1

Grundlagen der Funktionalen Programmierung

Durch das Einfache geht der Eingang zur Wahrheit.
Lichtenberg (Vermischte Schriften)

Wir wollen als Erstes einen ganz knappen Überblick über die fundamentalen Konzepte der Funktionalen Programmierung geben, nämlich Funktionen und Typen. Dazu gehört insbesondere ein kurzer Abriss zum Konzept der Funktionen höherer Ordnung, die ganz wesentlich zur Eleganz und zur Produktivität dieses Paradigmas beitragen. Dieses Kapitel dient vor allem der Auffrischung von elementaren Konzepten der Funktionalen Programmierung, wie sie in zahlreichen Lehrbüchern über ML, HASKELL, OPAL und andere Sprachen nachzulesen sind (z. B. [22, 50, 102, 138, 20, 81, 111, 32]).

1.1 Funktionen

Die Funktionale Programmierung basiert – nicht überraschend – auf dem Funktionsbegriff der Mathematik. Eine ***Funktion*** bildet Eingabewerte, d. h. Elemente aus dem *Definitionsbereich*, eindeutig auf Ausgabewerte, d. h. Elemente aus dem *Wertebereich*, ab.

In erster Näherung liefert das eine ganz simple Sicht auf funktionale Programme: Ein solches Programm besteht im Wesentlichen aus einer Sammlung von Funktionsdefinitionen, und ein Aufruf des Programms erfolgt durch die Applikation einer dieser Funktionen auf Eingabewerte.

Wir werden im Laufe dieses Buches natürlich noch sehen, dass es eine Fülle von Erweiterungen, Variationen und Spezialitäten gibt, aber in letzter Konsequenz wird sich zeigen, dass alles immer wieder auf diese elementare Sichtweise hinausläuft. Deshalb wollen wir im Folgenden diese elementaren Bausteine etwas genauer betrachten.

1.1.1 Funktionsdefinition

Ein funktionales ***Programm*** ist letztlich eine Menge von Funktionsdefinitionen. Für diese Definitionen gibt es eine Reihe von Schreibweisen, die wir im Folgenden kurz vergleichen wollen. Wir beginnen mit der heute beliebtesten Notation, die vor allem durch HASKELL populär gemacht wurde.

Definition (Funktionsdefinition)

Eine ***Funktionsdefinition*** erfolgt in der Form

DEF $f\ x_1 \ldots x_n = e$.

Hier wird eine Funktion mit Namen f eingeführt; die Namen $x_1, \ldots, x_n$ sind ***Parameter*** und der Ausdruck e wird als ***Rumpf*** bezeichnet. Ein oder mehrere der Parameter x_i können auch Tupel $(x_{i_1}, \ldots, x_{i_n})$ sein.

Als einfaches Beispiel einer solchen Funktionsdefinition betrachten wir die Berechnung von Dreiecksflächen, und zwar bei einem gleichseitigen Dreieck mit Seitenlänge a, bei einem rechtwinkligen Dreieck mit den Katheten a und b sowie bei einem allgemeinen Dreieck mit den Seiten a, b und c.

$$\begin{array}{lll}
\text{DEF } \mathit{area}\ a & = \frac{a^2}{4}\sqrt{3} & -- \textit{ gleichseitiges Dreieck} \\
\text{DEF } \mathit{area}\ a\ b & = \frac{a\ b}{2} & -- \textit{ rechtwinkliges Dreieck} \\
\text{DEF } \mathit{area}\ a\ b\ c & = \sqrt{s(s-a)(s-b)(s-c)} & -- \textit{ allgemeines Dreieck} \\
 & \quad \text{WHERE } s = \frac{a+b+c}{2} &
\end{array}$$

Diese Art der Darstellung ist ein Spezialfall der so genannten *musterbasierten Definition*, auf die wir in Abschnitt 1.1.7 gleich noch genauer eingehen werden. Außerdem sei schon jetzt darauf hingewiesen, dass diese Form sich auf subtile Weise von der Tupelform

$$\begin{array}{l}
\text{DEF } \mathit{area}(a,b) = \frac{a\ b}{2} \\
\text{DEF } \mathit{area}(a,b,c) = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{WHERE } s = \frac{a+b+c}{2}
\end{array}$$

unterscheidet. Mehr dazu in Abschnitt 1.1.5.

Man beachte auch, dass wir hier ganz im Sinne der syntaktischen Verabredungen des vorigen Kapitels den Rumpfausdruck nicht in altmodischer ASCII-Notation schreiben, sondern in eleganter mathematischer Notation.

Die Beispiele illustrieren auch das Prinzip des *Overloading*: Alle drei Funktionen heißen *area*; welche jeweils gemeint ist, lässt sich anhand der Parameterzahl erkennen.

Die obige Definition schließt auch die notationelle Variante der Mixfixoperatoren (s. Abschnitt 0.2) ein. Dabei ergibt sich höchstens das compilertechnische Problem, zu erkennen, was Funktionsname ist und was Parameter.

1.1.2 Ausdrücke

Das dritte der obigen *area*-Beispiele zeigt ein weiteres Sprachelement: Der Rumpfausdruck lässt sich durch *lokale Deklarationen* strukturieren, indem mittels WHERE-Klauseln abkürzende Namen für Teilausdrücke eingeführt werden. Man kann Deklarationen auch in Form von LET-IN-Klauseln voranstellen:

$$\begin{aligned}&\text{DEF } \mathit{area}\ a\ b\ c = \text{LET}\\ &\qquad\qquad s = \tfrac{a+b+c}{2}\\ &\qquad\text{IN}\\ &\qquad\qquad \sqrt{s(s-a)(s-b)(s-c)}\end{aligned}$$

Bei LET- und WHERE-Klauseln können mehrere Namen deklariert werden, die sich aufeinander beziehen dürfen. Die Regeln für die Reihenfolge der Deklarationen unterscheiden sich in den einzelnen Sprachen. Sie ist z. B. in OPAL beliebig, während in ML das Prinzip „Deklaration vor Verwendung“ eingehalten werden muss. OPAL verbietet zyklische Abhängigkeiten, während HASKELL sie erlaubt und ML dafür das Schlüsselwort `letrec` vorsieht.

Der Ausdruck im Rumpf kann neben der einfachen Komposition von Funktionen und Operatoren auch noch *Fallunterscheidungen* enthalten. Ein typisches Beispiel ist das Maximum zweier Zahlen:

$$\text{DEF } \mathit{max}\ a\ b = \text{IF } a \geq b \text{ THEN } a \text{ ELSE } b \text{ FI}$$

Durch Schachtelung solcher IF-THEN-ELSE-Ausdrücke lassen sich beliebig viele Fälle unterscheiden. Allerdings sieht z. B. OPAL eine Variante vor, die im Stil von E. W. Dijkstras *Guarded Commands* gleichberechtigte Fälle deutlicher als solche charakterisiert:

$$\begin{aligned}&\text{DEF } \mathit{max}\ a\ b = \text{IF } a \geq b \text{ THEN } a\\ &\qquad\qquad\qquad\text{IF } b \geq a \text{ THEN } b \text{ FI}\end{aligned}$$

Hier bleibt es dem Compiler überlassen, welchen der beiden Zweige er im Fall $a = b$ zur Berechnung auswählt. Die Eleganz dieser Variante zeigt sich deutlich, wenn wir z. B. das Maximum dreier Zahlen bestimmen wollen:

$$\begin{aligned}&\text{DEF } \mathit{max}\ a\ b\ c = \text{IF } a \geq b \wedge a \geq c \text{ THEN } a\\ &\qquad\qquad\qquad\qquad\text{IF } b \geq a \wedge b \geq c \text{ THEN } b\\ &\qquad\qquad\qquad\qquad\text{IF } c \geq a \wedge c \geq b \text{ THEN } c \text{ FI}\end{aligned}$$

1.1.3 λ-Notation

Funktionale Sprachen erlauben neben der oben gezeigten gleichungsartigen Funktionsdefinition, die an mathematische Gleichungen erinnert, üblicherweise auch die so genannte ***λ-Notation***. Der Ursprung dieser Notation[1] ist der λ-Kalkül [34, 17] von Church, der als theoretische Basis der funktionalen Sprachen betrachtet werden kann.

[1] Zur Entstehungsgeschichte des Zeichens „λ“ s. die Fußnote 2 auf Seite 8.

Definition (Funktionsdefinition in λ-Notation)

Eine ***Funktionsdefinition in λ-Notation*** erfolgt in der Form

DEF $f = \lambda x_1, \ldots, x_n \bullet e$

Durch das Fragment $\lambda x_1, \ldots, x_n$ werden die Variablen $x_1, \ldots, x_n$ im Ausdruck e gebunden. Den gesamten Ausdruck $\lambda x_1, \ldots, x_n \bullet e$ bezeichnet man als ***λ-Ausdruck*** oder auch als ***λ-Term***.

Die beiden Formen der Funktionsdefinition – musterbasiert und als λ-Term – sind völlig gleichwertig; sie unterscheiden sich bestenfalls als Geschmacksfrage.

Die obigen Beispiele der Dreiecksfläche (in der Tupelvariante) sehen in dieser Notation folgendermaßen aus (s. auch Abschnitt 1.1.5):

$$\begin{array}{ll}
\text{DEF } \mathit{area} = \lambda\, a \bullet \frac{a^2}{4}\sqrt{3} & -\!-\ \textit{gleichseitiges Dreieck} \\
\text{DEF } \mathit{area} = \lambda\, a, b \bullet \frac{a\ b}{2} & -\!-\ \textit{rechtwinkliges Dreieck} \\
\text{DEF } \mathit{area} = \lambda\, a, b, c \bullet \sqrt{s(s-a)(s-b)(s-c)} & -\!-\ \textit{allgemeines Dreieck} \\
\qquad \text{WHERE } s = \frac{a+b+c}{2} &
\end{array}$$

Bei der Funktionsdefinition stellt die λ-Notation nur eine alternative Schreibweise dar. Aber sie hat einen anderen wichtigen Vorteil: Mit ihr kann man Funktionen auch direkt aufschreiben und verwenden, ohne ihnen dazu einen expliziten Namen geben zu müssen. Solche *anonymen Funktionen* können z. B. dann genutzt werden, wenn man mit Funktionen als Argumenten arbeitet. Ein typisches Beispiel ist die Funktion *filter*, die wir in Abschnitt 1.2 gleich noch genauer betrachten werden:

$\ldots \mathit{filter}\ (\lambda\, x \bullet x < 0\,)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots$

Diese Funktion behält nur diejenigen Elemente einer Liste, die das Prädikat $(\lambda\, x \bullet x < 0\,)$ erfüllen, d. h. alle negativen Zahlen. Im Beispiel der obigen Liste $\langle 3, 5, -2, 0, 18, -12\rangle$ ist das Ergebnis also die Liste $\langle -2, -12\rangle$.

Als Kurzform verwendet z. B. die Sprache OPAL auch die ***Wildcard-Notation***, also $(\_\ \hat{}\ 2)$ als Kurzform für den λ-Term $(\lambda\, x \bullet x\ \hat{}\ 2)$. In anderen Sprachen wird das über so genannte *Sections* formuliert, d. h. in der Form $(\hat{}\ 2)$. Wir werden beide Versionen verwenden. Denn insbesondere das letztere Beispiel entsteht auch durch die Konvention von Kapitel 0, dass man den Platzhalter „ $\_$ “ am Anfang und Ende eines Mixfix-Symbols weglassen darf. Die obige Anwendung der Funktion *filter* kann also in den folgenden drei gleichwertigen Formen geschrieben werden:

$\ldots \mathit{filter}\ (\lambda\, x \bullet x < 0)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots$
$\ldots \mathit{filter}\ (\_ < 0)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots$
$\ldots \mathit{filter}\ (< 0)\ \langle 3, 5, -2, 0, 18, -12\rangle \ldots$

Man beachte, dass diese Kurzform mittels „ $\_$ “ nur bei Funktionen mit einem Parameter anwendbar ist; diese treten aber besonders häufig auf.

1.1.4 Funktionalitäten: Die Typisierung von Funktionen

Typen beschreiben die Funktionalität von Funktionen (und Ausdrücken), d. h. die Art (und Anzahl) der Eingabe- und Ausgabewerte.

Definition (Typ, Funktionalität)

Der ***Typ*** einer Funktion (ihre ***Funktionalität***) wird im Allgemeinen folgendermaßen geschrieben:

FUN f: $A \to B$ −− *Funktion*
VAL c: B −− *Konstante*

- Der Definitionsbereich der Funktion f ist A und der Wertebereich ist B. Wie in der Mathematik üblich werden beide durch einen Pfeil getrennt.
- Als Grenzfall von Funktionen haben wir auch *Konstanten*. Hier ist der Definitionsbereich leer. Aus Gründen der Lesbarkeit lassen wir den Pfeil weg und geben nur den Wertebereich B an. Als Schlüsselwort schreiben wir dann manchmal auch VAL anstelle von FUN.
- Definitions- und Wertebereich brauchen nicht nur aus elementaren Typen zu bestehen, sondern können auch Typausdrücke[2] sein, die aus *Produkttypen* $(A_1 \times \ldots \times A_n)$ und *Funktionstypen* $(A \to B)$ aufgebaut sind. Im letzteren Fall sprechen wir von *Funktionen höherer Ordnung*. Für diese potenziell sehr komplexen Typausdrücke gelten folgende Klammerregeln:
  - Die Tupelbildung „$\times$“ bindet stärker als die Funktionsbildung „ $\to$ “.
  - Die Funktionsbildung „ $\to$ “ bindet nach rechts, d. h., $A \to B \to C$ ist gleichbedeutend zu $A \to (B \to C)$.
- ***Sorten*** bilden die Basis für die Typbeschreibung. Das sind die Namen für elementare Mengen (wie *Nat*, *Int*, *Real*, *Char*, *String* usw.)

