Programmiersprachen
Vorlesung im WS 95/96

Harald Ganzinger

5. Dezember 1995

Inhaltsverzeichnis

1 Die Vorlesung

Die Vorlesung (4V+2Ü) ist eine Stammvorlesung im Bereich "Praktische Informatik", die regelmäßig (in der Regel einmal pro Jahr) angeboten wird. Sie richtet sich an Informatikstudenten nach dem Vordiplom. Weitere Voraussetzungen gibt es nicht. Leistungsnachweise werden in Form von Übungsscheinen ausgestellt, die die erfolgreiche Bearbeitung eines gewissen Prozentsatzes an Übungsaufgaben voraussetzen.

Beschreibung

Die Vorlesung bespricht die grundlegenden Konzepte heutiger Programmiersprachen aus der Sicht der praktischen Anwendung zur Lösung von Softwareproblemen.

Programmierung bedeutet nicht nur die Entwicklung und das Testen (oder, besser, Verifizieren) irgendeines korrekten und effizienten Programms zur Lösung eines bestimmten Anwendungsproblems. Es ist auch wichtig, daß das Programm gute Wartungseigenschaften besitzt. Ein Programm soll daher möglichst viele, in anderen Kontexten wiederverwendbare Teile besitzten, es soll leicht modifizierbar sein, wenn sich das ursprünglich gestellte Problem ändert, es soll von Leuten verstanden werden können, die an seiner Entwicklung nicht beteiligt waren, und vieles mehr.

In der Vorlesung untersuchen wir die Konzepte heutiger Sprachen hinsichtlich der Art und Weise, wie sie diese Anforderungen an die Programmierung unterstützen. Folgende Fragen werden deshalb typischerweise gestellt:

- Wie helfen Sprachkonzepte, bestimmte Programmier- und Programmentwicklungstechniken umzusetzen?
- Was sind die Programmiertechniken, für die die Sprache gemacht ist?
- Welches Rechenmodell steht hinter einem Sprachkonzept?
- Wie unterstützen Sprachkonzepte die Verifikation von Programmen?
- Wie wird Abstraktion von der konkreten Problemstellung ermöglicht?
- Wie wird Abstraktion von der konkreten Implementierungstechnik ermöglicht?
- Wie wird Strukturierung großer Programmsysteme unterstützt?
- Für welche Softwareprobleme bietet ein Konzept keine Unterstützung?
- Ist ein Konzept leicht und/oder effizient implementierbar, d.h. in Maschinensprache (welcher Maschinen?) übersetzbar?

Diese Fragen werden nicht theoretisch oder technisch detailiert behandelt, sondern anhand ausgewählter, nichttrivialer Programmierbeispiele erläutert. Verweise auf die theoretische oder technisch detailiertere Behandlung, die man in angrenzenden Vorlesungen (z.B. Übersetzerbau, Semantik von Programmiersprachen, Spezifikation und Verifikation, Logische Programmierung, λ-Kalkül und Typentheorie) lernen kann, werden gegeben.

Die Vorlesung orientiert sich an sprachübergreifenden Konzepten (Kontrollstrukturen, Datentypen, Berechnungsmodellen, Abstraktionsmechanismen), eine Sichtweise, die oft parallel, aber manchmal auch orthogonal zu den großen Programmierparadigmen (imperativ, funktional, logisch, objektorientiert) liegt. Die Vorlesung verwendet konkrete Programmiersprachen als Anschauungsmaterial, insbesondere ML, Haskell, Prolog, Smalltalk und C(++).

Die begleitenden Übungen sind teils theoretisch, aber zum größeren Teil praktisches Programmieren in einigen der genannten Sprachen.

Lernziele

- Beherrschen von grundlegenden Techniken des Programmierens in imperativen, funktionalen, logischen und objektorientierten Programmiersprachen
- Multilinguales Programmieren
- Erwerben von Wissen über die Art und Weise, wie einzelne Sprachkonzepte bei der Lösung gewisser Softwareprobleme helfen oder auch nicht helfen
- Einblick in die wesentlichen Konzepte der operationalen Semantik einiger Sprachen
- Erwerben von Verständnis für theoretische und praktische Problemstellungen bei der semantischen Erklärung oder bei der Implementierung von Programmiersprachen

Stichworte zum Inhalt

1. Funktionale Programmierung: Termtypen, Typpolymorphismus und Typinferenz, parametrische Module, Typklassen, Auswertungsstrategien (ML und Haskell)
2. Logische Programmierung: Unifikation, Einbringen primitiver Logiken über den Mechanismus der constraints, Ausführung als Constraintsimplifikation, nebenläufiges logisches Programmieren mit constraints
3. Vererbung und objekt-orientierte Programmierung (Smalltalk-80, C++)
4. grundlegende Konzepte von Nebenläufigkeit und Kommunikation

Literatur

Lehrbücher:

1. R. Sethi: Programming languages: concepts and constructs, Addison-Wesley 1989.
2. C.A.R. Hoare: Communicating Sequential Processes, Prentice-Hall, 1985.
3. D.A. Watt: Programming Language Concepts and Paradigms, Prentice-Hall 1989.
4. C. Ghezzi, M. Jazayeri: Programming Language Concepts, Wiley 1987.

