10 Funktionale Programmierung

Am Schluß des Kapitels 9 hatten wir den mächtigsten Datentypkonstruktor behandelt, die *Funktionenraumbildung*. Mit diesem Konstruktor können wir einerseits alle Datentypen auf Funktionen zurückspielen, andererseits überschreiten wir damit die Grenze zwischen der Beschreibung der Objekte, auf denen Operationen wirken, und den Operationen selbst. Zugleich verwischt sich die Trennung zwischen der Objektbeschreibung und der Handlungsvorschrift eines Algorithmus. Diese einheitliche Zugangsmöglichkeit, die beide Aspekte der Algorithmenkonstruktion unter einen Ansatz subsumiert, besteht aber nur in den Programmiersprachen, in denen die Funktionenraumbildung uneingeschränkt möglich ist. Zur Zeit kommen nur die funktionalen Programmiersprachen diesem Anspruch nahe. Wir werden uns daher in diesem Kapitel einerseits mit den zahlreichen Phänomenen, die mit der Definition und Benutzung von Funktionen verbunden sind, befassen und damit zugleich in die funktionale Programmierung einführen.
Erstes Phänomen ist die unbeschränkte Erweiterung des Funktionenkonzepts auf Funktionen höherer Ordnung.

10.1 Funktionen höherer Ordnung

Einen Funktionenraum über zwei Datentypen D' und D'' definiert man gem. Abschnitt 9.4.7 allgemein durch

$$\underline{\text{typ}}\ D \equiv [D' \rightarrow D''].$$

Handelt es sich bei den Datentypen D' und D'', für die keinerlei Beschränkungen bestehen, selbst wieder um Funktionenräume, so bilden die Funktionen $f \in D$ offenbar Funktionen aus D' auf Funktionen aus D'' ab. f ist dann eine Funktion *höherer Ordnung* (eine *higher order function*, Abk. *HOF*), ein sog. *Funktional*. Solche Funktionale kennt man seit langem in der Mathematik.
Beispiele:

1) Seien

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

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

   Dann ist das Integral $\int$ ein Funktional

   $\int: A \rightarrow B$,

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

   $\int(f)=F$,

   und es gilt die Beziehung:

   x

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

Umgekehrt ist auch der Ableitungsoperator $'$ ein Funktional

$$': \{f: IR \to IR \mid f \text{ ist differenzierbar}\} \to \{f: IR \to IR \mid f \text{ ist Funktion}\},$$

das jeder differenzierbaren Funktion ihre Ableitungsfunktion zuordnet:

$$'(f) = f'.$$

$\int$ und $'$ sind aus Informatiksicht Funktionale vom Typ $[[real \to real] \to [real \to real]]$.

2) Man betrachte für $a,b \in Z$ und $f: Z \to Z$ die Summation

$$\sum_{i=a}^{b} f(i) = f(a)+f(a+1)+f(a+2)+...+f(b).$$

Die Summe läßt sich als eine Funktion S mit einer Funktion $f: Z \to Z$ und den Grenzen $a,b \in Z$ als Argumenten interpretieren, d.h., man definiert

$S: \{f: Z \to Z\} \times Z \times Z \to Z$ mit

$$S(f,a,b) = \sum_{i=a}^{b} f(i).$$

S ist dann also ein informatisches Objekt vom Datentyp D definiert durch

$\underline{typ}\ D \equiv [([int \to int], int, int) \to int]$.

3) Wir definieren das Funktional

$twice: [int \to int] \to [int \to int]$ mit

$twice(f) = f \circ f$.

twice beschreibt die Selbstanwendung einer Funktion f. So gilt für die Standardfunktion auf skalaren Datentypen pred

$twice(pred) = pred \circ pred$ und

$twice(pred)\ (7) = pred(pred(7)) = 5$.

Allgemein beschreibt man die Komposition zweier beliebiger Funktionen durch

$komp: [A \to B] \times [C \to A] \to [C \to B]$ mit

$komp(f,g) = f \circ g$.

komp besitzt den Typ $[([A \to B],[C \to A]) \to [C \to B]]$.

Setzt man diesen Übergang von Funktionen zu Funktionalen fort, so 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 Funktionen aus dem vorigen Beispiel besitzen alle die Ordnung 2.
2) Die Konstante 17 besitzt die Ordnung 0. Man kann sie interpretieren als nullstellige Funktion

   17: unit→nat mit

   17()=17.

10.2 Currying

Eine Funktion kann, genau genommen, immer nur 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 en bloc als *ein* Argument in Form eines Tupels zuführen, z.B. für die Multiplikation

mult: IR×IR→IR mit

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

Dieses Erfordernis schränkt in der Praxis den Umgang mit Funktionen 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 würde man z.B. gerne bei der Funktion mult das erste Argument festhalten und

d=mult(2,·): IR→IR

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

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

Oder man betrachtet x und y als Währungen und definiert eine Umrechnungsfunktion von Dollar in DM durch

DollarDM: IR→IR mit

DollarDM=mult(1.521,·).

Beide Definitionen von d und DollarDM sind in der angegebenen Form nicht möglich, da man nicht einzelne Tupelelemente weglassen darf. Diesen Mangel können wir durch Änderung der Funktionsdefinition beseitigen, indem wir von einer allgemeinen (zweistelligen) Funktion

f: A×B→C,

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

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

übergehen, die man partiell auswerten kann, der man also 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. Diesen Übergang 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 heißt *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 so eine ein-eindeutige Beziehung zwischen n-stelligen Funktionen erster Ordnung und 1-stelligen Funktionalen n-ter Ordnung her. Da man in der gecurryten Darstellung F nicht mehr zwischen Argumenten und Funktionen unterscheiden kann – F ist eine Funktion, $F(a_1)$ ist eine Funktion, $F(a_1)(a_2)$ ist eine Funktion usw. –, setzt man die Klammern bei der gecurryten Version linksassoziativ, wie in

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

oder läßt sie ganz weg und schreibt einfach

$$F\ a_1\ a_2\ a_3\ ...\ a_n.$$

Man beachte aber, daß die partielle Auswertung nur von links nach rechts erfolgen kann.

Definition und Satz B:

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**.

Im folgenden werden wir Funktionen fast immer in gecurryter Version verwenden.

10.3 Informatischer Funktionsbegriff

Innerhalb der Informatik interessiert man sich vorwiegend für *berechenbare Funktionen*, also Funktionen f, für die es einen Algorithmus und nach der Churchschen These (s. Kapitel 3) auch eine Maschine gibt, die bei Eingabe von x den Funktionswert f(x) ausgibt. Man schränkt daher in der Praxis die Funktionenraumbildung [D'→D''] ebenfalls auf die berechenbaren Funktionen von D' nach D'' ein.

Vor der Konstruktion eines Algorithmus zu einer Funktion f entwirft man eine funktionale Spezifikation, die nur die Leistungen des gesuchten Algorithmus beschreibt, aber nicht wie diese Leistungen erbracht werden. Diese Spezifikationen sind daher wenig hilfreich, um daraus einen Algorithmus abzuleiten. Vielmehr sucht man dafür eine algorithmische Beschreibung des Weges, auf dem man zu jedem Argument von f in endlicher Zeit den Funktionswert ermitteln kann.

Beispiel: Eine übliche mathematische Definition für den größten gemeinsamen Teiler ggT zweier natürlicher Zahlen ist:

$ggT: IN_0 \times IN_0 \rightarrow IN_0$ mit

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

Diese Definition ist aus Sicht eines Programmierers wenig hilfreich, da sie sich nicht unmittelbar algorithmisch umsetzen läßt. Brauchbarer ist eine Definition, die ein konkretes algorithmisches Verfahren vorgibt, um zu a und b den ggT effektiv zu berechnen, z.B.:

$$ggT(a,b)= \begin{cases} a, & \text{falls } a=b, \\ ggT(b,a), & \text{falls } a<b, \\ ggT(a-b,b), & \text{sonst.} \end{cases}$$

Oder in mehr programmiersprachlicher Notation:

```
funktion ggT(a,b: nat) nat;
wenn a=b dann
    ergebnis a
sonst
    wenn a<b dann
        ergebnis ggT(b,a)
    sonst
        ergebnis ggT(a-b,b)
    ende
ende.
```

Denn mit den letzten beiden Definitionen kann man z.B. den ggT(40,12) schrittweise berechnen:

ggT(40,12)=ggT(28,12)=ggT(16,12)=ggT(4,12)=ggT(12,4)
=ggT(8,4)=ggT(4,4)=4.

Funktionale Definitionen dieser "ausführbaren" Art nennt man *Rechenvorschriften*. Dabei ist zunächst aber noch unklar, ob diese Definition des ggT überhaupt korrekt und für jede mögliche Parameterkombination sinnvoll ist und zum Ziel führt, denn offenbar ist die Definition selbstbezüglich, da die Beschreibung von ggT auf sich selbst zurückgeführt wird. Dieses Phänomen bezeichnet man als *Rekursion*; wir kommen im weiteren Verlauf dieses Kapitels darauf zurück.

Definition C:
Die funktionale Beschreibung eines Algorithmus nennt man **Rechenvorschrift**.
Ihre allgemeine Definition lautet:

$$\underline{\text{funktion}}\ \ f\ \ x_1{:}D_1\ x_2{:}D_2\ \ldots\ x_n{:}D_n \rightarrow D \equiv R.$$

Hierbei ist f der Bezeichner der Rechenvorschrift, $x_1,\ldots,x_n$ sind die paarweise verschiedenen **formalen Parameter**, die die Datentypen $D_1,\ldots,D_n$ besitzen. D ist der Datentyp des Funktionsergebnisses. f besitzt die **Funktionalität** $D_1\rightarrow(D_2\rightarrow(D_3\rightarrow\ldots(D_n\rightarrow D)\ldots))$.
Die Zeichen links vom Symbol $\equiv$ bezeichnet man als **Kopf** der Rechenvorschrift, die Zeichen rechts von $\equiv$ als **Rumpf**. R ist ein **Ausdruck** (**Term**) vom Typ D, der induktiv wie folgt definiert ist:
Elementare Bausteine:
(1) Ist E ein elementarer Datentyp und $x\in E$, so ist x ein Ausdruck vom Typ E.
(2) Für i=1,...,n ist x_i ein Ausdruck vom Typ D_i.
Konstruktoren:
(3) Ist g eine Rechenvorschrift der Funktionalität $E'\rightarrow E''$ und A ein Ausdruck vom Typ E', so ist $g(A)$ ein Ausdruck vom Typ E'' (Konstruktor "**Einsetzung**" oder – für $g=f$ – Konstruktor "**Rekursion**").
(4) Sind $A1$ und $A2$ beliebige Ausdrücke vom Typ E und B ein Ausdruck vom Typ $bool$, so ist auch

$$\underline{\text{wenn}}\ B\ \underline{\text{dann}}\ A1\ \underline{\text{sonst}}\ A2\ \underline{\text{ende}}$$

ein Ausdruck (Konstruktor "**Alternative**", **bedingter Ausdruck**) vom Typ E.
(5) Ist A ein Ausdruck vom Typ E, so ist auch (A) ein Ausdruck vom Typ E.
Alle Funktionen, die im Rumpf R von f verwendet werden, bezeichnet man als **Stützfunktionen** von f.
Ein **funktionales Programm** ist eine Folge $f_1,\ldots,f_k$ von Funktionsdefinitionen.

Notation: Wenn wir im folgenden von Funktionen sprechen, meinen wir – soweit nicht anders erwähnt – immer Funktionen in Form Rechenvorschriften.

Man beachte, daß f in obiger Definition vollständig gecurryt ist. Für einen Wert a vom Typ D_1 ist der Ausdruck (f a) eine Rechenvorschrift mit n-1 Parametern $x_2:D_2,...,x_n:D_n$ und Funktionalität $D_2\rightarrow(D_3\rightarrow(...\rightarrow(D_n\rightarrow D)...))$, die mit dem Parameter a partiell ausgewertet ist. Ferner schreibt man vor einen Ausdruck nicht mehr das Schlüsselwort ergebnis.
Auf die präzise Semantik von Ausdrücken, die Art und Weise, sie auszuwerten, gehen wir später ein. Hier genügt die umgangssprachliche Beschreibung: Die Auswertung folgt der bekannten mathematischen Auswertung von Ausdrücken unter Berücksichtigung von Prioritätsregeln und Klammern. Nur der bedingte Ausdruck ist neu und bedarf der kurzen Erläuterung: Der Wert von

wenn B dann A1 sonst A2 ende

ist der Wert von A1, falls der Wert von B=true ist, und der Wert von A2, falls der Wert von B=false ist.
Überraschen mag vielleicht, daß man bei Rechenvorschriften ganz andere Bausteine und Konstruktoren verwendet als bei Programmen in PRO. Das liegt daran, daß wir nun von dem imperativen Programmierstil, den PRO verfolgt, in den funktionalen Programmierstil gewechselt haben, dem ein anderer Baukasten zugrundeliegt. Auf die genauen Unterschiede zwischen beiden Stilen gehen wir in einem späteren Kapitel ein.

Beispiele:

1) Die obige Rechenvorschrift zur Berechnung des ggT lautet – nun gecurryt – im Formalismus der Definition C:

```
funktion ggT a:nat b:nat → nat ≡
wenn a=b dann a sonst
    wenn a<b dann ggT b a sonst ggT (a-b) b ende ende.
```

   In dieser Definition kommen die Bausteine (2) bis (5) aus Definition C vor.

2) Die Absolutfunktion:

```
funktion abs x:real → real ≡ wenn x≥0 dann x sonst -x ende.
```

3) Die Signum-Funktion:

```
typ vorzeichen ≡ {-1,0,1};
funktion sign x:real → vorzeichen ≡
wenn x<0 dann -1 sonst
    wenn x=0 dann 0 sonst 1 ende ende.
```

4) Die Multiplikation und Verdoppelung gem. Beispiel in 10.2:

```
funktion mult x:int y:int → int ≡ x*y.
funktion d ≡ mult 2.
```

   Beachte: Bei d gibt man keine Parameter und keine Typen an, denn d übernimmt ja einerseits den noch "freien" Parameter von mult, andererseits ergeben sich die Typen

von Parameter und Funktionsergebnis direkt aus der Definition von mult. Auch das Schlüsselwort funktion ist eigentlich überflüssig, denn auch das folgt aus der Definition mit mult 2.

5) Die bereits bekannte Funktion zum Mischen zweier Zahlenfolgen notiert man durch:

```
typ intlist ≡ {leer} | (int,intlist);
funktion misch f:intlist g:intlist → intlist ≡
wenn f=leer dann g sonst
    wenn g=leer dann f sonst
        wenn (erstes f)<(erstes g) dann (erstes f,misch (rest f) g)
                                   sonst (erstes g,misch f (rest g))
        ende
    ende
ende .
```

6) Auch Konstanten können wie früher gezeigt durch die Funktionsdefinition erfaßt werden:

```
funktion pi → real ≡ 3.1415926;
funktion kreisfläche r:real → real ≡ pi*r*r.
```

10.4 Von Rechenvorschriften zu Werten: Applikation

Die wichtigste Operation im Zusammenhang mit einer allgemeinen Rechenvorschrift

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

ist die **Anwendung** (**Applikation**, **Aufruf**) von f auf einen Satz geeigneter Objekte (**Argumente**, **aktuelle Parameter**) $a_1,...,a_n$ der jeweils vorgeschriebenen Typen $D_1,...,D_n$, in Zeichen:

f a_1 a_2 ... a_n.