Betrachten wir als Beispiel wieder die Berechnung des Flächeninhalts von Dreiecken. Wir geben zu den Definitionen die jeweiligen Funktionalitäten mit an. In der Tupelvariante ergibt sich:

FUN $area$: $Real \to Real$ −− *gleichseitiges Dreieck*
DEF $area\ a = \frac{a^2}{4}\sqrt{3}$

FUN $area$: $Real \times Real \to Real$ −− *rechtwinkliges Dreieck*
DEF $area(a, b) = \frac{a \cdot b}{2}$

FUN $area$: $Real \times Real \times Real \to Real$ −− *allgemeines Dreieck*
DEF $area(a, b, c) = \sqrt{s(s-a)(s-b)(s-c)}$
WHERE $s = \frac{a+b+c}{2}$

Man kann das Gleiche auch in der Form von Funktionen höherer Ordnung schreiben (s. Abschnitt 1.1.5):

[2] Eine detaillierte Diskussion von Typen und Typausdrücken werden wir in den Kapiteln 6-9 führen.

FUN *area*: $Real \to Real \to Real$ −− *rechtwinkliges Dreieck*
DEF $area\ a\ b = \frac{a\ b}{2}$

FUN *area*: $Real \to Real \to Real \to Real$ −− *allgemeines Dreieck*
DEF $area\ a\ b\ c = \sqrt{s(s-a)(s-b)(s-c)}$
WHERE $s = \frac{a+b+c}{2}$

Übrigens: Die Funktionalitäten zeigen das *Overloading* besonders deutlich.

1.1.5 Partielle Applikation und Currying

Funktionale Sprachen erlauben es, Funktionen auf weniger Argumente anzuwenden als nötig. Man nennt dies eine ***partielle Applikation***. Durch die partielle Applikation einer Funktion erhält man eine neue Funktion, die bei Anwendung auf die restlichen Argumente das gleiche Ergebnis wie die ursprüngliche vollständig applizierte Funktion liefert.

Um die partielle Applikation einer Funktion zu erlauben, müssen wir bei der Definition der Funktionalität anstelle des Produkts „ × “ den Funktionspfeil „ → “ schreiben. Diesen Übergang von der Tupelbildung zum Funktionspfeil nennt man ***Currying***, die dadurch entstehende neue Notation ***Curry-Notation***. Das haben wir in den obigen Beispielen der *area*-Funktionen gerade gesehen:

FUN *area*: $Real \times Real \to Real$ −− *Tupelversion*
DEF $area(a, b) = \frac{a\ b}{2}$

FUN *area*: $Real \to Real \to Real$ −− *Curry-Version*
DEF $area\ a\ b = \frac{a\ b}{2}$

Generell wird aus dem n-stelligen Tupeltyp $T_1 \times T_2 \times \ldots \times T_n \to T$ die Curry-Form $T_1 \to (\, T_2 \to \ldots \to (T_n \to T) \ldots)$. Mathematisch sind beide Typen isomorph.

Wenn wir die Curry-Version von *area* partiell applizieren, z. B. *area* 1.0, erhalten wir eine Funktion, die bei Eingabe der zweiten Kathete den Flächeninhalt des rechtwinkligen Dreiecks berechnet. Das kann sogar in eine entsprechende Definition gefasst werden:

FUN *partialArea*: $Real \to Real$ −− *braucht nur noch die zweite Kathete*
DEF $partialArea = area\ 1.0$

Die Funktion *partialArea* erlaubt also die Flächenberechnung für rechtwinklige Dreiecke, bei denen eine Kathete die vorgegebene Länge 1.0 hat. Die Definition könnte übrigens auch in der Form

DEF $partialArea\ b = area\ 1.0\ b$

oder in λ-Form

DEF $partialArea = \lambda\, b \bullet area\ 1.0\ b$

geschrieben werden. (Im λ-Kalkül spricht man hier von η-Reduktion.)

1.1.6 Typen

Das Thema *Typisierung* ist ein besonders aktives Forschungsfeld bei den modernen Programmiersprachen. Deshalb diskutieren wir es intensiv in den Kapiteln 6 bis 9. Für diesen einführenden Abschnitt beschränken wir uns auf ein eher intuitives Verständnis der elementarsten Konstruktionen.

Produkttypen (Tupeltypen)

Mit Produkttypen (auch *Tupeltypen* genannt) kann man logisch zusammengehörende Daten gleichen oder verschiedenen Typs in einem neuen Datentyp zusammenfassen.

Beispiel: Ein Punkt im zweidimensionalen Raum ist durch seine Koordinaten x und y charakterisiert. Wir definieren den Tupeltyp *Point*:

TYPE $Point = Real \times Real$ $--$ *ohne Konstruktoren und Selektoren*

Will man auf die einzelnen Elemente des Tupeltyps direkt zugreifen, kann man bei der Typdefinition auch Selektoren angeben. (Die Verwendung des Gleichheitszeichens anstelle des Doppelpunkts, der hier in den meisten Sprachen steht, wird in Kapitel 6 eingehend erläutert werden.)

TYPE $Point = (x = Real, y = Real)$ $--$ ... *mit Selektoren ...*

Dadurch erhalten wir die *Selektionsfunktionen* x und y, die wir auf einen Punkt p anwenden können. Mit $x(p)$ greifen wir dann beispielsweise auf die erste Komponente des Punktes p zu. (Wie wir gleich noch in Abschnitt 1.2 sehen werden, können wir das auch als $p.x$ schreiben – was eher an Notationen aus Sprachen wie JAVA erinnert.)

Schließlich können wir auch noch eine *Konstruktorfunktion*, hier *point*, angeben; in diesem Fall sprechen wir von ***Konstruktortypen***:

TYPE $Point = point(x = Real, y = Real)$ $--$ ... *sowie mit Konstruktor*

Das ist besonders im Zusammenhang mit dem so genannten *Patternmatching* nützlich, wie wir gleich noch sehen werden.

Summentypen

Neben Produkttypen braucht man auch ***Summentypen***, um inhaltlich verwandte Elemente, die aber strukturell unterschiedlich aufgebaut sein können, zusammenzufassen.

Betrachten wir neben Punkten weitere geometrische Elemente, die wir als entsprechende Typen definiert haben, dann können wir diese in einem Typ *Shape* zusammenfassen:

TYPE $Shape = Point \mid Line \mid Circle \mid Triangle \mid Rectangle$

Dann können wir immer da, wo ein Wert vom Typ *Shape* erwartet wird, Werte der Typen *Point*, *Line* etc. benutzen (Näheres in Abschnitt 6.5).

Ein besonders häufiger Spezialfall ist die Bildung von Summentypen, deren Varianten direkt als Konstruktortypen angegeben sind. Zum Beispiel können wir für einen Punkt zwei Darstellungen vorsehen:

TYPE *Point* = *koord*($x = Real$, $y = Real$)
| *polar*(*dist* = *Real*, *angle* = *Real*)

Die Konstruktoren können sogar mittels Overloading gleich benannt werden, wie das folgende Beispiel illustriert.

TYPE *Line* = *line*($p_1 = Point$, $p_2 = Point$)
| *line*($p = Point$, $angle = Real$, $length = Real$)

Rekursive Typen

Häufig werden Produkt- und Summentypen kombiniert, insbesondere dann, wenn die Definition der Typen ***rekursiv*** ist.

Beispiel: Eine Sequenz reeller Zahlen ist entweder leer, hier dargestellt durch $\Diamond$, oder sie besteht aus einer reellen Zahl, die mit einer weiteren Sequenz verkettet ist:

TYPE *RealSeq* = { $\Diamond$ }
| *prepend*(*ft* = *Real*, *rt* = *RealSeq*)

Diese Definition kombiniert einen Summentyp mit einem rekursiven Produkttyp. Den Konstruktor schreiben wir übrigens meistens als Infixoperator:

TYPE *RealSeq* = { $\Diamond$ }
| _ $\cdot\!:$ _(*ft* = *Real*, *rt* = *RealSeq*)

1.1.7 Musterbasierte Funktionsdefinition

Basierend auf der Definition von Datentypen mit Konstruktoren kann man in einfacher Weise so genannte ***musterbasierte Funktionsdefinitionen*** (engl.: *pattern-based definitions*) aufschreiben. Dies ist im Allgemeinen kürzer und eleganter als eine Definition, die auf einer Folge von IF-Abfragen basiert.

Beispiel: Die Funktion *length* berechnet die Länge einer Sequenz, indem sie diese durchläuft und die Elemente zählt.

FUN *length*: *Seq* $\rightarrow$ *Nat*
DEF *length* $\Diamond = 0$
DEF *length*(*prepend*(*first*, *rest*)) = 1 + *length*(*rest*)

Bei einer musterbasierten Funktionsdefinition DEF $f(c(x_1, \ldots, x_n), \ldots) = e$ sind die Argumente der linken Seite Konstruktorterme $c(x_1, \ldots, x_n)$. Man nennt sie auch *Muster* oder *Pattern*. Für eine Funktion gibt man dabei (in der Regel für ein Argument) so viele DEF-Deklarationen an wie der zugehörige Summentyp Varianten aufweist. Die Variablen $x_1, \ldots, x_n$ können dann im Rumpf so benutzt werden wie bisher die Variablen der linken Seite bzw. wie die durch ein λ-Konstrukt gebundenen Variablen.

Bei der Auswertung wird der Ausdruck dann mit jedem der Pattern der entsprechenden Funktion verglichen und die Regel mit passendem Muster ausgewählt. Dieses Vorgehen nennt man ***Patternmatching***.

Das Konzept der musterbasierten Definitionen überträgt sich in analoger Weise auch auf Infixoperatoren:

DEF $length\ \Diamond = 0$
DEF $length(first\ \cdot{:}\ rest) = 1 + length(rest)$

Wenn man das Prinzip des *best-fit* Patternmatching verwendet, kann man auch folgende Art von Definitionen schreiben:

FUN $fac{:}\ Nat \to Nat$
DEF $fac\ 0 = 1$
DEF $fac\ n = n \cdot fac(n-1)$

Hier würde das zweite Pattern – das ja nur eine Variable n ist – im Prinzip immer passen. Aber das Pattern 0 – ein konstanter Konstruktor – ist spezifischer und hat daher Vorrang. (In diesem Beispiel würde auch das so genannte *first-fit* Patternmatching funktionieren, das aber den generellen Nachteil hat, dass die Aufschreibungsreihenfolge der Definitionen relevant wird.)

1.1.8 Erweitertes Patternmatching

Die Eleganz des Programmierens lässt sich noch steigern, wenn man das Prinzip des Patternmatchings so verallgemeinert, dass nicht nur Konstruktorfunktionen zugelassen werden.

Beispiel: Die folgende – ziemlich artifizielle – Funktion wurde von Dijkstra benutzt, um Beweistechniken zu illustrieren.

FUN $fusc{:}\ Nat \to Nat$
DEF $fusc\ 0 = 1$
DEF $fusc\ 1 = 1$
DEF $fusc(2 \cdot n) = fusc\ n$
DEF $fusc(2 \cdot n + 1) = fusc(n) + fusc(n+1)$

Das ist im Grunde nicht korrekt, da „1“, „ $+$ “ und „ $\cdot$ “ keine Konstruktoren sind. Deshalb müssen wir in die Sprache Mechanismen aufnehmen, die ein solches erweitertes Patternmatching ermöglichen. Dazu verwenden wir das spezielle Schlüsselwort MATCHES.

Im obigen Beispiel müssen für einen Aufruf $fusc(x)$ vom Compiler die vier Fälle erkennbar sein: (x MATCHES 0), (x MATCHES 1), (x MATCHES $2 \cdot n$) und (x MATCHES $2 \cdot n + 1$). In den letzten beiden Fällen muss dabei noch die freie Variable n an eine entsprechende Zahl gebunden werden. Die technischen Details lassen wir hier offen, da sie eher in Bereiche wie Compilertechnik oder Constraint-Solving führen. Wir beschränken uns auf eine Skizze der notwendigen Mechanismen.

- Man braucht ein *Prädikat*, das den Erfolg des Matchings anzeigt. Im obigen Beispiel ist das

  x MATCHES $2 \cdot n \iff even\ x$
  x MATCHES $2 \cdot n + 1 \iff odd\ x$

- Außerdem braucht man Funktionen, die den freien Variablen im Muster ihre Instanzwerte zuordnen:

  x MATCHES $2 \cdot n \implies n = \frac{x}{2}$
  x MATCHES $2 \cdot n + 1 \implies n = \frac{x-1}{2}$

Die Prädikate werden vom Compiler benutzt, um die musterbasierte Definition in eine IF-Kaskade umzuwandeln, und die Instanziierungsfunktionen führen dann in entsprechenden LET-Anweisungen die Patternvariablen ein.