Spezialliteratur: wird in der Vorlesung angegeben.

2 Einleitung

Literatur: R. Sethi: Programming languages: concepts and constructs, Addison-Wesley 1989, Kapitel 1

- Ein Programm ist die Spezifikation einer Berechnung.
- Eine Programmiersprache ist eine Notation zum Schreiben von Programmen.
- Etwas Geschichte.
- Sprachentwicklung wird geleitet durch die Erkenntnis, daß Komplexität durch Strukturierung handhabbar wird.

2.1 Die Rolle von Struktur in der Programmierung

Berühmtes Beispiel: Mariner I-Absturz (1962)

> **if not** in radar contact **then** do not correct flight path

Einsichten: Dijkstra (1972)

> "The art of programming is the art of organising complexity."

Anforderungen an Programmiersprachen: Leichte

- Lesbarkeit
- Verstehbarkeit
- Modifizierbarkeit

von Programmen

Problem: Skalierbarkeit von Techniken

Aus der Psychologie:
Miller (1967): Menschen können nur mit bis zu sieben Dingen gleichzeitig operieren. Wichtig, daß in möglichst wenige Symbole möglichst viel Information gepackt wird.

Lösung: Abstraktion und Modularisierung

2.2 Die von-Neumann-Maschine

- Am Anfang waren die Maschinen.
- vNM-Entwicklung: späte 40er Jahre in Princeton unter John von Neumann
- Kontrolle—Arithmetik—I/O—Speicher
- kein Unterschied zwischen Daten und Befehlen bzgl. Speicherung
- Sprünge als Mittel, Iterationen zu programmieren

- Maschinensprache (einschl. Assembler) ist unverständlich; zählen wir nicht zu den Programmiersprachen

Nicht modelliert wird:

- Nebenläufigkeit, Kommunikation, Verteilung, Vernetzung, ...

2.3 Frühe Erfahrungen

2.3.1 Dominanz der Laufzeiteffizienz

FORTRAN als Alternative zur M.-Programmierung (Backus et al. 1957); für numerische Anwendungen; konzise, mathematische[1] Sprache; effiziente, automatische Übersetzung; Reduktion des Programmieraufwands auf 20%.

- `B*B - 4*A*C` sehr ähnlich zu $b^2 - 4ac$
- Beginn des Faches Übersetzerbau
- Effizienzeinbußen: Zeit zum Übersetzen; nicht optimierter Zielcode (Speicher und Zeit), selbst ein Faktor 2 gegenüber handgeschriebenen Programmen erschien nicht akzeptierbar
- Der Bau des Übersetzers war das dominierende Problem im FORTRAN-Projekt.
- wurde schnell die wichtigste Basis für die wissenschaftliche (numerische) Programmierung

Das Problem der Effizienzeinbußen wird immer wieder genannt, wenn argumentiert wird, warum eine der moderneren (z.B funktionalen oder logischen) Programmiersprachen nicht verwendet werden soll.

2.3.2 Vorteile der Hochsprachen

- Lesbarkeit
- schnellere, problemorientiertere Programmentwicklung
- Maschinenunabhängigkeit, Portabilität
- Frühes Fallbeispiel: Reimplementierung von Unix in C (Ritchie 1978)
  - neue Benutzer und Erweiterungen des Systems
  - Portabilität: ein maschinenunabhängiges Betriebssystem
  - Restrukturierung durch verbesserte Lesbarkeit
  - Ritchies Resumée: „Ganz egal wie viel mehr Zeit und Speicher verbraucht wird, wir würden niemals mehr in Assembler programmieren.“

[1] *For*mula *Tran*slation

2.3.3 Nochmals zur Effizienz

- Lesbarkeit und verbesserte Struktur tragen zur Effizienzverbesserung bei; Effizienz wird auf allen Entwurfsebenen beeinflußt, falsche Entscheidungen sind in einer Hochsprache leichter revidierbar
- Knuth 1971: Lange Geschichte der Optimierung der falschen Dinge; weniger als 4% des Codes sei verantwortlich für mehr als 50% der Laufzeit
- Verwendung von profilers

