Renaissance funktionaler Programmierung

Comeback

Rainer Fischbach

In der Informatik gibt es Entwicklungen, die nicht weithin sichtbar und anerkannt erfolgreich sind, doch dessen ungeachtet eine heimliche Erfolgsgeschichte schreiben, indem sie anderen entscheidende Impulse verleihen. So auch die funktionale Programmierung, die in letzter Zeit durch Scala und F# wieder größere Aufmerksamkeit erfährt.

Obwohl die Zahl der Anhänger funktionaler Programmierung unter einem Prozent aller Entwickler liegen dürfte, ist die Programmierwelt voll von Dingen, die in jener ihren Ursprung haben. Die IT-Expansion der letzten Jahrzehnte dürfte zusammen mit dem relativen Bedeutungsverlust des klassischen Informatikstudiums dazu geführt haben, dass der Anteil derer, die auch nur einen vagen Begriff von funktionaler Programmierung haben, geschwunden ist und weiter schwindet. Mit dem Siegeszug von C++, C# und Java sowie von Skriptsprachen wie Perl und Python ließ das Interesse an Alternativen nach. Zu den Opfern der Entwicklung zählten neben den vom imperativen Modell abweichenden Ansätzen auch die objektorientierten Minderheitensprachen wie Smalltalk und Eiffel. Zudem befinden sich die diversen Lisp-Dialekte, die lange als Einbettungs- und Erweiterungssprachen, etwa in der CAD-Welt, eine starke Rolle spielten, auf dem Rückzug.

Wie kaum eine andere Form des Programmierens ist die funktionale mit der theoretischen Informatik sowie der Künstlichen Intelligenz verbunden. Sie steht im Ruf, für die Masse der Programmierer eher unzugänglich zu sein. Letztere verwenden die oben genannten Sprachen, die alle in der IT wenigstens beim Namen kennen, während Lisp [13], Caml [3] oder Haskell [4] für die meisten „böhmische Dörfer“ blieben. Ob funktionale Newcomer wie Scala [9] und F# [11], die immerhin den Anschluss an die Hauptströmung der IT durch ihre Einbettung in die dort dominierenden Java- beziehungsweise .Net-Plattformen suchen, den Zustand überwinden, bleibt abzuwarten. Jedenfalls ist wieder ein zunehmendes Interesse an der funktionalen Programmierung zu verzeichnen, bei dem man anders als in den frühen 1990ern damit rechnen kann, dass den theoretischen Vorzügen der funktionalen Programmierung zumindest in Teilbereichen praktische Relevanz zugestanden wird. Dass ein aktueller Band [2], der beansprucht, bedeutende Programmiersprachen in Interviews mit ihren Schöpfern vorzustellen, mit ML [10] und Haskell [4] zwei wichtige funktionale Sprachen einschließt, lässt hoffen.

Es ist zwar nur ein fundamentales Merkmal, das funktionale Programme von den anderen unterscheidet, die hier unter Ausschluss der logischen, doch unter Einschluss der objektorientierten zusammenfassend als imperative bezeichnet seien. Es hat jedoch eine so große Tragweite, dass es gleich eine Reihe weiterer Unterschiede hervorbringt. Die Geschichte der funktionalen Programmierung und Sprachen bestand zum großen Teil in der Herausarbeitung der Konsequenzen, die sich daraus ergeben, dass der funktionale Ansatz Fragestellungen in den Vordergrund rückt, die beim imperativen eher verdeckt bleiben.

Was ist funktionale Programmierung?

Imperative Programme bestehen, dem Von-Neumann-Modell des Computers folgend, aus Anweisungen an einen Prozessor, die einen gegebenen, die Aufgabe repräsentierenden Ausgangszustand des Speichers schrittweise verändern, bis der gewünschte, das Ergebnis repräsentierende Endzustand erreicht ist. Ein funktionales Programm hingegen besteht aus einem Ausdruck, der auszuwerten ist, um das Ergebnis zu erhalten. Genauer gesagt stellt der Ausdruck eine Repräsentation des Ergebnisses dar – so bezeichnet ‚1 + 1‘ nichts anderes als den Ausdruck ‚2‘ – nur ist das eben nicht die kanonische Repräsentation des Werts. Seine Auswertung leistet genau das: Sie produziert die kanonische Repräsentation, etwa ‚2‘ aus ‚1 + 1‘. Der funktionale und der logische Ansatz gleichen sich im Anspruch, dass Programme ausführbare Spezifikationen sein sollten, in denen weniger festzuhalten sei, wie man zum Ergebnis komme, sondern mehr, was das Ergebnis sei [12]. Während logische Programme die Eigenschaften des Ergebnisses durch Prädikate spezifizieren, konstruieren funktionale einen Ausdruck, der es bezeichnet und auf seine kanonische Form zu reduzieren ist.