Bei der Auswertung eines Aufrufs ist eine weitere wichtige Operation beteiligt, die **Substitution**. Sie wandelt den Rumpf der mit f bezeichneten Rechenvorschrift in einen auswertbaren Ausdruck um, indem sie die formalen Parameter durch die aktuellen Parameter ersetzt.

Beispiel: Sei ggT definiert wie in Abschnitt 10.3. Durch Applikation von ggT auf die Argumente 40 und 12 und Substitution von a durch 40 und b durch 12 geht die Rechenvorschrift in den auswertbaren Ausdruck

```
wenn 40=12 dann 40 sonst
    wenn 40<12 dann ggT 12 40 sonst ggT (40-12) 12 ende ende
```

über. Diesen kann man durch Auswertung der Bedingungen zunächst zu ggT 28 12 und dann weiter auswerten.

Bei der Substitution der formalen durch die aktuellen Parameter besitzt man mehrere Freiheitsgrade, die durch unterschiedliche Substitutionsregeln geschlossen werden (s. Abschnitt 10.4.1).

10.4.1 Substitutionsregeln

Wenn eine Funktion (der Einfachheit halber eine einstellige)

funktion f x:D → D' ≡ R.

auf ein Argument E (=irgendein 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 von x durch E) und *wie oft* (an jeder Stelle, an der x vorkommt, oder einmalig) das Argument E ausgewertet wird.

Strategie 1: Call-by-value-Substitution.

Um f(E) zu berechnen, werte zunächst E aus. Ersetze x überall im Rumpf R durch den Wert von E und werte den so modifizierten Rumpf aus. Weil der formale Parameter nur den Wert des aktuellen Parameters übernimmt, spricht man auch von **Wertübergabe**. (Dies war auch die Strategie von PRO)

Beispiel: Sei

funktion 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: $A_1 \times ... \times A_n \rightarrow B$ heißt strikt, falls $g(a_1,...,a_n)=\perp$, sobald ein $a_i=\perp$ ist (s. Kapitel 3)).

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

  funktion 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 den bedingten Ausdruck

  wenn b dann e sonst e' ende

  Diesen Alternativkonstruktor können wir auch als Funktion

  wenn(b,e,e')

  darstellen. Verfolgt eine Programmiersprache nun konsequent die call-by-value-Strategie, so muß dies auch für die wenn-Funktion gelten. Dann kann man jedoch den bedingten Ausdruck nicht mehr in sinnvoller Weise verwenden. Dazu betrachte man die Rechenvorschrift

  funktion p x:int → int ≡ wenn(x=0,1,p(x-1))

  oder wie gewohnt

  funktion p x:int → int ≡

  wenn x=0 dann 1 sonst p(x-1) ende.

  Offenbar gilt p(x)=0 für alle x∈ int, x≥0. Andererseits führt jede Anwendung von p wegen der call-by-value-Strategie zu einer nicht terminierenden Berechnung, weil die Auswertung von p nicht abbricht:

  p 0 ⇒ wenn(0=0,1,p(-1)) ⇒

  wenn(true,1,p(-1=0,1,p(-2)))=wenn(true,1,p(false,1,p(-2))) ⇒ ...

  Folglich ist hier in jedem Falle eine Änderung der Strategie erforderlich: Bei einem bedingten Ausdruck darf je nach Wert der Bedingung nur entweder der wenn-Zweig oder der sonst-Zweig ausgewertet werden, aber nicht beide.

Strategie 2: Call-by-name-Substitution (**Namensübergabe**).

Um f(E) zu berechnen, ersetze x überall im Rumpf R durch den Text von E und werte den so modifizierten Rumpf aus. Hier wird der aktuelle Parameter E also erst dann ausgewertet, wenn er zur Auswertung des modifizierten Rumpfes benötigt wird.

Beispiele:

1) Mittels call-by-name wird das Ergebnis von null(d(d(d 3))) in einem Schritt korrekt berechnet. Der Wert des aktuellen Parameters d(d(d 3)) wird überhaupt nicht berechnet, da er nach textueller Ersetzung im Rumpf nicht mehr vorkommt.
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 wiederholte Auswertung desselben Ausdrucks, obwohl sein Wert schon feststeht. Denn jedes Vorkommen von (d 3) kann sofort durch seinen Wert 6 ersetzt werden, sobald (d 3) zum ersten Mal berechnet worden ist. Das gleiche gilt anschließend für d(d 3). Auf dieser Idee basiert die dritte Strategie.

Strategie 3: Call-by-need-Strategie (**lazy evaluation**).
Um f(E) zu berechnen, ersetze x überall in R textuell durch E und werte den so modifizierten Rumpf aus. Sobald E hierbei zum ersten Male ausgewertet wird, ersetze E im modifizierten Rumpf überall durch den soeben 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 wenn(b,e,e') wird wie gewünscht ausgewertet.

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.

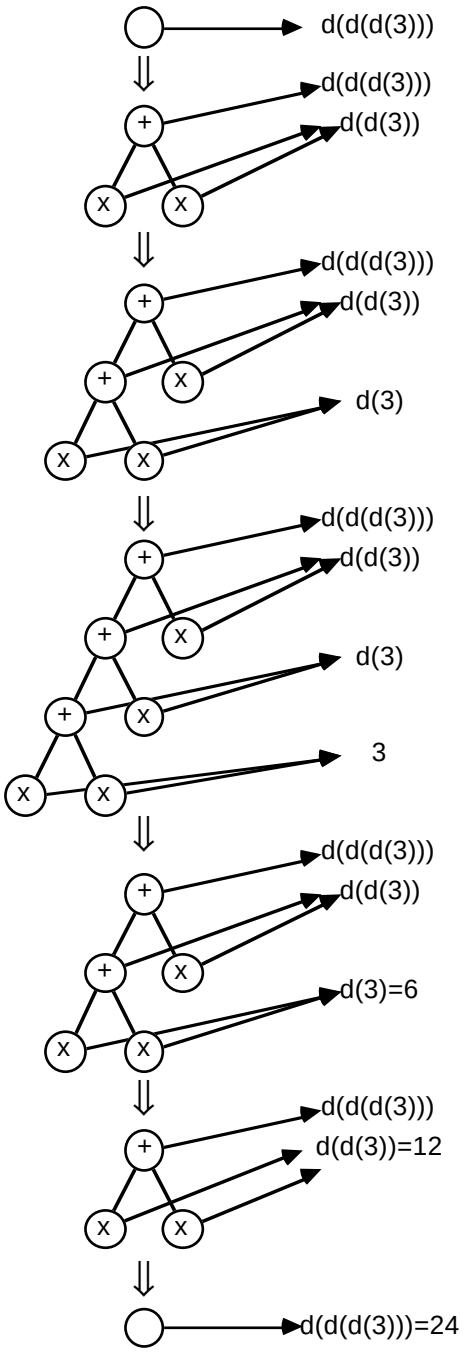


Abb. 1: Auswertung eines Ausdrucks mittels call-by-need

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 liefert aber auch dann den Wert 0, wenn der aktuelle Parameter undefiniert ist, also z.B. einen nicht terminierenden Funktionsaufruf enthält. Dies widerspricht der mathematischen Tradition, wonach ein Ausdruck nur dann eine Bedeutung besitzt, wenn auch alle Teilausdrücke eine Bedeutung haben.
Beispiel: Der Ausdruck

null(2 div 0)

wird zu 0 ausgewertet, obwohl der Parameter 2 div 0 nicht zulässig ist, denn durch Null darf man bekanntlich nicht dividieren.

Ein besonderer Vorteil der lazy-evaluation – deswegen wird diese Substitutionsregel so gefeiert – besteht in der Möglichkeit, prinzipiell unendliche Objekte (sog. **lazy lists**) zu definieren und mit ihnen zu operieren. Dies eröffnet neuartige und sehr elegante Programmiermöglichkeiten, aber das führt in dieser Vorlesung zu weit. Hier sei auf vertiefende Veranstaltungen über funktionale Programmierung verwiesen.

Soweit ein Vergleich der Vor- und Nachteile. Ungeklärt bleibt noch, ob die Substitutionsregeln die gleiche Mächtigkeit besitzen: Kann man alle berechenbaren Funktionen beschreiben, wenn man nur die call-by-value-, nur die call-by-name- oder nur die call-by-need-Strategie zur Verfügung hat? Wir können diese Frage hier nicht klären und verweisen auf Veranstaltungen über Semantik.

Wir werden uns trotz der erwähnten Nachteile für den Rest der Vorlesung für die call-by-value-Substitution entscheiden.
Das Typkonzept aus Kapitel 9 verbunden mit dem Funktionskonzept aus Kapitel 10 (Definition C) und der call-by-value-Substitution bildet eine neue (funktionale) Programmiersprache, die wir im folgenden als **FUN** bezeichnen.

10.4.2 Die Formularmaschine

Bisher haben wir Funktionen durch Texte dargestellt. Die Auswertung dieser Funktionen erfordert oft einige Mühe. Hierzu betrachte man das (rekursive) Beispiel

funktion p x:int → int ≡
wenn x=0 dann 1 sonst p(p(x-1)) ende,

dessen Berechnung für gegebene Parameterwerte schon recht unübersichtlich werden kann, wenn man sich vorher keine vernünftige Darstellung überlegt, um die zahlreichen Aufrufe von p überblicken zu können. Wir wollen uns in diesem Abschnitt mit einer Darstellung von Funktionen befassen, die eine geordnete Auswertung solcher Funktions-

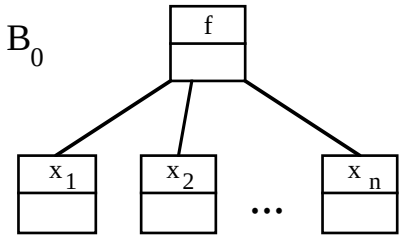
aufrufe gestattet. Dabei werden Funktionen durch sog. *Formulare* in graphischer Form veranschaulicht. Die Auswertung dieser so dargestellten Funktionen kann schematisch mit Papier und Bleistift durchgeführt werden. Art und Weise dieser Darstellung verbunden mit dem algorithmischen Verfahren der Auswertung von Funktionen bezeichnet man als *Formularmaschine*.

Definition D:

Sei f eine Rechenvorschrift definiert durch

$$\underline{\text{funktion}}\ \ f\ \ x_1{:}D_1\ x_2{:}D_2\ \ldots\ x_n{:}D_n \rightarrow D \equiv R.$$

Ein **Formular** für f ist ein (geordnetes) Paar geordneter markierter Bäume $(B_0, B(R))$, das wie folgt definiert ist: B_0 repräsentiert den Kopf der Rechenvorschrift:

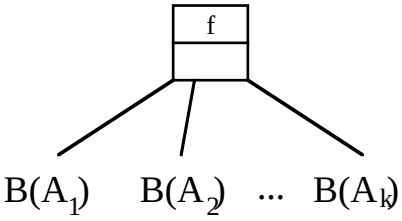


B(R) repräsentiert den Rumpf der Rechenvorschrift und ist induktiv wie folgt aufgebaut:

(1) Für jede Konstante x, ist das zugehörige Formular B(x) der Baum

(2) Für jeden formalen Parameter x, ist das zugehörige Formular B(x) der Baum

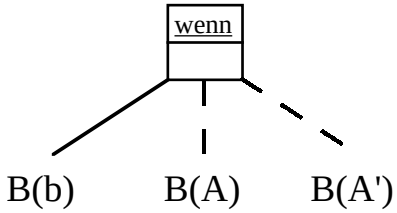
(3) Seien $A_1,\ldots,A_k$ Ausdrücke mit den zugehörigen Formularen $B(A_1),\ldots,B(A_k)$. Für jeden Aufruf $(f\ A_1\ \ldots\ A_k)$ einer k-stelligen Funktion f ist das zugehörige Formular B(f) der Baum



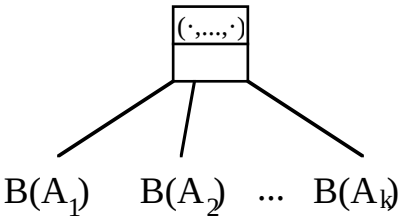
(4) Ist b ein boolescher Ausdruck mit Formular B(b) und sind A und A' beliebige Ausdrücke mit den Formularen B(A) und B(A'), so gehört zum Ausdruck

wenn b dann A sonst A' ende

das Formular B(wenn) mit (beachte die gestrichelten Kanten bei beiden Alternativen)

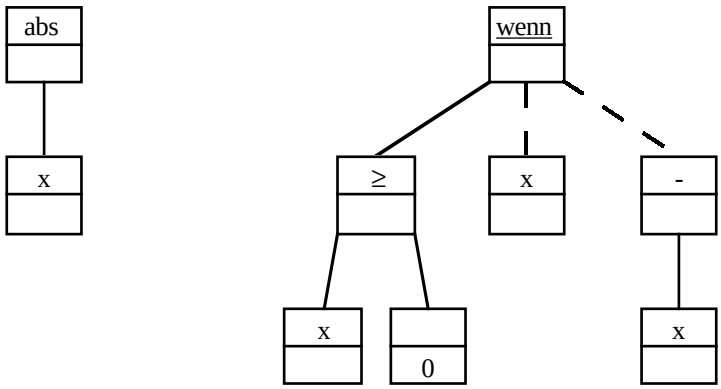


(5) Seien $A_1,...,A_k$ Ausdrücke mit den zugehörigen Formularen $B(A_1),...,B(A_k)$. Für jeden Tupelausdruck $(A_1,...,A_k)$ ist das zugehörige Formular $B((A_1,...,A_k))$ der Baum