2.4 Programmiersprachen als Maschinenerweiterungen

- Maschinen haben keine problemorientierte Struktur im Vergleich zu Formeln in FORTRAN
- Programmiersprachen erweitern die Maschine um grundlegende (mathematische) Struktur und um die Fähigkeit zum Bilden von problemorientierter Struktur
  - Berechnungsmodell: die reale Maschine scheint durch oder aber auch gerade nicht; Beispiel: Multiprocessing auf einer Maschine
  - Datentypen und Operationen: Typisierung von Daten, strukturierte Daten
  - Abstraktionsmittel: Typdefinitionen und Funktionsdefinitionen erweitern die Grundbegriffe der Sprache in problemorientierter Weise;
  - Problemlösung = Finden einer geeigneten Notation
  - Hat eine Sprache Mittel zur Abstraktion, sind Spezialsprachen unnötig. Beispiel: `sqrt` aus $+$, $-$; queues aus arrays
  - Erkennung von Fehlern zur Übersetzungszeit
  - Schutz vor unerlaubten Zugriffen auf interne Strukturen

2.5 Wie soll man ein Programm strukturieren?

Programmstruktur ist nicht inhärent sondern die Folge von Entwurfsentscheidungen

Beispiel: Auffinden von Duplikaten in Listen

Lösung 1:

```
sort | find_adjacent_duplicates
```

- top-down Reduktion des Problems auf zwei einfachere Teilprobleme
- Transformation elementarer Datenstrukturen
- Teilprobleme sind unabhängig, können in verschiedenen Sprachen implementiert werden
- vgl. FP oder pipes in Unix

Lösung 2:

```
for each element x in the list do
    if find(x) then write(x) else insert(x)
```

- Strukturierung beruht auf Verwendung eines abstrakten Datentyps mit dessen Hilfe die Begriffswelt problemorientiert erweitert wird
- bottom-up
- Verwendung einer Tabelle ohne daß man wissen muß, wie sie implementiert ist
- interagierende Objekte

2.6 Sprachbeschreibungen

- Syntax; Metasprachen klar (BNF, Syntaxdiagramme)
- Semantik
  - nur wenige Sprachen haben eine präzise oder sogar formale Semantikdefinition;
  - welche Metasprache? (ML ist das Acronym für Meta-Language.)
  - wozu?
- Pragmatik

2.7 Historisches

- Programmierung gab es schon vor der Erfindung des Computers.

  Beispiel: lochkartengesteuerter Webstuhl von Jacquard (1801); bis zu 25000 Karten mit bis zu 1050 Löchern für ein seidengewebtes Portrait von 0.5 Quadratmetern mit Auflösung 1000dpi.

  Beispiel: Turing-Maschine (um 1935)
- Historische Computer:

  mechanische Rechenmaschinen seit der Renaissance;

  erste Maschine mit Speicher, CPU ("mill") und Programmierung: Analytical Engine (Babbage 1864)
- siehe Sethi-Buch für Referenzen auf Arbeiten zur Geschichte der Computer und der Programmierung; Artikel im Informatik-Spektrum von F.L. Bauer
- Strukturierte Programmierung (Dijkstra 1972)
- Grundprinzipien der Datenabstraktion (Hoare 1972)
- BNF (Backus 1960; Naur 1963)
- Grundlagen für denotationale Semantik (Scott 1969); Beginn der funktionalen Programmierung

- Konzepte für Nebenläufigkeit und Kommunikation (Hoare CSP 1978, Milner CCS 1980)
- Logik als Programmiersprache (Colmerauer, Kowalksi 1974)

3 Funktionale Programmiersprachen

- Auswertung von Ausdrücken
- Werte hängen nur von den Werten der Teilausdrücke ab
- meistens keine Seiteneffekte
- strukturierte Daten mit impliziter Speicherverwaltung
- Funktionen als Bürger erster Klasse

Geschichte: Lisp (McCarthy 1958) war die erste Sprache, die Rekursion, Funktionen höherer Ordnung, Listen mit Speicherbereinigung und eine (in Lisp verfaßte) formale Sprachdefinition (Reflexivität!) anbot. FP (Backus 1978) hebt die Kombinatorprogrammierung hervor. Miranda (Turner 1986) hat verzörgerte Operandenauswertung, ebenso wie Haskell (Hudak, Wadler 1990). Letztere Sprache hat Typklassen, die Vererbungsmechanismen zu definieren gestatten. Funktionale Programmierung war immer stark mit rechnergestütztem Beweisen verbunden.

3.1 Übersicht

- Einführung in ML
- Programmiertechniken
- Verifikationstechniken, insbesondere strukturelle Induktion
- Rechenmodell Termersetzung
- Typinferenz in ML

3.2 Einführung in ML

Literatur:
R. Harper: Introduction to Standard ML, Carnegie Mellon University 1986–1993; siehe WWW-Seite
L.C. Paulson: ML for the Working Programmer, Cambridge University Press 1991.