Ein imperatives Programm, das die Fakultät von 10 berechnet, sieht in einer Syntax mit Abseitsregel (Gliederung durch Einrückung statt durch Begrenzer) so aus:

```
x = 10
fak = 1
while x > 0:
    fak = fak * x
    x = x - 1
```

Unter der Voraussetzung, dass *x* eine natürliche Zahl ist, enthält die Variable *fak* nach Abbruch der Schleife die Fakultät von *x*. Die Berechnung vollzieht sich durch Seiteneffekte auf den Speicher, nämlich die Änderung der Variablen *x* und *fak*. Das funktionale Programm ist frei von Seiteneffekten:

```
let
    fak 0 = 1
    fak x = x * fak (x - 1)
in fak 10
```

Die verwendete Notation lehnt sich an Haskell [4] an, und die Definition der Funktion *fak* spiegelt die mathematische. Parameter benötigen keine Klammern. Das Konstrukt *let* … *in* … bildet einen lexikalischen Abschluss. Er umfasst eine Menge von Bindungen von Werten an Namen, deren Gültigkeitsbereich der durch die Schlüsselworte markierte Textbereich plus der auf *in* folgende Ausdruck ist und deren Lebensdauer von der Verwendung des letzteren abhängt. Die Bindungen führen Namen ein, die innerhalb desselben Gültigkeitsbereichs für denselben Teilausdruck beziehungsweise seinen Wert stehen. Solche Bereiche sind in den heutigen funktionalen Sprachen statisch, also durch den Quelltext definiert. Das gilt auch für die modernen Lisp-Versionen wie Common Lisp und Scheme, bei deren Vorfahren die Gültigkeitsbereiche noch vom dynamischen Kontext abhingen.

Das Konzept einer Variablen, deren Wert sich durch Zuweisungen änderte, spielt keine Rolle. An die Stelle der Schleife, die das Ergebnis schrittweise durch Zuweisungen aufbaut, tritt die Rekursion, also die Verwendung der zu definierenden Funktion mit neuen Argumenten. Das ist allerdings nur legitim, wenn die Kette der rekursiven Aufrufe abbricht. Dazu sind nach endlich vielen Schritten Argumente zu übergeben, für welche die Funktion unmittelbar definiert ist, hier *0*.

Die Eleganz von Programmen, die unmittelbar mathematische Definitionen widerspiegeln, hat als Kehrseite oft einen höheren Speicherbedarf und eine erhöhte Laufzeit. Es bleibt zum Beispiel die ganze Folge der unausgewerteten Ausdrücke mit den rekursiven Aufrufen und den dazugehörigen Parametern im Stack stehen, bis alle vom terminalen Fall für den Parameterwert *0* aus rückwärtsschreitend ausgewertet sind. Setzt man etwa in naiver Weise die rekursive Definition der Fibonacci-Zahlen in ein funktionales Programm um, kommt noch dazu, dass dieselben Werte wiederholt berechnet werden. Doch lassen sich solche Ineffizienzen meist vermeiden, indem man das Programm in die sogenannte endrekursive Fassung bringt:

```
let
    fak x = fake 1 x
    where
        fake f 0 = f
        fake f n = fake (n * f) (n - 1)
in fak 10
```

Das Konstrukt … *where* … führt die Bindungen nach *where* in die davorstehenden Definitionen ein. Die Syntax, die der Diktion mathematischer Traktate entspricht, stellte Peter Landin 1965 mit ISWIM [6] (steht für „If you See What I Mean“) vor, das als Wurzel der modernen funktionalen Sprachen gilt. Dort tauchte zuerst die Abseitsregel auf, die Gliederung durch Einrücken, derer sich Occam, Miranda und Python bedienen.