Man beachte: Was wir hier für beliebige Muster sehr länglich aufschreiben müssen, wird vom Mechanismus der Summen- und Produkttypen automatisch geleistet, da der Compiler hier die entsprechenden Test- und Selektorfunktionen intern generiert.

1.1.9 Polymorphie

Kann man eine Funktion für Elemente unterschiedlichen Typs in gleicher Weise und unabhängig von ihrem Typ definieren, so spricht man von ***(parametrischer) Polymorphie***. Wir werden dieses Konzept in Kapitel 5 und Kapitel 8 ausführlich diskutieren. Für unsere einführenden Beispiele brauchen wir aber zumindest ein erstes intuitives Verständnis der Grundidee.

Die einfachste polymorphe Funktion ist die Identitätsfunktion:

FUN $id\colon \alpha \to \alpha$
DEF $id\ x = x$

Definition (polymorphe Funktion, polymorpher Datentyp, Typvariable)

Ein Datentyp ist ***polymorph***, wenn seine Definition von einem Typparameter abhängt. Der Typparameter wird auch als ***Typvariable*** bezeichnet, für die wir im Folgenden meist griechische Buchstaben $\alpha, \beta, \gamma, \ldots$ verwenden.

Eine Funktion heißt ***polymorph***, wenn ihre Funktionalität einen polymorphen Typ enthält.

Ein klassisches Beispiel sind Sequenzen über Elementen eines beliebigen Typs. Sie werden als polymorpher Datentyp definiert:

TYPE $Seq\ \alpha\ =\ \{\ \Diamond\ \}$
$\quad\quad\quad\quad\quad\quad\ \mid\ \ \_ \cdot\colon \_(ft = \alpha,\ rt = Seq\ \alpha)$

Auf solchen Sequenzen kann man polymorphe Funktionen definieren, die völlig unabhängig vom Typ ihrer Elemente sind. Ein typisches Beispiel ist die früher schon erwähnte Funktion *length*:

FUN $length$: $Seq\ \alpha \to Nat$
DEF $length(\Diamond) = 0$
DEF $length(first \cdot: rest) = 1 + length(rest)$

In Abschnitt 1.2 werden uns typische polymorphe Funktionen auf Sequenzen als Funktionen höherer Ordnung begegnen.

Anmerkung: Wie schon in Abschnitt 0.5 angesprochen, kann die Erkennung der Typvariablen unter Umständen Probleme machen. Diese Probleme werden wir in Kapitel 8 noch ausführlich diskutieren. Vorläufig umgehen wir das Problem, indem wir Typvariablen als griechische Buchstaben schreiben.

1.1.10 Eigenschaften (Propertys)

Bisher haben wir bei Funktionen die beiden klassischen Sprachkonstrukte skizziert: ihre *Definition* und ihre *Typisierung*. Und die meisten Sprachen beschränken sich auch auf diese beiden Features. Aber in vielen Fällen bräuchte man noch eine weitere Möglichkeit: Man möchte ausdrücken, dass Funktionen bestimmte Eigenschaften haben. Wir führen dazu das Sprachmittel der ***Propertys*** ein.

Beispiel: Die Sinus- und die Kosinusfunktion werden numerisch über geeignete Reihendarstellungen berechnet. Aber sie haben auch diverse mathematische Beziehungen zueinander:

DEF $sin(x) = \ldots$ «*Reihenentwicklung*» $\ldots$
DEF $cos(x) = \ldots$ «*Reihenentwicklung*» $\ldots$
PROP $cos(x) = sin(x + \frac{\pi}{2})$

Während diese Property für einen Compiler nicht viel nützen wird, gibt es zahlreiche Fälle, in denen das anders ist. Wichtige Eigenschaften von Funktionen sind z. B. die Assoziativität oder Kommutativität, die Existenz einer inversen Funktion oder eines neutralen Elements, die Idempotenz usw.

Manchmal ist es auch nützlich, die Propertys zu benennen. Dann kann man sich z. B. in Programmbeweisen oder -entwicklungen explizit auf sie beziehen:

PROP $cosToSin$ IS $cos(x) = sin(x + \frac{\pi}{2})$

Übrigens: Die Typisierung einer Funktion ist natürlich nichts anderes als eine spezielle Property. Mit anderen Worten,

FUN sin: $Real \to Real$

ist eigentlich nur eine syntaktische Variante folgender Property:

PROP sin: $Real \to Real$

Wir werden von den Propertys sehr starken Gebrauch machen, wenn wir in Kapitel 9 mit Spezifikationen und Typklassen arbeiten. In Kapitel 16 werden wir sie zur Entwicklung von Algorithmen heranziehen.

1.1.11 Sequenzen (Listen, Folgen)

Programmierparadigmen haben oft bestimmte Datenstrukturen, auf die sie besonders zugeschnitten sind. Bei imperativen Sprachen wie ALGOL, PASCAL, C oder JAVA – und ganz besonders bei FORTRAN – ist das der Array. Bei funktionalen Sprachen ist es die ***Liste***, oft auch ***Sequenz*** oder ***Folge*** genannt. Weil auch wir diese Struktur oft in Beispielen verwenden, geben wir hier kurz ihre wichtigsten Aspekte an.[3]

```
TYPE Empty = { ◊ }
TYPE Seq α  =  Empty
             |  _ ·: _(ft = α, rt = Seq α)
```

Sequenzen sind üblicherweise polymorph definiert mit den Konstruktoren „$\Diamond$“ (*empty*) und „ $\cdot\!:$ “ (*prepend*). Den Hilfstyp *Empty* führen wir der besseren Lesbarkeit halber ein.

Die Längenberechnung haben wir weiter oben schon als Beispiel programmiert. Interessant sind aber auch die Operationen „am falschen Ende“, also *front* und *last*, sowie „ $:\!\cdot$ “ (*append*).

```
FUN front: Seq α → Seq α                 -- Vorderteil der Sequenz
FUN last: Seq α → α                      -- letztes Element
FUN _ :· _: Seq α × α → Seq α            -- append

DEF front(x ·: ◊) = ◊
DEF front(x ·: rest) = x ·: front(rest)
DEF last(x ·: ◊) = x
DEF last(x ·: rest) = last(rest)
DEF ◊ :· x = x ·: ◊
DEF (a ·: rest) :· x = a ·: (rest :· x)
```

Man beachte, dass diese Definitionen wesentlich Gebrauch vom best-fit Patternmatching machen. Außerdem sind *front* und *last* für die leere Sequenz natürlich nicht definiert.

Weitere nützliche Funktionen sind die Konkatenation sowie eine schöne Mixfixnotation für die einelementige Sequenz:

```
FUN _ ++ _: Seq α × Seq α → Seq α        -- Konkatenation
FUN ⟨_⟩: α → Seq α                       -- einelementige Sequenz

DEF ◊ ++ s = s
DEF (a ·: rest) ++ s = a ·: (rest ++ s)
DEF ⟨x⟩ = x ·: ◊
```

Wir werden uns oft die Freiheit nehmen, nicht nur die einelementige Sequenz, sondern auch längere Sequenzen in der Mixfixnotation $\langle 1, 7, 3, 12, 9\rangle$ zu schreiben.

[3] Für die Benennung dieses Typs gibt es verschiedene Traditionen in der Literatur. Wir werden hier sowohl *Seq* α als auch *List* α verwenden.

Für Sequenzen hat man gerne eine multiple Konstruktorsicht, je nachdem, ob man sie von links nach rechts oder von rechts nach links verarbeiten will. Dabei drückt das Schlüsselwort GENERATED aus, dass die jeweils angegebenen Funktionen ausreichen, um alle Elemente des Typs zu erzeugen.[4]

PROP *Seq* GENERATED BY $\Diamond$, $\cdot$: $--$ *echter Konstruktor mit prepend*
PROP *Seq* GENERATED BY $\Diamond$, $:\cdot$ $--$ *Pseudokonstruktor mit append*

Die zugehörigen Matching-Propertys sehen dann so aus:

PROP s MATCHES $(lead :\cdot z) \iff s \neq \Diamond$
PROP s MATCHES $(lead :\cdot z) \implies lead = front(s) \wedge z = last(s)$

Man beachte, dass die Definitionen auf Basis dieser Pseudokonstruktoren im Allgemeinen extrem ineffizient sind, weil bei einem Pattern der Bauart

DEF $f(lead :\cdot x) = \ldots f(lead) \ldots x \ldots$

das Element x nur mit dem Aufwand $\mathcal{O}(n)$ erreicht werden kann, was insgesamt wegen der Rekursion zu einem Aufwand der Größenordnung $\mathcal{O}(n^2)$ führt. (In Kapitel 12 werden wir dazu allerdings noch Erfreuliches erfahren.)

In Abschnitt 1.2.2 werden wir gleich noch eine Reihe weiterer fundamentaler Operationen auf Listen kennen lernen, die unter dem Schlagwort *Map-Filter-Reduce* bekannt geworden sind.

Einige Sprachen, z. B. HASKELL, führen für den Listentyp spezielle Notationen ein: Mit [*Nat*] wird der Typ der Listen über *Nat* beschrieben, was unserem *Seq Nat* entspricht. Außerdem gibt es noch spezielle Schreibweisen für Listenkomprehension, endliche Listen etc.

1.2 Funktionale

Zum Kern der Funktionalen Programmierung gehört das Konzept der *Funktionen höherer Ordnung* oder der *Funktionale*. Durch sie kommt ein großer Teil der Eleganz, der Kompaktheit und damit der Produktivität dieses Programmierparadigmas zustande.

Während man in der Frühzeit der Programmierung noch die Befürchtung hatte, dass die Eleganz von Funktionalen mit hohen Effizienzverlusten bezahlt werden muss, weiß man heute, dass die Compiler solche Funktionen nahezu ohne Overhead implementieren können. Es gibt also keinen Grund mehr, darauf zu verzichten.

[4] Dieses so genannte *Erzeugungsprinzip* ist eine fundamentale Eigenschaft sowohl in der algebraischen Spezifikation von Datentypen als auch in Beweissystemen.

Definition (Funktional, Funktion höherer Ordnung)

Funktionen, die als Parameter oder Resultate wieder Funktionen haben, bezeichnet man als ***Funktionen höherer Ordnung*** bzw. ***Funktionale***.

Ist z. B. eine reelle Funktion gegeben, so kann man sie verschieben, spiegeln oder strecken (vgl. Abbildung 1.1). Das führt zu folgenden Funktionen:

FUN *shift*: $Real \to (Real \to Real) \to (Real \to Real)$
DEF *shift* $dx\ f\ x = f(x - dx)$

FUN *mirror*: $(Real \to Real) \to (Real \to Real)$
DEF *mirror* $f\ x = f(-x)$

FUN *stretch*: $Real \to (Real \to Real) \to (Real \to Real)$
DEF *stretch* $r\ f\ x = f(\frac{x}{r})$

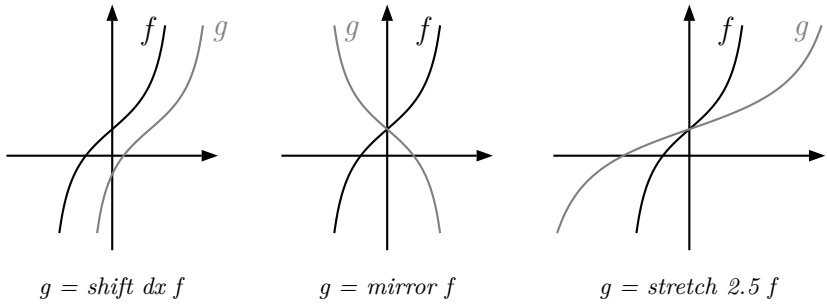

Abb. 1.1: Anwendung von Funktionalen auf eine Funktion f

Für Funktionen, die Resultate einer Berechnung sind, benutzt man auch gerne die λ-Notation:

FUN *shift*: $Real \to (Real \to Real) \to (Real \to Real)$
DEF *shift* $dx\ f = \lambda x \bullet f(x - dx)$

Dies drückt deutlich aus, dass z. B. durch die Applikation *shift*$(\frac{\pi}{2})$ *cos* eine Funktion erzeugt wird, deren Verlauf der Cosinus-Funktion entspricht, die aber um $\frac{\pi}{2}$ auf der x-Achse nach rechts verschoben ist.

Wir setzen im Folgenden dieses Konzept als bekannt voraus und erinnern hier nur an einige wichtige Funktionale, die wir im weiteren Verlauf immer wieder verwenden werden.

1.2.1 Allgemeine Funktionale

Es gibt eine Vielzahl von Funktionen höherer Ordnung, die aus der Mathematik wohl bekannt sind und deren Verfügbarkeit viele Programme eleganter

schreiben lässt. Einige repräsentative Vertreter dieser Funktionsfamilie haben wir in einer so genannten *Struktur* (vgl. Kapitel 4) in Programm 1.1 zusammengefasst. Man beachte, dass alle diese Funktionen polymorph sind.

