Funktionales Programmieren: eine neue Verbindung von Informatikunterricht und Mathematik

Hermann Puhlmann
Arbeitsgruppe Fachdidaktik
Fachbereich Mathematik
Technische Universität Darmstadt
puhlmann@mathematik.tu-darmstadt.de

April 1998

Eine Verbindung von Mathematik und Informatik wird in vielen Informatik-Schulbüchern dadurch hergestellt, dass mathematische Aufgaben als Anlass für die Einführung von Elementen einer Programmiersprache (meist Pascal) dienen. So findet man Prozentrechnungsaufgaben, wenn es um arithmetische Ausdrücke geht, und quadratische Gleichungen, um eine Anwendung für die Fallunterscheidung zu haben. Zur Motivation der Schleifenprogrammierung müssen oft iterative Verfahren aus der Numerik herhalten. Dieses Vorgehen wird weder den Schülern noch dem Fach gerecht: den Schülern deshalb nicht, weil Schwierigkeiten mit den mathematischen Themen direkt in Probleme beim Programmieren münden, und dem Fach nicht, weil der falsche Eindruck erweckt wird, dass das Programmieren numerischer Verfahren ein Thema der Informatik sei.

Dieser Beitrag soll einen anderen Bezug der beiden Fächer in den Blick rücken: die Analogie zwischen Computerprogrammen und mathematischen Funktionen. Bei beiden führt jede zulässige Eingabe zu einem eindeutig bestimmten Ausgabewert. Abbildung 1 veranschaulicht dies.

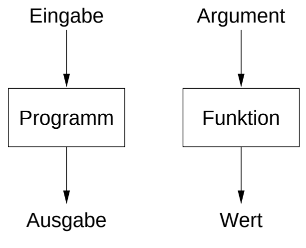

Abbildung 1: Programme und Funktionen erzeugen aus einer Eingabe einen Ausgabewert

Die eindeutige Beziehung zwischen Ein- und Ausgabe, die eine Funktionsdefinition herstellt, und der operationale Charakter eines Computerprogramms werden durch *funktionale Programmiersprachen* miteinander verbunden. Ein funktionales Programm besteht aus einer Menge von Funktionsdefinitionen, die auch in ihrer Notation an die mathematische Schreibweise angelehnt sind. Die Programmieraufgabe besteht darin, Funktionen zu finden, die die gewünschte Beziehung zwischen Ein- und Ausgabe beschreiben.

Ich möchte dies an drei Beispielen demonstrieren. Eine Einführung in die verwendete funktionale Sprache Standard ML gibt A. Schwill in [Sch93]. Die Beispiele dieses Artikels sind jedoch so gewählt, dass sie sich auch ohne Vorkenntnisse in ML erschliessen lassen.

Ein erstes funktionales Programm: der endliche Akzeptor

Das erste Beispiel eignet sich für den Einstieg in die Sprache ML, und man wird sehen, dass schon an einem kleinen funktionalen Programm wesentliche Aspekte des Funktionsbegriffs deutlich werden.

Die Aufgabe sei, von einer Folge von Nullen und Einsen zu entscheiden, ob die Anzahl der Einsen gerade ist. Die theoretische Informatik kennt hierzu das Automatenmodell des endlichen Akzeptors, der in Abbildung 2 graphisch dargestellt wird.

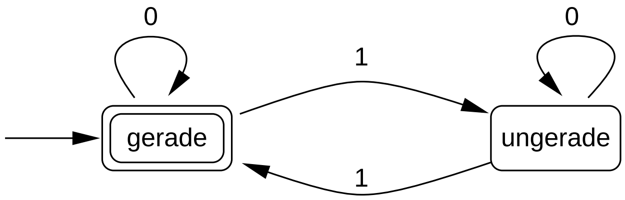

Abbildung 2: Ein endlicher Akzeptor zur Prüfung von Bitfolgen

Der Akzeptor hat zwei Zustände, „gerade“ und „ungerade“, wobei „gerade“ sowohl Start- als auch Endzustand (=Erfolgszustand) ist. Die Pfeile zeigen die Zustandsübergänge in Abhängigkeit von eingelesenen Symbolen an, und eine 0–1-Folge wird akzeptiert, wenn der Akzeptor am Schluss im Erfolgszustand ist.

