ((eq (lc-tag term) 'var) (list (lc-var->name term)))
        ((eq (lc-tag term) 'app) (append (lc-free-var (lc-app->l term))
                                         (lc-free-var (lc-app->r term))))
        ((eq (lc-tag term) 'abs)
         (remove (lc-abs->var term) (lc-free-var (lc-abs->body term))))))


(defun lc-new-var (var-list &optional start)
  "Return a variable not in the list of variables provided. The optional
second argument has to be a number and means that the output has to have
at least that value."
  (cond ((not start) (lc-new-var var-list 1))
        ((member start var-list) (lc-new-var var-list (+ 1 start)))
        (t start)))

(defun lc-new-varname (varname forbiddenvarnames)
  "Return a string that is similar to the first argument, but not a member
of the second argument."
  (cond ((member varname forbiddenvarnames)
```

```
         (lc-new-varname (format "%s'" varname) forbiddenvarnames))
        (t varname)))
```

Damit können wir die Substitution definieren. Wir erinnern uns, daß wir bei $(\lambda x.r)[y := s] = \lambda x.r[y := s]$ stets ohne Einschränkung davon ausgegangen sind, daß das x hinreichend neu war. Um dies in der Implementierung sicherzustellen benennen wir die gebundene Variable vorsichtshalber stets in eine ganz neue um.

```
(defun lc-subst (term varname new-term)
  "A clean substitution of the third argument for every occurence of the
second argument in the first."
  (cond ((eq (lc-tag term) 'var)
         (if (eq (lc-var->name term) varname) new-term term))
        ((eq (lc-tag term) 'app)
         (lc-app (lc-subst (lc-app->l term) varname new-term)
                 (lc-subst (lc-app->r term) varname new-term)))
        ((eq (lc-tag term) 'abs)
         (let ((new-var (lc-new-var
                         (append (lc-free-var new-term)
                                 (lc-free-var (lc-abs->body term))
                                 (list varname)
                                 ))))
           (lc-abs new-var (lc-abs->varname term)
                   (lc-subst
                    (lc-subst (lc-abs->body term)
                              (lc-abs->var term)
                              (lc-var new-var))
                    varname new-term)
                   (lc-abs->comment term))))))
```

Und damit definieren wir uns nun die β-Reduktion. Die Funktion nimmt einen Term und eine Position in einem Term und liefert als Ergebnis den entsprechend reduzierten Term. Man beachte, daß wir einen neuen Term zurückliefern und den alten Term nicht verändern. Seiteneffekte sind die Quelle alles Übels, also sollten wir sie nicht einsetzen, wo immer es sich vermeiden läßt.

Die Positionen innerhalb eines Terms sind implementiert als Liste aller der Verzweigungen wobei wir angeben, ob wir links abgebogen sind (dabei sehen wir den einen Sohn eines λ's als den linken an). Als Position eines Redex gilt die Position des betreffenden λ's. Dies wird gewisse Vorteile haben. Nur so viel sei schon veraten: alle Positionen die Redexe bezeichnen sind auf jeden Fall "wahr".

```
(defun lc-beta-reduce (term pos)
  "Reduce the beta-Redex described by pos (where pos has to describe the
lambda of the redex)."
  (cond ((null pos) term)
        ((equal pos '(t))
```

```
     (lc-subst (lc-abs->body (lc-app->l term))
               (lc-abs->var (lc-app->l term))
               (lc-app->r term)))
    ((eq (lc-tag term) 'abs)
     (lc-abs (lc-abs->var term)
             (lc-abs->varname term)
             (lc-beta-reduce (lc-abs->body term) (cdr pos))
             (lc-abs->comment term)))
    ((car pos)
     (lc-app (lc-beta-reduce (lc-app->l term) (cdr pos))
             (lc-app->r term)))
    (t
     (lc-app (lc-app->l term)
             (lc-beta-reduce (lc-app->r term) (cdr pos))))))
```

Für die Ausgabe werden wir die Darstellung eines Terms als Zeichenkette benötigen. Auch das läßt sich funktional bewerkstelligen. Erwähen sollte man noch, daß wir uns zwar über Einrückung keine Gedanken machen, aber vorsichtshalber genug Zeilenwechsel einbauen. Kommentare schließen wir durch Semikolon getrennt ans Ende der Zeile an. Außerdem fügen wir bei jedem λ noch die Position desselben ans Ende der Zeile als Kommentar an.

(Der aufmerksame Leser wird sich beim Betrachten des Codes wieder an den Default-Wert für optionale Argumente erinnern.)

