1 EINLEITUNG

Unabhängig von konkreten Programmiersprachen läßt sich das methodische Vorgehen beim Programmieren nach verschiedenen Programmierstilen klassifizieren.

Am weitesten verbreitet ist das prozedurale (von lat. procedere) Programmieren. Seine geschichtliche Entwicklung erstreckt sich von den ersten höheren Programmiersprachen, wie FORTRAN und ALGOL-60 über COBOL und PASCAL bis zu "C". Man formuliert auf abstraktem Niveau eine Folge von Maschinenbefehlen, die der Rechner nacheinander ausführen soll.

Daneben wurden eine Reihe von anderen Programmierstilen entwickelt:

Beim applikativen bzw. funktionalen Stil ist die Funktionsanwendung das beherrschende Sprachelement. Bereits früh entstand als erster Vertreter dieses Programmierstils die Sprache LISP. Sie hat inzwischen eine Reihe von moderneren Nachfolgesprachen gefunden. Dieser Programmstil hat sich insbesondere im Bereich der symbolischen Informationsverarbeitung und der Künstlichen Intelligenz (KI) durchgesetzt.

Beim prädikativen (logischen) Stil ist das Formulieren von prädikaten-logischen Formeln das beherrschende Sprachelement. Er wird ebenfalls im Bereich der "Künstlichen Intelligent", vor allem bei der Entwicklung von Expertensystemen, eingesetzt. Die bekannteste Sprache ist hierbei PROLOG.

Ebenfalls in diesem Bereich ist der objektorientierte Programmierstil entstanden, bei dem die Definition von Objekten mit Fähigkeiten zum Senden und Empfangen von Nachrichten im Vordergrund steht. Der bekannteste Vertreter ist die Sprache JAVA.

Viele Programmiersprachen unterstützen nur einen Programmierstil, z.B. FORTRAN den prozeduralen oder PROLOG den prädikativen. Von zunehmender Bedeutung sind aber auch Sprachen, die die Benutzung mehrerer Stile gestatten. In PASCAL programmiert man zwar üblicherweise prozedural; man kann aber auch den applikative Programmierstil einsetzen. In der Sprache LISP programmiert man überwiegend applikativ; es stehen aber auch der objektorientierte und der prozedurale Programmierstil zur Verfügung. Die Integration des prädikativen Stils bereitet prinzipielle Probleme und ist aktueller Gegenstand der Forschung.

Eine vergleichende Wertung verschiedener Programmierstile wird im Rahmen dieses Buches nicht erfolgen. Im Vordergrund steht vielmehr die Vermittlung

der Ideen des applikativen Programmierens, ihre programmiersprachlichen Ausprägungen, kleinere Anwendungsbeispiele und ein Einblick in spezielle Implementierungstechniken für applikative Sprachen.

Die Verbindung zwischen den einzelnen Abschnitten erfolgt über die Theorie der rekursiven Funktionen, den ungetypten λ-Kalkül und die kombinatorische Logik. Aus diesem Grund wird auch auf getypte Sprachen, wie ML oder HOPE und ihre Nachfolger nicht weiter eingegangen.

Wenden wir uns nun einer Klärung und Abgrenzung der Begriffe "applikatives" und "funktionales" Programmieren zu!

Generell kann man Programme als Funktionen im mathematischen Sinne deuten, durch die Eingabedaten in Ausgabedaten abgebildet werden. Bei den proceduralen Sprachen, die ganz entscheidend durch die von Neumann-Rechnerarchitektur geprägt sind, lassen sich die einzelnen Konstrukte eines Programms selbst jedoch nicht als Funktionen über einfachen Bereichen deuten. Ihre Bedeutung ist von dem Begriff der Adresse eines Speicherplatzes abhängig und von der Vorstellung einer sequentiellen Programmausführung geprägt. Zu einer formalen Behandlung benötigt man eine aufwendigere mathematische Begriffsbildung.

Betrachtet man zwei Funktionen f, g, die ganze Zahlen in ganze Zahlen abbilden, d.h. $f, g : \mathbb{Z} \to \mathbb{Z}$, so gilt das Kommutativgesetz $f(a) + g(a) = g(a) + f(a)$.

Wegen der Seiteneffekte, die durch die Wertzuweisungen bewirkt werden, gilt dieser einfache Sachverhalt nicht bei Sprachen wie PASCAL, d.h. "functions" in PASCAL verhalten sich nicht wie mathematische Funktionen:

$\underline{program}$ $P\ (output);$
$\underline{var}$ $a : integer;$
$\underline{function}$ $f(x : integer) : integer;$
$\underline{begin}$ $a := x + 1; f := a\ \underline{end};$
$\underline{function}$ $g(x : integer) : integer;$
$\underline{begin}$ $a := x + 2;\ g := a\ \underline{end};$
$\underline{begin}$
$a := 0; write(f(a) + g(a));$
$a := 0; write(g(a) + f(a));$
$\underline{end};$

Man erhält verschiedene Ausgaben:

$$
\begin{array}{llll}
f(a) & a = 1, & f(a) & = 1 \\
g(a) & a = 3, & g(a) & = 3 \\
f(a) + g(a) = 1 + 3 & & & = 4
\end{array}
$$

bzw.

$$
\begin{array}{llll}
g(a) & a = 2, & g(a) & = 2 \\
f(a) & a = 3, & f(a) & = 3 \\
g(a) + f(a) = 2 + 3 & & & = 5
\end{array}
$$

Der Funktionswert $f(a)$ ist also abhängig von seinem Vorkommen im Programm. Bei einer mathematischen Funktion bezeichnet f(a) jedoch stets denselben Funktionswert.

Weiterhin hat man in der Mathematik eine konsistente Benutzung von Namen. Die Gleichung $x^2-2x+1 = 0$ hat die Lösung $x = 1$. Niemand käme auf die Idee zu sagen, daß eine Lösung vorliegt, wenn man für das erste Vorkommen von x den Wert 3 nimmt und für das zweite den Wert 5 ($9-2*5+1 = 0$). Eine Variable steht also in ihrem Gültigkeitsbereich stets für denselben Wert. Diese Eigenschaft erfüllt die Variable a in dem Programmbeispiel nicht, da ihr Wert durch $a := x + 1$ bzw. $a := x + 2$ geändert wird.

Bei Argumentationen über Programme spielt der Begriff der Gleichheit eine entscheidende Rolle. Da, wie gezeigt, nicht einmal $f(a) = f(a)$ gilt, sind Beweise von Aussagen über derartige Programme wesentlich komplizierter als Beweise über mathematische Funktionen.

Das applikative und funktionale Programmieren beruht nun auf der Idee, das gesamte Programm durchgehend mit Sprachkonstrukten zu programmieren, die jede für sich eine mathematische Funktion darstellen. Insbesondere werden also Seiteneffekte und Zeitabhängigkeiten, wie sie sich aus der sequentiellen Programmausführung ergeben, ausgeschlossen.

Die Adjektive "applikativ" bzw. "funktional" werden von verschiedenen Autoren in recht unterschiedlicher Bedeutung benutzt. Einige betrachten beide Begriffe als Synonyme, andere verstehen hierunter zwei unterschiedliche Programmierstile. Daher soll zunächst eine Klärung dieser Begriffe gegeben werden. Das Wort "applikativ" kommt vom lateinischen 'applicare' ($\approx$ anwenden). Applikatives Programmieren ist somit Programmieren, bei dem das tragende Prinzip zur Programmerstellung die Funktionsapplikation, d.h. die Anwendung von Funktionen auf Argumente, ist. Das Wort "funktional" kommt vom mathematischen Begriff des Funktionals bzw. höheren Funktionals, d.h. Funktion, deren Argumente oder Ergebnisse wieder Funktionen sind. Funktionales Programmieren ist somit Programmieren, bei dem das tragende Konzept zur Programmerstellung die Bildung von neuen Funktionen aus gegebenen Funktionen mit Hilfe von Funktionalen ist.

Geht man von diesen beiden Definitionen aus, die sich nur an der ursprünglichen Bedeutung der Begriffe "applikativ" und "funktional" orientieren, so sieht man unmittelbar ihre Gemeinsamkeit. Läßt sich eine Funktion auf ein Argument, welches selbst eine Funktion ist, anwenden, so ist sie ein Funktional. Die Bildung einer neuen Funktion aus gegebenen Funktionen mit Hilfe eines Funktionals ist nichts anderes als die Applikation (Anwendung) dieses Funktionals. Andererseits gibt es auch gute Gründe dafür, zwischen den Begriffen "applikativ" und "funktional" zu differenzieren, wenn man mit "applikativ" das Operieren auf elementaren Daten (z.B. ganzen Zahlen) charakterisiert bzw. mit "funktional" das Operieren auf Funktionen.

Betrachten wir dazu ein Beispiel, an dem diese Unterscheidung der Programmierstile deutlich wird:

$$Fak : \mathbb{N}_0 \to \mathbb{N}$$
$$Fak(x) = \begin{cases} 1 & \text{falls } x = 0 \\ x * Fak(\lambda - 1) & sonst \end{cases}$$

Wir führen eine Variable x ein, die eine natürliche Zahl als Wert besitzt. Die Fallunterscheidung wird durch das Resultat der Applikation der Funktion null : $\mathbb{N} \to \{true, false\}$ gesteuert, d.h. durch $null(x)$. Durch Applikation der Vorgängerfunktion Vorg : $\mathbb{N} \to \mathbb{N}_0$, der Multiplikation Mult : $\mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ und durch rekursive Applikation von Fak erhält man wieder einen Ausdruck, der einen elementaren Wert aus N besitzt. Der bedingte Ausdruck ist hier eine dreistellige Funktion $\{true, false\} \times \{1\} \times \mathbb{N} \to \mathbb{N}$. In Infi$\lambda$-Notation hat man den Ausdruck null(x) $\to$ 1; Mult(x,Fak(Vorg(x))). Er präsentiert für einen gegebenen Wert x die natürliche Zahl x! und ist keine Funktion. Erst durch explizite Abstraktion nach der Variablen x erhält man eine Funktion. In diesem Zusammenhang ist Church's λ-Notation gebräuchlich:

$$\lambda x.null(x) \to 1; Mult(x, Fak(Vorg(x))).$$

Da nun eine Funktion definiert ist, kann sie benannt werden:

$$Fak = \lambda x.null(x) \to 1; Mult(x, Fak(Vorg(x))).$$

Da in dieser Funktion keine Seiteneffekte auftreten und keine Sequentialisierung vorliegt, kann in jeder Applikation, z.B. Fak(3), die Auswertung von anfallenden Teilausdrücken in beliebiger Reihenfolge und auch parallel erfolgen.

$$
\begin{aligned}
& Null(3) \rightarrow 1; Mult(3, Fak(Vorg(3))) \\
= \;& Null(3) \rightarrow 1; Mult(3, Fak(2)) \\
= \;& Null(3) \rightarrow 1; Mult(3, Null(2) \rightarrow 1; Mult(2, Fak(Vorg(2)))) \\
= \;& Null(3) \rightarrow 1; Mult(3, false \rightarrow 1; Mult(2, Fak(Vorg(2)))) \\
= \;& Null(3) \rightarrow 1; Mult(3, Mult(2, Fak(1))) \\
= \;& Null(3) \rightarrow 1; Mult(3, Mult(2, Null(l) \rightarrow 1; Mult(1, Fak(Vorg(1))))) \\
= \;& Mult(3, Mult(2, Mult(1, Fak(0)))) \\
= \;& Mult(3, Mult(2, Mult(1, Nu11(0) \rightarrow 1; Fak(Vorg(0))))) \\
= \;& Mult(3, Mult(2, Mult(1, 1))) \\
= \;& Mult(3, Mult(2, 1)) \\
= \;& Mult(3, 2) \\
= \;& 6
\end{aligned}
$$

Kennzeichnend für das applikative Vorgehen ist also, daß man über dem Bereich der elementaren Daten Ausdrücke aus Konstanten, Variablen und Funktionsapplikationen bildet, die als Wert stets elementare Daten besitzen. Funktionen entstehen daraus durch explizite Abstraktion nach gewissen Variablen. Die Konstruktion einer Funktion aus gegebenen Funktionen stützt sich auf das applikative Verhalten der gegebenen Funktionen auf elementaren Daten

Funktionen lassen sich aber auch anders konstruieren, indem man gegebene Funktionen durch Applikation von Funktionalen unmittelbar zu neuen Funktionen verknüpft. Man bildet Ausdrücke aus Funktionen und Funktionalen, die selbst wieder Funktionen sind. Funktionen $f, g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ werden z. B. durch das Funktional

$$\circ : [\mathbb{N}_0 \rightarrow \mathbb{N}_0] \times [\mathbb{N}_0 \rightarrow \mathbb{N}_0] \times \; [\mathbb{N}_0 \rightarrow \mathbb{N}_0]$$

zu der Komposition $g \circ f$ von f und g verknüpft. Mit diesem Konzept kann man die Funktion Fak ebenfalls entwickeln.

Ausgangspunkt sind die Basisfunktionen Null, Mult und Vorg wie im vorherigen Beispiel, sowie die Identität Id und die konstante Funktion $\bar{1}$.

Aus der zu konstruierenden Funktion Fak und der Vorgängerfunktion bildet man mit Hilfe des Funktionals $\circ$ (Komposition) die Funktion

$$Fak \circ Vorg \; .$$

Mit Hilfe der Identitätsfunktion und des Funktionals [ ] (Konstruktion) entsteht hieraus die Funktion

$$[Id, Fak \circ Vorg]$$

Die Konstruktion ist definiert durch

$$[f_1, ..., f_n](x) = (f_1(x), ..., f_n(x)) \ .$$

Durch nochmalige Anwendung der Komposition auf die Basisfunktion Mult und die obige Funktion erhält man

$$Mult \circ [Id, Fak \circ Vorg].$$

Mit Hilfe des Funktionals (!!) "Bedingung" bildet man schließlich aus den drei Funktionen $Null$, $\bar{1}$ und $Mult \circ [Id, Fak \circ Vorg]$ die gesuchte Funktion

$$Null \rightarrow \bar{1}; Mult \circ [Id, Fak \circ Vorg]$$

und benennt sie

$$Fak = Null \rightarrow \bar{1}; Mult \circ [Id, Fak \circ Vorg].$$

Im Unterschied zum vorherigen Vorgehen ist hier eine Abstraktion nicht notwendig. Die einzige Applikation auf elementare Daten erfolgt bei der Anwendung von Fak auf ein konkretes Argument (Programmstart), z.B. Fak(3) .

$$\begin{array}{ll}
 & Null \rightarrow \bar{1}; Mult \circ [Id, Fak \circ Vorg](3) \\
= & Null(3) \rightarrow \bar{1}(3); Mult \circ [Id, Fak \circ Vorg](3) \\
= & Mult[Id, Fak \circ Vorg](3) \\
= & Mult([Id, Fak \circ Vorg](3)) \\
= & Mult(Id(3), Fak \circ Vorg(3)) \\
= & Mult(3, Fak(Vorg(3))) \\
= & Mult(3, Fak(2)) \\
 & \vdots \\
= & Mult(3, Mult(2, Mult(1, Fak(0)))) \\
= & Mult(3, Mult(2, Mult(1, Nu11(0) \rightarrow \bar{1}(0); ...))) \\
= & Mult(3, Mult(2, Mult(1, 1(0)))) \\
= & Mult(3, Mult(2, Mult(1, 1))) \\
 & \vdots \\
= & 6
\end{array}$$

Die beiden vorgestellten Programme für Fak unterscheiden sich zwar nicht sehr stark, da beide unmittelbar auf dem rekursiven Algorithmus für die Fakultätsfunktion beruhen. Generell zeigt sich jedoch, daß die unterschiedlichen Programmierstile oft zu verschiedenen Algorithmen zur Lösung desselben Problems führen (siehe z.B. den Unterschied der Funktion "Länge" in Kapitel 3.1.4 zur üblichen rekursiven Lösung).

Dieser Unterschied rechtfertigt eine Differenzierung in eine im strengen Sinne funktionale Programmierung und eine im strengen Sinne applikative Programmierung.

Funktionales Programmieren (im strengen Sinne)

Funktionales .Programmieren ist ein Programmieren auf Funktionsniveau. Ausgehend von Funktionen werden mit Hilfe von Funktionalen neue Funktionen gebildet. Es treten im Programm keine Applikationen von Funktionen auf elementare Daten auf.

Applikatives Programmieren (im strengen Sinne)

Applikatives Programmieren ist ein Programmieren auf dem Niveau von elementaren Daten. Mit Konstanten, Variablen und Funktionsapplikation werden Ausdrücke gebildet, die als Werte stets elementare Daten besitzen. Durch explizite Abstraktion nach gewissen Variablen erhält man Funktionen.

Diese Differenzierung ist jedoch in der Literatur noch nicht allgemein gebräuchlich. Hier wird allgemein von applikativer oder funktionaler Programmierung als Oberbegriff gesprochen.

Wenden wir uns nun der Frage zu, wie man funktionale bzw. applikative Sprachen implementieren kann. Es erweist sich als ungünstig, daß von-Neumann-Rechner für das Operieren mit Speicherzellen, in denen elementare Daten abgelegt sind, konzipiert sind, und nicht für die Programmierung mit Funktionen. Die spezifischen Möglichkeiten zur effizienten Ausführung von funktionalen bzw. applikativen Programmen können von einem von Neumann-Rechner nicht voll genutzt werden. Dennoch lassen sich solche Sprachen auf derartigen Rechnern erfolgreich implementieren, wie durch LISP-Implementationen und LISP - Maschinen gezeigt wurde. Ein Ausschöpfen aller Vorteile ist erst bei alternativen Rechnerarchitekturen zu erwarten.