Programm 1.1 Nützliche Funktionen höherer Ordnung

```
STRUCTURE HigherOrder = {
  FUN _ . _: α → (α → β) → β                          -- Applikation, invers
  DEF x.f = f x
  FUN _ ∘ _: (β → γ) → (α → β) → α → γ                -- Funktionskomposition
  DEF (g ∘ f) x = g (f x)
  FUN _ ; _: (α → β) → (β → γ) → α → γ                -- Funktionskomposition
  DEF (f ; g) = g ∘ f
  FUN id: α → α                                       -- Identität
  DEF id x = x
  FUN K: α → β → α                                    -- Konstante Funktion
  DEF K a b = a
  FUN curry: (α × β → γ) → (α → β → γ)                -- Currying
  DEF (curry f) a b = f(a, b)
  FUN uncurry: (α → β → γ) → (α × β → γ)              -- Uncurrying
  DEF (uncurry g)(a, b) = g a b
  FUN _ ^ _: (α → α) × Nat → (α → α)                  -- Iterierte Applikation
  DEF f^0 = id                                        -- geschrieben f^n
  DEF f^n = f^(n−1) ∘ f
  FUN _ while _: (α → α) × (α → Bool) → (α → α)       -- Wiederholung
  DEF (f while p) x = IF p x THEN (f while p)(f x) ELSE x FI
}
```

Mit der invertierten Funktionsapplikation lässt sich eine Notation simulieren, die in der objektorientierten Programmierung sehr beliebt ist (und manchmal sogar als Konzept verkauft wird). Wir werden davon in Kapitel 4 extensiv Gebrauch machen, um unsere konzeptuelle Funktionssicht mit bekannten traditionellen Notationen in Einklang zu bringen.

Wir führen noch eine Konvention ein, die in einigen Anwendungen zu kompakten und eleganten Notationen führt. Sei eine Menge $f_1 : \alpha \to \beta_1, \ldots, f_n : \alpha \to \beta_n$ von Funktionen über dem gleichen Definitionsbereich gegeben. Dann lassen wir die *Tupelapplikation* zu:

$$(f_1, \ldots, f_n)(a) = (f_1(a), \ldots, f_n(a)) \quad -- \textit{ Tupelapplikation}$$

1.2.2 Catamorphismen (Map-Filter-Reduce)

Sehr viele elegante Algorithmen der Funktionalen Programmierung basieren auf dem so genannten *Map-Filter-Reduce*-Prinzip. Dahinter steckt die Beob-

achtung, dass die Algorithmen genau den Aufbau der zugrunde liegenden Datenstruktur widerspiegeln. Man spricht daher auch vom *Homomorphie-Prinzip* oder – wenn man Eindruck schinden will – von *Catamorphismen*. Diese Programmiertechnik ist in den letzten Jahren sehr stark von Richard Bird und Lambert Meertens ausgebaut worden. Genaueres findet man z. B. in [20], wo auch weitere Literaturhinweise stehen.

Anmerkung: Jeffrey Dean und Sanjay Ghemawat von der Firma Google beschreiben in [40] ein so genanntes MapReduce-Pattern, das in ihrer Firma in Hunderten von Programmen implementiert wurde, die täglich in mehr als Tausend Jobs viele Terabytes von Daten verarbeiten. Das ist ein deutliches Indiz für die zentrale Bedeutung dieses Paradigmas. Es stützt auch die These, dass dieses Muster in ungezählten PASCAL-, C- *oder* JAVA-*Programmen steckt – wenn auch unerkannt.*

Catamorphismen lassen sich im Prinzip auf allen möglichen Datenstrukturen in analoger Weise definieren. Wir betrachten sie hier exemplarisch für den Fall der endlichen Sequenzen (vgl. Abschnitt 1.1.11). In Kapitel 2 werden wir das Konzept auf unendliche Listen erweitern.

Die Map-Operation

Die *Map*-Operation wendet eine Funktion f auf alle Elemente einer Sequenz an.

FUN $map\colon (\alpha \to \beta) \to Seq\ \alpha \to Seq\ \beta$ $\quad -- \ Map\text{-}Funktion$
DEF $map\ f\ \Diamond = \Diamond$
DEF $map\ f\ (a \cdot\colon rest) = (f\ a) \cdot\colon (map\ f\ rest)$

Statt *map* werden wir meist „ $*$ “ schreiben und dabei faktisch eine Infix-Notation verwenden, die allerdings etwas trickreich realisiert ist, indem wir die Operation „ $*$ “ als Postfix-Operation mit Currying definieren. Der Vorteil ist, dass wir dann Terme wie $(f*)$ von vornherein als legale Konstrukte haben und Dinge wie $(g*) \circ (f*)$ schreiben dürfen.

FUN $\_*\colon\ (\alpha \to \beta) \to Seq\ \alpha \to Seq\ \beta$ $\quad -- \ alternative\ Schreibweise$
DEF $f * s = map\ f\ s$

Die Map-Operation lässt sich naheliegend auf mehrstellige Funktionen verallgemeinern. Wir erlauben also generell z. B. zu schreiben

$$\ldots + *(a,\ b) \ldots$$

was bedeutet, dass die beiden Sequenzen a und b elementweise addiert werden. Das lässt sich ebenso mit drei-, vier-, fünf-stelligen Operationen etc. machen. Wir verzichten darauf, das hier einzeln zu definieren.

Ein Problem muss allerdings geklärt werden: *Was geschieht bei verschieden langen Sequenzen?* Um die Undefiniertheiten möglichst gering zu halten, legen wir fest, dass in diesen Fällen das Ergebnis die Länge der *kürzesten* Argumentsequenz hat.

FUN map: $(\alpha_1 \times \alpha_2 \to \beta) \to (Seq\ \alpha_1 \times Seq\ \alpha_2 \to Seq\ \beta)$
DEF $map\ f\ (\Diamond, \Diamond) = \Diamond$
DEF $map\ f\ (\Diamond, s_2) = \Diamond$
DEF $map\ f\ (s_1, \Diamond) = \Diamond$
DEF $map\ f\ (a_1 \cdot{:}\ rest_1, a_2 \cdot{:}\ rest_2) = f(a_1, a_2) \cdot{:}\ map\ f\ (rest_1, rest_2)$

Den ersten Fall hätten wir weglassen können, weil er von den nächsten beiden jeweils mit abgedeckt wird. Die vollständige Definition ist aber klarer und deshalb besser.

Die Zip-Operation

Die Verallgemeinerung auf mehrstellige Funktionen macht die Map-Operation sehr flexibel. Aber für den zweistelligen Fall möchte man gerne die Infix-Schreibweise retten. Dafür hat sich in der Literatur der Name *Zip* eingebürgert, da die beiden Sequenzen „reißverschlussartig“ zusammengefügt werden.

FUN $\_ \setminus \_ / \_$: $Seq\ \alpha \times (\alpha \times \beta \to \gamma) \times Seq\ \beta \to Seq\ \gamma$ −− *Zip*
DEF $a \setminus \oplus / b = \oplus * (a, b)$ −− *geschrieben* $a \overset{\oplus}{\curlyvee} b$

Man beachte, dass bei verschieden langen Sequenzen die längere abgeschnitten wird; wir haben also insbesondere z. B. die Eigenschaft $\Diamond \overset{\oplus}{\curlyvee} b = \Diamond$.

Die Filter-Operation

Die *Filter*-Operation extrahiert aus einer Sequenz die Teilsequenz derjenigen Elemente, die ein gegebenes Prädikat p erfüllen.

FUN *filter*: $(\alpha \to Bool) \to Seq\ \alpha \to Seq\ \alpha$ −− *Filter-Funktion*
DEF $filter\ p\ \Diamond = \Diamond$
DEF $filter\ p\ (a \cdot{:}\ rest) =$ IF $p\ a$ THEN $a \cdot{:}\ filter\ p\ rest$
ELSE $filter\ p\ rest$ FI

Aus Gründen der Flexibilität geben wir zwei notationelle Varianten an, wobei wir bei der ersten wieder von Infix- auf Postfix-Notation übergehen, um z. B. Dinge wie $(g*) \circ (p\triangleleft) \circ (f*)$ schreiben zu können. Man beachte, dass wir dabei trotzdem die Notation $p \triangleleft s$ beibehalten können, die zumindest wie eine Infix-Notation aussieht.

FUN $\_\triangleleft$: $(\alpha \to Bool) \to Seq\ \alpha \to Seq\ \alpha$ −− *Filter-Funktion*
DEF $p \triangleleft s = filter\ p\ s$

FUN $\_ \triangleright \_$: $Seq\ \alpha \times (\alpha \to Bool) \to Seq\ \alpha$
DEF $s \triangleright p = p \triangleleft s$

Anmerkung: Diese Funktion lässt sich nicht *ohne weiteres auf Strukturen wie Bäume oder Matrizen übertragen, da sie die Gestalt der Datenstruktur zerstören kann. Um dieses Problem zu umgehen, führt man für die dann fehlenden Elemente „virtuelle“ Platzhalter ein, z. B. mit Hilfe der Datenstrukturen option in* OPAL *oder Maybe in* HASKELL *(s. Kapitel 8).*

Für gewisse Aufgaben brauchen wir weitere Varianten der Funktionsfamilie Filter, die mit zwei- statt mit einstelligen Prädikaten arbeiten. Im Wesentlichen geht es darum, die Sequenzelemente relativ zu ihren Nachbarn zu bewerten. Während aber das Filtern mit einstelligen Prädikaten offensichtlich ist, muss man sich bei zweistelligen Prädikaten zwischen mehreren möglichen Festlegungen entscheiden. Wir können z. B. bestimmen, dass $(<) \triangleleft s$ diejenigen Elemente beibehalten soll, die jeweils kleiner als ihr Nachfolger sind:

$$(<) \triangleleft \langle 2, 5, 1, 1, 6, 6, 4, 7\rangle = \langle 2, 1, 4\rangle$$

Diese Funktion kann folgendermaßen definiert werden:

```
FUN _◁: (α × α → Bool) → Seq α → Seq α          -- Filter-Funktion
DEF p ◁ ◊ = ◊
DEF p ◁ ⟨x⟩ = ◊
DEF p ◁ (x ·: y ·: rest) = IF p(x, y) THEN x ·: (p ◁ (y ·: rest))
                                      ELSE      (p ◁ (y ·: rest)) FI
FUN _ ▷ _: Seq α × (α × α → Bool) → Seq α
DEF s ▷ p = p ◁ s
```

Die Reduce-Operation

Schließlich betrachten wir noch die *Reduce*-Funktion, die die Elemente einer Sequenz mit Hilfe einer Operation „akkumuliert“. Klassische Beispiele sind die Summe aller Elemente, das Produkt aller Elemente etc. Diese Funktion muss allerdings in mehreren Varianten bereitgestellt werden, da die leere Sequenz bzw. fehlende Assoziativität Probleme machen können. Wir verwenden wieder eine Postfix- anstelle der Infix-Notation.

```
FUN _/:  (α × α → α) → Seq α → α    -- Reduce
FUN _↚:  (α × α → α) → Seq α → α    -- alternative Notation
DEF ↚ = /
DEF ⊕ /⟨a⟩ = a
DEF ⊕ /(a ·: rest) = a ⊕ (⊕ / rest)
```

Diese Funktion führt z. B. zu folgender Auswertung: $+ /\langle a, b, c, d, e\rangle = a + (b + (c + (d + e)))$. Deshalb verwenden wir, wenn wir die Richtung der Auswertung verdeutlichen wollen, auch das Symbol $\not\leftarrow$; das heißt, $\oplus / s \mathrel{\widehat{=}} \oplus \not\leftarrow s$. Das mag beim Minuszeichen anstelle von Plus nicht gewünscht sein. Deshalb gibt es auch einen Operator für die inverse Richtung.[5]

```
FUN _ ↛:  (α × α → α) → Seq α → α
DEF ⊕ ↛⟨a⟩ = a
DEF ⊕ ↛(lead ·: a) = (⊕ ↛ lead) ⊕ a
```

[5] Man beachte, dass wir hierbei erweitertes Patternmatching mittels der Funktion ·: verwenden, die ein Element „hinten“ an eine Sequenz anhängt.

Mit diesen Funktionen kann man schnell Algorithmen zusammensetzen. Ein typisches Beispiel ist die Standardabweichung einer Verteilung, die definiert ist als die Wurzel der Summe der Quadrate aller Abweichungen vom Mittelwert m:

$$\text{DEF } abw\ s = \sqrt{+ / (\_\ \hat{}\ 2) * |m - \_\ |\ * s} \quad \text{WHERE } m = mittel(s)$$
$$\text{DEF } mittel\ s = \frac{+/s}{length\ s}$$

Ein Problem bleibt: Die Reduce-Operatoren sind für die leere Sequenz nicht definiert. Für dieses Problem bieten sich zwei Lösungen an: eine konventionelle und eine fortschrittliche. Wir beginnen mit der fortschrittlichen.