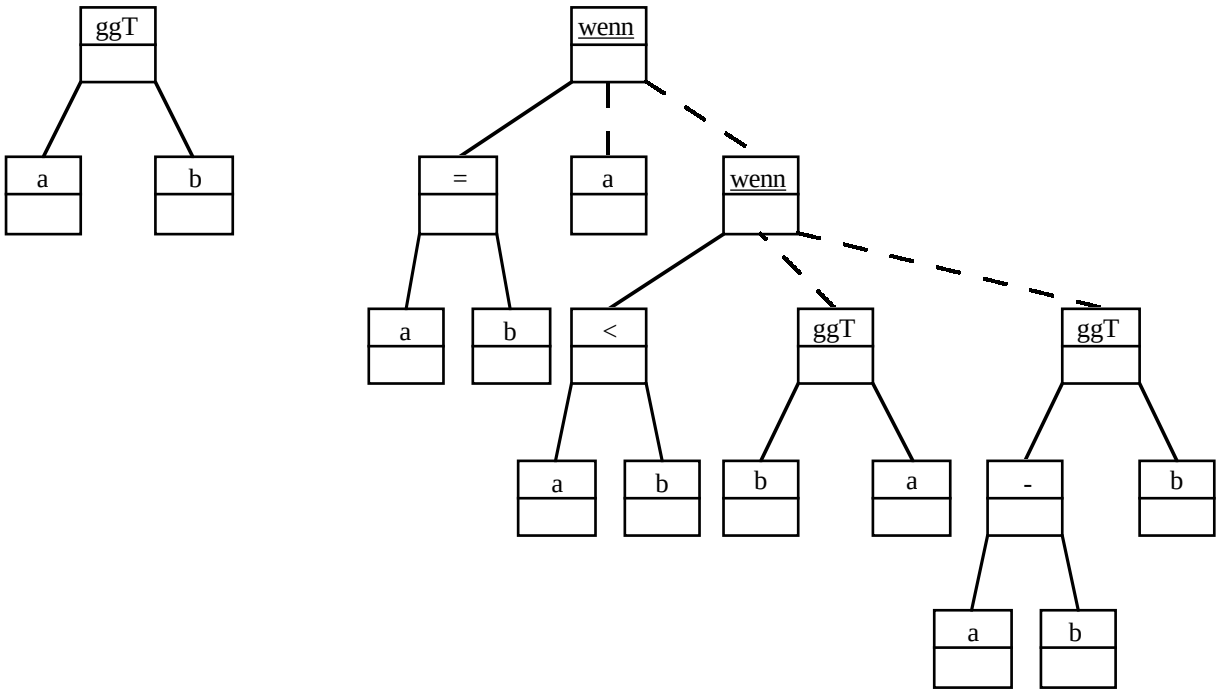


Beispiele:

1) Für die Funktion abs aus Abschnitt 10.3 lautet das Formular so:



2) Die Funktion ggT aus Abschnitt 10.3 ist durch folgendes Formular repräsentiert:



Soweit die Darstellung von Rechenvorschriften. Wie rechnet man mit diesen Formularen? Die Verfahrensweise orientiert sich an dem alltäglichen Umgang mit Formularen (etwa dem Formular für die Einkommensteuererklärung). Man trägt in gewisse Felder Werte ein, verknüpft die Werte von Feldern, überträgt Ergebnisse in andere Felder und kommt schließlich zu einem Resultat (z.B. dem zu versteuernden Einkommen).
Der Ablauf im einzelnen: Beim Aufruf $f\ a_1 \ldots a_n$ einer Funktion f, definiert durch

$$\underline{\text{funktion}}\ \ f\ \ x_1{:}D_1\ x_2{:}D_2 \ldots x_n{:}D_n \rightarrow D \equiv R,$$

legt man das Formular $(B_0,B(R))$ an und trägt die aktuellen Parametern a_i (dies sind wegen der Übergabeart call by value konkrete Werte) jeweils in den unteren Teil der mit x_i markierten Knoten zunächst von B_0 und danach von $B(R)$ ein. $(B_0,B(R))$ heißt danach *Inkarnation* der Funktion. Anschließend wertet man das Formular $B(R)$ aus, indem man die Ergebnisse der Teilbäume entsprechend der Operationen in ihren jeweiligen Wurzeln zur Wurzel des Gesamtbaumes schiebt. Kommt man zum Aufruf einer Rechenvorschrift g, die *nicht* elementar ist, so legt man ein Formular für g an und wertet dieses zunächst in gleicher Weise aus. Für elementare Rechenvorschriften ist kein neues Formular erforderlich. Einen Sonderfall bildet der bedingte Ausdruck: Hier wertet man zuerst die Bedingung aus; je nach Ergebnis trennt man den dann-Zweig oder den sonst-Zweig final vom Baum ab, der abgetrennte Zweig wird nicht mehr ausgewertet. Dieser Sonderfall wird durch die gestrichelten Kanten symbolisiert und spiegelt die Überlegung aus Abschnitt 10.4.1 wider, wonach eine strikte Auswertung des bedingten Ausdrucks, also eine Auswertung beider Alternativen, unsinnig ist.

Definition E:
Ein konkretes Exemplar des Formulars einer auszuwertenden Funktion f bezeichnet man als **Inkarnation** von f.

Beispiele:
1) Auswertung von abs(-7) in Abb. 2.
2) Auswertung von ggT(6,4) in Abb. 3. Man erkennt hier, daß die Formularmaschine offenbar auch für selbstbezügliche (*rekursive*) Funktionen korrekt arbeitet.

Die schematische Vorgehensweise, mit der man ein Formular auswertet, legt unmittelbar nahe, daß man das gleiche Verfahren auch automatisieren kann. Tatsächlich arbeitet ein Computer – und daran erinnert der Begriff "Formularmaschine", die hier aber nur eine Gedankenmaschine ist – Funktionsaufrufe in etwa dieser Form ab, wobei aber noch zahlreiche effizienzsteigernde Techniken eingesetzt werden. So braucht man z.B. nicht immer ein neues Formular anzulegen, sondern man verwendet immer das gleiche und *stapelt* in den Knoten nur die aktuellen Werte so aufeinander, daß immer die Werte der aktuellen Inkarnation sichtbar und die übrigen "verschattet" sind. Die fundamentale Datenstruktur zur Realisierung dieses Prinzips ist der Stapel, auf den wir spätestens in der Vorlesung Algorithmen, Daten, Programme II eingehen.

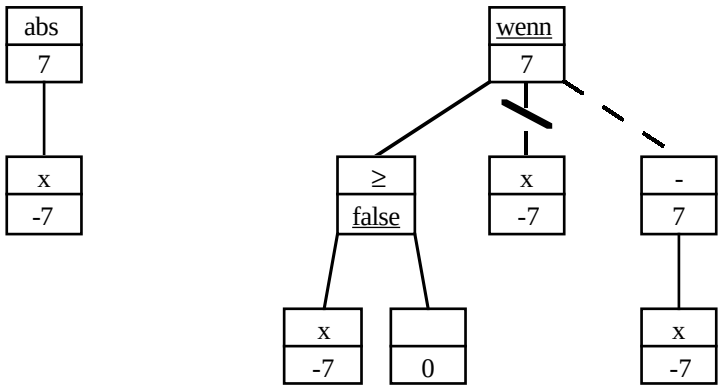


Abb. 2: Formularauswertung für abs(-7)

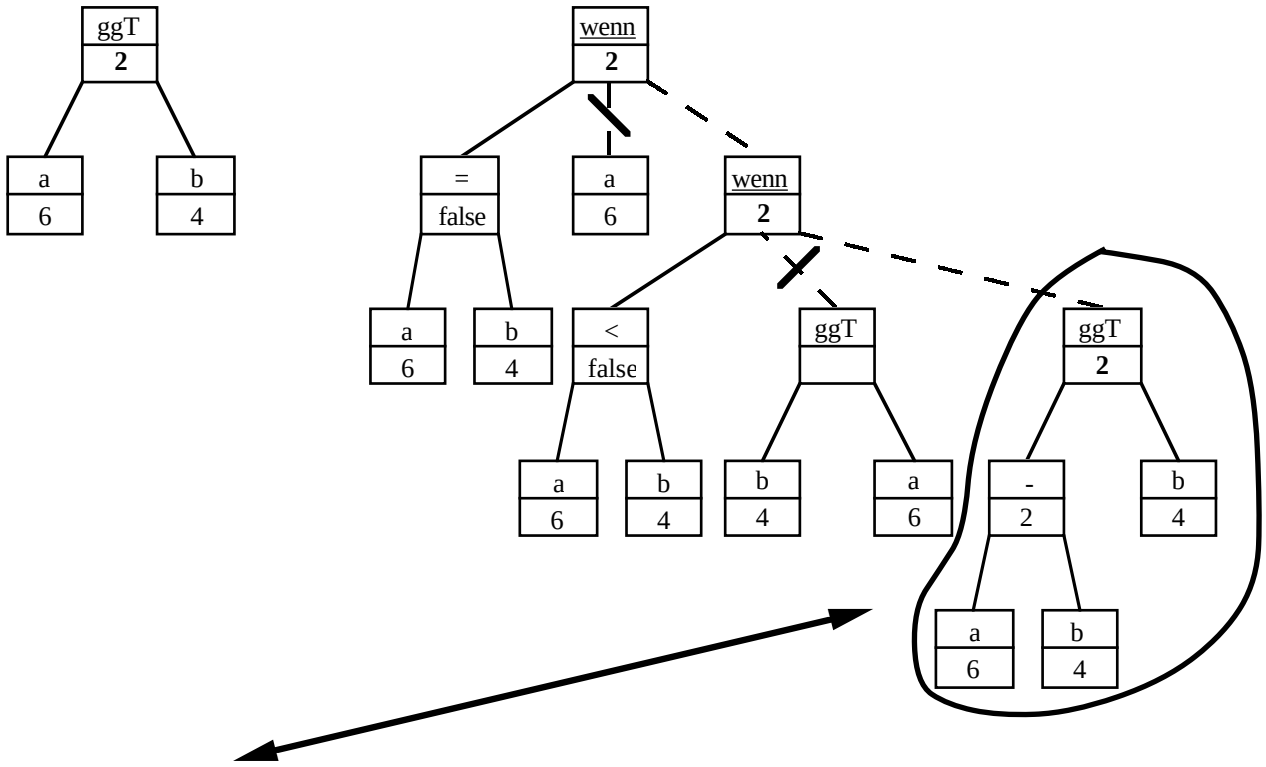


Abb. 3: Formularauswertung für ggT(6,4)

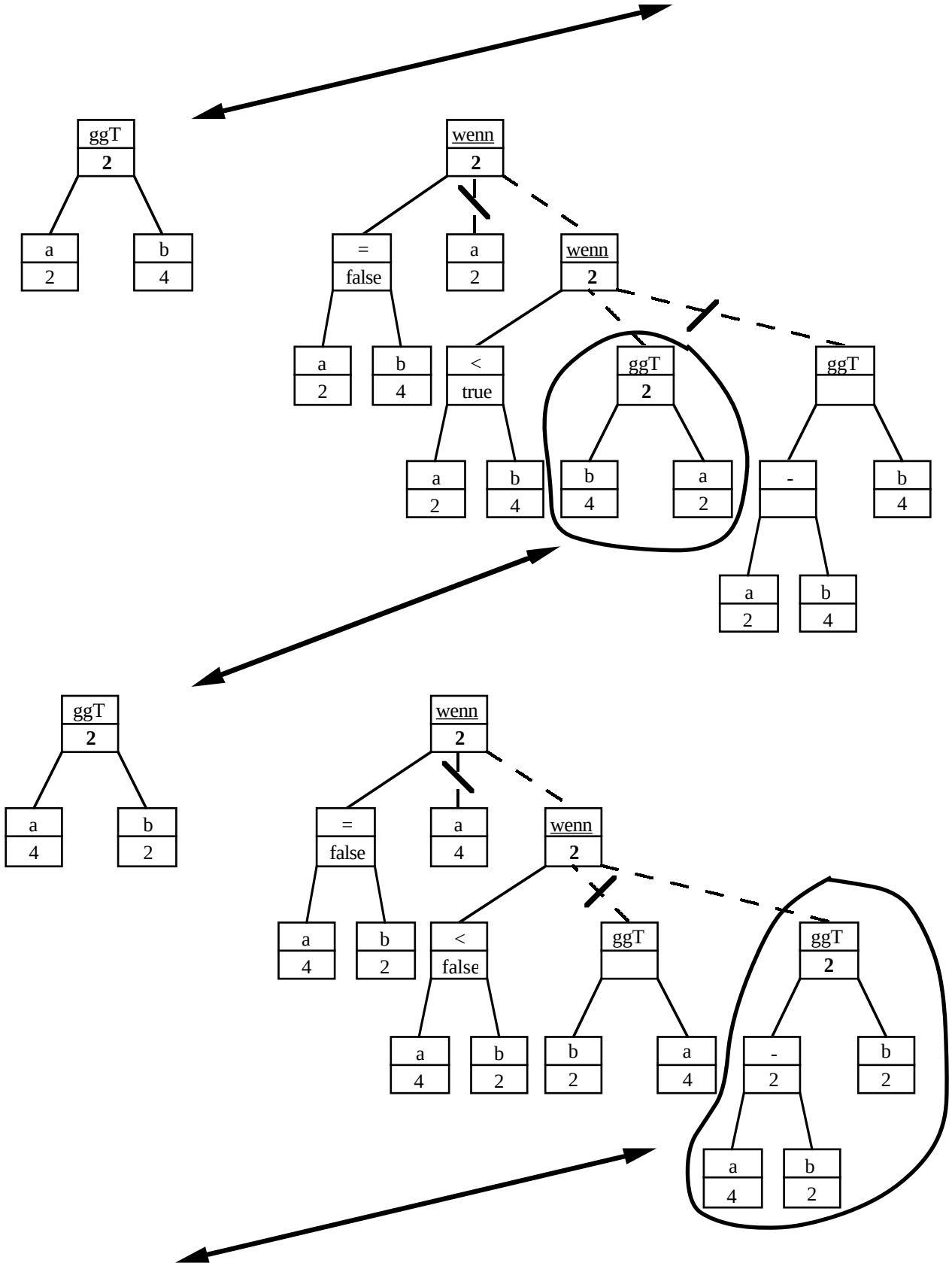


Abb. 3 (Forts.): Formularauswertung für ggT(6,4)