3.2.1 Sprachübersicht

- Funktionen sind Daten erster Klasse. Insbesondere können sie Argumente und Ergebnisse von Funktionsaufrufen sein oder in Variablen gespeichert werden. Die Anwendung (rekursiver) Funktionen auf ein Argument ist der hauptsächliche Kontrollmechanismus der ML-Programmierung.
- Interaktivität
- starke Typisierung: *Typinferenz* durch den Übersetzer, garantiert kein Typfehler zur Laufzeit
- polymorphes Typsystem mit automatisch inferierbaren allgemeinsten Typen
- parametrisierte, abstrakte Datentypen

- statische Bindung von Variablen
- typsichere Ausnahmebehandlung

Geschichte: Erste Versionen um 1975 von Gordon, Milner, Wadsworth im Zusammenhang mit Semantik und Verifikation von Programmiersprachen (LCF). Standard ML, welches in dieser Vorlesung behandelt wird, beinhaltet Ideen von HOPE und von Cardellis ML-Dialekt. Das Modulsystem stammt von MacQueen, vgl. Literaturangaben in Harpers Buch.

3.2.2 Primitive Ausdrücke, Werte und Typen

Interaktion

```
hg@mpii02039(88)% sml
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- 3+2;
val it = 5 : int
- 3+it;
val it = 8 : int
- it+cos(3.0);
std_in:4.1-4.11 Error: operator and operand don't agree (tycon mismatch)
  operator domain: int * int
  operand:         int * real
  in expression:
    + : overloaded (it,cos (3.0))
- real(it)+cos(3.0);
val it = 7.01000750339955 : real
-
```

Einheitstyp Der Type `unit` hat den einzigen Wert `()`. Wird gebraucht, wenn Wert uninteressant oder für parameterlose Funktionen.

Booleans

```
- if not true then true andalso false else true;
val it = true : bool
```

- `then`-Typ = `else`-Type
- `else` zwingend notwendig
- vordefinierte Funktionen `not`, `andalso`, `orelse`

Ganze Zahlen

```
- 75;
val it = 75 : int
- ~13;
val it = ~13 : int
```

- vordefinierte Funktionen `+`, `-`, `*`, `div`, `mod`
- vordefinierte Relationen `<`, `<=`, `>`, `>=`, `=`, `<>`,

Zeichenreihen

```
- "Programmiersprachen WS94/95";
val it = "Programmiersprachen WS94/95" : string
- "Programmiersprachen " ^ "WS94/95";
val it = "Programmiersprachen WS94/95" : string
- size it;
val it = 27 : int
- "\"Programmiersprachen\" WS94/95";
val it = "\"Programmiersprachen\" WS94/95" : string
- size it;
val it = 29 : int
```

Reelle Zahlen

```
- 3.14
= ;
val it = 3.14 : real
- 3.14;
val it = 3.14 : real
- 3E2;
val it = 300.0 : real
- 3.14E3;
val it = 3140.0 : real
```

- vordefinierte Funktionen `+`, `-`, , `*`, `/`, `sin`, `sqrt`, `exp`, usw.
- vordefinierte Relationen `<`, `<=`, `>`, `>=`, `=`, `<>`,
- keine implizite Anpassung zwischen `int` und `real`; keine gemischten Operandentypen in binären Funktionen und Relationen; explizite Anpassung durch `real` bzw. `floor`.

Tupel

```
- (true, ());
val it = (true,()) : bool * unit
- (1,false,"WS94/95",(true,12));
val it = (1,false,"WS94/95",(true,12)) : int * bool * string * (bool * int)
- (14 mod 3, not false) = (1+1, not (not true));
val it = true : bool
- (true, 7) = (true, 8);
val it = false : bool
- (true, 7) = (true, ());
std_in:0.0-0.0 Error: operator and operand don't agree (tycon mismatch)
  operator domain: (bool * int) * (bool * int)
```

```
  operand:         (bool * int) * (bool * unit)
  in expression:
    = ((true,7),(true,()))
```

- geordnete Tupel beliebiger Länge $n \geq 2$
- Gleichheit komponentenweise
- Typausdrücke von der Form $\tau_1 * \tau_2 * \ldots * \tau_n$

Listen

```
- nil;
val it = [] : 'a list
- 3 :: 4 :: nil;
val it = [3,4] : int list
- [3,4];
val it = [3,4] : int list
- (3 :: nil) :: (3 :: 4 :: nil) :: nil;
val it = [[3],[3,4]] : int list list
- ["Das", "ist", "eine", "Liste"];
val it = ["Das","ist","eine","Liste"] : string list
```

Listen werden konstruiert über `nil` (leere Liste) und `::` (ausgesprochen „cons“), womit ein neues Element an eine bestehende Liste vorne angehängt wird. Man spricht von einem *rekursiven Datentyp.*

Listen mit Elementen $e_1, \ldots, e_n$, $n \geq 0$, haben die Form e_1 `:: ... ::` e_n `:: nil`, wofür auch `[`e_1 `:: ... ::` e_n`]` geschrieben werden darf.

Die Elemente einer Liste müssen alle vom selben Typ τ sein, in welchem Fall dann τ `list` der Typ der Liste ist.