Motiviert durch die klassischen mathematischen Definitionen, die z. B. die Summe über der leeren Menge als 0 und das Produkt über der leeren Menge als 1 festlegen, wählen wir als Standardwert für $\oplus / \Diamond$ das *neutrale Element* der Operation $\oplus$. Das heißt, wir definieren die Reduce-Operation nicht über beliebigen Typen α und Operationen $\oplus$, sondern nur über solchen, die ein neutrales Element besitzen. Dazu benötigen wir das Sprachmittel der Typklassen, das in Kapitel 9 eingeführt wird. Mit den dort beschriebenen Notationen können wir Reduce folgendermaßen definieren:

$$\text{FUN } \_/\colon (\alpha \times \alpha \to \alpha) \to Seq\ \alpha \to \alpha \quad \text{VAR } \alpha\colon \mathit{Monoid} \quad -- \ Reduce$$
$$\text{DEF } \oplus / \Diamond = unit \qquad -- \ neutrales\ Element$$
$$\text{DEF } \oplus / (a\ \cdot\colon rest) = a \oplus (\oplus\ /\ rest)$$

In der Typklasse *Monoid* wird ein neutrales Element *unit* eingeführt. Und die Struktur *Nat* kann als Monoid bzgl. $(+, 0)$ und auch bzgl. $(\cdot, 1)$ charakterisiert werden. Entsprechendes gilt z. B. auch für Sequenzen *Seq* bzgl. $(+\!+, \Diamond)$ oder für Mengen *Set* bzgl. $(\cup, \emptyset)$ etc. (Mehr dazu in Kapitel 9.)

Die konventionelle Lösung besteht darin, das fehlende Element als zusätzliches Argument bereitzustellen, und zwar entweder über Parametertupel, über Currying oder über Mixfix-Notationen. (Wir wählen letztere Variante.)

Als zusätzlichen Effekt können wir jetzt die Operation etwas verallgemeinern: Das abschließende Element braucht nicht das neutrale Element zu sein und es kann sogar einen anderen Typ β haben (sofern $\oplus$ dazu passt).

$$\text{FUN } \_\ /\ \_\ |\ \_\colon (\alpha \times \beta \to \beta) \times Seq\ \alpha \times \beta \to \beta$$
$$\text{DEF } \oplus / \Diamond\ |\ e = e$$
$$\text{DEF } \oplus / a\ \cdot\colon rest\ |\ e = a \oplus (\oplus\ /\ rest\ |\ e)$$

In der BIBLIOTHECA OPALICA [1], also der Bibliothek der Sprache OPAL, wird das etwas anders geschrieben: $(\oplus, e)\ /\ S$.

Der generelle Hom-Operator

Alle in diesem Abschnitt eingeführten Funktionen können über eine einzige Funktion mit entsprechend vielen Parametern definiert werden:

$$\begin{array}{l}\text{FUN } hom\colon (\gamma \times \delta \to \delta) \times \delta \times (\beta \to \gamma) \times (\beta \to Bool) \times (\alpha \to \beta) \\ \qquad \to (Seq\ \alpha \to \delta) \\ \text{DEF } hom(\oplus,\ e,\ g,\ p,\ f)(\Diamond) = e \\ \text{DEF } hom(\oplus,\ e,\ g,\ p,\ f)(a\ \cdot\!:\ rest) = \\ \quad \text{IF } p\ (f\ a) \text{ THEN } (g\ (f\ a)) \oplus hom(\oplus,\ e,\ g,\ p,\ f)(rest) \\ \qquad\qquad \text{ELSE} \qquad\qquad\quad hom(\oplus,\ e,\ g,\ p,\ f)(rest) \text{ FI}\end{array}$$

Damit bestehen folgende Gleichungen (in denen wir einige der Funktionale aus Programm 1.1 auf Seite 25 verwenden):

$$\begin{array}{l}\text{PROP } f * s = hom(\cdot\!:,\ \Diamond,\ id,\ K\ true,\ f)(s) \\ \text{PROP } p \lhd s = hom(\cdot\!:,\ \Diamond,\ id,\ p,\ id)(s) \\ \text{PROP } \ \oplus /s \mid e = hom(\oplus,\ e,\ id,\ K\ true,\ id)(s)\end{array}$$

Umgekehrt kann auch *hom* als Komposition der anderen Operatoren definiert werden:

$$\text{PROP } hom(\oplus,\ e,\ g,\ p,\ f)(s) = \oplus\ /\ g * p \lhd f * s \mid e$$

Mit der Funktion *hom* kann unser obiges Beispiel der mittleren Abweichung so geschrieben werden:

$$\text{DEF } abw\ s = \sqrt{hom(+,\ 0,\ id,\ K\ true,\ (\_\ \hat{}\ 2) \circ |m - \_|)(s)} \quad \text{WHERE} \ldots$$

Dabei haben wir noch eine weitere der elementaren Gleichungen für das Map-Filter-Reduce-Paradigma benutzt:

$$\begin{array}{l}\text{PROP } g * f * s = (g \circ f) * s \\ \text{PROP } p \lhd f * s = (p \circ f) \lhd s\end{array}$$

Beispiel 1.1

Zum Abschluss dieser einführenden Erinnerung soll noch das Standardbeispiel *quicksort* zeigen, wie diese Funktionale eingesetzt werden können:

$$\begin{array}{l}\text{FUN } quicksort\colon Seq\ \alpha \to Seq\ \alpha \\ \text{DEF } quicksort\ \Diamond = \Diamond \\ \text{DEF } quicksort\ s = \text{LET } x \quad = arb\ s \\ \qquad\qquad\qquad\qquad\quad small \ = (< x) \lhd s \\ \qquad\qquad\qquad\qquad\quad medium \ = (= x) \lhd s \\ \qquad\qquad\qquad\qquad\quad large \ = (> x) \lhd s \\ \qquad\qquad\qquad\quad \text{IN} \\ \qquad\qquad\qquad\qquad (quicksort\ small) +\!\!+ medium +\!\!+ (quicksort\ large)\end{array}$$

1.3 Semantik und Auswertungsstrategien

Bei Sprachen unterscheidet man generell zwischen ihrer ***Syntax***, also der *Schreibweise* der einzelnen Sprachkonstrukte, und ihrer ***Semantik***, also der *Bedeutung* der einzelnen Sprachkonstrukte.

Bei funktionalen Sprachen sollte man erwarten, dass die Definition der Semantik trivial ist, weil sie sich direkt auf den mathematischen Funktionsbegriff stützt. Im Prinzip ist das auch so, aber es gibt doch ein paar Subtilitäten, die wir kurz ansprechen müssen. Wir können diese grundlegenden Fragen der Semantik hier nur skizzieren; für weitergehende Informationen verweisen wir auf Spezialliteratur, z. B. [135, 60, 126, 63, 146]. Für die speziellen Aspekte, die wir hier benötigen, sei auch auf den nach wie vor lesenswerten (und zum Glück wieder erhältlichen) Klassiker von Zohar Manna [96] verwiesen; auch in [50] findet man eine detaillierte Diskussion.

Im Rahmen dieses Buches sind vor allem zwei semantische Fragen von Bedeutung, weil die verschiedenen Programmiersprachen und -techniken sich in diesen beiden Punkten auf eine für das Programmieren relevante Weise unterscheiden.

- Wie werden die Argumente von Funktionen behandelt?
- Wie wird Rekursion behandelt?

Diese beiden Aspekte wollen wir im Folgenden kurz betrachten. Wir tun das im Rahmen der zwei wichtigsten Arten der Semantikdefinition von Sprachen, der so genannten *denotationellen Semantik* und der so genannten *operationalen Semantik*.

1.3.1 Denotationelle Semantik

Bei der ***denotationellen Semantik*** wird jedes Programm direkt als („Denotation“ für) eine mathematische Funktion charakterisiert.

Der kritischste Punkt ist dabei der Umgang mit ***partiellen Funktionen***. In der traditionellen Mathematik werden solche Funktionen gerne schamhaft „wegdefiniert“, aber in der Informatik kann man sich das nicht leisten. Fehlerhafte Operationen wie die Division $\frac{x}{y}$ für $y = 0$ oder nichtterminierende Programme wie

$\text{DEF } f(0) = 1$
$\text{DEF } f(x) = x \cdot f(x+1)$

sind Phänomene, die sich nicht ignorieren lassen.

Um solche Situationen auf der semantischen Ebene in den Griff zu bekommen, führt man ein spezielles Element „$\perp$“ (genannt „Bottom“) ein, das für „undefiniert“ steht. Statt zu sagen, *„die Funktion f ist an der Stelle a undefiniert“*, kann man dann einfach schreiben „$f(a) = \perp$“.

Anmerkung: Man beachte, dass das nur eine bequeme Kurznotation ist, kein syntaktisches Konstrukt der Programmiersprache. Ausdrücke wie IF $x = \perp$ THEN ... *sind in Programmen* ***nicht*** *zulässig!*

In diesem Zusammenhang betrifft eine wesentliche Designentscheidung bei Programmiersprachen die Frage, wie undefinierte Argumente von Funktionen behandelt werden. Dabei unterscheidet man zwei Arten von Funktionen: strikte und nichtstrikte.

Definition (Strikte und nichtstrikte Funktionen)

- ***Strikte Funktionen*** sind undefiniert, sobald mindestens eines der Argumente undefiniert ist. Kurz

  $$f(\ldots, \bot, \ldots) = \bot$$

- ***Nichtstrikte Funktionen*** können auch dann noch definiert sein, wenn eines ihrer Argumente undefiniert ist.

  $$f(\ldots, \bot, \ldots) \neq \bot \quad \text{(abhängig von den übrigen Argumenten)}$$

Bei Bedarf verwendet man auch eine präzisere Terminologie und sagt, *die Funktion f ist strikt (bzw. nichtstrikt) im i-ten Argument.*

Die große Mehrheit der Funktionen, die wir in der Programmierung verwenden, sind strikt. Aber es gibt auch bedeutende Ausnahmen: Das IF.THEN.ELSE.FI-Konstrukt kann als Funktion *if*(_ , _ , _) mit drei Argumenten aufgefasst werden. Diese Funktion ist strikt im ersten und nichtstrikt im zweiten und dritten Argument.[6]

In den meisten Programmiersprachen sind die benutzerdefinierten Funktionen grundsätzlich strikt; das gilt in nahezu allen imperativen Sprachen, aber auch in funktionalen Sprachen wie ML oder OPAL. In einigen wenigen Sprachen sind diese Funktionen dagegen nichtstrikt; das prominenteste Beispiel dafür ist HASKELL. Es gibt aber auch nichtstrikte ML-Dialekte.

Für die semantische Erklärung *rekursiver Deklarationen* wird dann eine spezielle partielle Ordnung „$\sqsubseteq$“ („*less defined*“) eingeführt, in der „$\bot$“ das kleinste Element ist. Über dieser Ordnung werden rekursive Deklarationen als so genannte *kleinste Fixpunkte* der zugehörigen Gleichungen interpretiert. Wir kommen auf diese Konstruktion in Kapitel 10 im Zusammenhang mit Fixpunktprogrammierung noch ausgiebiger zurück.

Die denotationelle Semantik ist vor allem dann relevant, wenn man „mit Programmen rechnen“ will (vgl. Abschnitt 1.4) und deshalb die gültigen Rechenregeln kennen muss.

1.3.2 Operationale Semantik

Bei der ***operationalen Semantik*** definiert man, wie die Programme auf einer geeignet konzipierten *abstrakten Maschine* ausgewertet werden. Die Menge

[6] Man kann zeigen, dass jede Programmiersprache mit Rekursion oder Iteration (also jede praktisch brauchbare Sprache) mindestens ein nichtstriktes Konstrukt enthalten muss. Meistens übernimmt das *if* diese Rolle.

aller Auswertungen induziert dann eine Funktion (die mit der denotationellen Semantik identisch sein sollte).

Auch in diesem Rahmen ist die Frage der Argumentauswertung ein zentraler Aspekt der Semantik; denn strikte und nichtstrikte Funktionen müssen sich unterscheiden. Allerdings zeigt sich, dass die operationale Semantik filigraner ist: Bei den nichtstrikten Funktionen gibt es unterschiedliche Varianten.

Die Essenz einer operationalen Semantik manifestiert sich in den so genannten *Auswertungsstrategien*, also der Art, wie die Maschine Funktionsargumente behandelt. In unserem Kontext sind vor allem drei Strategien von Bedeutung: Call-by-value, Call-by-name und Call-by-need.

Definition (Call-by-value, Call-by-name und Call-by-need)

- ***Call-by-value*** (auch *eager* genannt). Bei dieser Strategie werden alle Argumentausdrücke vollständig ausgewertet, bevor die Funktion (mit diesen Argument*werten*) aufgerufen wird. Das geschieht unabhängig davon, ob das Argument zur Berechnung des Funktionswerts überhaupt gebraucht wird oder nicht. Funktionen, die mit Call-by-value ausgewertet werden, sind *strikt*.
- ***Call-by-name***. Hier werden beim Funktionsaufruf die Argument*ausdrücke* selbst unausgewertet übergeben und erst bei der Verwendung der entsprechenden Parameter im Rumpf berechnet.[7] Funktionen, die mit Call-by-name ausgewertet werden, sind *nichtstrikt*.
- ***Call-by-need*** (auch ***lazy*** genannt). Das ist eine Variante der Call-by-name-Strategie, bei der durch so genanntes *Sharing* verhindert wird, dass die Argumentausdrücke unter Umständen mehrfach ausgerechnet werden. Funktionen, die mit Call-by-need ausgewertet werden, sind *nichtstrikt*.

