Vorwort

Dieses Buch vermittelt

- grundlegende Konzepte,
- fortgeschrittene Programmiertechniken,
- formale Grundlagen sowie
- Implementierungsaspekte

moderner funktionaler Programmiersprachen. Die Bezeichnung *funktionale Programmierung* weist auf die intensive Verwendung von Funktionen als Programmierkonstrukte hin. Warum überhaupt bedient man sich Funktionen beim Programmieren?

Funktionen als Grundlage der Programmierung

Funktionen werden schon seit langem in der Mathematik sehr erfolgreich eingesetzt; sie sind relativ einfach zu begreifende Objekte und finden gleichzeitig in sehr vielen Bereichen Anwendung. Insbesondere bieten Funktionen ein anschauliches Konzept von Berechnungen und damit auch von Programmen. Die funktionale Programmierung bezieht einen Großteil ihrer Attraktivität aus der Tatsache, daß der zugrundeliegende Funktionenbegriff sehr eng an den aus der Mathematik angelehnt ist.

Funktionen in der Mathematik

Eine Funktion $f : A \to B$ ist durch ihren *Definitionsbereich* A, ihren Wertebereich B sowie eine *Abbildungsvorschrift* gegeben. f ist das *Funktionssymbol* (oder der Name) der Funktion. Die Abbildungsvorschrift kann durch Aufzählung von Wertepaaren aus $A \times B$ oder aber durch eine Funktionsgleichung gegeben sein. Eine solche Funktionsgleichung führt auf der linken Seite eine oder mehrere *Variablen* ein, die auf der rechten Seite benutzt werden und für Werte aus dem Definitionsbereich stehen. Die rechte Seite der Funktionsgleichung ist durch einen *Ausdruck* gegeben, in dem *Konstanten* und Funktionen verwandt werden.

Als Beispiel betrachten wir die durch eine Gleichung definierte Funktion zur Berechnung des Kreisinhalts bei gegebenem Radius.

$$K(r) = \pi * r^2$$

Definitions- und Wertebereich von K sind die reellen Zahlen, das heißt $K : \mathbb{R} \to \mathbb{R}$. Der definierende Ausdruck der Funktionsgleichung enthält die Konstante π und verwendet die Multiplikations- und Quadratfunktion auf den reellen Zahlen.

Die Funktion K läßt sich in einer funktionalen Sprache genauso definieren und benutzen, wie man es aus der Mathematik gewohnt ist. Im Unterschied zu LISP oder Scheme zeichnen sich neuere funktionale Sprachen insbesondere durch sogenannt starke, polymorphe Typsysteme sowie (benutzerdefinierbare) Datentypen aus. Ein Typsystem verlangt, vereinfacht gesagt, daß jede Funktion einen genau festgelegten Definitions- und Wertebereich besitzen muß. Ein ganz wichtiger Aspekt von Datentypen ist die Möglichkeit, komplexe Wertebereiche zu formulieren und dafür Funktionsdefinitionen über pattern matching vorzunehmen.

Referentielle Transparenz

Eine ganz fundamentale Eigenschaft jeder mathematischen Notation ist, daß ein und dieselbe Variable an verschiedenen Stellen innerhalb ihres Geltungsbereiches immer den gleichen Wert bezeichnet. Dieses Prinzip der *referentiellen Transparenz* gilt in imperativen Programmiersprachen nicht, da ihre Semantik von einem impliziten Zustand abhängt. So erlaubt die Wertzuweisungsoperationen beispielsweise die folgende Programmzeile:

Wertzuweisung

```
i = i + 1
```

Mathematisch betrachtet stellt diese Gleichung eine nicht erfüllbare Aussage dar, das heißt, für die Gleichung gibt es keine Lösung. Imperative Sprachen halten dagegen sehr wohl eine Lösung bereit: Auf der linken Seite steht `i` z. B. für den Wert 5 und auf der rechten Seite für den Wert 4. Noch gravierendere Effekte ergeben sich aus der Verwendung von Seiteneffekten. Ist beispielsweise `f` eine Prozedur, die bei jedem Aufruf eine globale Variable um 1 erhöht und den Wert der Variable als Ergebnis liefert, so gilt

Seiteneffekte

$$\texttt{f} \neq \texttt{f}$$

In funktionalen Sprachen bestehen Programme grundsätzlich aus einer Reihe von Werte- und Funktionsdefinitionen; Zuweisungen an Variablen gibt es im allgemeinen nicht. Dadurch wird ein hohes Maß an referentieller Transparenz erreicht, wodurch funktionale Programme mathematischen Methoden

zugänglich werden: Man kann beispielsweise eine Variable in einem Ausdruck durch ihre Definition ersetzen, ohne die Bedeutung des Ausdrucks zu verändern. Dieses Prinzip nennt man auch *Substituierbarkeit*. Dies kann einerseits dazu dienen, Eigenschaften bestimmter Funktionen nachzuweisen, oder andererseits auch dazu, aus sehr einfachen Spezifikationen durch Transformationen effiziente Implementierungen zu gewinnen.