Der Typ von `nil` enthält die *Typvariable* `'a` (sprich „α“), da `nil` Element eines jeden Listentyps ist. `nil` ist ein *polymorphes Objekt*, welches aber nur in Listentypen enthalten ist. Ein Typ, der eine Typvariable enthält, heißt ein *polymorpher Typ. Instanzen* eines polymophen Typs erhält man, indem man Typvariablen durch Typen *substituiert. Monomorphe Typen* (Monotypen) sind Typen, die keine Typvariablen enthalten.

Man beachte den Unterschied zwischen (parametrischem) Polymorphismus und Überladung von Bezeichnern, auch ad hoc Polymorphismus genannt, wie er z.B. bei `+` oder `>=` auftritt.

Verbunde

Verbundtypen sind ähnlich den „Pascal records“ und „C structures“.

```
- {Name="Ganzinger", Alter=44};
val it = {Alter=44,Name="Ganzinger"} : {Alter:int, Name:string}
- {a=1,b=2} = {b=1+1, a=1};
val it = true : bool
- {a=1,b=2} = {b=1+1, c=1};
std_in:30.1-30.24 Error: operator and operand don't agree (tycon mismatch)
  operator domain: {a:int, b:int} * {a:int, b:int}
  operand:         {a:int, b:int} * {b:int, c:int}
```

```
in expression:
  = ({a=1,b=2},{b=+ : overloaded (1,1),c=1})
```

Verbundtypen sind endliche Mengen von benamten Feldern der Form

$$\{l_1\texttt{=}v_1\texttt{,}\ldots\texttt{,}l_n\texttt{=}v_n\},$$

$n \geq 1$.[2] Namen müssen eindeutig sein. Die zugehörigen Typausdrücke haben die Form

$$\{l_1\texttt{:}\tau_1\texttt{,}\ldots\texttt{,}l_n\texttt{:}\tau_n\},$$

wenn τ_i der Typ von v_i ist. Die Namen der Felder zählen also zur Typinformation. Die Reihenfolge der Felder ist irrelevant.

Tupeltypen $\tau_1 * \tau_2 * \ldots \tau_n$ sind Kurzschreibweisen für Verbundtypen der Form $\{\texttt{1:}\tau_1\texttt{,}\ldots\texttt{,n:}\tau_n\}$. D.h. die Ausdrücke `(2,true)` und `{1=2,2=true}` haben dieselbe Bedeutung. Als Feldnamen sind also Identifikatoren und ganze Zahlen zugelassen.

Bemerkungen

Man beachte die Uniformität der Art und Weise, wie in ML Werte der verschiedenen Typen mit *Konstruktoren* aus Konstanten gebildet werden. Jeder Wert ist also ein *Term*, an dessen Blättern Konstante atomarer Typen (Zahlen) oder (nullstellige) Konstruktoren (z.B. `nil`, `true`) stehen und an deren inneren Knoten (mehrstellige) Konstruktoren (z.B. `::`, `( , )`, `{name= , alter= }`) angewendet sind. Obwohl Konstruktoren Daten in eindeutiger Weise konstruieren, sind sie aber keine Funktionen im technischen Sinne von ML.

3.2.3 Funktionsdefinitionen

Funktionsdefinitionen sind spezielle Formen von bindungserzeugenden Konstrukten, die wir in ihrer ganzen Schönheit noch später kennenlernen werden.

Funktionen werden verwendet, indem sie auf Argumente angewendet werden. Funktionen in ML sind allesamt **einstellig**. Mehrstellige Funktionen können entweder durch tupelwertige Argumente oder durch Funktionen höherer Ordnung realisiert werden. Applikationen haben die Form `e1 e2`. Eine Applikation ist typmäßig korrekt, wenn `e1` einen Typ von der Form $\sigma \rightarrow \tau$ und `e2` dann den Typ σ hat. Ergebnis der Anwendung ist in diesem Fall ein Wert vom Typ τ. Diese *Typbedingung* garantiert, daß `e1` bei seiner Auswertung[3] eine Funktion liefern wird, die Argumente vom Typ von `e2` akzeptiert. σ bzw. τ heißen die Typen des *Definitions-* bzw. *Wertebereichs* der Funktion.

```
- size;
val it = fn : string -> int
- not;
val it = fn : bool -> bool
- not 3 ;
std_in:4.1-4.5 Error: operator and operand don't agree (tycon mismatch)
  operator domain: bool
  operand:         int
```

[2] Der leere Verbund {} ist eine andere Notation für `()` und hat den Typ `unit`

[3] Terminierung wird bei Aussagen dieser Form immer unterstellt. Natürlich kann kein statisches Typkonzept Terminierung garantieren.

```
  in expression:
    not 3
- size "abc";
val it = 3 : int
```