Der zusätzliche Parameter *f* der Hilfsfunktion akkumuliert das Ergebnis analog zur Variablen *fak* in der imperativen Lösung. Moderne Compiler erkennen die Endrekursion und eliminieren sie. Das heißt, sie ersetzen sie durch eine Schleife. Es sind keine unvollständig ausgewerteten Ausdrücke zu speichern, weil die Rekursion nicht als Teilausdruck vorkommt, sondern unmittelbar das Ergebnis liefert. Endrekursive Programme stehen deshalb imperativen in der Effizienz nicht nach.

Da funktionale Programme keine Seiteneffekte haben, durch die etwa eine Komponente den Zustand einer anderen veränderte, hat ein bestimmter Ausdruck immer dieselbe Bedeutung – unabhängig von seiner Position und der der Evaluationsfolge. Die Eigenschaft der referenziellen Transparenz macht sie grundsätzlich verständlicher als imperative und bietet bessere Voraussetzungen sowohl für den Einsatz von Techniken der formalen Programmverifikation als auch die Ausführung auf parallelen Architekturen. Die inhärente Sequenzialität von imperativen Programmen erschwert das zumindest, wenn die Abhängigkeit der Ergebnisse von einer bestimmten Ausführungsfolge sie nicht sogar verhindert. John Backus prägte in seiner Turing Award Lecture von 1978 [1] dafür den Begriff Von-Neumann-Flaschenhals. Das Potenzial zu seiner Überwindung bildet immer

Glossar

Concurrent Clean: An der Universität Nijmegen entwickelte funktionale Sprache mit nicht strikter Evaluation in der Tradition von Miranda, deren Schöpfer sich besonders der Implementierung auf parallelen Rechnerarchitekturen widmeten und zu diesem Zweck Alternativen zu den bis dahin vorherrschenden Kompilationstechniken entwickelten. Dabei zielten sie insbesondere auf die massiv parallelen Architekturen von Parsytec auf der Basis des Inmos-Transputers.

Inmos: Die britische Halbleiterfirma stellte in den 1980ern mit dem Transputer eine Prozessorarchitektur vor, die auf den Aufbau von parallelen Rechnerarchitekturen ausgelegt war. Transputer konnten über spezielle Links kommunizieren und wurden mit der Sprache Occam programmiert, die auf der durch C. A. R. Hoare entwickelten Logik kommunizierender Prozesse aufbaute. Inmos wurde 1989 an die französische SGS Thomson verkauft und erlosch 1994 als Marke. Verzögerungen bei der Einführung fortgeschrittener Transputermodelle (T9000) und die mangelnde Akzeptanz von Occam führten schließlich Mitte der 1990er zum Scheitern der zunächst durch Thomson weitergeführten Produktlinie.

Oberon: Nach Modula-2 eine weitere Schöpfung des Pascal-Schöpfers Niklaus Wirth. Er nähert sich darin durch ein erweitertes Record-Modell vorsichtig der Objektorientierung. Records und Pointer auf Records sind explizit unterschiedene Typen. Klassen als tragende Einheiten der Programmstruktur gibt es nicht.

Parsytec: Der Aachener Spezialist für massiv parallele Rechnerarchitekturen, der seine Rechner ursprünglich auf der Basis von herkömmlichen Prozessoren aufgebaut hatte, versprach sich vom Inmos-Transputer den entscheidenden Durchbruch und wurde so ein Opfer der verspäteten Markteinführung des fortgeschrittenen Modells T9000.

noch eines der Motive für ein beständiges Interesse an funktionaler Programmierung, wenn sich auch frühere Erwartungen an Entwicklungen wie Concurrent Clean infolge des ökonomischen Scheiterns der parallelen Architekturen von Inmos und Parsytec nicht erfüllten. Erfolge konnte dagegen Erlang mit funktionaler Parallelität in der Tabellenkommunikation feiern.

