Grundlagen und Methoden funktionaler Programmierung

Sommersemester 1999

Peter Padawitz

TU Dortmund, Germany

22. Januar 2020

Inhaltsverzeichnis

1 Funktionales Programmieren und ML

1.1 Programmierparadigmen

Darunter versteht man verschiedenen Auffassungen darüber, was die Ausführung eines Programms - bezogen auf das zu lösende Problem - bewirkt. Prinzipiell lassen sich drei Auffassungen voneinander abgrenzen:

a) Auswertung von Ausdrücken (einer formalen Sprache)

b) Beantwortung von Anfragen (an ein Informationssystem)

c) Manipulation von Objekten (der realen Welt)

Einerseits gibt es Problemstellungen, die von sich aus eine dieser Auffassungen nahelegen. So sind arithmetische Aufgaben sicherlich ein Prototyp für a), während man bei b) an Zugriffe auf eine Datenbank denkt (Wie lautet das Geburtsdatum von XY?) und c) den ganzen Bereich der Beobachtung und Steuerung konkreter Objekte abdeckt.

Andererseits sind a), b) und c) auch einzelnen Programmiersprachen zugrundeliegende Vorstellungen, die auf alle Probleme, die man mit der jeweiligen Sprache löst, übertragen werden. Dann ergibt sich die folgende Zuordnung von Sprachklassen zu Paradigmen:

a) *funktionale* und *applikative* Sprachen,

b) *relationale* und *logische* Sprachen,

c) *prozedurale, imperative* und *objektorientierte* Sprachen.

1.2 Literatur

Folgende Bücher führen in die funktionale Programmierung ein:

- *Appel*, Modern Compiler Implementation in ML, Cambridge University Press 1998
- *Bird, Wadler*, Einführung in die funktionale Programmierung, Hanser 1992
- *Bosworth*, A Practical Course in Functional Programming Using ML, McGraw-Hill 1995
- *Cousineau, Mauny*, The Functional Approach to Programming, Cambridge University Press 1998
- *Davie*, An Introduction to Functional Programming Systems using Haskell, Cambridge University Press 1992
- *Erwig*, Funktionale Programmierung, Oldenbourg 1999
- *Hinze*, Einführung in die funktionale Programmierung mit Miranda, Teubner 1992
- *Myers, Clack, Poon,* Programming with Standard ML, Prentice-Hall 1993
- ***Paulson*, ML for the Working Programmer**, Cambridge Unversity Press, 2. Auflage 1996
- *Pepper*, Funktionale Programmierung in Opal, ML, Haskell und Gofer, Springer 1998
- *Reade*, Elements of Functional Programming, Addison-Wesley 1989
- *Sethi*, Programming Languages: Concepts and Constructs, Addison-Wesley 1989
- *Stansifer*, ML Primer, Prentice-Hall 1992
- *Thiemann*, Grundlagen der funktionalen Programmierung, Teubner 1994
- *Ullman*, Elements of ML Programming, Prentice-Hall 1994

- *Watt*, Programmiersprachen, Hanser 1996
- *Wikström*, Functional Programming using Standard ML, Prentice-Hall 1987

Paulson ist das aktuelle Standardwerk. *Sethi* und *Watt* behandeln das funktionale Paradigma neben den anderen Paradigmen. Exemplare der Bücher von *Paulson* (allerdings nur in der 1. Auflage), *Reade* und *Sethi* sind in der Lehrbuchsammlung der Uni Dortmund vorhanden.

Die folgenden Einführungen in die gesamte Informatik entsprechen in München, Karlsruhe bzw. Berlin gehaltenen Grundstudiumsvorlesungen. Sie benutzen häufig das funktionale Paradigma.

- *Broy*, Informatik: Eine grundlegende Einführung, 2. Auflage, Bände I und II, Springer 1998
- *Goos*, Vorlesungen über Informatik, Bände I-IV, 2. Auflage, Springer 1997-1999
- *Pepper*, Grundlagen der Informatik, Oldenbourg 1992

Wir verwenden in der Lehrveranstaltung die Sprache (Standard) **ML** zur Darstellung von Konzepten und Methoden funktionaler Programmierung. Dabei werden auch einige allgemeine Begriffe aus der mathematischen Logik und der Theorie diskreter Strukturen eingeführt, die nützlich sind, um **Korrektheitsbedingungen** an – nicht nur funktionale – Programme zu beweisen und die Persistenz der Bedingungen über mehrere Phasen der Programmentwicklung hinweg sicherzustellen.

1.3 Kontextfreie Grammatik von ML

Ein Begriff aus der Sprachtheorie ist z.B. derjenige einer **kontextfreien Grammatik**, mit dem die Syntax der meisten Programmiersprachen beschrieben wird. Wir gehen darauf hier nicht allgemein ein, sondern benutzen ihn nur, um – vor der Behandlung der einzelnen Sprachkonstrukte in den folgenden Kapiteln – einmal die komplette Syntax von ML aufzuschreiben. Die Grammatik besteht aus **Regeln** oder **Produktionen** der Form

$$\mathrm{A} := \mathrm{B} \mid \mathrm{C} \mid \mathrm{D}$$

A, B, C, D sind Variablen (*Nichtterminale*) für Programmstücke. Die Regel bedeutet, kurz gesagt, daß jedes A ein B, ein C oder ein D ist. Außer Variablen und dem "Oder-Strich" | enthalten einige Regeln Konstanten (*Terminale*). Unten sind das alle nicht aus Großbuchstaben bestehenden Wörter (einschließlich des dicken senkrechte Strichs **|**. ε bezeichnet das *leere Wort*. Die Grammatik heißt kontextfrei, weil sie nur den kontextunabhängigen Teil einer Sprache definiert. Kontextabhängig sind demgegenüber Bedingungen wie "Ein Wert muß vor seiner Benutzung definiert werden" oder "Ein aktueller Parameter darf nur einen formalen Parameter gleichen Typs ersetzen".

IDENT := *String, der kein Schlüsselwort von ML ist*
EXCEPT := IDENT
SIGN := IDENT
STRUCT := IDENT
FUNCTOR := IDENT
VALUE := IDENT | _
NULLCON := IDENT
PRÄCON := IDENT | **ref** | *Standard-Präfix-Konstruktor*
INCON := IDENT | **::** | *Standard-Infix-Konstruktor*
PRÄFUN := IDENT | PRÄCON | ! | hd | tl | # | ≡ | *Standard-Präfix-Funktion*
INFUN := IDENT | INCON | ; | + | - | * | / | div | mod | = | < | > | <= | >= |
orelse | **andalso** | *Standard-Infix-Funktion*
TYPID := IDENT | TYPVAR IDENT | (TYPVARS) IDENT
TYPVAR := 'IDENT | ''IDENT

TYPVARS := TYPVAR | TYPVAR,TYPVARS

NAT := *natürliche Zahl*
CONST := NULLCON | () | **true** | **false** | NAT | NAT.NAT | *"String"* | nil | [] |
Standard-Konstante

DEF := VALDEF | FUNDEF | **infix** INFUN | EXCEPTDEF | TYPDEF | **open** STRUCT |
local DEFS **in** DEFS **end**
DEFS := DEF | DEF DEFS

VALDEF := **val** PAT = EXP | **val rec** PRÄFUN = EXP
FUNDEF := **fun** CLAUSES FUNDEFS
FUNDEFS := **and** CLAUSES FUNDEFS | ε
CLAUSE := PRÄFUN PAT = EXP | PAT INFUN PAT = EXP
CLAUSES := CLAUSE | CLAUSE ❙ CLAUSES
EXCEPTDEF := **exception** EXCEPT | **exception** EXCEPT **of** TYP

TYPDEF := **type** TYPID = TYP | **datatype** TYPID = CONDEFS |
abstype TYPID = CONDEFS **with** DEFS **end**
CONDEF := NULLCON | PRÄCON **of** TYP | INCON **of** TYP
CONDEFS := CONDEF ❙ CONDEF | CONDEFS

TYP := TYPID | TYPVAR | **int** | **string** | **bool** | **real** | **unit** |
TYPES | {RECORD} | TYP **list** | TYP **ref** | (TYP) |
TYP IDENT | (INSTANCES) IDENT
TYPES := TYP | TYP * TYPES | TYP − > TYPES
RECORD := VALUE:TYP | VALUE:TYP, RECORD
INSTANCES := TYP | TYP,INSTANCES

EXP := CONST | VALUE | STRUCT.VALUE |
PRÄFUN EXP | STRUCT.PRÄFUN EXP | EXP INFUN EXP |
VALUE := EXP | (EXPS) | [EXPS] | {RECEXP} |
if EXP **then** EXP **else** EXP | **case** EXP **of** MATCHES |
let DEFS **in** EXP **end** | PRÄFUN | **op** INFUN |
fn MATCHES | EXP EXP |
raise EXCEPT | **raise** EXCEPT EXP |
EXP **handle** MATCHES
EXPS := EXP | EXP,EXPS
RECEXP := VALUE = EXP | VALUE = EXP, RECEXP

MATCH := PAT => EXP
MATCHES := MATCH | MATCH ❙ MATCHES
PAT := CONST | VALUE | (PATS) | {RECPAT} | [PATS] |
PRÄCON PAT | PAT INCON PAT | VALUE **as** PAT
PATS := PAT | PAT,PATS
RECPAT := VALUE = PAT | VALUE = PAT, RECPAT

SIGNDEF := **signature** SIGN = **sig** SPECS **end**

SPEC := type IDENT | eqtype IDENT | val IDENT:TYP | **structure** STRUCT:SIGN
SPECS := SPECS SPEC | ε

STRUCTDEF := **structure** STRUCT = struct DEFS end |
structure STRUCT = FUNCTOR(DEFS) |
structure STRUCT = STRUCT
FUNCTORDEF := **functor** FUNCTOR(SPECS) = **struct** DEFS **end**

2 Typen, Ausdrücke und Definitionen

Alle traditionellen Programmiersprachen sind *imperativ*, d.h. befehlsorientiert, und *prozedural*, d.h. verfahrensorientiert. Die *Zuweisung* an eine *Programmvariable* ist der elementare Befehl einer imperativen Sprache. Man stellt sich dabei eine Speicherzelle vor mit einem Namen, der Programmvariablen, z.B. *x*, in den durch eine Zuweisung, z.B. *x:=5*, der Wert *5* eingesetzt wird. Durch die Zuweisung *x:=x+1* wird der Wert in der Zelle verändert: Aus *5* wird *6*.

Man sieht, daß den Konstrukten einer Programmiersprache ein Auswertungsmodell zugrundeliegt, im o.g. Fall das **von-Neumann-Modell**, für das die strikte Trennung zwischen Speicher, Recheneinheit und Steuerkomponenten charakteristisch ist, die durch Kanäle (*von-Neumann-bottleneck*) miteinander verbunden sind. Imperative Sprachkonstrukte sind ein direktes Abbild des von-Neumann-Modells. Die Programmvariable entspricht der Speicherzelle. Die Zuweisung beschreibt den Transport eines Wertes. Die Auswertung von Ausdrücken entspricht der Arbeit von Recheneinheiten. Bei dieser Sicht muß jede Aufgabenbeschreibung in eine *Befehlsfolge* transformiert werden, bevor der Rechner überhaupt etwas damit anfangen kann.

Erstens ist das imperative Modell, zumindest in seiner ausschließlichen Anwendung, wegen der Fortschritte auf dem Hardwaresektor nicht mehr realistisch. Zweitens verlangt der Entwurf großer Systeme in unterschiedlichsten Anwendungsbereichen stärker problem- und weniger maschinenorientierte Modelle.

Das funktionale Modell orientiert sich demgegenüber nicht an einer Maschine, sondern an der *Syntax* (Schreibweise) und der *Semantik* (Bedeutung) mathematischer Ausdrücke. Was beim imperativen Modell auf die arithmetischen Teile eines Programms beschränkt ist, wird hier zum grundlegenden Konzept. Daß das prinzipiell geht, wurde nicht von Informatikern entdeckt, sondern ist das Ergebnis einer mindestens 50-jährigen Entwicklung in der Mathematik, insbesondere der Algebra und der formalen Logik.

Bleiben wir zunächst bei mathematischen Ausdrücken wie dem folgenden:

$$((m+n) * abs(m-n)+1)\ div\ 2$$

m und n repräsentieren hier keine Speicherzellen, sondern sind Variablen im Sinne der formalen Logik, d.h. *Platzhalter* oder *Identifikatoren* für Werte. Weitere Komponenten des Ausdrucks sind *Konstanten (1* und *2*) sowie *Funktionen* (+, *, -, *abs* und *div*).

2.1 Funktionen in der Mathematik

Eine Funktion f ordnet Elementen einer *Quellmenge* (Definitionsbereich) A eindeutig Elemente einer *Zielmenge* (Wertebereich) B zu, geschrieben: $f : A \to B$. f ist **total**, wenn f jedem $a \in A$ ein $b \in B$ zuordnet. Andernfalls ist **f partiell**. Ist f n-stellig, dann ist A ein **Produkt**, z.B. $A_1 \times A_2 \times \cdots \times A_n$, und man nennt die Elemente von A **n-Tupel**, z.B. $a = (a_1, \ldots, a_n)$.

Die Definition von f ist

- **extensional**, d.h. eine Aufzählung der Paare (a, b) mit $f(a) = b$, die auch der *Graph* von f genannt wird,
- **intensional**, d.h. eine Menge von Regeln oder Gleichungen, die f beschreiben.

extensional	*intensional*
(2,4)	double(x) = 2*x
(3,6)	
(4,8)	
⋮	

extensional	*intensional*
(2,true)	even(x) = (x mod 2 = 0)
(3,false)	
(4,true)	
⋮	

Aus der Komposition von Funktionen entstehen **Terme** (= funktionale Ausdrücke). Sie dienen der (intensionalen) Definition weiterer Funktionen:

$$evenprod(x, y) = even(x * y)$$
$$sumbetween(m, n) = ((m + n) * (abs(m - n) + 1))\ div\ 2$$

Referentielle Transparenz nennt man die Forderung, daß die Bedeutung eines Ausdrucks aus der Bedeutung seiner Teilausdrücke berechnet werden kann.

Dieselbe Funktion kann i.a. auf mehrere Arten definiert werden. Eine Definition von f ist eine **Syntax** für f, während f selbst als **Semantik** der Definition von f bezeichnet wird. Z.B. definieren die ML-Programme

```
fun    evenprod(x,y) = even(x*y)
fun    eithereven(x,y) = even(x) orelse even(y)
```

dieselbe Funktion. Man sagt auch, daß die beiden Funktion(sdefinition)en **äquivalent** sind.

2.2 Typen

Die syntaktische Unterscheidung von Datenbereichen bezeichnet man als **Typisierung**. Sie zwingt den Benutzer zur Präzision beim Definieren von Funktionen und ermöglicht der Maschine die Erkennung semantischer Fehler, sofern sie auf Typ-Widersprüche zurückzuführen sind. Das ML-System leitet den Typ von Daten und Funktionen aus den Kontexten ab, in denen sie vorkommen (**Typinferenz**). Auf die Eingabe einer Wert- oder Funktionsdefinition antwortet das ML-System mit dem jeweils abgeleiteten Typ (unten kursiv notiert).

2.2.1 Primitive Typen

Integers (ganze Zahlen)

```
5
```
val it = 5 : int

```
5+3
```
val it = 8 : int

```
37 div 5; 37 mod 5
```
val it = 7 : int
val it = 2 : int

Reals (Reelle Zahlen)

```
3.14
```
val it = 3.14 : real

```
7.2/4.8
```
val it = 1.5 : real

```
7/4.8
```
Error: operator and operand don't agree (tycon mismatch)
*operator domain: real * real*
*operand: int * real*
in expression:
/ (7,4.8)

```
sqrt(2.0)
```
val it = 1.41421356237331 : real

Strings (Zeichenketten, *Worte*)

```
"this is a string"
```
val it = "this is a string": string

```
""
```
val it = "": string

```
"a"; "b"
```
val it = "a": string
val it = "b": string

```
"a"^"b"
```
val it = "ab": string

```
size("a"^"b")
```
val it = 2 : int

Booleans *(Boolesche Werte, Wahrheitswerte)*

```
true; false
```
val it = true : bool
val it = false : bool

```
not(true)
```
val it = false : bool

```
5 = 5
```
val it = true : bool

```
5 = 3
```
val it = false : bool

Der Typ **unit** steht für einen besonderen Datenbereich, der genau ein Element enthält, nämlich:
```
()
```
val it = () : unit

Sind $e_1, \ldots, e_n$ Ausdrücke beliebigen, möglicherweise verschiedenen Typs, dann hat der Ausdruck $(e_1; \ldots; e_n)$ denselben Typ wie e_n. Das Semikolon bewirkt, daß die Terme $e_1, \ldots, e_n$ hintereinander ausgewertet werden.

2.2.2 Zusammengesetzte Typen

Tupel

```
(true,1)
```
*val it = (true,1) : bool * int*

```
(2)
```
val it = 2 : int

```
((2,3,4),(true,4.1))
```
*val it = ((2,3,4),(true,4.1)) : (int * int * int) * (bool * real)*

```
(size"abc",size"de")
```
*val it = (3,2) : int * int*

Records

Gibt man den Komponenten eines Tupels Namen, dann spricht man von *Records*.

```
{a=true,b=4,c="hallo"}
```
val it = {a=true,b=4,c="hallo"} : {a:bool, b:int, c:string}

Listen und *Zeiger* bilden weitere zusammengesetzte Standardtypen. Sie werden in späteren Kapiteln ausführlich behandelt.

2.3 Definitionen

2.3.1 Werte

```
val a = 5+3
```
val a = 8 : int

```
val pair = (size"abc",size"de")
```
*val pair = (3,2) : int * int*

2.3.2 Funktionen

```
fun double(x:int) = x+x
```
val double = fn : int → int

```
fun treble(x) = 3*x
```
val treble = fn : int → int

```
fun sixtimes(x) = double(treble(x))
```
val sixtimes = fn : int → int

```
fun digit(d) = ord(d) >= ord"0" andalso ord(d) <= ord"9"
```
val digit = fn : string → bool

```
fun div_mod(x,y) = (x div y, x mod y)
```
*val div_mod = fn : int * int → int * int*

```
fun even(x) = (x mod 2) = 0
```
val even = fn : int → bool

```
fun evenprod(x,y) = even(x*y)
```

*val evenprod = fn : int * int → bool*

2.4 Fallunterscheidungen

2.4.1 Konditionale

```
fun min(x:int,y) = if x < y then x else y
```
*val min = fn : int * int → int*

```
if 5 = 5 then 2 else 3 div 0
```
val it = 2 : int
Der Ausdruck *3 div 0* wird hier nicht ausgewertet!

```
if 5 = 5 then 2 else 3.0 div 0
```
Error: operator and operand don't agree (tycon mismatch)
*operator domain: int * int*
*operand: real * int*
in expression:
div (3.0,0)
Der Ausdruck *3.0 div 0* wird auch hier nicht ausgewertet, aber ein Typfehler wird erkannt. *div* erwartet ein Argument des Typs $int * int$ (*operator domain*), erhält aber ein Argument des Typs $real * int$ (*operand*).

```
fun minmax(x:int,y) = if x < y then (x,y) else (y,x)
fun min(z) = #1(minmax(z))
fun max(z) = #2(minmax(z))
min(5,3)
```
*val minmax = fn : int * int → int * int*
*val min = fn : int * int → int*
*val max = fn : int * int → int*
val it = 3 : int

2.4.2 Argumentmuster

```
infix or
fun true or c = true          oder      fun b or c = case b of true => true
|       _ or c = c                                          |  _ => c
infix And
fun true And c = c            oder      fun b or c = case b of true => c
|       _ And c = false                                     |  _ => false
```
*val or = fn : bool * bool → bool*
*val And = fn : bool * bool → bool*

```
fun safediv(m,n) = case n of 0 => raise undefined
                           |  _ => m div n
```
*val safediv = fn : int * int → int*

```
val x = safediv(5,2)
```
val x = 2 : int

```
val x = safediv(5,0)
```
uncaught exception undefined

Musterdefinitionen sind Wertdefinitionen:

```
val (x,y) = (5,3)
```
val x = 5 : int
val y = 3 : int

```
val (x,(y,b)) = (7,(3+2,true))
```
val x = 7 : int
val y = 5 : int
val b = true : bool

2.5 Rekursive Definitionen

Füge n Kopien des Strings s zu einem String zusammen:

```
fun copy(n,s) = if n = 0 then ""else s^copy(n-1,s)
```
*val copy = fn : int * string → string*

Dies ist eine **linear-rekursive** Definition, weil *copy* auf der rechten Seite der Definitionsgleichung genau einmal aufgerufen wird. Für alle natürlichen Zahlem m und n gilt:

$$copy(m+n,s) = copy(m,s)^{\wedge}copy(n,s)$$
$$copy(m*n,s) = copy(m, copy(n,s))$$

copy ist eine partielle Funktion, weil die Auswertung von Aufrufen der Form $copy(\sim n, s)$[1] nicht terminiert. Mathematisch ausgedrückt: Die Funktion *copy* ist nur auf einer Teilmenge ihres Definitionsbereiches $int * string$ definiert. Die Einführung einer **Ausnahme** (*exception*) macht die *Undefiniertheitsstellen* von *copy* explizit:

exception undefined
```
fun copy(n,s) = if n < 0 then raise undefined
                else if n = 0 then "" else copy(n-1,s)^s
```

Beispiel 2.5.1 Summe der ersten n natürlichen Zahlen:

```
fun sum(0) = 0
|   sum(n) = n + sum(n-1)
 val sum = fn : int → int
```

Beispiel 2.5.2 Fibonacci-Folge Wieviele Kaninchenpaare (KPs) gibt es im n-ten Monat, wenn es im ersten Monat genau ein KP gibt und jedes KP vom zweiten Lebensmonat an jeden Monat ein KP erzeugt?

```
fun fib(0) = 0
|   fib(1) = 1
|   fib(n) = fib(n-1) + fib(n-2)
```

Diese Definition ist **baumartig-rekursiv**, weil fib auf der rechten Seite einer Definitionsgleichung mehrmals aufgerufen wird. Eine nicht-rekursive Definition von fib lautet wie folgt:

$$\text{fib(n)} = die\ (1/\sqrt{5}) * ((\sqrt{5}+1)/2)^n\ nächstliegende\ ganze\ Zahl.$$

[1] $\sim$ ist das negative Vorzeichen in ML.

Hier ist eine linear-rekursive Definition von fib:

```
fun fib1(n) = #1(fibAux(n))
and fibAux(0) = (0,1)
|   fibAux(n) = pair(fibAux(n-1))
and pair(x,y) = (y,x+y)
```

val fib1 = fn : int → int
*val fibAux = fn : int → int * int*
*val pair = fn : int * int → int * int*
Beispiel einer Berechnungsfolge:

```
fib1(3) = #1(fibAux(3))
        = #1(pair(fibAux(2)))
        = #1(pair(pair(fibAux(1))))
        = #1(pair(pair(pair(fibAux(0)))))
        = #1(pair(pair(pair(0,1))))
        = #1(pair(pair(1,1)))
        = #1(pair(1,2))
        = #1(2,3)
        = 2
```

Die erzeugten Aufrufe der Hilfsfunktion *pair* erst auf- und dann wieder abgebaut. Jeder Aufruf von *fibAux* führt zu höchstens einem weiteren Aufruf von *fibAux*. Das ist charakteristisch für lineare Rekursion.

Ein besonderer Fall linearer Rekursion ist die **repetitive Rekursion**, **Iteration** oder **Schleife**. Hier sind die rekursiven Aufrufe in keine weiteren Funktionsaufrufe eingebettet:

```
fun fib2(n) = fibAcc(n,0,1)
and fibAcc(0,x,y) = x
|   fibAcc(n,x,y) = fibAcc(n-1,y,x+y)
```

Beispiel einer Berechnungsfolge:

```
fib2(3) = fibAcc(3,0,1)
        = fibAcc(2,1,1)
        = fibAcc(1,1,2)
        = fibAcc(0,2,3)
        = 2
```

Beispiel 2.5.3 Repetitive Definition der Summe (vgl. 2.5.1):

```
fun sum1(n) = sumAcc(n,0)
and sumAcc(0,x) = x
|   sumAcc(n,x) = sumAcc(n-1,x+n)
```

Übersetzung von Integers in Strings:

```
fun stringofnat(n) = if n < 10
                     then chr(ord"0"+n)
                     else stringofnat(n div 10)^chr(ord"0"+(n mod 10))
```

```
fun stringofint(n) = if n < 0
                     then "~"^stringofnat(~n)
                     else stringofnat(n)
```
val stringofnat = fn : int → string
val stringofint = fn : int → string

2.6 Scopes und lokale Definitionen

Der *Scope* (Gültigkeitsbereich) eines Wertes bzw. einer Funktion beginnt auf der rechten Seite der jeweiligen Definitionsgleichung bzw. hinter den Definitionsgleichungen und endet vor der nächsten Definition, die denselben Namen trägt. Der Scope von mit *and* simultan definierten Werten bzw. Funktionen[2] beginnt gleichzeitig auf den rechten Seiten aller bzw. hinter allen Definitionsleichungen. Der Scope der Parameter einer Funktion ist i.a. die rechte Seite der jeweiligen Definitionsgleichung.

```
val x = 2
```
val x = 2 : int
```
val x = 3 and y = x
```
val x = 3 : int
val y = 2 : int
```
val (x,y) = (4,x)
```
val x = 4 : int
val y = 3 : int
```
fun g(0) = 0
|   g(n) = f(n-1)
and f(0) = 1
|   f(n) = g(n-1)
val n = g(5)
```
val n = 1 : int
```
val n = f(5)
```
val n = 0 : int

Das *let*-**Konstrukt** dient der lokalen Definition von Werten und Funktionen. Z.B. wird die mehrfache Berechnung von `ord(d)` in:

```
fun digit(d) = ord(d) >= ord"0" andalso ord(d) <= ord"9"
```

vermieden, wenn man das *let*-Konstrukt verwendet:

```
fun digit(d) = let val n = ord(d) in n >= ord"0" andalso n <= ord"9" end
```

Beispiel 2.6.1 Eine Zahl n wird in ein "Fenster"der Länge w geschrieben. Paßt sie nicht hinein, soll $*$ w-mal kopiert werden (vgl. Bsp. 2.5.3).

```
fun windowint(w,n) = let val intrep = stringofint(n)
                         val intwidth = String.length(intrep)
                     in  if w < intwidth
                         then copy(w,"*")
                         else copy(w-intwidth," ")^intrep end
```

[2] Es können nur *entweder* Wertdefinitionen *oder* Funktionsdefinitionen mit *and* verknüpft werden.

Beispiel 2.6.2 Linear-rekursive Definition von *fib* (vgl. 2.5.2):

```
fun fib1(n) = let val (x,_) = fibAux(n) in x end
and fibAux(0) = (0,1)
|   fibAux(n) = let val (x,y) = fibAux(n-1) in (y,x+y) end
```
val fib1 = fn : int → int
*val fibAux = fn : int → int * int*
Analog können Funktionen lokal definiert werden:

```
fun  fib1(n) = let fun fibAux(0) = (0,1)
                   |   fibAux(n) = let val (x,y) = fibAux(n-1)
                           in (y,x+y) end
                   val (x,_) = fibAux(n)
               in x end
```

oder vorangestellt mit *local*:

```
local fun fibAux(0) = (0,1)
      |   fibAux(n) = let val (x,y) = fibAux(n-1) in (y,x+y) end
in fun fib1(n) = let val (x,_) = fibAux(n) in x end
end
```

2.7 Polymorphie

```
fun id(x) = x
```
val id = fn : 'a → 'a

'a heißt **Typvariable**, *'a → 'a* **polymorpher Typ**. Da der Typ von *id* eine Typvariable enthält, nennt man *id* eine *polymorphe Funktion*. Das ML-System ermittelt aus dem Kontext, in dem Funktionen aufgerufen werden, deren jeweils *allgemeinsten* Typ (s.u.). Das nennt man Typableitung oder **Typinferenz**.

```
fun f(x,y,z) = (x*3,y,z^"hallo")
```
*val f = fn : int * 'a * string → int * 'a * string*

Der Typ von *f* ist eine **Instanz** des Typs *int * 'a * 'b → int * 'a * 'b*, der wiederum eine Instanz des Typs *'a * 'b * 'c → 'a * 'b * 'c* ist. Anstelle von *t ist eine Instanz von t'* sagt man auch: *t' ist allgemeiner als t*. Typen ohne Typvariablen heißen **monomorph**.

Eine Funktion *p* des Typs *'a → unit* heißt **Prozedur**, weil es bei ihr nicht um den berechneten Wert geht (der immer () ist, s. 4.2.1) sondern um – beabsichtigte – Seiteneffekte bei der Ausführung von *p*.

2.8 Overloading

ist die mehrfache Vergabe desselben Namens an mehrere Funktionen mit unterschiedlichen Typen, die nicht in der o.g. Instanzbeziehung zueinander stehen. Beispiele für *überladene* Funktionen sind diejenigen arithmetischen Funktionen, die sowohl auf *int* als auch auf *real* definiert sind.

```
fun f(x,y) = x+y
```
Error: overloaded variable "+"cannot be resolved
```
fun f(x,y) = x+y+3
```

*val f = fn : int * int → int*
Die zweite Definition von f ist korrekt, weil der Typ von f aus dem definierenden Ausdruck ermittelt werden kann.

```
fun f(x,y) = if x = y then 2 else 3
```
*val f = fn : 'a * 'a → int*
Die zweigestrichene Typvariable *'a* kann nur durch solche Typen ersetzt werden, auf denen die Funktionen = und <> (*ungleich*) definiert sind. *'a* wurde hier abgeleitet, weil die Definition von f die Funktion = verwendet.

```
fun g(x) = f(x,"AAA")
```
val g = fn : string → int
Oben ist festgelegt, daß beide Argumente von f denselben Typ haben. Also muß hier x den Typ *string* haben.

2.9 Funktionen höherer Ordnung

sind Funktionen, die als Argumente oder Werte mindestens eine Funktion haben. → ist neben ∗ (s. 4.2.2) ein wesentlicher **Typkonstruktor**. Mit ∗ werden kartesische Produkte gebildet, mit → Funktionenräume. Eine Funktion höherer Ordnung läßt sich daran erkennen, daß ihr Typ mindestens zweimal den Typkonstruktor → enthält.

Beispiel 2.9.1 Komposition zweier Funktionen:

```
infix o
fun (f o g)(x) = f(g(x))
```
*val o = fn : ('b → 'c) * ('a → 'b) → ('a → 'c)*
Die letzten beiden Klammern können entfallen, weil → immer *rechtassoziativ* gelesen wird.

Beispiel 2.9.2 Iteration nennt man die wiederholte Anwendung einer Funktion f, solange ein *Prädikat* p gilt. Prädikate werden als Boolesche Funktionen implementiert:

```
fun loop(p)(f)(x) = if p(x) then loop(p)(f)(f(x)) else x
```
val loop = fn : ('a → bool) → ('a → 'a) → 'a → 'a

Berechnung der kleinsten natürlichen Zahl, für die p nicht gilt:

```
fun least(p) = let fun plus1(n) = n+1
               in loop(p)(plus1)(0) end
```
val least = fn : (int → bool) → int

Berechnung der kleinsten natürlichen Zahl $\geq \sqrt{x}$:

```
fun intSqrt(x) = let fun p(n) = (n+1)*(n+1) < x
                 in least(p) end
val n = intSqrt(25)
```
val intSqrt = fn : int → int
val n = 5 : int

Den Übergang von einem funktionalen Typ *'a1 * 'a2 * ... * 'an → 'b* zum Typ *'a1 → 'a2 → ⋯ → 'b* nennt man **Currying**:

```
fun curry3(f)(x)(y)(z) = f(x,y,z)
```
*val curry3 = fn : ('a * 'b * 'c → 'd) → 'a → 'b → 'c → 'd*

```
fun uncurry3(f)(x,y,z) = f(x)(y)(z)
```
*val uncurry3 = fn : ('a → 'b → 'c → 'd) → 'a * 'b * 'c → 'd*

Für alle $n \in \mathbb{N}$ sind $curry_n$ und $uncurry_n$ demnach invers zueinander.

2.10 λ-Abstraktion

```
fun double(x) = 2*x
```

ist die Wertdefinition

```
val double = fn(x)=>2*x.
```

Die Syntax dieser Definitionen entstammt dem **λ-Kalkül**. Man schreibt dort $\lambda x.2*x$ anstelle von $fn(x) => 2*x$ und nennt Ausdrücke der Form $\lambda x.e$ **λ-Abstraktionen**. Der Wert einer λ-Abstraktion ist immer eine Funktion. Auf diese Weise können Funktionen auf Argumente angewendet werden, ohne daß jene vorher benannt werden müssen. So hat $(fn(x) => 2 * x)(5)$ wie $double(5)$ den Wert 10. Allgemein definiert

```
fun f(x1,...,xn) = e
```

dieselbe Funktion wie

```
val f = fn(x1,...,xn)=>e.
```

Enthält e einen rekursiven Aufruf von f, dann muß das Schlüsselwort *rec* vor den Funktionsnamen gesetzt werden:

```
val rec f = fn(x1,...,xn)=>e.
```

Z.B. ist

```
fun fact(0) = 1
|   fact(x) = x*fact(x-1)
```

äquivalent zu

```
val rec fact = fn(0) => 1
               |  x  => x*fact(x-1)
```

Auch simultane Definitionen sind möglich:

```
val rec even = fn(0) => true | x => odd(x-1)
and     odd = fn(0) => false | x => even(x-1)
val even = fn : int → bool
val odd = fn : int → bool
```

Genaugenommen sind die letzten beiden Funktionen definiert als **Lösung** des auf *val rec* folgenden **Gleichungssystems** in den Variablen *even* und *odd*. Über die Existenz solcher Lösungen denkt man in der mathematischen Logik nach.

2.11 Eifrige und verzögerte Auswertung

Eine (partielle) Funktion $f : A_1 \times A_2 \times \cdots \times A_n \to B$ heißt **strikt**, wenn jeder Ausdruck $f(e_1, \ldots, e_n)$ nur dann einen definierten Wert hat, wenn alle Teilausdrücke $e_1, \ldots, e_n$ definierte Werte haben. Andersrum: Ist einer der Terme $e_1, \ldots, e_n$ undefiniert (weil eine darin enthaltene Funktion partiell ist), dann ist auch $f(e_1, \ldots, e_n)$ undefiniert. Umgekehrt können nichtstrikte Funktionen manchmal auch dann ausgewertet werden, wenn gewisse Argumentausdrücke undefiniert sind, z.B. deshalb, weil deren Auswertung nicht terminiert. Bei der **Striktheitsanalyse** versucht ein Compiler herauszufinden, welche Argumente von f *needed* sind, d.h. auf jeden Fall

berechnet werden müssen, damit $f(e_1, \ldots, e_n)$ ausgewertet werden kann. Werden die anderen Argumente dann tatsächlich bei der Auswertung von $f(e_1, \ldots, e_n)$ übersprungen, spricht man von einer verzögerten oder **lazy** Auswertungsstrategie. Strikte Funktionen können nicht *lazy* ausgewertet werden, da sie ja immer alle ihre Argumente zur Wertberechnung brauchen. Aus strikten Funktionen bestehende Ausdrücke müssen also immer *bottom-up* (von innen nach außen) ausgewertet werden (eifrige oder **eager** Auswertungsstrategie).

In ML werden alle Funktionen als strikte Funktionen behandelt – außer den Booleschen Funktionen *andalso* ("und") und *orelse* ("oder") sowie dem Konditional $if - then - else$. Selbst diese Funktionen werden strikt behandelt, wenn man ihnen neue Namen gibt:

```
fun cond(b,x,y) = if b then x else y
val a = cond(true,5,3 div 0);
uncaught exception Div
```

Verzögerte Auswertung erreicht man dadurch, daß man diejenigen Argumente, die nicht bei jedem Aufruf ausgewertet werden sollen, entweder als Funktionen (*call-by-name*) oder als Zeiger (*call-by-reference*) übergibt. Auf beide Möglichkeiten werden wir noch ausführlich eingehen.

3 Korrektheitsbeweise

Funktionale Programme lassen sich leicht mit mathematischen Mitteln verifizieren. Die Basiskonstrukte einer funktionalen Sprache wie ML (Ausdrücke und Definitionen) werden nämlich auch in der Sprache der mathematischen Logik verwendet. Ausdrücke sowieso, aber auch Wert- und Funktionsdefinitionen, die im Grunde nichts anderes sind als **Gleichungen**, also Formeln der **Prädikatenlogik**. Deshalb können die Programme selbst ebenso wie Korrektheitsanforderungen an jene in derselben Sprache formuliert werden. Die präzise Bedeutung imperativer/objektorientierter Programme hingegen erschließt sich erst über **Zustandsmodelle**. Zur Formulierung und Verifikation von Aussagen über solche Programme verwendet man **dynamische Logiken**.[3]

Programmverifikation ist ein Teilgebiet der theoretischen Informatik, das allgemein in den Stammvorlesungen *Theorie der Programmierung* und *Logische Systeme der Informatik* behandelt wird, und, bezogen auf bestimmte Anwendungsgebiete, in *Künstlicher Intelligenz*, *Informationssystemen* und anderen Veranstaltungen zur Sprache kommt. Hier soll nur ein kleiner Einstieg in Begriffe und Methoden, die dort eine Rolle spielen, vermittelt werden.

Teilt man die Menge der durch Typen bezeichneten Datenbereiche grob ein, dann kann man unterscheiden zwischen **endlichen**, **diskreten** (abzählbaren) und **nichtdiskreten** (überabzählbaren) Datenmengen. Wir beschränken uns hier auf die ersten beiden. Ein Programm für eine extensional dargestellte (s. §2.1) endliche Funktion (= Funktion auf einer endlichen Menge) läßt sich sehr leicht verizieren. Man braucht ja nur für alle möglichen Eingaben zu testen, ob die jeweilige Ausgabe korrekt ist. Sobald die Eingabemenge unendlich ist, kann Verifikation nur darin bestehen, die Äquivalenz verschiedener intensionaler Darstellungen zu überprüfen. Dabei ist immer eine Darstellung die gegebene, deren Korrektheit vorausgesetzt wird, während eine andere dem Programm entspricht, das verifiziert werden soll. Es geht dann um den Beweis von Gleichungen zwischen Funktion(sdarstellungen). Sind die Funktionen partiell, dann werden daraus oft **bedingte Gleichungen**, wobei die Bedingung den Bereich der Daten, für die die Gleichung gelten soll, einschränkt. *Das* Mittel zum Beweis von Aussagen über unendliche diskrete Datenbereiche ist die **Induktion**. Es gibt viele unterschiedlich mächtige Varianten dieser Methode. Sie korrespondieren nicht nur zu bestimmten Datenmengen, sondern auch zum jeweiligen Typ der Rekursion, in der eine Funktion intensional dargestellt, also programmiert ist. Wir wollen hier zunächst die auf $\mathbb{N}$ zugeschnittene Methode der vollständigen Induktion und die allgemeinere Methode der Fixpunktinduktion kennenlernen, aber auch letztere zunächst nur auf $\mathbb{N}$ anwenden.

[3] Siehe Skript *Theorie der Programmierung*, Kapitel 10, http://ls5.cs.uni-dortmund.de/~peter/TdP.html.

3.1 Vollständige Induktion über $\mathbb{N}$

Sei $P(n)$ eine Eigenschaft natürlicher Zahlen, z.B. $sum(n) = sumAcc(n, 0)$. Für alle $n \in \mathbb{N}$ gilt $P(n)$, falls die Bedingungen

- $P(0)$, *Induktionsanfang*
- $\forall n > 0 : P(0) \wedge \cdots \wedge P(n-1) \;\Rightarrow\; P(n)$ *Induktionsschritt*

erfüllt sind. $P(0) \wedge \cdots \wedge P(n-1)$ heißt *Induktionsannahme* oder *Induktionsvoraussetzung (I.V.)*.

Beispiel 3.1.1 (vgl. Bspe. 2.5.1, 2.5.3)

$$\forall n : sum(n) = sum1(n) = sumAcc(n, 0).$$

folgt aus der *Verallgemeinerung*:

$$\forall n, x : sum(n) + x = sumAcc(n, x).$$

Wir definieren also:

$$P(n) \iff_{def} \forall x : sum(n) + x = sumAcc(n, x). \tag{1}$$

Beweis von (1):

$$\begin{array}{lll}
P(0) & \iff & \forall x : sum(0) + x = sumAcc(0, x) \\
 & \iff & \forall x : 0 + x = x \\
 & \iff & \forall x : x = x \\
 & \iff & true. \\
\forall n > 0 : P(n) & \iff & \forall x : sum(n) + x = sumAcc(n, x) \\
 & \iff & \forall x : n + sum(n-1) + x = \underline{sumAcc(n-1, x+n)} \\
 & \overset{I.V.}{\iff} & \forall x : n + sum(n-1) + x = \underline{sum(n-1) + x + n} \\
 & \iff & true.
\end{array}$$

Beispiel 3.1.2 (vgl. Bsp. 2.5.2)

$$\forall n : fib(n) = fib2(n) = fibAcc(n, 0, 1).$$

folgt aus der *Verallgemeinerung*:

$$\forall n, i : fib(n + i) = fibAcc(n, fib(i), fib(i + 1)).$$

Wir definieren also:

$$P(n) \iff_{def} \forall i : fib(n + i) = fibAcc(n, fib(i), fib(i + 1)). \tag{2}$$

Beweis von (2):

$$\begin{array}{lll}
P(0) & \iff & \forall i : fib(1 + i) = fibAcc(0, fib(i), fib(i + 1)) \\
 & \iff & \forall i : fib(i) = fib(i) \\
 & \iff & true. \\
\forall n > 0 : P(n) & \iff & \forall i : fib(n + i) = fibAcc(n, fib(i), fib(i + 1)) \\
 & \iff & \forall i : fib(n + i) = fibAcc(n - 1, fib(i + 1), fib(i) + fib(i + 1)) \\
 & \iff & \forall i : fib(n + i) = \underline{fibAcc(n - 1, fib(i + 1), fib(i + 2))} \\
 & \overset{I.V.}{\iff} & \forall i : fib(n + i) = \underline{fib(n - 1 + i + 1)} \\
 & \iff & true.
\end{array}$$

Wie erhält man die für einen Induktionsbeweis erforderlichen Verallgemeinerungen gewünschter Eigenschaften? Verallgemeinerungen lassen sich nicht systematisch konstruieren. Die folgende Alternative zur vollständigen Induktion erlaubt allerdings, bei einer bestimmten Form von P, kürzere Beweise und damit ein schnelleres Erkennen, ob ein Ansatz zur Verallgemeinerung falsch ist. Sie vermeidet insbesondere die für eine vollständige Induktion erforderliche Herleitung von Induktionsannahmen.