Es folgen einfache Beispiele für Funktionsdefinitionen.

```
- fun twice x = 2*x;
val twice = fn : int -> int
- twice 4;
val it = 8 : int
- fun fact x = if x=0 then 1 else x*fact(x-1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int
- fun plus(x,y) = x+y;
std_in:10.18 Error: overloaded variable cannot be resolved: +
- fun plus(x,y):int = x+y;
val plus = fn : int * int -> int
- plus (3,6);
val it = 9 : int
```

Folgendes sind die Bestandteile solcher Definitionen:

- Name der zu definierenden Funktion, z.B. `plus`,
- Parametermuster, z.B. `(x,y)`,
- optionale Ergebnistypbeschränkung, z.B. `:int`, die vornehmlich zur Auflösung von Typmehrdeutigkeiten, die aus Überladung entstehen, verwendet werden,
- Ausdruck, der das Funktionsergebnis zum jeweiligen Parametermuster spezifiziert, z.B. `x+y`.

Die Parametermuster liefern Bindungen an die in ihnen enthaltenen Variablen, wenn das Muster auf den jeweiligen aktuellen Parameterwert gelegt wird. Wesentlich ist, daß **mehr als eine Regel** (oder *Klausel*) der Form

```
<Funktionsname> <Muster> = <Ergebnisausdruck>
```

angegeben werden darf. Im allgemeinen sind es soviele, daß die Vereinigung der Muster **alle** möglichen Parameterwerte *überdeckt*. Die Klauseln werden durch `|` getrennt. Paßt mehr als eine Regel, wird die erste passende verwendet. Dafür jetzt ein Beispiel.[4]

```
- fun isNil   nil  = true
=   | isNil (_::_) = false;
val isNil = fn : 'a list -> bool
- isNil nil;
val it = true : bool
- isNil [2,3];
```

[4]Die =-Zeichen am Anfang von (Folge-)Zeilen ist Ausgabe des Interpreters wenn Eingaben über eine Zeile hinausgehen, nicht Eingabesyntax!

```
val it = false : bool
- isNil ["aa","ab"];
val it = false : bool
- fun append (nil,l) = l
=   | append (hd::tl,l) = hd :: append (tl,l);
val append = fn : 'a list * 'a list -> 'a list
- append ([],[1,2,3]);
val it = [1,2,3] : int list
- append (["auch", "eine"],["Liste"]);
val it = ["auch","eine","Liste"] : string list
- append ([1,2],["a"]);
std_in:3.1-3.20 Error: operator and operand don't agree (tycon mismatch)
  operator domain: int list * int list
  operand:         int list * string list
  in expression:
    append (1 :: 2 :: nil,"a" :: nil)
```

Bemerkungen:

- Beide Funktionen sind polymorph. Man bemerke aber bei `append`, daß diese Funktion zwar Listen von beliebigem Elementyp akzeptiert, daß aber die Elementtypen beider Argumentlisten gleich sein müssen (Typ $\alpha\,\texttt{list} * \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$). Der ML-Übersetzer findet den allgemeinsten Typ selbst heraus.

- `append` ist rekursiv. Auf der rechten Seite der Regeln sind also alle links eingeführten Bindungen, einschließlich der zu definierenden Funktion, sichtbar.

- Ist eine Teilstruktur eines Wertes irrelevant beim Mustertest, so verwendet man den *Joker* _, siehe zweite Regel für `isNil`.

- Die Verwendung von Mustern und Mustertest ist die einzige Möglichkeit, um auf Komponenten von Konstruktortermen zuzugreifen. Es gibt in ML **keine** expliziten Destruktoren.

Bevor wir zu interessanteren Beispielen kommen, wollen wir in Abschnitt 3.2.4 allgemeiner auf Identifikatoren, Bindungen und Deklarationen sowie Muster und Mustertest eingehen.

Funktionen müssen in SML nicht notwendig einen Namen haben. Sogenannte λ-*Ausdrücke* der Form

```
    fn <Muster> => <Ausdruck>
```

definieren *namenlose Funktionen*, die Argumentmuster `Muster` auf Ergebnisse `Ausdruck` abbilden. Diese kann man direkt in Applikationen verwenden oder in Deklarationsausdrücken. Dann sind auch (nichtrekursive) Funktionsdefinitionen spezielle Form von Wertedefinitionen.

```
fn x => x;
- val it = fn : 'a -> 'a
- it(1);
val it = 1 : int
```

```
- val id = fn x => x;
val id = fn : 'a -> 'a
- val plus = fn (x,y):int*int => x+y;
val plus = fn : int * int -> int
- plus(1,3);
val it = 4 : int
```

Diesen Aspekt werden wir in Abschnitt 3.7.2 vertiefen.