Der Sachverhalt, dass Programme dort Ausdrücke sind, die für Werte stehen, rückte das Thema der Datentypen und -strukturen von Anfang an in den Vordergrund der funktionalen Programmierung. Die Funktionsdefinition durch Gleichungen, die moderne funktionale Sprachen ermöglichen, ist tatsächlich nur „syntaktischer Zucker" über einer weiter unten vorgestellten elementaren Notation, die Funktionen unmittelbar durch eine spezielle Ausdrucksform, die sogenannte Lambda-Abstraktion, spezifiziert. Der durch die Gleichungen eingeführte Funktionsname ist ein elementarer Ausdruck, der für ein Funktionsobjekt steht, und ein solches ist ein Datenobjekt, das man wie eine Zahl in eine Liste stellen oder als Parameter an eine andere Funktion übergeben kann, die wiederum eine Funktion als Resultat liefert. Dass Funktionen Datenobjekte sind, zeichnet die funktionalen Sprachen aus und macht sie zugleich für Programmierer, die mit der dogmatischen Trennung von Daten und Programmen aufgewachsen sind, so schwer. Funktionen, die Funktionen als Parameter nehmen beziehungsweise als Resultate liefern, heißen Funktionale oder „high order functions". Der Begriff „Funktional" entstammt der Funktionalanalysis, die Funktionen als Punkte und Vektoren in Räumen auffasst. Ein klassisches Beispiel ist die Faltung, welche die Elemente einer Liste mit einer binären Funktion verknüpft. Listen mit Kopf-Schwanz-Struktur kennt die funktionale Welt seit Lisp [13], der ersten funktionalen Programmiersprache. Haskell verwendet für solche Listen, die in Lisp mit der *cons*-Funktion zu bilden sind, die Schreibweise *h:t* mit dem Infix-Konstruktor ‚:'. Es bietet jedoch auch Literale der Form *[1,2,3]* statt *1:2:3:[]*. Die Faltungsfunktion benötigt außer der Verknüpfungsfunktion noch einen Initialwert, der das Ergebnis für eine leere Liste definiert:

```
lfold f ini []    = ini
lfold f ini (h:t) = lfold f (f ini h) t
```

Die Klammern um *h:t* sind nötig, weil die Funktionsapplikation stärker bindet als der Listenkonstruktor und die Klammern um *f ini h*, weil sie linksassoziativ ist. Wie man sieht, lassen sich Konstruktoren auf der formalen Parameterposition verwenden: Sie definieren ein Muster, mit dem der aktuelle Parameter abzugleichen und zu zerlegen ist. *lfold* repräsentiert einen Wert – ein Funktionsobjekt. Es ist ein Funktional, das, angewandt auf eine passende Funktion, eine neue Funktion liefert:

```
let sum = lfold (+) 0
```

definiert *sum* als die Funktion, welche die Elemente einer Liste addiert. Die Notation *(+)* erzeugt aus dem Infix-Operator ‚+' eine in Präfix-Schreibweise anzuwendende Funktion. Die Applikation *sum [1,2,3]* liefert das Ergebnis 6. Funktionen mit mehreren Parametern kann man also schrittweise anwenden, wobei die Zwischenschritte neue Funktionen liefern, die man auf die folgenden Parameter appliziert. Solche Funktionen nennt man „curried functions" nach Haskell B. Curry, der zum Namensgeber von Haskell wurde.

Die Listenfunktionale *map* und *filter*, welche die meisten funktionalen Sprachen als Primitive anbieten, sind folgendermaßen definiert:

```
map f []    = []
map f (h:t) = f h : map f t
filter p []          = []
filter p (h:t) | p h      = h : filter p t
               | otherwise = filter p t
```

map wendet die Funktion *f* auf alle Elemente einer Liste an und liefert die Liste der Ergebnisse, während *filter* nur die Elemente einer Liste in der Ergebnisliste liefert, die das Prädikat *p* erfüllen.

Listen mit freier Länge waren eine Innovation, die ihren Weg in viele Programmiersprachen fand. Sie verlangen eine automatische Speicherverwaltung, die sich bei ihrem Lebensende um ihre Beseitigung kümmert. Auch das ist ein Standard aus der funktionalen Welt, der erst langsam den Weg in die imperative fand. Objektorientierte Sprachen folgten nach.