Um die Unterschiede dieser Strategien zu illustrieren, verwenden wir ein einfaches, wenn auch etwas artifizielles Beispiel.

$\text{DEF } foo(x, y) = \text{IF } x > 0 \text{ THEN } x \text{ ELSE } y \text{ FI}$

Betrachten wir folgenden Aufruf:

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$

Bei der *Call-by-value*-Strategie werden zuerst die Argumente ausgewertet, was auf $foo(1, \frac{1}{0})$ führt. Weil $\frac{1}{0}$ undefiniert ist, scheitert die Auswertung des zweiten Arguments, wodurch der ganze Aufruf undefiniert ist. (Es wird ein „Zero-divide-Error“ gemeldet.) Das ist umso ärgerlicher, weil für $x = 1$ der Parameter y gar nicht gebraucht wird. Insgesamt haben wir also die folgende Auswertung:

[7] Manchmal unterscheidet man noch genauer zwischen *Call-by-name* und *Call-by-expression*; aber in unserer Betrachtung genügt diese etwas gröbere Sichtweise.

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$ —— *call-by-value*
$\rightsquigarrow foo(1, \frac{1}{0})$
$\rightsquigarrow foo(1, \bot)$
$\rightsquigarrow \bot$

Bei der *Call-by-name*-Strategie werden die beiden Argumentausdrücke vorläufig nicht ausgewertet. Erst wenn die entsprechenden Parameter im Rumpf benötigt werden, erfolgt die Berechnung der Argumente. Das führt in unserem Beispiel auf folgende Auswertung:

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$ —— *call-by-name*
$\rightsquigarrow$ IF $cos(2\pi) > 0$ THEN $cos(2\pi)$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow$ IF $1 > 0$ THEN $cos(2\pi)$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow cos(2\pi)$
$\rightsquigarrow 1$

Weil für den gegebenen x-Wert der y-Wert nie gebraucht wird, hat die Undefiniertheit von $\frac{1}{sin(2\pi)}$ keine negative Auswirkung. Aber man erkennt in diesem Beispiel auch den gravierenden Nachteil der Call-by-name-Strategie: Der Ausdruck $cos(2\pi)$ wird zweimal ausgewertet!

Die *Call-by-need*-Strategie will genau dieses Effizienzproblem beheben. Indem der Compiler intern geeignete Pointer setzt, erreicht man so genanntes *Sharing*. Das heißt, sobald der Argumentwert das erste Mal berechnet wurde, steht er auch allen anderen Applikationsstellen zur Verfügung. Damit sieht unsere Auswertung folgendermaßen aus:

$foo(cos(2\pi), \frac{1}{sin(2\pi)})$ —— *call-by-need (lazy)*
$\rightsquigarrow$ IF $\boxed{cos(2\pi)} > 0$ THEN $\boxed{cos(2\pi)}$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow$ IF $\boxed{1} > 0$ THEN $\boxed{1}$ ELSE $\frac{1}{sin(2\pi)}$ FI
$\rightsquigarrow 1$

Die Call-by-need-Auswertung ist zwar effizienter als die Call-by-name-Auswertung; aber mit der Call-by-value-Auswertung kann sie trotzdem nicht mithalten. Deshalb werden Sprachendesigner, die auf Effizienz Wert legen, immer eine Call-by-value-Semantik vorziehen.[8] In Kapitel 2 werden wir allerdings sehen, dass Laziness ein wichtiges Hilfsmittel ist, um *unendliche Datenstrukturen* zu implementieren. Dort werden wir auch zeigen, wie sich „punktuelle“ Laziness in eine Call-by-value-Sprache einbauen lässt.

Im Folgenden werden wir – wie bei funktionalen Sprachen üblich – die Begriffe *Call-by-value* und *strikt* sowie *Call-by-need* und *lazy* jeweils synonym verwenden.

[8] Im Compilerbau wird die Technik der so genannten Striktheitsanalyse eingesetzt, um bei Call-by-name- oder Call-by-need-Sprachen den Overhead möglichst klein zu halten. Aber diese Technik ist aufwendig und findet nicht alle Optimierungen.

Anmerkung 1: In manchen Büchern, z. B. [50], wird bei Laziness und Call-by-need filigraner unterschieden. Laziness bedeutet dort, dass die Argumente von Konstruktor*funktionen nicht ausgewertet werden; das reicht aus, um unendliche Datenstrukturen zu implementieren.*

Anmerkung 2: Die Auswertungsstrategien funktionaler Sprachen stehen in engem Zusammenhang mit den Reduktionsstrategien von λ-Ausdrücken im λ-Kalkül. So korrespondiert die Call-by-value-Strategie zur so genannten Applicative-order-reduction. Die Call-by-name-Strategie entspricht der Normal-order-reduction. Genauere Betrachtungen hierzu findet man in [50].

1.4 Mit Programmen rechnen

Funktionale Programme sind semantisch direkt im mathematischen Funktionsbegriff verankert. Das hat einen außerordentlich angenehmen Nebeneffekt: Man kann mit den Programmen genauso rechnen, wie man es an anderen Stellen in der Mathematik gelernt hat. Dabei gibt es durchaus Analogien zum Vorgehen eines Ingenieurs, der eine Differenzialgleichung lösen will: Er kennt eine Reihe von Standardansätzen, die er der Reihe nach probiert, bis einer funktioniert. (Wenn keiner klappt, ist das Problem wirklich schwer.)

Anmerkung: Dieser Ansatz ist ein Zweig der Programmierung, der unter Begriffen wie „Programmtransformation“ oder „deduktive Programmierung“ bekannt geworden ist. Eine umfassende Darstellung findet sich z. B. in [18, 109]. Besonders weit wurde diese Methode von Bird und Meertens entwickelt, vor allem im Zusammenhang mit Map-Filter-Reduce-Programmen [21].

Dabei gibt es zwei prinzipielle Vorgehensweisen. Man kann die Regeln sehr rigoros und formal fassen (als so genannte *Transformationsregeln*), so dass sie weitgehend automatisch von entsprechenden Werkzeugen – im Idealfall vom Compiler – angewandt werden können. Oder man betrachtet das Ganze eher als eine Methode, mit der man seine Programme selbst weiterentwickeln kann. Dabei fängt man oft mit einer (nicht-ausführbaren) Spezifikation an, leitet daraus eine elegante und korrekte aber nicht besonders effiziente erste Lösung ab und entwickelt diese schließlich weiter in ein weniger elegantes aber dafür effizientes Programm. Wir werden uns im Folgenden an diese zweite Sicht halten.

Im Rahmen dieses Buches müssen wir uns auf eine exemplarische Skizze dieser Methodik beschränken. Dabei konzentrieren wir uns auf diejenigen Aspekte, die in späteren Kapiteln noch benötigt werden.

1.4.1 Von linearer Rekursion zu Tail-Rekursion

Ein wichtiges Effizienzproblem bei funktionalen Sprachen betrifft die *effiziente Ausführung rekursiver Funktionen*. In imperativen Sprachen ist das kein so wichtiges Thema, weil man dort primär Schleifen programmiert, die a priori effizient sind (dafür aber weniger elegant und fehleranfälliger).

Die einfachste Form von Rekursion ist die so genannte ***Tail-Rekursion***.[9] Bei dieser Form ist der rekursive Aufruf die äußerste („letzte“) Operation im jeweiligen Zweig der Fallunterscheidung. Ein typisches Beispiel ist die Berechnung des Rests bei der Division:

```
FUN mod: Nat × Nat → Nat
DEF mod(a, b) = IF a < b THEN a
                         ELSE mod(a − b, b) FI    −− Tail-Rekursion
```

Bei musterbasierten Definitionen ist das noch offensichtlicher: dort ist der rekursive Aufruf ganz außen.

Solche Tail-rekursiven Funktionen sind äußerst effizient. Jeder halbwegs ordentliche Compiler erkennt heute diese spezielle Rekursionsform und setzt sie im Maschinencode unmittelbar in Schleifen bzw. direkt in Gotos um.

Aus diesem Grund ist es interessant, funktionale Programme weitestgehend aus Tail-rekursiven Funktionen zusammenzusetzen. Leider ist diese Form aber in vielen Situationen nicht die eleganteste oder „natürlichste“ Formulierung. Ein typisches Beispiel ist die Bildung der Summe einer Sequenz von Zahlen (diesmal in musterbasierter Notation):

```
DEF sum ◊          = 0
DEF sum(x ·: rest) = x + sum(rest)   −− lineare Rekursion
```

Hier finden nach dem rekursiven Aufruf noch weitere Berechnungen statt. Man sagt auch, dass die Operation $(x + \_)$ „nachklappert“. Diese Form wird als ***lineare Rekursion*** bezeichnet. Man kann das optisch noch deutlicher hervorheben, indem man den Rumpf mit Hilfe von LET- oder WHERE-Deklarationen strukturiert. Dann haben linear rekursive Funktionen folgendes Muster:

```
DEF f(x) = z WHERE a = pre(x)
                   r = f(a)        −− lineare Rekursion
                   z = post(x, r)
```

Im obigen *sum*-Beispiel entspricht $z = post(x, r)$ der Operation $z = x + r$.

Weil das Argument x in der nachklappernden Operation *post* noch gebraucht wird, muss der Compiler einen Stack vorsehen, in dem dieses Argument zwischengespeichert wird. Dieser Stack macht Aufwand und führt daher zu einem Effizienzverlust.

Deshalb wird es zu einer interessanten Frage, wie man linear rekursive Funktionen in Tail-rekursive Funktionen umwandeln kann.

Eine Standardtechnik besteht in einer ***Einbettung***: Man definiert – orientiert an der Form der gegebenen Funktion f – eine neue Funktion $\widetilde{f}$, die in geeignetem Sinne eine „Verallgemeinerung“ von f darstellt. Dann versucht man, durch ein bisschen Rechnen diese neue Funktion in Tail-rekursive Form

[9] Dieses deutsch-englische Mischwort ist zwar hässlich, hat sich aber in der Literatur eingebürgert, weshalb wir die Bezeichnung hier beibehalten.

zu bringen. Bevor wir diese Methodik weiter erläutern, illustrieren wir sie an dem konkreten Beispiel der obigen Funktion *sum*.

Wir führen für den nachklappernden Teil einen weiteren Parameter z ein und definieren folgende Einbettung:

$$\text{DEF } \widetilde{sum}(seq, z) = z + sum(seq)$$

Weil die Addition das neutrale Element 0 besitzt, ist *sum* als Spezialfall von $\widetilde{sum}$ darstellbar (was den Begriff „Verallgemeinerung“ rechtfertigt):

$$sum(seq) = 0 + sum(seq) = \widetilde{sum}(seq, 0)$$

Der wesentliche Teil der Entwicklung besteht in dem Versuch, $\widetilde{sum}$ in eine Tail-rekursive Form zu bringen. Dazu betrachten wir die gleichen musterbasierten Fälle wie in der Originalfunktion *sum* und wenden ein bisschen Mathematik an:

$$\begin{array}{lll}
\widetilde{sum}(\lozenge, z) & = z + sum(\lozenge) & -- \textit{ Definition von } \widetilde{sum} \\
& = z + 0 & -- \textit{ Definition von sum} \\
& = z & -- \textit{ Mathematik}
\end{array}$$

$$\begin{array}{lll}
\widetilde{sum}(x \cdot\!: rest, z) & = z + sum(x \cdot\!: rest) & -- \textit{ Definition von } \widetilde{sum} \\
& = z + (x + sum(rest)) & -- \textit{ Definition von sum} \\
& = (z + x) + sum(rest) & -- \textit{ Mathematik} \\
& = \widetilde{sum}(rest, z + x) & -- \textit{ Definition von } \widetilde{sum}
\end{array}$$

Das Ergebnis dieser Berechnungen lässt sich in ein neues Definitionssystem umwandeln. Dieses System ist jetzt Tail-rekursiv!

$$\begin{array}{lll}
\text{DEF } sum(seq) & = \widetilde{sum}(seq, 0) & -- \textit{ Einbettung} \\
\text{DEF } \widetilde{sum}(\lozenge, z) & = z & \\
\text{DEF } \widetilde{sum}(x \cdot\!: rest, z) & = \widetilde{sum}(rest, z + x) & -- \textit{ Tail-Rekursion}
\end{array}$$

Bei dieser Rechnung haben wir zwei Eigenschaften der Addition gebraucht: sie ist assoziativ und hat das neutrale Element 0. Deshalb können wir die Essenz dieser Berechnung in eine allgemeine Regel fassen.

Regel (Tail-Rekursion modulo Assoziativität)

Eine Funktion, die folgendem Schema genügt (lineare Rekursion)

$$\begin{array}{ll}
\text{DEF } \mathcal{F}(\mathcal{A}) & = \mathcal{T} \\
\text{DEF } \mathcal{F}(\mathcal{B}) & = \mathcal{R} \oplus \mathcal{F}(\mathcal{P}) \quad -- \textit{ lineare Rekursion}
\end{array}$$

kann in ein Tail-rekursives Schema transformiert werden:

$$\begin{array}{ll}
\text{DEF } \mathcal{F}(\mathcal{X}) & = \widetilde{\mathcal{F}}(\mathcal{X}, \mathcal{E}) \qquad -- \textit{ Einbettung (neutrales Element)} \\
\text{DEF } \widetilde{\mathcal{F}}(\mathcal{A}, \mathcal{Z}) & = \mathcal{Z} \oplus \mathcal{T} \\
\text{DEF } \widetilde{\mathcal{F}}(\mathcal{B}, \mathcal{Z}) & = \widetilde{\mathcal{F}}(\mathcal{P}, \mathcal{Z} \oplus \mathcal{R}) \quad -- \textit{ Tail-Rekursion}
\end{array}$$

Voraussetzung dafür ist, dass der Operator „ $\oplus$ “ assoziativ ist und ein neutrales Element „$\mathcal{E}$“ besitzt.