3.2.4 Deklarationen

Alle Identifikatoren müssen vor ihrer Verwendung deklariert werden. Bei ihrer Deklaration werden sie zu einem Wert *gebunden*. Wesentlich ist, daß im *Gültigkeitsbereich* einer Deklaration dieser Wert fest ist und nicht, wie etwa in imperativen Sprachen, durch eine Wertzuweisung verändert werden kann. Die ML-Variablen sind also eher vergleichbar den **Konstanten** (vgl. **const**-Deklarationen in Pascal) imperativer Sprachen.

```
- val x = 4*5;
val x = 20 : int
- val s = "abc"^"def";
val s = "abcdef" : string
- val p = ( x, s );
val p = (20,"abcdef") : int * string
```

Phrasen der Form

```
        val <Identifikator> = <Ausdruck>
```

heißen *Wertedefinitionen*. Ihr Effekt ist die Bindung des Identifikators an den Wert der rechten Seite.

Die Redefinition eines Identifikators *verdeckt* die bereits bestehende Bindung in demselben Sinne, wie in einer inneren Prozedurdefinition in Pascal durch Redeklaration eine äußere Bindung verdeckt wird. Was der Unterschied zwischen einer Redefinition und einer Pascal-Wertzuweisung ist, werden wir später noch sehen,

```
- val x = 17;
val x = 17 : int
- val y = x;
val y = 17 : int
- val x = true;
val x = true : bool
- val z = x;
val z = true : bool
- val x = 17;
val x = 17 : int
- val x = true and y = x and w = z;
val x = true : bool
val y = 17 : int
val w = true : int
```

Folgene Regeln und Terminologie definieren die Semantik dieser Konstrukte:

- Deklarationen erzeugen *Bindungen* und Mengen von Bindungen bilden eine *Umgebung*. Es gibt verschiedene *Arten* von Deklarationen, z.B. für Werte `val` oder Funktionen `fun`; später sehen wir noch andere Arten.

- Die Auswertung eines jeden Ausdrucks erfolgt in einer bestimmten Umgebung, die dann die Werte für die Variablen des Ausdrucks festlegt.

- Die rechte Seite einer **Wert**deklaration wird in der **alten**, bisherigen Umgebung ausgewertet (der Wert des letzten `y` ist `17` und nicht `true`).

- Deklarationen können durch bestimmte Kombinatoren zusammengesetzt werden. So bedeutet `d1 ; d2` die *sequentielle Komposition*[5] von Deklarationen `d1` und `d2`. `d1` wird in der gegebenen Umgebung E ausgewertet. Der Effekt ist, daß die neuen Bindungen gemäß `d1` zu E hinzugefügt werden und alte Bindungen in E für redefinierte Identifikatoren verdeckt werden. In dieser neuen Umgebung erfolgt die Auswertung von `d2`, die dann in gleicher Weise die Umgebung modifiziert. Auf Interpreter-Eingabeebene sind also nacheinanderfolgende, durch „;“ getrennte Deklarationen als in diesem Sinn sequentiell zusammengesetzt zu betrachten.

- `and` trennt *parallele* Deklarationen von mehreren verschiedenen Identifikatoren. Die Definitionsrümpfe werden alle in der alten Umgebung ausgewertet.

- Bei Funktionsdefinitionen

  `fun f` m_1 `=` e_1 `| ... | f` m_n `=` e_n

  gelten folgende Bindungsregeln. Die Umgebung Hier können die Definitionen rekursiv sein und die neuen Bindungen für `f` und für die Variablen in den Mustern m_i sind bereits in e_i sichtbar. Letztere sind nur lokal in den e_i sichtbar.

Der Gültigkeitsbereich von Deklarationen kann lokal beschränkt werden.

```
- local val x = 10
= in
=       val u = x*x + x*x
=       val v = 2*x + (x div 5)
= end;
val u = 200 : int
val v = 22 : int
```

`local` erzeugt lokale Deklarationen für eine (zusammengesetzte) Deklaration zwischen `in` und `end`.

```
- let val x = 10
= in
=     x*x + 2*x +1
= end;
val it = 121 : int
```

`let` erzeugt lokale Deklarationen für die Auswertung eines Ausdrucks zwischen `in` und `end`.

[5]Das Semikolon ist optional. Man kann auch einfach `d1` und `d2` hintereinanderschreiben.

3.2.5 Muster

Wir haben schon bei den Funktionsdefinitionen gesehen, daß die Verwendung von Mustern die einzige Möglichkeit ist, um auf Substrukturen von Konstruktortermen zuzugreifen. Muster sind Konstruktorterme, die Variablen enthalten. Es ist nur natürlich, Muster auch in Wertdeklarationen zuzulassen. D.h. Wertdeklarationen haben die allgemeinere Form

```
val <Muster> = <Ausdruck>,
```

bei der der Ausdruck in der gegebenen Umgebung ausgewertet wird, das Muster des entstehenden Wertes gestestet wird und im Erfolgsfall die Variablen des Musters an die entsprechenden Subterme gebunden werden.

```
- val x = (17,true);
val x = (17,true) : int * bool
- val (left,right) = x;
val left = 17 : int
val right = true : bool
- val (left,false) = x;
std_in:48.1-48.20 Warning: binding not exhaustive
          (left,false) = ...

uncaught Bind exception std_in:48.1-48.20
```

- Warnung, wenn Muster nicht alle Möglichkeiten (diese sind nach der Typinferenz bekannt) abdeckt;
- Ausnahmebehandlung wird angestoßen, wenn Muster nicht paßt.
- Alle Muster müssen *linear* sein, d.h. Mehrfachvorkommen von Variablen sind nicht erlaubt.