In Standard ML wird der Akzeptor durch dieses Programm realisiert, das aus den drei Funktionen `ubergang`, `hakzeptor` (für „Hilfsakzeptor“) und `akzeptor` besteht:

```
fun ubergang ("gerade", 0)   = "gerade"
  | ubergang ("gerade", 1)   = "ungerade"
  | ubergang ("ungerade", 0) = "ungerade"
  | ubergang ("ungerade", 1) = "gerade"

fun hakzeptor (zustand, [])    = zustand
  | hakzeptor (zustand, e::es)
          = hakzeptor(ubergang(zustand, e), es)

fun akzeptor (anfzustand, endzustand, eingabe)
= (hakzeptor(anfzustand, eingabe) = endzustand)
```

Der Aufruf `akzeptor ("gerade", "gerade", [0, 1, 1, 1])` liefert das Ergebnis `false` und ein Aufruf mit der Folge `[1, 0, 0, 1]` liefert `true`.

Außerdem gibt ML Informationen über den Definitions- und Wertebereich der Funktionen aus. Für die Funktion `ubergang` ist diese Angabe:

```
val ubergang = fn: string * int -> string
```

Dies zeigt an, dass `ubergang` eine Funktion ist, deren Argument ein Paar aus einer Zeichenkette und einer Zahl ist. Der Funktionswert ist eine Zeichenkette, in diesem Beispiel „gerade“ oder „ungerade“.

Allgemeines über Funktionen — am Beispiel erkannt

Schon an der einfachen Funktion `ubergang` läßt sich einiges über Funktionen lernen: Zunächst können Funktionen durch *Wertetabellen* angegeben werden. Andererseits stößt man hierbei auf Grenzen, wenn der Definitionsbereich groß oder gar unendlich ist. Das zeigt sich auch in unserem Beispiel, denn ML wird warnen, dass die Definition von `ubergang` nicht alle möglichen Fälle der Eingabe von Zeichenketten und Zahlen erfasst. Man kann dies durch die Verwendung selbst definierter Datentypen vermeiden (also den Definitionsbereich einschränken) oder noch eine Behandlung der bisher nicht vorgesehenen Fälle hinzufügen. An dieser Stelle wird deutlich, dass zu einer Funktion nicht nur die *Zuordnungsvorschrift*, sondern auch *Definitions-* und *Wertebereich* gehören.

Funktionale Programme zeigen weiter auf, dass die Argumente und Werte von Funktionen *keineswegs immer Zahlen* sein müssen. Mehr noch: Dass auch bei nichtnumerischen Eingaben eine Definition „durch eine Formel“ möglich ist, indem nämlich auf die Struktur des Arguments zugegriffen wird, zeigt die Hilfsfunktion `hakzeptor`. Sie berechnet den Zustand, der bei Eingabe einer Liste von Eingabezeichen von einem Startzustand aus erreicht wird. Werden keine Eingabezeichen eingegeben (also die leere Liste `[]`), so bleibt der Zustand unverändert (erste Zeile der Funktionsdefinition). Andernfalls besteht die Eingabe aus einem ersten Zeichen `e` und der Liste der übrigen Eingabesymbolen, den „`es`“. Man findet dann den am Schluss erreichten Zustand, indem man erst einmal den Übergang mit dem Symbol `e` durchführt und den Ergebniszustand zusammen mit den übrigen Zeichen, den `es`, wieder an die Funktion `hakzeptor` übergibt. Der sukzessive Abbau der Folge der Eingabezeichen ist geeignet, *Rekursion* einzuführen, ohne dass auf die üblichen Beispiele der Fakultätsfunktion oder der Fibonacci-Zahlen Bezug genommen wird.

Schließlich zeigen alle drei Funktionen, dass eine Funktion *mehrere Argumente* haben kann (die man zu einem *Tupel* zusammenfasst). Damit kann man Funktionen von der eindimensionalen Sicht der Analysis in der Schule lösen und auf vektorielle Funktionen in der Linearen Algebra vorbereiten. Zugleich bietet sich die Möglichkeit, zwischen *Parametern* und *Funktionsargumenten* zu unterscheiden. Die hier definierte Funktion `akzeptor` realisiert den gewünschten Akzeptor ja erst nach Eingabe von Anfangs- und Endzustand. Gibt man dafür andere Werte an, so entsteht ein anderer Akzeptor. Insofern spielen `anfzustand` und `endzustand` die Rolle von Parametern, nicht von Argumenten. Die Spezialisierung

```
fun bitpruef (eingabe)
         = akzeptor("gerade", "gerade", eingabe)
```

mit Angabe der Parameterwerte kann dies noch hervorheben.