Andererseits gibt es bereits eine Reihe von neu entwickelten Rechnerarchitekturen. Ihre Programmierung erfordert zur optimalen Ausnutzung Sprachkonzepte, wie sie z.B. bei der applikativen und funktionalen Programmierung in natürlicher Weise gegeben sind.

Es zeichnet sich hier eine Wechselwirkung ab, wie sie zwischen der von-Neumann- Architektur und herkömmlichen Sprachen schon lange besteht.

2 MATHEMATISCHE GRUNDLAGEN

2.1 BERECHENBARE FUNKTIONEN

2.1.1 Einleitung

Ein Algorithmus ist ein Verfahren, mit dessen Hilfe man die Antwort auf Fragen eines gewissen Fragenkomplexes nach einer vorgeschriebenen Methode erhält. Er muß bis in die letzten Einzelheiten eindeutig angegeben und effektiv durchführbar sein. Insbesondere muß die Vorschrift, die den Algorithmus beschreibt, ein Text endlicher Länge sein. Ein Algorithmus heißt abbrechend, wenn er für jede Frage des betrachteten Fragenkomplexes nach endlich vielen Schritten ein Resultat liefert. Andernfalls heißt der Algorithmus nicht abbrechend. Eine Funktion f : M $\rightarrow$ N, M,N Mengen, heißt berechenbar, wenn es einen abbrechenden Algorithmus gibt, der für a $\in$ M den Funktionswert f (a) $\in$ N als Resultat liefert.

Während es für manche Fragenkomplexe im Laufe der Zeit gelungen ist, abbrechende Algorithmen zu entwickeln, die eine Antwort auf jede Frage des Problemkreises liefern, sind solche Versuche bei anderen Fragenkomplexen immer wieder fehlgeschlagen. Gibt es also Problemkreise, für die es prinzipiell unmöglich ist, einen Algorithmus zu finden, der auf alle Fragen des Problemkreises eine Antwort liefert? Um solche Unmöglichkeitsbeweise überhaupt versuchen zu können, genügt die anschauliche Vorstellung vom Begriff des Algorithmus bzw. der Berechenbarkeit nicht; man benötigt mathematische Definitionen.

Die historisch früheste Präzisierung ist die Definition der "rekursiven Funktionen", die, anders als heute üblich, einen speziellen Kalkül mit Gleichungen verwendet (Gödel 1931, Gödel 1934, Herbrand 1931, Kleene 1936 a). Nachdem in (Kleene 1936 b) die Äquivalenz dieser Definition mit einer anderen Präzisierung, nämlich der Definition der "λ-definierbaren Funktion" (siehe Kapitel II-2) gezeigt wurde, formulierte Church (1936 b) seine berühmte These, in der er vorschlug, den Begriff der λ-definierbaren Funktion sowie der rekursiven Funktion mit dem anschaulichen Begriff der berechenbaren Funktion, d.h. der intuitiv berechenbaren Funktion, gleichzusetzen.

1936 führte Turing (1936) den Begriff der Turing-Maschine zur operationalen Präzisierung des Begriffs "Algorithmus" ein und konnte zeigen, daß der daraus resultierende Begriff der "Turing-berechenbaren Funktion" mit dem der λ-definierbaren Funktionen äquivalent ist (Turing 1937).

Als ein weiterer Begriff, der sich als äquivalent mit den schon genannten Begriffen herausstellte, wurde von Kleene (1936 a) die μ-rekursiven Funktionen bzw. als Unterklasse davon die primitiv rekursiven Funktionen eingeführt. Auch alle nachfolgenden Bemühungen um Präzisierung des anschaulichen Begriffs der berechenbaren Funktionen haben stets zu Definitionen geführt, die sich als äquivalent zu einem der oben genannten Begriffe herausstellten. Dieses alles spricht für die Gültigkeit der Church'schen These, die heute in der Literatur bis auf ganz wenige Ausnahmen (Heymes 1961, §3) als adäquate mathematische Präzisierung des Begriffs der "berechenbaren Funktionen" akzeptiert wird. Wir benutzen sie in folgender Variante als Präzisierung des Begriffs "Algorithmus":

Jeder Algorithmus läßt sich durch eine Turing-Maschine verwirklichen.

Als weiteres Argument für diese nicht beweisbare These sei angeführt, daß man große Klassen von anschaulich gegebenen Algorithmen angeben kann, für deren Ausführung entsprechende Turing-Maschinen existieren. Überhaupt erscheint es plausibel, daß alle elementaren Schritte, die man bei der Ausführung eines anschaulich gegebenen Algorithmus (z.B. mit Bleistift und Papier) durchführt, auch mit Hilfe einer Turing-Maschine vollzogen werden können.

Eine ausführliche Darlegung der verschiedenen Berechenbarkeitsbegriffe und Beweise von Sätzen, aus denen ihre Äquivalenz folgt, findet man in dem Standardwerk von Hermes (1961).

Turing-Maschinen kann man hinsichtlich ihres prinzipiellen Aufbaus und ihrer Arbeitsweise als das abstrakte Konzept aller von Neumann-Rechenmaschinen ansehen.

A. Turing selbst hat mit großer Begeisterung an der Konstruktion der ersten elektronischen Rechenmaschinen mitgewirkt. Man kann also ein Programm für eine Rechenmaschine als ein Turing-Maschinenprogramm ansehen, durch das eine berechenbare Funktion von der Menge der Eingabedaten in die Menge der Ausgabedatendefiniert wird. Die Ausführung startet in einer Anfangskonfiguration, bei der sich der Rechner in einem Anfangszustand befindet und die Eingabedaten im Speicher abgelegt sind: Die Anweisungen des Programms definieren Konfigurationsübergänge, bei denen sich im allgemeinen der Rechnerzustand und die Speicherbelegung ändern. Wenn eine Endkonfiguration erreicht wird, charakterisiert durch einen Endzustand des Rechners, dann findet man im Speicher das Resultat der Ausführung des

Programms (Algorithmus). Diese Semantik von Programmen beschränkt sich jedoch nicht nur auf das maschinennahe Programmieren, z.B. mit einer Assemblersprache, sie hat auch die Definition der überwiegenden Mehrzahl der gebräuchlichen höheren Programmiersprachen wie FORTRAN, ALGOL 60, PASCAL und ihrer Nachfolgesprachen geprägt.Die Schwierigkeiten mit dem mathematisch recht unflexiblen Begriff der Turing-Berechenbarkeit äußern sich hier z.B., wenn man versucht, Sätze über FORTRAN-Programme zu beweisen, oder wenn man versucht, einen Kalkül für Programme zu entwickeln analog zum Rechnen mit algebraischen Ausdrücken. J. Backus hat 1977 in seiner Turing-Award Lecture (Backus 1978) in brillanter Weise die fatalen Auswirkungen der von Neumann- Rechnerarchitektur und damit letztlich der Turing-Berechenbarkeit auf den Prozeß der Programmerstellung analysiert und funktionales Programmieren als radikale Alternative gefordert.

In der mathematischen Logik hat man schon recht früh erkannt, daß man für die Untersuchung von Eigenschaften der berechenbaren Funktionen statt der Turing-Berechenbarkeit äquivalente, aber mathematisch leichter handhabbare Berechenbarkeitsbegriffefe benutzen sollte. Beim applikativen Programmieren definiert man Funktionen und verknüpft sie mit den Daten allein durch die Operation der Applikation einer Funktion auf Argumente. Ein hierfür zweckmäßiger Begriff der Berechenbarkeit sollte sich also unmittelbar auf Prinzipien zur Konstruktion von berechenbaren Funktionen aus gegebenen berechenbaren Funktionen und berechenbaren Basisfunktionen abstützen. Dieser Anforderung werden die Konzepte der μ-rekursiven Funktionen sowie die eng damit verwandten Konzepte der λ-definierbaren Funktionen bzw. der kombinatorisch definierbaren Funktionen in besonderer Weise gerecht.

Warnend sei jedoch schon hier erwähnt, daß man in der Regel die Menge der durch eine applikative Programmiersprache definierbaren Funktionen nicht mit der Menge der μ-rekursiven Funktionen identifizieren darf

2.1.2 Primitiv-rekursive Funktionen

Fast alle Funktionen, die beim praktischen Programmieren auftreten, gehören zu der Klasse der primitiv-rekursiven Funktionen. Diese ist jedoch eine echte Teilmenge der berechenbaren Funktionen. Erst durch eine Erweiterung zu den partiell rekursiven Funktionen erhält man einen Begriff, der sich als äquivalent mit dem der Turing-berechenbaren Funktionen erweist.

Unter dem *Definitionsbereich* $\mathrm{dom(f)} \subseteq \mathrm{M}$ einer Funktion $\mathrm{M} \rightarrow \mathrm{N}$ versteht man die Menge $\mathrm{dom(f)} = \{\mathrm{x} \mid \mathrm{f(x)} \text{ ist definiert}\}$.

f heißt *partielle Funktion* genau dann, wenn $dom(f) \subset M$.
f heißt *totale Funktion* genau dann, wenn $dom(f) = M$.

Mit $\mathbb{N}_0$ wird der Menge der natürlichen Zahlen einschließlich 0 bezeichnet, $\mathbb{N}_0^k$ ist das k-fache kartesische Produkt $\underbrace{\mathbb{N}_0 \times \mathbb{N}_0 \times \ldots \times \mathbb{N}_0}_{k-mal}$ für $k \geq 2$ mit der Erweiterung $\mathbb{N}_0^1 = \mathbb{N}_0$.

Bei der Untersuchung von Begriffen der Berechenbarkeit hat sich gezeigt, daß man das Rechnen in den verschiedenen Datenbereichen, wie z.B. der Menge aller endlichen Worte über einem endlichen Alphabet, stets auf das Rechnen mit natürlichen Zahlen zurückführen kann. Wir betrachten deshalb unmittelbar n - stellige Funktionen f : $\mathbb{N}_0^n \to \mathbb{N}_0$ mit n $\geq$ 0. Im Falle n = 0 bezeichnet der Name der Funktionen zugleich den Funktionswert, also ein Element aus $\mathbb{N}_0$. Deshalb sind die nullstelligen Funktionen die Konstanten.

Definition 2.1-1

Eine Funktion f: $\mathbb{N}_0^n \to \mathbb{N}_0$ mit n $\geq$ 0 heißt genau dann *primitiv-rekursiv*, wenn sie eine der unter 1. - 3. definierten Grundfunktionen ist, oder wenn sie durch endliche Anwendung der Konstruktionsschemata 4. oder 5. definiert ist.

1. Nullfunktionen:

   $K^n : \mathbb{N}_0^n \to \mathbb{N}^0$, $n \geq 0$
   $K^n(x_1, ..., x_n) = 0$

2. Projektionen:

   $P_i^n : \mathbb{N}_0^n \to \mathbb{N}$, $n \geq 1, 1 \leq i \leq n$
   $P_i^n(x_{1,}..., x_n) = x_i$

3. Nachfolgerfunktion:

   $N : \mathbb{N}_0 \to \mathbb{N}_0$
   $N(x) = x + 1$

   Durch N(x) erhält man den Nachfolger von x im Sinne der Peano-Axiome für die natürlichen Zahlen. Er wird oft mit x' bezeichnet.

4. Einsetzung:

Seien $g_i : \mathbb{N}_0^n \to \mathbb{N}$ mit $1 \leq i \leq m$ und $h : \mathbb{N}_0^m \to \mathbb{N}_0$ primitiv-rekursiv. Dann ist auch folgende Funktion primitiv-rekursiv:

$f : \mathbb{N}_0^n \to \mathbb{N}$

$f(x_1, ..., x_n) = h(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., n_n))$

5. Primitive Rekursion:

   Seien $g : \mathbb{N}_0^n \to \mathbb{N}_0$ und $h : \mathbb{N}_0^{n+2} \to \mathbb{N}_0$ primitiv-rekursiv. Dann ist auch folgende Funktion primitiv-rekursiv:

   $f : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$

   (I) $\quad f(x_1, ..., x_n, 0) \quad = g(x_1, ..., x_n)$

   (II) $\quad f(x_1, ..., x_n, N(y)) = h(x_1, ..., x_n, y, f(x_1, ..., x_n, y))$

   Diese Definition ist vollständig, da sich jedes $x_{n+1} \in \mathbb{N}_0$ als N(y) darstellen läßt, sofern $x_{n+1} \neq 0$ ist.

   Jede primitiv-rekursive Funktion ist total; das Gegenteil gilt jedoch nicht.

   Die Schemata für Einsetzung und primitive Rekursion sind sehr spezieller Art; es ist daher im allgemeinen nicht zu erwarten, daß Verallgemeinerungen, wie z.B.Schachtelung von Rekursionen (Abschnitt 1.5), wieder zu primitiv-rekursiven Funktionen führen. Einige Verallgemeinerungen führen jedoch nicht aus der Klasse der primitiv-rekursiven Funktionen heraus, z.B. Rekursion über das erste Argument von h in 5. anstelle des letzten Arguments.

Beispiele 2.1-1

1. Die Summe $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $x + 0 = x$ und $x + y' = x + y + 1 = (x + y)'$ für $y \geq 0$ ist primitiv-rekursiv, denn $+(x, 0) = P_1^2(x, 0)$ nach 2. und $+(x, y') = H(x, y, +(x, y))$, so daß primitive Rekursion gemäß 5. vorliegt. Die Hilfsfunktion $H(x, y, z) = N(P_3^3(x, y, z))$ ist primitiv-rekursiv nach 4. und 3..

2. Das Produkt $\cdot : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $x \cdot 0 = 0$ und $x \cdot y' = x \cdot y + x$ für $y \geq 0$ ist primitiv-rekursiv. Es gilt $\cdot(x, 0) = K^2$ und $\cdot(x, y' = H(x, y, \cdot(x, y))$, so daß primitive Rekursion vorliegt. Die Hilfsfunktion $H(x, y, z) = +(P_1^3(x, y, z)z)$ ist primitiv rekursiv. $\cdot(x, y') = H(x, y, \cdot(x, y)) = +(P_1^3(x, y, \cdot(x, y)), \cdot(x, y)) = +(x, \cdot(x, y))$.

3. Die identische Funktion $id : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $id(x) = x$ ist primitiv-rekursiv nach 5. für $n = 0 : id(0) = K^0 = 0, id(y') = H(y, id(y)), H(y, z) = +(N(K^0), z)$.

   Bemerkung:

   Zur Einsparung von Hilfsfunktionen kann man o.B.d.A. festsetzen, daß bei primitiver Rekursion nach 5. einige der Argumente der Funktion h fehlen dürfen. Entsprechend darf man bei Einsetzung nach 4. in jeder derFunktionen g_i einige Argumente weglassen.

4. Die Funktion $cond : \mathbb{N}_0^3 \to \mathbb{N}_0$ mit $cond(x_1, x_2, x_3) = x_2$, falls $x_1 \neq 0$ und $cond(x_1, x_2, x_3) = x_3$, falls $x_1 = 0$ ist primitiv-rekursiv, da $cond(0, x_2, x_3) = id(x_3)$ und $cond(y', x_2, x_3) = id(x_2)$.

2.1.3 Primitiv-rekursive Prädikate

Diese Prädikate spielen eine besondere Rolle bei der Definition primitiv-rekursiver Funktionen. Sie bilden eine wichtige Teilklasse der entscheidbaren Prädikate.

Definition 2.1-2

Sei $\mathbb{B} = \{\underline{true}, \underline{false}\}$ die Menge der Wahrheitswerte. Ein n-stelliges, totales Prädikat $p : \mathbb{N}_0^n \to \mathbb{B}$, $n \geq 1$ heißt genau dann *primitiv -rekursiv*, wenn seine charakteristische Funktion χ_p primitiv- rekursiv ist:

$$\chi_p : \mathbb{N}_0^n \to \mathbb{N}$$

$$\chi_p(x_1, ..., x_n) = \begin{cases} 0 & \text{falls } p(x_1, ..., x_n) = \underline{false} \\ 1 & \text{falls } p(x_1, ..., x_n) = \underline{true} \end{cases}$$

Diese Definition ist durch einen leicht zu beweisenden generellen Zusammenhang motiviert: Ein Prädikat q ist genau dann (intuitiv) entscheidbar, wenn χ_q(intuitiv) berechenbar ist. Ein Prädikat $q : \mathbb{N}_0^n \to \mathbb{B}$ heißt (intuitiv) entscheidbar, wenn es ein Verfahren gibt, das für einen beliebigen Wert (a_1, ... ,a_n) feststellt, ob $q(a_1, ..., a_n)$ oder $\nearrow q(a_1, ..., a_n)$ (bezeichnet die Negation) gilt.

Beispiele 2.1-2

1. Die Prädikate true, false : $\mathbb{N}_0 \to \mathbb{B}$ mit $true(x) = \underline{true}$ bzw. $false(x) = \underline{false}$ sind primitiv-rekursiv, da $N(K^1(x))$ bzw.$K^1(x)$ die charakteristischen Funktionen sind.

2. Das Prädikat $zero : \mathbb{N}_0 \to \mathbb{B}$ mit $zero(0) = \underline{true}$ und $zero(x) = \underline{false}$ für $x \neq 0$ ist primitiv-rekursiv, da $\chi_{zero} = cond(x, 0, 1)$ die charakteristische Funktion ist.

3. Das Prädikat $one : \mathbb{N}_0 \to \mathbb{B}$ mit $one(1) = \underline{true}$ und $one(x) = \underline{false}$ für $x \neq 1$ ist primitiv-rekursiv, da $one(x) = zero(V(x))$. Die Vorgängerfunktion $V : \mathbb{N}_0 \to \mathbb{N}_0$ mit $V(0) = 0$ und $V(y') = y$ ist offensichtlich primitiv-rekursiv.