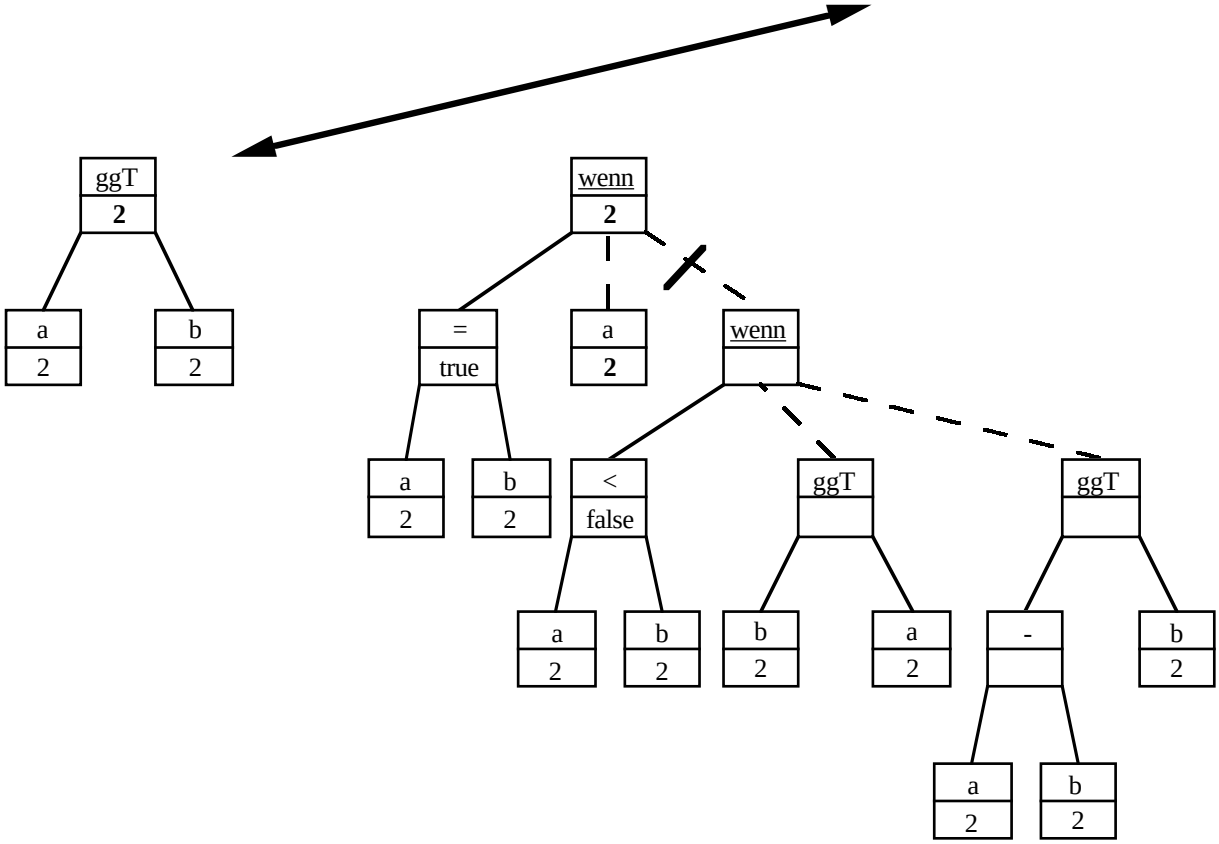


Abb. 3 (Forts.): Formularauswertung für ggT(6,4)

10.5 Von Ausdrücken zu Rechenvorschriften: Abstraktion

Den Konstruktor "Abstraktion" haben wir schon im Zusammenhang mit PRO kennengelernt. Zur erneuten 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) ein Parameter 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

$$f: IN^3 \rightarrow IN \text{ mit}$$

$$f(\pi,r,h)=\pi r^2 h$$ betrachten, oder auch als

f: IN→(IN→(IN→IN)) mit

f π r h=πr^{2}h betrachten, oder auch als

g: Z→IR mit

g(r)=πr^{2}h betrachten, oder auch als

h: IR→IR mit

h(x)=πr^{2}h (h ist jetzt eine konstante Funktion),

oder mit der intendierten Bedeutung "Zylindervolumen"

V: IR→(IR→IR) mit

V r h=πr^{2}h.

In der Mathematik lassen sich diese unterschiedlichen Auffassungen von der Bedeutung eines Ausdrucks 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-)Ausdrucks zu einer Funktion nebst geeigneter Parametrisierung.

Beispiel: Den Ausdruck für das Zylindervolumen abstrahieren wir zu:

funktion V r:real h:real → real ≡ pi*r*r*h.

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

funktion F r: real→real ≡ pi*r*r.

wodurch V vereinfacht werden kann zu

funktion V r:real h: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. Beweisgrundlage ist der sog. -Kalkül, eine ganz primitive funktionale Programmiersprache, in der es *nur* diese Konstruktoren gibt und sonst nichts, nicht einmal Datentypen, Zahlen, Wahrheitswerte oder arithmetische Operationen.

Die Parameter in Funktionen besitzen nur Platzhalterfunktion. Das bedeutet insbesondere, daß die Wahl der Bezeichner weitgehend irrelevant ist, sofern eine Umbenennung nur konsistent geschieht. Tatsächlich macht es keinen Unterschied, ob das Volumen wie in obigem Beispiel definiert ist oder durch

funktion V a:real b:real → real ≡ pi*a*a*b

oder durch

funktion V h:real r:real → real ≡ pi*h*h*r

aber nicht durch

funktion V pi:real r:real → real ≡ pi*pi*pi*r

Für diese Situation hat sich der Begriff der *Bindung* etabliert.

Definition F:
Ein Bezeichner innerhalb des Rumpfes einer Funktionsdefinition heißt **gebunden**, wenn er formaler Parameter der Funktion ist, anderenfalls heißt er **frei**. Der **Bindungsbereich** eines gebundenen Bezeichners ist die gesamte Rechenvorschrift.

Beispiel: In den Definitionen von V und F im letzten Beispiel sind jeweils r und h gebundene Bezeichner, pi und F sind freie Bezeichner.

Betrachten wir nochmal die Definition

funktion V r:real h:real → real ≡ (F r)*h.

Um diese Funktion für konkrete aktuelle Parameter auszuwerten, muß man auf den freien Bezeichner F zugreifen und nachprüfen, ob F eine Funktion ist, die mit dem Argument r etwas anfangen kann. Wie und wo findet man aber F? Was passiert, wenn man zwei verschiedene F findet? Und wenn man F anwendet, wo findet man dann pi? Für diese Problematik setzt man das Konzept der Bindung innerhalb von Funktionen auf Folgen von Funktionsdefinitionen, also funktionale Programme fort. Man unterscheidet im wesentlichen zwei verschiedene Strategien, die statische und die dynamische Bindung.
Bei der *statischen Bindung* werden Bezeichner bezgl. ihrer *Definitions*stelle gebunden. Dies hat folgende Konsequenzen: Wird ein Bezeichner für einen anderen Zweck neu definiert, so beziehen sich alle bisherigen Zugriffe auf seinen ursprünglichen Wert. Die neue Bedeutung des Bezeichners kommt erst für die nachfolgenden Zugriffe zum Tragen. Hiervon zu unterscheiden ist die *dynamische Bindung*, bei der alle Bezeichner bezgl. ihrer *Zugriffs*stelle an Werte gebunden werden.
Beispiel: Abb. 4 zeigt Unterschiede zwischen dynamischer und statischer Bindung. Die Neudefinition von F in der vorletzten Zeile wirkt sich bei der statischen Bindung nur auf nachfolgende Definitionen aus. Bei der dynamischen Bindung wirkt sie sich sofort aus mit der Folge, daß bereits der Aufruf V(17.0,3.0) die neue Definition von F verwendet.

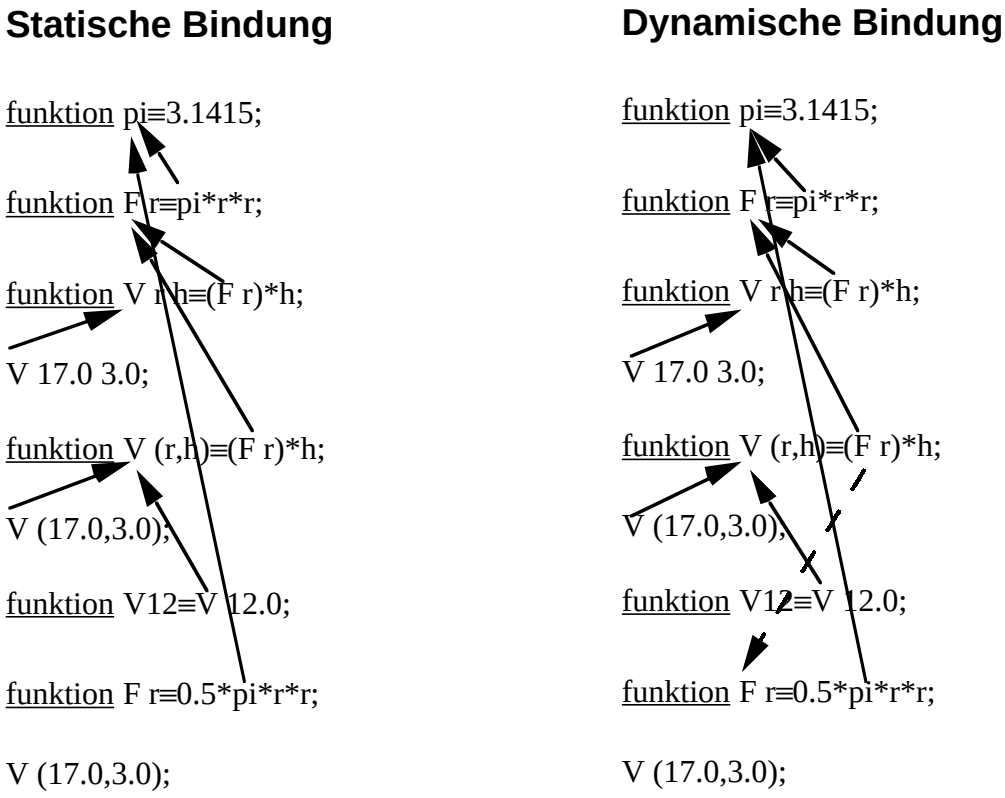


Abb. 4: Statische und dynamische Bindung

10.6 Rekursion

Innerhalb des Rumpfes einer Funktion f wird meist eine Vielzahl weiterer Funktionen (die Stützfunktionen von f) aufgerufen. Eine besondere Situation tritt auf, wenn unter den Stützfunktionen f selbst wieder vorkommt. In diesem Fall spricht man von *Rekursion*. f ist dann eine rekursive Funktion. Früher in diesem Kapitel hatten wir bereits die ggT-Funktion und waren über deren selbstbezügliche Definition verwundert. Gleichwohl konnten wir mit dieser Definition für die durchgespielten Beispiele zum korrekten Ergebnis gelangen.

Schon bei rekursiven Datentypen hatten wir erlebt, daß der Konstruktor "Rekursion" für einen Quantensprung sorgt, indem er von endlichen Strukturen zu unendlichen Strukturen führt. Ähnliches werden wir auch auf der operationalen Ebene feststellen. Rekursive Funktionen sind zugleich das geeignete Mittel zur Bearbeitung rekursiver Datentypen.

Schon aus der Schulmathematik kennt man rekursive Funktionen. Rekursive Darstellungen sind häufig kürzer und leichter verständlich, da sie charakteristische Eigenschaften einer Funktion oder eines Problems betonen. Ein typisches Beispiel ist die Fakultätsfunktion

$$f(n)=n!=1\cdot 2\cdot\ldots\cdot n.$$

Die rekursive Definition von f lautet:

$$f(n)=\begin{cases}1, & \text{falls } n=0,\\ n\cdot f(n-1), & \text{falls } n>0.\end{cases}$$

Um z.B. f(4) zu berechnen, ersetzt man die Funktionsanwendung fortwährend durch die rechte Seite der Definitionsgleichung, also

$$\begin{aligned} f(4) &= 4 \cdot \mathbf{f(3)} \\ &= 4 \cdot 3 \cdot \mathbf{f(2)} \\ &= 4 \cdot 3 \cdot 2 \cdot \mathbf{f(1)} \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot \mathbf{f(0)} \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24. \end{aligned}$$

Damit solch eine Rekursionsdefinition sinnvoll ist, müssen vor allem folgende beiden Bedingungen erfüllt sein:

1) In jedem Ersetzungsschritt einer Funktionsanwendung durch die rechte Seite der Definition vereinfacht sich das Argument der Funktion. (Bei der Fakultätsfunktion "vereinfacht" sich das Argument von n auf n-1.)
2) Die Ersetzung terminiert, d.h., es gibt mindestens einen einfachsten Fall, ein sog. *terminales* Argument, für das der Wert der Funktion unmittelbar gegeben ist. (Bei der Fakultätsfunktion ist das der Fall n=0, für den der Funktionswert direkt ablesbar ist.)

Bei der Konstruktion einer rekursiven Funktion f zu einem Problem P geht man genau umgekehrt vor. Gegeben ist eine funktionale Spezifikation von P. Man überlegt sich zunächst den einfachsten Fall von P. Typische Exemplare einfachster Fälle sind

- bei Problemen auf Zahlen die Zahlen Null oder Eins, sprich: Wie lautet die Lösung von P für die Eingabe Null oder Eins?
- bei Problemen auf Folgen oder Files die leere Folge oder das leere File, sprich: Wie lautet die Lösung von P für die Eingabe [ ]?
- bei Problemen auf Texten der leere Text, sprich: Wie lautet die Lösung von P für die Eingabe ""?
- bei Problemen auf Bäumen der leere Baum oder der Baum mit einem Knoten, sprich: Wie lautet die Lösung von P für den leeren Baum?

Anschließend stellt man eine Beziehung zwischen dem allgemeinen Problem der Problemgröße n und dem nächstkleineren Problem her unter der Annahme, man könne bereits das Problem der Größe n-1 lösen. Typische Überlegungen sind hier:

- Bei Zahlen: Man kennt die Lösung für Zahlen der Größe n-1. Wie ermittelt man daraus die Lösung für Zahlen der Größe n?
- Bei Folgen: Man kennt die Lösung für Folgen der Länge n-1. Wie ermittelt man daraus die Lösung für Folgen der Länge n?
- Bei Texten: Man kennt die Lösung für Texte der Länge n-1. Wie ermittelt man daraus die Lösung für Texte der Länge n?
- Bei Bäumen: Man kennt die Lösung für Bäume der Höhe n-1. Wie ermittelt man daraus die Lösung für Bäume der Höhe n?

Nach Beantwortung dieser Fragen formuliert man den einfachsten Fall und den Reduktionsschritt zwischen der Problemgröße n und er nächstkleineren als rekursive Funktion.

Definition G:
Die Definition eines Problems, eines Verfahrens oder einer Funktion durch sich selbst bezeichnet man als **Rekursion**.
Erscheint im Rumpf einer rekursiven Funktion f ein Aufruf von f selbst in der Form

funktion f ... ≡ ... f ...

so spricht man von **direkter Rekursion**.
Gibt es eine Folge von Funktionen $f=f_1,f_2,f_3,...,f_n$, $n\geq 2$ der Art, daß sich für $1\leq i\leq n-1$ jeweils f_i auf f_{i+1} und f_n wiederum auf $f=f_1$ abstützt, so spricht man von **indirekter Rekursion**.

Beispiele:

1) Wir definieren die Addition add zweier natürlicher Zahlen rekursiv. Dazu führen wir die Addition auf die Nachfolgerfunktion +1 zurück.
   Der einfachste Fall: Die Addition von 0 zu einer Zahl x liefert wieder x.
   Reduktionsschritt: Um x und y zu addieren, addiert man mit der Funktion selbst zunächst x und y-1 und erhöht das Ergebnis um Eins. Die Definition:

   funktion add x:nat y:nat → nat ≡
   wenn y=0 dann x sonst (add x (y-1))+1 ende.

   Nun ist z.B.

   add 5 3=(add 5 2)+1=(add 5 1)+1+1=(add 5 0)+1+1+1=5+1+1+1=8.