3.2 Fixpunktinduktion

Sei die Funktion $F : A \to B$ definiert durch k Gleichungen:

$$\begin{array}{lll} \texttt{fun} & \texttt{F(c}_1\texttt{)} & = \quad \texttt{g}_1\texttt{(c}_1\texttt{,F(u}_1\texttt{))} \\ & \ldots & \\ \mid & \texttt{F(c}_k\texttt{)} & = \quad \texttt{g}_k\texttt{(c}_k\texttt{,F(u}_k\texttt{))} \end{array}$$

wobei $g_1, \ldots, g_k$ Hilfsfunktionen sind. g_i wird auf c_i, d.i. das jeweilige Argument von F, und einen rekursiven Aufruf $F(u_i)$ angewendet. Das Folgende läßt sich leicht auch auf mehrere rekursive Aufrufe übertragen (wie in Beispiel 4 geschehen). Die Ausdrücke $g_i(c_i, F(u_i))$ werden zunächst wie folgt zerlegt:

$$\begin{array}{lll} \texttt{fun} & \texttt{F(c}_1\texttt{)} & = \quad \texttt{let val z = F(u}_1\texttt{) in g}_1\texttt{(c}_1\texttt{,z) end} \\ & \ldots & \\ \mid & \texttt{F(c}_k\texttt{)} & = \quad \texttt{let val z = F(u}_k\texttt{) in g}_k\texttt{(c}_k\texttt{,z) end} \end{array}$$

Diese Definition von F läßt sich direkt in ein **bedingtes Gleichungssystem** überführen: und in mathematische Form gebracht:

$$\begin{array}{lll} \forall z : \underline{F(u_1) = z} \;\Rightarrow\; \underline{F(c_1) = g_1(c_1, z)} & & r_1(EA_F) \\ \ldots & & \ldots \\ \forall z : \underline{F(u_k) = z} \;\Rightarrow\; \underline{F(c_k) = g_k(c_k, z)} & & r_k(EA_F) \end{array}$$

Damit haben wir die Definition von F in Aussagen über die **Ein-Ausgabe-Relation von** F:

$$EA_F(x, z) \iff_{def} F(x) = z$$

überführt. Wir ersetzen EA_F durch die *gewünschte* Ein-Ausgabe-Relation Q:

$$\begin{array}{lll} \forall z : Q(u_1, z) \;\Rightarrow\; Q(c_1, g_1(c_1, z)) & & r_1(Q) \\ \ldots & & \ldots \\ \forall z : Q(u_k, z) \;\Rightarrow\; Q(c_k, g_k(c_k, z)) & & r_k(Q) \end{array}$$

Satz 3.2.1 (Fixpunktinduktion) Für alle Prädikate Q gilt:

$$r_1(Q) \wedge \cdots \wedge r_k(Q) \quad \Rightarrow \quad \forall x : Q(x, F(x)).$$

Beweisidee: Man kann zeigen, daß EA_F das *kleinste* Prädikat ist, das $r_1(Q) \wedge \cdots \wedge r_k(Q)$ in Q *löst*, d.h.

- es gilt $r_1(EA_F) \wedge \cdots \wedge r_k(EA_F)$.
- Falls $r_1(Q) \wedge \cdots \wedge r_k(Q)$ gilt, dann ist EA_F in Q enthalten, d.h. für alle x, z folgt $Q(x, z)$ aus $EA_F(x, z)$. ❑

Beobachtungen:

- Mit Fixpunktinduktion wird die Korrektheit einer Ein-Ausgabebeziehung von F bewiesen.
- Im Gegensatz zu vollständiger Induktion braucht der Definitionsbereich von F keine Menge von Zahlen zu sein.
- F kommt in $r_1(Q) \wedge \cdots \wedge r_k(Q)$ nicht vor.

Beispiel 3.2.2 (vgl. Bspe. 2.5.1, 2.5.3, 3.1.1)

$$P(n) \iff_{def} \forall x : \underline{sum(n)} + x = sumAcc(n, x).$$

Hat $P(n)$ die Form $Q(n, sum(n))$? Ja, wenn Q wie folgt definiert wird:

$$Q(n, z) \iff_{def} \forall x : \underline{z} + x = sumAcc(n, x).$$

Zur Konstruktion der Bedingungen für die Fixpunktinduktion brauchen wir die Definition von sum:

```
fun sum(0) = 0
|   sum(n) = n + sum(n-1)
```

In zerlegter mathematischer Form:

$$\underline{sum(0)} = 0$$
$$\forall n > 0\ \forall z : \underline{sum(n-1)} = z \ \Rightarrow\ \underline{sum(n)} = n + z$$

Also bleibt zu zeigen:

$$Q(0, 0) \qquad\qquad r_1(Q)$$
$$\forall n > 0\ \forall z : Q(n-1, z) \ \Rightarrow\ Q(n, n+z) \qquad\qquad r_2(Q)$$

Beweis von $r_1(Q) \wedge r_2(Q)$:

$$\begin{array}{lll}
r_1(Q) & \iff & \forall x : 0 + x = sumAcc(0, x) \\
 & \iff & \forall x : x = x \\
 & \iff & true. \\
r_2(Q) & \iff & \forall n > 0\ \forall z : (\forall y : z + y = sumAcc(n-1, y)) \Rightarrow (\forall x : n + z + x = sumAcc(n, x)) \\
 & \iff & \forall n > 0\ \forall z : (\forall y : z + y = sumAcc(n-1, y)) \Rightarrow (\forall x : \underline{n} + z\underline{+x} = sumAcc(n-1, \underline{x+n})) \\
 & \iff & true.
\end{array}$$

Beispiel 3.2.3 (vgl. Bspe. 2.5.2, 3.1.2)

$$P(n) \iff_{def} \forall i : fib(n+i) = fibAcc(n, fib(i), fib(i+1)).$$

Hat $P(n)$ die Form $Q(n, fib(n))$? Nein, deshalb starten wir hier mit der ursprünglichen Bedingung und definieren:

$$P(n) \iff_{def} fib(n) = fibAcc(n, 0, 1).$$

Hat $P(n)$ jetzt die Form $Q(n, fib(n))$? Ja, wenn Q wie folgt definiert wird:

$$Q(n, z) \iff_{def} \underline{z} = fibAcc(n, 0, 1).$$

Zur Konstruktion der Bedingungen für die Fixpunktinduktion brauchen wir die Definition von fib:

```
fun fib(0) = 0
|   fib(1) = 1
|   fib(n) = fib(n-1) + fib(n-2)
```

In zerlegter mathematischer Form:

$$\underline{fib(0)} = 0$$
$$\underline{fib(1)} = 1$$
$$\forall n > 1\ \forall y, z : \underline{fib(n-1)} = y \wedge \underline{fib(n-2)} = z \ \Rightarrow\ \underline{fib(n)} = y + z$$

Also bleibt zu zeigen:

$$\begin{array}{ll} Q(0,0) & r_1(Q) \\ Q(1,1) & r_2(Q) \\ \forall n > 0\ \forall y, z : Q(n-1, y) \wedge Q(n-2, z) \ \Rightarrow\ Q(n, y+z) & r_3(Q) \end{array}$$

Beweis von $r_1(Q) \wedge r_2(Q) \wedge r_3(Q)$:

$$\begin{array}{lcl} r_1(Q) & \iff & 0 = fibAcc(0,0,1) \\ & \iff & 0 = 0 \\ & \iff & true. \\ r_2(Q) & \iff & 1 = fibAcc(1,0,1) \\ & \iff & 1 = fibAcc(0,1,1) \\ & \iff & 1 = 1 \\ & \iff & true. \\ r_3(Q) & \iff & \forall n > 1\ \forall y, z : y = fibAcc(n-1,0,1) \wedge z = fibAcc(n-2,0,1) \\ & & \qquad \Rightarrow\ y + z = fibAcc(n,0,1) \\ & \iff & \forall n > 1 : fibAcc(n-1,0,1) + fibAcc(n-2,0,1) = fibAcc(n,0,1) \\ & \iff & \forall n > 1 : fibAcc(n-2,1,1) + fibAcc(n-2,0,1) = fibAcc(n-1,1,1) \\ & \iff & \forall n > 1 : fibAcc(n-2,1,1) + fibAcc(n-2,0,1) = fibAcc(n-2,1,2) \\ & \iff & true. \end{array}$$

Der letzte Schritt setzt die Gültigkeit der Gleichung

$$fibAcc(n-2,1,1) + fibAcc(n-2,0,1) = fibAcc(n-2,1,2) \tag{3}$$

voraus. Um das zu beweisen, muß man wieder verallgemeinern. Ein üblicher Ansatz zur Verallgemeinerung ist die Ersetzung von Teilausdrücken durch Variablen. So wird (3) zu (4) verallgemeinert:

$$\forall n, x, y : fibAcc(n,x,y) + fibAcc(n,x',y') = fibAcc(n, x+x', y+y') \tag{4}$$

(4) läßt sich durch vollständige Induktion über n zeigen.

4 Listen

Eine Liste $[e_1, \ldots, e_n]$ ist eine endliche Folge variabler Länge, deren Elemente **denselben** Typ haben. Ein n-Tupel $(e_1, \ldots, e_n)$ ist demgegenüber Element eines n-fachen kartesischen Produktes $A_1 \times \cdots \times A_n$, hat also die feste Länge n. In ML hat die Liste $[e_1, \ldots, e_n]$ $n+1$ äquivalente Darstellungen:

```
      e1::e2::...::en-1::en::nil
oder  e1::e2::...::en-1::[en]
oder  e1::e2::...::[en-1,en]
      ...
oder  e1::[e2,...,en-1,en]
oder  [e1,e2...,en-1,en]
```

Beispiel einer Listendefinition:

```
val L = 1::[2,3,4]
```
val L = [1,2,3,4] : int list

4.1 Listenoperationen

ML stellt eine Reihe von Standardfunktionen auf Listen zur Verfügung.

Konkatenation zweier Listen

```
infix @
fun nil @ L = L
|   (x::L) @ L' = x::(L @ L')
```
*val @ = fn : 'a list * 'a list → 'a list*
Beispiel: [1,2,3]@[5,6,7] = [1,2,3,4,5,6]

Revertierung

```
fun rev(nil) = nil
|   rev(x::L) = rev(L)@[x]
```
val rev = fn : 'a list → 'a list
Beispiel: rev[1,2,3,5] = [5,3,2,1]

Kopfelement (head), Restliste (tail) und Länge

```
exception Hd and Tl
fun hd(nil) = raise Hd
|   hd(x::L) = x
fun tl(nil) = raise Tl
|   tl(x::L) = L
fun length(nil) = 0
|   length(x::L) = length(L)+1
```
val hd = fn : 'a list → 'a
val tl = fn : 'a list → 'a list
val length = fn : 'a list → int
Beispiele: hd[1,2,3,5] = 1 tl[1,2,3,5] = [2,3,5] length[1,2,3,5] = 4

Ist die Liste leer?

```
fun null(L) = L = nil
```
val null = fn : 'a list → bool
Beispiele: null[1,2,3,5] = false null[] = true

(n+1)-tes Element

```
exception Nth
fun nth(x::L,0) = x
|   nth(x::L,n) = nth(L,n-1)
|   nth _ = raise Nth
```
*val nth = fn : 'a list * int → 'a*
Beispiele: nth([1,2,3,5],3) = 5 nth([1,2,3,5],5) = Nth

Restliste vom (n+1)-ten Element an

```
exception Nthtail
fun nthtail(L,0)= L
|   nthtail(x::L,n) = nthtail(L,n-1)
|   nthtail _ = raise Nthtail
```
*val nthtail = fn : 'a list * int → 'a list*
Beispiele: nthtail([1,2,3,5],2) = [3,5] nthtail([1,2,3,5],4) = [] nthtail([1,2,3,5],5) = Nthtail

Anwendung einer Funktion *f : 'a → 'b* auf jedes Listenelement

```
fun map(f)(nil) = nil
|   map(f)(x::L) = f(x)::map(f)(L)
```
val map = fn : ('a → 'b) → 'a list → 'b list
Beispiel: map(fn(x)=>x+1)[1,2,3,5] = [2,3,4,6]

Faltung einer Liste zu einem Element

```
fun fold(f)(nil)(b) = b
|   fold(f)(x::L)(b) = f(x,fold(f)(L)(b))
```
*val fold = fn : ('a * 'b → 'b) → 'a list → 'b → 'b*
Die *zweistellige* Funktion $f :' a *' b \to' b$ wird, beginnend mit dem Anfangswert b, *rechtsassoziativ* auf die Listenelemente angewendet.

Beispiele von Faltungen:

Summe:	`fun sum(L) = fold(op + )(L)(0)`	z.B. sum[1,2,3,5] = 11
Produkt:	`fun prod(L) = fold(op * )(L)(1)`	z.B. prod[1,2,3,5] = 30
Konkatenation:	`fun conc(L) = fold(op @)(L)`	z.B. conc[[1,2],[2,3],[3,4],[5,6]] = [1,2,2,3,3,4,5,6]

`val mapconc = conc o map` ist äquivalent zu:
```
fun mapconc(f)(L) = fold(op @)(map(f)(L))(nil)
```
val mapconc = fn : ('a → 'b list) → 'a list → 'b list
Beispiel: mapconc(fn(x)=>[x,x+1])[1,2,3,5] = conc[[1,2],[2,3],[3,4],[5,6]] = [1,2,2,3,3,4,5,6]

Existiert in der ein Element mit der Eigenschaft *p : 'a → bool* ?

```
fun exists(p)(nil) = false
|   exists(p)(x::L) = p(x) orelse exists(p)(L)
```
val exists = fn : ('a → bool) → 'a list → bool
Wegen der verzögerten Auswertung von *orelse* (vgl. §2.11) bricht *exists* die Traversierung der Liste ab, sobald ein Element, für das p gilt, gefunden ist.

Ist x in L?

```
fun member(L)(x) = exists(fn(y)=>x=y)(L)
```
val member = fn : ''a list → ''a → bool
Äquivalente Definition:
```
fun member(nil)(y) = false
|   member(x::L)(y) = x = y orelse member(L)(y)
```

Gilt das Prädikat *p : 'a → bool* für alle Elemente der Liste?

```
fun forall(p)(nil) = true
|   forall(p)(x::L) = p(x) andalso forall(p)(L)
```
val forall = fn : ('a → bool) → 'a list → bool
Wegen der verzögerten Auswertung von *andalso* (vgl. §2.11) bricht *forall* die Traversierung der Liste ab, sobald ein Element, für das p nicht gilt, gefunden ist.
Äquivalente Definition:
```
fun forall(p)(L) = not(exists(fn(x)=>not(p(x)))(L))
```

***Komprehension* einer Liste zur Teilliste aller Elemente, die die Eigenschaft *p : 'a → bool* erfüllen:**

```
fun filter(p)(nil) = nil
|   filter(p)(x::L) = if p(x) then x::filter(p)(L) else filter(p)(L)
```
val filter = fn : ('a → bool) → 'a list → 'a list

Differenz zweier Listen:

```
fun diff(L1,L2) = filter(not o member(L2))(L1)
```
*val diff = fn : ”a list * ”a list → ”a list*
Beispiele: diff([3,4,5,6],[3,5]) = [4,6] diff([3,5],[3,4,5,6]) = []

4.2 Keller und Schlangen

Eine Liste L heißt *Keller (Stack)*, wenn nur mit den Funktionen *push, pop* und *top* auf L operiert wird, was man auch *LIFO-Strategie (last in first out)* nennt:

```
exception Pop and Top
fun push(x,K) = x::K
fun pop(x::K) = K
|   pop _ = raise Pop
fun top(x::K) = x
|   top _ = raise Top
```
*val push = fn : 'a * 'a list → 'a list*
val pop = fn : 'a list → 'a list
val top = fn : 'a list → 'a

Eine Liste Q heißt *Schlange (Queue)*, wenn nur mit den Funktionen *enqueue, dequeue* und *last* auf Q operiert wird, was man auch *FIFO-Strategie (first in first out)* nennt:

```
exception Dequeue and Last
fun enqueue(x,Q) = x::Q
fun dequeue[x] = nil
|   dequeue(x::Q) = x::dequeue(Q)
|   dequeue _ = raise Dequeue
fun last[a] = a
|   last(a::Q) = last(Q)
|   last _ = raise Last
```
*val enqueue = fn : 'a * 'a list → 'a list*
val dequeue = fn : 'a list → 'a list
val last = fn : 'a list → 'a

4.3 Listenerzeugung

Alle ganzen Zahlen zwischen m und n:

```
exception Interval
fun interval(m,n) = if m = n then [m]
                    else if m < n then m::interval(m+1,n)
                                  else raise Interval
```
*val interval = fn : int * int → int list*
z.B.: interval(5,8) = [5,6,7,8] interval(8,5) = Interval

Alle Primzahlen zwischen 2 und n (*Sieb des Eratosthenes*):

```
fun primes(n) = sift(interval(2,n))
and sift(nil) = nil
|   sift(x::L) = x::sift(filter(fn(y) => y mod x <> 0)(L))
```

```
val primes = fn : int → int list
val sift = fn : int list → int list
val L = primes(30)
val L = [2,3,5,7,11,13,17,19,23,29] : int list
```

4.4 Listensortierung

Sei $r :' a *' a \to bool$ eine Ordnung auf *'a*.
Ist eine Liste bzgl. *r* sortiert, d.h. gilt *r(x,y) = true* für alle aufeinanderfolgenden Elemente *x,y*?

```
fun sorted(r)(x::y::L) = r(x,y) andalso sorted(r)(y::L)
|   sorted _ _ = true
val sorted = fn : ('a * 'a → bool) → 'a list → bool
```

4.4.1 Sortieren durch Einfügen (*insertion sort*)

```
fun sort(r)(nil) = nil
|   sort(r)(x::L) = insert(r)(x,sort(r)(L))
and insert(r)(x,nil) = [x]
|   insert(r)(x,y::L) = if r(x,y) then x::y::L else y::insert(r)(x,L)
val sort = fn : ('a * 'a → bool) → 'a list → 'a list
val insert = fn : ('a * 'a → bool) → 'a * 'a list → 'a list
```

4.4.2 Sortieren durch Filtern (*quicksort*)

```
fun sort(r)(nil) = nil
|   sort(r)(x::L) = let val low = filter(fn(y)=>r(y,x))(L)
                        val high = filter(fn(y)=>not(r(y,x)))(L)
                    in sort(r)(low) @ x::sort(r)(high) end
val sort = fn : ('a * 'a → bool) → 'a list → 'a list
```

4.4.3 Sortieren durch Mischen (*mergesort*)

```
fun sort(r)(x::y::L) = let val (L1,L2) = split(L)
                       in merge(r)(sort(r)(x::L1),sort(r)(y::L2)) end
|   sort(r)(L) = L
and split(a::b::L) = let val (L1,L2) = split(L)
                     in (a::L1,b::L2) end
|   split(L) = (L,nil)
and merge(r)(nil,L) = L
|   merge(r)(L,nil) = L
|   merge(r)(x::L1,y::L2) = if r(x,y) then x::merge(r)(L1,y::L2)
                                       else y::merge(r)(x::L1,L2)
val sort = fn : ('a * 'a → bool) → 'a list → 'a list
val split = fn : 'a list → 'a list * 'a list
val merge = fn : ('a * 'a → bool) → 'a list * 'a list → 'a list
```

Der Aufruf *merge(r)([$x_1,\ldots,x_k$],[$y_1,\ldots,y_l$])* erzeugt $k + l \leq n$ rekursive Aufrufe von *merge*. Folglich bewirkt die baumartige Rekursion von *sort*, daß der Aufruf *sort(r)*[$x_1,\ldots,x_n$] insgesamt $(log_2 n) * n$ Aufrufe von *merge*

erzeugt. Bezüglich der Gesamtzahl rekursiver Aufrufe ist Mergesort also von *polynomiellem Aufwand.*

Verteilt man die *merge*-Aufrufe auf mindestens *n/2* Prozessoren verteilen, dann hat dieser Algorithmus nur noch *linearen Aufwand.* Das Mischen der von *split* erzeugten Teillisten könnte dann nämlich parallel erfolgen. Zunächst werden je zwei einelementige Listen gemischt, was jeweils zwei Schritte (rekursive Aufrufe von *merge*) erfordert. Daraus entstehen *n/2* zweielementige Listen, was jeweils vier Schritte erfordert. Daraus gehen vierelementige Listen hervor, usw., bis nur noch zwei *n/2*-elementige Listen zu mischen sind, wozu gerade n Schritte gebraucht werden. Die gesamte Schrittzahl beträgt also *2+4+...+n*, was kleiner als *2n* ist, so daß Mergesort tatsächlich linearen Aufwand bekommt.

Beispiel 4.4.4 Unter dem Gesichtspunkt der Parallelverarbeitung ist baumartige Rekursion der linearen vorzuziehen, wenn dadurch linearer zu logarithmischem Aufwand wird:

linear

```
fun fact(x) = if x = 0 then 1
              else x*fact(x-1)
```

logarithmisch

```
fun fact2(x) = prod(1,x)
and prod(x,y) = if x = y then x
                else let val mid = (x+y) div 2
                     in prod(x,mid)*prod(mid+1,y)
                     end
```

Aufwandsanalysen sind u.a. Gegenstand der LVs *Datenstrukturen* und *Effiziente Algorithmen.*

4.5 Induktion über Listen

Sei $P(L)$ eine Eigenschaft von Listen über $'a$, z.B. *sort(L) ist eine sortierte Permutation von L.* Vollständige Induktion (s. §3.1) läßt sich wie folgt auf den Datentyp $'a$ *list* übertragen. Für alle Listen L gilt $P(L)$, falls die Bedingungen

- $P(nil)$, *Induktionsanfang*
- $\forall L \neq nil : (\forall L' : length(L') < length(L) \Rightarrow P(L')) \;\Rightarrow\; P(L)$ *Induktionsschritt*

erfüllt sind. $\forall L' : length(L') < length(L) \Rightarrow P(L')$ heißt *Induktionsannahme* oder *Induktionsvoraussetzung (I.V.).*

Fixpunktinduktion (vgl. §3.2) hingegen kann ohne spezielle Anpassung an Listen direkt angewendet werden.

Beispiel 4.5.1 ***mergesort*** (s. §4.4.3)

$$P(L) \iff_{def} \forall L : sorted(\underline{sort(L)}) \wedge \underline{sort(L)} \sim L.$$

$L \sim L'$ bedeutet: L ist eine Permutation von L'. Hat $P(L)$ die Form $Q(L, sort(L))$? Ja, wenn Q wie folgt definiert wird:

$$Q(L, L') \iff_{def} sorted(\underline{L'}) \wedge \underline{L'} \sim L.$$

Zur Konstruktion der Bedingungen für die Fixpunktinduktion zerlegen wir die Definition von *sort* in bedingte Gleichungen:[4]

$$\underline{sort(nil)} = nil$$
$$\underline{sort([x])} = [x]$$
$$(L_1, L_2) = split(L) \wedge \underline{sort(x :: L_1)} = L_1' \wedge \underline{sort(y :: L_2)} = L_2' \;\Rightarrow\; \underline{sort(x :: y :: L)} = merge(L_1', L_2')$$

[4] Freie Variablen sind implizit allquantifiziert.

Also bleibt zu zeigen:

$$\begin{array}{ll} Q(nil, nil) & r_1(Q) \\ Q([x],[x]) & r_2(Q) \\ (L_1, L_2) = split(L) \wedge Q(x :: L_1, L_1') \wedge Q(y :: L_2, L_2') \;\Rightarrow\; Q(x :: y :: L, merge(L_1', L_2')) & r_3(Q) \end{array}$$

Beweis von $r_1(Q) \wedge r_2(Q)$:

$$\begin{array}{rcl} r_1(Q) & \iff & sorted(nil) \wedge nil \sim nil \\ & \iff & true. \\ r_2(Q) & \iff & sorted([x]) \wedge [x] \sim [x] \\ & \iff & true. \end{array}$$

Im folgenden Beweis von $r_3(Q)$ werden Lemmas verwendet, die Zusammenhänge zwischen Hilfsfunktionen und den Prädikaten *sorted* und $\sim$ wiedergeben:

Lemma 1 (*merge* und *sorted*):	$sorted(L_1) \wedge sorted(L_2) \;\Rightarrow\; sorted(merge(L_1, L_2))$
Lemma 2 (*merge* und $\sim$):	$merge(L_1, L_2) \sim L_1 @ L_2$
Lemma 3 (*split* und $\sim$):	$split(L) = (L_1, L_2) \;\Rightarrow\; L_1 @ L_2 \sim L$
Lemma 4 (Verträglichkeit von $\sim$ mit @):	$L_1 \sim L_3 \wedge L_2 \sim L_4 \;\Rightarrow\; L_1 @ L_2 \sim L_3 @ L_4$
Lemma 5 (Transitivität von $\sim$):	$L_1 \sim L \wedge L \sim L_2 \;\Rightarrow\; L_1 \sim L_2$

Beweis von $r_3(Q)$:

$$\begin{array}{lcl} r_3(Q) & \iff & (L_1, L_2) = split(L) \wedge sorted(L_1') \wedge L_1' \sim x :: L_1 \wedge sorted(L_2') \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; sorted(merge(L_1', L_2')) \wedge merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Zerlegung}{\Longleftarrow} & (L_1, L_2) = split(L) \wedge sorted(L_1') \wedge L_1' \sim x :: L_1 \wedge sorted(L_2') \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; sorted(merge(L_1', L_2')), \\ & & (L_1, L_2) = split(L) \wedge sorted(L_1') \wedge L_1' \sim x :: L_1 \wedge sorted(L_2') \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Lemma\ 1}{\Longleftarrow} & (L_1, L_2) = split(L) \wedge sorted(L_1') \wedge L_1' \sim x :: L_1 \wedge sorted(L_2') \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; sorted(L_1') \wedge sorted(L_2'), \\ & & (L_1, L_2) = split(L) \wedge sorted(L_1') \wedge L_1' \sim x :: L_1 \wedge sorted(L_2') \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Aussagenlogik}{\Longleftarrow} & (L_1, L_2) = split(L) \wedge sorted(L_1') \wedge L_1' \sim x :: L_1 \wedge sorted(L_2') \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Aussagenlogik}{\Longleftarrow} & (L_1, L_2) = split(L) \wedge L_1' \sim x :: L_1 \wedge L_2' \sim y :: L_2 \;\Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{split-Def.}{\Longleftarrow} & (x :: L_1, y :: L_2) = split(x :: y :: L) \wedge L_1' \sim x :: L_1 \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Lemma\ 3}{\Longleftarrow} & (x :: L_1) @ (y :: L_2) \sim x :: y :: L \wedge L_1' \sim x :: L_1 \wedge L_2' \sim y :: L_2 \\ & & \quad \Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Lemma\ 4}{\Longleftarrow} & (x :: L_1) @ (y :: L_2) \sim x :: y :: L \wedge L_1' @ L_2' \sim (x :: L_1) @ (y :: L_2) \\ & & \quad \Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Lemma\ 5}{\Longleftarrow} & L_1' @ L_2' \sim x :: y :: L \;\Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Lemma\ 2}{\Longleftarrow} & merge(L_1', L_2') \sim L_1' @ L_2' \wedge L_1' @ L_2' \sim x :: y :: L \;\Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Lemma\ 5}{\Longleftarrow} & merge(L_1', L_2') \sim x :: y :: L \;\Rightarrow\; merge(L_1', L_2') \sim x :: y :: L \\ & \overset{Aussagenlogik}{\Longleftarrow} & true \end{array}$$

Formulieren Sie die Beweisschritte verbal! Korrektheitsbeweise legen alle versteckten Annahmen (*Lemmas*) offen und helfen, Widersprüche im Entwurf zu entdecken.

5 Transformation und Verifikation

Allgemeine Methoden der Programmverifikation haben wir schon in Kap. 3 und §4.5 kennengelernt. In §5.4 wird speziell die Verifikation *iterativer* Programme behandelt (s. §2.5). Repetitive Rekursion ist nicht nur das am häufigsten verwendete und am leichtesten zu implementierende Programmschema, sondern erlaubt auch einen – aus der Fixpunktinduktion abgeleitete – besonderen Beweisansatz, der sich auf sog. **Programminvarianten** abstützt. Eine Invariante ist, kurz gesagt, eine sowohl vor Eintritt in einen Schleifenrumpf als auch nach Verlassen desselben gleichermaßen gültige Bedingung. Sie ist i.a. eine Verallgemeinerung der Ein/Ausgaberelation des iterativen Programms, das die Schleife enthält. Bevor wir auf die Invariantenmethode eingehen, wollen wir ein paar Transformationen rekursiver in äquivalente iterative Programme betrachten, wobei die Entrekursivierung durch Einführung verschiedener zusätzlicher Datenstrukturen erreicht wird.

5.1 Akkumulatoren

Wie man an den Beispielen 2.5.2 und 2.5.3 erkennt, besteht die Auswertung von Aufrufen einer iterativ-definierter Funktion f in der wiederholten Anwendung einer *Schleifenfunktion* f_{Acc} und zwar solange, bis deren Argumente eine Abbruchbedingung erfüllen. Die gegenüber f zusätzlichen Argumentstellen von f_{Acc} heißen *Akkumulatoren*, weil sie den jeweiligen Wert von f schrittweise akkumulieren (deutsch: anhäufen).

Die iterative Funktion $sum1$ (s. 2.5.3) ging hervor aus der *linear*-rekursiven Definition

```
fun  sum(0) = 0
|    sum(n) = n+sum(n-1)
```

Analog ergibt sich die iterative Funktion $fact1$:

```
fun  fact1(x) = factAcc(x,1)                    a:=1;
and  factAcc(0,a) = a          oder imperativ:  while x > 0 do (a:=a*x; x:=x-1);
|    factAcc(x,a) = factAcc(x-1,a*x)            fact1:=a
```

aus der linear-rekursiven Definition

```
fun  fact(0) = 1
|    fact(x) = x*fact(x-1)
```

der Fakultätsfunktion.[5]. Man kann diese Transformation wie folgt verallgemeinern. Seien $p : A \to bool$, $out, h : A \to B$, $h_1, h_2 : A \to A$ und $g : B \times B \to B$ Hilfsfunktionen, die bei der Definition von $F : A \to B$ verwendet werden. $p(x) = true$ kann auch bedeuten, daß $x \in A$ ein bestimmtes Muster hat, z.B., wenn A die Menge aller Listen ist:

$$p(L) = true \quad \Longleftrightarrow \quad \exists y, L' : L = y :: L'.$$

Ein Programm der Form

```
fun  F(x) = if p(x) then out(x) else g(F(h1(x)),h(x))
```

ist äquivalent zu:

```
fun  F1(x) = Floop(x,e)
and  Floop(x,acc) = if p(x) then g(f(x),acc) else Floop(h1(x),g(h(x),acc))
```

[5]Der Einsatz von Parallelrechnern würde wahrscheinlich eine andere Transformation nahelegen, z.B. von $fact$ zur *baumartig*-rekursiven Definition von $fact2$ (s. Bsp. 4.4.4)

falls g assoziativ ist und e ein rechtsneutrales Element bzgl. g ist, also für alle $x, y, z \in B$ die Gleichungen $g(g(x,y),z) = g(x,g(y,z))$ und $g(x,e) = x$ gelten.

5.2 Keller

Die eben genannte Transformation in ein iteratives Programm stellt also gewisse Bedingungen an die Ausgangsdefinition. Sie muß linear-rekursiv sein und die verwendete Hilfsfunktion g muß assoziativ sein und ein neutrales Element haben. Zur Übersetzung beliebiger Rekursion in Iteration reichen Akkumulatoren (die man sich ja als einzelne Register vorstellen kann) nicht aus. Zur Speicherung von Zwischenargumenten und/oder -werten der zu übersetzenden Funktion ist ein Keller erforderlich, also eine Liste veränderlicher Länge, auf die allerdings nur in der *LIFO*-Reihenfolge zugegriffen werden braucht (s. §4.2). Wir zeigen das Prinzip an folgendem Schema einer baumartig-rekursiven Funktion $F : A \to B$:

```
fun  F(x) = if p(x) then out(x) else g(F(h1(x)),F(h2(x)))
```

F ist äquivalent zu F_1:

```
fun  F1(x) = Floop([x],e)
and  Floop(nil,acc) = acc
|    Floop(x::L,acc) = if p(x) then FLoop(stack,g(out(x),acc))
                                else FLoop(h2(x)::h1(x)::L,acc)
```

falls e ein linksneutrales Element bzgl. g ist. Hier kann die sequentielle Bearbeitung der beiden Argumente $h1(x)$ und $h2(x)$ von F jedoch dazu führen kann, daß der Zeitaufwand gegenüber dem ursprünglichen Programm ansteigt. Dieser Effekt tritt nicht auf, wenn wir eine andere Transformation wählen, bei der nicht nur Argumente von F, sondern auch Werte von F im Keller zwischengespeichert werden. Dazu verallgemeinern wir das Schema von F ein wenig:

```
fun  F(x) = if p(x) then out(x) else g(x,F(h1(x)),F(h2(x)))
```

und leiten die Typen der Hilfsfunktionen f, g, h_1, h_2 aus dieser Definition ab: $g : C \times B \times B \to B$, $h_1, h_2 : A \to A$ und $out : A \to B$. Da Listen nur Elemente desselben Typs enthalten, müssen wir hier vorgreifen und zunächst einen *konstruktorbasierten Datentyp* definieren, der die Mengen A und B (disjunkt) vereinigt (s. Kap. 7):

```
datatype ('a,'b) SUM = arg of 'a | val of 'b
```

Der Keller hat dann den Typ $('a,'b)$ *SUM list*. F ist äquivalent zu F_2:

```
fun  F2(x) = Floop[arg(x)]
and  Floop(arg(x)::L) = if p(x) then FLoop(val(out(x))::L)
                                else FLoop(arg(h1(x))::arg(h2(x))::arg(x)::L)
|    Floop(val(y)::arg(x)::L) = Floop(arg(x)::val(y)::L)
|    Floop(val(z)::val(y)::arg(x)::L) = Floop(val(g(x,y,z))::L)
|    Floop[val(y)] = y
```

Aufgabe: Wenden Sie diese Transformation auf o.g. baumartig-rekursive Definitionen an!

Um die Äquivalenz zweier Funktion(sdefinition)en F und G nachzuweisen, zeigt man, daß für alle x im Definitionsbereich von F die Gleichung $F(x) = G(x)$ gilt, wobei die jeweiligen Annahmen (Assoziativität von g,

etc.) benutzt werden können. Ist G iterativ, dann kann man dazu die in §5.4 beschriebene Invariantenmethode verwenden.

5.3 Continuations

Hier wird die zu übersetzende Funktion F nicht mit Akkumulatoren oder einem Keller, sondern mit *Fortsetzungsfunktionen* versehen, die festlegen, wie die Werte von F weiterverarbeitet werden sollen. Formal: $F : A \to B$ geht über in $f_{Cont} : A \times (B \to B) \to B$ derart, daß für jede Fortsetzungsfunktion $c : B \to B$ gilt: $f_{Cont}(x, c) = c(f(x))$, also insbesondere: $f_{Cont}(x, fn(a) => a) = f(x)$.

Beispiel Fakultät (vgl. §5.1.1)

```
fun  fact2(x) = factCont(x,fn(a)=>a)
and  factCont(0,c) = c(1)
|    factCont(x,c) = factCont(x-1,fn(a)=>c(x*a))
```

Beispiel einer Berechnungsfolge von $fact2$:

```
fact2(3)
= factCont(3,fn(a)=>a)
= factCont(2,fn(a)=>a*3)
= factCont(1,fn(a)=>(a*2)*3)
= factCont(0,fn(a)=>((a*1)*2)*3)
= ((1*1)*2)*3
= (1*2)*3
= 2*3
= 6
```

Beispiel Mergesort (vgl. §4.4.3)

```
fun  sort1(r)(L) = sortCont(r)(L,fn(a)=>a)
and  sortCont(r)(x::y::L,c) = let val (L1,L2) = split(L)
                              in sortCont(r)(x::L1,fn(a)=>sortCont(r)(y::L2,
                                                   fn(b)=>c(merge(r)(a,b))))
                              end
|    sortCont(r)(L,c) = c(L)
```

Beispiel einer Berechnungsfolge von $sort1$ mit $r(x, y) = (x \leq y)$:

```
sort1(r)([5,1,3])
= sortCont(r)([5,1,3],fn(a)=>a)
= sortCont(r)([5,3],fn(a)=>sortCont(r)([1],
                     fn(b)=>merge(r)(a,b)))
= sortCont(r)([5],fn(a)=>sortCont(r)([3],
                  fn(b)=>sortCont(r)([1],
                  fn(c)=>merge(r)(merge(r)(a,b),c))))
= sortCont(r)([3],fn(b)=>sortCont(r)([1],
                  fn(c)=>merge(r)(merge(r)([5],b),c)))
= sortCont(r)([1],fn(c)=>merge(r)(merge(r)([5],[3]),c))
```

```
= merge(r)(merge(r)([5],[3]),[1])
= merge(r)(3::merge(r)([5],nil),[1])
= merge(r)(3::[5],[1])
= merge(r)([3,5],[1])
= 1::merge(r)([3,5],nil)
= 1::[3,5]
= [1,3,5]
```

ML-Programme werden vom Compiler zunächst in dieser Weise transformiert. Die mit Continuations versehenen Programme lassen sich direkter als die ursprünglichen in assemblerartige Befehlsfolgen übersetzen, ohne daß man sich hier – im Unterschied zur Kellertransformation – bereits auf bestimmte Datenstrukturen festlegt.

5.4 Verifikation iterativer Programme

Nochmal: Ein iteratives Programm besteht im Prinzip aus einer Funktion $F : A \to C$ und einer **Schleifenfunktion** $G : B \to C$, die von F aufgerufen wird. Das Programmschema hat i.a. folgende Form:

```
fun  F(x) = G(in(x))
and  G(y) = if p1(y) then G(loop_1(y))
            else if p2(y) then G(loop_2(y))
            ...
            else if pk(y) then G(loop_k(y))
            else out(y)
```

wobei $p_1, \ldots, p_k : B \to bool$, $in : A \to B$, $loop_1, \ldots, loop_k : B \to B$ und $out : B \to C$ Hilfsfunktionen sind und $y \in B$ aus mehreren Komponenten bestehen kann.

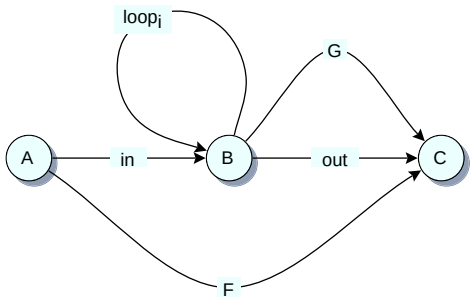


Figure 1. *Die Funktionen eines iterativen Programms*

Dieses Programmschema läßt sich direkt in eine (nichtrekursive) Prozedur übersetzen:

```
function F(x:A):C; var y:B;
                   begin y:=in(x);
                         while p1(y) or ... or pk(y)
                               do if p1(y) then y:=loop_1(y)
                                  else if p2(y) then y:=loop_2(y)
                                  ...
                                  else y:=loop_k(y)
                               od;
                         F:=out(y)
                   end
```

Die Definition von F entspricht einer Menge bedingter Gleichungen:

$$\begin{array}{rcl} G(in(x)) = z & \Rightarrow & F(x) = z \\ p_1(y) \wedge G(loop_1(y)) = z & \Rightarrow & G(y) = z \\ & \cdots & \\ \neg p_1(y) \wedge \cdots \wedge \neg p_{k-1}(y) \wedge p_k(y) \wedge G(loop_k(x)) = z & \Rightarrow & G(y) = z \\ \neg p_1(y) \wedge \cdots \wedge \neg p_k(y) & \Rightarrow & G(y) = out(y) \end{array}$$

Nach der Verallgemeinerung von Satz 3.2.1 auf mehrere Funktionen gilt eine Aussage der Form

$$\forall x : P(x, F(x)) \ \wedge \ \forall y : Q(y, G(y)),$$

falls P und Q die Bedingungen

$$\begin{array}{rcl} Q(in(x), z) & \Rightarrow & P(x, z) \\ p_1(y) \wedge Q(loop_1(y), z) & \Rightarrow & Q(y, z) \\ & \cdots & \\ \neg p_1(y) \wedge \cdots \wedge \neg p_{k-1}(y) \wedge p_k(y) \wedge Q(loop_k(x), z) & \Rightarrow & Q(y, z) \\ \neg p_1(y) \wedge \cdots \wedge \neg p_k(y) & \Rightarrow & Q(y, out(y)) \end{array}$$

erfüllen. Q heißt dann **Subgoal-Invariante für** P. Sie setzt die (Werte der) **Schleifenvariablen** y mit den Werten z von F in Beziehung. Üblicherweise zeigt man die Korrektheit iterativer Programme aber mithilfe einer **Hoare-Invariante** H. Diese ist ebenfalls ein zweistelliges Prädikat, das aber den Zusammenhang zwischen y und den *Argumenten* von F beschreibt. Die Anforderungen an H sind daher *dual* zu denen an Q:

$$\begin{array}{rcl} & & H(x, in(x)) \\ p_1(y) \wedge H(x, y) & \Rightarrow & H(x, loop_1(y)) \\ & \cdots & \\ \neg p_1(y) \wedge \cdots \wedge \neg p_{k-1}(y) \wedge p_k(y) \wedge H(x, y) & \Rightarrow & H(x, loop_k(y)) \\ \neg p_1(y) \wedge \cdots \wedge \neg p_k(y) \wedge H(x, y) & \Rightarrow & P(x, out(y)) \end{array}$$

Subgoal- und Hoare-Invariante lassen sich auseinander konstruieren:

$$Q(y, z) \quad \Longleftrightarrow_{def} \quad \forall x : H(x, y) \Rightarrow P(x, z).$$

$$H(x, y) \quad \Longleftrightarrow_{def} \quad \forall z : Q(y, z) \Rightarrow P(x, z).$$

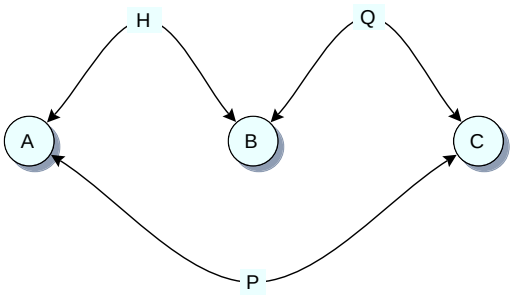


Figure 2. *Hoare- und Subgoal-Invariante*

In der Praxis findet man häufig eher eine Hoare-Invariante als eine Subgoal-Invariante. Da es ja um den Beweis der Korrektheit von F geht, also der Gültigkeit von $P(x, F(x))$, ist H bzw. Q nur ein Hilfsprädikat, das benötigt wird, weil F über die Schleifenfunktion G definiert wurde. Für die Programme $sum1$ (s. 2.5.3) und $fib2$ (s. 2.5.2) und das Prädikat

$$P(n, z) \ \Longleftrightarrow_{def} \ sum(n) = z \quad \text{bzw.} \quad P(n, z) \ \Longleftrightarrow_{def} \ fib(n) = z$$

sind z.B.

$$H(n, x, y) \ \Longleftrightarrow_{def} \ sum(n) = sum(x) + y \tag{5}$$

bzw.

$$H(n,x,y,z) \iff_{def} y = fib(n-x) \wedge z = fib(n-x+1) \quad (6)$$

Hoare-Invarianten. Wie werden auch in späteren Beispielen die Korrektheit eines iterativen Programms mithilfe einer Hoare-Invariante zeigen.

