2. Funktionen

2.1 Mathematischer Funktionsbegriff

Unter einer *Funktion* (im mathematischen Sinne) versteht man die eindeutige Zuordnung der Elemente einer Menge A zu den Elementen einer Menge B. Jedem Element von A darf höchstens ein Element von B zugeordnet sein, anderenfalls spricht man von einer *Relation*. Ist f eine solche Funktion, so schreibt man

$$f: A \rightarrow B.$$

$A \rightarrow B$ heißt *Funktionalität* oder, falls A und B Datentypen sind, *Typ* von f. A ist der *Quellbereich*, B der *Zielbereich* von f.

f ist *partiell*, wenn es Elemente $a \in A$ gibt, für die f(a) nicht definiert ist, anderenfalls *total*.

$$D(f)=dom(f)=\{a \in A \mid f(a) \text{ ist definiert}\} \subseteq A$$

und

$$B(f)=im(f)=\{b \in B \mid \exists a \in A: f(a)=b\} \subseteq B$$

heißen *Definitions-* bzw. *Bildbereich* von f.

In der Informatik vermeidet man die Verwendung partieller Funktionen meist, um eine einheitlichere Darstellung zu erhalten. Hierzu erweitert man Quell- und Zielbereich um das sog. *bottom-Element* $\perp$ und definiert

$$A^{\perp}:=A \cup \{\perp\},\ B^{\perp}:=B \cup \{\perp\}.$$

$A^{\perp}$ und $B^{\perp}$ heißen *Vervollständigungen* von A bzw. B. f erweitert man nun zu

$$f^{\perp}: A^{\perp} \rightarrow B^{\perp} \text{ mit}$$

$$f^{\perp}(a)= \begin{cases} f(a), & \text{falls } f(a) \text{ definiert,} \\ \perp, & \text{sonst.} \end{cases}$$

Bei berechenbaren Funktionen symbolisiert das Zeichen $\perp$ die Pseudoausgabe eines Algorithmus, der für die gewählte Eingabe nicht terminiert.

Wahlfreiheiten besitzt man, wenn A das kartesische Produkt von Mengen ist:

$$A=A_1 \times A_2 \times ... \times A_n.$$

Erweitert man nun die A_i um das bottom-Element, so ist die Frage, welchen Wert ein Ausdruck

$$f(a_1,...,a_n)$$

besitzt, in dem einige $a_i=\perp$ und einige $a_i \neq \perp$ sind. Meist entscheidet man sich hier für eine *strikte* Fortsetzung von f zu $f^{\perp}$, d.h. man setzt

$$f^{\perp}: A_1 \times A_2 \times ... \times A_n \rightarrow B \text{ mit}$$

$$f^{\perp}(a_1,...,a_n)= \begin{cases} f(a_1,...,a_n), & \text{falls } a_i \neq \perp \text{ für } 1 \leq i \leq n, \\ \end{cases}$$

$\perp$, falls $a_i=\perp$ für ein $i\in\{1,...,n\}$.

$f^{\perp}$ ist also undefiniert, wenn auch nur eines seiner Argumente undefiniert ist. Wir werden später Abschwächungen dieser Striktheit kennenlernen und Funktionen auch dann noch sinnvoll auswerten können, wenn einige ihrer Argumente undefiniert, also gleich $\perp$ sind. Für den Rest der Vorlesung (wenn nicht anders erwähnt) seien alle Wertemengen IN, IR, Z, ... stets vervollständigt, ohne daß wir dies durch das hochgestellte Zeichen $^{\perp}$ besonders kenntlich machen.

2.2 Funktionen höherer Ordnung

Handelt es sich bei den Elementen der Mengen A oder B einer Funktion $f: A\to B$ selbst wieder um Funktionen, so bildet f offenbar Funktionen aus A auf Funktionen aus B ab. f ist dann eine Funktion höherer Ordnung, ein sog. *Funktional*.

Beispiele:

1) Seien

$A=\{f: IR\to IR \mid f \text{ ist integrierbar}\}$,

$B=\{f: IR\to IR \mid f \text{ ist Funktion}\}$.