2) Wir definieren die Multiplikation mult zweier natürlicher Zahlen rekursiv. Dazu führen wir die Multiplikation auf die Addition zurück.
   Der einfachste Fall: Die Multiplikation einer Zahl x und 0 ist 0.

Reduktionsschritt: Um x und y zu multiplizieren, multipliziert man mit der Funktion selbst zunächst x und y-1 und addiert zum Ergebnis x. Die Definition:

```
funktion mult x:nat y:nat → nat ≡
wenn y=0 dann 0 sonst add (mult x (y-1)) x.
```

3) Die Exponentiation exp zweier natürlicher Zahlen durch Rückführung auf die Multiplikation.
Der einfachste Fall: $x^0=1$.
Reduktionsschritt: $x^y=x^{y-1}\cdot x$.
Die Definition:

```
funktion exp x:nat y:nat → nat ≡
wenn y=0 dann 1 sonst mult (exp x (y-1)) x ende.
```

4) Konkatenation zweier Folgen ganzer Zahlen definiert durch:

```
typ intlist ≡ {leer} | (int,intlist).
```

Der einfachste Fall: Konkateniert man eine beliebige Folge y mit einer leeren Folge, so ist y das Ergebnis.
Reduktionsschritt: Um eine Folge x mit einer Folge y zu konkatenieren, konkateniert man x ohne das erste Element mit y und stellt anschließend dieses erste Element dem Ergebnis voran. Die Definition:

```
typ intlist ≡ {leer} | (int,intlist);
funktion concat x:intlist y:intlist → intlist ≡
wenn x=leer dann y sonst (erstes x,concat (rest x) y) ende.
```

5) Spiegeln einer Linkssequenz ganzer Zahlen, z.B. (2,(65,(54,(3,(1,leer))))) -> (1,(3,(54,(65,(2,leer))))).
Der einfachste Fall: Spiegelt man eine leere Zahlenfolge, so ist die leere Folge das Ergebnis.
Reduktionsschritt: Um eine Folge mit n Elementen zu spiegeln, trennt man das erste Element ab, spiegelt den Rest und fügt das erste Element hinten an das Ergebnis an.
Die Definition:

```
typ intlist ≡ {leer} | (int,intlist);
funktion spiegel x:intlist → intlist ≡
wenn x=leer dann x sonst concat (spiegel (rest x)) (erstes x,leer) ende.
```

6) Gesucht ist ein Funktional map, das eine Funktion f:int→int und eine Linkssequenz ganzer Zahlen als Parameter erwartet und die Linkssequenz als Ergebnis liefert, in der f auf jedes Folgenelement angewendet wurde.
Der einfachste Fall: Für die leere Linkssequenz ist nichts zu tun; die leere Linkssequenz ist dann auch das Ergebnis.

Reduktionsschritt: Bei einer nicht-leeren Linkssequenz wendet man f zunächst auf das erste Element an, wendet anschließend map auf den Rest der Linkssequenz an und verknüpft die Ergebnisse. Die Definition:

typ intlist ≡ {leer} | (int,intlist);
funktion map f:[int→int] x:intlist → intlist ≡
wenn x=leer dann leer sonst (f (erstes x),map f (rest x)) ende.

Definiert man nun z.B.

funktion plus7 ≡ add 7.

so liefert der Aufruf

map plus7 (2,(45,(26,(65,54))))

die Ergebnisfolge

(9,(52,(33,(72,61)))).

7) Erzeugung einer Folge von k+1 ganzen Zahlen durch wiederholte Anwendung einer Funktion f auf einen Startwert a; gesucht ist also ein Funktional, das Folgen ganzer Zahlen der Form

(a,(f(a),(f^2(a),(f^3(a),(...,(f^k(a),leer)...))))

generiert.

Einfachster Fall: Für k=0 ist die Folge (a,leer) das Ergebnis.

Reduktionsschritt: Für k>0 bildet man die Folge der k Elemente ausgehend vom neuen Startwert f(a) und ergänzt vorne den Startwert a. Die Definition:

typ intlist ≡ {leer} | (int,intlist);
funktion generate f:[int→int] a:int k:int → intlist ≡
wenn k=0 dann (a,leer) sonst (a,generate f (f a) (k-1)) ende.

Die obigen Beispiele beschreiben jeweils *direkt rekursive* Funktionen. Das folgende Beispiel behandelt einen *indirekt rekursiven* Fall.

Beispiel:

8) Gesucht sind zwei Funktionen gerade und ungerade mit jeweils einem Parameter, die den Ergebnistyp bool besitzen und den Wert true liefern, falls der aktuelle Parameter eine gerade bzw. eine ungerade Zahl ist. Offenbar gilt:

Der einfachste Fall: Hier ist nun ein Paar einfachster Fälle für beide Funktionen gesucht. Offenbar gilt:

gerade(0)=true und
ungerade(0)=false.

Reduktionsschritt: Ferner stehen gerade und ungerade für x>0 in folgender Beziehung zueinander:

gerade(x)=ungerade(x-1) und

ungerade(x)=gerade(x-1).

Diese Überlegungen führen zu folgendem Funktionenpaar:

funktion gerade x:nat → bool ≡
wenn x=0 dann true sonst ungerade (x-1) ende.
funktion ungerade x:nat → bool ≡
wenn x=0 dann false sonst gerade (x-1) ende.

Schon aus diesen wenigen Beispielen wird deutlich, daß es nicht immer einfach ist rekursive Funktionen zu konstruieren oder gar herauszufinden, was eine rekursive Funktion leistet, genauer: welche Semantik sie besitzt. Die Ursache liegt vor allem daran, daß man gegenüber nicht-rekusiven Funktionen, die endliche Prozesse beschreiben, einen Schritt zu unendlichen Prozessen vollzogen hat. Rekursive Funktionen erfordern daher besondere Aufmerksamkeit, wenn es um den Nachweis ihrer Korrektheit bezgl. einer gegeben Spezifikation geht. Dazu werden anspruchsvolle mathematische Hilfsmittel benötigt, die wir in späteren Kapiteln ansatzweise einführen.

Wir wollen die Rekursion im folgenden anhand eines weiteren Standard-Beispiels studieren.

Beispiel: Türme von Hanoi.
Es handelt hierbei sich um ein altes ostasiatisches Spiel: Gegeben seien n Scheiben (n≥1) unterschiedlichen Durchmessers, die der Größe nach geordnet zu einem Turm geschichtet sind; die unterste Scheibe ist die größte. Der Turm steht auf einem Anfangsplatz mit der Nummer 1. Unter Verwendung eines Hilfsplatzes 3 soll der Turm auf den Zielplatz mit der Nummer 2 transportiert werden. Beim Transport sind folgende Bedingungen einzuhalten:

1) Es darf stets nur eine Scheibe, und zwar die oberste eines Turmes bewegt werden.
2) Zu keiner Zeit darf eine größere Scheibe auf einer kleineren liegen.

Abb. 5 zeigt die Anfangs-, eine Zwischen- und die Endsituation des Problems für n=4 Scheiben.

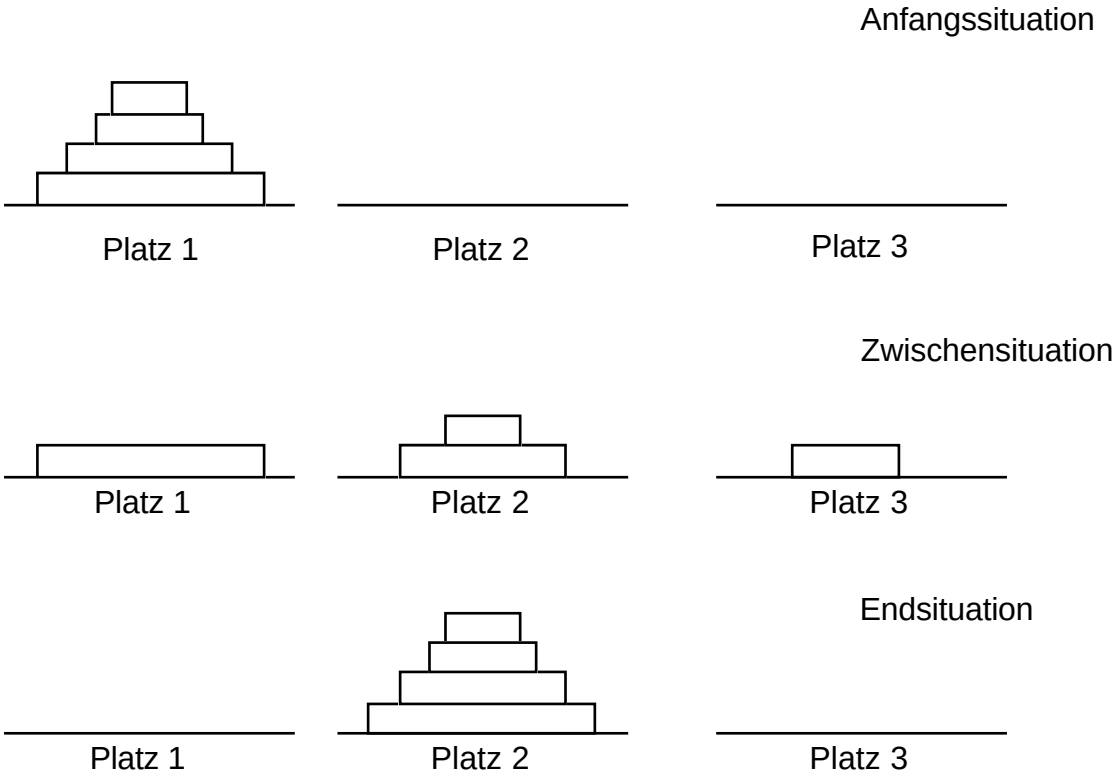


Abb. 5: Situation bei "Türme von Hanoi"

Die Aufgabe lautet: Entwickle ein Programm, das zu jedem n≥1 nacheinander ausgibt, welche Scheiben von welchem Platz auf welchen anderen Platz bewegt werden sollen.
Die rekursive Lösung erhält man leicht durch folgende Überlegungen:

a) Einfachster Fall: n=1.
   Falls n=1 ist, so transportiert man eine Scheibe vom Platz 1 auf Platz 2. Damit ist die Aufgabe gelöst. (Platz 3 benötigt man gar nicht.)

b) Reduktionsschritt.
   Falls n>1 ist, so sind n Scheiben von Platz 1 nach Platz 2 zu bewegen. Dazu bewegt man zunächst die n-1 obersten Scheiben von Platz 1 nach Platz 3, danach transportiert man die noch auf Platz 1 liegende größte Scheibe auf ihren endgültigen Platz 2, und anschließend bewegt man die n-1 Scheiben von Platz 3 nach Platz 2, also an ihre endgültige Position. Abb. 6 verdeutlicht die drei Schritte.

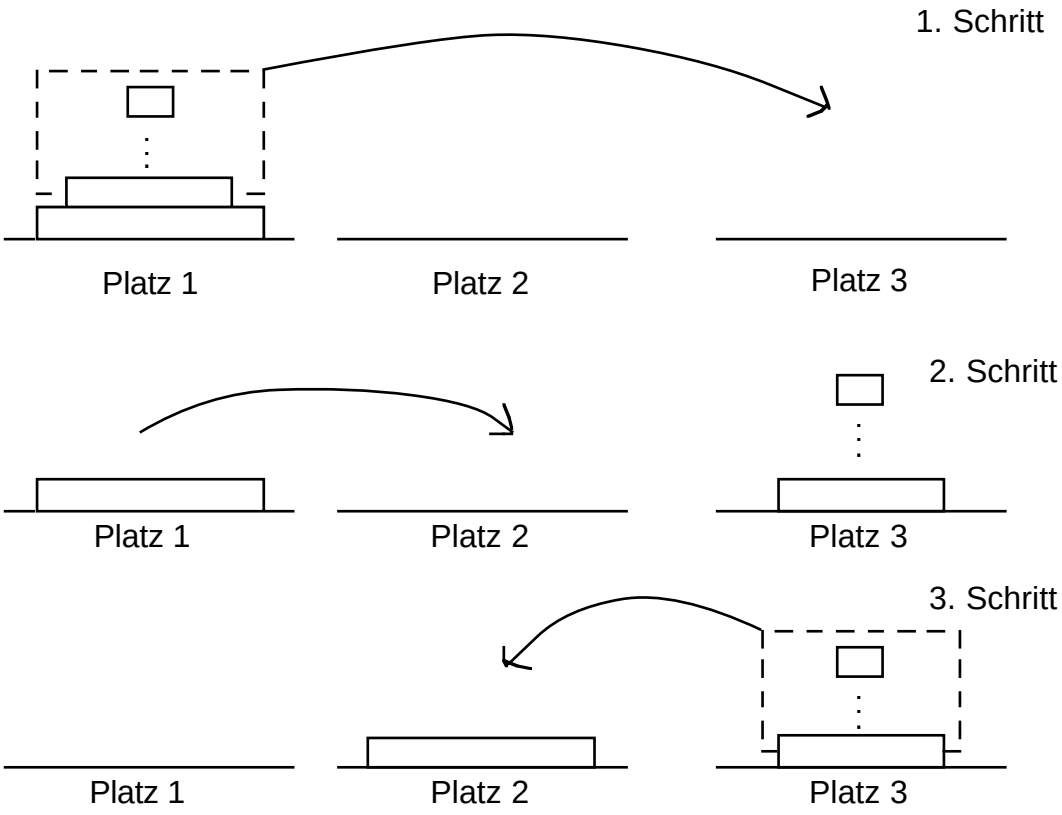


Abb. 6: Lösungsprinzip bei "Türme von Hanoi"

Wir haben also das Gesamtproblem

P = "Transportiere n Scheiben von Platz 1 nach Platz 2"

zurückgeführt auf die drei Teilprobleme

P_1 = "Transportiere n-1 Scheiben von Platz 1 nach Platz 3"

P_2 = "Transportiere eine Scheibe von Platz 1 nach Platz 2"

P_3 = "Transportiere n-1 Scheiben von Platz 3 nach Platz 2"

Die Probleme P, P_1 und P_3 unterscheiden sich offensichtlich nur in

- der Anzahl n der zu transportierenden Scheiben und
- dem Anfangsplatz a und dem Zielplatz z.

Diese Überlegung führt auf eine Funktion hanoi, die gerade diese Angaben als Parameter enthält. Nebenbei wird die Tatsache ausgenutzt, daß sich die Nummer des Hilfsplatzes durch den Ausdruck 6-a-z berechnen läßt. Sind z.B. n Scheiben von Platz 1 auf Platz 3 zu transportieren, so müssen zunächst die obersten n-1 von Platz 1 auf den Hilfsplatz 2=6-1-3 bewegt werden. Wir erhalten dann folgendes Programm:

```
typ platz ≡ {1,2,3};
typ zug  ≡ (platz,platz);
typ zuglist  ≡ leer | (zug,zuglist);
```

funktion hanoi n:nat a:platz z:platz → zuglist ≡
wenn n=0 dann leer sonst concat (hanoi (n-1) a (6-a-z)) ((a,z),hanoi (n-1) (6-a-z) z).