In dieser Regel stehen $\mathcal{A}$ und $\mathcal{B}$ für Konstruktorterme, $\mathcal{T}$, $\mathcal{P}$ und $\mathcal{R}$ für Ausdrücke in den Variablen dieser Muster, sowie $\mathcal{F}$, $\widetilde{\mathcal{F}}$, $\mathcal{X}$ und $\mathcal{Z}$ für Identifier.

Übrigens: das neutrale Element ist nicht unbedingt nötig; es geht auch mit Assoziativität alleine, wenn man eine etwas aufwendigere Einbettung in Kauf nimmt.

$$\begin{array}{lll} \text{DEF } \mathcal{F}(\mathcal{A}) & = \mathcal{T} & -- \textit{ sofortige Terminierung} \\ \text{DEF } \mathcal{F}(\mathcal{B}) & = \widetilde{\mathcal{F}}(\mathcal{P}, \mathcal{R}) & -- \textit{ Einbettung} \end{array}$$

Die Definition der Funktion $\widetilde{\mathcal{F}}$ bleibt unverändert.

Anmerkung: Neben der Assoziativität der nachklappernden Operation gibt es noch weitere algebraische Eigenschaften, mit deren Hilfe sich lineare Rekursion in Tail-Rekursion umwandeln lässt (Details findet man z. B. in [18]). In Kapitel 13 werden wir eine besonders wichtige Variante kennen lernen.

1.4.2 Ein „universeller Trick“: Continuations

Wenn Assoziativität nicht gegeben ist und auch keine der anderen Transformationen anwendbar ist, dann bleibt noch ein Trick übrig, der – zumindest formal – immer klappt. Man verwendet eine so genannte ***Continuation***, also eine Funktion, die – intuitiv gesprochen – die „Fortsetzung“ der Berechnung repräsentiert.

Warnung: Wir weisen schon jetzt darauf hin, dass dieser „universelle“ Trick im Allgemeinen ein bisschen Augenwischerei ist. (Genaueres am Ende dieses Abschnitts.)

Beispiel 1.2 (Anwendung von Continuations)

Wir betrachten die Auswertung eines Polynoms

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{1.1}$$

Diese Auswertung geschieht normalerweise nach dem so genannten *Hornerschema*:

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (\cdots + x \cdot (a_{n-1} + x \cdot (a_n)) \ldots))) \tag{1.2}$$

Das lässt sich sofort in ein (linear rekursives) Programm umschreiben, bei dem wir die Koeffizienten $a_0, \ldots, a_n$ als Sequenz repräsentieren.

$$\begin{array}{ll} \text{FUN } \mathit{horner}\colon & \mathit{Seq}\ \mathit{Real} \times \mathit{Real} \to \mathit{Real} \\ \text{DEF } \mathit{horner}(\Diamond,\ x) & = 0 \\ \text{DEF } \mathit{horner}(a\ \cdot\!\!:\ \mathit{rest},\ x) & = a + x \cdot \mathit{horner}(\mathit{rest}, x) \end{array}$$

Da die nachklappernde Operation $(a + x \cdot \_\ )$ nicht assoziativ ist, kann man die schematische Regel nicht anwenden. (Aber man kann versuchen, mit

einer geschickten Einbettung zum Erfolg zu gelangen. Wir überlassen das dem interessierten Leser.) Wir benutzen dieses Beispiel, um den universellen Trick mit den Continuations zu illustrieren.

Wir nehmen eine Einbettung vor, die als zusätzliches Argument die Continuation Γ enthält:

$$\begin{array}{l}\text{FUN } h\colon \mathit{Seq}\ \mathit{Real} \times \mathit{Real} \times (\mathit{Real} \to \mathit{Real}) \to \mathit{Real}\\ \text{DEF } h(s,\ x,\ \Gamma) = \Gamma(\mathit{horner}(s,\ x))\end{array}$$

Jetzt treiben wir unser übliches mathematisches Spiel:

$$\begin{array}{ll} h(\Diamond,\ x,\ \Gamma) & = \Gamma(\mathit{horner}(\Diamond,\ x))\\ & = \Gamma(0)\\ h(a\ \cdot\!\!:\ \mathit{rest}, x, \Gamma) & = \Gamma(\mathit{horner}(a\ \cdot\!\!:\ \mathit{rest},\ x))\\ & = \Gamma(a + x \cdot \mathit{horner}(\mathit{rest}, x))\\ & = \Gamma \circ (a + x \cdot \_)\ (\mathit{horner}(\mathit{rest}, x))\\ & = h(\mathit{rest},\ x,\ \Gamma \circ (a + x \cdot \_)) \end{array}$$

Damit erhalten wir das neue Definitionssystem:

$$\begin{array}{ll}\text{DEF } \mathit{horner}(s, x) & = h(s, x, \mathit{id})\\ \text{DEF } h(\Diamond, x, \Gamma) & = \Gamma(0)\\ \text{DEF } h(a\ \cdot\!\!:\ \mathit{rest}, x, \Gamma) & = h(\mathit{rest},\ x,\ \Gamma \circ (a + x \cdot \_))\end{array}$$

Warum klappt das immer? Der Grund ist ganz einfach: Die Funktionskomposition ist assoziativ und hat die Identitätsfunktion als neutrales Element. *Aber da ist ein Wermutstropfen*: Zwar kann man mit Hilfe von Continuations jede linear rekursive Funktion formal in eine Tail-rekusive Funktion verwandeln, aber damit ist im Allgemeinen kein Effizienzgewinn verbunden! Intern ist immer noch die gleiche Arbeit zu verrichten. (Compilertechnisch gesehen wird der Stack in den Heap verlagert, was den Aufwand sogar erhöht.)

Im obigen Beispiel der Funktion *horner* wird eine lange Continuation aufgebaut

$$\mathit{id} \circ (a_0 + x \cdot \_) \circ (a_1 + x \cdot \_) \circ \ldots \circ (a_n + x \cdot \_),$$

die am Schluss auf den Wert 0 angewandt wird. Der Aufbau dieser langen Funktionskomposition kostet intern natürlich sehr viel Zeit und Speicherplatz, so dass der Gewinn durch die Tail-Rekursion mehr als aufgebraucht wird.

Beispiel 1.3 (Exkurs: Spielen mit Continuations)

Welch skurrile Spielereien man mit diesen Continuations treiben kann, zeigt besonders deutlich folgendes kleine Beispiel. (Dieses Beispiel geht auf Oege de Moor zurück; wir präsentieren es hier in adaptierter Form.) Gegeben sei ein Baum mit natürlichen Zahlen an den Knoten. Er soll so umgeformt werden, dass an allen Knoten der maximale Wert steht, der im Originalbaum vorkommt. Man erwartet, dass man dazu zwei Baumdurchläufe braucht: einen, um das Maximum zu suchen, und einen zweiten, um den neuen Baum zu bau-

en. Aber es gibt ein Programm, das dieses Problem – zumindest scheinbar – in einem Durchlauf löst. Wir arbeiten uns in zwei Schritten zu diesem Programm vor.

Zunächst definieren wir den Baumtyp im Stil von HASKELL mit Hilfe von Currying:

$$\begin{array}{lll} \text{TYPE } \mathit{Tree} & = & \mathit{tree}\ \mathit{Tree}\ \mathit{Nat}\ \mathit{Tree} \quad -\!-\ \mathit{innerer\ Knoten} \\ & \mid & \mathit{leaf}\ \mathit{Nat} \qquad\qquad\qquad\quad -\!-\ \mathit{Blatt} \end{array}$$

Über diesem Typ definieren wir dann eine Funktion *convert*, die zu einem gegebenen Baum ein Paar von Werten liefert, bestehend aus dem maximalen Knoten des Baumes und einer Funktion, die aus diesem maximalen Wert den gewünschten neuen Baum erzeugt.

$$\begin{array}{l} \text{FUN } \mathit{convert}\colon \mathit{Tree} \to \mathit{Nat} \times (\mathit{Nat} \to \mathit{Tree}) \\ \text{DEF } \mathit{convert}(\mathit{leaf}\ x) = (x,\ \lambda m \bullet \mathit{leaf}\ m) \\ \text{DEF } \mathit{convert}(\mathit{tree}\ \mathit{left}\ x\ \mathit{right}) = \text{LET } (m_1, \varphi_l) = \mathit{convert}(\mathit{left}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (m_2, \varphi_r) = \mathit{convert}(\mathit{right}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad m = \mathit{max}(x, m_1, m_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{IN} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (m,\ \lambda m \bullet \mathit{tree}\ \varphi_l(m)\ m\ \varphi_r(m)) \end{array}$$

Der Aufruf dieser Funktion für einen gegebenen Baum t erfolgt dann in der Form

$$\ldots \text{ LET } (m, \varphi) = \mathit{convert}(t) \text{ IN } \varphi(m) \ \ldots$$

Als zweiten Schritt macht man im Wesentlichen das Ergebnis zu einem weiteren Argument, wobei man allerdings die Funktionalität des Parameters φ noch adaptieren muss:

$$\begin{array}{l} \text{FUN } \mathit{conv}\colon \mathit{Tree} \to (\mathit{Nat} \times (\mathit{Nat} \to \mathit{Tree} \to \mathit{Tree})) \to \mathit{Nat} \times (\mathit{Nat} \to \mathit{Tree}) \\ \text{DEF } \mathit{conv}(\mathit{leaf}\ x)(m, \varphi) = (\mathit{max}(x, m), \lambda m \bullet \varphi\ m\ (\mathit{leaf}\ m)) \\ \text{DEF } \mathit{conv}(\mathit{tree}\ \mathit{left}\ x\ \mathit{right})(m, \varphi) = \\ \quad \text{LET } m_0 = \mathit{max}(m, x) \\ \qquad (m_1, \varphi_l) = \mathit{conv}(\mathit{left})(m_0, \varphi) \\ \qquad (m_2, \varphi_r) = \mathit{conv}(\mathit{right})(m_1, \lambda m \bullet \lambda t \bullet \mathit{tree}\ \varphi_l(m)\ m\ t) \\ \quad \text{IN} \\ \qquad (m_2,\ \lambda m \bullet \varphi_r\ m) \end{array}$$

Der zweite Fall lässt sich etwas kompakter schreiben:

$$\begin{array}{l} \text{DEF } \mathit{conv}(\mathit{tree}\ \mathit{left}\ x\ \mathit{right})(m, \varphi) = \\ \quad \text{LET } (m_1, \varphi_l) = \mathit{conv}(\mathit{left})(\mathit{max}(m, x), \varphi) \\ \quad \text{IN} \\ \qquad \mathit{conv}(\mathit{right})(m_1, \lambda m \bullet \mathit{tree}\ \varphi_l(m)\ m) \end{array}$$

Der initiale Aufruf dieser Funktion muss als Argument im Wesentlichen Null und die Identität übergeben:

$$\ldots \text{ LET } (m, \varphi) = \mathit{conv}(t)(0, \lambda m \bullet \lambda t \bullet t) \text{ IN } \varphi(m) \ \ldots$$

Aber natürlich ist das ein Taschenspielertrick. Denn das zweite Ergebnis φ ist eine riesige Funktionskomposition, die alle Konstruktoren des Originalbaums enthält. Bei der Anwendung $\varphi(m)$ dieser Funktion auf das Maximum m wird deshalb die gesamte Struktur des Originalbaums rekonstruiert – und das ist natürlich der zweite Durchlauf.

Diese kleinen Beispiele zeigen, dass die Verwendung von Continuations mit Vorsicht erfolgen muss. Manchmal scheinen sie Probleme zu lösen, aber bei genauerem Hinsehen ist das nur ein oberflächliches Vortäuschen einer Lösung. Deshalb ist die Einführung von Continuations vor allem als Zwischenschritt zur Vorbereitung weiterer Transformationen interessant.[10]

1.4.3 Vereinfachung komplexerer Rekursionen

Lineare Rekursion ist immer noch eine recht einfache Rekursionsform. Richtig interessant wird es, wenn man komplexere Arten von Rekursion betrachtet. Ein berühmtes Beispiel ist die Fibonacci-Funktion:

$$\begin{array}{ll} \mathit{fib}(0) & = 1 \\ \mathit{fib}(1) & = 1 \\ \mathit{fib}(n+1) & = \mathit{fib}(n) + \mathit{fib}(n-1) \quad -- \textit{ Baum-Rekursion} \end{array}$$

Bei dieser Form spricht man von ***baumartiger Rekursion*** (kurz: ***Baum-Rekursion***), weil die Aufrufe baumartig verzweigen. Die Konsequenz ist in der Praxis dramatisch: Die Ausführung hat *exponentiellen Aufwand*.[11]

Die Addition ist assoziativ und hat das neutrale Element 0. Deshalb könnten wir versuchen, die entsprechende Transformation anzuwenden. Das Ergebnis hätte folgende Form:

$$\begin{array}{ll} \mathit{fib}(n) & = f(n, 0) \\ f(0, z) & = z + 1 \\ f(1, z) & = z + 1 \\ f(n+1, z) & = f(n,\ z + f(n-1, 0)) \quad -- \textit{ geschachtelte Rekursion} \end{array}$$

Dies ist eine weitere komplexe Rekursionsform, die so genannte ***geschachtelte Rekursion***, bei der ein rekursiver Aufruf als Argument einen weiteren rekursiven Aufruf enthält. Diese Form ist unter Effizienzgesichtspunkten genauso ungünstig wie die baumartige Rekursion. Deshalb führt dieser naive Versuch in eine Sackgasse.

