Teil IV

Datenstrukturen

Der funktionale Ansatz kann aufgrund seiner Eleganz und Kompaktheit die *Produktivität des Programmierprozesses* erheblich steigern. Allerdings ist diese Produktivitätssteigerung nur dann ein echter Vorteil, wenn die Produkte praktikabel sind – und das heißt vor allem: wenn sie hinreichend effizient sind.

Das größte Effizienzproblem bei funktionalen Programmen sind *Datenstrukturen*. Während es bei Algorithmen relativ problemlos möglich ist, jede imperative Form auch funktional nachzubilden und somit auf jeden Fall Effizienzverluste zu vermeiden (wenn auch manchmal auf Kosten der Eleganz), stellt sich die Situation bei Datenstrukturen komplizierter dar. Hier müssen die Vor- und Nachteile sorgfältig gegeneinander abgewogen und gut ausbalanciert werden.

Über langsame Sicherheit und riskante Effizienz

Die *Vorteile* funktionaler Datenstrukturen sind ähnlich denen im algorithmischen Bereich:

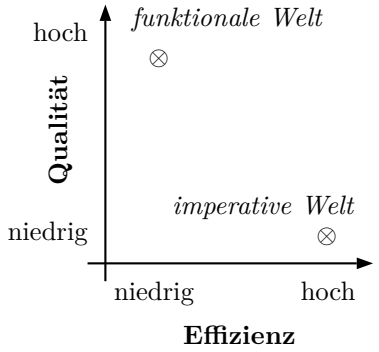
- Funktionale Datenstrukturen befinden sich auf einem *wesentlich höheren Abstraktionsniveau*, das durch eine *ganzheitliche Sicht* auf die Strukturen entsteht: Bei imperativen Sprachen muss man nahezu beliebige Konglomerate von untereinander verzeigerten Zellen managen, was auch bei größter Selbstdisziplin fast immer zu komplexen Fehlern führt. Im Gegensatz dazu arbeitet man in der Funktionalen Programmierung unmittelbar mit einer konzeptuellen Sicht auf die Datenstrukturen: Listen, Stacks, Queues,

Sequenzen, Bäume etc. Bildlich ausgedrückt: man hat nicht Säcke voller Mosaiksteinchen, sondern ganze Bilder.
- Diese ganzheitliche Sicht *unterstützt algorithmische Eleganz*. Die rekursiven Definitionen von Datenstrukturen induzieren unmittelbar homomorphe Algorithmen der Map/Filter/Reduce-Art.
- Man erhält *garantierte Sicherheit* gegen typische Fehler der imperativen Programmierung wie z. B. vorzeitiges Überschreiben von Datenelementen oder Kappen noch benötigter Pointer.

Demgegenüber steht der große *Nachteil* funktionaler Datenstrukturen:

- Die garantierte Sicherheit führt zu *Effizienzverlusten*: Im Zweifelsfall müssen ganze Datenstrukturen kopiert werden. Und auch die Techniken, mit denen dieses exzessive Kopieren verhindert werden kann, führen zu einem gewissen Overhead. Aber das ist eben der Preis, den man für die gewonnene Sicherheit in Kauf nehmen muss.

Imperative Datenstrukturen erhalten ihre Effizienz durch ein einziges Feature: *selektive Änderung* (*selective update*). Sprachen wie PASCAL, C, JAVA etc. haben Konstruktionen wie



$$\ldots\ A[i] = x\ ;\ \ldots$$
$$\ldots\ p\hat{}.next = q\ ;\ \ldots$$

die einen Array an einer Stelle ändern bzw. einen Pointer in einer Datenstruktur umsetzen. In funktionalen Sprachen sehen solche Operationen eher so aus:

$$\ldots \text{ LET } A' = upd(A, i, x) \ \ldots$$
$$\ldots \text{ LET } L' = cons(L, q) \ \ldots$$

Das heißt, es werden *komplette neue Datenstrukturen* gebildet – und das bedeutet neben hohem Speicher- vor allem auch viel Kopieraufwand.

Für diesen Preis erhält man allerdings auch einen hohen Gegenwert: *Sicherheit*. Die Compiler kopieren im Zweifelsfall immer die ganze Struktur, so dass auch nach der Änderung das Original noch vorhanden ist. Dass sich die Speicherverschwendung in Grenzen hält, dafür sorgen dann die Garbage-Kollektoren. Aber der Zeitverbrauch für das Kopieren bleibt bestehen – und das verursacht manchmal gewaltige Effizienzprobleme.

Also ist es ein zentrales Anliegen in der Funktionalen Programmierung und der Compiler für funktionale Sprachen, diese Ineffizienz so oft wie möglich zu vermeiden. Dazu werden eine Reihe spezieller Analyse- und Implementierungstechniken eingesetzt, die wir im Folgenden auf zwei Ebenen studieren wollen:

- *Programmiertechniken.* Als erstes betrachten wir Möglichkeiten, die Effizienz „mit Bordmitteln“ zu steigern, d. h. mit den Mitteln, die funktionale Sprachen den Programmierern anbieten.
- *Compilertechniken.* Dass funktionale Sprachen trotzdem hinreichend effizient sein können, liegt an den Implementierungstechniken, die in den Compilern eingesetzt werden (bzw. eingesetzt werden sollten). Diese werden wir in Kapitel 13 kurz diskutieren.

Die folgenden Betrachtungen gehen auf Arbeiten von Okasaki [107] zurück, die wiederum auf frühere Arbeiten z. B. von Hood und Melville [79] und Gries [62] aufsetzen. In jüngster Zeit wurde der Ansatz von Hinze und Paterson [77] mit Hilfe von Ideen aus dem Bereich der 2-3-Bäume weiter verfeinert.