   Das Zusammenspiel von primitiv-rekursiven Prädikaten und Funktionen zeigt sich, wenn man Funktionen durch die Angabe von Alternativen definiert. Zur Abkürzung benutzen wir

$$if\ p(\vec{x}) \qquad then\ f(\vec{x}) \quad else\ g(\vec{x}) = cond(\chi_p(\vec{x}), f(\vec{x}), g(\vec{x}))$$

Wenn $p(\vec{x})$ und $q(\vec{x})$ primitiv-rekursive Prädikate sind, dann sind auch die Negation $\nearrow p(\vec{x})$ sowie die Disjunktion $p(\vec{x}) \vee q(\vec{x})$ primitiv-rekursive Prädikate, da $\chi_{\nearrow p}(\vec{x}) = \underline{if}\ zero\ (\chi_p(\vec{x}))\ \underline{then}\ 1\ \underline{else}\ 0$, bzw. $\chi_{p \vee q}(\vec{x}) = \underline{if}\ zero\ (+(\chi_p(\vec{x}), \chi_q(\vec{x})))\ \underline{then}\ 0\ \underline{else}\ 1$. Wegen ihrer Reduzierbarkeit auf Disjunktion und Negation sind folglich auch die Konjuktion $p(\vec{x}) \wedge q(\vec{x})$, die Implikation $p(\vec{x}) \supset q(\vec{x})$ und die Äquivalenz $p(\vec{x}) \equiv q(\vec{x})$ primitiv-rekursive Prädikate.

Die Benutzung der Quantoren $\forall x$ und $\exists x$ führt im allgemeinen zu Prädikaten, die nicht mehr primitiv-rekursiv sind; es sei denn, der Bereich, über dem eine Variable x quantifiziert wird, wird auf die Werte $x = 0, ..., k$ beschränkt. Wenn Q ein n-stelliges primitiv-rekursives Prädikat ist, dann sind auch die folgenden Prädikate primitiv-rekursiv:

$P(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n, x_{n+1}) = \forall x_i \in \{0, ..., x_{n+1}\} : Q(x_1, ..., x_n), 1 \leq i \leq n$

$\tilde{P}(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n, x_{n+1}) = \forall x_i \in \{0, ..., x_{n+1}\} : Q(x_1, ..., x_n), 1 \leq i \leq n)$,

wobei der jeweilige Wert von x_{n+1} als obere Schranke des Bereichs dient, über den quantifiziert wird.

Anschaulich läßt sich die Beschränkung der Quantoren damit begründen, daß sowohl P als auch $\tilde{P}$ in endlich vielen Rechenschritten berechenbar sein müssen. Man betrachte als Beispiel das Prädikat $Q(x)$, welches definiert ist durch $Qx = \exists y\ f(x, y) = 0$, wobei f primitiv-rekursiv ist und y unbeschränkt quantifiziert ist. Falls es zu einem gegebenen $\overline{x}$ kein $\overline{y}$ gibt mit $f(x, y) = 0$, müßte man für alle Werte von y die Werte $f(\overline{x}, y)$ berechnen, um dieses festzustellen. Es ist also im allgemeinen nicht zu erwarten, daß $Q(x)$ primitiv-rekursiv ist.

Zur Vervollständigung sei erwähnt, daß auch das Potenzieren x^y, die Fakultät $x!$ und die modifizierte Differenz $\lambda - y$ mit dem Zusatz $\lambda - y = 0$ für $x < y$ primitiv-rekursiv sind, weiterhin auch die Prädikate $x = y$ und $x < y$.

Für das Rechnen mit natürlichen Zahlen hat man also eine reichhaltige Auswahl von primitiv-rekursiven Funktionen und Prädikaten zur Verfügung. Jedoch gehören auch eine Reihe von berechenbaren Funktionen nicht zu der Klasse der primitiv-rekursiven Funktionen, insbesondere solche, die partiell sind.

2.1.4 Partiell rekursive Funktionen

Der Begriff "partiell rekursiv" tritt auch häufig als Oberbegriff für irgendeinen der verschiedenen, äquivalenten Berechenbarkeitsbegriffe auf. Hier betrachten wir eine spezielle Erweiterung der Klasse der primitiv-rekursiven Funktionen, von der sich zeigen läßt, daß sie genau die Turing-berechenbaren Funktionen enthält. Durch dieses Resultat ist letztlich die Identifizierung des speziellen Begriffs "partiell rekursiv" mit "berechenbar" schlechthin zu verstehen. Nach der Bemerkung zur Definition 2-1.2 hat man deshalb auch mit dem Begriff der partiell rekursiven Prädikate eine adäquate Formalisierung der (intuitiv) entscheidbaren Prädikate.

Definitionen 2.1-3

Eine Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ mit $n \geq 0$ heißt genau dann *partiell rekursiv*, wenn sie eine der in Definition 2-1.1 eingeführten Grundfunktionen (Nullfunktion K^n, Projektion P_i^n, Nachfolgefunktion N) ist, oder wenn sie durch endliche Anwendung von Einsetzung, primitiver Rekursion und unbeschränkter Minimierung definiert ist.

Bei der Einsetzung ist zu berücksichtigen, daß ein Funktionswert $h(g_1(x_1, ..., x_n),$ $... , g_m(x_1, ..., x_n))$ undefiniert ist, falls einer der eingesetzten Funktionswerte $g_i(x_1, ..., x_n)$ undefiniert ist. Weiterhin ist "primitiv-rekursiv" jeweils durch "partiell rekursiv" zu ersetzen.

Ein n-stelliges Prädikat $p : \mathbb{N}_0^n \to \mathbb{B}$, $n \geq 1$ heißt genau dann *partiell rekursiv*, wenn seine charakteristische Funktion χ_p partiell rekursiv ist:

$\chi_p : \mathbb{N}_0^n \to \mathbb{N}_0$

$$\chi_p(x_1, ..., x_n) = \begin{cases} 0 & \text{falls } p(x_1, ..., x_n) = \underline{false} \\ 1 & \text{falls } p(x_1, ..., x_n) = \underline{true} \\ undefiniert & \text{falls } p(x_1, ..., x_n) \; undefiniert \end{cases} .$$

Sei $p : \mathbb{N}_0^{n+1} \to \mathbb{B}$, $n \geq 1$ ein partiell rekursives Prädikat. Dann ist auch die folgende durch *unbeschränkte Minimierung* definierte Funktion f partiell rekursiv:

$f : \mathbb{N}_0^n \to \mathbb{N}_0$
$f(\vec{x}) = h(\vec{x}, 0)$
$h : \mathbb{N}_0^n \times \mathbb{N}_0 \to \mathbb{N}_0$
$h(\vec{x}, y) = \underline{if}\ p(\vec{x}, y)\ \underline{then}\ y\ \underline{else}\ h(\vec{x}, N(y))$.

Der Wert von $f(x)$ ist das kleinste $\dot{y} \in \mathbb{N}_0$ mit $p(\vec{x}, \dot{y}) = \underline{true}$ unter der Voraussetzung, daß $p(\vec{x}, y) = \underline{false}$ für $y < \dot{y}$.Der Wert von $f(\vec{x})$ ist undefiniert, falls gilt:

1) $p(\vec{x}, y) = \underline{false}$ für alle $y \in \mathbb{N}_0$ oder
2) $p(\vec{x}, y)$ ist für ein gewisses $\hat{y}$ undefiniert und $p(\vec{x}, y) = \underline{false}$ für $y < \hat{y}$.

Beispiele 2.1-3

1. Die Funktion $undef(x)$, die für alle $x \in \mathbb{N}_0$ undefiniert ist, ist partiell rekursiv, da sie aus dem primitiv-rekursiven Prädikat $false(x, y) = \underline{false}$ für alle $x, y \in \mathbb{N}_0$ durch unbeschränkte Minimierung definierbar ist.

2. Eine modifizierte Vorgängerfunktion $\tilde{V} : \mathbb{N}_0 \to \mathbb{N}_0$ mit $\tilde{V}(0)$ undefiniert und $\tilde{V}(y') = y$ ist partiell rekursiv, da sie durch primitive Rekursion definierbar ist.

3. Die Funktion $m : \mathbb{N}_0 \to \mathbb{N}_0$ mit

$$m(x) = \begin{cases} \frac{x}{2} & \text{falls } x \text{ gerade} \\ undefiniert & sonst \end{cases}$$

ist partiell rekursiv, da sie durch unbeschränkte Minimierung definierbar ist:

$m(x) = h(x, 0)$
$h(x, y) = \underline{if\ \ }(y + y) = x\ \underline{then}\ y\ \underline{else}(x, N(y))$

Von besonderem Interesse ist der Fall der unbeschränkten Minimierung, bei dem das Prädikat $p(\vec{x}, y)$ total definiert ist und es zu jedem $\vec{x}$ ein y gibt mit $p(\vec{x}, y) = \underline{true}$, da dann $f(\vec{x})$ total definiert ist. Diese Situation wird auch "Anwendung des μ-Operators im Normalfall " genannt (Hermes 1961), wobei der μ-Operator der Funktion $h(\vec{x}, y)$ entspricht. Wenn man nur Anwendungen des μ-Operators im Normalfall zuläßt, erhält man die Klasse der μ-rekursiven Funktionen (Kleene 1936 a). Offensichtlich ist jede μ-rekursive Funktion intuitiv berechenbar, und offensichtlich ist auch jede primitiv-rekursive

Funktion μ-rekursiv.

Sei $p(x)$ eine $n \geq 2$-*stellige* Konjunktion bzw. Alternative, dann sind als Verallgemeinerungen zulässig

1.) *Permutation* $q(x_1, ..., x_n) = p(x_{\pi(1)}, ..., x_{\pi(n)})$, π Permutation von $1, ..., n$ und

2.) *Identifizierung* $r(x_1, ..., k_{k-1}, k_{k+1}, ..., x_n) = p(x_1, ..., x_{k-1}, X_i, x_{k+1}, ..., x_n)$ mit $1 \leq i, k \leq n$.

Satz 2.1-1

Die Operationen der Negation, der verallgemeinerten Konjunktionen und Alternativen sowie der beschränkten Quantifizierungen führen von μ-rekursiven Prädikaten wieder zu $\mu-$rekursiven Prädikaten. Dasselbe gilt für die Einsetzung einer μ-rekursiven Funktion in ein μ-rekursives Prädikat. Eine durch Fallunterscheidung mit Hilfe von μ-rekursiven Funktionen und μ-rekursiven Prädikaten definierte Funktion ist μ-rekursiv.

Ein etwas überraschendes Ergebnis der Theorie der partiell rekursiven Funktionen ist, daß man zur Definition einer partiell rekursiven Funktion den (unbeschränkten) μ-Operator höchstens einmal auf ein primitivrekursives Prädikat anwenden muß.

Die Beziehung zu Turing-berechenbaren Funktionen ergibt sich aus folgendem Satz:

Satz 2.1-2

Eine n-stellige partielle Funkion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ ist genau dann partiell rekursiv, wenn sie Turing-berechenbar ist.

2.1.5 Vergleich der betrachteten Klassen von Funktionen

Durch die verschiedenen Berechenbarkeitsbegriffe sind Klassen von Funktionen definiert worden, die bezüglich der Mengeninklusion $\subseteq$ verglichen werden sollen. Sei $F = \{f | f : \mathbb{N}_0^n \to \mathbb{N}_0, \text{ partiell}\}$ die Menge aller n-stelligen (partiellen) Funktionen mit der Teilmenge P aller partiell rekursiven Funktionen, der Teilmenge M aller μ-rekursiven Funktionen und der Teilmenge PR aller primitiv-rekursiven Funktionen. Es soll nun gezeigt werden, daß die Inklusionen $F \supseteq P \supseteq M \supseteq PR$ echt sind. Nach dem vorangehenden Satz umfaßt P genau die Menge der Turing-berechenbaren Funktionen. Da diese Menge abzählbar ist, während F überabzählbar ist, folgt

$F \supset P$ Beispiele für nicht Turing-berechenbare Funktionen werden meist mit Hilfe des Akzeptierungsverhaltens von Turing-Maschinen konstruiert (Manna 1974, Hermes 1961). Als Abgrenzung für das Definieren von berechenbaren Funktionen durch Gleichungen ist ein Beispiel von Kalmár (Heymes 1961) erwähnenswert, in dem ein Gleichungssystem angegeben wird, durch das eine Funktion eindeutig definiert ist, ohne daß man imstande ist, aus dem System beliebige Funktionswerte effektiv zu berechnen.

Aus Beispiel 2-1.3.folgt $P \supset M$ (M umfaßt genau die totalen Turing-berechenbaren Funktionen). Die Inklusion $M \supset PR$ ergibt sich aus der Ackermannfunktion $A(x)$ (Ackermann 1928), die zwar μ-rekursiv ist, jedoch nicht primitiv- rekursiv (Heymes 1961).

Wir wollen diese Funktion etwas näher anschauen:

Man betrachte folgendes Gleichungssystem für $f : \mathbb{N}_0^n \times \mathbb{N}_0 \to \mathbb{N}_0$:
$f(0, y) = Y'$
$f(x', 0) = f(x, 1)$
$f(x', y') = f(x, f(x', y))$

Durch Induktion über x läßt sich zeigen, daß für $(x, y) \in \mathbb{N}_0^2$ der Funktionswert $f(x, y)$ wohldefiniert ist und auch berechnet werden kann. Die Ackermannfunktion $A : \mathbb{N}_0 \to \mathbb{N}_0$ ist definiert durch $A(x) = f(x, x)$. Das bei der Definition von f benutzte Rekursionsschema fällt wegen der Schachtelung nicht unter das für primitive Rekursion zulässige Schema oder eine seiner Modifikationen. Wie kann man nun einsehen, daß $A(x)$ nicht primitiv-rekursiv ist? Wenn man mit Hilfe eines Programms Funktionswerte $f(x, y)$ berechnet, so fällt das enorme Wachstum auf, insbesondere in Abhängigkeit von x. In der Tat majorisiert f alle primitiv -rekursiven Funktionen.

Lemma 2.1-1

Zu jeder primitiv-rekursiven Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ gibt es eine Konstante $c \in \mathbb{N}_0$ der Art, daß für alle $x_l, ..., x_n \in \mathbb{N}_0$ gilt:
$g(x_l, ..., x_n) < f(c, x_l + ... + x_n)$, falls $n > 0$ bzw. $g < f(c, 0)$, falls $n = 0$.

Nehmen wir nun an, daß $A(x)$ primitiv-rekursiv sei, dann gibt es also $c \in \mathbb{N}_0$ mit $A(x) < f(c, x)$ für alle $x \in \mathbb{N}_0$.

Daraus ergibt sich aber für $x = c$ der Widerspruch
$A(c) < f(c, c) \underset{Def.}{<} A(c)$

und es folgt, daß $A(x)$ nicht primitiv-rekursiv ist.

Hinsichtlich der zentralen Bedeutung des Definierens von Funktionen durch

Gleichungen beim funktionalen bzw. applikativen Programmieren sind die Ackermannfunktion und Kalmär′sBeispiel deutliche Hinweise darauf, daß man dabei mit unerwarteten Phänomenen zu rechnen hat. Insbesondere kann man nicht davon ausgehen, daß durch alle Programme, die in Form eines Gleichungssystems niedergeschrieben sind, eine wohldefinierte berechenbare Funktion repräsentiert wird.

2.1.6 Effektive Bereiche

Wir wollen nun der Frage nachgehen, unter welchen Voraussetzungen man den Begriff der berechenbaren Funktion auf andere Bereiche als den der natürlichen Zahlen übertragen kann. Von praktischer Bedeutung sind z.B. Zahlen vom Typ integer bzw. real oder strukturierte Daten wie Arrays, Records und Listen. Betrachten wir im intuitiven Sinne berechenbare Funktionen $f : M_1$ *und* M_2 über Mengen M_1 und M_2, so ist es einsichtig, daß man für das effektive Berechnen von Funktionswerten auf jeden Fall "vernünftige" Benennungen für die Elemente von M_1 und M_2 benötigt. Man möchte z.B. in endlich vielen Rechenschritten entscheiden können, ob zwei Benennungen b_1, b_2 dasselbe Element $x \in M_1$ bezeichnen. Die Zeichenreihen $\frac{1}{3}$ und $\frac{2}{6}$ sind offensichtlich Namen für denselben Dezimalbruch; wie soll man jedoch effektiv berechnen, daß die unendliche Zeichenreihe 1.999... und die endliche Zahlenreihe 2 denselben Wert bezeichen?

Definition 2.1-4

Sei Σ^* die Menge aller endlichen Zeichenreihen über dem endlichen Alphabet Σ und $B \subseteq \Sigma^*$ Sei M eine Menge und $b : B \to M$ surjektiv. (B, b) heißt effektives Bezeichnungssystem von M, wenn

a) für $\alpha \in \Sigma^*$ intuitiv entscheidbar ist, ob $\alpha \in B$ oder ob $\alpha \notin B$ gilt und wenn

b) für $\alpha, \beta \in B$ intuitiv entscheidbar ist, ob $b(\alpha) = b(\beta)$ oder ob $b(\alpha) \neq b(\beta)$ gilt.

Wenn es zu M ein effektives Bezeichnungssystem gibt, dann heißt M *effektiver Bereich*.

Der Begriff "entscheidbar" ist hierbei im intuitiven Sinne zu verstehen und kann nicht präzisiert werden. Aber in vielen Fällen ist man sich darüber einig, ob ein effektiver Bereich vorliegt (Oberschelp 1981). Man verlangt also, daß es für $x \in M$ mindestens einen Namen gibt, daß entscheidbar ist, ob ein Name vorliegt und daß es entscheidbar ist, ob zwei Namen dasselbe

Element bezeichnen.

Beispiele 2.1-3

1. Alle endlichen Mengen sind effektive Bereiche.

2. $\mathbb{N}_0$ ist ein effektiver Bereich. Effektive Bezeichungssysteme sind z.B. das Dezimalsystem, das Dualsystem und die Strichnotation für natürliche Zahlen mit Hilfe der Nachfolgerfunktion $(0, 0', 0'', 0''', ...)$.

