1

Einleitung

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

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

Daneben wurden eine Reihe von anderer Programmierstile entwickelt:

Beim applikativen bzw. funktionalen Stil ist die Funktionsanwendung das beherrschende Sprachelement. Bereits früh entstand als erster Vertreter dieses Programmierstils die Sprache LISP. Sie hat inzwischen eine Reihe von moderneren Nachfolgesprachen gefunden. Dieser Programmstil hat sich insbesondere im Bereich der symbolischen Informationsverarbeitung und der Künstlichen Intelligenz (KI) durchgesetzt und findet zunehmend auch in der industriellen Praxis Verwendung.
Beim prädikativen (logischen) Stil ist das Formulieren von prädikatenlogischen Formeln das beherrschende Sprachelement. Er wird ebenfalls im Bereich der „Künstlichen Intelligenz“ vor allem bei der Entwicklung von Expertensystemen, eingesetzt. Die bekannteste Sprache ist hierbei PROLOG.
Ebenfalls in diesem Bereich ist der objektorientierte Programmierstil entstanden, bei dem die Definition von Objekten mit Fähigkeiten zum Senden und Empfangen von Nachrichten im Vordergrund steht. Der bekannteste Vertreter ist die Sprache JAVA.

Viele Programmiersprachen unterstützen nur einen Programmierstil, z.B. FORTRAN den prozeduralen oder PROLOG den prädikativen. Von zunehmender Bedeutung sind aber auch Sprachen, die die Benutzung mehrerer Stile gestatten. In PASCAL programmiert man zwar üblicherweise prozedural; man

kann aber auch den applikative Programmierstil einsetzen. In der Sprache LISP programmiert man überwiegend applikativ; es stehen aber auch der objektorientierte und der prozedurale Programmierstil zur Verfügung. Die Integration des prädikativen Stils bereitet prinzipielle Probleme und ist immer noch aktueller Gegenstand der Forschung.

Eine vergleichende Wertung verschiedener Programmierstile wird im Rahmen dieses Buches nicht erfolgen. Im Vordergrund steht vielmehr die Vermittlung der Ideen des funktionalen und applikativen Programmierens, ihre programmiersprachlichen Ausprägungen, kleinere Anwendungsbeispiele und ein Einblick in spezielle Implementierungstechniken für deratige Sprachen. Die Verbindung zwischen den einzelnen Abschnitten erfolgt über die Theorie der rekursiven Funktionen, den ungetypten λ-Kalkül und die kombinatorische Logik.

Wenden wir uns nun einer Klärung und Abgrenzung der Begriffe „applikatives“ und „funktionales“ Programmieren zu:

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

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

$$f(a) + g(a) = g(a) + f(a).$$

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

```
program P (output);
var       a : integer;
function f(x : integer) : integer;
    begin a := x + 1; f := a end;
function g(x : integer) : integer;
    begin a := x + 2; g := a end;
begin
    a := 0; write(f(a) + g(a));
    a := 0; write(g(a) + f(a));
end;
```

Man erhält verschiedene Ausgaben:

$$f(a) : a = 1, \quad f(a) = 1$$
$$g(a) : a = 3, \quad g(a) = 3$$
$$f(a) + g(a) = 1 + 3 = 4$$

bzw.

$$g(a) : a = 2, \quad g(a) = 2$$
$$f(a) : a = 3, \quad f(a) = 3$$
$$g(a) + f(a) = 2 + 3 = 5$$

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

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

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

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

Die Adjektive „applikativ“ bzw. „funktional“ werden von verschiedenen Autoren in recht unterschiedlicher Bedeutung benutzt. Einige betrachten beide Begriffe als Synonyme, andere verstehen hierunter zwei unterschiedliche Programmierstile. Daher soll zunächst eine Klärung dieser Begriffe gegeben werden.