Auf den Spuren tief in der Vergangenheit

Lisp-Objekte beziehungsweise Variablen haben zwar einen Typ, doch muss er nicht statisch festgelegt sein. Lisp-Programme modellieren alle Datenstrukturen durch Listen. Die Assoziationslisten, deren Elemente Paare von Werten sind, bildeten den Ausgangspunkt für Hash-basierte Implementierungen, die heute als Dictionaries eine Schlüsselrolle in Skriptsprachen spielen. Lisp-Programme haben selbst die Form von Listen und können sich zur Laufzeit verändern beziehungsweise erweitern, indem sie als Listen elaborierte Programmteile mit der *eval*-Funktion ausführen. Lisp steht damit am Anfang einer Tradition dynamischer Sprachen, die sich unter anderem in Smalltalk und Python fortsetzte.

Lisp, ISWIM und weitere funktionale Sprachen knüpfen an Traditionen an, die in die Zeit vor der Erfindung der elektronischen Digitalrechner zurückreichen. Der erwähnte Curry formulierte in den 1930ern den Kombinatorenkalkül von Neuem, den Moses Schönfinkel 1924 publiziert hatte. Die zweite Wurzel bildet der ebenfalls in den 1930ern von Alonzo Church und Stephen Kleene erfundene Lambda-Kalkül [7]. Schönfinkel, Church, Kleene und Curry ging es damals ebenso wie Turing, der um die Zeit das Modell eines Rechenautomaten entwarf, um mathematische Grundlagenprobleme wie die von Entscheidbarkeit und Berechenbarkeit. Der Kombinatoren- und der Lambda-Kalkül sind rein funktional. Es gibt keine primitiven Datenobjekte wie Zahlen, sondern nur Funktionen. Zahlen lassen sich dort erst durch Funktionen als sogenannte Church-Numerale modellieren. Der Lambda-Kalkül führt eine Notation für funktionswertige Ausdrücke ein, die – mit Varianten – Bestandteil jeder funktionalen Sprache und einiger anderer ist, etwa von Python, das eine umfangreiche, jedoch unselbstständige funktionale Subsprache enthält. Das erlaubt die Definition von anonymen Funktionen. In dem Haskell-Ausdruck

```
filter (\ x -> x > 100) [1, 101, 3, 1001]
```

steht der Teilausdruck in runden Klammern für die Funktion, die testet, ob ihr Argument größer als *100* ist. Die Lambda-Abstraktion bildet aus dem Ausdruck *x > 100* eine Funktionsdefinition, indem sie die Variable *x* universell bindet, das heißt zu einem Parameter werden lässt; wobei Haskell das Symbol „\" anstelle des von Lisp bekannten Schlüsselworts „lambda" verwendet. Wenn *x* eine (freie) Variable und *E* ein Ausdruck ist, dann bezeichnet *lambda x. E* die Funktion, die für jeden (korrekt getypten) Wert *e* den Wert von *E* hat, der sich durch Einsetzen von *e* für *x* ergibt. Die Anwendung von *filter* auf den Lambda-Ausdruck liefert eine Funktion, welche die