Aufgabe: Zeigen Sie, daß H in den Fällen (5) und (6) eine Hoare-Invariante ist!

Insgesamt wurden drei Methoden vorgeführt, um iterative Programme zu verifizieren: vollständige Induktion (§3.1), Fixpunktinduktion (§§3.2 und 4.5) und **Hoare-Induktion** (= Angabe einer Hoare-Invariante).

6 Funktionen als Datenstrukturen

Jede Funktionsdefinition ist immer auch ein *Algorithmus*, also ein Verfahren, *wie* die jeweiligen Funktionswerte berechnet werden sollen. Er zeigt sich zum einen in den Berechnungsfolgen, die durch Aufrufe der definierten Funktion erzeugt werden, und zum anderen in den jeweils verwendeten Hilfsfunktionen und Datenstrukturen. Wie man an den o.g. Programmtransformationen sieht, gehen Übergänge zwischen äquivalenten Definitionen i.a. einher mit der Einführung neuer Datenstrukturen (Akkumulatoren, Keller, Continuations, etc.). Eine Datenstruktur ist nicht schon dann geeignet, wenn sie wenig Platz verbraucht. Wichtiger ist oft der Zeitbedarf, der aber davon abhängt, welche Algorithmen sie benutzen, oder umgekehrt: wie gut sie auf diesen oder jenen Algorithmus zugeschnitten ist.

Das strenge Typkonzept funktionaler Sprachen erlaubt die Realisierung von Datenstrukturen auf mehreren Abstraktionsebenen. Auf höherer Ebene unterscheidet man eigentlich nur zwischen **Produkten** $A_1 \times \ldots \times A_n$, **Funktionstypen** $[A \to B]$, **Summen** $A_1 + \cdots + A_n$ – die den in Kap. 8 eingeführten konstruktorbasierten Typen entsprechen und zu denen die in Kap. 4 behandelten Listen gehören – sowie diversen Kombinationen aus Summen, Produkten und Funktionstypen. Hier die richtige Wahl zu treffen genügt in der Regel, um die Datenmanipulationen eines bestimmten Algorithmus präzise zu formulieren. Im Prinzip kann er damit auch "laufen", jedoch manchmal nur mit hohem Platz- und/oder Zeitbedarf. Dann besteht aber oft die Möglichkeit – unter Beibehaltung des algorithmischen Verfahrens – von ihm benutzten Datenstrukturen in eine andere Darstellung zu überführen, die den Aufwand verringert. Dies gilt insbesondere für diejenigen zahlreichen Datenstrukturen, die auf abstrakter Ebene Funktionen darstellen. Dazu gehören Felder, Matrizen, Tabellen, *Graphen*, *semantische Netze* u.ä.

So ist z.B. das in allen klassischen Programmiersprachen vorgesehene **n-dimensionale Feld** (*array*) auf abstrakter Ebene nicht anderes als eine n-stellige (partielle) Funktion $f : I_1 \times \ldots \times I_n \to A$, wobei die *Indexmengen* i.a. mit der Menge der ganzen Zahlen übereinstimmen. In ML ließe sich ein Feld als Objekt des folgenden polymorphen Typs auffassen:

```
type 'a feld = (int * ... * int) → 'a
```

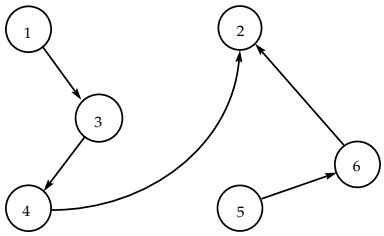
Charakteristisch für ein Feld ist der *direkte Zugriff* auf seine Elemente, also die folgenden Operationen:

```
fun  get(f,i) = f(i)
fun  update(f,i,a)(j) = if i = j then a else f(j)
```

Die Laufzeit eines Programms, das Felder benutzt, wird wesentlich bestimmt durch den Aufwand jedes einzelnen Aufrufs von *get* oder *update*. Ist der zu hoch, dann sollte man versuchen, jene Laufzeit dadurch zu beschleunigen, daß man die Definition von $'A\ feld$ ändert, diejenigen von *get* und *update* an die Änderung anpaßt, ansonsten aber das Programm läßt wie es ist. Das funktioniert natürlich nur, wenn alle Zugriffe des Programms auf Felder über die Operationen *get* und *update* laufen, also das Programm **modular** aufgebaut ist: $'A\ feld$, *get* und *update* bilden einen Modul, der nur die beiden Operationen **exportiert**.

6.1 Adjazenzmatrizen

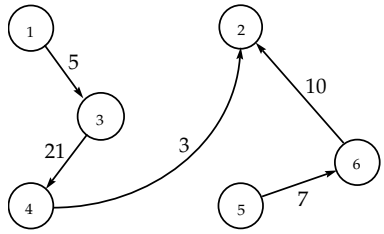
Ein **unmarkierter gerichteter Graph** G entspricht ebenfalls einer Funktion von $int \times int$ nach $bool$, wobei $G(i,j) = true$ genau dann gilt, wenn G eine Kante vom Knoten i zum Knoten j enthält. Man nennt diese Funktionsdarstellung die **Adjazenzmatrix** von G.



	1	2	3	4	5	6
1			t			
2						
3				t		
4		t				
5						t
6		t				

Figure 3. *Ein unmarkierter gerichteter Graph und seine Adjazenzmatrix*

Ein **markierter gerichteter Graph** G entspricht der partiellen Funktion von $int \times int$ nach A, deren Werte die jeweiligen Kantenmarkierungen liefern. Ist diese Funktion an der Stelle (i,j) definiert, dann enthält G eine Kante vom Knoten i zum Knoten j mit Markierung $G(i,j)$.



	1	2	3	4	5	6
1			5			
2						
3				21		
4		3				
5						7
6		10				

Figure 4. *Ein markierter gerichteter Graph und seine Adjazenzmatrix*

Der Graph von Fig. 4 läßt sich in ML als Funktion des Typs $int \times int \to int$ definieren:

```
exception no_path
fun G(1,3) = 5
|   G(3,4) = 21
|   G(4,2) = 3
|   G(5,6) = 7
|   G(6,2) = 10
|   G _ = raise no_path
```

*val G = fn : int * int → int*

Die Ausnahme *no_path* zeigt an, an welchen Stellen G undefiniert ist, d.h. zwischen welchen Knoten keine Kanten existieren (vgl. §2.5).

6.2 Kürzeste Wege

Wir denken an eine Anwendung, bei der die Knoten geographische Orte und die Kantenmarkierungen Weglängen zwischen den Orten repräsentieren. Für je zwei Orte i und j soll die Länge des kürzesten Weges von i nach j berechnet werden. Dazu verwenden wir den *Floyd-Warshall-Algorithmus*, der für jeden Knoten k prüft, ob ein Weg von i über k nach j existiert, der kürzer ist als die zuvor ermittelten Wege von i nach j, die nur Knoten der Menge $\{1_1, \ldots, 1_{k-1}\}$ passieren. Die gesamte Knotenmenge des Graphen sei $\{1, \ldots, n\}$.

Der Algorithmus führt diese Berechnung für alle Kanten gleichzeitig durch. In jedem Schritt wird der Graph

modifiziert und zwar so, daß zum Schluß eine Kante von i nach j genau dann mi lg markiert ist, wenn der kürzeste Weg von i nach j *im Ausgangsgraphen* die Länge lg hat. Um das zu erreichen, müssen wir zunächst im Ausgangsgraphen zwischen je zwei Knoten eine Kante ziehen und diese mit einer oberen Schranke für die entsprechende Weglänge versehen. Das geht wie folgt:

```
fun  transform(G)(i,j) = G(i,j) handle no_path => if i = j then 0 else 1000
```

Jede gegenüber G zusätzliche Kante zwischen zwei verschiedenen Knoten von $transform(G)$ wird mit 1000 markiert, weil wir davon ausgehen, daß diese Zahl von keiner später berechneten Weglänge überschritten wird. Außerdem wird eine Kante von jedem Knoten zu sich selbst gezogen und mit 0 markiert, weil das in diesem Fall bereits die kürzeste Weglänge ist.

In der Definition von *transform* wird die **Ausnahme** *no_path* **behandelt**: Bricht der durch den Aufruf $transform(G)(i,j)$ bewirkte Aufruf $G(i,j)$ durch den Aufruf der Ausnahme *no_path* ab (*raise no_path*), dann liefert $transform(G)(i,j)$ trotzdem einen definierten Wert, nämlich den des Ausdrucks e hinter *handle no_path =>*. Einen Ausdruck der Form

$$e \; handle \; except_1 \Rightarrow e_1 \mid \ldots \mid except_n \Rightarrow e_n$$

kann man sich als Fallunterscheidung

$$case \; e \; of \; except_1 \Rightarrow e_1 \mid \ldots \mid except_n \Rightarrow e_n$$

vorstellen, die aber nur die von e möglicherweise zurückgelieferten Ausnahmen behandelt.

Unter den o.g. Voraussetzungen liefert die Funktion $minGraph(G)$ den Graphen, dessen Kanten mit den jeweils kürzesten Wegen von G markiert sind:

```
fun  minGraph(G) = newGraph(transform(G),1)
and  newGraph(G,k) = if k > n then G else let fun G'(i,j) = min(G(i,j),G(i,k)+G(k,j))
                                         in newGraph(G',k+1)
                                         end
```

Der Aufruf $minGraph(G)$ erzeugt geschachtelte λ-Abstraktionen (s. §2.10):

$$\begin{array}{lll}
 & minGraph(G) & \\
= & newGraph(G_1, 1) & G_1 =_{def} transform(G) \\
= & newGraph(G_2, 2) & G_2 =_{def} \lambda(i,j).min(G_1(i,j), G_1(i,1) + G_1(1,j)) \\
 & \ldots & \\
= & newGraph(G_n, n) & G_n =_{def} \lambda(i,j).min(G_{n-1}(i,j), G_{n-1}(i,n_1) + G_{n-1}(n_1,j)) \\
= & newGraph(G_{n+1}, n+1) & G_{n+1} =_{def} \lambda(i,j).min(G_n(i,j), G_n(i,n) + G_n(n,j)) \\
= & G_{n+1} &
\end{array}$$

Erst nachdem alle λ-Abstraktionen $G_1, \ldots, G_{n+1}$ erzeugt worden sind, kann ein Ausdruck der Form $minGraph(G)(i,j)$ ausgewertet werden:

$$\begin{array}{ll}
 & minGraph(G)(i,j) = G_{n+1}(i,j) \\
= & min(G_n(i,j), G_n(i,n) + G_n(n,j)) \\
= & min(min(G_{n-1}(i,j), G_{n-1}(i,n) + G_{n-1}(n,j)), \\
 & \quad min(G_{n-1}(i,n), G_{n-1}(i,n) + G_{n-1}(n,n)) + min(G_{n-1}(n,j), G_{n-1}(n,n) + G_{n-1}(n,j))) \\
 & \ldots
\end{array}$$

Der eine Aufruf von G_{n+1} erzeugt 3 Aufrufe von G_n, von denen jeder jeweils drei Aufrufe von G_{n-1} erzeugt, usw., bis schließlich 3^n Aufrufe von $transform(G)$ produziert und ausgeführt werden. Die Darstellung von Graphen als funktionale Objekte bewirkt also, daß der Algorithmus **exponentiellen Aufwand** hat! Exponentieller Aufwand

sollte stets vermieden werden. Man erreicht das hier, indem man den Funktionstyp $int \times int \to int$ durch den Listentyp *int list list* ersetzt, der 2-dimensionale Matrizen repräsentiert. Ein Graph wird dabei durch seine Adjazenzmatrix (s. Fig. 4) repräsentiert. Eine entsprechend geänderte Definition von *minGraph* wird in §6.8 unter dem Namen *min_length* vorgestellt. *min_length* ersetzt die n-fache Schachtelung von λ-Abstraktionen durch (maximal) n Matrixmultiplikationen und hat deshalb nur noch **polynomiellen Aufwand**.

Der Vorteil der o.g. Version des Algorithmus besteht aber darin, daß diese Formulierung problemnäher ist und sich daher leichter seine Korrektheit zeigen läßt. Die Definition von *minGraph* ist iterativ, kann also mit Hoare-Induktion verifiziert werden. *newGraph* ist hier die Schleifenfunktion. Unter Verwendung der in §5.4 benutzten Bezeichnungen lauten die Ein/Ausgaberelation P von *minGraph* sowie eine Hoare-Invariante für P wie folgt:

$P(G, G') \iff_{def}$ G und G' sind markierte gerichtete Graphen mit der Knotenmenge $\{1, \ldots, n\}$, wobei jede Kante (i, j) von G' markiert ist mit der Länge des kürzesten Weges, der in G von i nach j führt.

$H(G, G', k) \iff_{def}$ G und G' sind markierte gerichtete Graphen mit der Knotenmenge $\{1, \ldots, n\}$, wobei jede Kante (i, j) von G' markiert ist mit der Länge des kürzesten Weges, der in G von i nach j führt und nur Knoten der Menge $\{1, \ldots, k-1\}$ passiert.

Aufgaben

- Zeigen Sie, daß H tatsächlich die in §5.4 allgemein formulierten Bedingungen an eine Hoare-Invariante erfüllt.
- Ändern Sie *minGraph* so, daß die Kanten des Ergebnisgraphen nicht mit der Länge der jeweils kürzesten Wege, sondern mit diesen selbst (dargestellt als Knotenlisten) markiert sind.

6.3 Adjazenzlisten

Für Graphalgorithmen, die keinen direkten Kantenzugriff erfordern, sondern einen Graphen von einem *Wurzelknoten* aus entlang seiner Kanten durchlaufen, ist die Darstellung als Funktion vom Typ $int \times int \to A$ ungeeignet. Stattdessen repräsentiert man in diesem Fall unmarkierte Graphen als Funktionen vom Typ

```
graph = int → int list
```

und markierte Graphen als Funktionen vom Typ

```
'a labGraph = int → ('a×int) list
```

Beide nennt man **Adjazenzlisten**-Darstellungen. $G : int \to (' a \times int)\ list$ bildet jeden Knoten x auf die Liste aller Paare (lab, y) ab, für die eine Kante von x nach y mit Markierung lab existiert.

Aufgabe: Definieren Sie Funktionen $fun2adlist : (int \times int \to' a) \to int \to (' a \times int)\ list$ und $adlist2fun : (int \to (' a \times int)\ list) \to int \times int \to' a$, die Adjazenzmatrizen in äquivalente Adjazenzlisten bzw. Adjazenzlisten in äquivalente Adjazenzmatrizen überführen.

Ein Algorithmus durchläuft einen Graphen entlang dessen Kanten, wenn er bestimmte Knoten oder Kanten sucht, um sie auszugeben oder zu verändern. Dabei ist nicht die Struktur des Graphen von Bedeutung, sondern nur die Menge der vom Wurzelknoten aus erreichbaren Knoten (oder Kanten). Im Prinzip gibt es drei Verfahren, diese Knoten aufzusammeln: Tiefensuche, Breitensuche und transitiver (Kanten-) Abschluß.

Die **Tiefensuche** speichert jeden erreichten Knoten x in der Liste V besuchter Knoten (*visited nodes*), springt zum ersten Element der Nachfolgerliste von x, fügt dieses zu V hinzu, usw., bis ein bereits in V vorhandener Knoten erreicht wird. Dann wird der Rest der zuletzt verlassenen Nachfolgerliste bearbeitet, usw., bis alle Nachfolgerlisten leer sind.

```
fun  depthfirst(G,V)(x) = if member(V)(x) then V else dfList(G,V@[x])(G(x))
and  dfList(G,V)(nil) = V
|    dfList(G,V)(x::L) = dfList(G,depthfirst(G,V)(x))(L)
```

depthfirst und *dfList* sind **wechselseitig-rekursiv** definiert. Da beide Funktionen auf der rechten Seite der letzten Gleichung geschachtelt vorkommen, ist dies insgesamt eine **geschachtelt-rekursive** Definition.

Beispiel 6.3.1 Graph vom Typ *graph* und die von *depthfirst* errechnete Liste der vom Knoten 1 aus erreichbaren Knoten:

```
fun G(1) = [2,3,4]
|   G(2) = [5,6]
|   G(3) = [7,8]
|   G(5) = [10]
|   G(6) = [10,11]
|   G(7) = [12,13]
|   G(9) = [1,2,3,5]
|   G _ = nil

val ds = depthfirst(G,nil)(1)
val ds = [1,2,5,10,6,11,3,7,12,13,8,4] : int list
```

Paulson (s. §1.2) verwendet eine einzige Funktion iterative (!) Funktion *depth*, aber zwei Listen L und V von Ausgangs- bzw. besuchten Knoten:

```
fun  depth(G,L) = depthLoop(G,L,nil)
and  depthLoop(G,nil,V) = V
|    depthLoop(G,x::L,V) = if member(V)(x) then depthLoop(G,L,V)
                                            else depthLoop(G,G(x)@L,x::V)
```

$depth(G, L)$ liefert alle von Knoten von L aus erreichbaren Knoten von G. Das läßt sich aus folgender allgemeinerer Aussage ableiten, die man leicht durch Fixpunktinduktion verifizieren kann:

(1) $depthLoop(G, L, V)$ besteht aus den Knoten von V und allen von Knoten von $L \setminus V$ erreichbaren Knoten von G.

Aufgaben

- Zeigen Sie (1) durch Fixpunktinduktion (s. §4.5).
- Ein Prädikat H sei wie folgt definiert:

  $$H(G, L, L', V) \iff_{def} \text{Jeder von } L \text{ aus erreichbare Knoten gehört zu } V \text{ oder ist von } L' \text{ auf einem Weg erreichbar, der keinen Knoten von } V \text{ passiert.}$$

  Zeigen Sie, daß H eine Hoare-Invariante für die Ein/Ausgaberelation von *depth* ist (s. §5.4).

depth läßt sich zu mithilfe einer Aufteilung der Knotenliste L in drei Listen $L1$, $L2$ und K zu einem Algorithmus abwandeln, der die Knoten von G **topologisch sortiert**, d.h. alle Knoten werden in einer Liste so angeordnet,

daß x nur dann vor y steht, wenn y von x aus erreichbar ist oder x und y *unvergleichbar* sind, d.h. weder x von y noch y von x aus erreichbar ist.

```
fun topsort(G,nil,nil,V,nil) = V
|   topsort(G,x::L1,L2,V,K) = if member(V@K)(x) then topsort(G,L1,L2,V,K)
                                                 else topsort(G,G(x),L1@L2,V,x::K)
|   topsort(G,nil,L2,V,x::K) = topsort(G,nil,L2,x::V,K)
|   topsort(G,nil,x::L2,V,nil) = topsort(G,x::L2,nil,V,nil)

fun topsortAll(G) = topsort(G,startL,nil,nil,nil)
```

Ist G azyklisch (!) und $startL$ irgendeine Aufzählung aller Knoten von G, dann liefert $topsortAll(G)$ eine topologisch sortierte Permutation von $startL$.

Aufgabe (schwierig): Zeigen Sie diese Behauptung durch Hoare-Induktion. Bilden die folgenden Bedingungen eine Hoare-Invariante $H(G, L1, L2, V, K)$ für *topsortAll*?

- V ist eine topologische Sortierung der Knoten von G, die nicht von $L1$ oder $L2$ aus erreichbar sind.
- $rev(K)$ is topologisch sortiert und enthält keinen Knoten mehrfach.
- Im Fall $K \neq nil$ sind alle Knoten von $L1$ von $hd(K)$ aus erreichbar.
- Es gibt keine Wege von V nach K.
- Alle von V und – im Fall $L1 = nil$ – von K aus erreichbaren Knoten gehören zu V oder K.

Die **Breitensuche** arbeitet auch auf zwei Knotenlisten L und V. Wie bei der Tiefensuche enthält V die besuchten Knoten. Die Liste *diff(L,V)* (s. §4.1) aller noch nicht in V vorhandenen Elemente von L wird zu V hinzugefügt. Anstatt jedoch mit dem ersten Element der Nachfolgerliste *eines* Knotens von L weiterzumachen, geht die Breitensuche gleichzeitig zu *allen* Elementen der Liste *MapUnion(G)(L)* der Nachfolger von Elementen von L. Alle Knoten des Graphen sind in V, sobald *diff(L,V)* leer ist.

```
fun  breadthfirst(G,V)(L) = let val L = diff(L,V)
                            in if null(L) then V
                               else breadthfirst(G,V@L)(MapUnion(G)(L))
                            end
and  MapUnion(G)(L) = fold(fn(L1,L2)=>L1@diff(L2,L1))(map(G)(L))(nil)
```

breadthfirst errechnet folgende Liste der vom Knoten 1 aus erreichbaren Knoten des Graph aus Bsp. 6.3.1:

```
val bs = breadthfirst(G,nil)[1]
```

val bs = [1,2,3,4,5,6,7,8,10,11,12,13] : int list

Einen Graphen G **transitiv abschließen** heißt von jedem Knoten aus eine Kante zu jedem von diesem aus erreichbaren Knoten ziehen. Das ist eine Transformation von G, zu deren schrittweiser Durchführung wir die Idee des Floyd-Warshall-Algorithmus von §6.2 nutzen können. Knoten werden wieder als Zahlen $1, \ldots, n$ dargestellt und für alle $1 \leq k \leq n$ bildet man den Graphen *newGraph(G,k)* aus dem Graphen G, indem man von jedem Knoten aus, zu dessen Nachfolgern k gehört, Kanten zu den Nachfolgern von k zieht.

```
fun  closure(G) = newGraph(G,1)
and  newGraph(G,k) = if k > n then G
                     else let fun G'(x) = let val L = G(x)
                                          in if member(L)(k) then Ldiff(G(k),L) else L end
                          in newGraph(G',k+1) end
```

closure errechnet folgende Liste der vom Knoten 1 aus erreichbaren Knoten des Graph aus Bsp. 6.3.1:

```
val cg = closure(G)(1)
```

val cg = [2,3,4,1,7,8,12,13,5,6,10,11] : int list

Wie *minGraph* hat auch *closure* exponentiellen Aufwand, der wieder durch die Matrixdarstellung von Graphen vermieden werden kann (s. §6.8).

6.4 Minimale Gerüste von Graphen

Der Algorithmus von Kruskal konstruiert ein minimales – auch Spannbaum genanntes – Gerüst eines kantengewichteten ungerichteten Graphen. Wir wollen ein ML-Programm entwickeln, das diesen Algorithmus realisiert. Dabei sollen auch in anderen Anwendungen benutzte Programmschemata *wiederverwendet* werden. Als Darstellungen des Ausgangsgraphen und des Spannbaums bieten sich Adjazenzmatrizen an (vgl. §6.1).

Zunächst soll aus G eine sortierte Liste seiner Kanten erzeugt werden. Das erfordert die Implementierung von drei Operationen:

(1) Zeilenweises Traversieren der Adjazenzmatrix.

(2) Umwandeln der Matrixeinträge $\neq 0$ in Kanten, dargestellt als Tripel (x, lab, y), wobei x der Quellknoten, lab die Markierung und y der Zielknoten der Kante ist.

(3) Sortierung der Kantenliste nach dem Gewicht der Kanten.

Alle drei Operationen lassen sich parallel durchführen, indem die Kantenliste gleich sortiert aufgebaut wird. Wir benutzen dazu die beiden Hilfsfunktionen *insert* und *merge* der Sortieralgorithmen *insertion sort* bzw. *merge sort*. Anstelle des schlichten Anfügens einer neuen Kante mit :: verwenden wir *insert*, anstelle der Konkatenation zweier Kantenlisten mit @ benutzen wir *merge*.

```
fun insert(r)(x)(nil) = [x]
|   insert(r)(x)(y::L) = if r(x,y) then x::y::L else y::insert(r)(x)(L)

fun merge(r)(nil,L) = L
|   merge(r)(L,nil) = L
|   merge(r)(x::L1,y::L2) = if r(x,y) then x::merge(r)(L1,y::L2)
                                       else y::merge(r)(x::L1,L2)
```

Man beachte, daß diese beiden Funktionen sortierte Listen stets auf sortierte Listen abbilden.

Zur Erzeugung der einzelnen Kanten beim Traversieren der Matrix verwenden wir eine Funktion *Arcs*, die nach dem gleichen Schema wie $fun2matrix$ (s. §6.5) und $procMatrix$ (s. §6.6) gebildet ist. *Arcs* ruft die Hilfsfunktion *sortRows* auf, die jeweils eine Zeile der Matrix bearbeitet. *sortRows* wiederum ruft die Funktion *sortRow* auf, die jeweils einen Eintrag einer Zeile bearbeitet, also die dem Eintrag entsprechende Kante erzeugt. Wie oben beschrieben, werden die so erzeugten Tellisten mit *insert* bzw. *merge* zusammengefügt, damit sofort eine sortierte Gesamtliste entsteht.

```
fun Arcs(size,G) = sortRows(fn(x,y)=>(x,G(x,y),y))(1)(size,size)

and sortRows(f)(i)(m,n) = if i > m then nil
                          else merge(r)(sortRow(fn(j)=>f(i,j))(1)(n),
                                        sortRows(f)(i+1)(m,n))

and sortRow(f)(j)(n) = if j > n then nil
                       else let val L = sortRow(f)(j+1)(n)
                                val (x,lab,y) = f(j)
                            in if lab = 0 then L else insert(r)(x,lab,y)(L) end
```

Die Vergleichsrelation r wird auf Kanten wie folgt definiert:

```
and r((_,lab1:int,_),(_,lab2,_)) = lab1 <= lab2
```

Wir konstruieren den Spannbaum von G aus der Liste L seiner Knoten und der sortierten Liste $Arcs(size, G)$ seiner Kanten *iterativ*. Die zugehörige Schleifenfunktion *loop* verändert schrittweise vier Parameter:

(1) Die sortierte Liste *Arcs* noch nicht bearbeiteter Kanten.

(2) Die aktuelle Knotenpartition, dargestellt als Funktion $part : int \to int$. *part* ordnet jedem Knoten x dasjenige Partitionselement zu, das x enthält. Jedes Partitionselement e ist durch einen eindeutigen *Repräsentanten* dargestellt, d.i. ein ausgezeichneter Knoten $x_e \in e$.

(3) Die Anzahl *size* der Elemente der Partition.

(4) Der aktuelle Spannbaum *tree*, wie G dargestellt als Adjazenzmatrix.

Die Partition *part* und der Spannbaum *tree* sind Funktionen, die von *loop* schrittweise verändert werden. Der gesamte Algorithmus wird nun durch folgende Funktion *SpanTree* implementiert. *SpanTree* initialisiert die vier Parameter von *loop* und ruft *loop* damit auf.

```
exception unconnected_graph

fun SpanTree(L,G)
       = let val size = length(L)       Anzahl der Knoten von G
         in loop Arcs(size,G)           sortierte Liste der Kanten von G
                 fn(x)=>x               Zu Anfang besteht jedes Partitionselement aus genau
                                        einem Knoten.
                 size                   Größe der Anfangspartition = Anzahl der Knoten von G
                 fn(x,y)=>0             leerer Spannbaum
         end

and loop _ _ 1 tree = tree              Partition einelementig ⇒ Spannbaum gefunden
|   loop nil _ _ _ = raise unconnected_graph
                                        Partition nicht einelementig und keine Kanten mehr da
                                        ⇒ G ist unzusammenhängend
|   loop (x,lab,y)::arcs part size tree
       = let val i = part(x)            i bezeichnet das Partitionselement, das x enthält.
             val j = part(y)            j bezeichnet das Partitionselement, das y enthält.
```

```
in if i = j then loop arcs part size tree
                                    keine Änderung von Partition und Spannbaum
             else loop arcs
                        fn(z) => if part(z) = i then j else part(z)
                                    Die beiden Elemente der Partition, die x bzw. y
                                    enthalten, werden vereinigt. Damit wird die
                                    Partition um ein Element verkleinert:
                        size-1
                                    und sichergestellt, daß loop terminiert.
                        fn(a) => if a = (x,y) then lab else tree(a)
                                    Die Markierung lab wird bei (x,y) in tree eingefügt.
end
```

Da die Funktion *SpanTree* iterativ definiert ist, beweisen wir ihre Korrektheit durch Hoare-Induktion (s. §5.4). Die gewünschte Ein-Ausgaberelation von *SpanTree* lautet wie folgt:

$$P(L, G, tree) \quad \Longleftrightarrow_{def} \quad tree \text{ ist ein minimaler Spannbaum von } G.$$

Eine passende Hoare-Invariante H hat sechs Parameter: die Eingabeparameter L und G sowie die möglichen Werte der Schleifenvariablen *arcs*, *part*, *size* und *tree*. Nach Definition von *loop* lauten die Anforderungen an H wie folgt:

(a) $H(L, G, Arcs(L, size, G), \lambda x.x, size, \lambda(x, y).0)$
(b) $size = 1 \wedge H(L, G, arcs, part, size, tree) \;\Rightarrow\; P(L, G, tree)$
(c) $size \neq 1 \wedge arcs = nil \wedge H(L, G, arcs, part, size, tree) \;\Rightarrow\; G$ ist unzusammenhängend
(d) $size \neq 1 \wedge arcs = (x, lab, y) :: arcs' \wedge part(x) = part(y) \wedge H(L, G, arcs, part, size, tree)$
$\quad\Rightarrow\; H(L, G, arcs', part, size, tree)$
(e) $size \neq 1 \wedge arcs = (x, lab, y) :: arcs' \wedge part(x) \neq part(y) \wedge H(L, G, arcs, part, size, tree)$
$\quad\Rightarrow\; H(L, G, arcs', part[z \leftarrow part(y) | part(z) = part(x)], size - 1, tree[(x, y) \leftarrow lab])$

Nach einigem Probieren stellt man fest, daß (a)-(d) gelten, wenn $H(L, G, arcs, part, size, tree)$ durch folgende vier Bedingungen definiert wird:

(1) *size* ist die Größe von *part*.

(2) Jeder Knoten von G gehört zu einem Element von *part*.

(3) Für jedes Element e von *part* sei L_e die Menge der Knoten von e. Weiterhin seien G_e und $tree_e$ die von L_e *erzeugten Teilgraphen* von G bzw. *tree*.[6] $tree_e$ ist ein minimaler Spannbaum von G_e.

(4) Jede Kante von $G \setminus arcs$ verbindet zwei Knoten desselben Elementes von *part*.

(1)-(4) gilt für die Anfangswerte von *arcs*, *part*, *size* und *tree*, wo jedes Partitionselement e genau einen Knoten enthält und jeder Knoten von G zu einem e gehört. G_e, $tree_e$ und $G \setminus arcs$ sind leer. Damit ist (a) erfüllt.

Es gelte (1)-(4) am Ausgang der Schleife, also wenn entweder *part* einelementig oder *arcs* leer ist. Im ersten Fall gehören wegen (1) und (2) alle Knoten von G zu dem einen Element von *part*. Also ist *tree* ein minimaler Spannbaum des gesamten Graphen G. Im zweiten Fall verbindet wegen (4) jede Kante von G zwei Knoten desselben Elementes von *part*. Wegen (1) besteht *part* aus mindestens zwei Elementen. Also ist G unzusammenhängend. Damit sind (b) und (c) erfüllt.

[6] Der von einer Knotenmenge K eines Graphen G erzeugte Teilgraph von G ist die Menge aller Kanten von G, deren Quell- und Zielknoten in K liegen.

Es gelte die Prämisse von (d), m.a.W. es gelte (1)-(4) für den ersten rekursiven Aufruf von *loop* in der dritten Definitionsgleichung von *loop*. Dann gehören Quell- und Zielknoten der ersten Kante K von *arcs* zum selben Element von *part*. K wird aus *arcs* entfernt. *part*, *size* und *tree* ändern sich nicht. Damit bleiben die Bedingungen (1)-(3) erhalten. K wird zu $G \setminus arcs$ hinzugefügt. Da Quell- und Zielknoten von K zum selben Element von *part* gehören, gilt auch (4) weiterhin. Damit ist (d) erfüllt.

Es gelte die Prämisse von (e), m.a.W. es gelte (1)-(4) für den zweiten rekursiven Aufruf von *loop* in der dritten Definitionsgleichung von *loop*. Dann gehören Quell- und Zielknoten der ersten Kante K von *arcs* zu verschiedenen Elementen von *part*, sagen wir e und e'. Diese werden zu einem Element *neu* zusammengefügt und *size* wird um 1 erniedrigt. Damit bleiben die Bedingungen (1) und (2) erhalten. Bedingung (3) könnte höchstens für das neue Partitionselement verletzt sein. Wegen (3) sind $tree_e$ und $tree_{e'}$ minimale Spannbäume von G_e bzw. $G_{e'}$. Man sieht sicher leicht ein, daß die Vereinigung von $tree_e$, $tree_{e'}$ und einer minimalen Kante von G, die einen Knoten von e mit einem Knoten von e' verbindet, einen minimalen Spannbaum von G_{neu} bildet. Genau so wird aber $tree_{neu}$ konstruiert. K ist tatsächlich eine minimale Kante von G, die einen Knoten von e mit einem Knoten von e' verbindet. Alle kleineren Kanten von G verbinden nämlich wegen (4) zwei Knoten desselben Elementes von *part*. Also bleibt Bedingung (3) erhalten. K wird zu $G \setminus arcs$ hinzugefügt, was die Bedingung (4) nicht verletzt, weil Quell- und Zielknoten von K zum neuen Partitionselement *neu* gehören. Damit ist (e) erfüllt.

6.5 Funktionen $\rightsquigarrow$ Matrizen

Matrizen bilden eine für zahlreiche Anwendungen äußerst geeignete Datenstruktur. Sie sind zentral in Anwendungen der linearen Algebra und werden zur Darstellung von Graphen und Transitionssystemen benutzt. Zunächst als Funktionen des Typs $int \times int \to A$ dargestellte Graphen lassen sich leicht in Matrizen, d.h. Listen des Typs *A list list* überführen und umgekehrt. Die Transformation von der einen in die andere Darstellung erfolgt mit den Funktionen $fun2matrix$ bzw. $matrix2fun$:

```
fun fun2matrix(f) = listRows(f)(1)

and listRows(f)(i)(m,n)
       = if i > m then nil
         else listRow(fn(j)=>f(i,j))(1)(n)::listRows(f)(i+1)(m,n)

and listRow(f)(j)(n) = if j > n then nil else f(j)::listRow(f)(j+1)(n)

fun matrix2fun(M)(i,j) = nth(nth(M,i-1),j-1)
```

Bei komplexen Operationen auf Graphen, die – im Gegensatz zu *SpanTree* – nicht-iterativ definiert sind, ist die Matrixdarstellung vorzuziehen. Sonst kann der Aufwand schnell exponentiell werden (s. §6.2). Übrigens darf $matrix2fun(M)$ nicht mit der *linearen Funktion* verwechselt werden, die im Rahmen Linearer Algebra als Matrix M repräsentiert wird.

6.6 Ausgabe von Matrizen

Die folgende Funktion *procMatrix* ist nach dem gleichen Schema definiert wie *Arcs*: *Arcs* traversiert eine Adjazenzmatrix G zeilenweise und konstruiert eine sortierte Liste der Kanten von G, *travFun* traversiert G ebenfalls zeilenweise und wendet eine Prozedur *proc* (= Funktion nach *unit*) auf jeden Eintrag an:

```
fun procMatrix(proc) = procRows(proc)(1)
```

```
and procRows(proc)(i)(m,n) = if i > m then ()
                             else (procRow(fn(j)=>proc(i,j))(1)(n);
                                   procRows(proc)(i+1)(m,n))

and procRow(proc)(j)(n) = if j > n then () else (proc(j); procRow(proc)(j+1)(n))
```

procMatrix kann beispielweise benutzt werden, um Matrizen in ASCII code auszugeben. Die auf die einzelnen Einträge angewendete Prozedur *proc* ist hier durch folgende Funktion $screen(f, str)$ gegeben. f ist eine Funktion des Typs $int \times int \to string$. $name : string$ ist ein Name für die auszugebende Matrix, der in ihre linke obere Ecke geschrieben werden soll.

```
fun screen(f,name)(i,j) = output(std_out,outString(f,name)(i,j))

and outString(_,name)(1,1) = name
|   outString _ (2,1) = "\n   "^genString" "(!entrysize)
|   outString _ (1,j) = outStr(makestring(j-1))
|   outString _ (2,j) = genString"-"(!entrysize)
|   outString _ (i,1) = "\n"^outStr(makestring(i-2))^" | "
|   outString(f,_)(i,j) = outStr(f(i-2,j-1))

and outStr(str) = genString" "(!entrysize-String.length(str))^str

and genString(str)(n) = fold(op ^)(genList(str)(n))""

and genList(x)(0) = nil
|   genList(x)(n) = x::genList(x)(n-1)
```

Ist $m \times n$ die Dimension der f entsprechenden Matrix, dann liefert folgende Funktion *outFun* den passenden Aufruf von *procMatrix* für die Ausgabe von f:

```
fun outFun(f,name)(m,n) = (output(std_out,"\n\n");
                           procMatrix(screen(f,name))(m+2,n+1);
                           output(std_out,"\n\n"))
```

Beispiel 1

```
fun G(1,2) = 3 | G(2,3) = 5 | G(3,1) = 1 | G _ = 0
val tree = SpanTree([1,2,3],G)

val _ = (entrysize:=3;
         outFun(makestring o G,"  G   ")(3,3);
         outFun(makestring o tree,"  tree")(3,3))
```

```
  G     1  2  3          tree   1  2  3
      ---------              ---------
  1 |   0  3  0           1 |   0  3  0
  2 |   0  0  5           2 |   0  0  0
  3 |   1  0  0           3 |   1  0  0
```

6.7 Matrizenarithmetik

Viele Algorithmen, die auf durch Matrizen dargestellten Funktionen operieren, können auch so formuliert werden, daß sie direkt auf Matrizen operieren. Dann setzt sich der Algorithmus zusammen aus einer Folge von Basisoperationen auf Matrizen, die i.w. aus der linearen Algebra bekannt sind: Transposition, Summe und Produkt. **Transposition** verändert die Struktur einer Matrix durch Spiegelung an ihrer Diagonalen. Paulson (s. §1.2) definiert diese Operation in ML wie folgt:

```
fun transpose(nil::_) = nil
|   transpose(M) = headcol(M)::transpose(tailcols(M))

and headcol(nil) = nil
|   headcol((x::_)::M) = x::headcol(M)

and tailcols(nil) = nil
|   tailcols((_::row)::M) = row::tailcols(M)
```

headcol bearbeitet die erste Spalte der Matrix M. *transpose* wird rekursiv aufgerufen mit der Restmatrix $tailcols(M)$, die aus den restlichen Spalten von M besteht.

Summenbildung ist eine zweistellige Matrixoperation, bei der auf jedes Paar von Einträgen an derselben Position der beiden Argumentmatrizen eine zweistellige Funktion f angewendet wird.

```
fun plus(f)(nil,nil) = nil
|   plus(f)(row1::M,row2::N) = rowsum(f)(row1,row2)::plus(f)(M,N)

and rowsum(f)(nil,nil) = nil
|   rowsum(f)(x::row1,y::row2) = f(x,y)::rowsum(f)(row1,row2)
```

Produktbildung hingegen basiert auf zwei skalaren Verknüpfungen f und g, einer *inneren*, z.B. ganzzahliger Multiplikation, und einer *äußeren*, z.B. ganzzahliger Addition. Haben die Matrizen M und N die Dimensionen $m \times n$ bzw. $n \times r$, dann errechnen sich die Einträge der Produktmatrix $P = M * N$ bekanntlich wie folgt:

$$P_{ij} \;=_{def}\; \sum_{k=1}^{n} M_{ik} * N_{kj}. \tag{1}$$

Auf beliebige innere und äußere Verknüpfungen f bzw. g verallgemeinert, läßt sich die Verknüpfung der Zeile M_i mit der Spalte N_j zum Wert P_{ij} als Faltung zweier Listen implementieren:

```
fun foldrowcol(f,g)([x],[y]) = f(x,y)
|   foldrowcol(f,g)(x::L1,y::L2) = g(f(x,y),foldrowcol(f,g)(L1,L2))
```

Die Produktmatrix wird dann wie folgt berechnet (vgl. Paulson, §3.9):

```
fun times(fg)(M,N) = rowsprod(fg)(M,transpose(N))

and rowsprod(fg)(nil,N) = nil
|   rowsprod(fg)(row::M,N) = rowprod(fg)(row,N)::rowsprod(fg)(M,N)

and rowprod(fg)(row,nil) = nil
|   rowprod(fg)(row,col::N) = foldrowcol(fg)(row,col)::rowprod(fg)(row,N)
```

Darauf aufbauend kann man **Matrixpotenzen** definieren:

```
fun power(fg)(M)(1) = M
|   power(fg)(M)(k) = times(fg)(power(fg)(M)(k-1),M)
```

Eine **Einheitsmatrix** E für f und g ist ein neutrales Element bzgl. der Matrixmultiplikation mit innerer Verknüpfung f und äußerer Verknüpfung g, d.h. für alle quadratischen Matrizen M, auf deren Einträgen f und g definiert sind, gilt:

$$times(f,g)(M,E) = times(f,g)(E,M) = M.$$

6.8 Transitiver Abschluß

Für quadratische Matrizen M, die Graphen repräsentieren, definieren wir nun die Operation des transitiven Abschlusses von M, mit der wir dann u.a. noch einmal kürzeste Wege in Graphen berechnen wollen (vgl. §6.2).

```
fun closure(f,g,E)(M) = iterate(f,g,E)(M)(length(M)-1)

and iterate(f,g,E)(M)(0) = let val n = length(M)
                           in fun2matrix(E)(n,n) end
|   iterate(f,g,E)(M)(k) = let val N = iterate(f,g,E)(M)(k-1)
                               val P = power(f,g)(M)(k)
                           in plus(g)(N,P) end
```

Schreibt man + für $plus(g)$ und M^k für $power(f,g)(M)(k)$, dann lautet $closure(f,g,E)(M)$ kurz:

$$E \ + \ M \ + \ M^2 \ + \ M^3 \ + \ \ldots \ + \ M^{n-1}, \tag{2}$$

wobei $n \times n$ die Dimension von M ist.