3. $\mathbb{Z}$ mit α bzw. $-\alpha$ als Namen für positive bzw. negative ganze Zahlen und α als Namen für Elemente von $\mathbb{N}_0$ ist ein effektiver Bereich.

4. $\mathbb{Q}$ mit $\frac{\alpha}{\beta}$ bzw. $-\frac{\alpha}{\beta}$ als Namen für rationale Zahlen und α, β als Namen für Elemente von $\mathbb{N}_0$ bzw. von $\mathbb{N}$ ist ein effektiver Bereich.

5. Die überabzählbaren Mengen $\mathbb{R}$ und $\mathbb{C}$ sind keine effektiven Bereiche. Es sind keine effektiven Bezeichnungssysteme bekannt. Insbesondere bilden die unendlichen Dezimaldarstellungen kein effektives Bezeichnungssystem.

6. Die Menge $\mathbb{N}_0^k$ ist ein effektiver Bereich mit den Zeichenreihen $(a_1, ..., a_k)$ als Namen für k-Tupel und α_l,...,a_k als Darstellungenvon $\mathrm{b}(\alpha_i) \in \mathbb{N}_0$.

7. Analog zu 6. schließt man, daß die Menge aller endlichen Folgen $\{\alpha_i\}_{i=1}^k$ und die aller endlichen Teilmengen $(\alpha_1, ..., \alpha_k)$ von $\mathbb{N}_0$ effektive Bereiche sind.

   Bemerkung: Die Menge aller Folgen und die aller Teilmengen sind keine effektiven Bereiche!

8. Die Menge Σ^* aller endlichen Zeichenreihen über einem endlichen Alphabet Σ ist ein effektiver Bereich.

Bei Programmiersprachen fallen skalare Datentypen unter effektive Bereiche gemäß 1., der Datentyp integer unter 3., der Datentyp real unter 4. bzw. 5., wobei zu bemerken ist, daß real implementationsbedingt zu einem effektiven Teilbereich von $\mathbb{R}$ wird. Arrays fallen unter 6., Records bzw. Sets unter 7. und Strings unter 8.

Der folgende Satz liefert die Rechtfertigung dafür, berechenbare Funktionen nur über natürliche Zahlen zu betrachten anstatt über effektiven Bereichen.

Satz 2.1-3

Jeder effektive Bereich läßt sich umkehrbar eindeutig und in beiden Richtungen berechenbar auf die natürlichen Zahlen abbilden.

Beweisskizze: Die Namen des effektiven Bezeichnungssystems lassen sich lexikographisch anordnen und abzählen.

Definition 2.1-5

Sei M eine Menge. Eine injektive Funktion $gd :\rightarrow \mathbb{N}_0$ heißt Gödelisierung(Gödel 1931), wenn gd und gd^{-1} intuitiv berechenbar sind und der Wertebereich $gd(M) \subseteq \mathbb{N}_0$ entscheidbar ist.

Zu jedem effektiven Bereich gibt es also eine Gödelisierung. Geht man umgekehrt von einer Gödelisierung gd von M aus, und hat man mit (B, b) ein effektives Bezeichnungssystem von $\mathbb{N}_0$, so ist $(B', gd^{-1} \circ b')$ mit $(b(B') = gd(M)$ und mit b' als Einchränkung von b auf B' ein effektives Bezeichnungssystem von M.

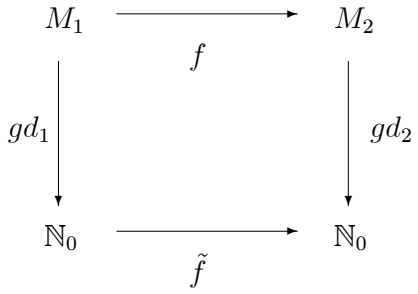
Satz 2.1-4

M ist ein effektiver Bereich genau dann, wenn es zu M eine Gödelisierung gibt.

Mit diesem Satz ist die Beziehung zwischen Berechenbarkeit in effektiven Bereichen und in $\mathbb{N}_0$ im wesentlichen geklärt.

Definition 2.1-6

Seien M_1, M_2 effektive Bereiche mit Gödelisierungen $gd1, gd2$ und sei $f : M_1 \rightarrow M_2$ eine partielle Funktion.



f heißt *partiell rekursiv* genau dann, wenn die Funktion $\tilde{f} : \mathbb{N}_0 \to \mathbb{N}_0$ partiell rekursiv ist:

$$\tilde{f}(x) = \begin{cases} gd_2(f(gd_1^{-1}(x))) & \text{falls } x \in gd_1(m_1) \text{ gerade} \\ 0 & sonst \end{cases}$$

Wenn f eine totale Funktion ist, heißt f auch (μ-) *rekursiv*.

Aus dieser Definition ergibt sich unmittelbar, daß f genau dann μ-rekursiv ist, wenn $\tilde{f}$ μ-rekursiv ist.

Motiviert wird die Definition 2-1.6 durch die Beobachtung, daß f genau dann intuitiv berechenbar ist, wenn $\tilde{f}$ intuitiv berechenbar ist.

Man findet häufig Definitionen des Begriffs der berechenbaren Funktion, bei denen statt $\mathbb{N}_0$ die Menge Σ^* der endlichen Zeichenreihen über einem endlichen Alphabet Σ zugrunde gelegt wird; insbesondere über dem Alphabet $\{0, 1\}$, z.B. Manna (1974). Eine Rechtfertigung dafür ist, daß in dem obigen Diagramm jedes (intuitive) Berechnungsverfahren für f ein Berechnungsverfahren für $\tilde{f}$ bewirkt und umgekehrt.

Gödelisierungen sind in erster Linie als ein theoretisches Hilfsmittel zu verstehen. In Programmiersprachen stehen die benötigten effektiven Bereiche in der Regel unmittelbar zur Verfügung. Eine Ausnahme bilden hier der reine λ-Kalkül und die kombinatorische Logik. Eigentlich steht $\mathbb{N}_0$ in realen Rechenmaschinen gar nicht zur Verfügung, sondern der vorhandene effektive Bereich ist die endliche Menge aller Bitfolgen, die eine bestimmte Länge nicht überschreiten können.

Beispiel 2.1-5

Typisch für Gödelisierungen ist das folgende Verfahren für k-Tupel bzw. endliche Folgen der Länge k mit Hilfe von Primzahlen. Die Folge der Primzahlen ist primitiv-rekursiv berechenbar:

$Po = 2, P1 = 3, P2 = 5, P3 = 7, P4 = 11, ...$

Sei $gd : \mathbb{N}_0^k \to \mathbb{N}_0$ mit

$gd(< x_0, ..., x_{k-1} >) = \Pi_{i<k} P_i^{x_i+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdot ... \cdot P_{n-1}^{x_{n-1}+1}$

und $gd(< >) = 1$

Die Funktion gd ist primitiv-rekursiv und injektiv nach dem Satz von der eindeutigen Zerlegbarkeit in Primzahlpotenzen. Die Umkehrfunktion gd^{-1} ist primitiv-rekursiv.

Sei $z = gd(< x_0, ..., x_{k-1} >\}$. Dann erhält man x_i. durch $z \mapsto \exp(i, z) \dot{-} 1$

und somit das Tupel bzw. die endliche Folge $< x_1, ..., x_{k-1} >$. Mit $\dot{-}$ ist die Differenz im Bereich der natürlichen Zahlen bezeichnet:

$$\lambda - y = \begin{cases} \lambda - y & \text{falls } x \geq y \\ 0 & sonst \end{cases}$$

Der Wert von $exp(i, z)$ ist der genaue Exponent, mit dem P_i in z aufgeht, falls $z > 1$ ist. Die Menge $gd(\mathbb{N}_0^k) = \{z \mid z > 0 \wedge (\forall_i \leq z)(P_{i+1} \mid z \Rightarrow P_i \mid z)\}$ ist entscheidbar.

Als weiterführende Lektüre über Berechenbarkeit, Entscheidbarkeit und damit verwandte Themen seien die Bücher von Hermes (1961), Schoenfield (1967), Boolos(1980), Rogers (1967) und Manna (1974) genannt.

2.2 DER λ-KALKÜL

2.2.1 Einleitung

Der hier betrachtete, ungetypte λ-Kalkül (Church 1932, 1941) ist eine Theorie der Funktionen, die von der Vorstellung geprägt ist, daß eine Funktion $f : D \rightarrow B$ im wesentlichen eine Rechenvorschrift ist, mit der zu jedem Argument $x \in D$ der Funktionswert $f(x) \in B$ berechnet wird, falls dieser existiert. Diese Vorstellung unterscheidet sich wesentlich von dem in der Mathematik üblichen Begriff, bei dem eine Funktion eine Teilmenge von $D \times B$ ist, durch die eine Abbildung der Elemente des Definitionsbereichs D in den Bildbereich B als statische Beziehung festgelegt ist.

Bei Funktionen des ungetypten λ-Kalküls spielen deshalb Mengen als Definitions bzw. Bildbereiche zunächst gar keine Rolle, sondern man konzentriert sich ausschließlich auf den Aspekt der Rechenvorschrift. Das Wesentliche einer Funktion erfaßt man z.B. bei der konstanten Funktion K mit Wert c durch die Gleichung $K(x) = c$ oder bei der identischen Funktion id durch $id(x) = x$, ohne daß man sich dabei um konkrete Mengen als Definitionsbereich für x bzw. als Bildbereich kümmern müßte.

Die Analogie zwischen Funktionen des ungetypten λ-Kalküls und Programmen geht soweit, daß man Funktionen uneingeschränkt auch als Argumente benutzen kann, so wie man es vom Programmieren her kennt. Ein Merkmal der von-Neumann-Rechnerarchitektur ist die Aufhebung der Unterscheidung zwischen Programm und Daten; beides sind Bitfolgen. Es ist im ungetypten λ-Kalkül also zulässig, Selbstapplikationen der Art $f(f)$ zu benutzen, die beim mengenorientierten Funktionsbegriff nur unter speziellen Voraussetzungen über die betrachteten Funktionen zulässig sind. Selbstapplikation muß

nicht unbedingt zu Widersprüchen führen.

Die Begründer des λ-Kalküls und der damit eng verwandten Theorie der "Kombinatorischen Logik" (Curry 1930) strebten einerseits die Entwicklung einer allgemeinen Theorie der Funktionen an; andererseits suchten sie nach Erweiterungen der Theorie um logische Begriffe, so daß diese als Grundlage für die mathematische Logik gewählt werden können Bei der Klärung des Begriffs der "berechenbaren Funktion" hat der λ-Kalkül eine zentrale Rolle gespielt (Kap. 2 -1.1); später hat er sich auch z.B. in der Rekursionstheorie als nützliches Hilfsmittel erwiesen. Mit der zweiten Zielsetzung war man weniger erfolgreich: Bis heute ist eine allgemein akzeptierte Begründung der Logik auf der Basis des λ-Kalküls bzw. einer seiner Varianten noch nicht geglückt. Einen Eindruck von den auftretenden Problemen wird am Ende des Abschnitts 2-3.4 gegeben.

Die Renaissance des λ-Kalküls in der Informatik begann mit den bahnbrechenden Arbeiten von McCarthy (1960, 1962, 1963) und Scott (1969, 1972), durch die die Relevanz für die Programmierung gezeigt wurde bzw. theoretische Probleme im Kalkül bereinigt werden konnten. Man kann den λ-Kalkül als das Paradigma einer Programmiersprache ansehen. Hier lassen sich sehr viele Phänomene in übersichtlicher Weise darstellen und untersuchen, gleichsam unter "Laborbedingungen". Die Beziehungen zwischen dem λ-Kalkül und realen Programmiersprachen sind vielfältiger Natur (Landin 1965, Morris 1968, Reynolds 1970, Gordon 1973, Plotkin 1975). In erster Linie interessieren wir uns hier für den Kalkül als eine funktionale Programmiersprache mit ganz einfacher Syntax und einer mathematisch definierten Semantik (Scott 1972, Stoy 1977, Barendregt 1981, Hindley 1972). Darüber hinaus soll exemplarisch aufgezeigt werden, daß der λ-Kalkül ein Bindeglied zwischen Mathematik und Programmiersprachen ist.

Über den λ-Kalkül hat man einen relativ einfachen Zugang zur denotationellen Semantik à la Scott (1971), (Stoy 1977, Wadsworth 1976, Gordon 1979), da hier die mathematischen Strukturen noch relativ einfach sind und das tragende Konzept der "Fixpunktbildung" durchschaubar ist.

Häufig benutzt man auch nur die λ-Notation von Funktionen, ohne dabei den vollen Kalkül mit einem mathematischen Modell zu meinen. Man benutzt die Notation zur eleganten Bezeichnung von gegebenen Funktionen und verwendet Teile des Kalküls zur Beschreibung des applikativen Verhaltens dieser Funktionen (McCarthy 1962, Bauer 1981).

2.2.2 Der klassische λ-Kalkül

Die genaue Bedeutung von λ-Termen variiert geringfügig bei den verschiedenen Anwendungen des λ-Kalküls. Gemeinsam sind jedoch stets die folgenden intuitiven Vorstellungen:

1. Ein λ-Term $\lambda x\ M$ stellt eine einstellige Funktion dar. In Analogie zum Konzept der Funktionsprozeduren kann man die Variable x als einen formalen Parameter ansehen und M als den Funktionsrumpf, d.h. die Bezeichnung eines Ausdrucks, der den Funktionswert bestimmt. Als Argumente bzw. Funktionswerte können sowohl "elementare" Daten als auch Funktionen (λ-Terme) auftreten. Falls ausschließlich λ-Terme auftreten, so spricht man vom reinen λ-Kalkül, sonst von einem angewandten λ-Kalkül.

2. Ein λ-Term $(M\ N)$ bezeichnet die Applikation des Terms M auf den Term N, das Argument.

3. Ein λ-Term $((\lambda xY)A)$, in dem x in Y auftritt, kann reduziert (ausgewertet, konvertiert) werden, indem man im Gültigkeitsbereich von x für alle Vorkommen von x den λ-Term A substituiert. Der so aus Y erhaltene Term $Sub_A^x[Y]$ kann als der Funktionswert von λxY für das Argument A betrachtet werden.

4. Es gibt 3 Typen von λ-Termen: Konstanten und Variablen sind atomare λ-Terme, Terme der Art λxM heißen Abstraktionen oder kurz "Lambdas" und Terme der Art $(M\ N)$ heißen Applikationen.

Beispiele 2.2-1

1. Der Term $\lambda(xy)$ entspricht der Anwendung einer Funktion F auf das Argument $y : \forall F : (\lambda x(xy)F = (Fy)$.

2. Der Term λxY entspricht der Anwendung der konstanten Funktion mit Wert $y : \forall F : ((\lambda xy)F) = y$.

3. Der Term λx(x x) entspricht der Selbstapplikation einer Funktion F : $\forall F : (\lambda x(xx)F) = (FF)$.

Die zunächst merkwürdig erscheinende Einschränkung auf einstellige Funktionen geht auf einen Sachverhalt zurück, der unabhängig voneinander von Schönfinkel (1924) und Curry (1930) entdeckt wurde. Unter sehr allgemeinen

Voraussetzungen gilt nämlich, daß zu einer gegebenen n-stelligen Funktion $f : [D_1 \times D_2 \times ... \times D_n] \rightarrow D$ stets eine äquivalente Funktion f_{curry} existiert mit

$f_{curry} : [D_1 \rightarrow [D_2 \rightarrow [...[D_n \rightarrow D]...]]]$
$f_{curry} = \lambda x_1(\lambda x_2(...(\lambda x_n f(x_1, x_2, ..., x_n))...))$

und somit mehrstellige Funktionen auf einstellige zurückgeführt werden können. Der Trick besteht darin, nicht alle Argumente auf einmal zu übergeben, sondern sie einzeln, der Reihe nach zu "verbrauchen". Man bezeichnet dieses Vorgehen zu Ehren eines der Entdecker als "Curryfizieren" (engl. to curry).

Beispiel 2.-2

Die zweistellige Funktion $h(x, y) = \lambda - y$ wird repräsentiert durch den λ-Term $h^* = \lambda x(\lambda y(\lambda - y))$.

$((h^*a)b)(= (\lambda x(\lambda y(\lambda - y))a)b)$
$= (\lambda y(a - y)b)$
$= a - b = h(a, b)$

Bei der Anwendung von h^* auf a fällt ein funktionales Resultat λy(a-y) an, wenn man h^* als eine Funktionsprozedur deutet. Auf dieser Möglichkeit beruht das Curryfizieren; deshalb ist seine Anwendung in Sprachen wie PASCAL nicht möglich.

2.2.2.1 Elementare Begriffe Die Menge der λ-Terme wird, wie bei Programmiersprachen üblich, durch eine kontextfreie Grammatik G_λ definiert:

Definition 2.2-1

Sei $\mathbf{V}$ eine abzählbare Menge von Variablen und $\mathbf{C}$ eine abzählbare Menge von Konstanten mit $\mathbf{V} \cap \mathbf{C} = \emptyset$ ist. Falls $\mathbf{C} = \emptyset$ ist, spricht man vom *reinen λ-Kalkül*.

$G_\lambda = \{T_\lambda, NT_\lambda, L_\lambda, P_\lambda\}$
$T_\lambda = \{\lambda, (,)\} \cup \mathbf{V} \cup \mathbf{C}$ terminale Zeichen
$NT_\lambda = \{L_\lambda, V_\lambda, C_\lambda\}$ nichtterminale Zeichen

Das Axiom ist $\mathrm{L}_\lambda \in \mathrm{NT}_\lambda$. Die Menge P_λ besteht aus folgenden Produktionen:

$L_\lambda ::= V_\lambda \mid C_\lambda \mid (\lambda V_\lambda L_\lambda) \mid (L_\lambda L_\lambda)$
$V_\lambda ::= x \mid y \mid ...$ $x, y \in \mathbf{V}$
$C\lambda ::= a \mid b \mid ...$ $a, b \in \mathbf{C}$

Ein Satz der Grammatik G_λ heißt λ-*Term*. Die Menge aller λ-Terme wird mit Λ bezeichnet.