Das Türme-von-Hanoi-Problem demonstriert besonders die Kürze und Eleganz rekursiver Lösungsalgorithmen.

Die ersten vier Schritte der Formularmaschine für den Aufruf hanoi (3,1,2) zeigt Abb. 7.

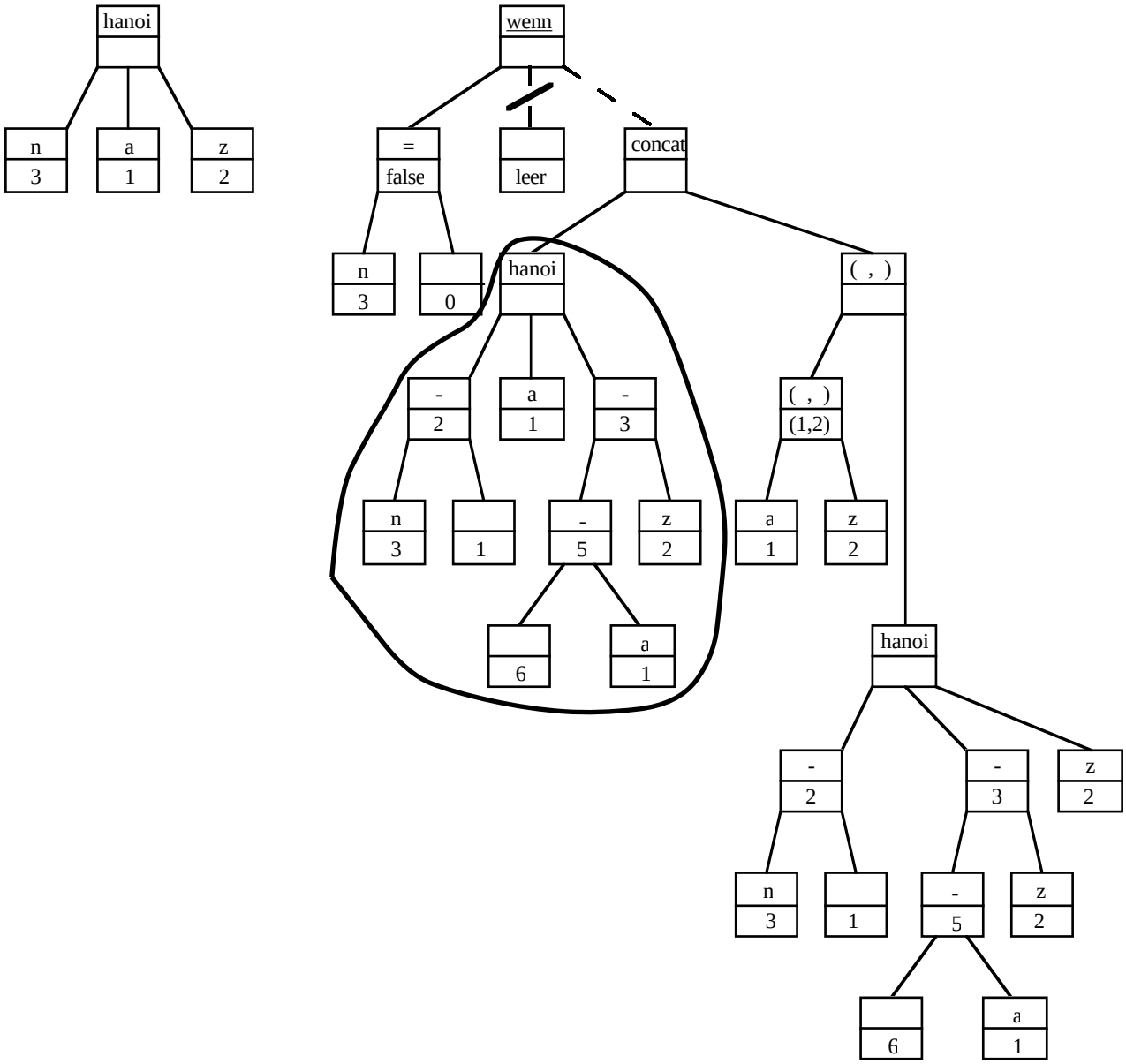


Abb. 7-1: Ausgangssituation der Formularmaschine für den Aufruf hanoi(3,1,2)

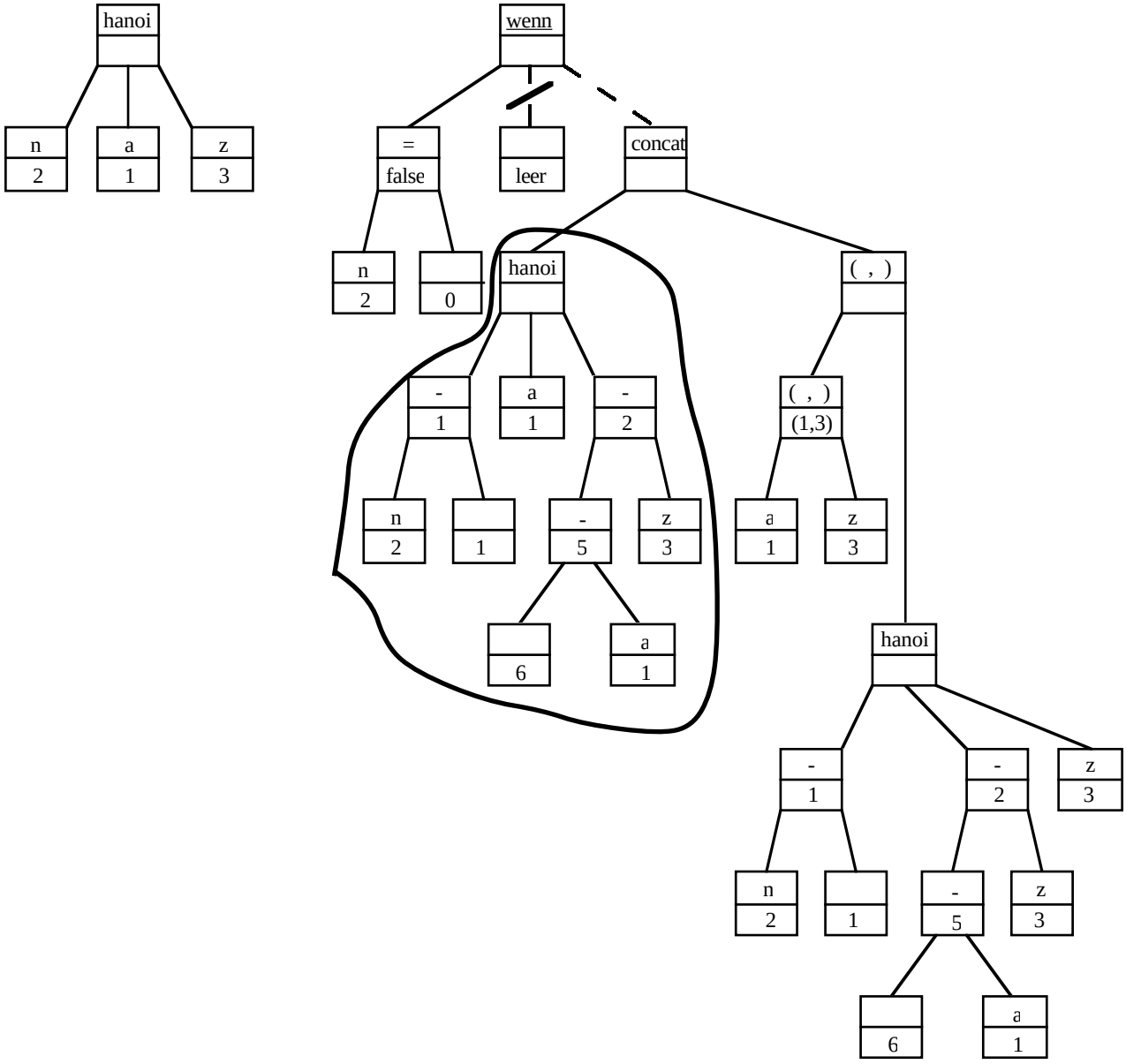


Abb. 7-2: Zweiter Schritt der Formularmaschine für den Aufruf hanoi(3,1,2)

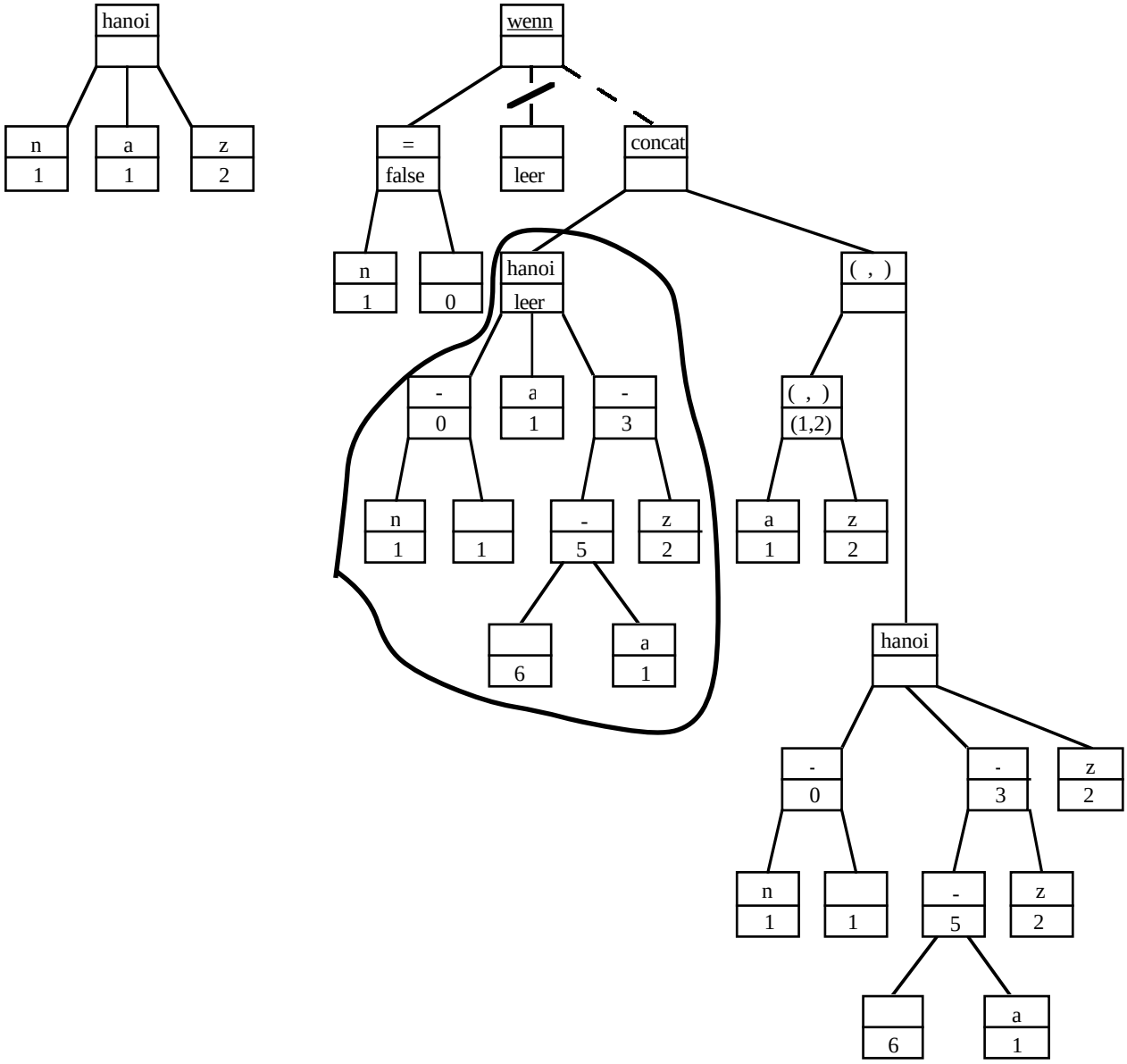


Abb. 7-3: Dritter Schritt der Formularmaschine für den Aufruf hanoi(3,1,2)

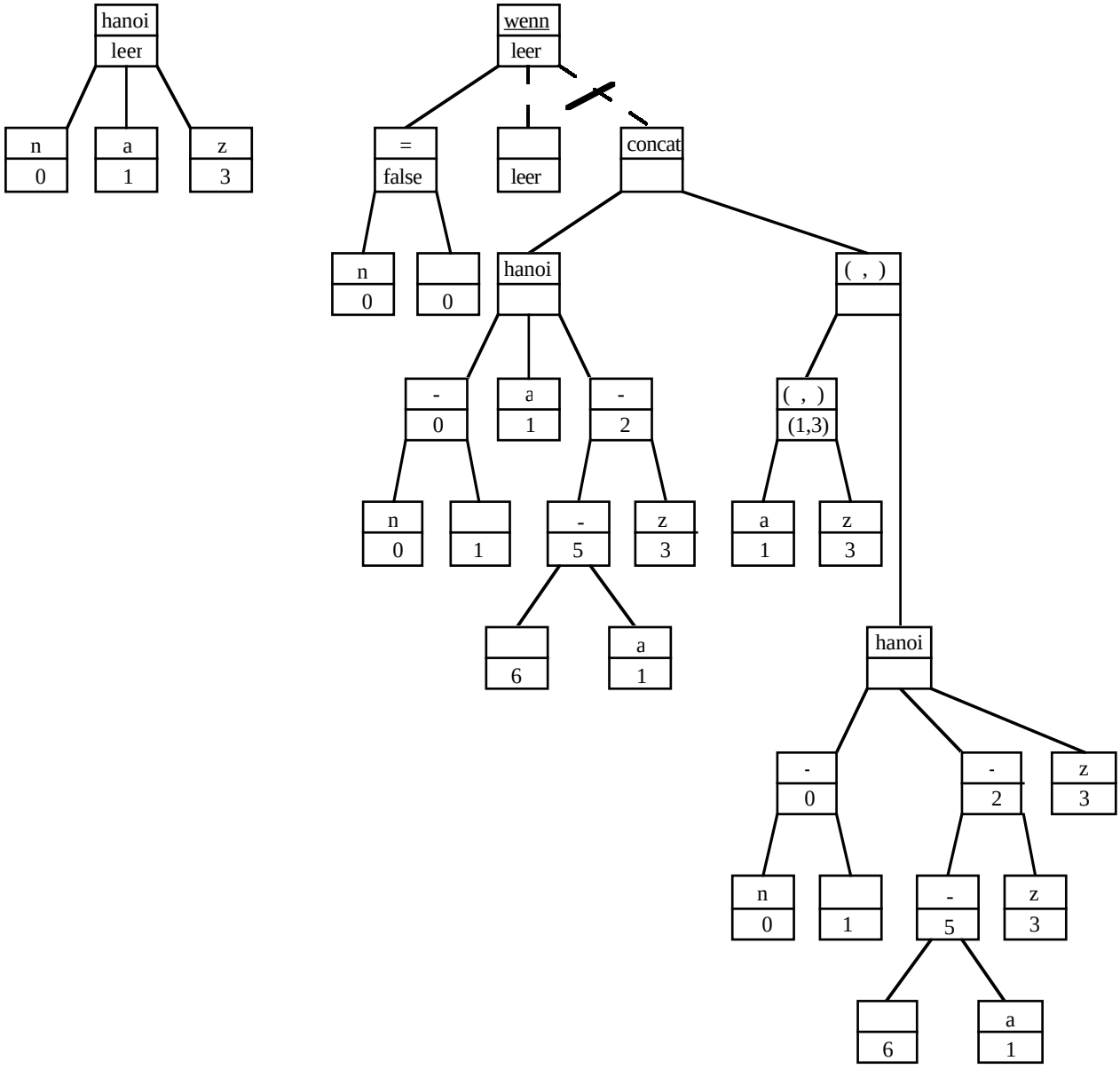


Abb. 7-4: Vierter Schritt der Formularmaschine für den Aufruf hanoi(3,1,2)

Sie haben nun eine Reihe von Beispielen dafür kennengelernt, wie man Probleme rekursiv lösen kann. Zu einigen Problemen kann man auch unmittelbar nicht-rekursive Lösungen angeben, z. B. für die Funktion gerade und ungerade von oben:

funktion gerade x:int → bool ≡ x mod 2=0.

funktion ungerade x:int → bool ≡ x mod 2=1.