Wir betrachten drei Instanzen von *closure*:

(A) M ist vom Typ *bool list list*, repräsentiert einen unmarkierten Graphen, $f = \wedge$ und $g = \vee$.

(B) M ist vom Typ *int list list*, repräsentiert einen gewichteten Graphen, $f = +$ und $g = min$.

(C) M ist vom Typ $(int \times (int\ list))\ list\ list$, repräsentiert wieder einen gewichteten Graphen, an der Stelle (i,j) steht jetzt aber ein Paar des Typs $int \times (int\ list)$, bestehend aus dem Gewicht der Kante (i,j) und einer Knotenliste, die einen Weg von i nach j angibt. f und g sind wie folgt definiert:

```
fun f((lg1,L1),(lg2,L2)) = if L1 = nil orelse L2 = nil
                           then (!maxint,nil) else (lg1+lg2,L1@tl(L2))
fun g((lg1,L1),(lg2,L2)) = if lg1 <= lg2 then (lg1,L1) else (lg2,L2)
```

Einheitsmatrizen für diese drei Fälle lauten wie folgt:

(A) `fun E(i,j) = i = j`

(B) `fun E(i,j) = if i = j then 0 else !maxint`

(C) `fun E(i,j) = if i = j then (0,[i,i]) else (!maxint,nil)`

Durch Induktion über k kann man leicht zeigen, daß an der Stelle (i,j) der Matrix $iterate(f,g,E)(M)(k)$ folgender Eintrag steht:

- im Fall A genau dann der Wert *true*, wenn in M ein Weg von i nach j existiert, der aus k Kanten besteht,
- im Fall B die Länge eines minimalen Weges von i nach j in M, der aus k Kanten besteht,
- im Fall C die Länge und die Knoten eines minimalen Weges von i nach j in M, der aus k Kanten besteht.

Demzufolge steht an der Stelle (i, j) der Matrix $closure(f, g, E)(M)$

- im Fall A genau dann der Wert *true*, wenn in M ein Weg von i nach j existiert,
- im Fall B die Länge eines minimalen Weges von i nach j in M,
- im Fall C die Länge und die Knoten eines minimalen Weges von i nach j in M.

Der Fall A begründet die Bezeichnung "transitiver Abschluß" von M: Ein unmarkierter Graph $G : int \times int \to A$ entspricht der reflexiven binären Relation $R = \{(i, j) \in A^2 \mid G(i, j) = true\}$ und es gilt:

$$matrix2fun(closure(f, g, E)(fun2matrix(G))) \;=\; R^*,$$

wobei man die Relation

$$R^* \;=_{def}\; \{(i, j) \in A^2 \mid \exists i_1, \ldots, i_n : i_1 = i \wedge i_n = j \wedge \forall 1 \leq k < n : (i_k, i_{k+1}) \in R\}$$

üblicherweise als transitiven Abschluß von R bezeichnet.

Die Fragen nach der Existenz eines Weges, der minimalen Weglänge und einem minimalen Weg lassen sich also alle drei durch Aufrufe von *closure* beantworten:

```
fun exists_path(M:bool list list) = let fun f(x,y) = x andalso y
                                        fun g(x,y) = x orelse y
                                        fun E(i,j) = i = j
                                    in closure(f,g,E)(M) end

fun min_length(M:int list list) = let fun E(i,j) = if i = j then 0 else !maxint
                                  in closure(op +,min,E)(M) end

fun min_path(M:(int * (int list)) list list)
        = let fun f((lg1,L1),(lg2,L2)) = if L1 = nil orelse L2 = nil
                                         then (!maxint,nil) else (lg1+lg2,L1@tl(L2))
              fun g((lg1,L1),(lg2,L2)) = if lg1 <= lg2 then (lg1,L1) else (lg2,L2)
              fun E(i,j) = if i = j then (0,[i,i]) else (!maxint,nil)
          in closure(f,g,E)(M) end
```

Zur Anwendung von *min_path* formen wir einen gegebenen Graphen $G : int \times int \to int$ in einen Graphen des Typs : $int \times int \to int \times (int\ list)$ um:

```
fun Path(G)(i,j) = let val lg = G(i,j)
                   in (lg, if lg = !maxint then nil else [i,j]) end
```

Neben dem Gewicht lg einer Kante (i, j) steht also zunächst die Liste $[i, j]$, falls (i, j) tatsächlich eine Kante von G ist, sonst wird dort *nil* eingetragen.

Beispiel 2 (für Fall A)

```
fun H(1,2) = true | H(1,3) = true | H(3,2) = true | H(1,5) = true
|   H(5,3) = true | H(5,4) = true | H(3,4) = true | H(4,2) = true
|   H(i,j) = i = j

val M = fun2matrix(H)(5,5)
val HP = matrix2fun(exists_path(M))

val _ = (entrysize:=6; outFun(makestring o H,"     H   ")(5,5);
                       outFun(makestring o HP,"   reach ")(5,5))

     H        1     2     3     4     5
         -----------------------------
     1 |   true  true  true false  true
     2 |  false  true false false false
     3 |  false  true  true  true false
     4 |  false  true false  true false
     5 |  false false  true  true  true

   reach      1     2     3     4     5
         -----------------------------
     1 |   true  true  true  true  true
     2 |  false  true false false false
     3 |  false  true  true  true false
     4 |  false  true false  true false
     5 |  false  true  true  true  true
```

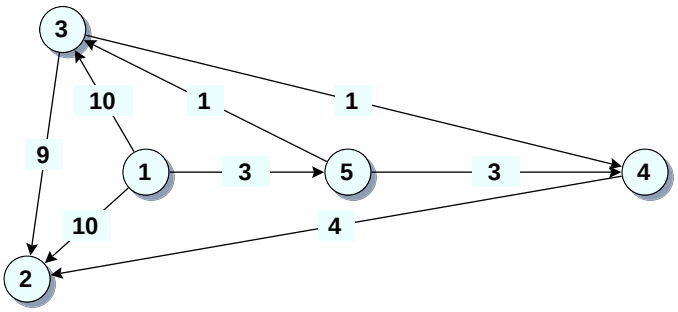


Figure 5. *Die Graphen G und H (= unmarkierte Version von G)*

Beispiel 3 (für die Fälle B und C)

```
fun G(1,2) = 10 | G(1,3) = 10 | G(3,2) = 9 | G(1,5) = 3 | G(5,3) = 1
|   G(5,4) = 3  | G(3,4) = 1  | G(4,2) = 4
|   G(i,j) = if i = j then 0 else !maxint

val M = fun2matrix(G)(5,5)
val GL = matrix2fun(min_length(M))

val N = fun2matrix(Path(G))(5,5)
val GP = matrix2fun(min_path(N))
```

```
fun outL(G)(i,j) = let val lg = G(i,j)
                   in if lg = !maxint then "X" else makestring(lg) end

fun outP(G)(i,j) = let val (lg,L) = G(i,j)
                   in if L = nil then "X"
                      else fold(op ^)(map(makestring)(L:int list))"" end

val _ = (entrysize:=3; outFun(outL(G),"    G ")(5,5);
                       outFun(outL(GL),"length")(5,5);
         entrysize:=6; outFun(outP(GP),"    path ")(5,5))
```

```
    G   1  2  3  4  5          length  1  2  3  4  5
      ---------------                ---------------
  1 |   0 10 10  X  3            1 |   0  9  4  5  3
  2 |   X  0  X  X  X            2 |   X  0  X  X  X
  3 |   X  9  0  1  X            3 |   X  5  0  1  X
  4 |   X  4  X  0  X            4 |   X  4  X  0  X
  5 |   X  X  1  3  0            5 |   X  6  1  2  0

    path      1     2     3     4     5
         ------------------------------
     1 |     11 15342   153  1534    15
     2 |      X    22     X     X     X
     3 |      X   342    33    34     X
     4 |      X    42     X    44     X
     5 |      X  5342    53   534    55
```

Die in den Fällen A, B und C verwendeten "äußeren" Funktionen g sind allesamt *idempotent*, d.h. für alle $x : bool$, $x : int$ bzw. $x : int \times (int\ list)$ gilt $g(x, x) = x$. In solchen Fällen kann $closure(f, g, E)$ optimiert werden zu:

```
fun closure1(f,g,E)(M) = let val n = length(M)
                             val N = plus(g)(fun2matrix(E)(n,n),M)
                         in power(f,g)(N)(n-1) end
```

Schreibt man wieder + für $plus(g)$ und M^k für $power(f, g)(M)(k)$, dann lautet $closure1(f, g, E)(M)$ kurz:

$$(E\ +\ M)^{n-1}, \tag{3}$$

wobei $n \times n$ die Dimension von M ist. Die Behauptung ist also, daß (2) mit (3) übereinstimmt, falls g idempotent ist. Das folgt sofort durch Induktion über n. Die Berechnung von $(E + M)^{n-1}$ läßt sich auch noch optimieren, indem man nur so lange neue Potenzen bildet, wie $(E + M)^k \neq (E + M)^{k+1}$ gilt.

Diese Anwendungen zeigen deutlich, wie mit dem Konzept der **Polymorphie** ein hohes Maß an **Wiederverwendbarkeit** der erstellten Programme erreicht werden kann.

7 Konstruktorbasierte Datentypen

Ein solcher Typ wird wie folgt definiert:

```
datatype dt = con1 of typ1 | ... | conn of typn
```

Beschreiben wird dadurch die **Summe** oder **disjunkte Vereinigung** $typ_1 \uplus \cdots \uplus typ_n$ der Typen $typ_1, \ldots, typ_n$. Die Symbole $con_1, \ldots, con_n$ heißen **Konstruktoren**. Sie dienen der Unterscheidung von Objekten des Typs dt. Ein Element a des Typs typ_i, $1 \leq i \leq n$, wird erst durch Voranstellung des Konstruktors con_i zu einem Element des Typs dt. con_i wird als Funktion von typ_i nach dt aufgefaßt. Ist $a \in typ_i$, dann ist der Ausdruck $con_i(a)$ ein Element von dt. typ_i kann selbst konstruktorbasiert sein. Allgemein sind die Elemente eines konstruktorbasierten Typs immer aus Standardwerten, Tupeln, λ-Abstraktionen (s. §2.10), Namen definerter Funktionen und/oder Konstruktoren aufgebaute Ausdrücke.

Im Gegensatz zur Definition eines nicht-konstruktorbasierten Typs wie z.B.

```
type t = int * (string→bool)
```

mit dem Schlüsselwort *type* können konstruktorbasierte Typen **rekursiv** sein, d.h. dt darf selbst unter den Typen $typ_1, \ldots, typ_n$ vorkommen. Einen rekursiven Typ bilden z.B. die in Kap. 4 behandelten Listen. Er hat einen nullstelligen und einen zweistelligen Konstruktor:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Tatsächlich sind Listen ja nicht anderes als aus *nil*, :: und Elementen einer Instanz von $'a$ aufgebauten Ausdrücken.

Konstruktorbasierten Typen kommen in der "realen Welt" kaum vor. Wie alle Ausdrücke dienen ihre Elemente eher der *Abstraktion* konkreter Daten, aus denen gewisse Bedeutungsinhalte herausgefiltert und als *Strukturinformation* sichtbar gemacht werden. Algorithmen werden dann nicht mehr direkt auf den konkreten Daten formuliert, sondern auf den (abstrakten) Konstruktorausdrücken. Unterschiede zwischen konkreten Daten werden zu Unterschieden zwischen den **Mustern**, auf die die abstrakten Daten jeweils passen. Rechenvorschriften zur Manipulation konkreter Daten werden zu Strategien, bestimmte Muster zu transformieren. Demzufolge werden konstruktorbasierte Typen insbesondere beim **symbolischen Rechnen** eingesetzt.

7.1 Symbolisches Rechnen

Beispiel 7.1.1 Symbolische Differentiation

```
datatype expr = con of int
              | var of string
              | sum of expr list
              | prod of expr list
```

con con : int → expr
con prod : expr list → expr
con sum : expr list → expr
con var : string → expr

Die vom ML-System vorgenommene Typisierung macht deutlich, daß Konstruktoren tats'achlich als Funktionen in durch sie definierten Typ, hier *expr*, betrachtet werden.

Der "konkrete" arithmetische Ausdruck $5 * (x + 2 + 3)$ entspricht folgendem Element des Typs *expr*:

```
prod[con(5),sum[var"x",con(2),con(3)]].
```

Die folgende Funktion definiert symbolische Differentiation abhängig vom jeweiligen Muster der ihrer möglichen Argumente:

```
fun d(x)(con _) = con(0)
|   d(var(x))(var(y)) = if x = y then con(1) else con(0)
|   d(x)(sum(L)) = sum(map(d(x))(L))
|   d(x)(prod(nil)) = con(0)
|   d(x)(prod[e]) = d(x)(e)
|   d(x)(prod(e::L)) = let val p = prod(L)
                            val e1 = prod[d(x)(e),p]
                            val e2 = prod[e,d(x)(p)]
                        in sum[e1,e2] end
```

val d = fn : expr → expr → expr

Beispiel 7.1.2 Kommando-Interpreter

```
datatype Int = Int of int
             | var of string
             | sum of Int list
             | prod of Int list
             | minus of Int * Int

datatype Bool = Bool of bool
              | greater of Int * Int
              | Not of Bool

datatype command = skip
                 | assign of string * Int
                 | seq of command list
                 | cond of Bool * command * command
                 | loop of Bool * command
```

con Int : int → Int
*con minus : Int * Int → Int*
con prod : Int list → Int
con sum : Int list → Int
con var : string → Int
con Bool : bool → Bool
con Not : Bool → Bool
*con greater : Int * Int → Bool*
*con assign : string * Int → command*
*con cond : Bool * command * command → command*
*con loop : Bool * command → command*
con seq : command list → command
con skip : command

Das konkrete Programm *while x > 0 do fact:=fact*x; x:=x-1 od* entspricht folgendem Element des Typs *command*:

```
val prog = loop(greater(var"x",Int(0)),
                seq[assign("fact",prod[var"fact",var"x"]),
                    assign("x",minus(var"x",Int(1)))])
```

Diese Darstellung des Programms bezeichnet man als **abstrakte Syntax**. Programme werden von ihren **Compilern** (Übersetzern) zunächst in abstrakte Syntax überführt, die Strukturinformation enthält, die ein Compiler

benötigt, um das zunächst nur als Zeichenfolge gegebene Quellprogramm in die jeweilige Zielsprache zu übersetzen. Gleiches gilt für **Interpreter** (Auswerter, Evaluatoren), die Programme nicht übersetzen, sondern, abhangig von Eingaben, aktuellen Parametern o.ä., ausführen.

Ein Interpreter für unser Beispiel besteht aus drei Funktionen:

$$\begin{array}{rcl} evalInt & : & Int \to (string \to int) \to int \\ evalBool & : & Bool \to (string \to int) \to bool \\ evalCom & : & command \to (string \to int) \to string \to int \end{array}$$

In Abhängigkeit vom Speicher-**Zustand** $s : string \to int$, d.h. der jeweiligen Zuordnung von Werten zu Variablen (hier als Strings repräsentiert), reduzieren $evalInt$ und $evalBool$ Int- bzw. $Bool$-Ausdrücke zu ganzzahligen bzw. Booleschen Werten, während $evalCom$ $command$-Terme auf nach ihrer Ausführung erreichte Folgezustände abbildet.

```
fun evalInt(Int(n))(s) = n
|   evalInt(var(x))(s) = s(x)
|   evalInt(sum(L))(s) = let fun f(e) = evalInt(e)(s)
                          in fold(op + )(map(f)(L))(0) end
|   evalInt(prod(L))(s) = let fun f(e) = evalInt(e)(s)
                           in fold(op * )(map(f)(L))(1) end
|   evalInt(minus(e1,e2))(s) = evalInt(e1)(s) - evalInt(e2)(s)

fun evalBool(Bool(b))(s) = b
|   evalBool(greater(e1,e2))(s) = evalInt(e1)(s) > evalInt(e2)(s)
|   evalBool(Not(e))(s) = not(evalBool(e)(s))

fun evalCom(skip)(s) = s
|   evalCom(assign(x,e))(s)
      = (fn(y) => if x = y then evalInt(e)(s) else s(y))
|   evalCom(seq(L))(s) = fold(fn(c,s)=>evalCom(c)(s))(rev(L))(s)
|   evalCom(cond(e,c1,c2))(s)
      = if evalBool(e)(s) then evalCom(c1)(s) else evalCom(c2)(s)
|   evalCom(loop(e,c))(s)
      = evalCom(cond(e,seq[c,loop(e,c)],skip))(s)
```

Wir definieren einen Anfangszustand $init : \{x, fact\} \to int$:

```
fun init"x"= 4
|    init"fact"= 1
```

und wenden den Interpreter darauf an: `val ergebnis = evalCom(prog)(init)"fact"`

val ergebnis = 24 : int

Wir wiederholen noch einmal das Schema der Definition eines konstruktorbasierten Typs:

$$datatype\ dt\ =\ c_1\ of\ type_1 \mid \ldots \mid c_m\ of\ type_m \mid c_{m+1} \mid \ldots \mid c_n.$$

Für alle $1 \leq i \leq m$ ist $type_i$ ein Typ(ausdruck), der dt enthalten darf. In diesem Fall ist dt ein **rekursiver Typ**. Der **Konstruktor** c_i ist eine Funktion vom Typ $type_i \to dt$, (für alle $1 \leq i \leq m$) bzw. ein Wert vom Typ dt (für alle $m > i \leq n$).

Die Menge der Werte vom Typ dt und der Werte anderer in der Definition von dt möglicherweise enthaltener Typen ist **induktiv definiert**:

- Strings, Boolesche Werte, ganze und reelle Zahlen sind vom Typ *string*, *bool*, *int* bzw. *real*.
- Für $k \geq 1$ und alle $1 \leq i \leq k$ sei a_i ein Wert vom Typ $type'_i$. Dann hat $(a_1, \ldots, a_k)$ den *Produkttyp* $type'_1 \times \ldots \times type'_k$.
- Sind A und B die Mengen der Werte vom Typ *type* bzw. *type'*, dann hat jede (partielle) Funktion $f : A \to B$ den Typ $type \to type'$.
- Für jeden Typ *type* ist *nil* vom Typ *type list*.
- Ist a vom Typ *type* und L vom Typ *type list*, dann ist auch $a :: L$ vom Typ *type list*.
- Sei $1 \leq i \leq n$. Ist a vom Typ $type_i$, dann ist $c_i(a)$ vom Typ dt.

dt ist ein **polymorpher Datentyp**, wenn in $type_1, \ldots, type_m$ Typvariablen vorkommen. Sie müssen vor dt aufgelistet werden:

$$datatype\ ('a_1, \ldots, 'a_k)\ dt\ =\ c_1\ of\ type_1 \mid \ldots \mid c_m\ of\ type_m \mid c_{m+1} \mid \ldots \mid c_n.$$

Bei $k = 1$ entfallen die runden Klammern. Analog zu anderen polymorphen Typen erhält man durch Ersetzung der Typvariablen $'a_1, \ldots, 'a_k$ durch Typen $type'_1, \ldots, type'_k$ eine **Instanz** von dt. Sie wird mit $(type'_1, \ldots, type'_k)\ dt$ bezeichnet.

7.2 Konstruktoren versus Ausnahmen

In komplexen Algorithmen kann das Aufrufen (*raise*) und Abfangen (*handle*) von Ausnahmen (*exceptions*) benutzt werden, um Teilalgorithmen abzubrechen (in imperativen Programmiersprachen oft durch ein *exit*-Kommando realisiert) und anschließend an deren Aufrufstellen fortzufahren. Wir werden diese Verwendung von Ausnahmen an dem in Kapitel 12 vorgestellten Algorithmus demonstrieren.

Mathematisch betrachtet sind Ausnahmen nichts anderes als zusätzliche Werte, die eingeführt werden, um partielle Funktionen zu **totalisieren**. Zusätzliche Werte erhält man aber auch, wenn man Datentypen um nullstellige Konstruktoren erweitert. Da wir aber (zumindest in ML) keine Teilmengenbeziehung zwischen Datentypen (die übrigens der Eltern-Erben-Beziehung in objektorientierten Programmen entspricht), entsteht mit jedem neuen Konstruktor ein neuer Datentyp, der dementsprechend auch neu definiert werden muß:

```
datatype dt = con1 of typ1 | ... | conn of typn
datatype newdt = C of dt | except1 of typ'1 | ... | except_k of typ'_k
```

Das ist etwas umständlich, hat aber (fast) dieselbe Bedeutung wie die Deklaration von Ausnahmen *except*$_1$ *of* *typ*$'_1$,...,*except*$_k$ *of* *typ*$'_k$.[7] Für den Fall einer einzelnen Ausnahme E, die von einer Funktion f aufgerufen wird und nach deren Aufruf *sofort* abgefangen wird, gilt folgende Äquivalenz zwischen entsprechenden Programmstücken:

E *als Ausnahme*	**E** *als Konstruktor*
	`datatype dt = C of typ1`
`exception E of typ2`	`  | E of typ2`
`fun f(p1) = e1:typ1`	`fun f(p1) = C(e1)`
`|  f(p2) = raise E(e2)`	`|  f(p2) = E(e2)`
`f(e) handle E(p) => e'`	`case f(e) of C(x) => x`
	`  |  E(p) => e'`

[7] Auch Ausnahmen können Parameter haben! Mehr dazu ebenfalls in Kap. 12.

Beim Programmieren sollte man eher wie auf der linken Seite verfahren. Zum Nachweis der Korrektheit eines Programms, das Ausnahmen verwendet, muß man aber deren Bedeutung im Auge haben, die gerade durch die Formulierung auf der rechten Seite gegeben ist.

Variablen, die das Muster p enthält, werden bei der Ausführung von *handle* $E(p)$ Teilausdrücke von e_2 zugewiesen, womit praktisch Daten aus dem "Rumpf" von f (wo e_2 ausgewertet wird), nach außen transportiert werden. Dieses "nach-oben-Reichen" von Werten wird in imperativen Programmen durch allerlei unübersichtliche **Rücksprünge** realisiert, die zwar in der letztendlichen Implementierung eines Algorithmus effizient ausführbar sein mögen, dessen halbwegs formalen Korrektheitsbeweis aber unmöglich machen. Rücksprungbefehle sollten deshalb auf höheren Ebenen eines Softwareentwurfs stets vermieden werden!

Das o.a. Schema einer Ausnahmebehandlung zeigt noch nicht die ganze Stärke diese Konzeptes. Werte können hiermit nämlich nicht nur aus *einer* Funktion heraus zu deren Aufrufstellen, sondern auch aus einer mehrstufigen Schachtelung von Funktionsaufrufen heraus noch oben transportiert werden. Das entspricht dann wirklich beliebigen Rücksprüngen, ist aber der Korrektheitsanalyse leichter zugänglich. Bei allen Funktionen, durch deren Aufrufe eine Ausnahme nur nach oben durchgereicht wird, braucht man sich nur die jeweiligen Definitions- und Wertebereiche um entsprechende Ausnahme-Konstruktoren erweitert denken. Was vorher eine einzige Ausnahme war, kann dabei zu mehreren Konstruktoren verschiedener Datentypen werden, je nachdem durch welche Definitions- und Wertebereiche die Ausnahme hindurchgereicht wird, bevor sie ein *handle* abfängt.

7.3 Abstrakte Datentypen

verbinden die Definition eines konstruktorbasierten Typs mit Wert- und Funktionsdefinitionen, die auf Konstruktoren des Typs zugreifen. Das Schema der Definition eines abstrakten Datentyps ist eine Erweiterung desjenigen für konstruktorbasierte Typen:

$$\begin{array}{ll} \mathit{abstype} & ('a_1, \ldots, 'a_k)\ \mathit{adt} \ = \ c_1\ \mathit{of}\ \mathit{type}_1 \mid \ldots \mid c_m\ \mathit{of}\ \mathit{type}_m \mid c_{m+1} \mid \ldots \mid c_n \\ \mathit{with} & \text{Wert- und Funktionsdefinitionen } \mathit{end} \end{array}$$

Die Konstruktoren von *adt* können jetzt nur noch in den von *with* und *end* "eingekapselten" Definitionen verwendet werden. In außerhalb von *with* ... *end* auftretenden Ausdrücken dürfen sie nicht vorkommen. Das hat den Zweck, in späteren Entwurfsschritten die Definition von *adt* verändern zu können, ohne das gesamte Programm nach Elementen von *adt* durchsuchen zu müssen, um sie an die neue Definition anzupassen. Alle notwendigen Änderungen finden zwischen *with* und *end* statt, da ausßerhalb der Kapsel keine Zugriffe auf die Konstruktoren $c_1, \ldots, c_n$ erfolgen.

Man spricht vom *abstrakten* Datentyp, weil es sich bei seinen möglichen Änderungen im Laufe einer Programmentwicklung i.a. um **Verfeinerungen** (*refinements*) handelt, von einer abstrakten Version des Datentyps hin zu einer konkreten Implementierung, die auf die jeweils gegebene Sprach- und Rechnerumgebung zugeschnitten sind. Abstrakte Datentypen eignen sich daher oft für **Prototyp**-implementierungen auf einer höheren Entwicklungsphase. Auf jeden Fall sind sie ein nützliches Konstrukt zur **Modularisierung** von Programmen.[8] Als Modul betrachtet hat ein abstrakter Datentyp eine klar definierte **Exportschnittstelle**. Diese ist durch die zwischen *with* und *end* definierten Werte und Funktionen gegeben, während die Konstruktoren des Datentyps nicht exportiert werden, also von außen unsichtbar bleiben.

Beispiel 7.3.1 Container (s. Kap. 15) In §15.2 wird ein abstrakter Datentyp *cont* (*Container*) mit zwei Konstruktoren *new* und *add* definiert. Darin eingekapselt ist eine Reihe von Funktionen, die direkt auf diese abstrakte Darstellung von Containern zugreifen. *cont* ist in ein umfangreiches Programm eingebettet, das die eingekapselten Funktionen benutzt, aber eben nicht die Konstruktoren von *cont*. Will man nun in einem späteren Entwurfsschritt die Containerdarstellung ändern und z.B. durch durch Matrizen ersetzen, dann braucht man

[8] Weitere Modularisierungskonstrukte von ML werden in Kap. 11 behandelt.

nur die Definitionen der eingekapselten Funktionen modifizieren und kann alles andere unverändert lassen. Da nur der Modul *cont* betroffen ist, wird auch seine Verfeinerung als *abstype* formuliert.[9]

```
abstype refined_cont = c of obj list list
with    fun     size(M) = (length(M),length(hd(M)))
        fun     len(c(M)) = #1(size(M))
        fun     hei(c(M)) = #2(size(M))

        fun     New(l,h) = c(fun2matrix(fn _=>empty)(h,l)
        fun     Add(a,i,j,c(M)) = let fun f(x,y) = if inside(a,i,j)(y,x) then a
                                                   else matrix2fun(M)(x,y)
                                  in c(fun2matrix(f)(size(M))) end

        exception notEmpty
        fun     isNew(c(M)) = let fun p(x,y) = if matrix2fun(M)(x,y) = empty then ()
                                               else raise notEmpty
                              in (procMatrix(p)(size(M)); true)
                                 handle notEmpty => false end

        fun     Objs(c(M)) = objsRows(matrix2fun(M))(1)(size(M))
        and     objsRows(f)(i)(m,n) = if i > m then nil
                                      else union(objsRow(fn(j)=>f(i,j))(1)(n),
                                                 objsRows(f)(i+1)(m,n))
        and     objsRow(f)(j)(n) = if j > n then nil
                                   else insert(f(i,j),objsRow(f)(j+1)(n))
        and     union(nil,L) = L
        |       union(x::L,L') = insert(x,union(L,L'))
        and     insert(x,L) = if member(L)(x) then L else x::L

        fun     makeEmpty(c(M)) = c(fun2matrix(fn _=>empty)(size(M)))

        exception found of obj
        fun     pos(c(M),a) = let fun p(x,y) = if matrix2fun(M)(x,y) <> a then ()
                                               else raise found(y,x)
                              in (procMatrix(p)(size(M)); (0,0))
                                 handle found(i,j) => (i,j) end

        exception undef
        fun     get(c(M),i,j) = let val a = matrix2fun(M)(j,i)
                                in if a = empty then raise undef else a end

        and     inside ...  (s. 12.2)

end     (* refined_cont *)
```

Die einem 2-dimensionalen Container C mit Höhe *hei* und Länge *len* entsprechende Matrix M hat *hei* Zeilen, *len* Spalten und an der Stelle (i, j) den Eintrag a, falls die Position (i, j) von C mit dem Objekt a belegt ist. *empty*

[9] Wir verwenden in den §§6.5 u. 6.6 definierte Matrixoperationen.

bezeichnet das "leere Objekt", das als ein definierter Wert vom Typ *obj* vorausgesetzt wird. Der Konstruktor c ist hier nur aus syntaktischen Gründen erforderlich. Außer der Umbenennung des Typnamens *obj list list* in $refined\_cont$ hat c keine Bedeutung. Die Semantik der Funktionen von *cont* ist in Kap. 12 beschrieben.

Die Definitionen von $isNew$ und *pos* enthalten mit Ausnahmen realisierte **Rücksprünge** (s. §7.2). Die Ausnahmen *notEmpty* bzw. $found$ werden ggf. beim Passieren eines Matrizeneintrags aufgerufen, was bewirkt, daß der Matrizendurchlauf abgebrochen wird. Erst am Ende des Aufrufs von $isNew$ bzw. *pos* wird *notEmpty* bzw. $found$ abgefangen. In diesem Fall liefert $isNew$ den Wert $false$ und *pos* die von der Abbruchstelle nach außen transportierte Position (i, j). Das beiden Fällen gemeinsame Schema der Ausnahmebehandlung entspricht einer Wertdefinition:

$$val\ v \quad = \quad (t; k)\ handle\ ex \Rightarrow u,$$

wobei t und u beliebige Ausdrücke sind, k eine Konstante ist und ex ein Muster, dessen äußerstes "Funktionssymbol" eine Ausnahme ist.[10] Der Wert von v ist der Wert von k, falls ex bei der Auswertung von t nicht aufgerufen wird. Andernfalls ist der Wert von v der Wert von u. Das Schema ähnelt einem Konditional, das vom Ergebnis eines Mustervergleichs gesteuert wird:

$$val\ v \quad = \quad case\ t\ of\ ex \Rightarrow u \mid \_ \Rightarrow k.$$

Hier muß aber immer zunächst t komplett ausgewertet werden, bevor die Entscheidung zwischen k und u getroffen werden kann. Das hieße im Fall von $isNew$ und *pos*, daß immer erst die gesamte Matrix traversiert wird, obwohl die Werte von $isNew$ bzw. *pos* oft schon früher festliegen.

Neben der größeren Übersichtlichkeit ist die getrennte Verfeinerung einzelner Moduln auch deshalb von Vorteil, weil zum Nachweis der Korrektheit eines Gesamtentwurfs nur die einzelnen Modulverfeinerungen untersucht werden müssen. Im obigen Beispiel wäre gerade zu zeigen, daß die Funktionsdefinitionen von $refined\_cont$ alle durch die Funktionsdefinitionen von *cont* gegebenen Gleichungen erfüllen.

8 Binäre Bäume

Die Elemente eines konstruktorbasierten Typs sind aus dessen Konstruktoren aufgebaute Ausdrücke. Sie haben also eine Baumstruktur. Daher ist es naheliegend, umgekehrt Baumstrukturen als solche Typen zu implementieren. Formal ist ein Baum ein gerichteter Graph (s. §6.1), dessen Knoten von einem einzigen *Wurzel*knoten aus erreichbar sind und jeweils höchstens eine einlaufende Kante haben.

Binäre Bäume haben darüberhinaus die Eigenschaft, daß jeder Knoten höchstens zwei (direkte) Nachfolgerknoten hat. Kantenmarkierungen gibt es hier i.a. nicht. Stattdessen sind die Knoten markiert. Die Identität eines Knotens ergibt sich aus seiner Position im Baum, so daß wir keine Typvariable für die Knotenmenge, wohl aber für die Menge der Knotenmarkierungen (auch Knoteneinträge genannt) benötigen:

```
datatype 'a bintree = mt | T2 of ('a bintree) * 'a * ('a bintree)
```
con mt : 'a bintree
*con T2 : 'a bintree * 'a * 'a bintree → 'a bintree*

mt (*empty*) bezeichnet den "leeren Baum". Jeder nichtleere Baum entspricht einem *bintree*-Element der Form $T_2(l, x, r)$. Hierbei bezeichnet l linkem Unterbaum, x die Wurzel r den rechten Unterbaum. Ein **Blatt** ist ein Knoten ohne Nachfolger:

```
fun leaf(x) = T2(mt,x,mt)
```
val leaf = fn : 'a → 'a bintree

[10] k und u müssen vom selben Typ sein!

Entsprechend der **Faltung** von Listen mithilfe 2-stelliger Funktionen (s. 4.1) definieren wir eine Funktion *treefold*, die binäre Bäume durch *bottom-up* Anwendung einer 3-stelligen Funktion f auswertet:

```
fun treefold(f)(mt)(a) = a
|   treefold(f)(T2(l,x,r))(a) = f(treefold(f)(l)(a),x,treefold(f)(r)(a))
```
*val treefold = fn : ('a * 'b * 'a → 'a) → 'b bintree → 'a → 'a*

Die **Höhe** eines binären Baumes ist die Länge seines längsten Weges von der Wurzel zu einem Blatt:

```
fun height(t) = treefold(fn(m,_,n)=>max(m,n)+1)(t)(0)
```
val height = fn : 'a bintree → int

Ein binärer Baum heißt **Suchbaum**, wenn seine Knoteneinträge geordnet sind und zwar so, daß für jeden Teilbaum $T_2(l, x, r)$ von B alle in l vorkommenden Knoteneinträge kleiner als x und alle in r vorkommenden Knoteneinträge größer als x sind. Die folgende transformiert Listen in Suchbäume:[11]

```
fun  list2bintree(L) = fold(insert)(L)(mt)
and  insert(z,mt) = leaf(z)
|    insert(z:int,T2(l,x,r)) = if z = x then T2(l,x,r)
                               else if z < x then T2(insert(z,l),x,r)
                                             else T2(l,x,insert(z,r))
```

val list2bintree = fn : int list → int bintree
*val insert = fn : int * int bintree → int bintree*

Entsprechend der rechtsassoziativen Listenfaltung *fold* (s. §4.1) fügt *list2bintree* die Elemente einer Liste $[z_1, \ldots, z_n]$ in umgekehrter Reihenfolge in einen Baum ein (s. Fig. 6).

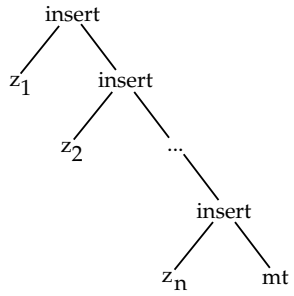


Figure 6. *Ein Aufruf von list2bintree*

Ein Baum ist **balanciert**, wenn alle seine Wege von der Wurzel zu einem Blatt dieselbe Länge haben und alle seine Knoten keine oder genau zwei auslaufende Kanten haben. Binäre Bäume können - unter Beibehaltung der Suchbaumeigenschaft - nur bis auf eine Weglängendifferenz von ± 1 ausbalanciert werden, während *2-3-Bäume*, deren Knoten bis zu drei auslaufende Kanten haben, vollständig balancierbar sind.

```
exception notBalanced
fun  balanced(t) = (check_height(t); true) handle notBalanced => false
and  check_height(mt) = 0
|    check_height(T2(l,x,r)) = let val (m,n) = (check_height(l),check_height(r))
                               in if m = n then m+1 else raise Unbalanced end
```

[11] Wir beschränken uns hier auf Listen vom Typ *int list* und auf die Ordnung $<$ auf *int*, obwohl *list2bintree* natürlich – wie die Sortieralgorithmen von §4.4 – auf beliebige Eintragstypen und Ordnungen darauf übertragen werden kann.

8.1 Traversierungen binärer Bäume

Knoten eines binären Baumes können in mindestens vier verschiedenen Reihenfolgen besucht werden. Die folgenden Funktionen realisieren die vier Traversierungsarten und liefern die Liste aller Knoteneinträge in der jeweiligen Besuchsreihenfolge.

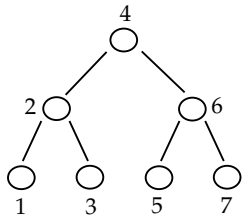


Figure 7. *Infixordnung*

Ist der traversierte Baum ein Suchbaum, dann entsteht beim **Infixdurchlauf** eine sortierte Liste der Kontenein träge.

```
fun Infix(t) = treefold(fn(L1,x,L2)=>L1@x::L2)(t)(nil)
```
val Infix = fn : 'a bintree → 'a list

In Kombination mit *list2bintree* (s.o.) erhält man also einen weiteren Sortieralgorithmus für Listen (vgl. §4.4):

```
fun  treesort(L) = Infix(list2bintree(L))
```

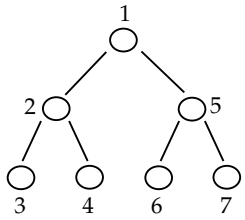


Figure 8. *Präfixordnung*

Der **Präfixdurchlauf** entspricht der Tiefensuche in beliebigen gerichteten Graphen (s. §6.3).

```
fun Prefix(t) = treefold(fn(L1,x,L2)=>x::L1@L2)(t)(nil)
```
val Prefix = fn : 'a bintree → 'a list

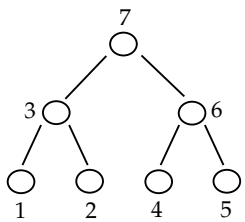
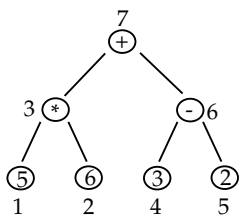


Figure 9. *Postfixordnung*

In der **Postfixreihenfolge** werden die Knoten z.B. dann besucht, wenn der Baum einen arithmetischen Ausdruck darstellt, der unter Verwendung eines Kellers ausgewertet wird (vgl. §5.2).

```
fun Postfix(t) = treefold(fn(L1,x,L2)=>L1@L2@[x])(t)(nil)
```

val Postfix = fn : 'a bintree → 'a list



Kellerzustandsfolge

5
6,5
*,6,5
30
3,30
2,3,30
-,2,3,30
1,30
+,1,30
31

Figure 10. *Postfixauswertung*

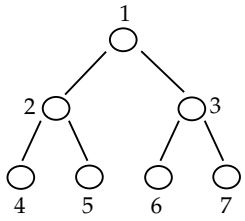


Figure 11. *Heapordnung*

Ein Durchlauf in **Heapordnung** entspricht der Breitensuche in beliebigen gerichteten Graphen (s. §6.3).

```
fun  Heaplist(t) = breadthfirst(nil)[t]
and  breadthfirst(L)(nil) = L
|    breadthfirst(L)(tL) = breadthfirst(L@mapconc(root)(tL))(mapconc(subtrees)(tL))
and  root(mt) = nil
|    root(T2(_,x,_)) = [x]
and  subtrees(mt) = nil
|    subtrees(T2(l,_,r)) = [l,r]
```

mapconc wurde in §4.1 definiert.

8.2 Heaps

Die Speicherung eines binären Baums in einem eindimensionalen Feld (= Funktion auf *int*; s. §6.1) folgt oft der Heapordnung. Bei der Rücktransformation vom Feld zum Baum werden vom n-ten Feldelement eine Kante zum $2n$-ten und eine Kante zum $(2n+1)$-ten Feldelement gezogen. Da ein Feld F i.a. einer *partiellen* Funktion entspricht, setzen wir voraus, daß die Definition von F eine Ausnahme *undef* verwendet, die an allen Stellen, an denen F nicht definiert ist, aufgerufen wird.

```
exception undef and Mt
fun  feld2bintree(F) = makeTree(F,1)
and  makeTree(F,n) = let val x = F(n) handle undef => raise Mt
```

```
                    val l = makeTree(F,2*n) handle Mt => mt
                    val r = makeTree(F,2*n+1) handle Mt => mt
                in T2(l,x,r) end
```

Stellen, an denen F nicht definiert ist, werden hier also in Exemplare des leeren Baums mt übersetzt. Ist das Feld als Liste gegeben, dann liefert $feld2bintree$ die Umkehrfunktion von $Heaplist$:

```
fun  list2heap(L) = let fun F(n) = nth(L,n-1) handle Nth => raise undef
                    in feld2bintree(F) end
```

Beispiel

```
val L = [2,4,67,1,5,90,78,55,33,127,1000,0,-23,-3,567,89,34,6,111,-5]
val t1 = list2bintree(L)
```

```
val L1 = Infix(t1)
```
val L1 = [-23,-5,-3,0,1,2,4,5,6,33,34,55,67,78,89,90,111,127,567,1000] : int list

```
val L2 = Heaplist(t1)
```
val L2 = [-5,-23,111,6,567,-3,34,127,1000,0,33,89,5,55,90,1,78,4,67,2] : int list

```
val t2 = list2heap(L2)
```

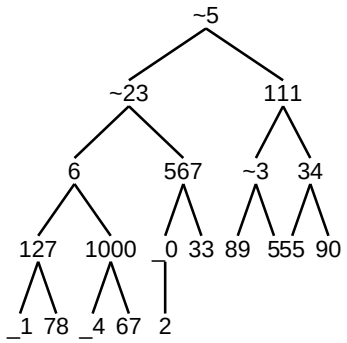


Figure 12. *t_2, ausgegeben mit dem in Kapitel 9 entwickelten ML-Programm*

Man sieht, daß ein mit *list2heap* erzeugter binärer Baum bis auf eine Weglängendifferenz von ± 1 ausbalanciert ist, wobei alle längeren Wege "links unten" gebündelt sind. Ein solcher Baum heißt **Heap**, wenn er außerdem **partiell geordnet** ist, d.h. die Knoteneinträge eines jeden Weges sortiert sind.