Das Wort „applikativ“ kommt vom lateinischen 'applicare' ($\approx$ anwenden). Applikatives Programmieren ist somit Programmieren, bei dem das tragende Prinzip zur Programmerstellung die Funktionsapplikation, d.h. die Anwendung von Funktionen auf Argumente, ist. Das Wort „funktional“ kommt vom mathematischen Begriff des Funktionals bzw. höheren Funktionals, d.h. Funktionen, deren Argumente oder Ergebnisse wieder Funktionen sind. Funktionales Programmieren ist somit Programmieren, bei dem das tragende Konzept zur Programmerstellung die Bildung von neuen Funktionen aus gegebenen Funktionen mit Hilfe von Funktionalen ist.

Geht man von diesen beiden Definitionen aus, die sich nur an der ursprünglichen Bedeutung der Begriffe „applikativ“ und „funktional“ orientieren, so sieht man unmittelbar ihre Gemeinsamkeit. Läßt sich eine Funktion auf ein Argument, welches selbst eine Funktion ist, anwenden, so ist sie ein Funktional.

Die Bildung einer neuen Funktion aus gegebenen Funktionen mit Hilfe eines Funktionals ist nichts anderes als die Applikation (Anwendung) dieses Funktionals. Andererseits gibt es auch gute Gründe dafür, zwischen den Begriffen „applikativ“ und „funktional“ zu differenzieren, wenn man mit „applikativ“ das Operieren auf elementaren Daten (z.B. ganzen Zahlen) charakterisiert bzw. mit „funktional“ das Operieren auf Funktionen.

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

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

Zunächst sei die Entwicklung eines zugehörigen Programms an einer Vorgehensweise charakterisiert, die über elementaren Daten operiert.

Wir führen eine Variable x ein, die eine natürliche Zahl als Wert besitzt. Die Fallunterscheidung wird durch das Resultat der Applikation der Funktion

$$null : \mathbb{N} \to \{true, false\}$$

gesteuert, d.h. durch $null(x)$.

Durch Applikation der Vorgängerfunktion

$$Vorg : \mathbb{N} \to \mathbb{N}_0,$$

der Multiplikation

$$Mult : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$

und durch rekursive Applikation von Fak erhält man wieder einen Ausdruck, der einen elementaren Wert aus $\mathbb{N}$ besitzt. Der bedingte Ausdruck ist hier eine dreistellige Funktion

$$\{true, false\} \times \{1\} \times \mathbb{N} \to \mathbb{N}.$$

In Infix-Notation hat man den Ausdruck

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

Er präsentiert für einen gegebenen Wert x die natürliche Zahl $x!$ und ist keine Funktion. Erst durch explizite Abstraktion nach der Variablen x erhält man eine Funktion. In diesem Zusammenhang ist Church's λ-Notation gebräuchlich:

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

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

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

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

$$
\begin{aligned}
& Fak \\
&= Null(3) \to 1; Mult(3, Fak(Vorg(3))) \\
&= Null(3) \to 1; Mult(3, Fak(2)) \\
&= Null(3) \to 1; Mult(3, Null(2) \to 1; Mult(2, Fak(Vorg(2)))) \\
&= Null(3) \to 1; Mult(3, \underline{false} \to 1; Mult(2, Fak(Vorg(2)))) \\
&= Null(3) \to 1; Mult(3, \overline{Mult}(2, Fak(1))) \\
&= Null(3) \to 1; Mult(3, Mult(2, Null(1) \to 1; Mult(1, Fak(Vorg(1))))) \\
&= Mult(3, Mult(2, Mult(1, Fak(0)))) \\
&= Mult(3, Mult(2, Mult(1, Null(0) \to 1; Fak(Vorg(0))))) \\
&= Mult(3, Mult(2, Mult(1, 1))) \\
&= Mult(3, Mult(2, 1)) \\
&= Mult(3, 2) \\
&= 6
\end{aligned}
$$

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

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

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

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

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

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

$$Fak \circ Vorg\ .$$

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

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

Die Konstruktion ist hierbei definiert durch

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

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

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

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

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

und benennt sie

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

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

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

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

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

Funktionales Programmieren (im strengen Sinne)

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

Applikatives Programmieren (im strengen Sinne)

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

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

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

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

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

http://www.springer.com/978-3-540-89091-1

Funktionale und Applikative Programmierung
Grundlagen, Sprachen, Implementierungstechniken
Lippe, W.
2009, X, 353 S., Hardcover
ISBN: 978-3-540-89091-1