Übersicht funktionaler Programmiersprachen

primär	abgeleitet	Schöpfer/Jahr	abgeleitet von/beeinflusst durch	Merkmale
Lisp	–	John McCarthy 1958	Lambda-Kalkül	listenbasierte Syntax, dynamisches Typsystem, dynamische Listen, dynamische Gültigkeitsbereiche, automatische Speicherverwaltung, ergänzende imperative Merkmale
–	Common Lisp	Kommittee 1984, 1994	Lisp, Objektorientierung	statische Gültigkeitsbereiche, großer Sprachumfang, mächtiges Objektsystem (CLOS m. mehrfachem Erben und Multimethoden)
–	Scheme	Gerald Sussman/Guy Steele 1975	Lisp, Pascal	statische Gültigkeitsbereiche, Einfachheit, Continuations
ISWIM	–	Peter Landin 1965	Lambda-Kalkül, ALGOL	elegante Syntax, Abseitsregel, statische Gültigkeitsbereiche, strikte Evaluation
Hope	–	Rod Burstall, Dave MacQueen, Don Sanella 1980	ISWIM, Universelle Algebra	mächtiges statisches Typsystem, algebraische Datentypen
ML	–	Robin Milner 1973	ISWIM, Universelle Algebra	elegante Syntax, strikte Evaluation, mächtiges statisches Typsystem (Mindley-Milner), ergänzende imperative Merkmale, automatische Speicherverwaltung
–	SML	Robin Milner et al. 1990	ML, Clu, Modula	mächtiges Modulsystem, algebraische Datentypen, abstrakte Datentypen, formale Semantik
–	Caml	ENS/INRIA 1985	ML	algebraische Datentypen
–	Camllight	–	Caml	einfache Teilmenge von Caml
–	OCaml	–	Caml, Objektorientierung	ergänzende objektorientierte Merkmale
–	F#	Don Syme (MS Reserch) 2002	OCaml	–
Erlang	–	Joe Armstrong, Ericsson 1987	Lisp, Prolog	parallele Prozesse (Actors, Synchronisation durch Message-Parsing)
Leda	–	Timothy Budd 1995	Modula, Prolog, Eiffel, Scheme	vereinigt objektorientierte, funktionale und logische Programmierung, wahlweise strikte und nicht strikte Evaluation
Pizza	–	Martin Odersky 1997	Java, ML	Obermenge von Java, mächtiges statisches Typsystem (lokal Mindley-Milner + objektorientiert)
Scala	–	Martin Odersky 2004 (EPFL)	ML, Java, Objektorientierung	konventionelle Syntax, strikte und optional nicht strikte Evaluation, mächtiges statisches Typsystem (lokal Mindley-Milner + objektorientiert + beschränkte parametrische Polymorphie + Traits) ergänzende imperative Merkmale, automatische Speicherverwaltung
SASL	–	David Turner 1972/176	ISWIM	ursprünglich strikte Evaluation, Neuimplementierung mit nicht strikter Evaluation
Miranda	–	David Turner 1983	SASL, Lambda-Kalkül, Kombinatorenkalkül, ML	elegante Syntax (Abseitsregel + ZF-Notation), nicht strikte Evaluation, mächtiges statisches Typsystem (Mindley-Milner), rein funktional, automatische Speicherverwaltung
–	Haskell	Komitee 1990	Miranda	Typ-Klassen
–	Gofer	–	Haskell	einfache Teilmenge von Haskell
–	Concurrent Clean	Univ. Nijmegen 1987	Miranda	für parallele Architekturen

Elemente aus einer Liste liefert, die größer als 100 sind.

Haskell bietet ein von Miranda übernommenes, an die Schreibweise der Mengenlehre (ZF-Notation) angelehntes Konstrukt, das *map* und *filter* in eleganter Weise zu kombinieren gestattet:

```
[x * x | x <- ls, x > 100]
```

liefert die Liste der Quadrate aller Elemente aus der Liste *ls*, die größer als 100 sind. Die Notation, die inzwischen Einzug in weitere Sprachen fand, ist nicht das einzige, was Haskell und Miranda auszeichnet. Anders als Lisp und die ML-Sprachen werten jene die aktuellen Funktionsparameter nur bei Bedarf aus, also wenn sie tatsächlich in die Berechnung eingehen. Das nicht strikte, „faule“ Verhalten weisen in den meisten Sprachen nur die binären logischen Operatoren auf. Auf den Streit „faule“ versus strikte Evaluation, der viele wissenschaftliche Papiere hervorgebracht hat, soll der Artikel allerdings nicht weiter eingehen.

Der Lambda-Kalkül bildet nicht nur den Kern aller funktionalen Sprachen, sondern spielt zudem eine Schlüsselrolle in der Theorie der Datentypen – auch der objektorientierten. Die Theorie bildet den Ausgangspunkt einer Logik der Programmkorrektheit beziehungsweise ihres Beweises. Die Generation von funktionalen Sprachen, die seit den 1970er-Jahren entstand, ging aus dem Kontext hervor. ML (Meta Language) [10], das Robin Milner in Edinburgh als Basis des Beweissystems LCF (Logic for Computable Functions) entwickelte, ist heute in mehreren Varianten wie SML (Standard ML), Caml [3] beziehungsweise OCaml (Categorial abstract Machine Language aus dem INRIA Paris) und F# (für die .Net-Plattform aus dem Microsoft Lab im englischen Cambridge) präsent.