8.2.1 Heapsort

Ein binärer Baum wird mit folgender Funktion *heapify* partiell geordnet. Die Hilfsfunktion $sift$ permutiert aufeinanderfolgende Knotenpaare solange, bis die parteille Ordnung hergestellt ist.

```
 fun  heapify(t) = treefold(sift o T2)(t)(mt)
and  sift(t as T2(T2(l,y,r),x,mt)) = if x <= y:int then t
                                     else T2(sift(T2(l,x,r)),y,mt)
|    sift(t as T2(t1 as T2(l,y,r),x,t2 as T2(l',z,r')))
                = if x > y:int then if y <= z
```

```
                                        then T2(sift(T2(l,x,r)),y,t2)
                                        else (* x > y > z *)
                                             T2(t1,z,sift(T2(l',x,r')))
                                   else if x > z
                                        then (* y >= x > z *)
                                             T2(t1,z,sift(T2(l',x,r')))
                                        else t
        |    sift(t) = t
```

Ist x eine Variable und e ein Muster, dann ist auch der Ausdruck **x as e** ein Muster. Es hat dieselbe Bedeutung wie e, ist aber darüberhinaus unter dem Namen x verfügbar (i.a. innerhalb einer Funktionsdefinition).

heapify läßt sich zu einem Sortieralgorithmus für Listen erweitern, dem *heapsort*. Die unsortierte Liste wird mit *list2heap* und *heapify* in einen Heap h überführt. Da dessen Wurzel das kleinste Element unter allen Knoteneinträgen von h enthält, wird dieser Knoten aus h entfernt und zum ersten Element der sortierten Liste. Dann wird das - bzgl. der Heapordnung - größte Blatt von h zur neuen Wurzel von h und sein Eintrag mithilfe von *sift* solange "nach unten geschoben", bis er von Nachfolgereinträgen nicht mehr überschritten wird. h hat jetzt einen Knoten weniger und seine Wurzel enthält das zweitkleinste Element der Ausgangsliste. Die eben beschriebene Prozedur wird wiederholt. Ist n die Listenlänge, dann terminiert der Algorithmus nach maximal n Iterationen dieser Prozedur mit einer sortierten Permutation der Ausgangsliste.

```
 fun  loop(nil) = nil
|    loop[x] = [x]
|    loop(x::L) = let val L = last(L)::dequeue(L)              (s. 4.2)
                       val sift_list = Heaplist o sift o list2heap
                   in x::loop(sift_list(L)) end
val  heapify_list = Heaplist o heapify o list2heap
val  heapsort = loop o heapify_list
```

Effiziente Verfeinerungen (s. §7.3) von *heapsort* implementieren die Hilfsfunktionen *sift_list* und *heapify_list* von *loop* ohne den Umweg über Bäume direkt auf den entsprechenden Listen (oder Feldern).

8.3 Der Huffman-Algorithmus

erzeugt einen Binärcode für eine gegebene Zeichenmenge gemäß der Häufigkeit (*frequency*) jedes einzelnen Zeichens in einem gegebenen Text. Je öfter ein Zeichen auftritt, desto kürzer ist die ihm entsprechende Bitfolge. In diesem Sinne liefert der Algorithmus einen optimalen Code.

8.3.1 Text einlesen

Zunächst wird der Text mit der folgenden Funktion *read* aus einem File INPUTn eingelesen und als Zeichenliste vom Typ *string list* der weiteren Verarbeitung zur Verfügung gestellt:

```
fun  read(n:int) = let val file = open_in("INPUT"^makestring(n))
                       fun instring(s) = if end_of_stream(file) then s
                                         else instring(s^input_line(file))
                       val chars = explode(instring"")
                   in close_in(file); chars end
```

Die Standardfunktion *explode* : *string* → *string list* zerlegt einen String in seine einzelnen Zeichen.

8.3.2 Codebaum erzeugen

Aus der von *read* erzeugten Zeichenliste bildet *makefreq* ein Paar, bestehend aus der Menge s aller Zeichen des Textes (dargestellt als Liste) und der Funktion $freq$, die jedem Zeichen seine Häufigkeit im Text zuordnet:

```
fun  makefreq(chars) = let fun loop(acc)(nil) = acc
                           |   loop(s,freq)(c::chars)
                                 = let val s = if member(s)(c) then s else c::s
                                       fun new_freq(x) = if x = c then freq(x)+1
                                                                  else freq(x)
                                   in loop(s,new_freq)(chars) end
                       in loop(nil,fn _=>0)(chars) end
```

Aus der Zeichenmenge s wird eine Liste einknotiger Bäumen gebildet:

```
fun  makeTrees(s) = map(fn(c)=>leaf[c])(s)
```

Der Kern des Algorithmus ist eine Schleife, deren Rumpf eine Liste tL binärer Bäume verändert, deren Knoten Teilmengen von s enthalten. In jedem Iterationsschritt werden zwei Bäume t_1 und t_2 von tL mit Wurzeleinträgen s_1 bzw. s_2 durch einen neuen Baum t mit Wurzeleintrag $s_1@s_2$ und Unterbäumen t_1 und t_2 ersetzt. Dadurch wird tL immer kürzer, bis nur noch ein einziger Baum mit dem Wurzeleintrag *chars* übrigbleibt:

```
fun  Huffman(s,freq) = loop(freq) o makeTrees(s)
and  loop(freq)[t] = t
|    loop(freq)(tL) = let val t1 = Min(freq)(tL)
                          val tL = diff(tL,[t1])          (s. 4.1)
                          val t2 = Min(freq)(tL)
                          val tL = diff(tL,[t2])
                          val t = T2(t1,root(t1)@root(t2),t2)
                      in loop(freq)(t::tL) end
and  Min(freq)(t::tL) = let val freqsum = sum o map(freq) o root
                            fun g(t1,t2) = if freqsum(t1) <= freqsum(t2)
                                           then t1 else t2
                        in fold(g)(tL)(t) end
and  root(T2(_,x,_)) = x
```

$freqsum(t)$ berechnet also die Summe der Häufigkeiten der Zeichen des Wurzeleintrages von t. *Huffman* hat folgende Ein/Ausgaberelation:

$P(s, freq, t) \iff_{def}$ Für alle Zeichen $c, d \in s$ und Wege v, w von der Wurzel von t zu dem Blatt, das mit c bzw. d markiert ist, gilt:
$freq(c) < freq(d) \Rightarrow Länge(v) \geq Länge(w)$.

Wir zeigen die Korrektheit durch Hoare-Induktion (s. §5.4). Da *loop* nicht auf einem einzelnen Baum, sondern einer Baumliste operiert, hat eine passende Hoare-Invariante H neben den Eingabeparametern s und $freq$ die jeweils aktuelle Baumliste tL als weiteres Argument. Wir stellen fest, daß es für alle $c \in s$ genau ein Blatt eines Baumes von tL gibt, daß mit c markiert ist, also genau einen Weg $w(c, tL)$, der von der Wurzel zu diesem Blatt hinführt. Wie üblich ist die Hoare-Invariante eine *Verallgemeinerung* der P:

$H(s, freq, tL) \iff_{def} tL$ ist eine Baumliste derart, daß für alle Zeichen $c, d \in s$ gilt:

$(\mathbf{H_1(tL)})$ $freq(c) < freq(d) \Rightarrow Länge(w(c, tL)) \geq Länge(w(d, tL))$,

$(\mathbf{H_2(tL)})$ $freq(c) < freq(d)$ und $Länge(w(c, tL)) = Länge(w(d, tL))$ und c, d gehören zu zwei verschiedenen Bäumen t_1 bzw. t_2 von tL $\Rightarrow freqsum(t_1) < freqsum(t_2)$.

loop wählt in jedem Iterationsschritt aus der Baumliste tL zwei Bäume t_1 und t_2 aus, die minimal bzgl. $freqsum$ sind, und ersetzt tL durch die Baumliste tL', die aus tL entsteht, indem t_1 und t_2 zu $t =_{def} T2(t1, root(t_1)@root(t2), t_2)$ zusammengefügt werden. Die entscheidende Anforderung an H (vgl. §5.4) ist demnach die Implikation

$$H(s, freq, tL) \quad \Rightarrow \quad H(s, freq, tL').$$

Widerspruchsargument: Was wäre, wenn $H(s, freq, tL)$ gälte, $H(s, freq, tL')$ jedoch nicht?

1. Fall: $H_1(tL')$ gilt nicht. Dann gäbe es Zeichen c, d mit $freq(c) < freq(d)$ und $Länge(w(c, tL')) < Länge(w(d, tL'))$. Wegen $H_1(tL)$ wäre $Länge(w(c, tL)) = Länge(w(d, tL))$. Also gehört d o.B.d.A. zu t_1, c jedoch zu einem Baum $t' \in tL \setminus \{t1, t2\}$. Wegen $H_2(tL)$ wäre $freqsum(t') < freqsum(t_1)$, was der Wahl von t_1 und t_2 als minimale Elemente von tL widerspricht.

2. Fall: $H_2(tL')$ gilt nicht. Dann gäbe es Zeichen c, d mit $freq(c) < freq(d)$ und $Länge(w(c, tL')) = Länge(w(d, tL'))$ und c, d gehören zu zwei verschiedenen Bäumen t_1 bzw. t_2 von tL' mit $freqsum(t_1) \geq freqsum(t_2)$. Wegen $H_2(tL)$ gehört c o.B.d.A. zu t_1, d jedoch zu einem Baum von $tL \setminus \{t1, t2\}$. Umgekehrt geht's nicht wegen $freq(c) < freq(d)$ und der Wahl von t_1 und t_2 als minimale Elemente von tL. Dann wäre

$$Länge(w(c, tL)) < Länge(w(c, tL')) = Länge(w(d, tL')) = Länge(w(d, tL)),$$

was aber wegen $freq(c) < freq(d)$ $H_1(tL)$ widerspricht.

Selbstverständlich gilt H für die erste Baumliste tL, mit der *loop* aufgerufen wird, weil diese aus einzelnen Blättern besteht, also für alle $t \in tL$ $freqsum(t)$ mit der Häufigkeit des in t gespeicherten Zeichens zusammenfällt.

8.3.3 Codieren und decodieren

Die folgende Funktion *code* codiert den Text aus INPUTn unter Zuhilfenahme des von *Huffman* erzeugten Codebaumes in eine Bitfolge.

```
fun  code(n:int) = let val chars = read(n)
                       val (s,freq) = makefreq(chars)
                       val t = case chars of [c] => T2(leaf[c],[c],mt)
                                           |    _ => Huffman(s,freq)
                       val bits = mapconc(traverse(t))(chars)
                   in (bits,t) end

and  traverse(T2(mt,_,_)) _ = nil
|    traverse(T2(l,_,r))(c) = if member(root(l))(c) then 0::traverse(l)(c)
                                                    else 1::traverse(r)(c)
```

Denkt man sich alle nach links (bzw. rechts) laufenden Kanten des Codebaumes t mit 0 (bzw. 1) markiert, dann liefert die Markierung des Weges $w(c, t)$ von der Wurzel zum Blatt mit dem Eintrag c gerade die Codierung des Zeichens c (s.o.).

Die folgende Funktion *decode* übersetzt den als Bitfolge codierten Text wieder in den ursprünglichen Text und benutzt dabei den von *Huffman* erzeugten Codebaum.

```
fun  decode(bits)(t) = reverse(bits)(t)(t)
and  reverse(bits)(T2(mt,[c],mt))(t) = c^reverse(bits)(t)(t)
|    reverse(nil) _ _ = ""
|    reverse(0::bits)(T2(l,_,_))(t) = reverse(bits)(l)(t)
|    reverse(1::bits)(T2(_,_,r))(t) = reverse(bits)(r)(t)
```

Was tut wohl die folgende Funktion?

```
fun  wow(m:int,n:int) = let val (bits,t) = code(m)
                            val text = decode(bits)(t)
                            val file = open_out("OUT"^makestring(n))
                        in output(file,text); close_out(file) end
```

9 Bäume mit beliebigem Ausgrad

Die Menge der Bäume, deren Knoten eine beliebige (endliche), möglicherweise von Knoten zu Knoten wechselnde, Nachfolgerzahl (= Ausgrad) haben, läßt sich als folgender Datentyp *tree* beschreiben. Anstelle des *bintree*-Konstruktors *mt* verwenden wir hier den *list*-Konstruktor *nil*.

```
infix %
datatype 'a tree = % of 'a * ('a tree list)
```

tree und *list* (s. Kap. 7) sind demnach wechselseitig rekursiv definiert. *tree list* liefert die Menge der Baum*listen*.

```
fun leaf(x) = x%nil
fun root(x%tL) = x
fun subtrees(x%tL) = tL
fun height(x%tL) = fold(max)(map(height)(tL))(0)+1
```

val leaf = fn : 'a → 'a tree
val root = fn : 'a tree → 'a
val height = fn : 'a tree → int

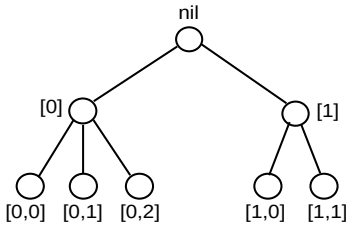


Figure 13. *Knotenidentifizierung*

Jeder Knoten K eines binären Baumes kann über eine - dem Weg von der Wurzel zu K entsprechende - Bitfolge identifiziert werden (analog der im letzten Abschnitt behandelten Binärcodierung). Bei Bäumen mit beliebigem Kontenausgrad wird daraus eine Folge (Liste) L natürlicher Zahlen. Wie der Datentyp *tree* selbst, so ist fast jede Baumoperation wechselseitig rekursiv mit einer Baumlistenoperation definiert.

Knotenzugriff

```
fun get(x%tL,nil) = x
|   get(x%tL,nL) = getL(tL,nL)
and getL(t::tL,0::nL) = get(t,nL)
```

```
|   getL(t::tL,n::nL) = getL(tL,(n-1)::nL)
```
*val get = fn : 'a tree * int list → 'a*
*val getL = fn : 'a tree list * int list → 'a*

Baum ⇝ binärer Baum

```
fun tree2bintree(t) = tree2bintreeL[t]
and tree2bintreeL(nil) = mt
|   tree2bintreeL((x%tL)::tL') = T2(tree2bintreeL(tL),x,tree2bintreeL(tL'))
```
val tree2bintree = fn : 'a tree → 'a bintree
val tree2bintreeL = fn : 'a tree list → 'a bintree

9.1 Bäume zeichnen mit PostScript

Wir wollen einen kleinen Compiler bauen, der Ausdrücke in Präfixnotation so in Graphikbefehle (PostScript-Programme) übersetzt, daß deren Ausführung durch einen PostScript-Interpreter das Zeichnen den Ausdrücken entsprechender Bäume bewirkt. Die eingelesenen Bäume könnten z.B. imperative Programme in abstrakter Syntax sein (s. Bsp. 7.1.2) wie das folgende:

```
loop(greater(var(x),Int(0)),
    seq(assign(f,prod(var(f),var(x))),
        assign(x,m(var(x),1)),
        assign(f,prod(var(f),var(assign(m(var(x),Int(1)),x)))),
        assign(m(var(x),Int(1)),x),
        assign(x,m(var(x),Int(1)))))
```

Unser Compiler erzeugt aus diesem Ausdruck eine Folge von Graphikbefehlen, bei deren Ausführung der Graph von Fig. 14 gezeichnet wird.

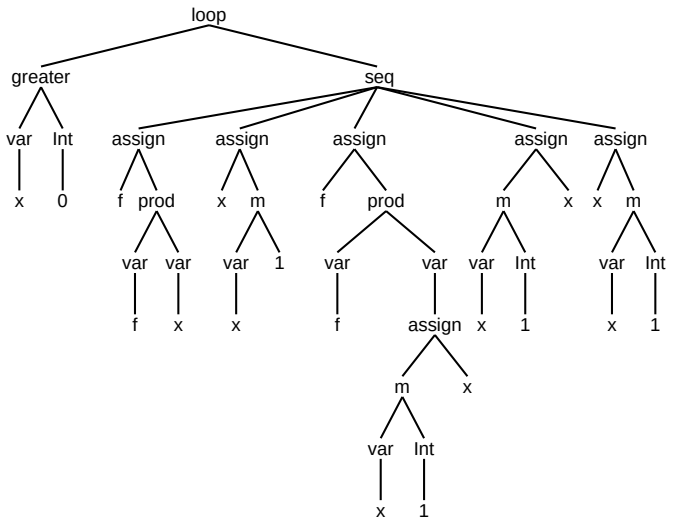


Figure 14. *Ein Baum wie wir ihn mögen*

9.1.1 Ein Parser

Zunächst brauchen wir einen **Parser**, der die Klammerstruktur mit *read* (s. §8.3.1) eingelesener Ausdrücke erkennt und in Elemente des Typs *string tree* übersetzt.

Generell übersetzt ein Parser Symbollisten in **Terme**, also Elemente eines konstruktorbasierten Datentyps. In der Regel stellt der Parser zunächst fest, ob die eingegebene Symbolliste überhaupt compilierbar ist und erzeugt ggf. Fehlermeldungen. In unserem Beispiel wird im Fehlerfall die Ausnahme *no_tree* aufgerufen. Ein für jede Übersetzungs- oder Auswertungsfunktion f wichtiges Entwurfsprinzip ist die geeignete Zerlegung des jeweiligen Definitions- oder Wertebereichs in mehrere möglicherweise wechselseitig-rekursive Datentypen $dt_1, \ldots, dt_n$ und eine entsprechende Zerlegung von f in n Teilfunktionen. So bestand der Kommando-Interpreter von 7.1.2 z.B. aus drei Funktionen *evalInt*, *evalBool* und *evalCom*. Hier sind es die beiden Typen *tree* und *tree list*, zwischen denen der Parser unterscheiden muß. Also benötigen wir zwei parse-Funktionen:

$$\begin{aligned} &\mathit{parseTerm} : \textbf{string list} \times \mathit{string} \rightarrow \textbf{string tree} \times \mathit{string\ list} \\ &\mathit{parseTerms} : \textbf{string list} \times \mathit{string\ tree\ list} \rightarrow \textbf{string tree list} \times \mathit{string\ list} \end{aligned}$$

Die fettgedruckten Ein- bzw. Ausgabetypen bezeichnen die eigentliche Definitions- bzw. Wertebereiche der parse-Funktionen. Hinzu kommen **vererbte Attribute** (= zusätzliche Eingabeparameter) und/oder **abgeleitete Attribute** (= zusätzliche Ausgabeparameter), die für die jeweilige Übersetzungsaufgabe notwendige Zusatzinformation darstellen. Sie wird entweder an die Funktion übergeben (*vererbt*) oder von dieser mitberechnet (*abgeleitet*). Bezogen auf die Struktur der zu übersetzenden Eingabe werden vererbte Attribute "von außen nach innen" *top down*) transportiert, abgeleitete Attribute hingegen "von innen nach außen" *bottom up*).

Im Beispiel beschreibt das abgeleitete *string list*-Attribut die Resteingabe nach Erzeugung eines Baums bzw. einer Baumliste. Das vererbte *string*-Attribut von *parseTerm* ist der bereits erkannte Teil der Wurzel des zu erkennenden Baums. Das vererbte *string tree list*-Attribut von *parseTerms* ist die bereits erkannte Teilliste der zu erkennenden Baumliste.

```
exception no_tree

fun  parseTerm(" "::chars,symbol) = parseTerm(chars,symbol)
|    parseTerm("("::chars,symbol) = let val (tL,chars) = parseTerms(chars,nil)
                                     in (symbol%tL,chars) end
|    parseTerm(","::chars,symbol) = (leaf(symbol),","::chars)
|    parseTerm(")"::chars,symbol) = (leaf(symbol),")"::chars)
|    parseTerm(x::chars,symbol) = parseTerm(chars,symbol^x)
|    parseTerm(nil,"") = raise no_tree
|    parseTerm(nil,symbol) = (leaf(symbol),nil)

and  parseTerms(" "::chars,tL) = parseTerms(chars,tL)
|    parseTerms(","::chars,tL) = parseTerms(chars,tL)
|    parseTerms(")"::chars,tL) = (tL,chars)
|    parseTerms(chars,tL) = let val (t,chars) = parseTerm(chars,"")
                            in parseTerms(chars,tL@[t]) end
```

9.1.2 Knotenkoordinaten berechnen

Der von *parseTerm* erzeugte Baum wird nun mit den Positionen der einzelnen Knoten "dekoriert", die später in Graphikbefehle eingesetzt werden müssen. Das besorgen die Funktionen

$$\begin{aligned} &\mathit{coordTree} : \textbf{string tree} \to \mathit{real} \times \mathit{real} \to (\textbf{string} \times \textbf{real} \times \textbf{real})\ \textbf{tree} \times \mathit{real} \\ &\mathit{coordTrees} : \textbf{string tree list} \to \mathit{real} \times \mathit{real} \to (\textbf{string} \times \textbf{real} \times \textbf{real})\ \textbf{tree list} \times \mathit{real}. \end{aligned}$$

Sie übersetzen einen Baum bzw. eine Baumliste mit Knoten a vom Typ *string* in einen Baum bzw. eine Baumliste mit Knoten (a, x, y) vom Typ $\mathit{string} \times \mathbb{R} \times \mathbb{R}$. So gesehen wird der Knoten a mit seinen Koordinaten x und y dekoriert.

```
fun  coordTree(a%nil)(x,y) = (leaf(a,x,y),strlg(a))
|    coordTree(a%tL)(x,y) = let val (ctL,b) = coordTrees(tL)(x,y-30.0)
                                val (_,x,_)%_ = hd(ctL)
                                val (_,z,_)%_ = last(ctL)    (s. 4.2)
                                val x = x*0.5+z*0.5
                                val r = strlg(a)
                                val b = if b >= r then b else r
                            in ((a,x,y)%ctL,b) end

and  strlg(a) = real(String.length(a)*7)

and  coordTrees(nil) _ = (nil,0.0)
|    coordTrees(t::tL)(x,y) = let val (ct,b) = coordTree(t)(x,y)
                                   val (ctL,b') = coordTrees(tL)(x+b,y)
                               in (ct::ctL,b+b') end
```

Einschließlich ihrer vererbten und abgeleiteten Attribute (Paare bzw. einzelne reelle Zahlen) haben *coordTree* und *coordTrees* folgende Ein/Ausgaberelationen:

$\mathit{coordTree}(t)(x, y) = (ct, b) \iff (x, y)$ ist die linke obere Ecke und b die Breite von t,
$\mathit{coordTrees}(tL)(x, y) = (ctL, b) \iff (x, y)$ ist die linke obere Ecke und b die Breite von tL.

9.1.3 Knoten und Kanten zeichnen

Der von *coordtree* erzeugte Baum ist ein Zwischencode, der nun durchlaufen werden muß, um PostScript-Code zu berechnen, dessen Ausführung bewirken soll, daß für einen Knoten (a, x, y) der String a um die Position (x, y) zentriert geschrieben wird und Kanten zu den Nachfolgern von (a, x, y) gezogen werden.

```
fun   traverseAll((a,x,y)%ctL) = moveto(x,y+10.0)^center(a)^
                                 traverse(ctL)(x,y)^" showpage"
and  moveto(x:real,y:real) = makestring(x)^" "^makestring(y)^" moveto\n"
and  center(a) = "("^a^") dup stringwidth pop -2 div -8 rmoveto show\n"

and  traverse(nil) _ = ""
|    traverse(((a,x,y)%ctL)::ctL')(pred) = moveto(pred)^lineto(a,x,y)^
                                           traverse(ctL)(x,y)^
                                           traverse(ctL')(pred)
```

```
and  lineto(a,x:real,y:real) = makestring(x)^" "^makestring(y+10.0)^" lineto\n"^
                               center(a)^"stroke\n"
```

Der von $center(a)$ erzeugte PostScript-Code bewirkt die um den aktuellen Punkt zentrierte Ausgabe von a. PostScript ist übrigens eine weitgehend funktionale Sprache mit postfix-notierten Funktionsaufrufen. Zur Bedeutung der hier erzeugten Kommandos verweisen wir auf einschlägige Handbücher.

Der Kern des letzten Übersetzungsschrittes ist die rekursive Funktion

$$traverse : (\mathbf{string} \times \mathbf{real} \times \mathbf{real})\ \mathbf{tree} \to real \times real \to \mathbf{string},$$

deren vererbtes Attribut $pred \in \mathbb{R}^2$ folgende Bedeutung hat:

$$traverse(ctL)(pred) \;=\; code \iff pred \text{ sind die Koordinaten des Vorgängerknotens von } ctL.$$

$pred$ muß an alle Elemente der Baumliste ctL vererbt werden, weil die Koordinaten von deren Wurzeln die Zielpunkte der von $pred$ aus zu ziehenden Linien sind.

Die Größe des Fensters, in das der Baum gezeichnet werden soll, hängt ab von dessen Höhe und der x-Koordinate seines am weitesten rechts stehenden Blattes. $maxX$ berechnet diese Koordinate wie folgt:

```
fun   maxX((_,x,_)%nil) = x
|     maxX((_,x,_)%ctL) = maxXL(ctL)

and   maxXL[ct] = maxX(ct)
|     maxXL(ct::ctL) = maxXL(ctL)
```

Die Prozedur $draw$ kombiniert $parseTerm$, $coordTree$ und $traverseAll$ zum Gesamtübersetzer, der einen Ausdruck aus dem File INPUTm einliest und den entsprechenden Baum in den File TREEn zeichnet:

```
fun  draw(m:int)(n:int)(scale:real)
       = let val chars = read(m)            (s. 8.3.1)
             val (t,_) = parseTerm(chars,"")
             val file = open_out("TREE"^makestring(n)^".eps")
             val h = real(height(t))*30.0
             val scalestr = makestring(scale)
             val (ct,_) = coordTree(t)(40.0,h)
             val code = "%!PS-Adobe-3.0 EPSF-3.0\n%%BoundingBox: 10 10 "^
                        makestring(scale*40.0+scale*maxX(ct))^" "^
                        makestring(scale*30.0+scale*h)^"\n"^
                        scalestr^" "^scalestr^" scale\n"^
                        "/Helvetica 9 selectfont\n"^traverseAll(ct)
         in output(file,code); close_out(file) end
```

Der Aufruf $draw$ 1 1 1.0 erzeugt folgenden PostScript-Code zum Zeichnen des Baumes von Fig. 14:[12]

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 10 10 444.0 300.0
1.0 1.0 scale /Helvetica 9 selectfont
```

[12] Funktionsaufrufe sind postfix-notiert. Der Name einer Funktion (lineto, center, moveto, usw.) folgt auf deren Argumente, die hier meistens Koordinaten sind.

```
/center { dup stringwidth pop -2 div -8 rmoveto show } def
148.9375 280.0 moveto (loop) center 148.9375 270.0 moveto
50.5 250.0 lineto (greater) center stroke 50.5 240.0 moveto
40.0 220.0 lineto (var) center stroke 40.0 210.0 moveto
40.0 190.0 lineto (x) center stroke 50.5 240.0 moveto
61.0 220.0 lineto (Int) center stroke 61.0 210.0 moveto
61.0 190.0 lineto (0) center stroke 148.9375 270.0 moveto
247.375 250.0 lineto (seq) center stroke 247.375 240.0 moveto
110.0 220.0 lineto (assign) center stroke 110.0 210.0 moveto
89.0 190.0 lineto (fact) center stroke 110.0 210.0 moveto
131.0 190.0 lineto (prod) center stroke 131.0 180.0 moveto
117.0 160.0 lineto (var) center stroke 117.0 150.0 moveto
117.0 130.0 lineto (fact) center stroke 131.0 180.0 moveto
145.0 160.0 lineto (var) center stroke 145.0 150.0 moveto
145.0 130.0 lineto (x) center stroke 247.375 240.0 moveto
174.75 220.0 lineto (assign) center stroke 174.75 210.0 moveto
166.0 190.0 lineto (x) center stroke 174.75 210.0 moveto
183.5 190.0 lineto (minus) center stroke 183.5 180.0 moveto
173.0 160.0 lineto (var) center stroke 173.0 150.0 moveto
173.0 130.0 lineto (x) center stroke 183.5 180.0 moveto
194.0 160.0 lineto (1) center stroke 247.375 240.0 moveto
242.5625 220.0 lineto (assign) center stroke 242.5625 210.0 moveto
215.0 190.0 lineto (fact) center stroke 242.5625 210.0 moveto
270.125 190.0 lineto (prod) center stroke 270.125 180.0 moveto
243.0 160.0 lineto (var) center stroke 243.0 150.0 moveto
243.0 130.0 lineto (fact) center stroke 270.125 180.0 moveto
297.25 160.0 lineto (var) center stroke 297.25 150.0 moveto
297.25 130.0 lineto (assign) center stroke 297.25 120.0 moveto
281.5 100.0 lineto (minus) center stroke 281.5 90.0 moveto
271.0 70.0 lineto (var) center stroke 271.0 60.0 moveto
271.0 40.0 lineto (x) center stroke 281.5 90.0 moveto
292.0 70.0 lineto (Int) center stroke 292.0 60.0 moveto
292.0 40.0 lineto (1) center stroke 297.25 120.0 moveto
313.0 100.0 lineto (x) center stroke 247.375 240.0 moveto
346.25 220.0 lineto (assign) center stroke 346.25 210.0 moveto
330.5 190.0 lineto (minus) center stroke 330.5 180.0 moveto
320.0 160.0 lineto (var) center stroke 320.0 150.0 moveto
320.0 130.0 lineto (x) center stroke 330.5 180.0 moveto
341.0 160.0 lineto (Int) center stroke 341.0 150.0 moveto
341.0 130.0 lineto (1) center stroke 346.25 210.0 moveto
362.0 190.0 lineto (x) center stroke 247.375 240.0 moveto
384.75 220.0 lineto (assign) center stroke 384.75 210.0 moveto
376.0 190.0 lineto (x) center stroke 384.75 210.0 moveto
393.5 190.0 lineto (minus) center stroke 393.5 180.0 moveto
383.0 160.0 lineto (var) center stroke 383.0 150.0 moveto
383.0 130.0 lineto (x) center stroke 393.5 180.0 moveto
404.0 160.0 lineto (Int) center stroke 404.0 150.0 moveto
404.0 130.0 lineto (1) center stroke showpage
```

10 Dynamische Objekte

können ihre Werte auch innerhalb ein und desselben Gültigkeitsbereiches verändern (s. §2.6). Zwar verändert in folgendem Programmstück die Variable a zweimal ihren Wert:

```
val a = 5
val a = 6
val a = a+1
val a = 5 : int
val a = 6 : int
val a = 7 : int
```

Mit jeder Redefinition von a wird aber ein neuer Scope eröffnet und dabei der alte gelöscht. Es werden also hintereinander drei *statische* Objekte definiert, deren Werte sich nicht ändern. Das Programmstück ist nur eine Kurzform des folgenden:

```
val a1 = 5
val a2 = 6
val a3 = a2+1
```

Hier wird deutlich, daß die Werte 5, 6 und 7 nicht notwendig in derselben (virtuellen) Speicherzelle abgelegt werden. Will man gerade das zu erreichen, dann muß a als dynamisches Objekt, d.h. als **Zeiger** (**Referenz**), definiert werden, der auf das eigentliche Objekt (hier vom Typ *int*) verweist:

```
val a = ref(5)
val _ = (a:=6; a:= !a+1)
val a = ref 7 : int ref
```

a selbst hat jetzt den Typ $int\ ref$. Die Wertdefinition *val a = ref(5)* erzeugt sowohl das Element 5 als auch den Zeiger a darauf. Zeiger müssen stets einen monomorphen Typ haben (s. 2.7)! Beim Übersetzen einer Zeigerdefinition wird nämlich Speicherplatz für das Element, auf das der Zeiger zeigt, reserviert, abhängig vom Typ des Elementes. Hätte es einen polymorphen Typ, dann wüßte der Compiler nicht, in welchen Grenzen sich die Größe des Objekts (oder seiner Komponenten) zur Laufzeit bewegen würde.

Die **Zuweisung** a:=6 setzt den Zeiger a auf das Element 6. Als Funktion hat := den polymorphen Typ *'a ref * 'a → unit*. Das Semikolon ";" erzwingt die Hintereinanderausführung von Funktionsaufrufen (s. §2.2.1). "!" bezeichnet die **Dereferenzierungsfunktion**, die den polymorphen Typ *'a ref → 'a* hat. Durch den Aufruf !a erreicht man den Wert, auf den a zeigt.

Wir fassen die hier eingeführten Standardfunktionen zusammen:

- Referenzierung *ref : 'a → 'a ref* ist ein polymorpher Konstruktor,
- Dereferenzierung *! : 'a ref → 'a* ist eine polymorphe Funktion,
- Zuweisungsoperator *:= : ('a ref * 'a) → unit* und Sequentialisierung *; : ('a_1 * ... * 'a_n) → 'a_n* sind ebenfalls polymorphe Funktionen.

10.1 Statische und dynamische Bindung

An den vorangehenden Beispielen erkennt man noch nicht den semantischen Unterschied zwischen der Definition eines Wertes x vom Typ a einerseits und der Zuweisung an eine Variable x vom Typ a, die ja eigentlich ein Wert vom Typ $ref\ a$ ist. Dieser Unterschied wird erst deutlich, wenn man zunächst x in der Definition einer Funktion f benutzt, später verändet und schließlich f aufruft:

x statisch		x dynamisch
`val x = 7`		`val x = ref(7)`
`fun f(y) = x+y`		`fun f(y) = !x+y`
`val x = x+2`		`val _ = x:= !x+2`
	`val z = f(3)`	
val x = 9 : int		*val x = ref 9 : int ref*
val z = 10 : int		*val z = 12 : int*

In die linke Definition von f könnte bereits der Compiler den Wert von x (also 7) einsetzen, weil hier der unmittelbar vor der Definition von f gültige Wert von x mit dem innerhalb der Definition gültigen Wert übereinstimmt. Die darauffolgende Wertdefinition *val a = a+2* ändert darn nichts mehr. In der rechten Definition von f hingegen bleibt der Wert von $!x$ solange unbekannt, bis f aufgerufen wird, was erst in der die Wertdefinition *val z = f(3)* geschieht. Inzwischen hat die Zuweisung *x:= !x+2* aber den Wert von $!x$ von 7 zu 9 erhöht, so daß bei der Ausführung des Aufrufs $f(3)$ nunmehr 9 für $!x$ eingesetzt wird. So ergeben sich schließlich zwei unterschiedliche Werte von $f(3)$.

Links ist x **statisch gebunden**: Für jeden Aufruf von f gilt derselbe – an der Stelle der **Definition** von f gültige – Wert von x. Rechts ist x **dynamisch gebunden**: Für jeden Aufruf von f gilt möglicherweise ein anderer – an der jeweiligen **Aufruf**stelle gültige – Wert von x.

Beispiel In §8.3.3 kopiert die Funktion *decode* den Baum t, bevor sie ihn an *reverse* übergibt:

```
fun  decode(bits)(t) = reverse(bits)(t)(t)
```

Dasgleiche passiert bei rekursiven Aufrufen von *reverse*. Bei hinreichend großem t kann dieses Kopieren zu einer merklichen Zunahme des Zeit- und Platzbedarfs von *decode* führen. Da *decode* die erzeugte Kopie von t niemals verändert, genügt es, an ihrer Stelle einen Zeiger rt an *reverse* zu übergeben:

```
fun  decode(bits)(t) = let val rt = ref(t) in reverse(bits)(t)(rt) end
and  reverse(bits)(T2(mt,[c],mt))(rt) = c^reverse(bits)(!rt)(rt)
|    reverse(nil) _ _ = ""
|    reverse(0::bits)(T2(l,_,_))(rt) = reverse(bits)(l)(rt)
|    reverse(1::bits)(T2(_,_,r))(rt) = reverse(bits)(r)(rt)
```

In der ersten Gleichung von *reverse* wird rt dereferenziert und, da rt auf die Wurzel des gesamten Codebaums zeigt, dieser einer erneuten Traversierung zugänglich gemacht.

Nun wird aber auch das verbliebene Exemplar des Codebaums während der Decodierung nicht verändert, sondern nur durchlaufen, so daß wir noch einen Schritt weitergehen und auch dieses Exemplar nicht **by value**, sondern nur **by reference** an *reverse* übergeben. Der dafür erforderliche Zeiger *st* muß allerdings bei jedem Aufruf von *reverse* dereferenziert werden, damit weiterhin die Werte dieser Funktion durch das jeweilige Argumentmuster bestimmt werden können:

```
fun  decode(bits)(t) = let val rt = ref(t) in reverse(bits)(rt)(rt) end
and  reverse(nil)(st)(rt) = (case !st of T2(mt,[c],mt) => c | _ => "")
|    reverse(b::bits)(st)(rt)
            = case (!st,b) of (T2(mt,[c],mt),_) => c^reverse(b::bits)(st)(rt)
                           |  (T2(l,_,_),0) => reverse(bits)(ref(l))(rt)
                           |  (T2(_,_,r),1) => reverse(bits)(ref(r))(rt)
```

10.2 Die Türme von Hanoi

Eine Menge von Bauklötzen ist vom Ort a zum Ort c zu transportieren, wobei ein weiterer Ort b als Zwischenlager dient. Die Klötze sollen mit nach oben abnehmender Breite übereinandergestapelt werden. Jeder während des Transports erreichte **Zustand** (*state*) wird als Tripel (A, B, C) von Listen der bei a, b bzw. c gelagerten Klötze dargestellt. Eine zulässige Zustandsfolge, die den Transport von drei Klötzen 1, 2 und 3 beschreibt, lautet z.B. wie folgt:

```
A : 1 2 3 B : C :
A : 2 3 B : C : 1
A : 3 B : 2 C : 1
A : 3 B : 1 2 C :
A : B : 1 2 C : 3
A : 1 B : 2 C : 3
A : 1 B : C : 2 3
A : B : C : 1 2 3
```

Ist L die Liste der zu transportierenden Klötze, dann stellt (L, nil, nil) den Anfangszustand dar. Den Übergang von einem Zustand zum jeweiligen Folgezustand beschreibt die **Übergangsfunktion**

$$trans : \ 'klotz\ state \to ort^3 \to int \to \ 'klotz\ state.$$

Sie wird von $hanoi : \ 'klotz\ list \to \ 'klotz\ state$ im Anfangszustand aufgerufen:

```
datatype ort = a | b | c
type 'klotz state = 'klotz list * 'klotz list * 'klotz list
fun  hanoi(L) = let val n = length(L)
                    val state = (L,nil,nil)
                in outState(state); trans(state)(a,b,c)(n) end

and  trans(state) _ (0) = state
|    trans(state)(x,y,z)(n)
           = let val (A,B,C) = trans(state)(x,z,y)(n-1)
                 val state = case (x,z) of (a,b) => (tl(A),hd(A)::B,C)
                                         |  (a,c) => (tl(A),B,hd(A)::C)
                                         |  (b,a) => (hd(B)::A,tl(B),C)
                                         |  (b,c) => (A,tl(B),hd(B)::C)
                                         |  (c,a) => (hd(C)::A,B,tl(C))
                                         |  (c,b) => (A,hd(C)::B,tl(C))
             in outState(state); trans(state)(y,x,z)(n-1) end
```

Die Funktion $outState : \ 'klotz\ list \to unit$ gibt Zustände aus (s.u.). Die Anzahl der Aufrufe von *trans* hängt von der Länge der Klotzliste L ab. *trans* wird für jedes $1 \leq i \leq n$ zweimal rekursiv aufgerufen. Das zweite Argument (x, y, z) von *trans* gibt die Orte an, die jeweils Ausgangsort (x), Zwischenlager (y) bzw. Zielort (z) sind. Jeder von (x, y, z) angenommene Wert ist eine Permutation von (a, b, c).

Der Aufruf trans(state)(x,y,z)(n) zerlegt den Auftrag *Schaffe n Objekte von x über y nach z* in drei Teile:

- *Schaffe* $n-1$ *Objekte von x über z nach y*: trans(state)(x,z,y)(n-1).
- *Transportiere das einzige bei x verbliebene Objekt nach z*. Hier wird der Zustand verändert (und anschließend ausgegeben).

- *Schaffe* $n-1$ *Objekte von* y *über* x *nach* z:trans(state)(y,x,z)(n-1).

Aufgabe: Definieren Sie die Funktion *outState* so, daß bei jedem Aufruf von *hanoi* die jeweils berechnete Zustandsfolge wie in obigem Beispiel ausgegeben wird.

Zwar werden von *trans* keine Zustände kopiert. Es kann sich bei ihnen aber um große Objekte handeln, deren *by value* Übergabe an *trans* aufwendig ist. Analog der Referenz auf einen Codebaum übergeben wir also lieber nur einen Zeiger auf den aktuellen Zustand. Nachdem die Funktion *trans* diesen geändert hat, liefert sie einen Zeiger auf den neuen Zustand zurück:

```
datatype ort = a | b | c
type state = int list * int list * int list
val  rstate = ref((nil,nil,nil):state)
fun  hanoi(L) = let val n = length(L)
                in rstate:=(L,nil,nil);
                   outState(!rstate); trans(rstate)(a,b,c)(n) end

and  trans(rstate) _ (0) = rstate
|    trans(rstate)(x,y,z)(n)
             = let val (A,B,C) = !trans(rstate)(x,z,y)(n-1)
                   val state = ... s.o. ...
               in outState(state); trans(ref(state))(y,x,z)(n-1) end
```

val hanoi = fn : int list → state ref
*val trans = fn : state ref → ort * ort * ort → state ref*

Leider können wir den Algorithmus hier nicht polymorph für beliebige Klotztypen formulieren, weil Zeiger nur auf Objekte monomorphen Typs zeigen dürfen.

Wir wandeln *hanoi* jetzt in eine Übersetzungsfunktion *hanoiPS* um, die PostScript-Programme erzeugt, bei deren Ausführung Bilder der jeweiligen Zustandsfolgen entstehen – wie in Fig. 15 (vgl. §9.1.3).