Fortgeschrittene Programmiertechnik:
Curry-Operation und Funktionen höherer Ordnung

Wir greifen den Aspekt der Funktionen mit Parametern noch einmal auf. In mathematischer Notation schreibt man eine solche Funktion etwa als $f_a(x)$. Ist a ein reeller Parameter, und sind Funktionsargument und -wert ebenfalls stets reell, so ist damit für $a \in \mathbb{R}$ eine Funktion $f_a : \mathbb{R} \to \mathbb{R}$ gegeben, wobei $\mathbb{R} \to \mathbb{R}$ der Typ der Funktion ist. Was ist aber der Typ der Funktion f, solange a noch nicht angegeben ist? Bild 3 zeigt verschiedene Interpretationsmöglichkeiten. Dabei macht die erste Zeichnung keinen Unterschied zwischen a und x. Sind beide Eingaben wirklich gleichberechtigt, so fasst man sie zu einem Paar zusammen, das dann als *ein* Funktionsargument an f übergeben wird. Das zeigt die zweite Zeichnung, und insofern hat f den Typ $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Dies entspricht jedoch nicht der üblichen Interpretation des Parameters a, die schließlich in der letzten Zeichnung zum Ausdruck kommt. Hier ist f eine Funktion, der der Wert a als Argument übergeben wird. Ist a angegeben, so ist das Ergebnis, also der Funktionswert von f, wieder eine Funktion, nämlich die Funktion f_a. Deren Typ ist $\mathbb{R} \to \mathbb{R}$, also hat f den Typ $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$.

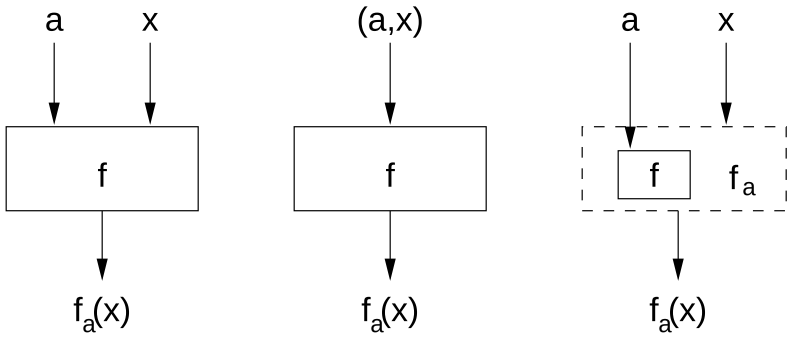

Abbildung 3: Interpretationen einer Funktion mit Parameter

Den Übergang von einer Funktion mit Typ $A \times B \to C$ zur dazu gehörigen Funktion mit dem Typ $A \to (B \to C)$ bezeichnet man nach dem englischen Mathematiker H. B. Curry als *Curry-Operation*. Funktionen in Curry-Darstellung haben den Vorteil, dass man sie *teilweise auswerten* kann. Man muss also nicht das vollständige Paar $(a, b) \in A \times B$ angeben, sondern erhält schon bei Angabe von a ein Ergebnis, nämlich eine Funktion f_a, die nur noch von dem einen Argument b abhängt.

Durch Verwendung der Curry-Darstellung können auch in ML Funktionen flexibler verwendet werden. Man schreibt die Argumente einer Funktion dazu nicht mehr in Klammern, sondern einfach durch Leerzeichen anstatt Kommas getrennt hinter den Funktionsnamen. Schreibt man die Funktion `akzeptor` in dieser Form, so lässt sich bei der Spezialisierung zu `bitpruef` die partielle Auswertung ausnutzen:

```
fun akzeptor anfzustand endzustand eingabe
    = (hakzeptor(anfzustand, eingabe) = endzustand)

fun bitpruef = akzeptor "gerade" "gerade"
```

Die weiteren Programmbeispiele verwenden alle die Curry-Darstellung.