Dann ist das Integral $\int$ ein Funktional

$\int: A\to B$,

das jeder integrierbaren Funktion f eine reellwertige Funktion F, die Stammfunktion, zuordnet:

$\int(f)=F$,

und es gilt:

$$\int_0^x f(t)\,dt=F(x).$$

2) Das Kronecker-Symbol δ_α, $\alpha\in$ IR, aus der linearen Algebra ist ein Funktional

$\delta_\alpha: \{f: IR\to IR \mid f \text{ ist Funktion}\}\to IR$ mit

$\delta_\alpha(f)=f(\alpha)$.

3) Wir definieren das Funktional

$\text{twice}: \{f: IR\to IR \mid f \text{ ist Funktion}\}\to\{f: IR\to IR \mid f \text{ ist Funktion}\}$ mit

$\text{twice}(f)=f\circ f$.

In diesem Sinne kann man Funktionen bezgl. ihrer Ordnung klassifizieren.

Definition A:

- Daten sind nullstellige Funktionen, sie besitzen die Ordnung 0 und heißen **Konstanten**.
- Die **Ordnung** einer Funktion ist das Maximum der Ordnungen ihrer Argumente zuzüglich 1.

Funktionen der Ordnung ≥2 heißen auch **Funktionale**.

Beispiele:

1) Die Konstante 17 besitzt die Ordnung 0. Man kann sie als nullstellige Funktion
   17: $\rightarrow$IN
   interpretieren mit
   17()=17.
2) Die Funktionen aus obigem Beispiel besitzen die Ordnung 2.

2.3 Currying

Eine Funktion kann höchstens ein Argument besitzen. Bei Funktionen, die mehrere Argumente benötigen, muß man den Umweg über das kartesische Produkt der Wertemengen des Quellbereichs gehen und der Funktion die Argumente als ein Argument in Form eines Tupels zuführen, z.B. für die Multiplikation

mult: $Z\times Z\rightarrow Z$ mit

mult(x,y)=xy (eigentlich: mult **((**x,y**))**).

Diese Vorgehensweise schränkt in der Praxis den Umgang mit Funktionen höherer Ordnung beträchtlich ein. Viele Funktionen besitzen nämlich auch dann eine "vernünftige" Bedeutung, wenn man sie *partiell auswertet*, d.h. nur auf eine Auswahl ihrer Tupelelemente anwendet. So kann man z.B. bei der Funktion mult das erste Argument festhalten und

mult(2,$\cdot$): $Z\rightarrow Z$

als Verdoppelungsfunktion

d: $Z\rightarrow Z$

auffassen, die dann nach Bedarf auf ein Argument x angesetzt werden kann:

d(x)=mult(2,x)=2x.

Diesen Übergang von einer Funktion

f: $A\times B\rightarrow C$,

der man zur Auswertung ein vollständiges Tupel $(a,b)\in A\times B$ zuführen muß, zu einer Funktion 2. Ordnung

F: $A\rightarrow(B\rightarrow C)$,

der man zunächst a zuführen kann und eine Funktion 1. Ordnung

F(a): $B\rightarrow C$

erhält, die dann auf b angewendet das Ergebnis

$$F(a)(b)=f(a,b)$$

liefert, bezeichnet man als *currying* (nach dem engl. Mathematiker H.B. Curry 1958, ursprüngl. erfunden von dem dt. Mathematiker M. Schönfinkel 1924). Die Umkehrung dieses Prozesses nennt man *uncurrying*.

Ist f eine n-stellige Funktion

$$f: A_1\times...\times A_n\rightarrow B,$$

so wendet man die curry-Operation (n-1)-mal an und erhält ein Funktional n-ter Ordnung

$$F: A_1\rightarrow(A_2\rightarrow(A_3\rightarrow...\rightarrow(A_n\rightarrow B)...))$$

mit

$$F(a_1)(a_2)(a_3)...(a_n)=f(a_1,a_2,a_3,...,a_n).$$

Currying stellt also eine ein-eindeutige Beziehung zwischen n-stelligen Funktionen erster Ordnung und 1-stelligen Funktionalen n-ter Ordnung her.