Die Lösungen für gerade und ungerade sind in diesem Fall sogar kürzer als die rekursiven und sollten der Effizienz wegen bevorzugt werden.

Ein besonders extremes Beispiel ist die Fibonacci-Funktion, die gewisse Wachstumsprozesse beschreibt. Sie ist mathematisch definiert durch

$$f(n)=\begin{cases}1, & \text{falls } n=1 \text{ oder } n=2,\\ f(n-1)+f(n-2), & \text{sonst,}\end{cases}$$

oder informatisch durch

funktion f n:nat → nat ≡

wenn n=1 oder n=2 dann 1 sonst f(n-1)+f(n-2).

Hier die ersten Funktionswerte:

n	1	2	3	4	5	6	7	8
f(n)	1	1	2	3	5	8	13	21.

Die Auswertung dieser Funktion ist extrem zeitaufwendig. Ursache ist die doppelte Rekursion, die dazu führt, daß die einzelnen Terme, auf die sich die rekursiven Aufrufe abstützen, exponentiell häufig ausgewertet werden müssen. Eine Baumdarstellung der einzelnen Aufrufe für f(7) zeigt Abb. 8.

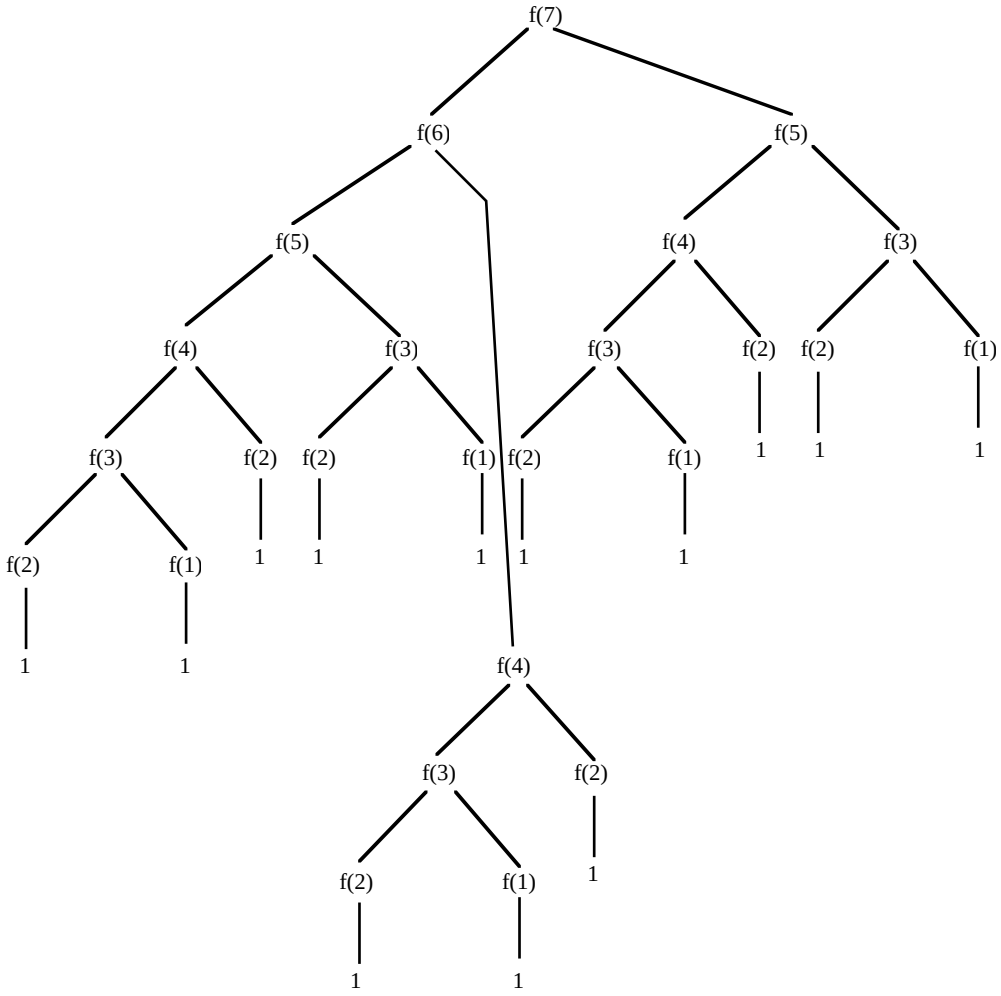


Abb. 8: Auswertung der Fibonacci-Funktion als Baum

10.7 Polymorphie

In der Praxis erscheint die strenge Typisierung manchmal als zu enges Konzept, um Probleme angemessen, flexibel und elegant zu lösen. Zur Motivation drei

Beispiele:

1) Oft benötigten wir im Programm Sequenzen unterschiedlicher Grundtypen. Statt nun alle Sequenzen einzeln zu definieren – wie wir es bisher immer getan haben –, legt man den (immer gleichen) Grundaufbau einer Sequenz einmalig typunabhängig fest. Benötigt man eine Sequenz eines bestimmten Typs, z.B. int, so "ruft man" die Grundstruktur mit dem vorgesehenen Typ int auf und deklariert damit zugleich eine Sequenz mit int-Elementen.
2) Zum Sortieren von Zahlen, Zeichen, Records usw. muß man in streng typisierten Programmiersprachen jeweils eine eigene Sortierfunktion schreiben. Tatsächlich unterscheiden sich die einzelnen Sortierfunktionen nur durch die Typen der zu sortierenden Objekte. Zweckmäßiger wäre es, nur eine Sortierfunktion für alle (sortierbaren) Typen zu schreiben und der Sortierfunktion beim Aufruf die zu sortierenden Objekte nebst ihrem Datentyp und ggf. die vorliegende Ordnungsrelation als Parameter zu übergeben.
3) Man betrachte die Identitätsfunktion

   funktion id x:D→D≡x.

   Sie ist eine sinnvolle Funktion auf *allen* vorstellbaren Datentypen D. Es scheint daher wenig zweckmäßig, für jeden vorkommenden Datentyp D eine eigene Definition anzugeben, wobei der Funktionsbezeichner noch jeweils unterschiedlich gewählt werden muß. Vielmehr sollte id auf allen Typen D des Universums definiert sein. Welche Identitätsfunktion man im konkreten Fall aufruft, ergibt sich entweder aus dem Typ des aktuellen Parameters oder dadurch, daß man beim Aufruf nicht nur den aktuellen Parameter, sondern auch seinen aktuellen Datentyp angibt; z.B. folgt aus dem Aufruf

   id(-7),

   daß D=int ist. Bei id(7) muß man jedoch angeben, ob 7∈nat oder 7∈int gemeint ist, z.B. durch den Aufruf

   id(7:nat).

Diese Lösungsansätze werden von dem Konzept der Polymorphie erfaßt. Polymorphie (griech.) bedeutet Vielgestaltigkeit; polymorphe Datentypen besitzen die Eigenschaft, mehrere Gestalten annehmen zu können. Zugrunde liegt die Idee, den Abstraktionskonstruktor von Funktionen auf Datentypen zu übertragen. Man erhält dann sog. *Typfunk-*

tionen mit formalen Parametern und einem Funktionsrumpf. Der Rumpf ist ein Typausdruck, der unter Verwendung der formalen Parameter, beliebiger anderer Datentypen oder Typfunktionen und der bekannten Typkonstruktoren gebildet wurde. Bei Aufruf der Typfunktion mit aktuellen Parametern (=irgendwelche konkreten Typen) wird der Rumpf ausgewertet und ein konkreter Typ generiert.
Bezeichnung: Kommen in einer Datentypdefinition formale Typparameter vor, so kennzeichnen wir diese durch große griechische Buchstaben (meist Δ).

Beispiel: Abb. 9 zeigt die Analogie zwischen der Abstraktion von Funktionen und der Abstraktion von Datentypen.

Funktionen		**Datentypen**
3+5	konkreter Ausdruck/konkreter Typ	(int,real)
↓	*Abstraktion*	↓
funktion add x:int y:int → int ≡ x+y.	Funktion <-> Typfunktion	typ paare Δ Δ' ≡ (Δ,Δ')
↓	*Applikation*	↓
add 4 12 => 16	konkreter Wert <-> konkreter Typ	paare int bool => (int,bool)

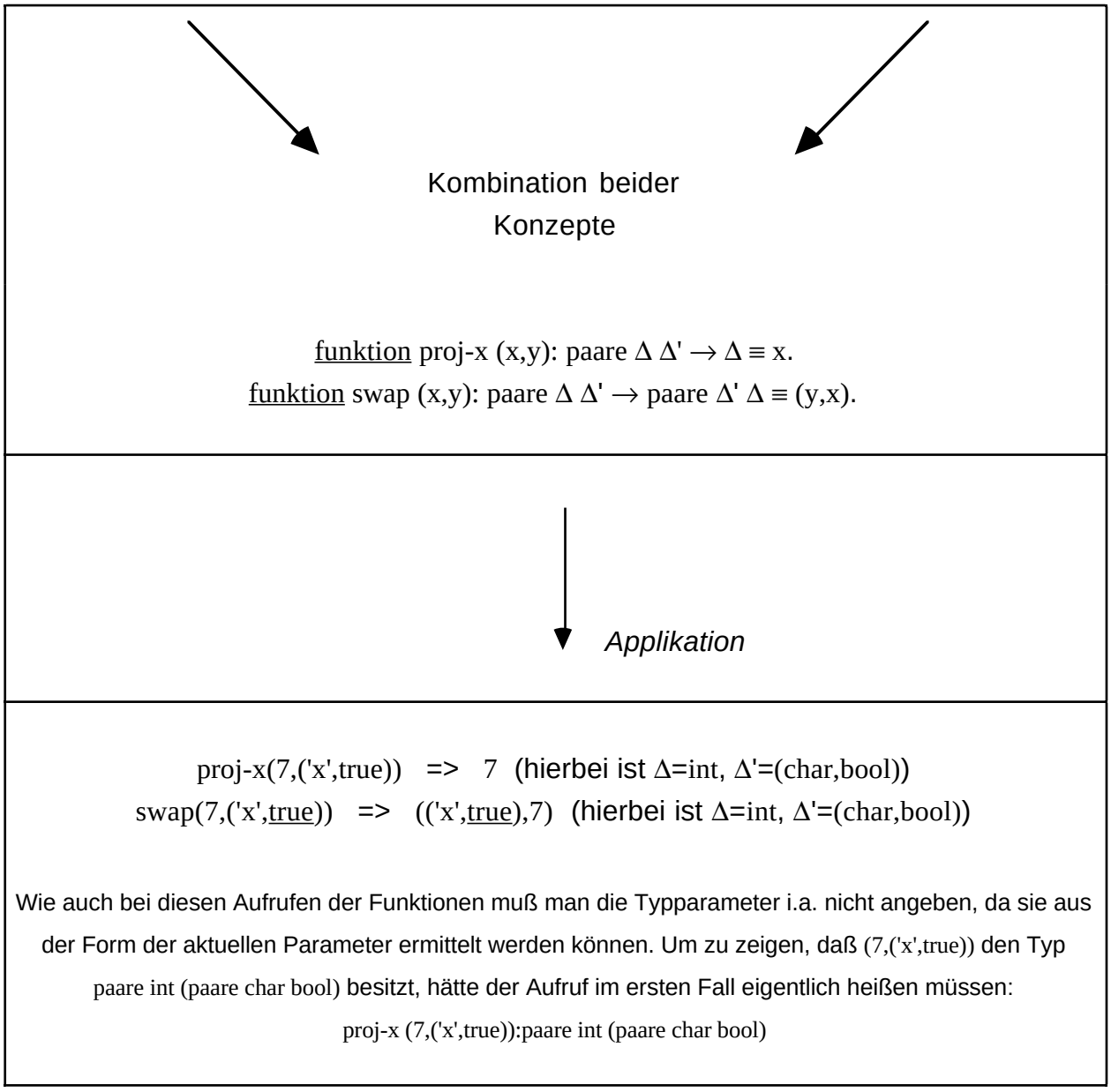


Abb. 9: Abstraktion bei Funktionen und Datentypen

Definition H:
Ein Datentyp heißt **polymorph**, wenn er einen Typparameter enthält. Eine Funktion heißt **polymorph**, wenn der Typ eines ihrer Argumente oder des Ergebnisses polymorph ist.

Polymorphe Typen definiert man analog zu Rechenvorschriften.

Definition I:

Eine polymorphe Typdefinition der Form

typ T Δ_1 ... Δ_n ≡ R.

heißt **Typfunktion**. Hierbei ist T der Bezeichner der Typfunktion, $\Delta_1,...,\Delta_n$ sind die formalen Typparameter, R ist der **Rumpf** der Typfunktion und ein Typausdruck, der induktiv wie folgt definiert ist:

elementare Bausteine:

(1) Jeder elementare Typ und jeder enumerierte Typ ist ein Typausdruck.

(2) Für i=1,...,n ist Δ_i ein Typausdruck.

Konstruktoren:

(3) Sind $A,B,A_1,...,A_n$ Typausdrücke, so auch die durch Potenzmengenbildung, Aggregation, Generalisation und Funktionenraumbildung gewonnen Ausdrücke

2^A, $(A_1,...,A_n)$, $A_1 \mid ... \mid A_n$ und $[A \rightarrow B]$.

(4) Sind $A_1,...,A_n$ Typausdrücke und U eine Typfunktion definiert durch

typ U Δ_1' ... Δ_n' ≡ R',

so ist auch (U A_1 ... A_n) ein Typausdruck.