Damit haben wir Funktionen kennengelernt, deren Werte Funktionen sind. Genauso ist es aber auch möglich, Funktionen zu definieren, die Funktionen als Argumente haben. Solche „funktionenverarbeitende“ Funktionen heißen *Funktionen höherer Ordnung* und werden in der Mathematik Funktionale genannt. Die auffälligsten Funktionale im Mathematikunterricht sind das Integral $\int$ und das Differential $\frac{d}{dx}$, die jeweils eine Funktion in eine Funktion überführen. Aber auch die Funktion max, die das Maximum einer beschränkten Funktion liefert, ist eine Funktion höherer Ordnung. Schließlich ist auch die Schreibweise $\sin^2(x)$ nicht nur klammersparend im Vergleich zu $(\sin(x))^2$, sondern zeigt die Verwendung des Funktionals $(\cdot)^2$, das einer Funktion $f : \mathbb{R} \to \mathbb{R}$ die Funktion f^2 mit $f^2(x) = (f(x))^2$ zuordnet.

Dass Schüler an diesen Stellen überhaupt Funktionen erkennen und Definitions- und Wertebereiche benennen können, kann meines Erachtens durch die Verwendung einer funktionalen Sprache gefördert werden. Dabei ist es wieder möglich, die Begriffe aus informatischen Anwendungen heraus zu entwickeln, wie das nächste Beispiel zeigt.

Sortieren und Polymorphie

In diesem Abschnitt wird mit dem Sortieren von Daten ein Dauerthema der Informatik aufgegriffen. Wir betrachten das Verfahren „Sortieren durch Einfügen“, wobei die Daten als Liste ganzer Zahlen vorliegen sollen. Beim Sortieren durch Einfügen wird aus den Ausgangsdaten sukzessive eine neue, sortierte Liste aufgebaut, indem jedes Element der Ausgangsliste an die richtige Stelle des bereits sortierten Anteils der Daten eingefügt wird. Ein erstes ML-Programm dafür ist schnell geschrieben:

```
fun einfueg a []      = [a]
  | einfueg a (x::xe) = if a<x then (a::x::xe)
                               else (x::(einfueg a xe))

fun sort []      = []
  | sort (x::xe) = einfueg x (sort xe)
```

Das Programm liest man so: Die Funktion `einfueg` fügt das Element `a` in die leere Liste `[]` ein, indem es die einelementige Liste `[a]` erzeugt. Eine nicht leere Liste besteht aus einem ersten Element `x` und den weiteren Elementen, den „`xe`“n. Ist `a` kleiner als `x`, so wird es an den Anfang der Liste gestellt, andernfalls bleibt `x` vorne und `a` wird in die Liste der `xe` eingefügt.

Die Definition der Funktion `sort` liest man so: Die leere Liste `[]` ist schon sortiert, und man muss nichts machen. Bei einer nicht leeren Liste `(x::xe)` werden die `xe` sortiert. In diese dann sortierte Liste wird das Element `x` an der richtigen Stelle eingefügt.

Bei der Ausführung dieses Sortierprogramms zeigt sich eine vermeintliche Schwäche des funktionalen Programmierens, die oft auf den ausgiebigen Gebrauch der Rekursion zurückgeführt wird. Der durch Funktionsauswertungen zu einem Endergebnis zu „reduzierende“ Ausdruck wird nämlich erst einmal weit aufgebläht. Zur Darstellung der Einzelschritte wird das — sonst in ML nicht verwendete — Symbol `==` benutzt.

```
sort [8,3,5]
== einfueg 8 (sort [3,5])
== einfueg 8 (einfueg 3 (sort [5]))
== einfueg 8 (einfueg 3 (einfueg 5 (sort [])))
```

Die Eingabedaten müssen sich also *alle* zuerst zum Einfügen anstellen, ehe mit der Konstruktion der sortierten Liste begonnen wird. Es geht dann weiter mit

```
== einfueg 8 (einfueg 3 (einfueg 5 []))
== einfueg 8 (einfueg 3 [5])
== ... = [3,5,8]
```