Definition und Satz A:

Zu jeder Funktion

$$f: A_1\times...\times A_n\rightarrow B$$

gibt es genau eine Funktion

$$F: A_1\rightarrow(A_2\rightarrow(A_3\rightarrow...\rightarrow(A_n\rightarrow B)...))$$

mit

$$F(a_1)(a_2)(a_3)...(a_n)=f(a_1,a_2,a_3,...,a_n).$$

Der Übergang von f zu F heißt **currying**, die Umkehrung **uncurrying**.

2.4 Informatischer Funktionsbegriff

Innerhalb der Informatik interessiert man sich vorwiegend für berechenbare Funktionen, also Funktionen f, für die es eine (Turing-)Maschine gibt, die bei Eingabe von x den Funktionswert f(x) ausgibt. Für die Konstruktion solch einer Maschine ist eine deskriptive Definition von f wenig hilfreich, vielmehr benötigt man eine algorithmische Beschreibung des Weges, auf dem man zu jedem Argument von f in endlicher Zeit den Funktionswert ermitteln kann.

Beispiel: ggT zweier natürlicher Zahlen. Eine übliche mathematische Definition, die sich nicht unmittelbar algorithmisch umsetzen läßt, ist:

$$ggT: IN_0\times IN_0\rightarrow IN_0 \text{ mit}$$

$$ggT(a,b)=max\{t\in IN_0 \mid t\backslash a \text{ und } t\backslash b\}.$$

Brauchbarer aus informatischer Sicht ist eine Definition, die ein konkretes algorithmisches Verfahren vorgibt, um zu a und b den ggT zu berechnen:

ggT(a,b)=if a=b then a else

if a<b then ggT(b,a) else ggT(a-b,b).

Definitionen dieser Art nennt man *Rechenvorschriften*.

Definition A:

Eine **Rechenvorschrift** ist die funktionale Beschreibung eines Algorithmus. Die allgemeine Form einer Rechenvorschrift lautet:

function f x_1:D_1 x_2:D_2 ... x_n:D_n → D ≡ R.

Kopf der Rechenvorschrift **Rumpf** der Rechenvorschrift

Hierbei sind f, x_1,...,x_n beliebige Bezeichner. f ist der **Name** der Rechenvorschrift, x_1,...,x_n sind die **formalen Parameter** und haben die Datentypen D_1,...,D_n. D ist der Datentyp des Ergebnisses. R ist ein Ausdruck, der unter beliebiger Verwendung

- von elementaren Funktionen auf den Grundtypen int, nat, real, bool, char,
- von bereits definierten Rechenvorschriften (Prinzip der *Einsetzung*), auch von f selbst (Prinzip der *Rekursion*),
- von Alternativen (if <Bedingung> then <Ausdruck> else <Ausdruck> fi).

gebildet worden ist.

Man beachte, daß f in obiger Definition als vollständig gecurried betrachtet wird. Für einen Wert a vom Typ D_1 ist dann also der Ausdruck (f a) eine Rechenvorschrift mit n-1 Parametern x_2:D_2 ... x_n:D_n → D.

Beispiele:

1) Die obige Rechenvorschrift zur Berechnung des ggT lautet nun exakt im Formalismus der Definition A:

   function ggT (a,b):nat×nat → nat ≡

   if a=b then a else

   if a<b then ggT(b,a) else ggT(a-b,b) fi fi.

2) Die bereits bekannte Funktion

   function misch f:intlist g:intlist → intlist ≡ ...

   mischt zwei Folgen ganzer Zahlen.

2.5 Von Rechenvorschriften zu Werten: Applikation

Die wohl wichtigste Operation im Zusammenhang mit einer allgemeinen Rechenvorschrift

function f x_1:D_1 x_2:D_2 ... x_n:D_n → D ≡ R.

ist die **Anwendung** (**Applikation**) von f auf einen Satz geeigneter Objekte (**Argumente**, **aktuelle Parameter**), in Zeichen:

$f\ a_1\ a_2\ ...\ a_n$.