```
datatype ort = a | b | c
type state = int list * int list * int list
val  rstate = ref((nil,nil,nil):state)
fun  hanoiPS(L) = let fun exp2(0) = 1
                      |   exp2(k) = 2*exp2(k-1)
                      val n = length(L)
                      val h = exp2(n)*100
                      val _ = rstate:=(L,nil,nil)
                      val code1 = "%!PS-Adobe-3.0 EPSF-3.0\n"^
                                  "%%BoundingBox: 5 5 560 "^makestring(h)^
                                  "\n20 "^makestring(h-80)^
                                  " translate\n1 setlinewidth\n"^
                                  outState(!rstate)
                      val (_,code2) = trans(rstate)(a,b,c)(n)
                      val file = open_out"HANOI.eps"
                  in output(file,code1^code2); close_out(file) end

and  trans(rstate) _ (0) = (rstate,"")
```

```
|    trans(rstate)(x,y,z)(n)
            = let val (rstate,code1) = trans(rstate)(x,z,y)(n-1)
                  val (A,B,C) = !rstate
                  val state = ... s.o. ...
                  val (rstate,code2) = trans(ref(state))(y,x,z)(n-1)
              in (rstate, code1^outState(state)^code2) end

and  outState(A,B,C) = rectangles(A,length(A)-1)^"200 0 translate\n"^
                       rectangles(B,length(B)-1)^"200 0 translate\n"^
                       rectangles(C,length(C)-1)^"-400 0 translate\n"^
                       "0 0 moveto 520 0 lineto stroke\n"^
                       "0 -80 translate\n"

and  rectangles(nil,_) = ""
|    rectangles(x::T,n) = makestring(n*10)^" "^makestring(n*10)^" "^
                          makestring(x*20)^" 10 rectstroke\n"^rectangles(T,n-1)
```

val hanoiPS = fn : int list → string
*val trans = fn : state ref → ort * ort * ort → state ref * string*
val outState = fn : state → string
*val rectangles = fn : int list * int → string*

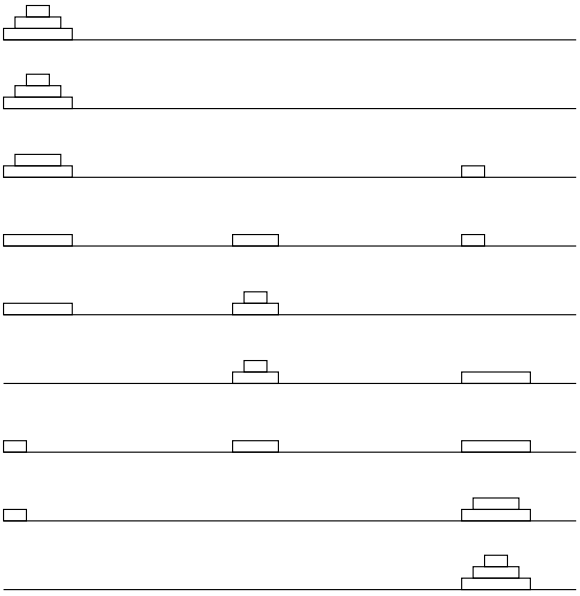


Figure 15. *hanoiPS[1,2,3]*[13]

Eine weitere Variante von *hanoi* liefert anstelle einer Zustandsfolge die Sequenz der Einzelaktionen, aus denen sich ein Transportvorgang zusammensetzt.

```
datatype ort = a | b | c
fun  actions(L) = transAct(a,b,c)(length(L))
```

[13] Probieren Sie's aus bis $n = 6$. Für größeres n muß das PostScript-Programm mit anderen Fenstermaßen, Abständen, usw. parametrisiert werden.

```
and  transAct _ (0) = nil
|    transAct(x,y,z)(n) = let val preActs = transAct(x,z,y)(n-1)
                              val postActs = transAct(y,x,z)(n-1)
                          in preActs@(x,z)::postActs end
```

Jede Einzelaktion (= Transport eines Klotzes) ist hier dargestellt als Paar *(Ausgangsort,Zielort)*. Ähnlich wie die vorangehende Version von *hanoi* erzeugt auch *actions* Code, nur sind es hier keine Graphikbefehle, sondern Steuerkommandos, die z.B. einen Roboter veranlassen könnten, die Klötze in der gewünschten Weise zu transportieren.

10.3 Verkettete Datenstrukturen

In imperativen Sprachen wird eine Liste, ein Baum oder ein Graph meistens als Menge ihrer Elemente bzw. seiner Knoten dargestellt, auf die mittels Zeiger zugegriffen wird. Die Zeiger stellen die jeweilige (lineare, baumartige oder gar zyklische) Struktur her, in der die Elemente/Knoten angeordnet sein sollen. Tatsächlich handelt es sich hierbei um nichtrekursive *Implementierungen* (Verfeinerungen; s. §7.3) rekursiver oder funktionaler Datentypen. Diese sind beim Entwurf eines Programms wegen ihrer direkten Zugänglichkeit zu Korrektheitsüberlegungen verzeigerten Strukturen i.a. vorzuziehen. Auf einer späteren Entwicklungsstufe verlangt die dort vorgegebene Implementierungssprache allerdings oft, daß funktionale und rekursive Typen in Zeigerstrukturen übersetzt werden. Dabei werden, grob gesagt, alle in Typdefinitionen auftretenden rekursiven Typaufrufe und Funktionspfeile durch den Zeigerkonstruktor ref ersetzt. Aus dem rekursiven Typ *a list* wird z.B.:

```
infix &
datatype 'a elem = Nil | & of 'a * 'a elem ref
```
con Nil : 'a elem
*con & : 'a * 'a elem ref* $\rightarrow$ *'a elem*

Hier werden eigentlich zwei Typen definiert, nämlich $'a\ elem$ und $'a\ elem\ ref$. Letzterer ist der Typ der Zeiger auf Listenelemente. Eine Liste von n Elementen des Typs a entsteht durch Verkettung von n Objekten des Typs $'a\ elem$, graphisch dargestellt wie in Fig. 16 dar.

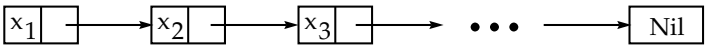


Figure 16. *Verkettete Liste*

Traversiert werden verkettete Listen entlang ihrer Zeiger. Die folgende Funktion

$$Map(f) : 'a\ elem\ ref \rightarrow 'b\ elem\ ref$$

wendet $f : 'a \rightarrow 'b$ auf jedes Listenelement an, implementiert also $map(f) : 'a\ list \rightarrow 'b\ list$ auf verketteten Listen (vgl. §4.1):

```
fun  Map(f)(re) = case !re of Nil => ref(Nil)
                           |  x&next => ref(f(x)&Map(f)(next))
```

Die folgende Funktion übersetzt jede "abstrakte" Liste des Typs $'a\ list$ in eine verkettete Liste und liefert einen Zeiger auf das erste Listenelement:

```
fun  list2pointer(nil) = ref(Nil)
|    list2pointer(x::L) = ref(x&list2pointer(L))
```

val list2pointer = fn : 'a list → 'a elem ref

I.a. bestimmt die Reihenfolge, in der Elemente einer Menge M besucht werden sollen, die Art der Verkettung der Struktur, die M repräsentiert. So unterscheidet man z.B. die oben eingeführten *einfach verketteten* Listen von *Ringlisten* und *doppelt verketteten* Listen:

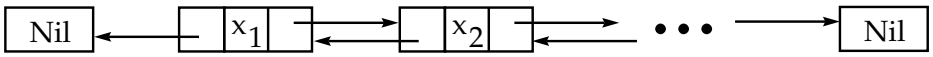


Figure 17. *Doppelt verkettete Liste*

```
datatype 'a delem = Nil | el2 of 'a delem ref * 'a * 'a delem ref
```
con Nil : 'a delem
*con el2 : 'a delem ref * 'a * 'a delem ref → 'a delem*

10.3.1 Bäume und Graphen

des funktionalen Typs $'a \rightarrow\ 'a\ list$ können auf zweierlei Weise in verkettete Strukturen überführt werden (vgl. Kap. 6 und 9):

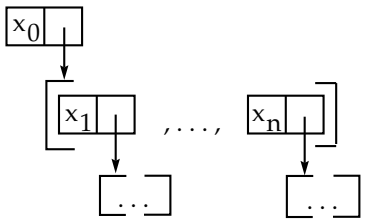


Figure 18. *Graph mit "abstrakten" Nachfolgerlisten*

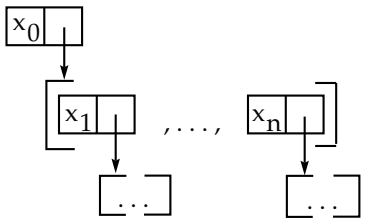


Figure 19. *Graph mit verketteten Nachfolgerlisten*

```
infix %
datatype 'a node = % of 'a * 'a node list ref
```
*con % : 'a * 'a node list ref → 'a node*
oder

```
datatype 'a binode = Nil | node2 of 'a * 'a binode ref * 'a binode ref
```
con Nil : 'a binode
*con node2 : 'a * 'a binode ref * 'a binode ref → 'a binode*

'a node macht den Funktionspfeil des abstrakten Graphtyps $a \rightarrow a\ list$ zum Zeigerkonstruktor ref. *'a binode* eliminiert darüberhinaus den abstrakten Listenkonstruktor *list* durch Verkettung aller (direkten) Nachfolger

eines jeden Knotens. Damit wird – analog zur Übersetzung beliebiger in binäre Bäume (s. Kap. 9) – jeder Graph zum **binären Graphen**, egal wieviele Nachfolger seine Knoten haben.

Die folgende Funktion implementiert jeden abstrakten Graphen des Typs *'a → 'a list*, dessen Knoten von *root* aus erreichbar sind, als Verkettung von Objekten des Typs *'a node* und liefert einen Zeiger auf das aus *root* gebildete Objekt dieses Typs:

```
fun  graph2pointer(G)(root)
              = let fun makeNodes(G)(nil) = nil
                    |   makeNodes(G)(x::L) = let val succs = makeNodes(G)(G(x))
                                                 val sibls = makeNodes(G)(L)
                                             in (x%ref(succs))::sibls end
                in ref o hd o makeNodes(G)[root] end
```

val graph2pointer = fn : ('a → 'a list) → 'a → 'a node ref
val makeNodes = fn : ('a → 'a list) → 'a list → 'a node list

Die Übersetzung eines Graphen des Typs *a → a list* in einen binären Graphen mit Knoten des Typs *a node2* lautet wie folgt:

```
fun  graph2bipointer(G)(root)
              = let fun makeNode(G)(nil) = Nil
                    |   makeNode(G)(x::L) = let val succs = makeNode(G)(G(x))
                                                val sibls = makeNode(G)(L)
                                            in node2(x,ref(succs),ref(sibls)) end
                in ref o makeNode(G)[root] end
```

val graph2bipointer = fn : ('a → 'a list) → 'a → 'a binode ref
val makeNode = fn : ('a → 'a list) → 'a list → 'a binode

Traversiert werden binäre Graphen wie verkettete Listen entlang ihrer Zeiger. Die folgende Funktion

MapG(f) : 'a binode ref → 'b binode ref

wendet *f : 'a → 'b* auf jeden Graphknoten an:

```
fun  MapG(f)(rn)
       = case !rn of Nil => ref(Nil)
                   |  node2(x,succs,sibls)
                          => ref o node2(f(x),MapG(f)(succs),MapG(f)(sibls))
```

11 Modularisierung

Ein Modul ist ein in sich abgeschlossenes Programmstück, dessen Funktion unabhängig vom Rest des Programms definiert und oft auch übersetzt werden kann. Es kann folgende **Schnittstellen** (**interfaces**) zu seiner Umgebung haben:

- eine **Import**-Schnittstelle, das sind alle Typen und Werte (einschließlich Funktionen, Prozeduren und Referenzen) aus seiner Umgebung, die es benutzen darf,

- eine **Parameter**-Schnittstelle, das sind alle Typen und Werte aus seiner Umgebung, die ihm explizit übergeben werden,
- eine **Export**-Schnittstelle, das sind alle Typen und Werte, die es seiner Umgebung zur Verfügung stellt.

Die Importschnittstelle eines abstrakten Datentyps (s. §7.3) besteht aus allen vor ihm definierten Typen und Werten. Es gibt keine expliziten Parameter, und alle Komponenten des abstrakten Datentyps *außer den Konstruktoren* bilden seine Exportschnittstelle.

Mehr Möglichkeiten der Strukturierung als der *abstype* bietet das *structure*-Konstrukt von ML. Eine Struktur ist ein wie das gesamte Programm aufgebauter Modul. Er kann alle Arten von Definitionen: Typ-, Wert-, Funktions- und wiederum Strukturdefinitionen, enthalten. Eine Struktur ist wie ein Wert ein *statisches* Objekt. Wie man Werte zu Typen zusammenfaßt, so kann man Strukturen zu **Signaturen** zusammenfassen. Wie man Funktionen auf Wertemengen definiert, so kann man Funktionen auf Strukturmengen definieren, die dann **Funktoren** genannt werden. Es ergibt sich folgende Analogie:

Wertebene	**Modulebene**
Wert	Struktur
Typ	Signatur
Funktion $f : type_1 \to type_2$	Funktor $F : signature_1 \to signature_2$

11.1 Strukturen

Eine Strukturdefinition hat folgendes Schema:

```
structure Modul = struct local lokale Definitionen
                          in exportierte Definitionen end
                  end
```

Enthält *Modul* keine lokalen Definitionen, dann bilden *alle* zwischen `struct` und `end` definierten Objekte die Exportschnittstelle von *Modul*. Die Importschnittstelle besteht wie die eines *abstype* aus allen vor der Struktur definierten Objekten. Außerhalb von *Modul* ist ein exportiertes Objekt f unter dem – durch den Strukturnamen *qualifizierten* – Namen *Modul.f* zugänglich. Der Befehl

```
open Modul
```

öffnet die Struktur, d.h. ihr Inhalt wird an die Befehlsstelle kopiert. Danach sind alle exportierten Objekte von *Modul* ohne Qualifizierung durch den Strukturnamen verfügbar. Damit wird die Kennzeichnung der Zugriffe auf *Modul* nicht mehr erzwungen, so daß ein Teil der mit der Einführung von *Modul* beabsichtigten Strukturierung des Gesamtprogramms verlorengeht.

Strukturdefinitionen dürfen geschachtelt werden, allerdings nicht im Wechsel mit Funktions- oder Wertdefinitionen. Damit läßt sich z.B. das Konzepte der **Vererbung** (**inheritance**) objektorientierter Sprachen realisieren:

```
structure Erbe = struct open Modul
                         neue oder geänderte Definitionen
                 end
```

Erbe übernimmt (erbt) alle Objekte von *Modul*. Neue Definitionen können dazugefügt werden. Es dürfen aber auch Objekte von *Modul* geändert werden, indem ein bereits in *Modul* definiertes Objekt f eine neue Definition erhält, womit die alte automatisch überschrieben wird. Außerhalb von *Modul* und *Erbe* sind dann mehrere Definitionen von f verfügbar: die *Modul*-Definition unter dem Namen *Modul.f* und die *Erbe*-Definition unter dem Namen *Erbe.f*.

11.2 Signaturen

Eine Signatur ist der Typ einer Struktur. Sie wird als geklammerte **Spezifikation** definiert:

`signature sigM = sig` *Spezifikation* `end`

Die Struktur *Modul* wird zu einer Struktur der Signatur *sigM*, wenn wir *sigM* wie einen Typ ans Ende der Strukturdefinition stellen:

`structure Modul : sigM = struct` *Definitionen* `end`

Wie bei einem typisierten Ausdruck $e : t$ vom Compiler geprüft wird, ob e wirklich den Typ t hat, so wird hier festgestellt, ob *Modul* eine Struktur der Signatur *sigM* ist. Dazu werden die Definitionen von *Modul* mit den Komponenten von *sigM* verglichen. Die Spezifikation von *sigM* könnte z.B. lauten:

```
type t      val a : bool      val f : t list → bool
```

Dann prüft der Compiler, ob in *Modul* ein Typ t, ein Boolescher Wert a und eine Funktion $f : t\ list \to bool$ definiert sind. *Modul* darf weitere Definitionen enthalten, die von *sigM* verlangten müssen aber auf jeden Fall vorkommen. Wie das Beispiel zeigt, wird ein Typ allein durch einen Namen spezifiziert, Werte und Funktionen hingegen durch Name und Typ. Lautet die Definition von *sigM* also wie folgt:

```
signature sigM = sig type t
                     val a : bool
                     val f : t list -> bool end
```

dann ist z.B.

```
structure Modul : sigM = struct type t = int
                                 val a = true
                                 fun f(L) = null(tl(L))
                                 val b = 0 end
```

ein korrekt "typisierter" Modul. Man sieht an dem Beispiel, daß der Typ einer Funktion f, die in der Struktur *Modul* definiert wird, *allgemeiner* sein darf als der Typ der Spezifikation von f in der Signatur von *Modul*: Da *Modul* den Typ t durch *int* definiert, müßte f eigentlich als Funktion vom Typ $int\ list \to bool$ definiert werden. In *Modul* hat f aber den allgemeineren Typ $'a\ list \to bool$.

Eine Signatur kann auch eine Strukturspezifikation enthalten. Diese ist durch einen Namen für die Struktur und eine Signatur gegeben, z.B.:

`structure S : sigM`

Eine Struktur S wird definiert wie in §11.1 beschrieben oder durch Gleichsetzung mit dem Namen einer schon früher definierten Struktur, z.B.:

`structure S = Modul`

11.3 Funktoren

Betrachtet man Strukturen als Modul*werte*, dann sind **Funktoren** Modul*funktionen*. Funktoren sind immer erster Ordnung, d.h. Funktoren können nur auf Strukturen, nicht aber auf Funktoren angewendet werden, und sie liefern auch nur Strukturen. Ist der Funktor einstellig, hat er also genau ein (Struktur-) Argument und besteht die Signatur dieses Argumentes nur aus wenigen Komponenten, dann kann man diese Komponenten auch direkt – ohne Zusammenfassung zu einer Signatur – übergeben. Äquivalent zur Funktordefinition

```
functor F(structure S : sigM) = struct Definitionen end
```

ist also z.B. die folgende:

```
functor F1(type t val a : bool val f : t list → bool)
          = struct Definitionen end
```

Was hier – praktisch als *formaler Parameter* – an $F1$ übergeben wird, ist gerade die Spezifikation, also der "Rumpf" von $sigM$.

Die Definitionen im Rumpf der Definition von F (oder $F1$) bilden wieder eine Struktur, und zwar diejenige, die F als Wert liefert. Hat diese die Signatur, sagen wir, $sigFM$, dann könnte man F tatsächlich als Funktion auffassen. Ihr Definitionsbereich ist die Menge aller Strukturen der Signatur $sigM$. Ihr Wertebereich ist die Menge aller Strukturen der Signatur $sigFM$, kurz: $F : sigM \rightarrow sigFM$. Damit findet sich das funktionale Konzept auch auf der Modulebene wieder!

Leider dürfen Funktoraufrufe nicht geschachtelt werden. Syntaktisch korrekte Aufruf von F und $F1$ sind z.B. folgende Strukturdefinitionen:

```
structure FModul = F(structure S = Modul)
```

bzw.

```
structure F1Modul = F1(type t = int
                       val a = true
                       fun f(L) = null(tl(L)))
```

Auf $FModul$ und $F1Modul$ können wieder Funktoren angewendet werden, nicht jedoch direkt auf die rechten Seiten dieser Gleichungen. Auch *open* (s. §11.1) darf nur auf Strukturnamen und nicht auf Funktoraufrufe angewendet werden.

Wie schon in §11.2 bemerkt, darf der Typ einer definierten Funktion f allgemeiner sein als der von einer Spezifikation von f verlangte Typ dieser Funktion. Die Umkehrung ist nicht erlaubt, weil sie zu Typkonflikten bei der Aktualisierung von Funktorparametern führen kann. Erwartet z.B. ein Funktor G eine polymorphe Funktion f, von der G eine echte Instanz benutzt, dann muß diese Instanz auch Instanz aller möglichen Aktualisierungen von f sein. Das ist aber nur dann gewährleistet, wenn die Aktualisierungen von f einen mindestens so allgemeinen Typ wie f haben.

Beispiel 11.3.1 *Typkonflikt bei der Aktualisierung eines Funktionsparameters f*

```
functor G(val f : 'a list -> 'a) = struct fun g(L:int list) = f(L)
                                          fun h(L:bool list) = f(L) end
structure GModul = G(fun f(L) = hd(L)+1)
```

Error: value type in structure doesn't match signature spec
name: f
spec: 'a list → 'a
actual: int list → int

Beispiel 11.3.2 *Wörterbuch (dictionary)* Der folgende Funktor definiert, abhängig von einem Schlüsseltyp *key* und einer Relation *less* auf Schlüsseln, eine Struktur, die aus dem Datentyp *'a dict* und den Funktionen *lookup* und *enter* besteht. *mt* bezeichnet ein leeres Wörterbuch. *item(k,a,above,below)* beschreibt ein Wörterbuch, das

unter dem Schlüssel k aufgeschlagen ist und dort den Eintrag a hat, während *above* und *below* das vor k bzw. hinter k stehende Teilwörterbuch bezeichnen.

eqtype wird einem formalen Typparameter vorangestellt, der nur durch Typen aktualisiert werden soll, auf denen = und <> (ungleich) definiert ist (vgl. §2.8).

```
functor makeDict(eqtype key  val less : key * key -> bool)
         = struct datatype 'a dict = mt | item of key * 'a * 'a dict * 'a dict
                  fun lookup(item(k,a,above,below),k')
                       = if k = k' then a else if less(k,k') then lookup(above,k')
                                                             else lookup(below,k')
                  fun enter(mt,k,a) = item(k,a,mt,mt)
                  |   enter(item(k,a,above,below),k',b)
                       = if k = k' then item(k,b,above,below)
                                   else if less(k,k')
                                        then item(k,a,enter(above,k',b),below)
                                        else item(k,a,above,enter(below,k',b))
           end
```

Ein korrekter Aufruf von *makeDict*:

```
structure intDict = makeDict(type key = int  fun less(k:int,k') = k < k')
```

Eine Struktur, die sich als Funktoraufruf der Form $F(F(S))$ beschreiben läßt, könnte man als **reflexiven Modul** bezeichnen. So etwas kommt in der "realen Welt" durchaus vor. Grundlegend für die Modularisierung des in Kap. 12 behandelten Regal-Beispiels ist z.B. die Feststellung, daß Container aus Objekten bestehen – was einen Funktor *Cont* nahelegt – und daß Regalteile Container sind und gleichzeitig Objekte eines "höheren" Containers, dem Regalsystem. Letzteres ist also im Prinzip durch einen doppelten Aufruf von *Cont*: *Cont(Cont(Obj))*, beschrieben (s. §12.5).

11.4 Objektorientierte Programmierung

In §11.1 wurde angedeutet, wie sich die in objektorientierten Sprachen (*OO-Sprachen*) übliche Vererbungsbeziehung zwischen Moduln in einer Strukturdefinition darstellt. Wenn in *Erbe* keine Definitionen von *Modul* geändert, sondern nur neue hinzugefügt werden, dann überträgt sich die Vererbungsbeziehung auf die Signaturen *sigM* und *sigE* von *Modul* bzw. *Erbe*: *sigM* ist eine **Teilsignatur** *sigE*. Umgekehrt ist die Menge der *sigE*-Strukturen eine Teilmenge der Menge der *sigM*-Strukturen. Vererbungsbeziehungen bedeuten also im Grunde nichts anderes als Teilmengenbeziehungen. Die Moduln selbst heißen in OO-Sprachen üblicherweise **Klassen**. Eine Klasse enthält die Definition eines ausgezeichneten **dynamischen Objektes** (s. §10) *dynObj*, auf das sich die anderen Definitionen der Klasse beziehen und dessen Typ mit dem Klassennamen gleichgesetzt wird. Erzeugt wird *obj* aber erst, wenn man die Klasse selbst aufruft. Klassen einerseits und ihre Aufrufe andererseits können wir in ML dadurch realisieren, daß wir erstere als Funktoren darstellen:

```
functor class() = local val dynObj = ref( first_value)
                  in  Definitionen end
structure newObj = class()
```

Objektdeklarationen einer OO-Sprache sind also in ML Definitionen von Strukturen, die sich aus Funktoraufrufen ergeben. Klassen*parameter* entsprechen Funktorargumenten, die dann meistens zur Initialisierung von *dynObj* benutzt werden, z.B.:

```
functor classp(val x:int) = local val dynObj = ref(x)
                            in  Definitionen end
structure newObj = classp(val x = 5)
```

Das dynamische Objekt *dynObj* ist lokal definiert, womit – wie in OO-Sprachen üblich – darauf nur mehr indirekt über Aufrufe anderer in der Klasse definierter Werte oder Funktionen zugegriffen werden kann.

Beispiel 11.4.1 *Keller als Klasse* (vgl. §4.2)

```
functor intStack() = struct local val stack = ref(nil:int list)
                                exception Top
                            in fun lg() = length(!stack)
                               fun empty() = null(!stack)
                               fun full() = lg() > 100
                               fun push(x) = if full() then () else stack:=x::(!stack)
                               fun pop() = if empty() then () else stack:=tl(!stack)
                               fun top() = if empty() then raise Top else hd(!stack)
                            end
                     end
structure s1 = intStack()
structure s2 = intStack()
val x = (s1.push(5); s1.push(8); s1.pop(); s2.push(s1.top()); s1.push(6);
         s2.top()+s1.top())
```

val x = 5 : int

Man sieht an diesem Beispiel, wie durch Aufrufe von Funktionen einer Klasse mehrere dynamische Objekte desselben Typs verändert bzw. "beobachtet" werden. Funktionen nach *unit* werden in imperativen Sprachen **Prozeduren** genannt. Sind sie in Klassen definiert, spricht man von **Methoden** und betrachtet ihre Aufrufe als an das Objekt, in dessen Klasse sie definiert sind, geschickte **Nachrichten**. Das macht deutlich, weshalb OO-Sprachen insbesondere bei der Programmierung **verteilter Systeme** eingesetzt werden. Die anderen Funktionen der Klasse heißen **Attribute**, weil sie i.d.R. Teilinformation über den jeweiligen Zustand des dynamischen Objektes liefern.

Polymorph ist ein Funktor, wenn er Typparameter hat. In OO-Sprachen nennt man entsprechende Klassen **generische Klassen**, z.B.:

```
functor genStack(type entry) = struct local val stack = ref(nil:entry list)
                                      exception Top
                                      in ... s.o. ... end
                               end
structure s1 = genStack(type entry = int)
structure s2 = genStack(type entry = bool)
val x = (s1.push(5); s1.push(8); s1.pop(); s2.push(s1.top()<6); s2.top())
```

val x = true : bool

entry ist keine Typvariable, sondern ein Typparameter. Ein Objekt *stack* gibt es erst, wenn der Funktor *genStack* aufgerufen wird. *entry* muß durch einen monomorphen Typ aktualisiert werden, weil *stack* ein Zeiger ist und Zeiger immer monomorphe Typen haben müssen (s. Kap. 10).

Die oben erwähnte Teilmengenbeziehung zwischen einer Klasse und ihren Erben kann in manchen funktionalen Sprachen auch auf der (primitiveren) Typebene formuliert werden. So könnte eine Typdeklaration $int < real$ ausdrücken, daß *int* ein **Subtyp** von *real* ist, was die Konsequenz hätte, daß alle *int*-Objekte gleichzeitig *real*-Objekte sind. Man kann sich leicht vorstellen, daß Typinferenz in Sprachen, die ein solches *Subtyping* erlaubt, aufwendiger ist als in Sprachen ohne Subtyping wie z.B. ML.

Es ist kein Zufall, daß gerade OO-Sprachen Subtyping erlauben (in Form von Vererbungsbeziehungen), während funktionale damit zurückhaltender umgehen. Erstens gibt es in OO-Sprachen kaum Typinferenz (weshalb man die meisten Programmierfehler erst zur Laufzeit erkennt). Zweitens heißt eine Sprache *objekt*orientiert, wenn ihre Konzepte erzwingen, daß Funktionen den Objekten (Typen, Klassen), auf denen sie operieren, zu- und damit *untergeordnet* werden. Typinferenz hingegen basiert auf der entgegengesetzten Sichtweise: Die Definition einer Funktion in Gestalt des sie definierenden funktionalen Ausdrucks *bestimmt die Objekte* (Typen, Definitions- und Wertebereiche) fest, auf denen sie operiert. Subtyping impliziert aber, daß Funktionen mehrere Typen haben können, wobei der jeweils gültige nicht von der Funktionsdefinition, sondern vom jeweiligen (Argument-) Objekt abhängt.

Man sollte deshalb, sofern die gegebene Sprache kein eigenes Subtyping-Konstrukt vorsieht, einen Typ mit Subtypen als eine Klasse mit Erben realisieren. Ein konstruktorbasierter Datentyp

$$\texttt{datatype dt = con}_1\ \texttt{of typ}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{con}_n\ \texttt{of typ}_n$$

eignet sich dafür weniger, weil sich kein anderer Typ, auch kein gewünschter Subtyp von *dt*, auch nur einen einzigen Konstruktor mit *dt* teilen kann. Folglich können auf *dt*-Mustern basierende Funktionsdefinitionen nicht einfach auf Subtypen vererbt werden. Sie müssen – auf Subtypmustern – neu definiert werden! Der Schritt von *dt* zu entsprechenden Klassendefinitionen bestünde in ML darin, für jeden Konstruktor con_i einen Funktor einzuführen:

$$\texttt{functor Con}_i\texttt{(x:typ}_i\texttt{) = struct fun F(y) = e}_i\texttt{(x) ... end}$$

Die Funktionsdefinition im Rumpf von Con_i entsteht dabei aus der Gleichung $f(con_i(x), y) = e_i(x, y)$ der Definition einer Funktion f auf *dt*. Das ursprünglich durch den Term $con_i(a)$ dargestellte Objekt wird jetzt durch den Funktoraufruf

$$\texttt{structure obj = Con}_i\texttt{(a)}$$

erzeugt. Der Funktionsaufruf $f(con_i(a), b)$ wird zu $obj.f(b)$. Im Gegensatz zur Menge der Konstruktoren von *dt* kann man die Klassenmenge $\{Con_1, \ldots, Con_n\}$ an weitere Strukturen vererben und so Subtypen von *dt* definieren. Die Erben können die Definition von f erweitern, ohne daß dies Änderungen der "Eltern" $\{Con_1, \ldots, Con_n\}$ erfordert.

Beispiel 11.4.2 Eine kleine Klassenhierarchie: Punkte, Polygone und Rechtecke

```
functor Point(val xy:real*real)
                = struct local val point = ref(xy)
                         in fun xy() = !point
                            fun new(xy') = point:=xy'
                            fun translate(a,b) = new(#1(!point)+a,#2(!point)+b)
                            fun scale(c) = new(#1(!point)*c,#2(!point)*c)
                            fun distanceTo(x',y') = let val a = #1(!point)-x'
                                                        val b = #2(!point)-y'
                                                    in sqrt(a*a+b*b) end
```

```
                        end
                end

structure p = Point(val xy = (1.2,3.4))
val xy = (p.translate(7.0,4.0); p.xy())
val d = p.distanceTo(8.2,7.4)
```

*val xy = (8.2,7.4) : real * real*
val d = 0.0 : real

```
functor Polygon(val pL:(real*real)list)
         = struct local val poly = ref(pL)
                       structure p = Point(val xy = (0.0,0.0))
                in fun new(pL) = poly:=pL
                   fun translate(ab) = let fun f(xy) = (p.new(xy);
                                                        p.translate(ab);
                                                        p.xy())
                                       in new(map(f)(!poly)) end
                   fun perimeter() = let fun f(xy::z::pL) = (p.new(xy);
                                                             p.distanceTo(z)+f(z::pL))
                                       |   f _ = 0.0
                                       val xy::pL = !poly
                                     in f(xy::pL@[xy]) end
                end
           end

structure poly = Polygon(val pL = [(1.2,3.4),(5.6,3.3),(2.9,8.1)])
val umfang1 = poly.perimeter()
val umfang2 = (poly.translate(3.8,12.1); poly.perimeter())
```

val umfang1 = 14.9064037419787 : real
val umfang2 = 14.9064037419787 : real

```
functor Rectangle()
         = struct local structure rect = Polygon(val pL = nil)
                       val le = ref(0.0)
                       val he = ref(0.0)
                       structure p = Point(val xy = (0.0,0.0))
                in fun new(x,y,l,h) = let val x' = x+l
                                          val y' = y+h
                                      in rect.new[(x,y),(x',y),(x',y'),(x,y')];
                                         le:=l; he:=h end
                   fun len() = !le
                   fun hei() = !he
                   fun perimeter() = 2.0*(!le + !he)
                   fun diagonal() = p.distanceTo(!le,!he)
                end
           end
```

```
structure poly = Polygon(val pL = [(1.2,3.4),(7.0,3.4),(7.0,16.0),(1.2,16.0)])
val umfang1 = poly.perimeter()
structure rect = Rectangle()
val umfang2 = (rect.new(1.2,3.4,5.8,12.6); rect.perimeter())
val dia = rect.diagonal()
```

val umfang1 = 36.8 : real
val umfang2 = 36.8 : real
val diagonal = 13.8708327075198 : real

Es stellt sich die Frage, ob diese Strukturen den Aufwand einer Klassenrealisierung wirklich lohnen oder ob sie als statische Typen nicht einfacher und übersichtlicher implementierbar sind. Oft ist die Verwendung objektorientierter Konstrukte zur Lösung kleiner Probleme (**programming in the small**) zu aufwendig. Klassen und andere Modulkonzepte sind eher für große Systeme geeignet (**programming in the large**), die modularisiert werden, damit die einzelnen Moduln unabhängig voneinander entwickelt, verfeinert und später über ihre Schnittstellen zusammengesetzt werden können. Die Beispiele in diesem Abschnitt sollten das Klassenkonzept und seine Beziehung zum Typkonzept veranschaulichen. Seine wirkliche Anwendung liegt im Bereich großer, insbesondere verteilter, Softwaresysteme, wo dann auch seine dynamische Komponente eine wesentliche Rolle spielt (**concurrent programming**). Mehrere Klassenhierarchien werden zu ganzen **Klassenbibliotheken** zusammmengesetzt, die wie Werkzeuge flexibel nutzbar sein sollen.

12 Regale bauen

In diesem Abschnitt sollen mehrere oben eingeführte Sprachkonzepte bei der Implementierung einer größeren Aufgabe eingesetzt werden. Hier geht es also nicht darum, ein neues Konzept an beliebigen kleinen Beispielen zu demonstrieren, sondern umgekehrt, ausgehend von einer praktischen Problemstellung, einige Sprachkonzepte oder algorithmische Ansätze gezielt auszuwählen, um bestimmte Teilaufgaben zu lösen. Im einzelnen werden die folgenden Themen behandelt:

- das Aufzählen von Permutationen einer Liste (12.1),
- die Darstellung interner Objekte mithilfe konstruktorbasierter Datentypen (12.2),
- das Codieren von Eingabeinformation in internen Objekten (12.3),
- die Steuerung des Datenflusses mithilfe von Ausnahmen und dynamischen Variablen (12.4),
- die Modularisierung mithilfe von Signaturen, Strukturen und Funktoren (12.5),
- die Übersetzung interner Objekte in Ausgabecode (12.6),
- zustandsbasiertes Programmieren (12.7).

Die Problemstellung: Zweidimensionale Container sollen mit Objekten verschiedener Größe und verschiedenen Typs gefüllt werden. Dabei sind veränderliche Restriktionen zu beachten, die als erlaubte oder verbotene Nachbarschaftsbeziehungen zwischen Objekten ausgedrückt werden. Das Ergebnis ist eine erlaubte Ladung einer gegebenen Menge von Containern mit einer gegebenen Menge von Objekten. Nachdem eine Ladung berechnet wurde, wird sie graphisch ausgegeben. Der Benutzer soll dann weitere Ladungen (= Anordnungen der Objekte in den Containern) anfordern können. Früher berechnete Ladungen dürfen sich wiederholen, aber die Berechnung neuer Ladungen sollte nicht in in einen Zyklus geraten und alle erlaubten Ladungen sollten irgendwann erreicht werden. Ein Regal entsteht, da auch die Container auf einer vorgegebenen Fläche unterschiedlich angeordnet

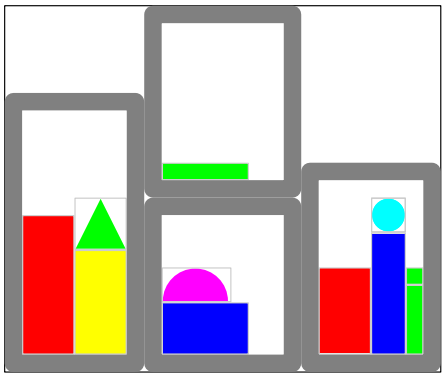


Figure 20. *Ein 2-dimensionales Regal*

werden können. Programmkomponenten, die das Laden der Objekte in die Container steuern, sollten auch für den Zusammenbau eines Regals aus Regalteilen (= beladenen Containern) "wiederverwendbar" sein (*reusability requirement*).

12.1 Permutationen

Da wir die *Seitenansicht* eines Regals herstellen wollen, wird jeder Container von unten nach oben und auf jeder Ebene von links nach rechts beladen. So können von vornherein unrealistische vertikale Zwischenräume vermieden werden. Die Anordnung der Objekte in einer Liste bestimmt die Reihenfolge, in der sie in die Container eingefüllt werden, und damit die Beladung selbst. Wir brauchen einen Algorithmus, der mehrere Permutationen der Objektliste berechnet, damit verschiedene Beladungen gefunden werden können. Der folgende zählt *alle* Permutationen auf:

```
fun  allperms[x] = [[x]]
|    allperms(x::L) = mapconc(insert(x))(allperms(L))(nil)      (s. 4.1)

and  insert(x)(L) = let fun shift(L,x,nil) = [L@[x]]
                        |   shift(L,x,y::R) = (L@x::y::R)::shift(L@[y],x,R)
                    in shift(nil,x,L) end
```

val allperms = fn : 'a list → 'a list list
`val L = allperms[1,2,3]`
val L = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]] : int list list

Die Hilfsfunktion $insert(x)$ bildet alle Permutationen von $x :: L$ derart, daß die Reihenfolge der Elemente von L innerhalb von $x :: L$ erhalten bleibt. Nur die Position von x variiert. Die Korrektheit von *allperms* ist leicht einzusehen. Der Berechnungsaufwand ist jedoch sehr hoch, insbesondere wenn man berücksichtigt, daß wir nicht unbedingt *alle* Permutationen ermitteln wollen. Eine n-elementige Liste hat n! Permutationen. Sind die Elemente komplexe Datenstrukturen, dann kann dieser überexponentielle Aufwand den Algorithmus unbrauchbar machen. Das Problem liegt in der zweiten Gleichung von *allperms*: Um Permutationen von $x :: L$ zu erhalten, müssen zunächst alle Permutationen von L berechnet werden.

Geeigneter ist die folgende Funktion $nextperm(r)$, die, abhängig von einer totalen Ordnung[14] $r : A \times A \to bool$ auf den Listenelementen, zu einer Liste L die in umgekehrter **alphabetischer** oder **lexikographischer Ordnung** bzgl. r auf L folgende Permutation berechnet. Die kleinste (größte) Liste in dieser Ordnung ist die bzgl. r absteigend (aufsteigend) sortierte Permutation von L. Mit der kleinsten fangen wir an, mit der größten hören wir auf. *sort* sei irgendein Sortieralgorithmus (s. §4.4).

```
fun   firstperm(r) = sort(not o r)

exception no_nextperm
fun  nextperm(r)(x::L) = next(r)([x],L)

and  next(r)(L1,z::L2)
       = if r(hd(L1),z) then next(r)(z::L1,L2)
         else let fun swap[x] = z::x::L2
                  |   swap(x::y::L) = if not(r(y,z)) then x::swap(y::L)
                                      else (z::y::L)@(x::L2)
              in swap(L1) end
|    next(r)(_,nil) = raise no_nextperm

fun  nthperm(0) _ (L) = L
|    nthperm(n)(r)(L) = nthperm(n-1)(r)(nextperm(r)(L))
```

Die Korrektheit von *nextperm* wird im Buch von Paulson begründet. $nthperm(n)(r)(L)$ liefert, beginnend mit L, die nächsten $n+1$ Permutationen von L in umgekehrter lexikographischer Ordnung bzgl. r.

Die lexikographische Ordnung bzgl. $\leq$: $int \times int \to bool$ ist z.B. wie folgt definiert:

```
fun   lex(nil,nil) = true
|    lex(x::L,y::L') = (x:int) < y orelse (x = y andalso lex(L,L'))
```

12.2 Konstruktorbasierte Datentypen

Unter der Annahme, daß ein Typ *obj* für die einzufüllenden Objekte gegeben ist, definieren wir Containerbeladungen als abstrakten Datentyp *cont* (vgl. §7.3). Der Wert $new(l,h)$ steht für einen leeren Container der Länge l und der Höhe h. Der Wert $add(a,i,j,c)$ bezeichnet den Container, der das Objekt a an der (Anfangs-) Position (i,j) enthält und ansonsten mit c übereinstimmt.

```
abstype cont = new of int * int | add of obj * int * int * cont

with    val     New = new
        val     Add = add

        fun     len(new(l,_)) = l
        |       len(add(_,_,_,c)) = len(c)

        fun     hei(new(_,h)) = h
```

[14] Total muß sie sein, damit immer entschieden werden kann, ob eine Permutation kleiner oder größer als eine andere ist. Antisymmetrie hingegen ist nicht gefordert, obwohl zu berücksichtigen ist, daß $nextperm(r)$ Permutationen überspringen wird, wenn r nicht antisymmetrisch ist. Das wiederum ist dann sinnvoll, wenn von einer Permutation P aus schneller eine von P stark abweichende Permutation erreicht werden soll (vgl. §12.5).