Das Problem scheint darin zu bestehen, dass es die funktionale Schreibweise nicht erlaubt, Zwischenergebnisse zu notieren. Sonst könnte ja erst einmal die Liste konstruiert werden, in der die 8 und die 3 bereits sortiert sind, ehe auch noch die 5 einsortiert wird. Genau dies erlaubt die Rekursion mit *Akkumulatortechnik*, bei der ein Argument der rekursiven Funktion verwendet wird, um Zwischenergebnisse zu speichern. Dazu benötigt man eine Hilfsfunktion `hsort`, in der die eigentliche Arbeit gemacht wird.

```
fun hsort [] schon_sortiert      = schon_sortiert
  | hsort (x::xe) schon_sortiert = hsort xe (einfueg x schon_sortiert)
```

Wir lesen dies so: Um eine leere Liste in eine bereits sortierte Liste einzusortieren, muss man nichts tun. Eine Liste `(x::xe)` wird in eine bereits sortierte Liste einsortiert, indem man das erste Element `x` in die sortierte Liste einfügt und die übrigen `xe` in die daraus resultierende Liste — wieder mit `hsort` — einsortiert.

Eine Liste sortiert man nun, indem man sie in eine zu Beginn leere Liste einsortiert. Damit der Benutzer der Sortierfunktion dies nicht wissen muss, stellt man eine „komfortable“ Funktion `sort2` zur Verfügung:

```
fun sort2 liste = hsort liste []
```

Der Aufruf `sort2 [8,3,5]` zeigt, dass die Akkumulatortechnik das Aufblähen des zu vereinfachenden Ausdrucks verhindert[1].

```
sort2 [8,3,5]
== hsort [8,3,5] []
== hsort [3,5] (einfueg 8 [])
== hsort [3,5] [8]
== hsort [5] (einfueg 3 [8])
== hsort [5] [3,8]
== ... == [3,5,8]
```

Obwohl im Beispiel eine Liste ganzer Zahlen (`int list`) sortiert wird, haben wir bei der Formulierung des Sortierens durch Einfügen nirgends von speziellen Eigenschaften ganzer Zahlen Gebrauch gemacht. Benötigt wird lediglich die Vergleichbarkeit

[1] Ohne es zu erwähnen, wurde die Akkumulatortechnik im Übrigen auch schon bei der Funktion `hakzeptor` verwendet.

zweier Daten vom Typ `int`. Der Vergleichsoperator `<` lässt sich in ML aber auch auf Gleitkommazahlen (`real`) und Zeichenketten (`string`) anwenden. So überrascht es nicht, dass genau dieselbe Sortierfunktion auch für diese Datentypen funktioniert.[2] Sie lässt sich im Prinzip sogar für alle nur denkbaren Datentypen, auch für selbstdefinierte, verwenden, solange deren Elemente miteinander verglichen werden können. Damit ist das Sortieren ein Beispiel einer *polymorphen* Funktion, also einer Funktion, die bei unveränderter Funktionsdefinition für Daten verschiedener Typen benutzt werden kann.

Damit die Sortierfunktion wirklich universell verwendbar wird, ist allerdings noch eine kleine Änderung nötig. Das Zeichen `<` ist nur für Grunddatentypen definiert. Zur Verwendung mit beliebigen Datentypen muss der Sortierfunktion die gewünschte Vergleichsfunktion explizit als Argument übergeben werden. Damit erhalten wir die polymorphen Varianten der oben angegebenen Funktionen:

```
fun p_einfueg kleinerals a []
                    = [a]
  | p_einfueg kleinerals a (x::xe)
                    = if (kleinerals a x) then (a::x::xe)
                         else (x::(p_einfueg kleinerals a xe))

fun p_hsort kleinerals [] schon_sortiert
    = schon_sortiert
  | p_hsort kleinerals (x::xe) schon_sortiert
    = p_hsort kleinerals xe (p_einfueg kleinerals x schon_sortiert)

fun p_sort kleinerals liste = p_hsort kleinerals liste []
```

Die Polymorphie dieser Funktionen erkennt man auch an den Typangaben, die ML zu diesen Funktionen macht:

```
val p_einfueg = fn: ('a -> 'a -> bool) -> 'a -> 'a -> 'a list
val p_hsort = fn: ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
val p_sort = fn: ('a -> 'a -> bool) -> 'a list -> 'a list
```

Die mit dem Hochkomma eingeleitete Typangabe `'a` weist auf eine Typvariable hin und ist für polymorphe Funktionen kennzeichnend. Sie zeigt, dass an dieser Stelle ein beliebiger Datentyp auftreten darf. Erst beim Aufruf der Funktion wird der tatsächlich verwendete Typ aufgrund der angegebenen Argumente bestimmt und an allen Stellen für die Typvariable eingesetzt. Stellt ML bei einem Aufruf von `p_sort` also fest, dass die Vergleichsfunktion zwei ganze Zahlen vergleicht, so wird die Typvariable `'a` mit `int` belegt. Entsprechend muss nun anstelle einer `'a list` eine `int list` eingegeben werden, und auch das Ergebnis wird eine `int list` sein.