Ein λ-Term mit der Struktur $(\lambda V_\lambda L_\lambda)$ heißt *Abstraktion* (oder λ-Funktion oder kurz "Lambda").

Ein λ-Term mit der Struktur $(L_\lambda L_\lambda)$ heißt *Applikation*.

Ein λ-Term $x \in \mathbf{V}$ bzw. $a \in \mathbf{C}$ heißt atomarer λ-Term.

Beispiele 2.2-3

1) x — V_λ — L_λ

2) $(\quad \lambda \quad x \quad y \quad)$ — V_λ — $V_\lambda \; L_\lambda$ — L_λ

3) $(\quad \lambda \quad x \quad (\quad x \quad y \quad) \quad)$ — $V_\lambda \; V_\lambda$ — $V_\lambda \quad L_\lambda \; L_\lambda$ — L_λ — L_λ

Im λ-Kalkül sind die folgenden Konventionen üblich:

1. Vordere Zeichen $a, b, c, ...$ des Alphabets bezeichnen Konstanten.
1. Hintere Zeichen $x, y, z, ...$ des Alphabets bezeichnen Variablen.
   Großbuchstaben $M, N, L, ...$ bezeichnen beliebige λ-Terme.
2. Die syntaktische Gleichheit von $M, N \in \Lambda$ wird durch $M = N$ bezeichnet. Das Zeichen "= " ist für eine andere Relation reserviert.
3. Häufig wird die Variable eines Lambdas vom Rumpf durch " . " getrennt, d.h. $L ::= (\lambda V_\lambda . L_\lambda)$.
4. Zur Einsparung von Klammern dienen die Regeln:
   - Äußerste Klammerpaare dürfen fehlen
   - $MN_1...N_n$ bedeutet $(...((MN_1)N_2)...N_n)$ (Linksklammerung) ,
   - $\lambda x_1 x_2 ... x_n . M$ *bedeutet* $\lambda x_1 (\lambda x_2 ... (\lambda x_n \; M)...))$

Beispiele 2.2-4

1. $\lambda xy.\lambda x \equiv (\lambda x(\lambda y(\lambda x)))$

2. $\lambda xy.yx(\lambda z.z) \equiv (\lambda x(\lambda y((yx)(\lambda z\ z))))$

Für eine Variable x, die in einem Term M vorkommt, fragt man, ob sie dort durch ein λx gebunden ist bzw. anderenfalls in M frei vorkommt.

Definitionen 2.2-2

Für $M \in \Lambda$ ist die Menge $FV(M)$ der freien Variablen von M und die Menge $BV(M)$ der gebundenen Variablen von M induktiv definiert durch:
$FV(x) = \{x\}$, $FV(a) = \emptyset$
$FV(M\ N) = FV(M) \cup FV(N)$
$FV(\lambda xM) = FV(M) \backslash \{x\}$
und
$BV(x) = \emptyset$, $BV(a) = \emptyset$
$BV(M\ N) = BV(M) \cup BV(N)$
$BV(\lambda xM) = BV(M) \cup \{x\}$

Eine Variable x ist *frei* in M genau dann, wenn $x \in FV(M)$.

Eine Variable x ist *gebunden* in M genau dann, wenn $x \in BV(M)$.

Beispiele 2_2.5

x λxx $\lambda x(y$ $\lambda y\ (y\ x)\ )x$

↑ ↑ ↑ ↑ ↑ ↑

frei gebunden frei gebunden frei

Eine Variable x sei nicht frei in M. Dann gilt $x \in BV(M)$ <u>oder</u> $x \notin BV(M) \cup FV(M)$, d.h. x kommt in M gar nicht vor. Die Begriffe "gebunden" und "frei" sind also nicht komplementär

Definition 2.2-3

$M \in \Lambda$ heißt geschlossener λ-Term oder auch Kombinator genau dann, wenn $FV(M\} = \emptyset$ Die Teilmenge aller geschlossener λ-Terme ist $\Lambda^0 \subset \Lambda$.

Die klassische Semantik (Church 1941} des λ-Kalküls ist eine "Reduktionssemantik", bei der ein Term seine Bedeutung dadurch erhält, daß man in ihm der Reihe nach alle reduzierbaren Applikationen ausrechnet. Eine Analogie ist das Umformen von arithmetischen Ausdrücken nach den üblichen Regeln. Man benötigt einen Mechanismus, der das Einsetzen eines λ-Terms N für gewisse Vorkommen der Variablen x in dem λ-Term M vornimmt.

Definition 2.2-4

Die Substitution $Sub_N^x[M]$ des λ-Terms N für alle freien Vorkommen von x im λ-Term M ist wie folgt induktiv definiert:

la. $Sub_N^x[x] = N$

lb. $Sub_N^x[a] = a \qquad , a \in (\mathbf{Y} \cup \mathbf{C}) \backslash \{x\}$

2. $Sub_N^x[(M_1 M_2)] = (Sub_N^x[M_1] Sub_N^x[M_2])$

3. $Sub_N^x[\lambda x M] = \lambda x M$

4. $Sub_N^x[\lambda y M] = \begin{cases} \lambda y Sub_N^x[M] & \text{falls } y \notin FV(N) \text{ oder } x \notin FV(M) \\ \lambda z Sub_N^x[Sub_z^y[M]] & \text{falls } y \in FV(N) \text{ und } x \in FV(M) \end{cases}$
wobei z eine neue Variable ist,die weder in N noch in M vorkommt

Punkt 4. der Definition behandelt die Fälle, in denen man gebundene λ-Variablen systematisch umbenennen muß, damit Namenskonflikte vermieden werden. Eine analoge Situation findet man bei ALGOL-ähnlichen Programmiersprachen, wo systematisches Umbenennen beim Expandieren von Prozeduraufrufen als Bestandteil der "Kopierregel" verlangt wird. Viele Programmiersprachen, die auf dem λ-Kalkül beruhen, haben Fehler in der Implementation von Punkt 4. der Substitution!

Die Notwendigkeit von Umbenennungen wird deutlich im Vergleich mit der "naiven" Substitution $\overline{Sub}$, bei der gilt:

5. $\overline{Sub}_N^x[\lambda y M] = \lambda y \overline{Sub}_N^x[M] \qquad .$

Wir betrachten eine konstante Funktion $\lambda y x$ und benennen sie in eine konstante Funktion $\overline{Sub}_W^x[\lambda y x] = \lambda y w$ um. Wählen wir jedoch y statt w, so ergibt sich die identische Funktion $\overline{Sub}_y^x[\lambda y x] = \lambda y y$. Bei der korrekten Substitution erhält man auch in diesem Falle eine konstante Funktion:

$Sub_y^x[\lambda y x] = \lambda z Sub_y^x[Sub_z^x[x]] = \lambda z Sub_y^x[x] = \lambda z y$

Bei der naiven Substitution kann es leicht geschehen, daß freie Variablen in einem einzusetzenden Term durch ein λ parasitär gebunden werden und dadurch die urspriingliche Bindungsrelation verfälscht wird:

$\overline{Sub}_{(yz)}^x[\lambda y(xy)] = \lambda y((yz)y)$
$\qquad\uparrow \qquad\qquad\qquad\uparrow$
y frei y gebunden

$Sub_{(yz)}^x[\lambda y(xy)] = \lambda w[Sub_{(yz)}^x[Sub_w^y[(xy)]]]$
$= \lambda w[Sub_{(yz)}^x[(xw)]$
$= \lambda w((yz)w) \qquad , y$ freie Variable

Die grundlegende Definition des λ-Kalküls ist der Begriff der "λ-Konvertierbarkeit" von Termen, durch den eine Gleichheitsrelation eingeführt wird.

Definition 2.2-5

Terme $M, N \in \Lambda$ heißen *konvertierbar* bzw. *gleich* ($M = N$), wenn diese Aussage aus folgenden Axiomen und Deduktionsregeln ableitbar ist:

Axiomenschemata:

$\lambda x M = \lambda y Sub_y^x[M] \quad , \quad y \notin FV(M)$ (α-Konversion)

$(\lambda x M)N = Sub_N^x[M]$ (β-Konversion)

$M = M$ (Reflexivität)

Deduktionsregeln:

$N = M \Rightarrow N = M$ (Symmetrie)

$M = N,\ N = L \Rightarrow M = L$ (Transitivität)

$M = N \Rightarrow MZ = NZ$ (Gültigkeit von = in Linkskontexten)

$M = N \Rightarrow ZM = ZN$ (Gültigkeit von = in Rechtskontexten)

$M = N \Rightarrow \lambda x M = \lambda x N$ (schwache Extensionalität)

Der λ-Kalkül ist also eine Theorie $\boldsymbol{\lambda}$ mit Gleichungen $M = N$. Beweisbarkeit in $\boldsymbol{\lambda}$ wird mit $\boldsymbol{\lambda} \vdash M = N$ bezeichnet; meist jedoch nur durch $M = N$. Es gibt jedoch in $\boldsymbol{\lambda}$ selbst keine logischen Verknüpfungen; es ist ein reiner Kalkül mit Gleichungen.

Häufig spricht man informell über $\boldsymbol{\lambda}$ und benutzt dabei logische Verknüpfungen und Quantoren, z. B.

$\forall M : (\lambda x\ x)M = M$

$M = N \Rightarrow MM = NN \wedge MN = NM$

mit der Bedeutung

$\forall M \in \Lambda \qquad \boldsymbol{\lambda} \vdash (\lambda x.x)M = M$

$\boldsymbol{\lambda} \vdash \mathbf{M} = \mathbf{N} \Rightarrow \boldsymbol{\lambda} \vdash MM = NN$ und $\boldsymbol{\lambda} \vdash MN = NM$

Gängige Bezeichnungen für $\boldsymbol{\lambda}$ sind: λ -Kalkül, $\lambda\beta$-Kalkül, λK-Kalkül, λKβ-Kalkül. Im Gegensatz zu einer eingeschränkten Version des Kalküls (λI-Kalkül) ist in $\boldsymbol{\lambda}$ der Kombinator $K \equiv \lambda xy.x$ vorhanden.

Die Beziehung zur syntaktischen Gleichheit ergibt sich aus $M \equiv N \Rightarrow M = N$.

Die Umkehrung gilt jedoch nicht.

Beispiele 2.2-6

Bei zusätzlicher Verwendung der naiven Substitution gilt: $\forall M, N : M = N$. Das ist ein Hinweis, wie empfindlich der Gleichheitsbegriff im λ-Kalkül ist.

Beweis:

Sei $F \equiv \lambda xy.yx$. Es gilt mit Sub: $\forall M, N : FMN = NM$. Insbesondere gilt $Fyx = xy$. Andererseits gilt mit $\overline{Sub}$:

$Fyx \equiv ((\lambda x(\lambda y.yx))y)x = (\lambda y.yy)x = xx$

$\uparrow$ frei, $\uparrow$ gebunden

Aus $xy = xx$ folgt mit den Deduktionsregeln

$(\lambda xy.xy)IM = (\lambda xy.xx)IM$.

Dabei ist $M \in \Lambda$ ein beliebiger λ-Term und $I \equiv \lambda x.x$.

$IM = II$

$M = I$

Für $M, N \in \Lambda$ gilt also $M = I = N$. Logisch???

2.2.2.2 Reduktionsregeln des $\lambda - Kalküls$ Durch den Begriff der Reduktion soll das Berechnen des Funktionswertes von λxM für das Argument N präzisiert werden.

Definition 2.2-6

Ein λ-Term P wird zu einem Term P' reduziert, $P \to P'$, wenn einer der folgenden zwei Reduktionsschritte auf P oder einen Unterterm von P angewandt wird.

1. α-Reduktion (Systematisches Umbenennen)
   $\lambda xM \underset{\alpha}{\to} \lambda y Sub^x_y[M]$, $y \in FV(M)$
2. β-Reduktion (Kopierregel)
   $(\lambda xM)N \underset{\beta}{\to} Sub^x_N[M]$

Falls X durch eine endliche, eventuell leere Folge von Reduktionsschritten zu Y reduzierbar ist, schreibt man $X \overset{*}{\to} Y$.

Durch die Forderung $y \notin FV(M)$ in der α-Reduktion sollen unzulässige Umbenennungen folgender Art ausgeschlossen werden:

$\lambda x(xy) \underset{\alpha}{\to} \lambda y\ Sub^x_y[(xy)] = \lambda y(y\ y)$

$\uparrow$ frei, $\uparrow$ gebunden

Da $\underset{\alpha}{\overset{*}{\rightarrow}}$ eine Äquivalenzrelation ist, die die syntaktische Gleichheit $\equiv$ umfaßt, wird diese auf α-äquivalente Terme ausgedehnt. Wir identifizieren also z.B. $\lambda xx \equiv \lambda yy$ auf syntaktischem Niveau und betrachten im Kalkül keine α-Reduktionen. Die einzige relevante Anwendung des systematischen Umbenennens findet man explizit in Fall 4. der Definition von $Sub_N^x[M]$.

Die β-Reduktion gestattet die eigentliche Berechnung von Werten für λ-Terme. In der Terminologie ALGOL-ähnlicher Sprachen ist x der formale Parameter einer Prozedur mit dem Rumpf M, die durch $(\lambda xM)N$ mit dem aktuellen Parameter N aufgerufen wird. Die Prozedur selbst erhält jedoch keinen Namen.

Beispiele 2.2-7

1) Von besonderer Bedeutung sind die Terme
$I \equiv \lambda xx$, $K \equiv \lambda xy.x$, $S \equiv \lambda xyz.xz(yz)$
mit den charakteristischen Eigenschaften:

$$
\begin{aligned}
IM &\equiv (\lambda xx)M \underset{\beta}{\rightarrow} M \\
KMN &\equiv (((\lambda x(Vyx))\ M)\ N) \underset{\beta}{\rightarrow} ((\lambda y\ M)\ N) \underset{\beta}{\rightarrow} M \\
SMNL &\equiv (((\lambda x(\lambda y(\lambda z(xz(yz))))M)N)L) \\
&\underset{\beta}{\rightarrow} (((\lambda y(\lambda z(Mz(yz))))N)L) \\
&\underset{\beta}{\rightarrow} ((\lambda z(MZ(Nz)))L) \\
&\underset{\beta}{\rightarrow} ML(NL)
\end{aligned}
$$

2) Ein Beispiel mit Umbenennung beim Reduzieren:
$\lambda x(\lambda y(xy))(zy) \rightarrow Sub^x_{(zy)}[\lambda y(xy)] = \lambda w Sub^x_{(zy)}[Sub^y_{(w)}[(xy)]] = \lambda w Sub^x_{(zy)}[(xw)] = \lambda w(zyw)$.

Die Beziehung zwischen der Gleichheitsrelation (Def. 2-2.5) und der Reduzierbarkeit ist folgendermaßen:

Satz 2.2-1

Es gilt $X = Y$ genau dann, wenn Y aus X durch eine endliche eventuell leere Folge von Reduktionsschritten und invertierten Reduktionsschritten hervorgeht.

Diese Aussage ist einsichtig, da die Relation $=$ die kleinste durch $\underset{\beta}{\rightarrow}$ erzeugte Äquivalenzrelation ist.

2.2.2.3 Extensionale Gleichheit von Funktionen Von einem Gleichheitsbegriff kann man verlangen, daß funktional äquivalente Terme M, N mit $MV = NV$ für alle $V \in A$ gleich sind, also $M = N$ gilt. Die Definition 2-2.5 erfüllt diese Forderung jedoch nicht. Sei $M = \lambda x(yx)$ und $N \equiv y$. Es gilt zwar

$\forall V : MV = yV = NV,$

aber $M \neq N$.

Definition 2.2-7

Terme $M, N \in \Lambda$ heißen *extensional gleich*, wenn zusätzlich zu den Axiomenschemata und Deduktionsregeln aus Definition 2-2.5 die Deduktionsregel:

$\forall V : MV = NV \Rightarrow M = N$ (Extensionalität) gilt.

Als zugehörige Erweiterung des Reduktionsbegriffs hat man

3. η-Reduktion

$\lambda x(Mx) \underset{\eta}{\rightarrow} M \quad , x \notin FV(M).$

Die η -Reduktion entspricht einer Erweiterung von Definition 2-2.5 um das Axiomenschema:

$\lambda x(Mx) = M \quad , x \notin FV(M)$ (η-Reduktion).

Satz 2.2-2

Der $\lambda\beta$-Kalkül mit extensionaler Gleichheit ist äquivalent zum $\lambda\beta$-Kalkül, erweitert um das Axiomenschema der η-Reduktion.

Beweis:

1. Sei extensionale Gleichheit gegeben. Für die "einfache" Gleichheit gilt:
   $\forall V : \lambda x(Mx)V = MV$ falls $x \notin FV(M)$.
   Mit der Extensionalität folgt $\underline{\lambda x(Mx) = M, x \notin FV(M)}$.
2. Sei das Axiomenschema der η-Reduktion gegeben. Seien M, N Terme mit:
   $\forall V : MV = NV$

$$\begin{array}{rl} Mx = Nx & \text{mit } V \equiv x \text{ und } x \notin FV(MN) \\ \lambda x(Mx) = \lambda x(Nx) & \text{mit der Regel für schwache Extensionalität} \\ \underline{M = N} & \text{mit der Regel für } \eta\text{-Reduktion} \end{array}$$

.