Bei der Anwendung ist eine weitere wichtige Operation beteiligt, die **Substitution**. Sie macht aus der mit f bezeichneten Rechenvorschrift einen auswertbaren Ausdruck, indem sie die formalen Parameter durch die aktuellen Parameter ersetzt.

Beispiel: Sei ggT wie in 2.2.4 definiert. Durch Applikation von ggT auf das Argument (24,15) und Substitution von a durch 24 und b durch 15 geht die Rechenvorschrift in den auswertbaren Ausdruck

if 24=15 then 24 else
    if 24<15 then ggT(15,24) else ggT(9,15) fi fi

über. Diesen kann man zunächst zu ggT(9,15) und dann weiter auswerten.

2.5.1 Substitutionsregeln

Wenn eine Funktion (der Einfachheit halber eine einstellige)

function f x:D → D' ≡ R.

auf ein Argument E (z.B. einen Ausdruck) angewendet wird, so muß der formale Parameter x innerhalb des Rumpfes R der Rechenvorschrift durch das Argument substitutiert werden. Hierfür gibt es mehrere Ersetzungsstrategien, die sich darin unterscheiden, wann (vor oder nach der Substitution) und wie oft (an jeder Stelle oder einmalig) das Argument E ausgewertet wird:

1) **Call-by-value-Strategie**: Um f(E) zu berechnen, werte zunächst E aus. Ersetze x überall im Rumpf R durch den Wert von E und werte den Ergebnisausdruck aus.
   Beispiel:
   Sei
   function d x:int → int ≡ x+x.
   Dann wird der Ausdruck
   d(d(d(3)))
   wie folgt ausgewertet:
   d(d(d(3))) ⇒ d(d(3+3)) ⇒ d(d(6)) ⇒ d(6+6) ⇒ d(12) ⇒ 12+12 ⇒ 24.
   Die call-by-value-Strategie nennt man auch *strikte* Auswertungsstrategie, weil sie die Striktheit von Funktionen korrekt widerspiegelt (Zur Erinnerung: g: $A_1 \times ... \times A_n \to B$ heißt strikt, falls $g(a_1,...,a_n)=\perp$, sobald ein $a_i=\perp$ ist).
   Trotz ihrer mathematischen Sauberkeit besitzt die call-by-value-Strategie in der Praxis eine Reihe von Nachteilen:
   - Sie führt häufig zu ineffizienten Berechnungen.

Beispiel:

Man betrachte die konstante Funktion

function null x:int → int ≡ 0.

Dann wird

null(d(d(d(3))))

ausgewertet zu

null(d(d(d(3)))) ⇒ null(d(d(3+3))) ⇒ null(d(d(6))) ⇒ ...
⇒ null(12+12) ⇒ null(24) ⇒ 0.

Hier muß also das Argument von null zunächst überflüssigerweise ausgewertet werden, obwohl der Funktionswert 0 bereits apriori feststeht.

- Man betrachte die Funktion (den bedingten Ausdruck)

  if(b,e,e') (besser bekannt als: if b then e else e' fi).

  Verfolgt eine Programmiersprache konsequent die call-by-value-Strategie, so kann man den bedingten Ausdruck nicht mehr in der gewünschten Form darstellen. Dazu betrachte man die Rechenvorschrift

  function fak x:nat → nat ≡ if(x=0,1,x·fak(x-1)).

  Dann führt die Anwendung zu einer nicht terminierenden Berechnung:

  fak 0 ⇒ if(0=0,1,0·fak(-1)) ⇒ if(true,1,0· f(-1=0,1,-1·fak(-2))) ⇒ ...

  Folglich ist hier in jedem Falle eine Änderung der Strategie erforderlich: Bei einem bedingten Ausdruck darf nur entweder der then-Zweig oder der else-Zweig ausgewertet werden, aber nicht beide.