Eine zentrale Rolle spielt in ML das statische Typsystem, das jedoch nicht zu den expliziten Typdeklarationen zwingt, die viele Programmierer lästig finden: Der Compiler basiert auf der Logik der Datentypen und kann den Typ von Ausdrücken beziehungsweise Funktionen ableiten. SML führt zudem ein Modulsystem ein, das abstrakte Datentypen mit einer strikten Trennung von Signatur und Implementierungen zu definieren erlaubt.

In dem als Hindley-Milner Typing bekannten Typsystem von ML und den anderen darauf aufbauenden Sprachen wie Haskell spielt parametrische Polymorphie eine Schlüsselrolle. Der allgemeinste Listentyp ist dort *[alpha]*, wobei *alpha* der Typparameter ist, während etwa eine Liste von Ganzzahlen wie *[1,2,3]* den Typ *[int]* hat. Das Funktional *map* hat den Typ

```
(alpha -> beta) -> [alpha] -> [beta]
```

Es bildet eine Funktion von *alpha* nach *beta* in eine Funktion ab, die eine Liste von Elementen mit dem Typ *alpha* in eine von Elementen des Typs *beta* abbildet.

Parametrische Polymorphie ist analog zur universellen Quantifikation in der Prädikatenlogik zu verstehen und bedeutet so viel wie „für alle Typen alpha“. Damit lässt sich ein Problem behandeln, das in objektorientierten Sprachen, die nur Inklusionspolymorphie kennen, Ursache beständigen Ärgers ist. Der Typ einer Liste von Ganzzahlen ist kein Untertyp der allgemeinen Listenklasse, die als Elementtyp nur *object* haben kann. Listenoperationen wie das Sortieren, die den Zugriff auf die Elemente erfordern, sind in dem Kontext nicht typsicher, weil sie Downcasts erfordern.

Von Pizza zu Scala

Der Ärger über die in Java ursprünglich fehlende parametrische Polymorphie [5] bildete den Ausgangspunkt für eine der interessantesten unter den neueren funktionalen Sprachen: Scala [9]. Ihr Schöpfer Martin Odersky entwickelte

Einfluss auf
Prolog (dynamische Listen), Smalltalk (zusätzlich funktionale Teilsprache, zur Laufzeit generierter Code) Python, Ruby (zusätzlich Dictionaries)
objektorientierte Sprachen, Python
–
alle modernen funktionalen Sprachen, Python, Occam, Miranda (Abseitsregel)
–
alle modernen funktionalen Sprachen
–
–
–
–
–
OO-Sprachen, Scala
–
Scala
–
–
Python (ZF-Notation)
–
–
–

mit Pizza [8] zunächst eine Erweiterung von Java, die nicht nur jenes Defizit behob, sondern die Sprache um weitere Merkmale aus der funktionalen Welt ergänzte: Funktionen sind eigenständige Objekte und Datentypen ähnlich wie in ML konstruierbar. Der parametrische Listentyp mit den Konstruktoren „[]“ und „:“ sowie der Typ „Bool“ mit den Konstruktoren „true“ und „false“ beruhen in den ML-artigen Sprachen auf der Anwendung des Konzepts der algebraischen Datentypen. Die folgende Definition führt einen Typ für Binärbäume über einem beliebigen Typ für die Werte an den Knoten ein:

```
data BinTree t = BinEmpty | BinNode t (BinTree t)
                                      (BinTree t)
```

Die Konstruktoren *BinEmpty* und *BinNode* eignen sich ebenfalls zur Bildung von Mustern in formalen Parametern, die mit den aktuellen abzugleichen sind, um sie zu zerlegen und den passenden Zweig der Funktionsdefinition auszuwählen wie in der folgenden:

```
inorder BinEmpty      = []
inorder (BinNode v l r) = inorder l ++ [v] ++ inorder r
```

Die Funktion listet die Werte der Knoten von links nach rechts, wobei ++ der Verkettungsoperator für Listen ist.