Der wesentliche Unterschied zwischen dem $\lambda\beta$-Kalkül und dem oben betrachteten $\lambda\beta\eta$-Kalkül (kurz: $\lambda\eta$-Kalkül) sind also nur Gleichungen der Art

$\lambda x(Mx) = M$.

Die η–Reduktionen sind überwiegend von theoretischem Interesse (Hindley 1972, Barendregt 1981), da es vom Standpunkt des Programmierens her irrelevant ist, ob $\lambda x(Mx)$ oder M selbst auf ein Argument angewendet wird. Wir werden deshalb weiterhin den $\lambda\beta$-Kalkül betrachten und im Einzelfall darauf hinweisen, wenn die hier vorgestellten, elementaren Sachverhalte im $\lambda\beta\eta$ -Kalkül abweichend sind.

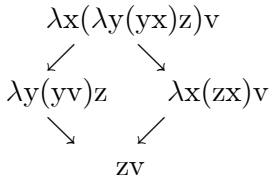
2.2.2.4 Die Church- Rosser Eigenschaft Beim Reduzieren von λ-Termen muß man mit den folgenden drei Situationen rechnen:

1. Man gelangt in eindeutiger Weise zu einem nicht weiter reduzierbaren Term, dem Resultat: $\lambda x(\lambda yz)zt \equiv (\lambda x(\lambda yz)z)t \rightarrow (\lambda yz)t \rightarrow z$

2. Man erhält kein Resultat, da das Reduzieren nicht abbricht. Der geschlossene Term $\Omega \equiv \lambda x(xx)\lambda x(xx)$ ist von dieser Art.

   $\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow ...$

3. Man hat verschiedene Reduktionsfolgen zur Auswahl



Nun stellt sich die Frage, ob die Reduktionsfolgen stets zu dem gleichen Resultat führen, sofern eines existiert. Da dieses der Fall ist (Satz 2-2.3), darf man nicht reduzierbare Terme als "Normalformen" bezeichnen.

Definition 2.2-8

1. Ein λ-Term λxMN heißt *β-Redex.*

   Ein λ-Term $\lambda x(Mx), x \notin FV(M)$ heißt *η-Redex.*

   Ein Redex ist ein β-Redex bzw. ein *η-Redex.*

2. Ein λ-Term M *ist in Normalform* genau dann, wenn kein Unterterm von M ein Redex ist.

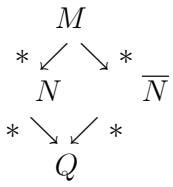
3. Ein λ-Term M *hat eine Normalform* genau dann, wenn ein λ-Term N in Normalform existiert und $M \underset{\beta,\eta}{\overset{*}{\rightarrow}} N$.

Der folgende Satz ist das berühmte Church-Rosser-Theorem für den λ-Kalkül. Es wurde in Church (1941) erstmalig formuliert. Der sehr umfangreiche Beweis wurde in Curry (1958) im Detail analysiert. Leichter zu verstehen sind die Beweise, die in Hindley (1972) bzw. Barendregt (1981) publiziert sind.

Satz 2.2-3 (Church-Rosser):

Sei $M \in \Lambda$ mit $M \overset{*}{\rightarrow} N$ und $M \overset{*}{\rightarrow} \overline{N}$, dann existiert $Q \in \Lambda$ mit $N \overset{*}{\rightarrow} Q$ und $\overline{N} \overset{*}{\rightarrow} Q$.

Skizze:

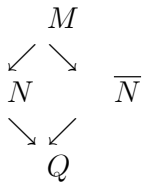


Korollar 1:

Seien N und $\overline{N}$ Normalformen von M, dann gilt $N \equiv \overline{N}$.

Beweis:

Aus dem Church-Rosser-Theorem folgt die Existenz von Q mit



Da N und $\overline{N}$ Normalformen sind, enthalten N und $\overline{N}$ keine Redices. Folglich gilt $N \equiv Q$ und $\overline{N} \equiv Q$.

Die Normalform eines λ-Termss ist also bis auf α-Reduktionen eindeutig bestimmt, sofern eine Normalform existiert. Die Frage nach der Existenz einer Normalform selbst ist unentscheidbar. Dies war eines der ersten Resultate über Entscheidbarkeiten und wurde von Church entdeckt (Church 1936 a,b).

Der in Definition 2-2.5 eingeführte Gleichheitsbegriff $X = Y$ wird durch folgende Variante des Church-Rosser-Theorems motiviert. X und Y werden

gleichgesetzt, da beide dieselbe Normalform besitzen.

Korollar 2: (Church-Rosser-Theorem, 2. Form)

Sei $M = N$, dann existiert Q mit $M \xrightarrow{*} Q$ und $N \xrightarrow{*} Q$.

Beweis:

Man induziert über die Anzahl n der Reduktionen und inversen Reduktionen, die sich aus $M = N$ ergeben. Mit $X \rightleftarrows Y$ ist gemeint, daß entweder $X \to Y$ oder $Y \to X$ gilt. Für $n = 1$ gilt $M \to N$ bzw. $M \leftarrow N$ und somit $Q \equiv N$ bzw. $Q \equiv M$.

Beim Induktionsschritt gehen wir aus von
$M \rightleftarrows ... \rightleftarrows N' \rightleftarrows N$.

Es gilt $M = N'$und nach Induktionsvoraussetzung existiert P mit $M \to P$ und $N' \to P$.

a) $N' \longleftarrow N$:
Dann gilt für $Q \equiv P$ offensichtlich $M \xrightarrow{*} Q$ und $N \xrightarrow{*} Q$.

b)$N' \to N$:
Wegen des Church-Rosser-Theorems existiert Q mit $P \xrightarrow{*} Q$ und $N \xrightarrow{*} Q$. Es gilt dann $M \xrightarrow{*} Q$ und $N \xrightarrow{*} Q$.

!!!Graphik!!!

Korollar 3:

Sei $M = N$ und sei N in Normalform, dann gilt $M \xrightarrow{*} N$.

Beweis:

Wegen Korollar 2 existiert Q mit $M \xrightarrow{*} Q$ und $N \xrightarrow{*} Q$.
Da N in Normalform ist, gilt $N \equiv Q$.

Korollar 4:

Sei $M = N$. Dann gilt entweder
a) M und N haben dieselbe Normalform oder
b) M und N haben beide keine Normalform.

Beweis:

a) Seien $\overline{M}$ bzw. $\overline{N}$ Normalformen von M bzw. N. Aus $M = N$ folgt $\overline{M} = \overline{N}$ und
mit Korollar 3 gilt $\overline{M} \xrightarrow{*} \overline{N}$. Da $\overline{M}$ Normalform ist, folgt $\overline{M} \equiv \overline{N}$.

b) Es existiere o.B.d.A. keine Normalform für M, aber $\overline{N}$ als Normalform von N.

Aus $M = N$ und $N \xrightarrow{*} \overline{N}$ folgt $M = \overline{N}$ und nach Korollar 3 gilt $M \xrightarrow{*} \overline{N}$ im Widerspruch zur Annahme, daß M keine Normalform besitzt.

Als Spezialfall erhält man:

Korollar 5:

Seien M, N Normalformen mit $M = N$, dann gilt $M \equiv N$.

Von besonderem Interesse ist die Negation des Korollars:

Seien M, N verschiedene Normalformen, d.h. $M \not\equiv N$, dann gilt $M \neq N$.

Man betrachte nun die beiden λ-Terme $K \equiv \lambda xy.x$ und $S \equiv \lambda xyz.xz(yz)$, die Normalformen sind mit $K \not\equiv S$. Dann gilt $S \neq K$. Es gibt also wenigstens zwei nicht ineinander konvertierbare Terme. Der λ-Kalkül mit $=$ als Gleichheit ist also konsistent in dem Sinne, daß nicht alle Terme gleich sind.

Wie leicht der Kalkül durch Erweiterungen der Gleichheitsrelation inkonsistente werden kann, zeigt folgendes Beispiel.

Beispiel 2.2-8

Wir erweitern $=$ um das Axiom $K = S$ und erhalten:

$$\begin{aligned} KXYZ &= SXYZ \\ XZ &= XZ(YZ) \\ \text{Sei } X \equiv Z \equiv I: \quad I &= YI \\ \text{Sei } Y \equiv KM: \quad I &= M \end{aligned}$$

Für $M, N \in \Lambda$ gilt also $M = I = N$, d.h. $=$ ist durch $K = S$ inkonsistent geworden.

Der folgende Satz zeigt, daß man im reinen λ-Kalkül Terme mit verschiedenen Normalformen nicht identifizieren darf und gibt dadurch Aufschluß über konsistente Gleichheitsbegriffe auf λ-Termen. Die Frage bleibt offen, ob man alle Terme ohne Normalform identifizieren darf.

Satz 2.2-4

Seien M, N Terme des reinen λ-Kalküls, die eine Normalform besitzen, dann gilt entweder schon $M = N$, oder die Hinzunahme des Axioms $M = N$ zum Gleichheitsbegriff $=$ führt zur Inkonsistenz.

Der Beweis beruht auf einem tiefliegenden Resultat im $\lambda\eta$ -Kalkül (Böhm 1968):

Wenn $\tilde{M} \not\equiv \tilde{N}$ verschiedene Normalformen sind, dann ist das Axiom $\tilde{M} = \tilde{N}$ eine inkonsistente Erweiterung der Gleichheit. Seien $\tilde{M}$ bzw. $\tilde{N}$ die Normalformen von M bzw. N.

a) $\overline{M} \not\equiv \overline{N}$:
Nach dem Vorangehenden ist dann $\overline{M} = \overline{N}$ eine inkonsistente Erweiterung und folglich auch $M = N$.

b) $\overline{M} \equiv \overline{N}$:
Dann gilt $M = \overline{M} = \overline{N} = N$

Bisher wurde der λ-Kalkül als ein formales System mit Gleichheitsbegriff und Reduktionsregeln betrachtet, wobei die Intention stets deutlich wurde, daß λ-Terme das applikative Verhalten von Funktionen modellieren sollten. Man kann nun fragen, welche konkrete Bedeutung einem beliebigen Term zukommen soll. Gesucht ist eine mathematische Struktur (Modell), in der man den λ-Termen eine Semantik zuordnen kann. Hier gibt es natürlich viele Möglichkeiten.

Eine erste, wenn auch noch unzulängliche Methode, ist die von Church, bei der man jedem λ-Term seine Normalform zuordnet, sofern vorhanden.

Definition 2.2-9

Sei M ein λ-Term. Seine Semantik $Val[M]$ ist gegeben durch

$$Val[M] = \begin{cases} N & \text{falls } N \text{ die Normalform } N \text{ hat} \\ undefiniert & \text{sonst} \end{cases} .$$

Es gilt offenbar

$$E = F \Longrightarrow Val[E] = Val[F] \quad .$$

Die Umkehrung gilt nur für den Fall, daß $Val[E]$ und $Val[F]$ definiert sind.

Als problematisch stellt sich heraus, alle Terme ohne Normalform als undefiniert anzusehen (s. Kap. 2-2.5).

Im Zusammenhang mit Programmiersprachen wie LISP ähnliche Sprachen ist die durch Val gegebene Reduktionssemantik durchaus brauchbar. Unter der Annahme, daß η-Reduktionen nicht erlaubt sind, berechnet man den Wert einer Funktion fn für ein gegebenes Argument z im wesentlichen durch das Bestimmen der Normalform von $fn(z)$ mit Hilfe der Substitution $Sub_y^x[M]$. Wenn man dabei nicht zu einer Normalform gelangt, so entspricht das intuitiv einer nicht terminierenden Rechnung, und es ist durchaus vernünftig, $Val[fn(z)]$ als undefiniert anzusehen.

2.2.3 Wahrheitswerte und logische Verknüpungen

In diesem und dem folgenden Abschnitt soll gezeigt werden, daß sich auch Daten mit den zugehörigen Operationen allein durch λ-Terme darstellen lassen. Damit rundet sich das Bild vom reinen λ-Kalkül als das Paradigma einer funktionalen.Programmiersprache ab.

Ein bedingter Ausdruck $\underline{if}\ B\ \underline{then}\ S_1\ \underline{else}\ S_2\ \underline{fi}$ mit $B \in \{\underline{true}, \underline{false}\}$ mit der Bedeutung, daß $B = \underline{true}$ zur Ausführung von S_1 führt und $B = \underline{false}$ zu der von S_2, kann man als dreistellige Funktion $if-then-else-fi\ (B, S_1, S_2)$ auffassen. Ersetzt man $if - then - else - fi$ durch $\lambda xyz.(xyz)$, $\underline{true}$ durch λxy.x und $\underline{false}$ durch λxy.y, so reduziert $\lambda xyz.(xyz)BS_1S_2$ zu (BS_1S_2) und für den Fall, daß $B = \underline{true}$ ist, zu S_1 und daß $B = \underline{false}$ ist, zu S_2. Die den Wahrheitswerten zugeordneten Terme wirken also wie Selektoren auf ihren Argumenten, wodurch die Darstellung der Wahrheitswerte im λ-Kalkül als "zweistellige Funktionen" T und F motiviert ist.

Definition 2.2-10

Seien $T, F \in \Lambda$ definiert durch:
i) $T \equiv \lambda xy.x$ und
ii) $F \equiv Xxy.y$.

Satz 2.2-5

Die Booleschen Funktionen Negation, Konjunktion und Disjunktion lassen

sich durch folgende Terme darstellen:

$$\begin{aligned} 1.) \quad & \underline{not} = \lambda x((xF)T) \\ 2.) \quad & \underline{and} = \lambda xy.((xy)F) \\ 3.) \quad & \underline{or} = \lambda xy.((xT)y) \end{aligned}$$

Beweis:

zu 1) $\underline{not}\ T \equiv \lambda x((xF)T)T = ((TF)T) = TFT \underset{i)}{=} F$

$\underline{not}\ F \equiv \lambda x((xF)T)F = ((FF)T) = FFT \underset{ii)}{=} T$

zu 2) Es werden zwei Hilfsrechnungen vorangestellt.

$\underline{and}\ T \equiv \lambda xy.((xy)F)T = \lambda y((Ty)F) \underset{i)}{=} \lambda yy = I$

Die identische Funktion I wurde in Beispiel 2-2.7 zusammen mit $K = \lambda xy.x$ eingeführt.

$\underline{and}\ F \equiv \lambda xy.((xy)F)F = \lambda y((Fy)F) \underset{ii)}{=} \lambda yF = KF$

Damit gilt nun:

$\underline{and}\ TT = IT = T$
$\underline{and}\ TF = IF = F$
$\underline{and}d\ FT = KFT = F$
$\underline{and}F = KFF = F$

zu 3) $\underline{or}\ T = \lambda xy.((xT)y)T = \lambda y((TT)y) \underset{i)}{=} \lambda yT = KT$

$\underline{or}\ F = \lambda xy.((xT)y)F = \lambda y((FT)y) \underset{ii)}{=} \lambda yy = I$

$\underline{or}\ TT = KTT = T$
$\underline{or}\ TF = KTF = T$
$\underline{or}\ FT = IT = T$
$\underline{or}\ FF = IF = F$

Wir haben hier ein erstes Beispiel für den applikativen Programmierstil. Charakteristisch ist in diesem speziellen Fall, daß man keine Trennung von Datenstrukturen und Kontrollstrukturen hat; beides sind λ-Terme. In den üblichen höheren proceduralen Programmiersprachen haben Funktionsprozeduren keine Funktionsprozeduren als Resultat. Deshalb hat der λ-Kalkül und der darauf basierende applikative Programmierstil kein unmittelbares Analogon in diesen Sprachen.

2.2.4 Arithmetik und λ-Definierbarkeit

Entsprechend zum Vorgehen im vorigen Abschnitt werden Terme eingeführt, die die natürlichen Zahlen darstellen, sowie Terme, durch die die Nachfolger- bzw. Vorgängerfunktion repräsentiert wird. Weiterhin muß man die Darstellung der Null von den übrigen Zahlen unterscheiden können. Wenn diese generellen Voraussetzungen erfüllt sind, kann man Arithmetik durch Systeme von λ-Termen definieren.

Definition 2.2-11

Jeder natürlichen Zahl $n \in N_0$ wird ein λ-Term $n \in \Lambda$ zugeordnet:
$\underline{0} \equiv \lambda xy.x$
$\underline{1} \equiv \lambda xy.(y\ \underline{0})$
$\vdots$
$\underline{n} \equiv \lambda xy.(y\ \ \underline{n-1})$.

Zahlen werden also durch paarweise verschiedene Normalformen dargestellt.

Satz 2.2-6

Es gibt λ-Terme $suc, pred$ und $zero$ mit

1. $suc\ \underline{n} = \underline{n+1}$
2. $pred\ \underline{n+1} = \underline{n}$ und $pred\ \underline{0}$ undefiniert
3. $zero\ \underline{0} = T$ und $zero\ \underline{n+1} = F$,

die die Nachfolgerfunktion, die Vorgängerfunktion und den Test auf 0 darstellen.

Beweis:

Zu 1) Man wähle $suc \equiv \lambda zxy.(yz)$. Dann gilt
$suc\ \underline{0} = \lambda xy.(y\underline{0}) = \underline{1}$ und
$suc\ \underline{n} = \lambda xy.(y\underline{n}) = \underline{n+1}$

Zu 2) Man wähle $pred \equiv \lambda x((x\Omega)I)$. Ω ist ein Term ohne Normalform (2-2.2.4) und $I = \lambda xx$ die identische Funktion. Man beachte, daß die Definition von $\underline{0}$ und T übereinstimmen.