Substituierbarkeit

Inhalt des Buches

Im ersten Kapitel werden die Konzepte moderner funktionaler Sprachen (Funktionen, Typisierung, Polymorphismus, Datentypen und pattern matching, Funktionen höherer Ordnung) am Beispiel der Sprache ML vorgestellt. Eine Reihe von Autoren vertritt die Meinung, daß darüber hinaus auch verzögerte Auswertung ein grundlegendes Element funktionaler Programmierung ist und somit Bestandteil einer funktionalen Sprache sein sollte. Dementsprechend wird in einigen Sprachen (z. B. Miranda oder Haskell) jeder (Teil-) Ausdruck grundsätzlich nur dann ausgewertet, wenn er zur Berechnung des Gesamtergebnisses auch wirklich benötigt wird. Es gibt in der Tat Situationen, in denen verzögerte Auswertung sehr hilfreich ist und Vorteile gegenüber der *call-by-value* Parameterübergabe bietet, wie man sie von C oder Pascal kennt. Wir diskutieren diesen Aspekt im zweiten Kapitel und stellen dort eine Technik vor, mit der man verzögerte Auswertung von Ausdrücken in call-by-value Sprachen wie ML realisieren kann.

Das dritte Kapitel ist formalen Methoden gewidmet, mit denen man Eigenschaften von Programmen beweisen kann oder aber auch Programme transformieren kann (um z. B. deren Effizienz zu steigern).

In Kapitel 4 betrachten wir den Lambda-Kalkül, der ein formales Funktionenmodell zur Beschreibung aller rekursiven Funktionen darstellt. Mit der Formalisierung als Kalkül wird die automatische Verarbeitung von Funktionen ermöglicht. Den Lambda-Kalkül kann man sicherlich als *das* formale Modell für alle funktionalen Programmiersprachen bezeichnen.

Danach beschreiben wir im fünften Kapitel die Formalisierung von Typsystemen für funktionale Sprachen. Insbesondere geben wir einen Algorithmus an, der für ein Programm den jeweils allgemeinsten Typ automatisch zu ermitteln vermag.

Das letzte Kapitel schließlich vertieft das Verständnis funktionaler Sprachen durch die Untersuchung verschiedener Implementierungsaspekte. Dies umfaßt die Beschreibung eines Interpreters, einer speziellen Maschinenarchitektur und eines Compilers. Für Interpreter, Compiler und Maschine beschreiben wir auch die Realisierung in ML selbst.

Adressaten und Lernziele

Dieses Buch wendet sich primär an Informatik-Studenten und Programmierer, die bereits etwas Erfahrung im Programmieren haben, und die sich einen raschen, aber dennoch tiefergehenden Einblick in die funktionale Programmierung verschaffen wollen.

Ein Lernziel ist der Erwerb spezieller Techniken der funktionalen Programmierung (z. B. pattern matching, polymorphe Funktionen, Funktionen höherer Ordnung, verzögerte Auswertung). Diese werden an kleineren (in den Kapiteln 1 und 2) und auch größeren Beispielen (in Kapitel 6) erprobt, so daß der Leser nach Durcharbeiten des Buches in der Lage sein sollte, selbst komplizierte Programmieraufgaben durch konsequente Definition geeigneter Datentypen und Funktionen zu bewältigen.

Ein weiteres Lernziel bildet das Verständnis der formalen Grundlagen funktionaler Sprachen. Dies beinhaltet zum einen natürlich den Stoff der Kapitel 4 und 5, zum anderen aber vermittelt gerade die Implementierung einer Sprache Einsichten in innere Strukturen, die durch rein formale Definitionen zwar genau erfaßt werden können, oft aber nur schwer zu verstehen sind.

Schließlich soll demonstriert werden, daß wegen ihrer starken formalen Basis funktionale Sprachen einen einfachen Zugang mathematischer Methoden erlauben (z. B. zum Zwecke des Korrektheitsbeweises oder der optimierenden Transformationen). Damit soll dokumentiert werden, daß es in funktionalen Sprachen sehr viel einfacher ist, korrekte Software zu realisieren, als in imperativen Sprachen, ohne daß dies zu Effizienzeinbußen führen muß.

Weitere Informationen

Einige zusätzliche Informationen zu diesem Buch sind über das World-Wide Web zugänglich:

```
http://www.fernuni-hagen.de/inf/pi4/erwig/buch/fp/
```

Dort sind insbesondere online-Versionen aller im Buch abgebildeten Programme verfügbar. Einige Links bezüglich der im Kurs verwandten Programmiersprache ML sind unter anderem unter

```
http://www.dcs.ed.ac.uk/home/cdw/archive.html
http://cm.bell-labs.com/cm/cs/what/smlnj/links.html
```

zusammengetragen. Neben allgemeiner Information und Dokumentation sind dort auch Verweise auf verschiedene Implementierungen zu finden.

Alle abgebildeten Programme genügen dem aktuellen ML-Standard von 1997. In den Beispielen werden zuweilen die Antworten des ML-Interpreters auf die Eingaben gezeigt. Daran kann man insbesondere Details des Typsystems erklären und auch mögliche Programmierfehler besprechen. Wir verwenden hier das frei erhältliche System „Standard ML of New Jersey“, in der Version 110. Dies ist unter anderem für Unix und Windows 95/Windows NT verfügbar:

```
http://cm.bell-labs.com/cm/cs/what/smlnj/
```

Danksagung

Grundlage für das vorliegende Buch sind Kursmaterialien zu einer Vorlesung, die wiederholt an der FernUniversität Hagen gehalten worden ist. Dabei haben eine Reihe von Studenten durch Hinweise auf Fehler zu Verbesserungen beigetragen. Außerordentliches Engagement haben dabei André Piotrowski und Thomas Salvador gezeigt, denen ich an dieser Stelle besonders dafür danken möchte.

Ungeachtet der vielen Überarbeitungen sind gewiß noch einige Fehler in diesem Buch verblieben. Für Hinweise auf solche Fehler wäre ich sehr dankbar.