```
(defun lc-term->string (term &optional pos)
  "Get a string-representation of a term"
  (cond
   ((eq (lc-tag term) 'app)
    (format "(%s\n%s)"
            (lc-term->string (lc-app->l term) (append pos (list t)))
            (lc-term->string (lc-app->r term) (append pos (list nil)))))
   ((eq (lc-tag term) 'var)
    (if (lc-var->comment term)
        (format "%s ;; %s\n"
                (lc-var->name term)
                (lc-var->comment term))
      (lc-var->name term)))
   ((eq (lc-tag term) 'abs)
    (let ((name (lc-new-varname (lc-abs->varname term)
                                (lc-free-var term))))
      (format
       "(lambda (%s)   ;; %s   ;; %s\n %s)"
       name
       (if (lc-abs->comment term) (lc-abs->comment term) "")
       pos
       (lc-term->string
        (lc-subst (lc-abs->body term)
                  (lc-abs->var term)
```

```
              (lc-var name))
        (append pos (list t))))))))
```

Damit hätten wir den ungetypten λ-Kalkül implementiert. Jetzt basteln wir uns noch eine schöne Benutzeroberfläche! Dazu müssen wir uns in die Niederungen der Seiteneffekte begeben. Wir brauchen globale Variablen[67] für den aktuellen Term, einen Puffer in den wir schreiben können (denn irgendwie müssen wir ja mit dem Benutzer reden und dabei möglichst wenig anderen "Anwendungen" dazwischenfunken) und natürlich eine Liste der alten Werte, denn irgendeine Form von `undo` hätte man dann doch ganz gerne.

Der entsprechende Befehl ist `defvar`. Es wird eine Variable der betreffende Wert zugewiesen, aber nur, wenn diese nicht schon definiert ist. Das (optionale) dritte Argument ist wieder der Dokumentationsstring.

Außerdem stellen wir eine Funktion bereit, die alle Werte wieder zurücksetzt; da diese vom Benutzer aus interaktiv (etwa via `M-x`) aufgerufen werden können werden soll müssen wir sie als `(interacive)` deklarieren[68]. Diese Funktion kann noch mehr leisten: wir wechseln in den Puffer und teilen mit das der Inhalt `Lisp`-Code ist (naja, nicht wirklich, aber die Syntax ist die von Lisp). Damit ist schon mal festgelegt, nach welchen Kriterien eingerückt und die Syntax hervorgehoben werden soll, wenn es so weit ist. Dieser Funktionalität wegen rufen wir `(lc-init)` auch gleich auf.

```
;;; Output
;;; ------

(defvar lc-current-term (lc-app (lc-app lc-comb-S lc-comb-K) lc-comb-K)
  "The term currently under consideration")

(defvar lc-output (get-buffer-create "*lc: Output buffer*"))
(defvar lc-history '() "The terms formerly under consideration")
(defun lc-init ()
  "initialize everything for using lc"
  (interactive)
  (set-buffer lc-output)
  (lisp-mode)
  (setq lc-history nil))
```

[67]Das klingt so wunderbar: "Diese Variable muß global sein. Wenn wir uns andererseits erinnern, daß eigentlich *alle* Variablen global sind und das noch dazu in einem flachen Namensraum, dann wird einem schon ganz anders. Andererseits ist das ungeschriebene Motto bei `emacs`-Lisp: "Hauptsache, es funktioniert!"

[68]Das ist ein Hauch von Abkapselung: nur entsprechend deklarierte Funktionen können direkt aufgerufen werden. Auf diese Weise kann man den Benutzer davor bewahren versehentlich interne Funktionen mit ungewünschten Seiteneffekten aufzurufen. Als netter Nebeneffekt ist die `Tab`-Ergänzung natürlich um so nützlicher je weniger Funktionen in Frage kommen. Der Grund für den Zwang zur Deklaration ist freilich ein ganz anderer: da es auch Funktionen mit Argumenten gibt, muß man mitteilen, wo diese herkommen; ob als Präfix, durch Frage (welche?!) im Minibuffer,...

```
(lc-init)
```

Nun kommen wir zur (ebenfalls interaktiven) `print`-Funktion: Man wechsle in den entsprechenden Puffer, lösche dessen gesamten Inhalt füge den String ein, der den Term darstellt und sage, daß die Syntax hervorgehoben werden soll `(font-lock-fontify-buffer)` und die Einrückung korrekt erstellt werden soll `(lisp-indent-region...)`. Außerdem ändern wir mit `local-set-key` mal schnell für diesen Puffer sämtliche Tastenbelegungen. Wir wollen ja hier sowieso nicht Text editieren, sondern Redexe umdrehen!