2) **call-by-name-Strategie**: Um f(E) zu berechnen, ersetze x überall im Rumpf R textuell durch E und werte den Ergebnisausdruck aus.
   Hier wird der aktuelle Parameter also erst ausgewertet, wenn er benötigt wird.
   Beispiele:
   1) Mittels call-by-name wird das Ergebnis von null(d(d(d(3)))) in einem Schritt korrekt berechnet.
   2) Andererseits wird der Ausdruck

      d(d(d(3)))

      wie folgt ausgewertet:

      d(d(d(3))) ⇒ d(d(3))+d(d(3)) ⇒ d(3)+d(3)+d(d(3)) ⇒ 3+3+d(3)+d(d(3)) ⇒
      6+d(3)+d(d(3)) ⇒ ...

   Der Nachteil dieser Strategie ist die ineffiziente mehrfache Auswertung desselben Ausdrucks. Offenbar nutzt diese Strategie die Eigenschaft der referenziellen Transparenz nicht aus, nach der ein Ausdruck stets durch einen anderen gleichen Werts ersetzt werden kann. Daher kann z.B. in Beispiel 2 jedes Vorkommen von d(3)

sofort durch seinen Wert 6 ersetzt werden, sobald d(3) zum ersten Mal berechnet worden ist. Auf dieser Idee basiert die dritte Strategie.

3) **call-by-need-Strategie** (**lazy evaluation**): Um f(E) zu berechnen, ersetze x überall in R textuell durch E und werte den Ergebnisausdruck aus. Sobald E hierbei zum ersten Male ausgewertet wird, ersetze E in R überall durch den berechneten Wert.

   Beispiele:

   1) Der Ausdruck null(d(d(d(3)))) wird wie gewünscht unmittelbar zu 0 ausgewertet.
   2) Wir berechnen d(d(d(3))); die eckigen Klammern begrenzen die Rümpfe von d:

      d(d(d(3))) ⇒ [d(d(3))+d(d(3))] ⇒ [[d(3)+d(3)]+d(d(3))] ⇒

      [[[3+3]+d(3)]+d(d(3))] ⇒ Hier liegt d(3)=6 vor und wird im zugehörigen Rumpf ersetzt

      [[6+6]+d(d(3))] ⇒ Hier liegt d(d(3))=12 vor und wird im zugehörigen Rumpf ersetzt

      [12+12] ⇒ 24.
   3) Der bedingte Ausdruck if(b,e,e') wird korrekt ausgewertet.