(5) Ist A ein Typausdruck, so auch (A).

Man beachte, daß T in obiger Definition gecurryt ist; auch bei Typfunktionen kennen wir also diese von Funktionen bekannte Darstellung. Für einen Datentyp A ist der Ausdruck (T A) eine Typfunktion mit n-1 Parametern $\Delta_2,...,\Delta_n$, die mit dem Parameter A partiell ausgewertet ist.

Auf die präzise Semantik von Typausdrücken, die Art und Weise, sie auszuwerten, gehen wir nicht ein. Es genügt zu wissen, daß die Auswertung der bekannten mathematischen Auswertung von Ausdrücken unter Berücksichtigung von Prioritätsregeln und Klammern folgt .

Beispiele:

1) Den polymorphen Typ aller inhomogenen Paare D×D' für zwei beliebige Datentypen D und D' kennen wir schon:

typ paare Δ Δ' ≡ (Δ,Δ').

Hier wird eine Typfunktion paare definiert mit den Typparametern Δ und Δ' und Rumpf (Δ,Δ'). Diese Typfunktion verwendet man nun, um eine weitere Typfunktion zur homogenen Paarbildung zu definieren:

typ hpaare Δ ≡ paare Δ Δ.

Wendet man hpaare auf konkrete Datentypen an, so kann man z.B. folgende Datentypen generieren:

typ ip ≡ hpaare int; ip ist (int,int)

typ bp ≡ hpaare bool; bp ist (bool,bool)

typ ipaarvonpaar ≡ hpaare (hpaare int). ipaarvonpaar ist ((int,int),(int,int))

2) Die folgende Typfunktion liefert zu gegebenem Datentyp die Menge aller Abbildungen des Datentyps in sich (Automorphismen):

typ automorph Δ ≡ [Δ→Δ].

Zum Datentyp

automorph bool

gehört z.B. die Funktion nicht. Zum Datentyp

automorph (hpaare Δ)

gehört u.a. die Funktion swap mit

funktion swap (x,y): paare Δ Δ'→paare Δ' Δ ≡(y,x).

Weitere polymorphe Funktionen sind:

funktion id x:Δ→Δ ≡ x.

funktion projx (x,y): paare Δ Δ'→Δ ≡ x.

funktion projy (x,y): paare Δ Δ'→Δ' ≡ y.

funktion null x: Δ→nat ≡ 0.

3) Wir definieren den polymorphen Typ list, der uns zu jedem Datentyp D die Menge aller endlichen Linkssequenzen mit Werten aus D zur Verfügung stellt:

typ list Δ ≡ {leer} | (Δ,list Δ) .

list int beschreibt dann die schon bekannte Menge aller endlichen Linkssequenzen ganzer Zahlen. Die leere Liste ist durch das Symbol leer dargestellt.

Die bekannte (nun polymorphe) Funktion erstes besitzt dann die Funktionalität

funktion erstes f: list Δ→Δ ≡ ... .

10.8 Gleichheit auf Datentypen

Unter den polymorphen Funktionen, die in einer Programmiersprache üblicherweise zur Verfügung stehen, befinden sich auch die Funktionen zum Test auf Gleichheit und Ungleichheit zweier Objekte eines Datentyps, definiert durch

=: Δ×Δ→bool,

≠: Δ×Δ→bool.

Beide Funktionen bereiten unter gewissen Umständen erhebliche Probleme. Während die Gleichheit u.a. auf den elementaren Datentypen oder kartesischen Produkten dieser Typen wohldefiniert und effizient realisierbar ist, bekommt man Schwierigkeiten, wenn Δ ein Funktionstyp D→D' ist. Hier gibt es zunächst unterschiedliche Möglichkeiten zu definieren, wann zwei Funktionen f,g: Δ gleich sind, wann also

f=g bzw. f≠g gilt.

Definition J:
Zwei Funktionen f,g: $D \to D'$ heißen
- **intensional gleich**, wenn ihre Beschreibungen identisch sind;
- **extensional gleich**, wenn für alle x vom Typ D gilt:

$$f\ x = g\ x.$$

Beispiel: Die drei Funktionen

funktion double1 n:int→int ≡ 2n.
funktion double2 n:int→int ≡ n+n.
funktion double3 n:int→int ≡ 3n-n.

sind extensional gleich und intensional paarweise verschieden.

Bemerkung: Offenbar ist die intensionale Gleichheit effizient zu überprüfen, da dies praktisch auf einen Vergleich von Programmtexten hinausläuft. Andererseits gibt es keinen Algorithmus, der für zwei beliebige Funktionen entscheidet, ob sie extensional gleich sind, da das Problem nicht berechenbar ist.

In der Regel entscheidet man sich für die Interpretation der Gleichheit im extensionalen Sinne, da es nur auf die Werte von Ausdrücken ankommen soll, nicht jedoch darauf, wie diese berechnet werden. Folglich ist man gezwungen, das Problem der Nicht-Berechenbarkeit in den Griff zu bekommen. Zwei Lösungen bieten sich hierfür an:
1. Lösung: Man erlaubt den Gleichheitstest in voller Polymorphie

$$=: \Delta \times \Delta \to bool$$

und nimmt dann in Kauf, daß = auf Typen Δ, in denen Funktionen vorkommen, undefiniert ist oder zu Fehlern führt.
2. Lösung: Man schränkt die Polymorphie des Gleichheitstests ein und erlaubt den Test nur auf Typen, bei denen keine Berechenbarkeitsprobleme auftreten. Hierzu isoliert man aus dem Universum aller Typen die sog. *Gleichheitstypen* (*equality types*) und führt hierfür spezielle Typparameter ein, die man durch ein hochgestelltes Gleichheitszeichen markiert (z.B. $\Delta^=$). Damit ist = dann eine polymorphe Funktion nur auf Gleichheitstypen:

$$=: \Delta^= \times \Delta^= \to bool.$$

Gleichheitstypen definiert man induktiv durch die Vorschrift:
1) Die elementaren Datentypen int, bool, nat, real, char und text sind Gleichheitstypen.
2) Jede Gleichheitstypvariable $\Delta^=$ ist ein Gleichheitstyp.
3) Sind $\Delta, \Delta_1, ..., \Delta_n$ Gleichheitstypen, so auch deren Aggregation $(\Delta_1, ..., \Delta_n)$, deren Generalisation $\Delta_1 \mid ... \mid \Delta_n$ und die Potenzmenge 2^Δ.

Beispiele: Gleichheitstypen sind z.B.

$(bool,char)$,
$2^{(bool,char)}$
$int \mid char$,
$(int,(\Delta^{=},bool))$,

aber nicht

(Δ,int) oder
$[bool \to bool]$,

obwohl es in $[bool \to bool]$ nur *endlich* viele Funktionen gibt, zwischen denen man ggf. Gleichheit zu überprüfen hätte.

10.9 Typinferenz

In vielen Programmiersprachen muß man für alle Objekte bei ihrer Definition zugleich ihren Typ festlegen, so auch in unserer fiktiven funktionalen Programmiersprache. In moderneren Programmiersprachen ist dies nicht mehr erforderlich. Hier kann man nahezu alle Typangaben weglassen. Stattdessen ist der Übersetzer vermöge eines leistungsfähigen *Typinferenzsystems* fast immer in der Lage, aus den Operationen und der Darstellung ggf. beteiligter Konstanten die Typen aller beteiligten Objekte, der Ausdrücke und der Funktionen korrekt abzuleiten und die erforderlichen Typprüfungen durchzuführen. Voraussetzung ist eine strenge Typisierung.

Beispiel: Man betrachte den arithmetischen Ausdruck

$(x+1)*(y-z)$.

Offenbar ist 1 eine Konstante vom Typ int. Dann muß auch x vom Typ int sein, weil man mittels der polymorphen Operation $+$ nur entweder zwei $real$-Objekte oder zwei int-Objekte verknüpfen darf. Das gleiche gilt für die Operation $*$, so daß auch der Teilausdruck $(y-z)$ vom Typ int sein muß. Damit müssen dann auch y und z int-Objekte sein. Ohne eine einzige Typdefinition kann man also den Objekten x,y,z jeweils genau einen Typ zuordnen und überprüfen, ob die Objekte auch in anderen Zusammenhängen entsprechend verwendet wurden.

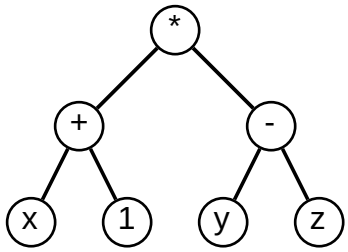


Abb. 10: Arithmetischer Ausdruck in Baumdarstellung

Häufig erfolgt solch eine Typinferenz bottom-up anhand der Baumdarstellung des Ausdrucks. Durch Bottom-up Analyse gewinnt man hier folgendermaßen den Typ des Ergebnisses: Man ergänzt den Baum an den Blättern um konkrete Typen, wenn sie für die jeweiligen Objekte bekannt sind, oder um Typvariablen, wenn sie nicht bekannt sind. Sukzessive schiebt man dann die Typen nach oben und eliminiert die Typvariablen. Ggf. sind hierzu mehrere Baumdurchläufe erforderlich (Abb. 11).

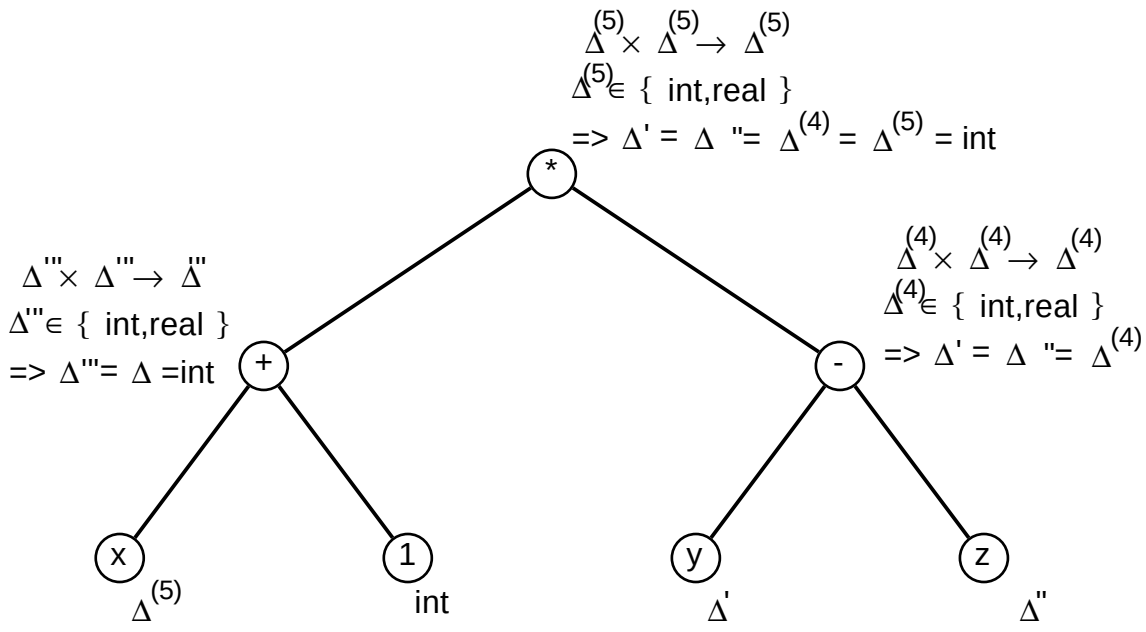


Abb. 11: Inferenz des Typs eines Ausdrucks

Ähnlich geht man auch vor, wenn polymorphe Funktionen beteiligt sind. Wir wollen hierfür zunächst einige einfache Typinferenzregeln definieren und das Verfahren sodann an einigen Beispielen erläutern.

Typinferenzregeln:

1) Regel für *Funktionsanwendung:* Gilt $(f\ x):\Delta$, dann setze $x:\Delta'$ und $f:\Delta'\to\Delta$ für einen neuen Typ Δ' ($E:\Delta$ bedeutet hier und im folgenden also: E besitzt den Typ Δ).
2) Regel für *Gleichsetzung:* Hat man $x:\Delta$ und $x:\Delta'$ abgeleitet, so setze $\Delta=\Delta'$. (Hier kommt die strenge Typisierung zum Zuge: Verschiedene Datentypen Δ und Δ' sind disjunkt: Gehört ein Objekt x zu Δ und Δ', so muß zwangsläufig $\Delta=\Delta'$ gelten!)
3) Regel für *Funktionalität:* Falls $\Delta\to\Delta'=\Delta''\to\Delta'''$ abgeleitet wurde, so setze $\Delta=\Delta''$ und $\Delta'=\Delta'''$.

Beispiele:

1) Gegeben sei die polymorphe Funktion

   function comp f:Δ g:Δ' x:Δ"→Δ"'≡f(g(x)).

   Wie lautet der Typ von comp? Aus Regel 1 folgt:

   $g(x)$: $\Delta^{(4)}$

   f: $\Delta^{(4)}\rightarrow\Delta'''$ für einen neuen Typ $\Delta^{(4)}$.

   Analog folgt:

   x: $\Delta^{(5)}$

   g: $\Delta^{(5)}\rightarrow\Delta^{(4)}$ für einen neuen Typ $\Delta^{(5)}$.

   Mit Regel 2 folgt dann aus dem Funktionskopf von comp:

   f: $\Delta=\Delta^{(4)}\rightarrow\Delta'''$

   g: $\Delta'=\Delta^{(5)}\rightarrow\Delta^{(4)}$

   x: $\Delta''=\Delta^{(5)}$.

   Folglich besitzt comp den Typ:

   $\Delta\rightarrow\Delta'\rightarrow\Delta''\rightarrow\Delta'''=(\Delta^{(4)}\rightarrow\Delta''')\rightarrow(\Delta^{(5)}\rightarrow\Delta^{(4)})\rightarrow\Delta^{(5)}\rightarrow\Delta'''$.

2) Man betrachte die Funktion:

   function g f:Δ→Δ'≡f(g(f)).

   Wie lautet der Typ von g? Aus Regel 1 folgt:

   $g(f)$: Δ''

   f: $\Delta''\rightarrow\Delta'$ für einen neuen Typ Δ''.

   Analog:

   f: Δ'''

   g: $\Delta'''\rightarrow\Delta^{(4)}$ für einen neuen Typ $\Delta^{(4)}$.

   Mit Regel 2 folgt dann aus dem Funktionskopf von g:

   f:$\Delta=\Delta'''=\Delta''\rightarrow\Delta'$ und

   g: $\Delta\rightarrow\Delta'=\Delta\rightarrow\Delta''=\Delta'''\rightarrow\Delta^{(4)}$,

   also nach Regel 3:

   $\Delta=\Delta'''$, $\Delta'=\Delta''=\Delta^{(4)}$.

   Dies liefert als Typ von g:

   $(\Delta'\rightarrow\Delta')\rightarrow\Delta'$.