```
(defun lc-print ()
  "Print the current term in lc-output"
  (interactive)
  (set-buffer lc-output)
  (erase-buffer)
  (insert (lc-term->string lc-current-term))
  (font-lock-fontify-buffer)
  (lisp-indent-region (point-min) (point-max))
  (local-set-key "+" 'lc-reduce)
  (local-set-key "-" 'lc-back)
  (local-set-key "u" 'lc-back)
  (local-set-key "." 'lc-back)
  (local-set-key "l" 'lc-next-lo)
  (local-set-key "L" 'lc-lo-normalize)
  (local-set-key "r" 'lc-next-right)
  (local-set-key "R" 'lc-right-normalize)
  )
```

Jetzt ist der Zeitpunkt gekommen, an dem wir verraten sollten, wie es funktioniert, daß wir uns mit dem Cursor auf einen Redex stellen und einfach `+` drücken können: Nun gut, daür ist offenbar `lc-reduce` zuständig, aber wie arbeit sie? Erschreckend einfach, denn nicht umsonst haben wir ja die Position des λ's als Kommentar ans Zeilenende gefügt. Man gehe also zum Zeilenende, suche rückwärts nach dem nächsten Semikolon, vorwärts zum nächsten Leerzeichen und der `Lisp`-Ausdruck, der dort steht bezeichnet die Position des Redexes. So einfach geht das!

```
(defun lc-reduce ()
  "Reduce the current redex"
  (interactive)
  (setq lc-history (cons lc-current-term lc-history))
  (end-of-line)
  (search-backward ";")
  (search-forward " ")
  (setq lc-current-term
        (lc-beta-reduce lc-current-term
```

```
                      (read (current-buffer))))
  (lc-print))

(defun lc-back ()
  "Go one step back in history"
  (interactive)
  (if (null lc-history)
      (message "No history!")
    (setq lc-current-term (car lc-history))
    (setq lc-history (cdr lc-history))
    (lc-print)))
```

lc-back brauchte wohl keine großen Erklärungen; ebensowenig das Reduzieren eines oder aller Redexe nach einer gewissen Reduktionsstrategie.

```
(defun lc-next-lo ()
  "Reduce the left-most-outermost redex"
  (interactive)
  (setq lc-history (cons lc-current-term lc-history))
  (setq lc-current-term
        (lc-beta-reduce lc-current-term
                        (lc-lo-redex lc-current-term)))
  (lc-print))


(defun lc-lo-normalize ()
  "Normalize a term using the strategy lc-lo-redex"
  (interactive)
  (setq lc-history (cons lc-current-term lc-history))
  (lc-lo-normalize-aux)
  (lc-print))


(defun lc-lo-normalize-aux ()
  "Normalize a term using the strategy lc-lo-redex"
  (let ((redex (lc-lo-redex lc-current-term)))
        (if (null redex)
            "Term normal."
          (setq lc-current-term (lc-beta-reduce lc-current-term redex))
          (lc-lo-normalize-aux))))

(defun lc-next-right ()
  "Reduce the right-most redex"
  (interactive)
  (setq lc-history (cons lc-current-term lc-history))
  (setq lc-current-term
        (lc-beta-reduce lc-current-term
                        (lc-right-redex lc-current-term)))
  (lc-print))
```

```
(defun lc-right-normalize ()
  "Normalize a term using the strategy lc-right-redex"
  (interactive)
  (setq lc-history (cons lc-current-term lc-history))
  (lc-right-normalize-aux)
  (lc-print))


(defun lc-right-normalize-aux ()
  "Normalize a term using the strategy lc-right-redex"
  (let ((redex (lc-right-redex lc-current-term)))
        (if (null redex)
                "Term normal."
          (setq lc-current-term (lc-beta-reduce lc-current-term redex))
          (lc-right-normalize-aux))))
```

Zur Implementierung der Reduktionsstrategien sei noch mal auf die Wahrheitswertkonvention hingewiesen, die uns hier ein implizites `maybe` liefert: wir geben entweder den betreffenden Redex zurück oder `nil`, wenn der betreffende Term normal ist.