```
        |       hei(add(_,_,_,c)) = hei(c)

        fun     pos(add(a,i,j,c),b) = if a = b then (i,j) else pos(c,b)

        fun     Objs(new _) = nil
        |       Objs(add(a,_,_,c)) = a::Objs(c)

        fun     makeEmpty(new(l,h)) = new(l,h)
        |       makeEmpty(add(a,_,_,c)) = makeEmpty(c)

        exception undef
        fun     get(new _,_,_) = raise undef
        |       get(add(a,i,j,c),x,y) = if inside(a,i,j)(x,y) then a else get(c,x,y)

        and     inside(a,i:int,j:int)(x,y) = i <= x andalso x < i+le(a) andalso
                                             j <= y andalso y < j+he(a)

        fun     frame(new(l,h)) = new(l+2,h+2)
        |       frame(add(a,i,j,c)) = add(a,i+1,j+1,frame(c))

end     (* cont *)
```

Die Funktion *inside* verwendet Funktionen $le, he : obj \to int$, die wie der Typ *obj* als gegeben vorausgesetzt werden (vgl. §12.5).

In Bsp. 7.3.1 haben wir *cont* zum abstrakten Datentyp $refined\_cont$ verfeinert, der eine Containerbeladung nicht mehr als $\{new, add\}$-Term repräsentiert, sondern als eine Matrix implementiert, die einer diskrete graphische Darstellung des Containers wiedergibt. Z.B. wird der "abstrakte" Container

$$add(obj_1, 1, 1, add(obj_2, 1, 4, add(obj_3, 5, 4, add(obj_4, 2, 4, new(8, 6)))))$$

durch folgende Matrix implementiert:[15]

```
  6 |   1  1  1  3  3  0  0  0
  5 |   1  1  1  3  3  0  0  0
  4 |   1  1  1  4  4  4  4  4
  3 |   1  1  1  4  4  4  4  4
  2 |   1  1  1  4  4  4  4  4
  1 |   1  1  1  2  2  2  2  2
      ------------------------
        1  2  3  4  5  6  7  8
```

12.3 Scanning und Parsing

Alle von einem File eingelesenen Daten bilden für das verarbeitende Programm zunächst nur einen einzigen großen String. Die Eingabe zu Fig. 20 (Objekte und Container) lautet z.B.:

```
(3,6,ry)(5,1,rg)(2,2,ct)(5,3,rb)(4,2,av)(1,1,rg)(1,4,rg)(2,2,ct)(3,8,rr)
```

[15] Genaugenommen ist die Matrix ein Objekt von Typ *obj list list*, das hier als String dargestellt ist.

```
(3,3,tg)(3,5,rr)(2,7,rb)
conts (7,8),(6,14),(7,9),(6,10)
```

Wir wenden zunächst die Einlesefunktion *read* an (siehe §8.3.1). $read(n)$ liest den Inhalt des Files INPUTn und erstellt daraus die Zeichenliste *chars*. Als **Scanning** oder **lexikalische Analyse** bezeichnet man den ersten Teil der Verarbeitung von *chars*, der darin besteht, mehrere Zeichen zu Symbolen zusammenzufassen. Das Ergebnis ist aber immer noch eine *Liste* von Strings. Im Unterschied dazu nennt man den darauffolgenden Umbau dieser Liste in einen Baum **Parsing** oder **syntaktische Analyse** (siehe auch §9.1.1). Hier werden viele Eingabefehler erkannt und gemeldet. Im Fall einer syntaktisch korrekten Eingabe werden irrelevante Symbole aus der Symbolliste entfernt und der Rest in einen Term (**Syntaxbaum**) übersetzt, dessen Struktur wesentliche Teile der Bedeutung der ursprünglichen Eingabe wiedergibt.

In unserem Regal-Beispiel umfaßt die Eingabe eine Objektliste und eine Liste leerer Container. Im Sinne des oben definierten abstype *cont* vereinbaren wir, daß ein Objekt als Tripel (l, h, tc) zweier natürlicher Zahlen l und h und zweier Buchstaben t und c eingegeben wird, wobei l, h, t, c die Länge, Höhe, den Polygontyp bzw. die Farbe des Objektes bezeichnet. Der Parser soll sicherstellen, daß eine Auflistung solcher Tripel als Objektliste interpretiert wird. Wenn im Eingabefile der String *conts* auftritt, werden darauffolgende Paare (l, h) natürlicher Zahlen als leere Container der jeweiligen Länge l und Höhe h interpretiert. Demnach muß der Parser drei Arten von Strukturen unterscheiden und geeignet übersetzen:

- Zahlen als Komponenten von Objekten oder Containern,
- Buchstaben als Typ- bzw. Farbinformation,
- Tripel vor dem String *conts* als Objekte,
- Paare hinter *conts* als Container.

Daraus ergeben sich folgende parse-Funktionen.

```
datatype polytype = rect | arc | circ | tri
exception no_type and no_color and wrong_size

fun  parseNo(x::chars,no) = if ord(x)>47 andalso ord(x)<58
                            then parseNo(chars,no^x)
                            else (makeint(no),chars)

fun  parseType(x::chars) = (case x of "r" => rect
                                    |  "a" => arc
                                    |  "c" => circ
                                    |  "t" => tri
                                    |  _ => raise no_type, chars)

fun  parseColor(x::")"::chars) = (case x of "b" => 1
                                         |  "g" => 2
                                         |  "t" => 3
                                         |  "r" => 4
                                         |  "v" => 5
                                         |  "y" => 6
                                         |  _ => raise no_color, chars)
```

```
|    parseColor _ = raise no_color

fun  parseObjects("c"::"o"::"n"::"t"::"s"::chars,objs) = (objs,chars)
|    parseObjects("("::chars,objs)
       = let val (l,chars) = parseNo(chars,"")
             val (h,chars) = parseNo(chars,"")
             val (t,chars) = parseType(chars)
             val (c,chars) = parseColor(chars)
         in if (t = arc andalso l <> 2*h) orelse
               (t = circ andalso l <> h) then raise wrong_size
            else parseObjects(chars,(l,h,t,c)::objs) end
|    parseObjects(x::chars,objs) = parseObjects(chars,objs)

fun  parseConts(nil,conts) = conts
|    parseConts("("::chars,conts) = let val (l,chars) = parseNo(chars,"")
                                        val (h,chars) = parseNo(chars,"")
                                    in parseConts(chars,(l,h)::conts) end
|    parseConts(x::chars,conts) = parseConts(chars,conts)

fun  parseAll(chars) = let val (objects,chars) = parseObjects(chars,nil)
                           val conts = parseConts(chars,nil)
                       in (objects,conts) end
```

Allen parse-Funktionen liegt das gleiche Typschema zugrunde:

$$
\begin{aligned}
&\mathit{parseNo} : \textbf{string list} \times \mathit{string} \to \textbf{int} \times \mathit{string\ list}\\
&\mathit{parseType} : \mathit{string\ list} \to \textbf{polytype}\\
&\mathit{parseColor} : \mathit{string\ list} \to \textbf{int}\\
&\mathit{parseObjects} : \mathit{string\ list} \times (\mathit{int} \times \mathit{int} \times \mathit{int})\ \mathit{list} \to (\textbf{int} \times \textbf{int} \times \textbf{int})\ \textbf{list} \times \mathit{string\ list}\\
&\mathit{parseConts} : \mathit{string\ list} \times (\mathit{int} \times \mathit{int})\ \mathit{list} \to (\textbf{int} \times \textbf{int})\ \textbf{list}
\end{aligned}
$$

Das *string list*-Argument ist immer die Resteingabe vor Aufruf der jeweiligen parse-Funktion. Der *string list*-Wert ist immer die Resteingabe nach Erzeugung einer Struktur (Zahl, Polygontyp, Objekt- oder Containerliste). Im zweiten Argument der parse-Funktion wird diese Struktur schrittweise aufgebaut. Zusammen mit der Resteingabe bildet sie den Wert der parse-Funktion.

parseNo übersetzt eine Zeichenkette in die entsprechende natürliche Zahl, wenn ein nicht als Ziffer interpretierbares Zeichen in der Eingabe erscheint.

In §9.1 wurde nach dem gleichen Schema ein Parser für Bäume definiert.

12.4 Ausnahmen und dynamische Variablen

Den Kern des Regal-Beispiels bildet die Funktion

$$\mathit{fill} : \mathit{obj\ list} \to \mathit{cont\ list} \to \mathit{cont\ list},$$

durch deren Aufruf $\mathit{fill}(\mathit{objs})(\mathit{contSizes})$ die Objekte von *objs* in leere Container mit den Abmessungen von *contSizes* in der durch die beiden Listen gegebenen Reihenfolge eingefüllt werden sollen. *fill* ruft

$$\mathit{fillConts} : \mathit{obj\ list} \to \mathit{cont\ list} \to \mathit{cont\ list} \to \mathit{cont\ list}$$

auf. *fillConts* hat zwei *cont list*-Argumente. Das erste besteht aus dem nächsten zu beladenden und den bereits gefüllten Containern, das zweite besteht aus den noch verfügbaren leeren Containern. Ist diese Liste leer (*null*(*emptyConts*)), die Objektliste aber noch nicht, dann wird die Ausnahme *nextPerm* aufgerufen, "nach außen" hinter den umschließenden Aufruf von *fill* transportiert und dort durch einen Aufruf von

$$newPerm : obj\ list \to cont\ list \to cont\ list$$

abgefangen, der die nächste Permutation der Objektliste ermittelt (vgl. §12.1). Andernfalls wird *fillConts* mit dem nächsten leeren Container aufgerufen.

```
val   objSizes = map(fn(a)=>(le(a),he(a)))
val  contSizes = map(fn(c)=>(len(c),hei(c)))
val  contSize = ref(0,0)

fun  sum(pairs) = fold(op +)(map(op * )(pairs))(0)

exception conts_too_small and nextPerm and Full and Restart of int*int

fun  fill(objs)(conts) = if sum(objSizes(objs)) > sum(contSizes(conts))
                         then raise conts_too_small
                         else fillConts(objs)[hd(conts)](tl(conts))
                              handle nextPerm => newPerm(objs)(conts)

and  fillConts(nil)(conts)(emptyConts) = rev(conts)@emptyConts
|    fillConts(objs)(c::conts)(emptyConts)
       = let val (objs,c) = (contSize:= (len(c),hei(c)); addObjs(objs,c)(1,1))
         in if objs = nil then rev(conts)@[c]@emptyConts
            else if null(emptyConts) then raise nextPerm
                 else fillConts(objs)(hd(emptyConts)::c::conts)(tl(emptyConts))
         end

and  newPerm(objs)(conts) = let val objs = nextperm(r)(objs)
                            in fillConts(objs)[hd(conts)](tl(conts))
                               handle nextPerm => newPerm(objs)(conts) end
```

fillConts ruft

$$addObjs : obj\ list \times cont \to int \times int \to obj\ list \times cont$$

mit dem nächsten zu beladenden Container c auf, dessen Abmessungen vorher in der globalen dynamischen Variablen *contSize* gespeichert werden, damit weiter unten definierte Hilfsfunktionen darauf zugreifen können, ohne daß der Wert von *contSize* als **transienter** Parameter bis dorthin übergeben werden muß.

Einerseits dient es der Übersicht, wenn eine Funktion nicht zu viele Parameter hat, die sie selbst gar nicht verändert, sondern nur weiterreicht. Andererseits führt die Verwendung dynamischer Variablen leicht zu Entwurfsfehlern, weil man nicht davon ausgehen kann, daß jeder Aufruf einer Funktion, die auf ein dynamisches Objekt zugreift, dieses im selben Zustand vorfindet. Für einen formalen Korrektheitsbeweis müssen dynamische Objekte sowieso als versteckte Parameter der zugreifenden Funktionen betrachtet werden.

$addObjs(objs, c)(lwb)$ sucht für die Objekte von *objs* freie Plätze im Container c ab der Koordinate *lwb* (lower bound), weil kleinere Koordinaten bereits belegt sind. *addObjs* ruft

$$addObj : obj \times cont \to int \times int \to cont$$

mit dem ersten Objekt a von $objs$ auf. Ist der Versuch, einen Platz für a in c ab der Position lwb zu finden, erfolgreich, dann liefert $addObj(a, c)(lwb)$ die Erweiterung von c um a. Andernfalls gibt es zwei mögliche Gründe, warum der Versuch gescheitert ist:

- $addObj(a, c)(lwb)$ erreicht die Ausnahme $Restart(new\_lwb)$, wenn a zwar nicht an die Position lwb gelegt werden kann, aber möglicherweise ab der Position new_lwb ein - mit allen Restriktionen - verträglicher Platz für a existiert.
- $addObj(a, c)(lwb)$ erreicht die exception $Full$, wenn a - unabhängig von allen weiteren Restriktionen - in c nicht mehr hineinpaßt.

Im ersten Fall wird mit $tryAgain$ ab der Position new_lwb ein neuer Versuch gestartet. Im zweiten Fall endet auch der Aufruf von $addObjs$ und es werden die alten Werte sowohl von $objs$ als auch von c an die Aufrufstelle von $addObjs$ in $fillConts$ zurückgegeben.

```
and   addObjs(nil,c) _ = (nil,c)
|     addObjs(a::objs,c)(lwb)
        = let val (l,_) = !contSize
              val lwb = free_lwb(c)(lwb)(l)
              val (c,full) = (addObj(a,c)(lwb) handle Restart(new_lwb)
                                                   => tryAgain(a,c)(new_lwb),
                              false)
                             handle Full => (c,true)
          in if full then (a::objs,c) else addObjs(objs,c)(lwb) end

and  free_lwb(c)(i,j)(l) = if free(c,i,j) then (i,j)
                           else if i < l then free_lwb(c)(i+1,j)(l)
                                         else free_lwb(c)(1,j+1)(l)

and  free(c,i,j) = (get(c,i,j); false) handle undef => true

and  tryAgain(a,c)(lwb) = addObj(a,c)(lwb)
                          handle Restart(new_lwb) => tryAgain(a,c)(new_lwb)
```

Über Parameter von Ausnahmen (wie hier im Fall von *Restart*) können Werte von der Aufrufstelle der Ausnahme innerhalb eines Ausdrucks nach außen transportiert und dort verwendet werden (siehe auch §7.2). Das ließe sich auch mit dynamischen Variablen erreichen. Oben haben wir gesehen, daß man dynamische Variablen u.a. dazu verwendet, um Funktionsparameter zu vermeiden, die Werte lediglich nach innen weiterreichen. Mit parametrisierten Ausnahmen lassen sich umgekehrt dynamische Variablen vermeiden, die dazu dienen, Werte nach außen zu reichen. Eine disziplinierte Ausnahmebehandlung wie in ML ist auch die weitaus übersichtlichere Alternative zu den bei manchen Programmierern leider immer noch beliebten **Sprunganweisungen**. Sprunganweisungen verletzen die **Kompositionalität** eines Programms, d.i. die Bedingung, aus der Bedeutung seiner Teile auf seine Gesamtbedeutung schließen zu können, oft so stark, daß ein halbwegs formaler Korrektheitsbeweis praktisch unmöglich ist.

Die Ausnahme $Full$ wird direkt von $addObj$ aufgerufen:

```
and  addObj(a,c)(i,j) = let val (x,y) = (i+le(a)-1,j+he(a)-1)
                            val (l,h) = !contSize
                        in if y > h then raise Full
```

```
            else if x > l then addObj(a,c)(1,j+1)
            else if free_base(c,i,x,j) andalso
                    is_adjacent_to(c,a,i,j,y) andalso
                    is_on_top_of(c,a,i,x,j) andalso
                    is_centered(c,a,i,j) andalso
                    is_l_aligned(c,a,i,j) andalso
                    is_r_aligned(c,a,x,j)
                 then Add(a,i,j,c) else addObj(a,c)(i+1,j)
        end
```

Die Ausnahme *Restart* hingegen wird erst bei der Prüfung von Bedingungen an die Plazierung von a erreicht. Die Bedingungen lauten im einzelnen: Würde a beginnend bei (i, j) plaziert, dann gilt

- $free\_base(c, i, x, j)$, falls die j-te Ebene von c über die gesamte Länge von a unbesetzt ist;
- $is\_adjacent\_to(c, a, i, j, y)$, falls a mit seinen linken und rechten Nachbarobjekten bzgl. einer vorgegebenen Relation $adjacent\_to$ verträglich ist;
- $is\_on\_top\_of(c, a, i, x, j)$, falls a mit den direkt darunterliegenden Objekten bzgl. einer vorgegebenen Relation on_top_of verträglich ist;
- $is\_centered(c, a, i, j)$, falls das Bit $centered(a)$ gesetzt ist oder a zentriert auf einem Objekt liegt[16];
- $is\_l\_aligned(c, a, i, j)$, falls das Bit $l\_aligned(a)$ gesetzt ist oder a linksbündig auf einem Objekt liegt,
- $is\_r\_aligned(c, a, x, j)$, falls das Bit $r\_aligned(a)$ gesetzt ist oder a rechtsbündig auf einem Objekt liegt.

Die Codierung dieser Prädikate als Boolesche ML-Funktionen findet man in §15.2.

12.5 Modularisierung

Die bisher vorgestellten Teile des Regalprogramms werden mithilfe eines Funktors *Container* strukturiert. *Container* hat die Parameterstruktur *Object*, die zweimal aktualisiert wird (*Obj* bzw. $Shelf$), also zwei Bild-Strukturen: *objCont* und $shelfCont$, liefert (s. Kap. 11). Fig. 21 zeigt die Gesamtstruktur. Bemerkenswert ist hier besonders der "hierarchische" Zugriff von $Shelf$ auf *objCont* und die "Flexibilisierung" von *Container*-Funktionen (hier: $is\_centered$) durch die unterschiedliche Aktualisierung von Funktionskomponenten (hier: *centered*), die dem Parameter (hier: *Object*) angehören.

Der Permutationsalgorithmus aus §12.1, der Parser aus §12.2 sowie weitere Hilfsfunktionen werden zunächst in einer Struktur *Aux* (*auxiliaries*) zusammengefaßt (siehe §15.2). Wie in §11.1 beschrieben, erfolgen Zugriffe auf in einer Struktur definierte Typen, Werte, usw. über den Strukturnamen, z.B.: $Aux.sum$.

Der Datentyp *cont* (s. §12.2) und die Funktionen des Ladealgorithmus (§12.4) hängen ab von mehreren Parametern, die sich alle auf die in Containern plazierten Objekte beziehen. Deshalb bietet sich es sich an, *cont* und $fill$ in einen Funktor mit einem Strukturparameter *Obj* der Signatur *Object* "einzukapseln", die genau jene Parameter enthält:

```
signature Object = sig eqtype obj
                       val le : obj -> int
```

[16] Unter dieser Bedingung würden Objekte linksbündig übereinandergestapelt, wenn sich die Länge eines Objektes von der des darüberliegenden um höchstens eine Einheit unterscheidet. Um das zu vermeiden, wird a um eine Einheit nach rechts verschoben, falls j ungerade ist.

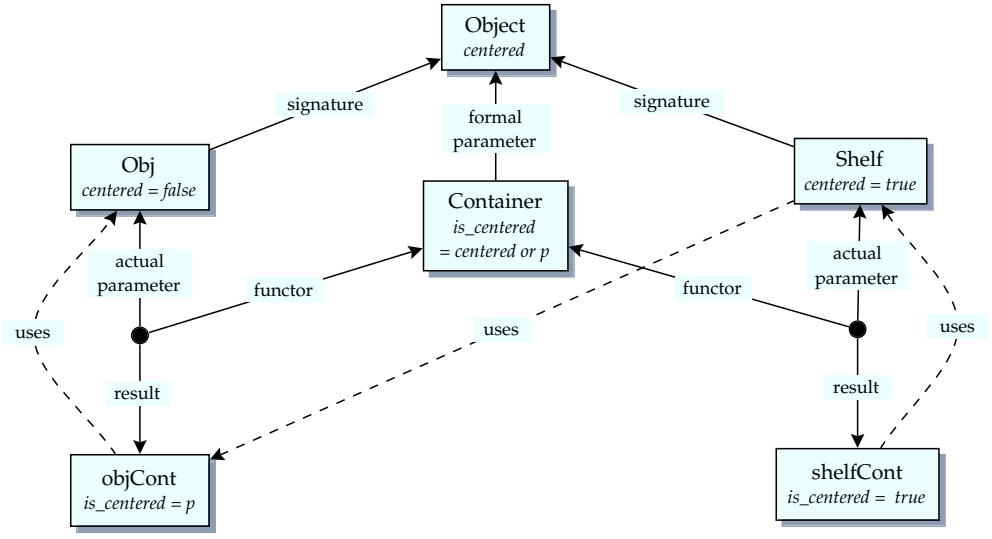


Figure 21. *Module des Regalprogramms*

```
        val he : obj -> int
        val leq : obj * obj -> bool
        val l_aligned : obj -> bool
        val r_aligned : obj -> bool
        val centered : obj -> bool
        val adjacent_to : obj * obj -> bool
        val on_top_of : obj * obj -> bool
    end

functor Container(structure Obj : Object)
        = struct open   Obj
                 abstype cont ... with  ...   end
                 fun    fill ...
                 ...
          end    (* Container *)
```

Beide Beladungen, sowohl die der Container mit Objekten als auch die Stapelung aller beladenen Container (*shelves*) in einem großen Container, erfolgen nach demselben Ladealgorithmus $fill$, dem Kern des Funktors $Container$. Deshalb ist es naheliegend, beide Beladungsvorgänge als Aufrufe von $fill$ in geeigneten Aufrufen (Instanzen) von $Container$ zu implementieren. Die erste Instanz ist die Struktur $objCont$, d.i. die Anwendung von $Container$ auf irgendeine Aktualisierung des Parameters Obj (s.o.). Die zweite Instanz ist die Struktur $shelfCont$, d.i. die Anwendung von $Container$ auf die $Shelf$ genannte Aktualisierung von Obj. Man beachte die Aufrufhierarchie: Die Objekte von $Shelf$ sind Zeiger auf die Container von $objCont$ sind.[17]

```
structure Obj = struct  type obj = int*int*int
                        fun le(l,_,_) = l
                        fun he(_,h,_) = h
                        fun leq((l,h,_),(l',h',_)) = Aux.lex([l,h],[l',h'])
                        fun l_aligned _ = true
```

[17] Die Container selbst können nicht Objekte von $Shelf$ sein, weil $objCont.cont$ ein *abstype* ist und als solcher kein *eqtype*, wie von der Signatur *Object* gefordert. Das macht auch Sinn, weil die Gleichheit zweier Zeiger (vom Typ $objCont.cont\ ref$) i.a. schneller entschieden werden kann als die Gleichheit (der Beladungen) zweier Container (vom Typ $objCont.cont$).

```
                        fun r_aligned _ = true
                        fun centered _ = false
                        fun adjacent_to((_,_,t:int),(_,_,t'))
                                = t > 1 andalso t' > 1
                        fun on_top_of((l:int,h,t),(_,_,t'))
                                = l >= h andalso t > 1
                end

structure objCont = Container(structure Obj = Obj)

structure Shelf = struct type obj = objCont.cont ref
                         val le = objCont.len o !
                         val he = objCont.hei o !
                         fun leq(sh,sh') = Aux.lex([le(sh),he(sh)],[le(sh'),he(sh')])
                         fun l_aligned _ = false
                         fun r_aligned _ = false
                         fun centered _ = true
                         fun adjacent_to _ = true
                         fun on_top_of _ = true
                  end

structure shelfCont = Container(structure Obj = Shelf)
```

12.6 Compiling

Das Modularisierungsschema von Fig. 21 wird wiederverwendet, um die Menge der Daten und Funktionen zur zweidimensionalen Darstellung von *objCont.cont*- bzw. *shelfCont.cont*-Objekten strukturieren (s. Fig. 22). Analog zu den Bildern *objCont* und *shelfCont* des Funktors *Container* sind *drawObjs* und *drawShelves* Bilder des Funktors *Display*, in dem entsprechender PostScript-Code erzeugt wird.[18] Analog zum "hierarchischen" Zugriff von *Shelf* auf *objCont* in Fig. 21 gibt es in Fig. 22 einen Zugriff von $shelfPict$ auf $drawObjs$. Ausserdem wird die *Display*-Funktion *objCode* durch unterschiedliche Aktualisierungen der Parameterfunktionen *interior* und *border* "flexibilisiert".

```
signature Picture = sig type obj and cont
                        val le : obj -> int
                        val he : obj -> int
                        val len : cont -> int
                        val hei : cont -> int
                        val pos : cont * obj -> int * int
                        val Objs : cont -> obj list
                        val space : int
                        val interior : int * int -> obj -> string
                        val border : int * int -> obj -> string
                  end

functor Display(structure Pict : Picture)
        = struct open Pict
```

[18] Vgl. §9.1.3.

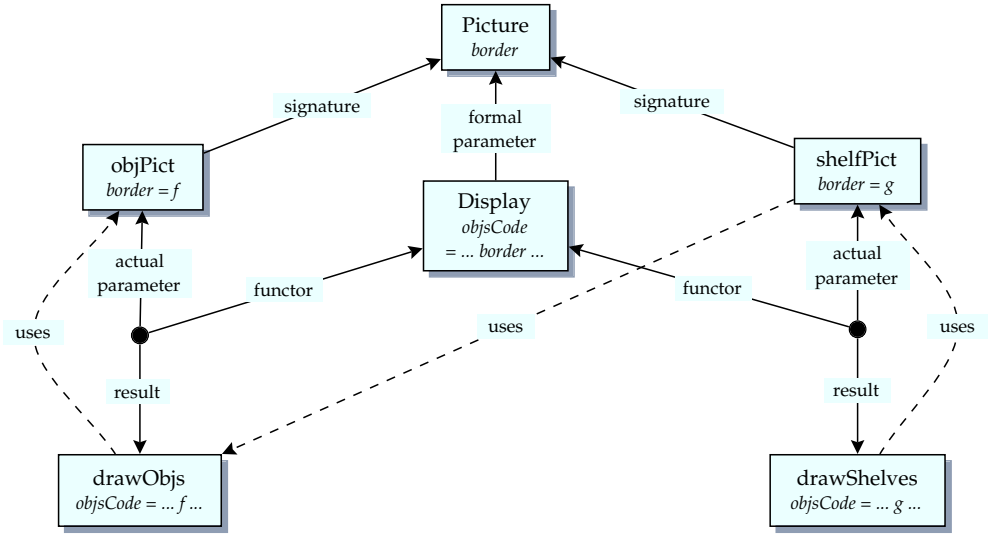


Figure 22. *Weitere Module des Regalprogramms (vgl. Fig. 21)*

```
            fun code(c,n)
                 = let val file = open_out("CONT"^makestring(n:int)^".eps")
                       val sp = makestring(space)^" "
                       val (l,h) = (len(c)*space,hei(c)*space)
                       val profile = ... see Section 15.2 ...
                       val code = profile^objsCode(c)(Objs(c))^"showpage"
                   in output(file,code); close_out(file) end

            and objsCode(c)(nil) = ""
            |   objsCode(c)(a::objs)
                 = let val p = pos(a,c)
                   in interior(p)(a)^border(p)(a)^objsCode(c)(objs) end

            fun shift(x,y)(c)
                 = let val str1 = makestring(x*space)^" "
                       val str2 = makestring(y*space)^" translate\n"
                   in str1^str2^objsCode(c)(Objs(c))^"-"^str1^"-"^str2 end

        end    (* Display *)

structure objPict = struct open objCont
                         val space = 15
                         val interior = Aux.objfill(space)
                         val border = Aux.objstroke(space)
                    end

structure drawObjs = Display(structure Pict = objPict)

structure shelfPict = struct open shelfCont
                          val space = 15
                          fun interior(x,y) = drawObjs.shift(x-1,y-1) o !
```

```
                        fun border(p)(sh)
                             = Aux.shelfstroke(space)(p)(le(sh),he(sh))
                    end

structure drawShelves = Display(structure Pict = shelfPict)
```

Die Aktualisierungen von *interior* und *border* geben an, wie die Innenfläche bzw. der Rand von Objekten "ausgemalt" werden soll. Das zu einem elementaren Objekt gehörige Rechteck wird abhängig vom Typ t des Objektes gefärbt und im Containerrechteck positioniert, und zwar relativ zu einem Koordinatensystem mit Ursprung $(0, 0)$. Später bewirkt ein Aufruf von *translate* in $shelfPict$ die Verschiebung jenes Containerrechtecks und seines Inhalts (!) zu seiner Position innerhalb des großen Containers $CONT$, in dem alle shelves (= beladenen Container) gestapelt werden.

12.7 Structures in concert

Das Hauptprogramm stellt dynamische Variablen für $CONT$ sowie für eine Objekt- und eine Shelf-Liste zur Verfügung:

```
val   CONT = ref(0,0)
val  Objects = ref(nil:Obj.obj list)
val  Shelves = ref(nil:Shelf.obj list)
```

Die Prozedur *init* liest eine Objekt- und eine Container-Liste ein und initialisiert *Objects* und *Shelves*:

```
fun  init(infile:int)
      = let val chars = Aux.read(infile)
            val (objects,conts) = Aux.parseAll(chars)
        in Objects:= Aux.firstperm(Obj.leq)(objs);
           Shelves:= Aux.firstperm(Shelf.leq)(map(ref o objCont.New)(conts))
        end
```

Die Prozeduren *nexts* und *nexto* permutieren die Listen !*Shelves* bzw. !*Objects*:

```
fun   nexto(n) = Objects:= Aux.nthperm(n)(Obj.r)(!Objects)
fun  nexts(n) = Shelves:= Aux.nthperm(n)(Shelf.r)(!Shelves)
```

Die Prozedur *cont* initialisiert den großen Container $CONT$:

```
fun   cont(l)(h) = CONT:= (l,h)
```

Die Prozedur $fills$ plaziert die Container von !*Shelves* in !$CONT$ in der durch die Liste !*Shelves* gegebenen Reihenfolge und gibt die entsprechende Beladung von !$CONT$ aus:

```
fun   fills(outfile:int)
       = let val shelves = map(ref o objCont.frame o !)(!Shelves)
             val [CONT] = shelfCont.fill(shelves)[shelfCont.New(!CONT)]
         in drawShelves.code(CONT,outfile) end
```

Man beachte die "Rahmung" der Container von !*Shelves* mit *objCont.frame* vor ihrer Plazierung in !*CONT*.

Die Prozedur *fillo* füllt die Objekte von !*Objects* in die Container von !*Shelves* in der durch die Liste !*Objects* gegebenen Reihenfolge. Anschließend werden die Liste *shelves* der beladenen Container im großen Container !*CONT* gestapelt und die entsprechende Beladung von !*CONT* ausgegeben:

```
fun   fillo(outfile:int)
       = let val shelves = (objCont.fill(!Objects) o map(!))(!Shelves)
             val shelves = (Shelves:= map(ref)(shelves);
                            map(ref o objCont.frame)(shelves))
             val [CONT] = shelfCont.fill(shelves)[shelfCont.New(!CONT)]
         in drawShelves.code(CONT,outfile) end
```

Die Prozedur *mt*() entleert die Container von !*Shelves*, ohne sie aus !*CONT* zu entfernen:

```
fun   mt() = Shelves:= map(ref o objCont.makeEmpty o !)(!Shelves)
```

Steht im File *CONT*1 die Zeichenfolge:

```
 (3,6,ry)(5,1,rg)(2,2,ct)(5,3,rb)(4,2,av)(1,1,rg)(1,4,rg)
(2,2,ct)(3,8,rr)(3,3,tg)(3,5,rr)(2,7,rb)
conts (7,8),(6,14),(7,9),(6,10)
```

dann steht nach Ausführung der Befehlsfolge *init 1; nexts 5; cont 25 21; fills 1;* das Regal von Fig. 23 im File *CONT*1.*eps*. Gibt man dann *nexto 1200; fillo 1;* ein, dann enthält dieser File die Fig. 20. Mit *mt(); fills 3;* wird dort Fig. 23 wiederhergestellt.

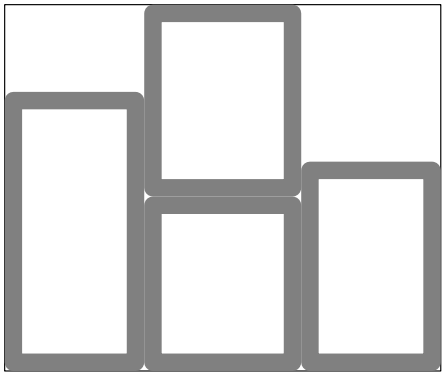


Figure 23.

Die Prozeduren dieses Abschnitts bilden eine kleine Kommandosprache, deren Befehle in beliebiger Reihenfolge interaktiv aufgerufen werden können. Prozeduren erzeugen und verändern eine Menge dynamischer Variablen (hier: *CONT*, *Objects* und *Shelves*), die man zusammen als internen System-**zustand** bezeichnen könnte (siehe auch §12.4). Für diesen Teil des Programm macht in der Tat nur der imperative, zustandsbasierte Programmierstil Sinn.

13 Ströme und verzögerte Auswertung

Ströme heißen solche Listen, die im Sinne verzögerter Auswertung (s. §2.11) i.d.R. nur teilweise erzeugt werden. In ML definiert man sie wie folgt:

```
infix &
datatype 'a stream = Nil | & of 'a * (unit → 'a stream)
```
con Nil : 'a stream
*con & : 'a * (unit → 'a stream) → 'a stream*

```
fun ones() = 1&ones
fun nats(n)() = n&nats(n+1)
fun alternates() = 0&(fn()=>1&alternates)
```
val ones = fn : unit → int stream
val nats = fn : int → unit → int stream
val alternates = fn : unit → int stream

Ein Strom $a\&s$ hat a als erstes Element. Der "Reststrom" s ist aber nicht, wie man erwarten würde, vom Typ *'a stream*, sondern eine Funktion vom Typ *unit → 'a stream*. So wird die Auswertung von $a\&s$ verzögern, denn erst ein *Aufruf* von s liefert wieder ein Objekt vom Typ *'a stream* und damit das zweite Element von $a\&s$, usw.

Netzwerke werden oft mit Strömen und Funktionen auf Strömen modelliert. Z.B. simuliert die folgende Funktion einen Kanal, auf dem Elemente des Typs $'a$ transportiert werden und dabei manchmal verlorengehen. Dieses Verhalten wird mit einem *int*-Strom modelliert, der parallel zum $'a$-Strom läuft. Ein Element x des $'a$-Stromes geht genau dann verloren, wenn "zur selben Zeit" eine 0 im *int*-Strom auftritt.

```
fun channel(nil,_) = nil
|   channel(x::L,s) = case s() of 0&s => channel(L,s)
                                |  i&s => x::channel(L,s)
```
*val channel = fn : 'a list * (unit → int stream) → 'a list*

Analog zu den in §4.1 eingeführten Listenoperationen lassen sich entsprechende Stromoperationen definieren. Z.B.:

```
fun head(s) = let val x&_ = s() in x end
fun tail(s) = let val _&s = s() in s end
```
val head = fn : (unit → 'a stream) → 'a
val tail = fn : (unit → 'a stream) → unit → 'a stream

Die ersten n Elemente eines Stroms
```
fun first_n(s,0) = nil
|   first_n(s,n) = case s() of Nil => nil
                             |  x&s => x::first_n(s,n-1)
```
*val first_ n = fn : (unit → 'a stream) * int → 'a list*

Anfügen einer (endlichen) Liste an einen Strom
```
infix $
fun nil$s = s()
|   (x::L)$s = x&(fn()=>L$s)
```
*val $ = fn : 'a list * (unit → 'a stream) → 'a stream*

Stromversion von *mapconc* (s. §4.1)
```
fun Mapconc(f)(s) = case s() of Nil => Nil
                              |  x&s => f(x)$(fn()=>Mapconc(f)(s))
```
val Mapconc = fn : ('a → 'b list) → (unit → 'a stream) → 'b stream

Mischen zweier Ströme
```
fun merge(s,s') = case (s(),s'()) of
                      (Nil,_) => s'()
```

```
    |   (x&s,Nil) => x&s
    |   (x&s,y&s') => x&(fn()=>y&(fn()=>merge(s,s')))
```

*val merge = fn : (unit → 'a stream) * (unit → 'a stream) → 'a stream*

merge verarbeitet abwechselnd Elemente der Ströme *s* und *s'*. Diese Funktion könnte z.B. einen Netzwerkknoten mit zwei Eingabekanälen und einem Ausgabekanal beschreiben. *merge* behandelt die beiden Eingabekanäle *fair*: Jede eingehende "Nachricht" wird nach endlicher Wartezeit weitertransportiert.

Beispiel 13.1 Die Menge der Permutationen einer endlichen Liste ist zwar endlich, aber i.a. sehr lang: Eine n-elementige Liste hat n! Permutationen. In §12.1 haben wir gesehen, daß der naive rekursive Algorithmus *allperms* die Bildung jeder dieser Permutationen zum gleichen Zeitpunkt beendet, so daß dieser Algorithmus *kombinatorischen* Aufwand hat. alternative Algorithmus *nthperm* (s. §12.1) zählt demgegenüber iterativ nur die bzgl. einer lexikographischen Listenordnung ersten Permutationen einer Liste auf. Sollen nicht alle n! Permutationen erzeugt werden, dann ist *nthperm allperms* vorzuziehen. Trotz verzögerter Auswertung kann auch eine Stromversion von *allperms* den Aufwand dieses Algorithmus' nicht verringern:

```
fun  Allperms[x]() = [x]&(fn()=>Nil)
|    Allperms(x::L)() = Mapconc(insert(x))(Allperms(L))      s. 12.1 u. 13.1
```

val Allperms = fn : 'a list → unit → 'a list stream

```
val L = first_n(Allperms[1,2,3],10)
```
val L = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]] : int list list

Beispiel 13.2 Die Liste der **Zerlegungen (Partitionen)** einer n-elementigen Liste ist bei großem n ebenfalls sehr lang. Ihre Mächtigkeit $c(n)$ erfüllt nämlich die Gleichungen:

$$\begin{aligned} c(0) &= 1 \\ c(n+1) &= \sum_{i=0}^{n} \binom{n}{i} * c(i) \end{aligned}$$

und der *Binomialkoeffizient* $\binom{n}{i}$ ist ein Quotient von Fakultäten: $n!/(n-i)! * i!$. Ein dem Definitionsschema von *allperms* ähnlicher Algorithmus zur Aufzählung aller $c(n)$ Partitionen lautet wie folgt:

```
fun  parts(nil) = [nil]
|    parts[x] = [[[x]]]
|    parts(x::L) = mapconc(glue(x))(parts(L))

and  glue(x)(L) = let fun shift(L,x,nil) = [[x]::L]
                      |   shift(L,x,r::R) = ((x::r)::L@R)::shift(r::L,x,R)
                  in shift(nil,x,L) end
```

val parts = fn : 'a list → 'a list list

```
val L = parts[1,2,3]
```
val L = [[[1,2,3]],[[1],[2,3]],[[1,2],[3]],[[1,3],[2]],[[1],[3],[2]]] : int list list list

Wieder wird die Ergebnisliste nicht Element für Element (hier: Partition für Partition) aufgebaut. Man muß warten, bis alle Permutationen erzeugt sind, bevor man sich die erste ansehen kann. Die Stromversion von *parts*:

```
fun  Parts(nil)() = nil&(fn()=>Nil)
|    Parts[x]() = [[x]]&(fn()=>Nil)
|    Parts(x::L)() = Mapconc(glue(x))(Parts(L))
```

val Parts = fn : 'a list → unit → 'a list list stream

schafft da keine Abhilfe. ✌

Das Programmieren mit Strömen unterscheidet sich kaum vom Programmieren mit verketteten Listen (s. §10.3). Vergleichen wir einmal beide Datentypdefinitionen:

```
infix &
datatype 'a stream = Nil | & of 'a * (unit → 'a stream)
datatype 'a elem = Nil | & of 'a * 'a elem ref
```

Dem Übergang vom Strom s zur λ-Abstraktion `fn()=>s` (vgl. §2.10) entspricht offenbar die Einführung eines Zeigers: $e \rightsquigarrow ref(e)$. Der Aufruf `(fn()=>s)()` der λ-Abstraktion wird zur Dereferenzierung: $!ref(e) = e$. Sowohl λ-Abstraktionen als auch Referenzen sind programmiersprachliche Mittel zur Trennung zwischen Ausdrücken einerseits und deren Werten andererseits. Es würde sich bestimmt lohnen, den Zusammenhang zwischen λ-Abstraktionen und verketteten Datenstrukturen genauer zu untersuchen. Möglicherweise entdeckt man dabei allgemeine Transformationsschemata ähnlich den in Kap. 5 behandelten, die es erlauben würden, auch dynamische Datenstrukturen und Algorithmen zunächst rein funktional zu entwerfen und dann mehr oder weniger automatisch in "konkrete" Zeigertypen und -manipulationen zu übersetzen.

14 Logisches Programmieren

14.1 Auswerten versus Lösen

Zum einen sind die *first class citizens* einer logischen Programmiersprache wie *Prolog* nicht Funktionen, sondern Relationen. Zum anderen – und das ist der wesentliche Unterschied zu funktionalen Sprachen – besteht die Ausführung eines logischen Programms nicht in der **Auswertung** eines Terms (= funktionalen Ausdrucks), sondern in der **Lösung** von Gleichungen, oder allgemeiner: in der Beantwortung von Anfragen. Ein logisches Programm repräsentiert die Daten-, Informations-, Wissensbasis[19], relativ zu der die Anfragen beantwortet werden sollen. Logische Programme und Anfragen sind prädikatenlogische Formeln. Die Frage *Hat die Funktion* $f(x) = x^2$ *eine Nullstelle?* wird z.B. durch die Formel $\exists x : x^2 = 0$ dargestellt. Der Interpreter einer logischen Programmiersprache betrachtet eine solche Formel als Eingabe und transformiert sie in eine oder mehrere Lösungen ihrer existenzquantifizierten Variablen, z.B. $x = 0$.

Der Interpreter legt nur die allgemeine Strategie fest, nach der beliebige Programmformeln auf Anfragen angewendet werden, damit Lösungen berechnet werden können. Erst zusammen mit dem aktuellen logischen Programm wird daraus ein auf die jeweiligen Anfragen zugeschnittener Lösungsalgorithmus. Prolog-Interpretern verwenden **Resolutionsstrategien**, die schon Anfang der 60er Jahre zur Automatisierung mathematischer Beweise entwickelt wurden.

Ein allgemeines Lösungsverfahren, das für beliebige Programme und Datentypen funktioniert, kann eigentlich nur sehr ineffizient sein. Man arbeit deshalb zur Zeit an Sprachimplementierungen, die für spezielle Datentypen (insbesondere Boolesche und arithmetische) auch spezielle Lösungsverfahren verwenden (*constraint logic programming*) oder die, wann immer möglich, die Suche nach Lösungen durch das Auswerten funktionaler Ausdrücke beschleunigen (*functional-logic programming*).

Letzteres verlangt die Erkennung *funktionaler Abhängigkeiten* in den durch logische Programme definierten Relationen. Die Funktion $f : A \to B$ definiert ihre Ein-Ausgabe-Relation EA_f als Teilmenge von $A \times B$ (s. §3.2). Dann ist die zweite Stelle von EA_f von der ersten funktional abhängig, d.h. für alle $a \in A$ gibt es genau ein $b \in B$ mit $(a, b) \in EA_f$.

[19] oder was uns noch für weitere wohlklingende Wortschöpfungen für ein und dieselbe Sache begegnen mögen

Beispiel 14.1.1 Schauen wir uns eine gegenüber 4.4.2 leicht abgewandelte[20] Version des *quicksort*-Programms an:

```
fun  quicksort[] = []
|    quicksort(x::L) = let val (low,high) = filter(x,L)
                           val L1 = sort(low)
                           val L2 = sort(high)
                           val L3 = L1@x::L2
                       in L3 end
and  filter(x,[]) = ([],[])
|    filter(x,y::L) = let val (low,high) = filter(x,L)
                      in if y <= x then (y::low,high)
                                   else (low,y::high) end
```

In §3.2 haben wir gezeigt, wie man die Definition einer Funktion F in ein bedingtes Gleichungssystem überführt, das die Ein-Ausgabe-Relation von F beschreibt. Führt man diese Transformationen auf den Definitionen von $F = quicksort$ und $F = filter$ durch und ersetzt man anschließend jede Gleichung der Form $F(x) = z$ durch die Formel $EA_F(x, z)$, dann erhält man:

```
EA_quicksort([],[])
EA_filter(x,L,low,high) /\ EA_quicksort(low,L1) /\ EA_quicksort(high,L2)
                        /\ L1@x::L2 = L3  =>  EA_quicksort(x::L,L3)
EA_filter(x,[],[],[])
     y <= x /\ EA_filter(x,L,low,high)  =>  EA_filter(x,y::L,y::low,high)
not(y <= x) /\ EA_filter(x,L,low,high)  =>  EA_filter(x,y::L,low,y::high)
```

Mit ein paar syntaktischen Änderungen werden daraus Prolog-Programme für die Ein-Ausgabe-Relationen von *quicksort* und $filter$:

```
EA_quicksort([],[]).
EA_quicksort([x|L],L3) :- EA_filter(x,L,low,high), EA_quicksort(low,L1),
                          EA_quicksort(high,L2), EA_append(L1,[x|L2],L3).
EA_filter(x,[],[],[]).
EA_filter(x,[y|L],[y|low],high) :- y <= x, EA_filter(x,L,low,high).
EA_filter(x,[y|L],low,[y|high]) :- not(y <= x), EA_filter(x,L,low,high).
```

Was hat man damit gewonnen? Um eine Anfrage wie $\exists L : EA_{quicksort}([2, 5, 3, 4, 1], L)$ zu beantworten, wäre sicher das funktionale Programm vorzuziehen. Prolog beantwortet aber auch "umgekehrte" Anfragen wie $\exists L : EA_{quicksort}(L, [1, 2, 3, 4, 5])$! ☙

Die zuletzt genannte Anfrage läßt sich mit einem funktionalen Programm nicht beantworten, weil sie mehrere Lösungen hat. Deshalb sind logische Programme gerade dort geeignet, wo anstelle einer funktionalen Abhängigkeit die nichtdeterministische Auswahl aus mehreren korrekten Ergebnissen verlangt wird. So wählt z.B. das Prolog-Programm

```
select([x|L],x).
select([x|L],y) :- select(L,y).
```

[20] und auf $r =_{def} \leq$ beschränkte

ein Element aus einer gegebenen Liste aus. Die Anfrage $\exists x : select([2,4,3,7,9], x)$ hat *fünf* Lösungen: $X = 2$, $X = 4$, $X = 3$, $X = 7$ und $X = 9$. Man erhält sie alle, wenn man die Anfrage fünfmal wiederholt. Logische Programme bieten sich also insbesondere zur Lösung von Aufzählungsproblemen an. Die Funktion *allperms*, die alle Permuationen einer Liste aufzählt (s. 12.1), läßt sich leicht in ein Prolog-Programm umwandeln, das Anfragen an die Relation $perm =_{def} \{(L, P) \mid P \text{ ist eine Permutation von } L\}$ beantwortet:

```
perm([],[]).
perm([x|L],P) :- perm(L,Q), insert(x,Q,P).
insert(x,Q,[x|Q]).
insert(x,[y|Q],[y|P]) :- insert(x,Q,P).
```

Die Anfrage $\exists P : perm([1,2,3], P)$, sechsmal wiederholt, liefert alle sechs Permutationen von $[1,2,3]$. Das Programm für *perm* ist kürzer als die Definition von *allperms*, weil letztere eine Strategie zur Aufzählung aller Permutationen enthält, während *perm* auf eine allgemeine, in das Prolog-System eingebaute Strategie zurückgreift. Wegen ihrer Allgemeinheit ist diese aber nicht die für jedes Aufzählungsproblem beste (s.o.).

Hier noch eine analoge Übersetzung der Funktion *parts* zur Aufzählung von Listenpartitionen (s. Bsp. 13.2) in ein Prolog-Programm, das Anfragen an die Relation $part =_{def} \{(L, P) \mid P \text{ ist eine Partition von } L\}$ beantwortet:

```
part([x],[[x]]).
part([x,y|L],P) :- part([Y|L],Q), glue(x,Q,P).
glue(x,[],[[x]]).
glue(x,[L|Q],[[x|L]|Q]).
glue(x,[L|Q],[L|P]) :- glue(x,Q,P).
```

Die Anfrage $\exists P : part([1,2,3], P)$, fünfmal wiederholt, liefert alle fünf Partitionen von $[1,2,3]$.

14.2 Rekursive Wertdefinitionen

Eine Wertdefinition `val a = e` darf nur dann rekursiv sein, d.h. e darf a enthalten, wenn a einen Funktionstyp hat. Dann müssen wir aber `val rec a = e` schreiben (s. §2.10). Warum diese Einschränkung? Weil nur in diesen Fällen der Wert von a durch Auswertung des Ausdrucks e berechnet werden kann. Um allgemeinere Wertdefinitionen auszuführen, bräuchten wir eine logische Programmiersprache! Allgemein kann `val a = e` nämlich nur bedeuten, daß der Wert von a eine Lösung der Gleichung $a = e$ in a ist. Es gibt aber Fälle, in denen man rekursive Wertdefinitionen in λ-Abstraktionen überführen und so auch in funktionalen Programmen verwenden kann.

Beispiel Ein binärer Baum t (s. Kap. 8) soll durch einen Baum t' ersetzt werden, der in allen seinen Blättern das Minimum der Blatteinträge von t enthält. Dazu könnte man die folgenden Funktionen *minE* und *replace* benutzen, die das Minimum der Blatteinträge eines Baumes berechnet bzw. einen gegebenen Eintrag in alle Blätter eines Baumes schreibt:

```
fun  minE(T2(mt,x,mt)) = x
|    minE(T2(t1,x,t2)) = min(minE(t1),minE(t2))
fun  replace(T2(mt,x,mt),z) = leaf(z)
|    replace(T2(t1,x,t2),z) = T2(replace(t1),x,replace(t2))
fun  rep_by_min(t) = replace(t,minE(t))
```

Den doppelten Durchlauf von t könnte man dadurch zu vermeiden versuchen, daß man *replace* und *minE* parallel aufruft, was zu folgender Definition von *rep_ by_ min* mit einer lokalen rekursiven Wertdefinition führt:

```
fun  rep_by_min(t) = let val (t',z) = (replace(t,z),minE(t)) in t' end
```

Schauen wir uns die Schema dieser Definition an:

```
fun  f(x) = let val (y,z) = (g(x,z),h(x)) in y end
```

Um die rekursive Wertdefinition zu eliminieren, machen wir den Teil ihrer rechten Seite, der den Wert von $f(x)$ liefert, zu einer λ-Abstraktion in der "rekursiven Variable" z. Anstelle der rekursiven Wertdefinition verwenden wir also die Funktion gh mit $gh(x) =_{def} (\lambda z.g(z,x), h(x))$ in der Defintion von f:

```
fun  f(x) = let val (F,z) = gh(x) in F(z) end
```

Beide Definitionen von f sind tatsächlich äquivalent:[21]

```
        let val (F,z) = gh(x) in F(z) end
=       let val (F,z) = (#1(gh(x)), #2(gh(x))) in F(z) end
=       let val (F,z) = (#1(gh(x)), #2(gh(x))) in #1(gh(x))(z) end
=       let val (y,z) = (#1(gh(x))(z), #2(gh(x))) in #1(gh(x))(z) end
=       let val (y,z) = (#1(gh(x))(z), #2(gh(x))) in y end
=       let val (y,z) = (g(x,z), h(x)) in y end
```

Um eine rekursive Definition von gh abzuleiten, betrachten wir die Schemata der Definitionen von $g = replace$ bzw. $h = minE$:

```
fun  g(T2(mt,x,mt),z) = leaf(z)
|    g(T2(t1,x,t2),z) = T2(g(t1),x,g(t2))
fun  h(T2(mt,x,mt)) = x
|    h(T2(t1,x,t2)) = min(h(t1),h(t2))
```

Daraus ergibt sich zunächst eine rekursive Definition für die Funktion (g,h) mit $(g,h)(x,z) =_{def} (g(x,z), h(x))$:

```
fun  (g,h)(T2(mt,x,mt),z) = (leaf(z), x)
|    (g,h)(T2(t1,x,t2),z) = let val (t1',z1) = (g,h)(t1,z)
                                val (t2',z2) = (g,h)(t2,z)
                            in (T2(t1',x,t2'), min(z1,z2)) end
```

Wegen $(g,h)(x,z) = (\#1(gh(x))(z), \#2(gh(x)))$ legt diese Definition von (g,h) folgende von gh nahe:

```
fun  gh(T2(mt,x,mt)) = (fn(z)=>leaf(z), x)
|    gh(T2(t1,x,t2)) = let val (F1,z1) = gh(t1)
                           val (F2,z2) = gh(t2)
                       in (fn(z)=>T2(F1(z),x,F2(z)), min(z1,z2)) end
```

Die Äquivalenz dieser Definitionen von (g,h) bzw. gh zu ihren ursprünglichen läßt sich leicht durch Induktion über die Größe ihres Baumargumentes zeigen.

Aufgaben

- Führen Sie den Äquivalenzbeweis aus!

[21] $\#i(tup)$ liefert die i-te Komponente des Tupels tup.

- Übersetzen Sie das Programm für *gh* in eine Version, die auf Bäumen mit Knoten des Typs *int binode* arbeitet (siehe §10.3.1), und implementieren Sie dabei – in Anlehnung an die Bemerkung am Ende von Kap. 13 – die λ-Abstraktionen als Zeigersetzungen!

15 Anhang: code listings

15.1 drawTree

Es folgt das in Kapitel 9 vorgestellte Programm zur Übersetzung linear-notierter Bäume in PostScript-Code.

```
infix   %
datatype 'a tree = % of 'a * ('a tree list)

fun     leaf(a) = a%nil
fun     height(a%tL) = fold(max)(map(height)(tL))(0)+1

fun     last[x] = x
|       last(x::L) = last(L)

(* parse terms *)

exception no_term

fun     parseTerm(" "::chars,symbol) = parseTerm(chars,symbol)
|       parseTerm("("::chars,symbol)
          = let val (tL,chars) = parseTerms(chars,nil)
            in (symbol%tL,chars) end
|       parseTerm(","::chars,symbol) = (leaf(symbol),","::chars)
|       parseTerm(")"::chars,symbol) = (leaf(symbol),")"::chars)
|       parseTerm(x::chars,symbol) = parseTerm(chars,symbol^x)
|       parseTerm(nil,"") = raise no_term
|       parseTerm(nil,symbol) = (leaf(symbol),nil)

and     parseTerms(" "::chars,tL) = parseTerms(chars,tL)
|       parseTerms(","::chars,tL) = parseTerms(chars,tL)
|       parseTerms(")"::chars,tL) = (tL,chars)
|       parseTerms(chars,tL) = let val (t,chars) = parseTerm(chars,"")
                               in parseTerms(chars,tL@[t]) end

(* compute coordinates *)

fun     strlg(a) = real(String.length(a))*7.0

fun     coordTree(a%nil)(x,y) = (leaf(a,x,y),strlg(a))
|       coordTree(a%tL)(x,y) = let val (ctL,b) = coordTrees(tL)(x,y-30.0)
                                    val (_,x,_)%_ = hd(ctL)
                                    val (_,z,_)%_ = last(ctL)
                                    val x = x*0.5+z*0.5
```

```
                                    val r = strlg(a)
                                    val b = if b >= r then b else r
                                in ((a,x,y)%ctL,b) end

and     coordTrees(nil) _ = (nil,0.0)
|       coordTrees(t::tL)(x,y) = let val (ct,b) = coordTree(t)(x,y)
                                     val (ctL,b') = coordTrees(tL)(x+b,y)
                                 in (ct::ctL,b+b') end

(* draw edges *)

fun     gotoRoot((a,x,y)%ctL) = moveto(x,y+10.0)^center(a)^traverse(ctL)(x,y)^
                                " showpage"

and     moveto(x:real,y:real) = makestring(x)^" "^makestring(y)^" moveto\n"

and     center(a) = "("^a^") dup stringwidth pop -2 div -8 rmoveto show\n"

and     traverse(nil) _ = ""
|       traverse(((a,x,y)%ctL)::ctL')(pred) = moveto(pred)^lineto(a,x,y)^
                                             traverse(ctL)(x,y)^
                                             traverse(ctL')(pred)

and     lineto(a,x:real,y:real) = makestring(x)^" "^makestring(y+10.0)^
                                  " lineto\n"^center(a)^"stroke\n"

(* draw tree *)

fun     maxX((_,x,_)%nil) = x
|       maxX((_,x,_)%ctL) = maxXL(ctL)

and     maxXL[ct] = maxX(ct)
|       maxXL(ct::ctL) = maxXL(ctL)

fun     draw(infile:int)(outfile:int)(scale:real)
          = let val file = open_in("TREEIN"^makestring(infile))
                fun instring(s) = if end_of_stream(file) then s
                                  else instring(s^input_line(file))
                val chars = explode(instring"")
                val (t,_) = parseTerm(chars,"")
                val file = (close_in(file);
                            open_out("TREEOUT"^makestring(outfile)^".eps"))
                val h = real(height(t))*30.0
                val scalestr = makestring(scale)
                val (ct,_) = coordTree(t)(40.0,h)
                val code = "%!PS-Adobe-3.0 EPSF-3.0\n"^
                           "%%BoundingBox: 10 10 "^
                           makestring(scale*40.0+scale*maxX(ct))^" "^
```

```
                        makestring(scale*30.0+scale*h)^"\n"^
                        scalestr^" "^scalestr^" scale\n"^
                        "/Helvetica 9 selectfont\n"^gotoRoot(ct)
          in output(file,code); close_out(file) end
```

15.2 buildShelves

Es folgt das in Kapitel 12 behandelte Programm zur Positionierung von Objekten in Regalen und deren Zusammensetzung zu Regalsystemen.

```
structure Aux = struct

val     permno = ref(0)

fun     writePermno() = (output(std_out,
                              "\nnumber of perms: "^makestring(!permno)^"\n");
                         permno:= 0)

fun     sum(pairs) = fold(op +)(map(op * )(pairs))(0)

fun     lex(x:int,y:int,x',y') = x < x' orelse (x = x' andalso y <= y')

fun     filter(p)(nil) = nil
|       filter(p)(x::L) = if p(x) then x::filter(p)(L) else filter(p)(L)

fun     sort(r)(nil) = nil
|       sort(r)(x::L) = let val low = filter(fn(y)=>r(y,x))(L)
                            val high = filter(fn(y)=>not(r(y,x)))(L)
                        in sort(r)(low) @ x::sort(r)(high) end

fun     firstperm(r) = sort(not o r)

exception no_nextperm

fun     nextperm(r)(x::L) = (permno:=(!permno)+1; next(r)([x],L))

and     next(r)(L1,z::L2)
          = if r(hd(L1),z) then next(r)(z::L1,L2)
            else let fun swap[x] = z::x::L2
                     |   swap(x::y::L) = if not(r(y,z)) then x::swap(y::L)
                                         else (z::y::L)@(x::L2)
                 in swap(L1) end
|       next(r)(_,nil) = raise no_nextperm

fun     nthperm(0) _ (L) = L
|       nthperm(n)(r)(L) = nthperm(n-1)(r)(nextperm(r)(L))

fun     read(infile:int)
```

```
          = let val file = open_in("INPUT"^makestring(infile))
                fun instring(s) = if end_of_stream(file) then s
                                  else instring(s^input_line(file))
                val chars = explode(instring"")
            in close_in(file); chars end

datatype polytype = rect | arc | circ | tri
exception no_type and no_color and wrong_size

fun     makeint"" = 0
|       makeint(s) = let val i = size(s)-1
                     in makeint(substring(s,0,i))*10+ordof(s,i)-48 end

fun     parseNo(x::chars,no) = if ord(x)>47 andalso ord(x)<58
                               then parseNo(chars,no^x)
                               else (makeint(no),chars)

fun     parseType(x::chars) = (case x of "r" => rect
                                       |  "a" => arc
                                       |  "c" => circ
                                       |  "t" => tri
                                       |  _ => raise no_type, chars)

fun     parseColor(x::")"::chars) = (case x of "b" => 1
                                           |  "g" => 2
                                           |  "t" => 3
                                           |  "r" => 4
                                           |  "v" => 5
                                           |  "y" => 6
                                           |  _ => raise no_color, chars)
|       parseColor _ = raise no_color

fun     parseObjects("c"::"o"::"n"::"t"::"s"::chars,objs) = (objs,chars)
|       parseObjects("("::chars,objs)
          = let val (l,chars) = parseNo(chars,"")
                val (h,chars) = parseNo(chars,"")
                val (t,chars) = parseType(chars)
                val (c,chars) = parseColor(chars)
            in if (t = arc andalso l <> 2*h) orelse
                  (t = circ andalso l <> h) then raise wrong_size
               else parseObjects(chars,(l,h,t,c)::objs) end
|       parseObjects(x::chars,objs) = parseObjects(chars,objs)

fun     parseConts(nil,conts) = conts
|       parseConts("("::chars,conts) = let val (l,chars) = parseNo(chars,"")
                                           val (h,chars) = parseNo(chars,"")
                                       in parseConts(chars,(l,h)::conts) end
|       parseConts(x::chars,conts) = parseConts(chars,conts)
```

```
fun     parseAll(chars) = let val (objects,chars) = parseObjects(chars,nil)
                              val conts = parseConts(chars,nil)
                          in (objects,conts) end

(*      Postscript-Funktionen      *)

fun     makestring2(x:int,y:int) = makestring(x)^" "^makestring(y)^" "
fun     makestring2r(x:real,y:real) = makestring(x)^" "^makestring(y)^" "
fun     makestring4(x:real,y:real,l:real,h:real)
          = makestring2r(x,y)^makestring2r(l,h)

fun     Scale(x,y,l,h,space,scale) = let val x = real(x*space-space)+scale
                                         val y = real(y*space-space)+scale
                                         val l = real(l*space)-scale-scale
                                         val h = real(h*space)-scale-scale
                                     in (x,y,l,h) end

fun     objfill(space)(x,y)(l,h,t,c)
          = let val (x,y,l,h) = Scale(x,y,l,h,space,1.0)
                val c = case c of 1 => "0 0 1"
                                | 2 => "0 1 0"
                                | 3 => "0 1 1"
                                | 4 => "1 0 0"
                                | 5 => "1 0 1"
                                | 6 => "1 1 0"
                fun str(a,b) = c^" setrgbcolor "^makestring2r(a,b)
            in case t of rect => str(x,y)^makestring2r(l,h)^"rectfill\n"
                       | arc  => str(x+h,y)^makestring(h)^" 0 180 arc fill\n"
                       | circ => let val r = 0.5*h
                                 in str(x+r,y+r)^" "^makestring(r)^
                                    " 0 360 arc fill\n" end
                       | tri => str(x,y)^" moveto "^
                                makestring2r(x+l,y)^"lineto "^
                                makestring2r(x+l*0.5,y+h)^"lineto fill\n" end

fun     objstroke(space)(x,y)(l,h,_,_)
          = let val (x,y,l,h) = Scale(x,y,l,h,space,0.5)
            in "1 setlinewidth .8 setgray "^
                makestring4(x,y,l,h)^"rectstroke\n" end

fun     shelfstroke(space)(x,y)(l,h)
          = let val (x,y,l,h) = Scale(x,y,l,h,space,7.5)
            in "15 setlinewidth .5 setgray "^
               makestring4(x,y,l,h)^" rectstroke\n"  end

end     (* Aux *)
```

```
signature Object = sig eqtype obj
                       val le : obj -> int
                       val he : obj -> int
                       val leq : obj * obj -> bool
                       val l_aligned : obj -> bool
                       val r_aligned : obj -> bool
                       val centered : obj -> bool
                       val adjacent_to : obj * obj -> bool
                       val on_top_of : obj * obj -> bool
                   end

functor Container(structure Obj : Object) = struct

open    Obj

abstype cont = new of int * int | add of obj * int * int * cont

with    val     New = new
        val     Add = add

        fun     len(new(l,_)) = l
        |       len(add(_,_,_,c)) = len(c)

        fun     hei(new(_,h)) = h
        |       hei(add(_,_,_,c)) = hei(c)

        fun     pos(a,add(b,i,j,c)) = if a = b then (i,j) else pos(a,c)

        fun     Objs(new _) = nil
        |       Objs(add(a,_,_,c)) = a::Objs(c)

        fun     makeEmpty(new(l,h)) = new(l,h)
        |       makeEmpty(add(a,_,_,c)) = makeEmpty(c)

        exception undef

        fun     get(new _,_,_) = raise undef
        |       get(add(a,i,j,c),x,y)
                  = if inside(a,i,j)(x,y) then a else get(c,x,y)

        and     inside(a,i:int,j:int)(x,y)
                  = i <= x andalso x < i+le(a) andalso
                    j <= y andalso y < j+he(a)

        fun     frame(new(l,h)) = new(l+2,h+2)
        |       frame(add(a,i,j,c)) = add(a,i+1,j+1,frame(c))

end     (* cont *)
```

```
fun     free(c,i,j) = (get(c,i,j);false) handle undef => true

fun     free_lwb(c)(i,j)(l) = if free(c,i,j) then (i,j)
                              else if i < l then free_lwb(c)(i+1,j)(l)
                              else free_lwb(c)(1,j+1)(l)

val     occupied = not o free

exception conts_too_small and nextPerm and Full and Restart of int*int

val     objSizes = map(fn(a)=>(le(a),he(a)))
val     contSizes = map(fn(c)=>(len(c),hei(c)))
val     contSize = ref(0,0)

fun     fill(objs)(conts)
          = if Aux.sum(objSizes(objs)) > Aux.sum(contSizes(conts))
            then raise conts_too_small
            else fillConts(objs)[hd(conts)](tl(conts))
                 handle nextPerm => newPerm(objs)(conts)

and     fillConts(nil)(conts)(emptyConts) = rev(conts)@emptyConts
|       fillConts(objs)(c::conts)(emptyConts)
          = let val (c,objs) = (contSize:=(len(c),hei(c));
                                addObjs(objs)(c)(1,1))
            in if objs = nil then rev(conts)@[c]@emptyConts
               else if null(emptyConts) then raise nextPerm
                    else fillConts(objs)(hd(emptyConts)::c::conts)
                                        (tl(emptyConts))
            end

and     newPerm(objs)(conts)
          = let val objs = Aux.nextperm(leq)(objs)
            in fillConts(objs)[hd(conts)](tl(conts))
               handle nextPerm => newPerm(objs)(conts) end

and     addObjs(nil)(c) _ = (c,nil)
|       addObjs(a::objs)(c)(lwb)
          = let val (l,_) = !contSize
                val lwb = free_lwb(c)(lwb)(l)
                val (c,full) = (addObj(a,c)(lwb)
                                handle Restart(lwb) => tryAgain(a,c)(lwb),
                                false) handle Full => (c,true)
            in if full then (c,a::objs) else addObjs(objs)(c)(lwb) end

and     tryAgain(a,c)(lwb) = addObj(a,c)(lwb)
                             handle Restart(lwb) => tryAgain(a,c)(lwb)
```

```
and     addObj(a,c)(i,j)
          = let val (x,y) = (i+le(a)-1,j+he(a)-1)
                val (l,h) = !contSize
            in if y > h then raise Full
               else if x > l then addObj(a,c)(1,j+1)
               else if free_base(c,i,x,j) andalso
                       is_adjacent_to(c,a,i,j,y) andalso
                       is_on_top_of(c,a,i,x,j) andalso
                       is_centered(c,a,i,j) andalso
                       is_l_aligned(c,a,i,j) andalso is_r_aligned(c,a,x,j)
                    then Add(a,i,j,c) else addObj(a,c)(i+1,j) end

and     free_base(c,i,x,j)
          = i > x orelse if free(c,i,j) then free_base(c,i+1,x,j)
                         else let val b = get(c,i,j)
                                  val (k,_) = pos(b,c)
                              in raise Restart(k+le(b),j) end

and     is_adjacent_to(c,a,i,j,y)
           = j > y orelse
             let val lfree = free(c,i-1,j)
                 val rfree = free(c,i+1,j)
             in (lfree andalso rfree) orelse
                let val l_adjacent = lfree orelse adjacent_to(get(c,i-1,j),a)
                    val r_adjacent = rfree orelse adjacent_to(a,get(c,i+1,j))
                in if l_adjacent andalso r_adjacent
                   then is_adjacent_to(c,a,i,j+1,y)
                   else raise Restart(i+1,y-he(a)+1) end end

and     is_on_top_of(c,a,i,x,j)
           = j = 1 orelse i > x orelse
             if occupied(c,i,j-1) andalso on_top_of(a,get(c,i,j-1))
             then is_on_top_of(c,a,i+1,x,j) else raise Restart(i+1,j)

and     is_centered(c,a,i,j)
           = centered(a) orelse j = 1 orelse
             (occupied(c,i,j-1) andalso
              let val b = get(c,i,j-1)
                  val (k,_) = pos(b,c)
                  val lspace = i-k
                  val rspace = k+le(b)-i-le(a)
                  val m = j mod 2
              in (lspace >= 0 orelse raise Restart(k,j)) andalso
                 (rspace >= 0 orelse raise Restart(k+le(b),j)) andalso
                 (lspace <= rspace+m orelse raise Restart(k+le(b),j)) andalso
                 (rspace+m <= lspace+1 orelse
                  raise Restart(i+(rspace+m-lspace) div 2,j)) end)
```

```
and     is_l_aligned(c,a,i,j)
          = l_aligned(a) orelse j = 1 orelse
            (occupied(c,i,j-1) andalso
             let val b = get(c,i,j-1)
                 val (k,_) = pos(b,c)
             in i = k orelse raise Restart(k+le(b),j) end)

and     is_r_aligned(c,a,x,j)
          = r_aligned(a) orelse j = 1 orelse
            (occupied(c,x,j-1) andalso
             let val b = get(c,x,j-1)
                 val (k,_) = pos(b,c)
                 val rb = k+le(b)-1
             in x = rb orelse if x > rb then raise Restart(rb+1,j)
                                        else raise Restart(rb-le(a),j) end)

end     (* Container *)

signature Picture = sig type obj and cont
                        val pos : obj * cont -> int*int
                        val le : obj -> int
                        val he : obj -> int
                        val len : cont -> int
                        val hei : cont -> int
                        val Objs : cont -> obj list
                        val space : int
                        val interior : int * int -> obj -> string
                        val border : int * int -> obj -> string
                    end

functor Display(structure Pict : Picture) = struct open Pict

fun     code(c,n) = let val file = open_out("CONT"^makestring(n:int)^".eps")
                        val sp = makestring(space)^" "
                        val (l,h) = (len(c)*space,hei(c)*space)
                        val profile = "%!PS-Adobe-3.0 EPSF-3.0\n"^
                                      "%%BoundingBox: 10 10 "^
                                      Aux.makestring2(l+20,h+20)^
                                      "\n/Helvetica 10 selectfont\n"^
                                      "1 setlinejoin "^sp^sp^"translate\n"^
                                      "0 setgray 0 0 "^Aux.makestring2(l,h)^
                                      "rectstroke\n"
                        val code = profile^objsCode(c)(Objs(c))^"showpage"
                    in output(file,code); close_out(file) end

and     objsCode(c)(nil) = ""
|       objsCode(c)(a::objs) = let val p = pos(a,c)
                               in interior(p)(a)^border(p)(a)^objsCode(c)(objs)
```

```
                              end

fun     shift(x,y)(c) = let val str1 = makestring(x*space)^" "
                            val str2 = makestring(y*space)^" translate\n"
                        in str1^str2^objsCode(c)(Objs(c))^"-"^str1^"-"^str2 end

end     (* Display *)

structure Obj = struct val R = Aux.rect
                       type obj = int*int*Aux.polytype*int
                       fun le(l,_,_,_) = l
                       fun he(_,h,_,_) = h
                       fun leq((l,h,_,_),(l',h',_,_)) = Aux.lex(l,h,l',h')
                       fun l_aligned _ = true
                       fun r_aligned _ = true
                       fun centered(_,_,t,_) = t = R
                       fun adjacent_to((_,_,t,c),(_,_,t',c'))
                             = t <> R orelse c <> 4 orelse
                               t' <> R orelse c' <> 4
                       fun on_top_of((l:int,h,t,c),(_,_,t',_))
                             = l >= h andalso (t <> R orelse c <> 4)
                               andalso t' = R
                end

structure objCont = Container(structure Obj = Obj)

structure objPict = struct open objCont
                           val space = 15
                           val interior = Aux.objfill(space)
                           val border = Aux.objstroke(space)
                    end

structure drawObjs = Display(structure Pict = objPict)

structure Shelf = struct type obj = objCont.cont ref
                          val le = objCont.len o !
                          val he = objCont.hei o !
                          fun leq(sh,sh') = Aux.lex(le(sh),he(sh),le(sh'),he(sh'))
                          fun l_aligned _ = false
                          fun r_aligned _ = false
                          fun centered _ = true
                          fun adjacent_to _ = true
                          fun on_top_of _ = true
                  end

structure shelfCont = Container(structure Obj = Shelf)

structure shelfPict = struct open shelfCont
```

```
                            val space = 15
                            fun interior(x,y) = drawObjs.shift(x-1,y-1) o !
                            fun border(p)(sh)
                                 = Aux.shelfstroke(space)(p)(le(sh),he(sh))
                        end

structure drawShelves = Display(structure Pict = shelfPict)

(*      Commands     *)

val     CONT = ref(0,0)
val     Objects = ref(nil:Obj.obj list)
val     Shelves = ref(nil:Shelf.obj list)

fun     init(infile:int)
          = let val chars = Aux.read(infile)
                val (objs,conts) = Aux.parseAll(chars)
            in Objects:= Aux.firstperm(Obj.leq)(objs);
               Shelves:= Aux.firstperm(Shelf.leq)(map(ref o objCont.New)(conts))
            end

fun     cont(l)(h) = CONT:=(l,h)

fun     nexts(n) = Shelves:= Aux.nthperm(n)(Shelf.leq)(!Shelves)
fun     nexto(n) = Objects:= Aux.nthperm(n)(Obj.leq)(!Objects)

fun     fills(outfile:int)
          = let val shelves = map(ref o objCont.frame o !)(!Shelves)
                val [CONT] = (Aux.permno:= 0;
                              let open shelfCont
                              in fill(shelves)[New(!CONT)] end)
            in Aux.writePermno(); drawShelves.code(CONT,outfile) end

fun     fillo(outfile:int)
          = let val shelves = (Aux.permno:= 0;
                               objCont.fill(!Objects)(map(!)(!Shelves)))
                val shelves = (Aux.writePermno();
                               Shelves:= map(ref)(shelves);
                               map(ref o objCont.frame)(shelves))
                val [CONT] = (Aux.permno:= 0;
                              let open shelfCont
                              in fill(shelves)[New(!CONT)] end)
            in Aux.writePermno(); drawShelves.code(CONT,outfile) end

fun     mt() = Shelves:= map(ref o objCont.makeEmpty o !)(!Shelves)
```

15.3 compFitness

Die Moduln

`Picture`, `Display`, `objCont`, `objPict`, `drawObjs`, `shelfCont`, `shelfPict`, `drawShelves`

von *buildShelves* (siehe 15.2) werden in folgendem Programm *compFitness* wiederverwendet, das Objekte an vorgegebenen Koordinaten positioniert und die Summe aller Überschneidungsflächen sowie die Position der größten Überschneidungsfläche berechnet und graphisch hervorhebt. Ebenfalls aus *buildShelves* übernommen werden die Funktionen *read*, *makeint*, *parseNo*, *Scale* und der abstype *cont*. *fun2matrix* und *procMatrix* sind 6.5 bzw. 6.6 definiert.

```
structure Aux = struct

fun     search(a,b)((c,x)::L) = if c = a orelse c = b then x
                                else search(a,b)(L)

fun     remove(a,(b,x)::L) = if a = b then L else (b,x)::remove(a,L)

fun     insert(a)(nil) = [a]
|       insert(a,x:int)((b,y)::L) = if x <= y then (a,x)::(b,y)::L
                                               else (b,y)::insert(a,x)(L)

fun     upd1(f,a,x)(b) = if a = b then x else f(b)
fun     upd2(f,(a,b),x:int)(c,d) = if (a,b) = (c,d) orelse (a,b) = (d,c)
                                   then f(c,d)*x else f(c,d)

exception arg of int*int

fun     getArg(f,b)(n) = let fun g(a) = if f(a) = b then raise arg(a) else ()
                         in (procMatrix(g)(n); (0,0)) handle arg(a) => a end

fun     sum(nil) = 0
|       sum(L::M) = fold(op +)(L)(0)+sum(M)

fun     Max(nil) = 0
|       Max(L::M) = max(fold(op max)(L)(0),Max(M))
```

Der folgende **Parser** erzeugt aus einer Folge F von Quadrupeln (x, y, l, h) die Liste *objs* aller Objekte (a, x, y, l, h) von F, die Liste XL aller Paare (a, x) und $(a, x + l)$ und die Liste YL aller Paare (a, y) und $(a, y + h)$, wobei a die Position von (x, y, l, h) in F ist. Die Listen XL und YL werden dabei nach der jeweils zweiten Komponente ihrer Elemente sortiert.

```
fun     parseObjs("c"::"o"::"n"::"t"::chars)(objs) = (objs,chars)
|       parseObjs("("::chars)(a:int,objs,XL,YL)                (* s.o. *)
          = let val (x:int,chars) = parseNo(chars,"")
                val (y:int,chars) = parseNo(chars,"")
                val (l,chars) = parseNo(chars,"")
                val (h,chars) = parseNo(chars,"")
```

```
            in parseObjs(chars)(a+1,(a,x,y,l,h)::objs,
                                   insert(a,x)(insert(a,x+l)(XL)),
                                   insert(a,y)(insert(a,y+h)(YL))) end
|       parseObjs(_::chars)(objs) = parseObjs(chars)(objs)

fun     parseCont("("::chars) = let val (l,chars) = parseNo(chars,"")
                                    val (h,chars) = parseNo(chars,"")
                                in (l,h) end
|       parseCont(_::chars) = parseCont(chars)

fun     parseAll(chars) = let val (objs,chars) = parseObjs(chars)(1,nil,nil,nil)
                              val (l,h) = parseCont(chars)
                          in (objs,l,h) end
```

Es folgt die Berechnung aller Überschneidungsflächen und Eintrag der Flächenwerte in die Matrix $XYdiff$. Der Aufruf *fitness(XL,YL,n)* liefert das Tripel, bestehend aus der Summe aller Einträge von $XYdiff$, dem maximalen Eintrag von $XYdiff$ und dessen Position in $XYdiff$. Die Einträge von $XYdiff$ ergeben sich wie folgt:

```
        XYdiff(a,b) = if Xdiff(a,b) <> 0 then Xdiff(a,b)*Ydiff(a,b)
                      else Xdiff(b,a)*Ydiff(a,b)
```

wobei für alle a, b $Xdiff(a,b) = 0$ oder $Xdiff(b,a) = 0$ oder $Xdiff(a,b) = Xdiff(b,a)$ gilt und die folgenden Äquivalenzen erfüllt sind:

$$\begin{array}{rcl}
Xdiff(a,b) \neq 0 & \iff & x1(a) \leq x1(b) \leq x2(a) \\
Xdiff(a,b) = x2(a) - x1(b) & \iff & x1(a) \leq x1(b) \leq x2(a) \leq x2(b) \\
Xdiff(a,b) = x2(b) - x1(b) & \iff & x1(a) \leq x1(b) \leq x2(b) < x2(a) \\
Ydiff(a,b) \neq 0 & \iff & y1(a) \leq y1(b) \leq y2(a) \\
Ydiff(a,b) = y2(a) - y1(b) & \iff & y1(a) \leq y1(b) \leq y2(a) \leq y2(b) \\
Ydiff(a,b) = y2(b) - y1(b) & \iff & y1(a) \leq y1(b) \leq y2(b) < y2(a)
\end{array}$$

```
exception ret of int*int->int

fun     diff(f,_,nil) = f
|       diff(f,upd,(a,x)::L) = diffAux(f,upd,a,L)
                               handle ret(f) => diff(f,upd,remove(a,L))

and     diffAux(f,upd,a,(b,x:int)::L) = if a = b then raise ret(f)
                                        else let val y = search(a,b)(L)
                                                 val f = upd(f,(a,b),y-x)
                                             in diffAux(f,upd,a,remove(b,L)) end

fun     fitness(XL,YL,n) = let val Xdiff = diff(fn _=>0,upd1,XL)
                               val XYdiff = diff(Xdiff,upd2,YL)
                               val M = fun2matrix(XYdiff)(n,n)
                               val m = Max(M)
                               val (a,b) = getArg(XYdiff,m)(n,n)
                           in (sum(M),m,a,b) end
```

```
(*      Postscript-Funktionen      *)

fun     makestring2(x:int,y:int) = makestring(x)^" "^makestring(y)^" "
fun     makestring4(x:real,y:real,l:real,h:real)
          = makestring(x)^" "^makestring(y)^" "^makestring(l)^" "^
            makestring(h)^" "

fun     makeColor(0) = " 0 1 1 0"
|       makeColor(a) = let fun f(x) = if x div 4 > 0 then 1+f(x div 4) else 1
                           fun g(0) = " .15" | g(1) = " .4" | g(2) = " .65"
                           |   g(3) = " .9"
                           fun h(x,0) = ""
                           |   h(x,n) = h(x div 4,n-1)^g(x mod 4)
                           fun iter(0) = "" | iter(n) = " 0"^iter(n-1)
                           val n = f(a)
                       in h(a,n)^iter(4-n) end   (*  n > 4 ==> error !  *)

fun     objfill(space)(x,y)(a,_,_,l,h)
          = let val (x,y,l,h) = Scale(x,y,l,h,space,1.0)
            in makeColor(a)^" setcmykcolor "^makestring4(x,y,l,h)^
               "rectfill\n"  end

fun     objstroke(space)(x,y)(a,_,_,l,h)
          = let val (x,y,l,h) = Scale(x,y,l,h,space,0.5)
            in "1 setlinewidth "^(if a = 0 then makeColor(0) else "0 0 0 .4")^
               " setcmykcolor "^makestring4(x,y,l,h)^"rectstroke\n"  end

fun     shelfstroke(space)(x,y)(l,h)
          = let val (x,y,l,h) = Scale(x,y,l,h,space,7.5)
            in "15 setlinewidth .5 setgray "^
               makestring4(x,y,l,h)^"rectstroke\n"  end

end     (* Aux *)

signature Object = sig eqtype obj
                       val XY : obj -> int*int
                       val le : obj -> int
                       val he : obj -> int
                   end

functor Container(structure Obj : Object) = struct

open    Obj

fun     pos(a,_) = XY(a)

fun     fill(a::objs)(c) = let val (x,y) = XY(a)
                           in fill(objs)(Add(a,x,y,c)) end
```

```
|       fill(nil)(c) = c

end     (* Container *)

structure Obj = struct type obj = int*int*int*int*int
                       fun XY(_,x,y,_,_) = (x,y)
                       fun le(_,_,_,l,_) = l
                       fun he(_,_,_,_,h) = h
                end

structure Shelf = struct type obj = objCont.cont ref
                         fun XY _ = (1,1)
                         val le = objCont.len o !
                         val he = objCont.hei o !
                  end

fun     drawFit(infile:int)(outfile:int)
          = let val chars = Aux.read(infile)
                val (objs,l,h) = Aux.parseAll(chars)
                val (n,objs,XL,YL) = objs
                val n = n-1
                val (all:int,m:int,a:int,b:int) = Aux.fitness(XL,YL,n)
                val str = "\nsum of intersections: "^makestring(all)^
                          "\nmaximal intersection: "^makestring(m)^" between "^
                          "object "^makestring(a)^" and object "^
                          makestring(b)^"\n"
                val (a,xa,ya,la,ha) = nth(objs,n-a)
                val (b,xb,yb,lb,hb) = nth(objs,n-b)
                val xc = max(xa,xb)
                val yc = max(ya,yb)
                val lc = min(xa+la,xb+lb)-xc
                val hc = min(ya+ha,yb+hb)-yc
                val objs = (0,xc,yc,lc,hc)::objs
                val shelf = let open objCont
                            in (ref o frame o fill(objs) o New)(l,h) end
                val CONT = let open shelfCont
                           in (fill[shelf] o New)(l+2,h+2) end
            in drawShelves.code(CONT,outfile); output(std_out,str) end
```

Angewendet auf die Eingabe

```
(4,4,11,2)(8,8,3,3)(5,5,3,3)(7,7,3,3)(11,15,3,3)(16,3,3,3)(4,4,2,11)
(14,24,14,2)(18,13,3,3)(15,5,3,3)(7,17,3,3)(21,15,1,12)
cont (30,26)
```

liefert *drawFit* die Beladung von Fig. 24 und die Ausgabe

```
sum of intersections: 48
maximal intersection: 4 between object 4 and object 2
```

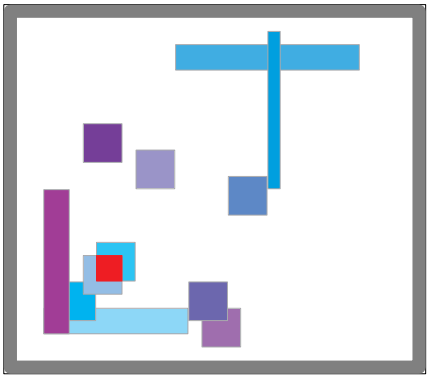


Figure 24.