---

[10] Eine ganz andere – und sehr wichtige – Verwendung von Continuations werden wir in Kapitel 17 kennen lernen: Sie sind ein wesentlicher Bestandteil der so genannten Monaden-basierten Ein-/Ausgabe.

[11] Man kann sich den Spass machen, diese Funktion in irgendeiner beliebigen Programmiersprache aufzuschreiben und dann der Reihe nach $\mathit{fib}(10)$, $\mathit{fib}(20)$, $\mathit{fib}(30)$, $\mathit{fib}(40)$, ... aufzurufen. Dabei lernt man überraschend schnell, was „exponentieller Aufwand“ bedeutet. (Um das Experiment spannender zu machen, kann man vorher schätzen, ab wann das Ganze nicht mehr machbar ist.)

Trotzdem können wir Funktionen wie *fib* unter geeigneten Randbedingungen in einfachere Form verwandeln. Das soll im Folgenden gezeigt werden.

Allerdings nehmen wir dazu nicht *fib*, sondern eine andere Funktion, die noch etwas kniffliger aussieht. Aber die Entwicklung lässt sich völlig analog auf *fib* übertragen (was wir dem interessierten Leser als Übung überlassen).

Wir hatten weiter vorne schon die etwas artifizielle Funktion *fusc* kennen gelernt, die hier interessant ist, weil sie ein recht komplexes Rekursionsmuster aufweist: In einem Zweig hat sie Tail-Rekursion, im anderen Baum-Rekursion.

$$
\begin{array}{lll}
\text{DEF } fusc(0) & = 1 & \\
\text{DEF } fusc(1) & = 1 & \\
\text{DEF } fusc(2n) & = fusc(n) & -- \textit{ Tail-Rekursion} \\
\text{DEF } fusc(2n+1) & = fusc(n) + fusc(n+1) & -- \textit{ Baum-Rekusion}
\end{array}
$$

Wir wollen zeigen, dass sich auch solche uneinheitlichen Formen sytematisch behandeln lassen. Wir folgen hier einer besonders eleganten Darstellung, die von David Gries präsentiert wurde. Anstatt wie üblich mehrere Hilfsparameter einzuführen, verwenden wir die Eleganz der mathematischen Vektor- und Matrizenrechnung. (Letztlich ist ein Vektor v nichts anderes als eine flexible Kurznotation für mehrere Variablen $v_1, \ldots, v_n$.)

Indem wir die beiden rekursiven Aufrufe in einen Vektor verwandeln, können wir den letzten Rekursionszweig als Skalarprodukt schreiben. Und mit dem kleinen Trick, dass Multiplikation mit 0 einen Wert annulliert, können wir auch den anderen Rekursionszweig in die gleiche Form bringen.[12]

$$
\begin{array}{lll}
\text{DEF } fusc(0) & = 1 & \\
\text{DEF } fusc(1) & = 1 & \\
\text{DEF } fusc(2n) & = (1 \;\; 0) \cdot \begin{pmatrix} fusc(n) \\ fusc(n+1) \end{pmatrix} & -- \textit{ Baum-Rekursion} \\
\text{DEF } fusc(2n+1) & = (1 \;\; 1) \cdot \begin{pmatrix} fusc(n) \\ fusc(n+1) \end{pmatrix} & -- \textit{ Baum-Rekursion}
\end{array}
$$

Jetzt führen wir eine Einbettung ein, indem wir (auf Verdacht) eine Hilfsfunktion f definieren, die gerade der Vektorbildung entspricht.

$$\text{DEF } f(n) = \begin{pmatrix} fusc(n) \\ fusc(n+1) \end{pmatrix}$$

Diese neue Funktion f stützt sich nach wie vor auf *fusc* ab. Deshalb versuchen wir als nächstes, direkte Rekursionsgleichungen für f abzuleiten, die dem Muster der Definition von *fusc* folgen. Für die Terminierungszweige ist das sehr einfach

[12] *Vorsicht*! Diese Definition ist nicht ganz korrekt. Unter Call-by-value würde *fusc*(2) nicht terminieren. Denn die Tatsache, dass der Wert durch die Multiplikation mit 0 annulliert wird, hilft unter dieser Auswertungsstrategie nichts. Aber weil es sich nur um eine Zwischenform handelt, sind wir etwas großzügiger und ignorieren dieses vorübergehende Phänomen.

$$f(0) = \begin{pmatrix} \mathit{fusc}(0) \\ \mathit{fusc}(1) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
$$f(1) = \begin{pmatrix} \mathit{fusc}(1) \\ \mathit{fusc}(2) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Bei den beiden Rekursionszweigen müssen wir etwas mehr rechnen. Wir beginnen mit dem ersten Zweig $f(2n)$:

$$\begin{aligned} f(2n) &= \begin{pmatrix} \mathit{fusc}(2n) \\ \mathit{fusc}(2n+1) \end{pmatrix} && -- \textit{ Definition von } f \\ &= \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n) + \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Definition von fusc} \\ &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Mathematik} \\ &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ Definition von } f \end{aligned}$$

Der andere Zweig $f(2n+1)$ lässt sich ganz analog durchrechen:

$$\begin{aligned} f(2n+1) &= \begin{pmatrix} \mathit{fusc}(2n+1) \\ \mathit{fusc}(2n+2) \end{pmatrix} && -- \textit{ Definition von } f \\ &= \begin{pmatrix} \mathit{fusc}(n) + \mathit{fusc}(n+1) \\ \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Definition von fusc} \\ &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathit{fusc}(n) \\ \mathit{fusc}(n+1) \end{pmatrix} && -- \textit{ Mathematik} \\ &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ Definition von } f \end{aligned}$$

Diese Gleichungen benutzen wir jetzt als Definitionssystem. Dazu müssen wir uns prinzipiell noch davon überzeugen, dass die so eingeführten rekursiven Definitionen terminieren (hier trivial) und dass alle Operationen wohl definiert sind. Der Wert von $\mathit{fusc}(n)$ ist aufgrund der Einbettung gerade die erste Komponente von $f(n)$.

$$\begin{aligned} &\text{DEF } \mathit{fusc}(n) && = f(n).1 \\ &\text{DEF } f(0) && = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &\text{DEF } f(1) && = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &\text{DEF } f(2n) && = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ lineare Rekursion} \\ &\text{DEF } f(2n+1) && = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot f(n) && -- \textit{ lineare Rekursion} \end{aligned}$$

Da die Matrixmultiplikation assoziativ ist und ein neutrales Element hat, könnten wir die entsprechende Transformationregel anwenden. Allerdings würden wir dadurch 2×2-Matrizen als zusätzliche Argumente erhalten, was unnötig aufwendig ist, weil wir uns ja nur für den Wert $f(n).1$, also die erste Komponente des Resultatvektors, interessieren. Deshalb nehmen wir keine Einbettung mit Matrizen vor, sondern nur eine Einbettung mit Vektoren:

$$\text{DEF } \widetilde{f}(n,(z_1,z_2)) \;= (z_1 \;\; z_2) \cdot f(n)$$

Es gibt zwar keinen Vektor $(z_1 \;\; z_2)$, für den diese Multiplikation den Wert $f(n)$ unverändert liefert (dazu ist die Einheits*matrix* notwendig, nicht ein Vektor), aber da uns ja ohnehin nur die erste Komponente interessiert, genügt uns die Einbettung

$$\text{DEF } \mathit{fusc}(n) \;= f(n).1 \;= (1 \;\; 0) \cdot f(n)$$

Für $\widetilde{f}(n)$ können wir unsere Standardrechnung durchführen (was wir dem interessierten Leser als Übung überlassen), so dass am Ende das folgende System entsteht:

$$\begin{array}{ll}
\text{DEF } \mathit{fusc}(n) & = \widetilde{f}(n,(1,0)) \\
\text{DEF } \widetilde{f}(0,(z_1,z_2)) & = z_1 + z_2 \\
\text{DEF } \widetilde{f}(1,(z_1,z_2)) & = z_1 + z_2 \\
\text{DEF } \widetilde{f}(2n,(z_1,z_2)) & = \widetilde{f}(n,\,(z_1+z_2,\,z_2)) \\
\text{DEF } \widetilde{f}(2n+1,(z_1,z_2)) & = \widetilde{f}(n,\,(z_1,\,z_1+z_2))
\end{array}$$

Natürlich hätten wir diese weitere Einbettung auch gleich bei der ersten Hälfte der Entwicklung mit einbauen können, so dass wir direkt von der Baum-Rekursion zur Tail-Rekursion gekommen wären, ohne die lineare Rekursion zwischenzuschalten. Aber das hätte die Präsentation wesentlich unleserlicher gemacht.

Diese kleinen Beispiele sollten hinreichend illustrieren, wie gut man mit funktionalen Programmen arbeiten kann. Dabei ist es prinzipiell egal, ob diese Umformungen „von Hand“ erfolgen oder automatisch im Compiler.

1.5 OPAL, ML und HASKELL

Die meisten funktionalen Sprachen sind sich relativ ähnlich und lassen im Allgemeinen alle hier diskutierten Formen der Notation zu. Meist wird die gleichungsartige Definition von Funktionen gegenüber der λ-Notation als Normalfall vorgezogen. Natürlich gibt es aber syntaktische Unterschiede bzgl. der Verwendung von Symbolen und Schlüsselwörtern.

Beispiel: Wie man in OPAL ein Filter-Funktional definiert und auf eine anonyme Funktion und eine Liste anwendet, haben wir schon gesehen. In HASKELL und ML sieht das bis auf kleine syntaktische Unterschiede im Prinzip genauso aus. In HASKELL schreibt man den Typ der Sequenzen von Elementen eines Typs a als $[a]$ und unsere Operation „ $\cdot$: “ einfach als „: “. Die

Typisierung wird mit „ :: “ notiert. Damit sieht *filter* dann folgendermaßen aus:

```
filter :: ( a → Bool ) → [a] → [a]                    -- HASKELL-Notation
filter p [] = []
filter p (x: xs) = if p x then x: (filter p xs) else filter p xs
```

Eine Anwendung hat dann z. B. folgende Form:

```
filter(\ x → x > 0) [3, 5, −2, 0, 18, −12]            -- HASKELL-Notation
```

In ML sieht das Ganze folgendermaßen aus:

```
fun filter p [] = [] |                                 -- ML-Notation
    filter p (x :: xs) = if p x then x :: (filter p xs) else filter p xs
```

Eine Anwendung hat dann z. B. folgende Form:

```
filter (fn x => x > 0) [3, 5, −2, 0, 18, −12]          -- ML-Notation
```

Bestimmte Standardfunktionen und -typen, die sich wie in den meisten Programmiersprachen in vordefinierten Strukturen (OPAL, ML) oder Modulen (HASKELL) befinden, können unterschiedliche Namen und Varianten haben, auch wenn sich in vielen Fällen hierfür feste Bezeichner eingebürgert haben. Beispielsweise wird das Reduce-Funktional $\not\prec$ in HASKELL und ML als *foldr* bezeichnet; die Reduce-Version für die inverse Richtung $\not\succ$ heißt dort *foldl*. Der Datentyp *option* aus der OPAL-Struktur *Option* ermöglicht es, mit optionalen Werten umzugehen und damit mit der Situation, dass eine Funktionsauswertung auch fehlschlagen kann. HASKELL und ML haben entsprechende Datentypen *Maybe* bzw. *option* mit gleicher Wirkung.

Während man in OPAL bei der Deklaration einer Funktion den Typ explizit angeben muss und das System prüft, ob der Rumpf typkorrekt ist, können in HASKELL und ML Typangaben (in den meisten Fällen) weggelassen werden, da das System diese inferiert.

Wie schon oben erwähnt, können in OPAL Strukturen polymorph sein, d. h. polymorphe Datentypen und Funktionen enthalten. Diese kann man dann im Programm mit geeigneten Datentypen instanziieren. Im Gegensatz dazu wird in ML und HASKELL Polymorphie nicht global für ganze Strukturen festgelegt, sondern individuell für jeden Typ und jede Funktion einzeln.

Auf einige dieser Unterschiede werden wir in späteren Kapiteln noch genauer eingehen.

http://www.springer.com/978-3-540-20959-1

Funktionale Programmierung
Sprachdesign und Programmiertechnik
Pepper, P.; Hofstedt, P.
2006, XVIII, 492 S. 57 Abb., Softcover
ISBN: 978-3-540-20959-1