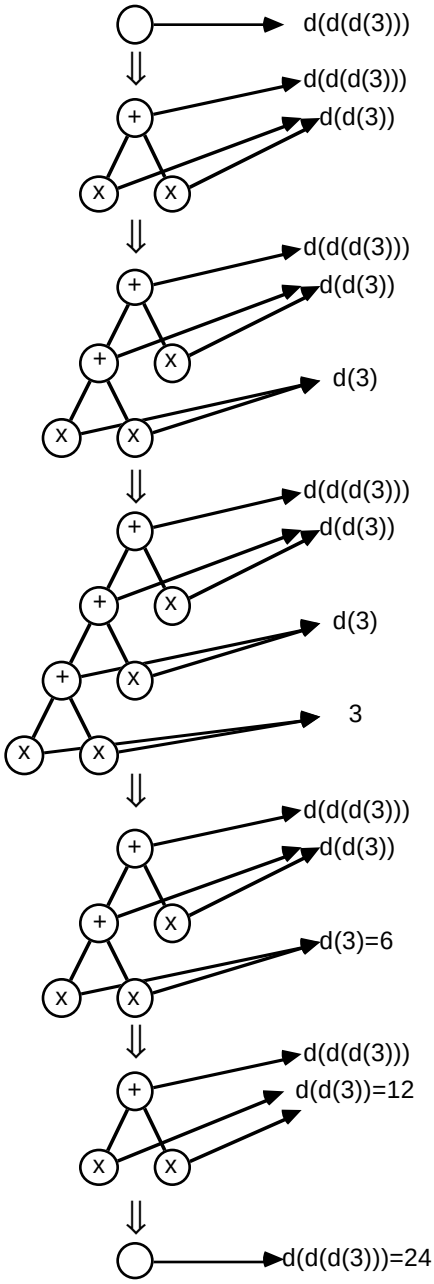

Abb. 1: Auswertung eines Ausdrucks mittels call-by-need

In der Praxis kann man die call-by-need-Strategie durch Verweise realisieren. Jedes Vorkommen eines formalen Parameters wird durch einen Verweis auf das zugehörige Argument ersetzt. Wird nun das Argument zum ersten Mal ausgewertet, so ist dieser Wert über den Verweis von jedem Vorkommen des formalen Parameters zugänglich. *Beispiel:* Abb. 1 zeigt die Auswertung von d(d(d(3))) unter Verwendung dieser Technik.

Welche Substitutionsregel ist nun die beste? Offenbar führt call-by-need häufig zu effizienten Auswertungen von Ausdrücken, andererseits benötigt man hierzu eine umfangreiche Verwaltung der Vorkommen identischer Ausdrücke. Ein Nachteil gegenüber call-by-value besteht in der Aufweichung der mathematischen Betrachtung von Funktionen: So führt zwar die Auswertung von

null(d(d(d(3))))

unmittelbar zum korrekten Wert 0, weil das Argument nicht ausgewertet zu werden braucht. null(E) liefert aber auch den Wert 0, wenn der Ausdruck E undefiniert ist, also z.B. nicht terminiert. Dies widerspricht der mathematischen Tradition, wonach ein Ausdruck nur dann eine Bedeutung besitzt, wenn auch alle Teilausdrücke eine Bedeutung haben.

Einen besonderen Vorteil der lazy-evaluation werden wir im folgenden noch skizzieren: Er besteht in der Möglichkeit, prinzipiell unendliche Objekte (**lazy lists**, **streams**) zu definieren. Dies eröffnet neuartige und sehr elegante Programmiermöglichkeiten. Betrachten wir hierzu folgende

Beispiele:

1) Man betrachte die Funktion

function from n:int→int≡[n]•from(n+1).

Offenbar beschreibt (from n) die unendliche Folge

[n, n+1, n+2, ...].

Natürlich kann man auf dem Rechner nicht effektiv mit unendlichen Listen arbeiten, aber man kann sie wie oben endlich beschreiben, und man kann auf endliche Teilfolgen zugreifen und sie manipulieren, z.B. durch die Funktion

function take k:int l:intlist→intlist≡
if k=0 then [ ] else
    if l=[ ] then [ ] else [first l]•(take (k-1) (rest l)),

die die ersten k Elemente einer Liste l liefert. Welches Ergebnis liefert nun

take 3 (from 17) ?

Setzt man als Substitutionsregel call-by-value voraus, so ist das Ergebnis offenbar undefiniert, da der aktuelle Parameter from 17 zunächst ausgewertet werden muß und diese Berechnung nicht terminiert. Anders bei lazy evaluation: Hier wird der Ausdruck from 17 nur insoweit ausgewertet, als er für die Berechnung des Gesamtausdrucks benötigt wird. Es ergibt sich folgende Auswertung:

take 3 (from 17) => [first (from 17)]•(take 2 rest(from 17)) =>
    [first ([17]•(from 18))]•(take 2 rest([17]•(from 18))) =>

[17]•(take 2 rest([17]•(from 18))) =>
[17]•[first (from 18)]•(take 1 rest(from 18)) =>
[17]•[first ([18]•(from 19))]•(take 1 rest([18]•(from 19))) =>
[17]•[18]•(take 1 (from 19)) =>
[17,18]•(take 1 (from 19)) =>
[17,18]•[first (from 19)]•(take 0 rest(from 19)) =>
[17,18]•[first ([19]•(from 20))]•(take 0 rest([19]•(from 20))) =>
[17,18]•[19]•(take 0 rest([19]•(from 20))) =>
[17,18,19]•[ ] => [17,18,19].