Spezielle Sortierfunktionen für konkrete Datentypen lassen sich aus der Funktion `p_sort` leicht gewinnen. Dabei sieht man die Eleganz, die das Konzept der Funktionen höherer Ordnung ermöglicht. Ist `alpha_kleiner` eine Funktion, die zwei Zeichenketten auf die lexikographische Ordnung testet, so lässt sich eine Liste von Zeichenketten mit

[2] In manchen ML-Versionen muss der Datentyp dann aber explizit angegeben werden.

```
fun alphasort = p_sort alpha_kleiner
```

sortieren. Die Typangabe ist

```
val alphasort = fn: string list -> string list
```

Mit einer Funktion `liste_kuerzer`, die die Länge zweier Listen vergleicht, erhalten wir eine Funktion, die eine Liste von Listen ihrer Länge nach ordnet.

```
fun listsort = p_sort liste_kuerzer
```

Die Funktion `listsort` ist wieder polymorph, denn sie lässt sich zum Sortieren einer Liste von Listen beliebigen Typs verwenden:

```
val listsort = fn: 'a list list -> 'a list list
```

Im Informatik-Unterricht bietet es sich an, neben dem Sortieren durch Einfügen noch ein weiteres Sortierverfahren mit anderer Komplexität zu behandeln, z. B. Quicksort, das man in [Wik87, MCP93] findet. Als Komplexitätsmaß kann man die Anzahl der Funktionsaufrufe verwenden. Manche ML-Systeme (New Jersey ML, Caml) bieten die Möglichkeit, diese automatisch zählen zu lassen, so dass ein heuristischer Einstieg in die Komplexitätstheorie möglich ist.

Wir wenden uns aber einem letzten Beispiel zu, das verstärkt an die Mathematik anknüpft.

Das Differential: Ein Mini-Computeralgebra-System

Die Konstruktion eines Mini-Computeralgebra-Systems zeigt, mit welch geringem Aufwand in ML auch komplexe Anwendungen programmiert werden können. Das nachfolgende Programm differenziert Terme, die aus Variablen, den Operatoren `+`, `-` und `*` und der Exponentialfunktion `exp` aufgebaut sein können. Zur Darstellung der Terme wird zuerst ein Datentyp `TERM` definiert.

```
datatype TERM = Const of int
              | Var of string
              | Op of string * TERM * TERM
              | Func of string * TERM
```

Einen Eindruck von der Darstellung der Terme in ML-Notation gibt Tabelle 1.

5	`Const 5`
x	`Var "x"`
$x+y$	`Op("+", Var "x", Var "y")`
e^x	`Func("exp", Var "x")`

Tabelle 1: Darstellung von Termen

Die Funktion `diffx` differenziert Terme nach der Variablen `x`. Dabei realisiert etwa die vorletzte Zeile der folgenden Funktionsdefinition die Produktregel und die letzte Zeile die Kettenregel.

```
fun diffx (Const i)      = Const 0
  | diffx (Var y)        = if y="x" then Const 1
                                    else Const 0
  | diffx (Op("+",l,r)) = Op("+", diffx l, diffx r)
  | diffx (Op("-",l,r)) = Op("-", diffx l, diffx r)
  | diffx (Op("*",l,r)) = Op("+", Op("*", diffx l, r),
                                    Op("*", l, diffx r))
  | diffx (Func("exp",y))
                         = Op("*", Func("exp",y), diffx y)
```

Natürlich fehlen der Funktion `diffx` in dieser Form noch die Behandlung zahlreicher Standardfunktionen, und die Quotientenregel ist noch nicht eingebaut. Für den Informatik-Unterricht bietet sich dadurch ein weites Feld, in dem Schülerinnen und Schüler Verbesserungen anbringen können. Genauso können sie weitere Funktionen schreiben, die die Eingabe von Termen bequemer machen und auch die Ausgabe vereinfachen. Beispiele für solche Erweiterungen finden sich in [Wik87].