Pizza verschmilzt das Konzept der algebraischen Datentypen mit dem der Klasse im Konstrukt der Klasse mit Fallunterscheidung (case class) einführt. Zudem bringt es die Option mit, Typparameter durch Prädikate einzuschränken. Scala übernimmt die Pizza-Merkmale, verzichtet jedoch auf die Kompatibilität mit Java, nicht aber auf die Implementierung durch die Java Virtual Machine (JVM): Das heißt, Scala ist zwar keine Obermenge von Java, doch erzeugt der Compiler Code, der auf der JVM (beziehungsweise in der .Net-Umgebung, für die auch ein Compiler existiert) ausführbar ist. Scala-Module können Java-Module importieren und umgekehrt. Scala bleibt bei einer konventionellen, an C und Java angelehnten Syntax und behält die Klasse als fundamentale Programmeinheit bei. Javas Signaturen ersetzt es durch sogenannte Traits, die Signaturen partiell implementieren können.

Auf OCamls Spuren – F#

Während Scala mit Recht als multiparadigmatische Sprache gilt, die funktionale, imperative und objektorientierte Merkmale gleichgewichtig enthält, bleibt F# viel stärker in der Tradition der funktionalen, auf ML zurückgehenden Sprachen, insbesondere von OCaml. Die ML-Sprachen boten, anders als die rein funktionalen wie Miranda und Haskell, immer die Option, unter bestimmten Bedingungen mit Zuweisungen zu programmieren. OCaml und F# erweitern das durch das Mittel, den ML-Verbunddatentyp neben passiven Feldern auch mit Methoden (member functions) zu versehen. Der Typ entspricht den Records von Pascal beziehungsweise den C-Structs und bildet üblicherweise den Ort, an dem Zuweisungen sinnvoll sind. Das erinnert an die vorsichtige Form der Objektorientierung, der sich Niklaus Wirth in Oberon annäherte. Die imperativen und objektorientierten Merkmale sind an untergeordneter Stelle in den Kontext einer Sprache in der ML-Tradition eingebettet. Neu ist jedoch die Ausführbarkeit in der .Net-Umgebung und der Zugriff auf ihre Bibliotheken.

Mit Scala und F# gibt es zwei Sprachen, die es nicht nur erlauben, die Vorzüge der funktionalen Programmierung mit denen der objektorientierten zu kombinieren, sondern auch voll in die Umgebungen eingebunden sind, die heute die beiden Hauptströme der Softwareentwicklung bilden. Sie vermögen damit sowohl von den dort versammelten Ressourcen zu profitieren als auch ihnen neue Impulse zu geben. (ane)

RAINER FISCHBACH

ist Senior Consultant bei der Engineering Consulting & Solutions GmbH in Neumarkt.

Literatur

[1] John Backus; Can programming be liberated from the von Neumann style? A functional style and its algebra of programs; Communications of the ACM; August 1978 (Bd. 21; Nr. 8), S. 613-641

[2] Frederico Biancuzzi, Shane Warden (Hrsg.); Masterminds of programming; Conversations with the creators of major programming languages; O'Reilly 2009

[3] Guy Cousineau, Michel Mauny; The Functional approach to programming; Cambridge University Press 1998

[4] Anthony J. T. Davie; An introduction to functional programming systems using Haskell; Cambridge University Press 1992

[5] Rainer Fischbach; Kalter Kaffee: Java: Programmiersprache der Zukunft?; *iX* 10/96, S. 84-89

[6] Peter J. Landin; The next 700 programming languages; Communications of the ACM; März 1966 (Bd. 9; Nr. 3), S. 157-165

[7] Greg Michaelson; An introduction to functional programming through Lambda Calculus; Addison-Wesley 1989

[8] Martin Odersky, Philip Wadler; Pizza into Java; Translating theory into practice Proc. 24th ACM Symposium on Principles of Programming Languages, 1997, S. 146-159

[9] Martin Odersky, Lex Spoon, Bill Venners; Programming in Scala; Artima 2007

[10] Lawrence C. Paulson; ML for the working programmer; 2. Aufl., Cambridge University Press 1996

[11] Don Syme, Adam Granicz, Antonio Cisternino; Expert F#; apress 2007

[12] David A. Turner; Functional programs as executable specifications; In: C. A. R. Hoare, J. C. Sheperdson; Mathematical logic and programming languages; Prentice-Hall 1985, S. 29-54

[13] Patrick Henry Winston, Berthold Klaus Paul Horn; Lisp; 3. Aufl., Addison-Wesley 1989