$pred\ \underline{0} \equiv \lambda x((x\underline{\Omega})I)0 = (0\Omega)I \equiv T\Omega I = \Omega$

$$\begin{aligned} pred\ \underline{n+1} &\equiv \lambda x((x\Omega)I)\lambda x(\lambda y(y\underline{n})) \\ &= (\lambda x(\lambda y(y\underline{n}))\Omega)I \\ &= \lambda y(y\underline{n})I \\ &= I\underline{n} = \underline{n} \end{aligned}$$

Zu 3) Man wähle $zero \equiv \lambda x((xT)\lambda xF)$

$zero\ \underline{0} \equiv \lambda x((xT)\lambda xF)\underline{0} = (\underline{0}T)\lambda xF = T$

$$\begin{aligned} zero\ \underline{n+1} &\equiv \lambda x((xT)\lambda xF)\lambda x(\lambda y(y\underline{n})) \\ &= (\lambda x(\lambda y(y\underline{n}))T)\lambda xF \\ &= \lambda y(y\underline{n})\lambda xF \\ &= \lambda xFn = F \end{aligned}$$

Im folgenden wird zero gar nicht explizit auftreten, da wir uns eine Eigenschaft der gewählten Zahlendarstellung zunutze machen werden, die die benötigten Fallunterscheidungen für $n = 0$ und $n > 0$ enthält.

i) $$(\underline{n}\ M)N = \begin{cases} N & \text{falls } n = 0 \\ N\ \ \underline{n-1} & \text{falls } n > 0 \end{cases}$$

Da $\underline{0} = T$ ist, ist der Fall $n = 0$ bewiesen. Sonst gilt:

$(\underline{n}\ M)N \equiv (\lambda x(\lambda y(y\ \ \underline{n-1}))M)N = N\ \ \underline{n-1}$.

Satz 2.2-7

Es gibt λ-Terme sum und prod mit

1. $sum\ \underline{m}\ \underline{n} = \underline{m+n}$ und
2. $prod\ \underline{m}\ \underline{n} = \underline{m*n}$ $f\ddot{u}r\ m, n \in \mathbb{N}_0$

Beweis:

Zu l)
Die Summe $m + n$ soll unter Anwendung der Eigenschaft i) im Prinzip nach dem rekursiven Schema

$$\begin{aligned} sum\ \underline{m}\ \underline{n} := \underline{if}\ n = 0\ \underline{then}\ \underline{m} \\ \underline{else}\ suc(sum\ \underline{m}\ \underline{n-1})\ fi \end{aligned}$$

berechnet werden. Unter der Annahme, daß sich ein λ-Term für *sum* finden läßt, der folgender Gleichung genügt:

(*) $sum = \lambda xy.((yx)\lambda z(suc(sum\ x\ z)))$[1] ,

beweist man durch vollständige Induktion:

$sum\ \underline{m}\ \underline{0} = ((\underline{0}\ \underline{m})\lambda z(suc(sum\ \underline{m}\ z))) \underset{i)}{=} \underline{m}.$

Es gelte $sum\ m\ k = m + k$ für $k \geq 0$.

$$\begin{aligned} sum\ \underline{m}\ \underline{k+l} &= ((\underline{k+1}\ \underline{m})\lambda z(suc(sum\ \underline{m}\ z))) \\ &= \lambda z(suc(sum\ \underline{m}\ z))\underline{k} \\ &= suc(sum\ m\ k) \\ &= suc\ \underline{m+k} = \underline{m+k+1} \end{aligned}$$

Zu 2)

Unter der Annahme, daß sich ein λ-Term für *prod* finden läßt, der der Gleichung

(**) $prod = \lambda xy.((y\ \underline{0})\lambda z(sum(prod\ x\ z)x))$

genügt, beweist man durch vollständige Induktion:

$prod\ \underline{m}\ \underline{0} = ((\underline{0}\ \underline{0})\lambda z(sum(prod\ \underline{m}\ z)m)) = \underline{0}$.

Es gelte $prod\ \underline{m}\ \underline{k = m * k}$ für $k \geq 0$.

$$\begin{aligned} prod\ \underline{m}\ \underline{k+l} &= ((\underline{k+1}\ \underline{0})\lambda z(sum(prod\ \underline{m}\ z)\underline{m})) \\ &= \lambda z(sum(prod\ \underline{m}\ z)\underline{m})\underline{k} \\ &= sum(prod\ \underline{m}\ \underline{k})\underline{m} = \underline{m * (k+l)} \end{aligned}$$

Es muß nun noch bewiesen werden, daß es entsprechende λ-Terme für *sum* und prod gibt, da in der formalen Definition der Syntax von λ-Termen Rekursionen nicht zugelassen sind. Anhand der beiden Fälle soll ein generelles Verfahren zur Bestimmung von λ-Termen, die einer rekursiven Gleichung genügen, demonstriert werden.

Sei $A \equiv \lambda xy.y(xxy)$,und sei $Y \equiv AA$.

Dann gilt

ii) $YM = AAM = M(AAM) = M(YM)$ für $M \in \Lambda$.

Der Term YM ist also Fixpunkt von M; daher der Name *Fixpunktkombinator* für Y. Man betrachtet die rechte Seite der Gleichung von *sum* und kürzt

[1] Der Satz 2-2.4 gilt nicht, da eine der gleichgesetzten Normalformen syntaktisch eine "Konstante" ist.

sum durch s ab:

$\lambda xy.((yx)\lambda z(suc(sxz)))$.

Man abstrahiert nach der Funktion s

$F = \lambda sxy.((yx)\lambda z(suc(sxz)))$

und definiert:

$sum' \equiv YF$.

Es muß betont werden, daß sum' nur noch ein Name für den λ-Term YF ist; sum' tritt in YF nicht auf ! Die Gleichung (*) wird durch sum' erfüllt:

$$\begin{aligned} sum' &\equiv YF \\ &= F(YF) \\ &= \lambda sxy.((yx)\lambda z(suc(sxz)))\ (YF) \\ &= \lambda xy.((yx)\lambda z(suc((YF)x\ z))) \\ &= \lambda xy.((yx)\lambda z(suc(sum'x\ z))) \quad . \end{aligned}$$

Aus der rechten Seite der Gleichung (**) für $prod$ erhält man durch Abstraktion nach der Funktion prod, abgekürzt mit p,

$G \equiv \lambda pxy.((y0)\lambda z(sum'(pxz)x))$

und definiert

$prod' \equiv YG.$

Die Gleichung (**) wird durch $prod'$ erfüllt.

Schwierigkeiten kann man beim Rechnen mit sum' bzw.$prod'$ allerdings dann bekommen, wenn man anfängt, YF bzw. YG vollständig zu reduzieren, da diese Terme nach ii) keine Normalform haben. Andererseits sieht man deutlich, daß man deshalb YF bzw. YG nicht schlechthin als undefiniert ansehen darf.

Nach den vorgestellten Beispielen für das applikative Programmieren mit λ-Termen erscheint es zumindest plausibel, daß man alle μ-rekursiven Funktionen durch λ-Terme definieren kann.

Definition 2.2-12

Eine totale Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ heißt λ-definierbar genau dann, wenn es einen λ-Term F gibt mit

$F\underline{x_1}...\underline{x_n} = \underline{f(x_1, ..., x_n)}$ für $(x_1, ..., x_n) \in \mathbb{N}_0^n$.

Satz 2-2.8 (Kleene 1936)

Eine totale Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ ist genau dann λ-definierbar, wenn sie μ-rekursiv ist.

Beweisskizze:

1. Alle μ-rekursiven Funktionen sind λ-definierbar. Die Grundfunktionen sind λ-definierbar durch

   $K^n \equiv \lambda x_1 ... x_n.0$

   $P_i^n \equiv \lambda x_1 ... x_n.x_i$

   $N^i \equiv \lambda x.(suc\ x)$

   Wir nehmen Bezug auf Definition 2-1.1 und bezeichnen den zu einer Funktion f gehörenden definierenden Term mit F.

   Einsetzung:
   $F \equiv \lambda \vec{x}.H(G_1\vec{x})...(G_m\vec{x})$

   Primitive Rekursion:

   $$F \equiv \lambda \vec{x}k.\underline{if}\ zero\ k\ \underline{then}\ G\ \vec{x}$$
   $$\underline{else}\ \ H\vec{x}(pred\ \ k)(F\vec{x}(pred\ \ k))$$

   Minimierung: (Def. 2-1.3)
   $F \equiv \lambda \vec{x}.H\vec{x}0$

   $H \equiv Y(\lambda h\vec{x}y.if\ P\vec{x}y\ then\ y\ else\ h\ \vec{x}(suc\ y))$

2. Alle λ-definierbaren Funktionen sind μ-rekursiv.

   Sei f durch F λ-definiert. Da die Zahlen durch verschiedene Normalformen dargestellt sind, gilt $f(\vec{x}) = m$ genau dann, wenn $F\vec{x} = m$ im λ-Kalkül gilt. Damit kann man systematisch die Funktionswerte von f berechnen, da die Axiome für " = " ein effektives Verfahren liefern, um die Gültigkeit von $F\vec{x} = m$ festzustellen, wenn man normierte Ableitungen benutzt.

Damit man den Begriff der λ-Definierbarkeit auf partielle Funktionen erweitern kann, benötigt man zunächst im λ-Kalkül eine adäquate Darstellung der Situation, daß ein Funktionswert undefiniert ist.

2.2.5 Terme mit undefinierter Bedeutung

Church hat nur Termen, die eine Normalform haben, eine Bedeutung zugeordnet. Terme ohne Normalform sind bedeutungslos, d.h. undefiniert (Church 1941). Mit dieser Festsetzung kann man in dem von Church betrachteten λI-Kalkül konsistent arbeiten und insbesondere alle partiell rekursiven Funktionen darstellen. Der λI-Kalkül ist eine Teilmenge des λ-Kalküls, in der ein λ-Term mit Normalform nur Teilterme enthält, die ebenfalls eine Normalform haben. Ein λI-Term mit Bedeutung hat also ausschließlich Teilterme ebenfalls mit Bedeutung. Danach ist z.B. $KI\Omega$ kein λI-Term.

Wenn man nun alle Terme ohne Normalform als bedeutungslos ansieht, dann ist es natürlich, sie alle miteinander zu identifizieren. Das ist im λI-Kalkül eine konsistente Erweiterung des Gleichheitsbegriffs von Termen, jedoch nicht in dem hier betrachteten Kalkül (bei Church λK-Kalkül):

Beispiel 2.2-9

Sei $M \equiv \lambda x(xK\Omega)$, und sei $N \equiv \lambda x(xS\Omega)$.
Da M und N keine Normalform haben, gelte $M = N$. Damit folgt
$K = MK = NK = S$
und nach Beispiel 2-2.8 folgt, daß die Gleichheit " = " inkonsistent erweitert wurde.

Eine weitere Schwierigkeit entsteht bei der Komposition von partiellen Funktionen.

Beispiel 2.2-10

Sei $f(n) = 0$ und sei $g(n)$ undefiniert für $n \in \mathbb{N}_0$, dann ist auch $f(g(n))$ undefininert für $n \in \mathbb{N}_0$. Unter der Annahme, daß "undefiniert" durch "hat keine Normalform" dargestellt wird, wählt man
$F \equiv K\underline{0}$ und $G \equiv K\Omega$ für f bzw. g.

Aber
$F \circ G \equiv \lambda x(F(Gx)) = \lambda x(K\underline{0}(G\ x)) = \lambda x\underline{0}$

ist keine Funktion, durch die $f \circ g$ λ-definiert wird.

Da man mit dem Begriff der Normalform auch in Scott's mathematischen Modellen des λ-Kalküls Probleme hat (Wadsworth 1971, 1976), die auf Church's Identifikation von "undefiniert" mit "hat keine Normalform" beruhen, betrachtet man heute eine echte Teilmenge der λ-Terme ohne Normalform, nämlich

die Menge der "unlösbaren" λ-Terme bzw. der λ-Terme "ohne Kopfnormalform" als diejenigen λ-Terme, denen keine Bedeutung zukommt.

Definition 2.2-13

Ein geschlossener λ-Term $M \in \Lambda^0$ heißt *lösbar* (Barendregt 1971) genau dann, wenn es λ-Terme $N_1, ..., N_n$ gibt mit
$MN_1...N_n = I$.

Ein beliebiger λ-Term $M \in \Lambda$ heißt lösbar, wenn $\lambda x_1...x_k.M$ lösbar ist, wobei $\{x_1, ..., x_k\} = FV(M)$. $M \in \Lambda$ heißt *unlösbar*, wenn er nicht lösbar ist.

Da I eine Normalform ist, folgt aus $MN_1...N_n = I$ stets $MN_1...N_n \xrightarrow{*} I$. Ein lösbarer λ-Term hat also, als Funktion gedeutet, wenigstens einen Satz von Argumenten, für den ein Funktionswert definiert ist.

Beispiel 2.2-11

1. S ist lösbar durch $SIII = I$
2. $xI\Omega$ ist lösbar durch $\lambda x(xI\Omega)K = I$
3. Y ist lösbar durch $Y(KI) = KI(Y(KI)) = I$
4. Ω ist unlösbar, denn aus $\Omega\vec{N} \rightarrow M$ folgt $M = \Omega\vec{N'}$ mit $\vec{N} \rightarrow \vec{N}$; also kann $\Omega\vec{N} = I$ nicht erfüllt werden.

Ein λ-Term $\lambda x_1...x_n.xM_1...M_m$ mit $m, n \geq 0$ läßt sich höchstens zu einem λ-Term $\lambda x_1...x_n.xM'_1...M'_m$ mit $M_i \xrightarrow{*} M'_i$ für alle i reduzieren; d.h. der "Kopf" des Termes ändert sich nicht mehr.

Definition 2.2-14

Ein λ-Term M ist in *Kopfnormalform*[2] (Wadsworth 1971) genau dann, wenn $M \equiv \lambda x_1...x_n.xM_1...M_m, m, n \geq 0$. Für $n = 0$ ist M eine Applikation $xM_1...M_m$.

Die Variable x heißt *Kopfvariable* von M. Ein λ-Term M hat eine *Kopfnormalform*, wenn es ein M' in Kopfnormalform gibt mit $M = M'$.

[2]Der Begriff ist mißverständlich, da keine Teilmenge der Normalformen charakterisiert wird.

Wenn M keine Kopfnormalform ist, dann hat es die Gestalt
$M \equiv \lambda x_1 ... x_n.(\lambda x.M_0)M_1 ... M_m$ mit $n \geq 0, m \geq 1$,
und $(\lambda x.M_0)M_1$ heißt *Kopfredex* von M.

Ein λ-Term M hat genau dann eine Kopfnormalform, wenn die Folge der Reduktionen der Kopfredizes terminiert. Es ist klar, daß jede Normalform auch eine Kopfnormalform ist.

Satz 2.2-9 (Wadsworth):

M ist unlösbar genau dann, wenn M keine Kopfnormalform hat.
Insgesamt hat man folgende Beziehungen:

- "undefiniert" ist gleichwertig mit "unlösbar"
- "unlösbar" ist gleichwertig mit "hat keine Kopfnormalform"
- Aus "hat keine Kopfnormalform", folgt "hat keine Normalform"
- Aber aus "hat keine Normalform", folgt nicht "hat keine Kopfnormalform"

Es gibt eine Reihe von gewichtigen Gründen (Barendregt 1981) für die Identifizierung von "undefiniert" und "unlösbar". Insbesondere treten nun in Scott's Modellen des λ-Kalküls keine Probleme mehr auf.

Wir können nun eine adäquate Definition von λ-Definierbarkeit für partielle Funktionen geben.

Definition 2.2-15

Eine partielle Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ heißt λ -definierbar genau dann, wenn es einen λ-Term F gibt, so daß für $(x_1, ..., x_n) \in \mathbb{N}_0^n$ gilt:

$$F\underline{x_1}...\underline{x_n} = \begin{cases} \underline{f(x_1, ..., x_n)} & \text{falls } f(x_1, ..., x_n) \text{ definiert ist} \\ \underline{unlösbar} & \text{sonst} \end{cases} .$$

Satz 2.2-10

Eine partielle Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ ist genau dann λ-definierbar, wenn sie partiell rekursiv ist.

Einen Beweis findet man in Barendregt (1981).

Mit Hilfe des Begriffs der Kopfnormalform läßt sich nach Lèvy eine konsistente Semantik für den λ-Kalkül definieren, die sich unmittelbar auf den Kalkül und seine Begriffe abstützt. Man kann dieses Modell "syntaktisch" und "algebraisch" charakterisieren im Gegensatz zu Scott's Modellen, die auf topologischen Begriffen beruhen.

Man konstruiert auf einer speziellen Teilmenge N der Menge aller Kopfnormalformen eine algebraische Struktur N^∞ und eine partielle Ordnung $<$, die es gestattet, jedem λ-Term $M \in \Lambda$ einen "syntaktischen" Wert in N^∞ zuzuordnen, der mit $val[M] \in N^\infty$ bezeichnet wird (Lèvy 1976, 1977).

Anschaulich ist $val[M]$ der im allgemeinen unendliche Term, den man durch Reduzieren aller Redices von M erhält. Teile dieses unendlichen Terms, die keine Kopfnormalform besitzen, sind undefiniert. Etwas verkürzt dargestellt gilt für $M \in A$:

$$val[M] = \begin{cases} N & \text{falls } N \text{ Normalform von } M \text{ ist} \\ undefiniert & \text{falls } M \text{ keine Normalform hat} \\ \text{unlösbar} & \text{sonst} \quad . \end{cases}$$

$\overline{N}$ ist derGrenzwert in N^∞ von approximierenden Kopfnormalformen aus N für den vollständig reduzierten Term M.

2.2.6 Fixpunkte

Fixpunkte spielen bei allen Programmiersprachen die entscheidende Rolle bei der Semantikdefinition für Rekursionen und while-Schleifen. Im λ-Kalkül werden die wesentlichen Ideen bei der Benutzung von Fixpunkten besonders deutlich, da sie sich hier syntaktisch formulieren lassen, ohne expliziten Bezug auf eines von Scott's mathematischen Modellen des λ-Kalküls nehmen zu müssen.