Obwohl die Möglichkeiten zur Programmierung eines Mini-Computeralgebra-Systems hiermit nur angedeutet sein können, sollen die Vorteile für den Informatikunterricht aufgezeigt werden. Ein solches Projekt bringt zwar wieder ein mathematisches Thema in den Informatikunterricht hinein, es wird aber — im Gegensatz zu den eingangs kritisierten Beispielen — nicht als Aufhänger zur Einführung eines Programmiersprachenelements missbraucht. An die Stelle der Umsetzung eines Algorithmus rückt nun der Gedanke der Erstellung eines Informatiksystems. Solche Systeme kennen die Schülerinnen und Schüler vom Umgang mit den komplexeren Taschenrechnern und vom Computereinsatz im Mathematikunterricht. Sie können nun im Informatikunterricht die Einsicht gewinnen, dass das Differenzieren mechanisch durchführbar ist. Dazu müssen sie noch nicht einmal geübte Differenzierer sein. Es genügt für den Informatikunterricht, wenn sie die Differentiationsregeln in einem Regelwerk nachschauen. Die vielfältige Erweiterbarkeit des Mini-Computeralgebra-Systems eröffnet zudem ausgezeichnete Möglichkeiten zur Weiterarbeit in Projektform.

Wirkung auf Informatik und Mathematik in der Schule

Die angeführten Beispiele zeigen, dass funktionale Sprachen als Hilfsmittel für einen Informatikunterricht geeignet sind, in dem nicht das Beherrschen der gerade „aktuellen" Programmiersprache im Mittelpunkt steht. Vielmehr dient die Programmierung hier der Umsetzung des eigentlichen Themas, nämlich informatischer Konzepte.

Das einfache Beispiel des endlichen Akzeptors zeigt, dass Informatik-Inhalte schon mit bescheidenen programmiersprachlichen Mitteln umgesetzt werden können. Es ist also nicht nötig, der eigentlichen Informatik einen Programmierkurs voranzustellen. Auch die Programmierung von Sortierverfahren erfordert zunächst geringe Kenntnisse der Programmiersprache. Es zeigt sich aber, dass hier einerseits fortgeschrittene Sprachkonzepte eingeführt werden können, andererseits auch die Verbindung zu anderen Themen der Informatik, etwa der Komplexitätstheorie möglich ist. Schließlich wird mit der Programmierung des Mini-Computeralgebra-Systems aufgezeigt, dass funktionale Sprachen auch zur Realisierung anspruchsvoller Informatiksysteme geeignet sind, ohne dass die Programme gross und unlesbar werden müssen.

Während bei der Verwendung imperativer Sprachen viel Energie darauf verwendet wird, das zugrunde liegende Maschinenmodell zu verstehen, baut die funktionale Programmierung nur auf dem Funktionsbegriff und dem Auswerten von Funktionen auf. An dieser Stelle entsteht eine neue Verbindung des Informatik- und des Mathematikunterrichts. Die Informatik bezieht sich nun nicht mehr auf ausgewählte numerische Beispiele als „Motivation“ für die Programmierung. Statt dessen knüpft sie an den in der Mathematik geprägten Funktionsbegriff an. Es ist dabei nicht nötig, dass die Schüler ein genaues und richtiges Verständnis dieses Begriffs mitbringen. Es genügt die grobe Vorstellung, dass man in eine Funktion einen Wert hineingeben kann und dann ein Ergebnis herauskommt. Davon ausgehend vermag die funktionale Programmierung dann zu weiteren Einsichten über Funktionen zu verhelfen und kann so das mathematische Verständnis fördern.

Literatur

[MCP93] Colin Myers, Chris Clack und Ellen Poon. *Programming with Standard ML*. Prentice Hall, 1993.

[Sch93] Andreas Schwill. Funktionale Programmierung mit Caml. *Log In*, 13(4):20–30, 1993.

[Wik87] Ake Wikström. *Functional Programming Using Standard ML*. International Series in Computer Science. Prentice Hall, 1987.

ML für DOS im Internet: http://www.dina.kvl.dk/~sestoft/mosml.html