```
;; Normalization
;; -------------

(defun lc-lo-redex (term)
  "return the left-most-outer-most redex in a given term"
  (cond ((eq (lc-tag term) 'var) nil)
        ((eq (lc-tag term) 'abs)
         (let ((body-redex (lc-lo-redex (lc-abs->body term))))
           (if body-redex
               (cons t body-redex)
             nil)))
        ((eq (lc-tag term) 'app)
         (if (eq (lc-tag (lc-app->l term)) 'abs)
             '(t)
           (let ((left-redex (lc-lo-redex (lc-app->l term))))
             (if left-redex
                 (cons t left-redex)
               (let ((right-redex (lc-lo-redex (lc-app->r term))))
                 (if right-redex
                     (cons nil right-redex)
                   nil))))))))

(defun lc-right-redex (term)
  "return path to the right-most-redex in a given term"
  (cond ((eq (lc-tag term) 'var) nil) ;; a variable is always normal
        ((eq (lc-tag term) 'abs)
```

```
     (let ((body-redex (lc-right-redex (lc-abs->body term))))
       (if body-redex
           (cons t body-redex)
         nil)))
    ((eq (lc-tag term) 'app)
     (let ((arg-redex (lc-right-redex (lc-app->r term))))
       (if arg-redex
           (cons nil arg-redex)
         (if (eq (lc-tag (lc-app->l term)) 'abs)
             '(t)
           (let ((fun-redex (lc-right-redex (lc-app->l term))))
             (if fun-redex
                 (cons t fun-redex)
               nil))))))))
```

Die abschließenden Beispiele brauchen wohl keine großen Erklärungen. Man beachte, wie die `interactive` Deklaration es vollkommen einfach ermöglicht im Minibuffer nach natürlichen Zahlen zu fragen.

```
;; Church-numerals
;; ---------------

(defun lc-cn (n)
  (lc-abs 1 "step"
          (lc-abs 2 "base"
                  (lc-cn-aux n))
          (format "Church numeral %d" n)))

(defun lc-cn-aux (n)
  (if (= n 0)
      (lc-var 2)
    (lc-app (lc-var 1) (lc-cn-aux (- n 1)))))

(setq lc-identity
      (lc-abs 1 "x" (lc-var 1) "Identity"))

(defun lc-n-m-I-example (n m)
  "Clear history and set the current term to n m I."
  (interactive "nFirst Church numeral: \nnSecond Church numeral: ")
  (setq lc-history nil)
  (setq lc-current-term
        (lc-app (lc-app (lc-cn n)
                        (lc-cn m))
                lc-identity))
  (lc-print))

;; Fixed-point combinators
;; -----------------------
```

```
(setq lc-T (lc-abs 1 "x" (lc-abs 2 "y"
                                 (lc-app (lc-var 2)
                                         (lc-app (lc-app (lc-var 1)(lc-var 1))
                                                 (lc-var 2)))
                                 )
                   "Kombinator T"))

(setq lc-theta (lc-app lc-T lc-T))

(defun lc-theta-example ()
  "Clear history and set the current term theta"
  (interactive)
  (setq lc-history nil)
  (setq lc-current-term lc-theta)
  (lc-print))

(defun lc-BBB-example ()
  "Clear history and set the current term BBB"
  (interactive)
  (setq lc-history nil)
  (setq lc-current-term lc-BBB)
  (lc-print))



(setq lc-B (lc-abs 1 "x" (lc-abs 2 "y" (lc-abs 3 "z"
                                               (lc-app (lc-var 1)
                                                       (lc-app
                                                        (lc-var 2)
                                                        (lc-var 3)))))
                   "Kombinator B" ))
(setq lc-BBB (lc-app (lc-app lc-B lc-B) lc-B))
```

12 TEX: Der Kreis schließt sich

Da der Autor diese Kursunterlagen tippt fällt ihm ein, daß eines der wichtigsten Beispiele kombinatorischer Logik noch nicht behandelt wurde. Der vorliegende Text ist mit TEX erstellt; doch da dies Gegenstand eines eigenen Kurses sein könnte, hier nur kurz der Beweis, daß TEX kombinatorisch vollständig ist:

```
\def\S#1#2#3{#1{#3}{#2{#3}}}
\def\K#1#2{#1}
```

Literatur

[1] Henk Barendregt. The type free lambda calculus. In Jon Barwise, editor, *Handbook of Mathematical Logik*, chapter D.7, pages 1091–1132. North-Holland Publishing Company, 1977.

[2] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[3] Felix Joachimski and Ralph Matthes. Standardization and confluence for a lambda calculus with generalized applications. In Leo Bachmair, editor, *Rewriting Techniques and Applications, Proceedings of the 11th International Conference RTA 2000, Norwich, UK*, volume 1833 of *Lecture Notes in Computer Science*, pages 141–155. Springer Verlag, 2000.

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