Satz 2.2-11 (Fixpunkttheorem):

Für alle λ-Terme F gibt es einen λa-Term Φ mit
$F\Phi = \Phi$.

Φ heißt *Fixpunkt* von F.

Beweis:

Sei $W = \lambda x(F(xx))$ und sei $\Phi \equiv \mathbf{WW}$. Dann gilt

$\Phi \equiv \mathbf{WW} = \lambda x(F(xx)\mathbf{W} = F(\mathbf{WW}) = F\ \Phi$

Definition 2-2.16:

Ein geschlossener λ-Term M mit $MF = F(MF)$ für alle $F \in A$ heißt *Fixpunktkombinator*.

Da MF also stets ein Fixpunkt von F ist, ist der Name für M gerechtfertigt.

Satz 2.2-12

Die Terme Y_c und Y sind Fixpunktkombinatoren:
$Y_c \equiv \lambda f((\lambda x(f(xx)))\ \ (\lambda(f(xx))))$,
$Y \equiv \mathbf{AA}$ mit $\mathbf{A} \equiv \lambda xy.y(xxy)$.

Der Term Y heißt auch "*Curry's paradoxer Kombinator*" (2-3.4).

Beweis:

Sei $\mathbf{W} \equiv \lambda x(F(xx))$ für $F \in \Lambda$.
$Y_cF = \mathbf{WW} \stackrel{*)}{=} F(\mathbf{WW}) = F(Y_cF)$
$YF \equiv AAF = F(AAF) \equiv F(YF)$.

Man hat zwar $YF \underset{\beta}{\rightarrow} F(YF)$, aber nicht $)Y_cF \underset{\beta}{\rightarrow} F(Y_cF)$, da in *) eine inverse β-Reduktion vorgenommen wird !

Eine Anwendung von Y ist bereits im Beweis von Satz 2-2.7 erfolgt.

Beispiel 2.2-12

Wir betrachten die Gleichung
$F = \lambda xy.FyxF$

und suchen nach einem λ-Term für F, d.h. nach einer Lösung der Gleichung. Dieser λ-Term verhält sich dann so, als sei er rekursiv definiert.

Sei $C(f) \equiv \lambda xy.fyxf$, dann erhält man aus obiger Gleichung durch λ-Abstraktion nach F
$F = (\lambda f.C(f))F$.

Gesucht ist also ein Fixpunkt von $\lambda f.C(f)$ und ein solcher ist nach Satz 2-2.12 der gesuchte Term $Y(\lambda f.C(f))$.

Diese Art Gleichungen zu lösen, ist rein syntaktisch zu verstehen. Sie ist jedoch unter sehr generellen Voraussetzungen (Stoy 1977) verträglich mit der Bildung von kleinsten Fixpunkten in Scott's semantischen Modellen des λ-Kalküls.

Satz 2.2-13

Sei $C(f)$ die Notation für einen λ-Term mit der freien Variablen f. Dann existiert ein λ-Term F, für den gilt:

1. $F = (\lambda f.C(f))F$.
2. $F \xrightarrow{*} C(F)$.

Dabei ist $C(F) = Sub_F^f[C(f)]$.

Zum Beweis setzt man $F \equiv Y(\lambda f.C(f))$.

Mit diesem Satz wird die "rekursiv definierte Funktion" F als Fixpunkt von $\lambda f.C(f)$ charakterisiert. Teil 2. beschreibt das konkrete Rechnen mit solchen Fixpunkten. Eine korrekte Implementation von rekursiven Funktionen muß von diesem Satz ausgehen.

Rekursive Gleichungssysteme der Form

$$\begin{aligned} F_1 &= \lambda\vec{f}.C_1(\vec{f})F_1...F_n. \\ &\vdots \\ F_n &= \lambda\vec{f}.C_n(\vec{f})F_1...F_n \end{aligned}$$

mit $\vec{f} = f_1, ..., f_n$ werden sinngemäß über folgenden Satz gelöst:

Satz 2-2.14 (Fixpunkttheorem):

Für alle λ-Terme$\lambda\vec{f}.C_1(\vec{f}), ..., \lambda\vec{f}.C_n(\ \vec{f})$ mit $\vec{f} = f_1, ..., f_n$ gibt es λ-Terme $F_1, ..., F_n$ mit

$$\begin{aligned} F_1 &= \lambda\vec{f}.C_1(f)F_1...F_n. \\ &\vdots \\ F_n &= \lambda\vec{f}.C_n(f)F_1...F_n \end{aligned}$$

2.2.7 Reduktionsstrategien

Beim Reduzieren von λ-Termen hat man in der Regel die Auswahl zwischen verschiedenen Folgen von Reduktionsschritten. Wegen der Church-Rosser-Eigenschaft können verschiedene Reduktionsfolgen jedoch nie zu verschiede-

nen Normalformen führen; allerdings kann es geschehen, daß gewisse Reduktionsfolgen nicht abbrechen:

1. $KI\Omega \underset{\beta}{\rightarrow} I$

2. $KI\Omega \equiv KI((\lambda x.xx)(\lambda x.xx)) \underset{\beta}{\rightarrow} KI\Omega \underset{\beta}{\rightarrow} ...$

Von besonderem Interesse sind deshalb solche Reduktionsstrategien, bei denen garantiert ist, daß die Reduktion eines gegebenen λ-Terms mit seiner Normalform terminiert, sofern diese existiert. Wenn man beim Reduzieren eines λ-Terms streng von links nach rechts vorgeht, so wie in 1., dann bleiben die Argumente des am weitesten links stehenden Redex zunächst unausgerechnet. Bei der Reduktion dieses Redex kann es sich nun herausstellen, daß gewisse Argumente ganz aus dem λ-Term verschwinden, z.B. Ω in $KI\Omega$. Im Gegensatz zum Vorgehen in 2. verstrickt man sich beim Reduzieren von links nach rechts nicht in die Berechnung von Teiltermen ohne Normalform, die für die Normalform des gesamten λ-Terms ohne Bedeutung sind.

Ein Redex A ist *links* von einem Redex B in einem Term M, wenn A und B an verschiedenen Stellen in M stehen, und es gilt:
!!! Graphik!!!

Definition 2.2-17

Eine Reduktionsfolge $M \underset{\beta}{\overset{*}{\rightarrow}} N$ heißt *Normalfolge*, wenn bei jedem Reduktionsschritt das am weitesten links stehende Redex reduziert wird.

Eine Reduktionsfolge $M \underset{\beta}{\overset{*}{\rightarrow}} N$ heißt *Standardreduktionsfolge*, wenn das Reduzieren von links nach rechts erfolgt.

Im Unterschied zu einer Normalfolge können also bei einer Standardreduktionsfolge gewisse Redices übersprungen werden.

Praktisch geht man wie folgt vor: Nach der Reduktion eines Redex R werden alle Lambdas von Redices markiert, die links von R sind. Redices mit indiziertem Lambda dürfen nicht reduziert werden. Vorhandene Indizierungen ändern sich nicht durch Reduktionen.

Beispiel 2.2-13

1. $I(Ia) \equiv I((\lambda xx)a) \underset{\beta}{\rightarrow} Ia$

2. $I(Ia) \equiv (\lambda xx)(Ia) \underset{\beta}{\rightarrow} Ia$

   Beide Folgen sind Standardreduktionsfolgen, 2. ist die eindeutig bestimmte Normalfolge.

3. $I(Ia) \equiv I((\lambda xx)a) \underset{\beta}{\rightarrow} Ia \underset{\beta}{\rightarrow} a$

4. $I(Ia) \equiv (\lambda xx)(Ia) \underset{\beta}{\rightarrow} Ia \underset{\beta}{\rightarrow} a$

Wenn man auf Normalform reduziert, dann gibt es genau eine Standardreduktionsfolge, hier 4), die auch die Normalfolge ist. Die Folge 3. ist keine Standardreduktionsfolge.

Es gilt folgender berühmter Satz von Curry und Feys (Curry 1958):

Satz 2-2.15 (Standardisierungstheorem):

Zu jeder Reduktionsfolge $M \underset{\beta}{\overset{*}{\rightarrow}} N$ gibt es eine Standardreduktionsfolge.

Einen Beweis findet man in Curry (1958); einen etwas einfacheren in Barendregt (1981).

Für die praktische Anwendung ist folgende Variante von großer Bedeutung:

Satz 2.2-16

Sei N Normalform von M, dann existiert eine Normalfolge $M \underset{\beta}{\overset{*}{\rightarrow}} N$.

Für Programmiersprachen, die auf dem λ-Kalkül beruhen wie z.B. die LISP-ähnlichen Sprachen, wäre es also korrekt, Parameterübergabe mit "call by name" durchzuführen. Häufig wird jedoch eine effizientere Art der Parameterübergabe vorgesehen: "call by value". Dabei wird in einem Redex das Argument auf Normalform reduziert, ehe es im Rumpf der λ-Abstraktion substituiert wird. Falls die Reduktion des Arguments nicht terminiert, kann das betrachtete Redex nicht gemäß "call by value" reduziert werden. Leider führt diese Strategie nicht immer zu der Normalform eines λ-Terms, wie ii) des eingangs genannten Beispiels zeigt. In diesem Sinne ist "call by value" weniger mächtig als "call by name". Plotkin (1975) hat Programmiersprachen mit "call by value" bzw. mit "call by name" aus der Sicht des λ-Kalküls näher untersucht.

Die Reduktion von λ-Termen über Normalfolgen, d.h. gemäß "call by name" läßt sich durch eine Auswertungsfunktion $eval : \Lambda \rightarrow \Lambda$ definieren. Dazu

führen wir eine Hilfsfunktion $eval' : \Lambda \to \Lambda$ ein, durch die das am weitesten links stehende Redex eines λ-Terms reduziert wird, und ein Prädikat $\beta : \Lambda \to \{Bool\}$, mit dem festgestellt wird, ob ein Term ein Redex enthält.

$$
\begin{array}{lll}
\beta(a) & = false & \textbf{für } a \in C \\
\beta(x) & = false & \textbf{für } x \in V \\
\beta(\lambda x M) & = \beta(M) & \\
\beta((MN)) & = true & \text{für } M \equiv \lambda x L \\
\beta((MN)) & = \beta(M) \vee \beta(N) & \text{für } M \not\equiv \lambda x L \\
eval'[a] & = a & \textbf{für } a \in C \\
eval'[x] & = x & \textbf{für } x \in V \\
eval'[\lambda x M] & = \lambda x\ eval'[M] & \\
(*)\quad eval'[(MN)] & = \begin{cases} Sub^x_N[L] & \text{falls } M \equiv \lambda x L \\ (M\,\text{eval}'[N]) & \text{falls } M \not\equiv \lambda x L \text{ und } \beta(M) = false \\ (eval\prime[M]N) & \text{falls } M \not\equiv \lambda x L \text{ und } \beta(M) = true \end{cases} & \\
eval[M] & = \begin{cases} eval[\text{eval}'[M]] & \text{falls } \beta(M) = true \\ M & \text{sonst} \end{cases} &
\end{array}
$$

Man erhält eine Auswertungsfunktion $eval_y : \Lambda \to \Lambda$ gemäß "call by value", wenn man in der Definition von $eval'$ bei $(*)$ wie folgt ersetzt:

$eval'[(MN)] = Sub^x_{\text{eval}_n[N]}[L]$ falls $M \not\equiv \lambda x L$.

Das Thema "Reduktionsstrategie" ist eines der delikaten Kapitel im λ-Kalkül. Für die genannten, ganz einfachen Strategien hat man Implementationstechniken auf Rechenmaschinen, wodurch ihre besondere Bedeutung erklärbar ist. Bei der generellen Untersuchung von Reduktionsstrategien im λ-Kalkül hat man es mit sehr komplexen Problemen zu tun. Dazu gehört z.B. die Charakterisierung von "korrekten" Strategien bezüglich der Semantik in einem gegebenen Modell des λ-Kalküls. Eine Einführung in die Problematik, allerdings über die Theorie der rekursiven Programmschemata, findet man in Vuillemin (1974). Standardartikel sind Wadsworth (1976), Hyland (1975), Berry (1977).

Ein weiteres Problem sind "optimale" Strategien, die von Levy (1977) untersucht wurden. Man sucht Strategien, die eine minimale Anzahl von β-Reduktionen benötigen, wenn ein Term zu einer Normalform reduziert werden kann. Weder "call by name" noch "call by value" sind optimal.

2.2.8 Angewandter λ-Kalkül

Nach Church spricht man vom reinen λ-Kalktül, wenn die Menge der Konstanten leer ist ($\mathbf{C} = \emptyset$) und vom angewandten λ-Kalkül, falls $\mathbf{C} \neq \emptyset$ ist

(Definition 2-2.1). Da im reinen λ-Kalkül alle natürlichen Zahlen $\mathbb{N}_0$ und alle partiell rekursiven Funktionen $f : N_0^n \to \mathbb{N}$ dargestellt werden können, ist der angewandteλ-Kalkül für die Theorie von geringerem Interesse.

Für die praktische Anwendung bei Programmiersprachen ist es jedoch sinnvoll, einige Elemente aus **C** als Daten und einige als *Basisfunktionen* zu verwenden, so daß man z.B. folgendermaßen reduzieren kann:
$+ \;\; 2 \;\; 3 \to 5$ oder *cons* $A \;\; B \to (A.B)$

Solche Reduktionen mit Elementen aus **C** heißen δ-Reduktionen. Es handelt sich also nicht um eine spezielle Reduktion, sondern um eine Menge von Reduktionen. Es können jedoch nicht alle Terme, vom praktischen Standpunkt aus betrachtet, als sinnvoll angesehen werden, z.B. $+(3 = 5)7$.

Definition 2.2-18

Im angewandten λ-Kalkül ohne η-Reduktion sind δ-Reduktionen definiert als eine Menge von Regeln

1. $M \to N$, wobei für $M, N \in \Lambda$ gilt:
2. $M \equiv aM_1...M_n$ mit $a \in \mathbf{C}$ und $M_1, ..., M_n \in \Lambda$.
3. Alle Teilterme von M sind irreduzibel bezüglich β- und δ-Reduktionen.
4. $FV(M \;\; N) = \emptyset$.

Wenn M_i kein δ-Redex ist, dann ist M_i. in β-Normalform. δ-Reduktionen erfolgen stets nach allen β-Reduktionen. Die Church-Rosser-Eigenschaft gilt auch für den angewandten λ-Kalkül mit δ-Reduktionen. Wegen 2. der Definition müssen δ-Reduktionen mit "call by value" als Reduktionsstrategie angewandt werden.

Definition 2.2-19

Im angewandten λ-Kalkül ist ein Term M genau dann in Normalform, wenn M kein β- bzw. δ-Redex enthält.

Über den angewandten λ-Kalkül hat man einen bequemeren Zugang zur Definition einer formalen Semantik von Programmiersprachen als im reinen λ-Kalkül. Landin (1965) hat gezeigt, daß man die Semantik von ALGOL 60 über den angewandten λ-Kalkül formal definieren kann. LISP kann man unmittelbar als angewandten λ-Kalkül definieren (Perrot 1979, Greussay 1977,

Simon 1980).

Die Hinzunahme von Konstanten zum reinen λ-Kalkül hat gravierende Konsequenzen für Scott's Modelle des reinen λ-Kalküls. Die Einzelheiten entnehme man Stoy(1977), Plotkin (1975) und auch Gordon (1973).

Als weiterführende Lektüre über den klassischen λ-Kalkül, ist das Werk von Curry et.al. (1958) über den λ-Kalkül und kombinatorische Logik zu empfehlen. Das Buch von Stoy über denotationelle Semantik (Stoy 1977) enthält eine sehr leicht verständliche Einführung in Scott's Modelle des λ-Kalküls. Eine Einführung in das Thema "denotationelle Semantik" findet man auch in dem Buch von Gordon (1979). In Meyer (1982) werden Modelle für den hier vorgestellten ungetypten λ-Kalkül angegeben, die auf elementaren, algebraischen Konzepten beruhen. Logische Aspekte stehen in dem ebenfalls gut zu lesenden Büchlein von Hindley et.al. (1972) im Vordergrund. Der Tagungsband Böhm (1975) gibt eine Übersicht über die vielfältigen Aktivitäten zum Thema λ-Kalkül. Das Standardwerk über den λ-Kalkül von Barendregt (1981) ist leider, wie auch die Mehrzahl der Originalartikel, keine leichte Lektüre!

2.3 KOMBINATORISCHE LOGIK

2.3.1 Einleitung

Die kombinatorische Logik ist eine Theorie, die mit dem λ-Kalkül sehr eng verwandt ist. Sie wurde jedoch davon unabhängig schon in den zwanziger Jahren von Schönfinkel (1924) und Curry (1930) entwickelt. Curry beabsichtigte, über die kombinatorische Logik eine alternative Grundlage der Mathematik zu entwickeln. Bis heute ist jedoch noch kein befriedigendes, logisches System dieser Art bekannt. Curry's Paradoxon (Kap. 2-3.4) gibt einen Eindruck von den Problemen, auf die man stößt.

In der Informatik möchte man mit der kombinatorischen Logik im Prinzip dieselbe Art von Rechnungen wie mit dem λ-Kalkül durchführen können, nämlich das Reduzieren von Termen. In der kombinatorischen Logik benötigt man dazu jedoch nicht das Konzept der durch λ gebundenen Variablen, um das Substituieren im Term zu steuern. Es entfallen alle in Kapitel 2-2.2.1 genannten technischen Schwierigkeiten mit der Substitution. Insbesondere ist das systematische Umbenennen von Variablen nicht erforderlich. Weiterhin beruht das Rechnen auf ganz wenigen, überaus simplen Regeln. Diese Eigenschaften tragen wesentlich dazu bei, daß das maschinelle Reduzieren von kombinatorischen Termen in besonders einfacher Weise durchgeführt