2) Wir definieren die Folge aller geraden natürlichen Zahlen. Offenbar erhält man diese durch Verdoppelung aller Elemente in der Liste (from 1) oder besser durch elementweise Addition beider Listen:

```
function addlists a:intlist b:intlist→intlist≡
if a=[ ] then b else
    if b=[ ] then a else
        [(first a)+(first b)]•(addlists (rest a) (rest b)).
```

Wir beobachten nun, daß die nullstellige Funktion

```
function nat →intlist≡from 1.
```

die natürlichen Zahlen beschreibt. Dann repräsentiert die nullstellige Funktion

```
function evennat →intlist≡addlists nat nat.
```

die Folge aller geraden natürlichen Zahlen.

Aufgabe:
Man schreibe eine Funktion fib, die die unendliche Folge aller Fibonacci-Zahlen beschreibt und werte den Ausdruck

```
take 4 fib
```

aus.

Die Programmiersprache ML, die wir später behandeln, verwendet die call-by-value-Strategie. Durch einen Trick kann man jedoch auch hier dafür sorgen, daß Ausdrücke lazy ausgewertet werden und so ebenfalls unendliche Objekte beschreiben können.

2.6 Von Ausdrücken zu Rechenvorschriften: Abstraktion

Zur Motivation betrachte man den Ausdruck

$$\pi r^2 h.$$

Die Bedeutung dieses Ausdrucks kann ein bestimmter Wert sein, wenn π, r und h bereits konkrete Werte besitzen. Es kann sich aber auch um eine Funktion in den Variablen π, r

und h handeln. Aus unserer Kenntnis der elementaren Mathematik wissen wir jedoch, daß es sich bei π um eine Konstante handelt, deren Wert nicht frei wählbar ist. Wofür stehen r und h? Mit h kürzt man in der Physik das Planck'sche Wirkungsquantum (ebenfalls eine Konstante) ab, andererseits kann es sich bei dem Ausdruck auch um die Formel für das Volumen eines Zylinders handeln. In diesem Fall ist h keine Konstante, sondern (wie r) eine Variable für die Höhe des Zylinders.
Ein Ausdruck läßt also i.a. einen großen Freiraum zur funktionalen Interpretation zu. Man kann $\pi r^2 h$ als Funktion

f: $IN^3 \rightarrow IN$ mit
$f(\pi,r,h)=\pi r^2 h$ betrachten, oder auch als
g: $Z \rightarrow IR$ mit
$g(r)=\pi r^2 h$ betrachten, oder auch als
h: $IR \rightarrow IR$ mit
$h(x)=\pi r^2 h$ (h ist jetzt eine konstante Funktion),

oder mit der intendierten Bedeutung

V: $IR^2 \rightarrow IR$ mit
$V(r,h)=\pi r^2 h$.

In der Mathematik lassen sich diese Mißverständnisse meist aus dem Zusammenhang ausräumen. In der Informatik müssen wir die Interpretationsfreiheiten, um keine Probleme bei der Implementierung zu bekommen, explizit beseitigen und genau festlegen, welche Funktionalität ein Ausdruck besitzen soll, was also seine Parameter und deren Datentyp sein sollen und welchen Datentyp das Ergebnis haben soll.
Diesen Übergang von einem Ausdruck zu einer Rechenvorschrift bezeichnet man als **Abstraktion**. Aus programmiersprachlicher Sicht handelt es sich bei der Abstraktion um die Zusammenfassung eines (Teil-)Programms oder (Teil-)Ausdrucks zu einer Prozedur oder Funktion nebst geeigneter Parametrisierung.

Beispiel:
Den Ausdruck für das Zylindervolumen abstrahieren wir zu:

function V (r,h): real×real → real ≡ $\pi r^2 h$.

Weiter abstrahieren wir in V den Teilausdruck πr^2 (=Kreisfläche) zu

function F r: real→real ≡ πr^2.

wodurch V vereinfacht werden kann zu

function V (r,h): real×real → real ≡ (F r)h.

Man kann zeigen, daß die drei Operationen Abstraktion, Applikation und Substitution die grundlegenden für die funktionale Programmierung sind in dem Sinne, daß sie ausreichen, um alle berechenbaren Funktionen zu beschreiben.