Weitere Beispiele:

```
- val l = [1,2,3];
val l = [1,2,3] : int list
- val [x,y,z] = l;
std_in:49.1-49.15 Warning: binding not exhaustive
          x :: y :: z :: nil = ...
val x = 1 : int
val y = 2 : int
val z = 3 : int
- val r = {name="hugo", used = true};
val r = {name="hugo",used=true} : {name:string, used:bool}
- val {used=u, name=n} = r;
val n = "hugo" : string
val u = true : bool
- val {used=u, ...}=r;
val u = true : bool
- val u = #used r;
```

```
val u = true : bool
- fun uu {used = u, ...} = u;
std_in:55.1-55.26 Error: unresolved flex record in let pattern
  type: {used:'Y, '...Z}
```

- ... ist der Joker, der auf eine beliebige Anzahl von Verbundfeldern paßt.
- Bei Verbundtypen müssen alle Felder und deren Typen aus dem Kontext hervorgehen, eventuell müssen Typeinschränkungen verwendet werden. Den Polymorphismus, der in der objektorientierten Programmierung durch Verbunderweiterung erfolgt (Vererbung von Instanzvariablen), hat man in SML leider nicht. In einer ML-Variante von Cardelli gab es ihn einmal.
- `#name r` ist eine abkürzende Schreibweise für den Zugriff auf das Feld `name` eines Verbundes `r`.

Muster brauchen nicht flach zu sein, sondern können beliebig geschachtelte Terme sein.

```
- val l = [(1,["H","G"]),(2,["U","W"])];
val l = [(1,["H","G"]),(2,["U","W"])] : (int * string list) list
- val _::(key,_::second)::tl = l;
val key = 2 : int
val second = ["W"] : string list
val tl = [] : (int * string list) list
```

3.3 Operationale Semantik von erststufigem ML

In diesem Kapitel beschreiben wir die operationale Semantik eines Fragmentes von ML, in dem nur Funktionen erster Stufe prorgammiert werden können, also Funktionen als Argumente oder Ergebnisse von Funktionen nicht erlaubt sind. Volles ML mit Funktionen höherer Stufe werden wir in Abschnitt 3.10 kennenlernen, nachdem wir gesehen haben werden, wie Funktionen höherer Stufe in der Programmierung verwendet werden. Das erststufige Fragment von ML ist besonders einfach und läßt sich mit Standardmethoden der Termersetzung erster Stufe behandeln.

3.3.1 Mini-ML erster Stufe

Mini-ML erster Stufe (MML1) ist ein Ausschnitt von ML gemäß folgender Syntax:

p	$::=$	$d;e$	Programm
d	$::=$	fd_1 `and` ... `and` fd_n	Definitionsteil
fd	$::=$	`fun` $f\ m_1$ `=>` e_1 `|` ... `|` $f\ m_k$ `=>` e_k	Funktionsdefinition
m	$::=$	x	Variable
	$\mid$	$c(m_1, \ldots, m_k)$, c Konstruktor	Konstruktorapplikation
e	$::=$	x	Variable
	$\mid$	$g(e_1, \ldots, e_k)$	Applikation
	$\mid$	`if` e `then` e' `else` e''	Fallunterscheidung

wobei im Falle der Applikation g ein k-stelliger Konstuktor oder eine vordefinierte oder programmdefinierte Funktion sein darf. (Nachfolgend verwenden wir immer c als Metavariable für Konstruktoren, f für vordefinierte oder programmdefinierte Funktionen, g für Funktionen oder Konstruktoren und m für Muster.)

In einem MML1-Programm gibt es nur globale, eventuell verschränkt rekursive Definitionen von Funktionen, in denen dann ein Ausdruck auszuwerten ist. Alle Funktionen sind erster Stufe, und Applikationen sind beschränkt indem ihr Funktionsteil eine der deklarierten oder vordefinierten Funktionen sein muß. Die primitiven Typen sind wie in SML, unter den Konstruktoren seien auch alle Konstanten subsumiert. Um Problemen mit der Behandlung von Umgebungen aus dem Weg zu gehen, lassen wir nur einen globalen Definitionsteil d zu.[6] Komplexere, geschachtelte Definitionsstrukturen lassen sich durch Funktionen höherer Ordnung erklären, die wir zunächst nicht behandeln wollen. Daher gibt es auch keine allgemeinen λ-Ausdrucke in MML1.

MML1 unterliege denselben semantischen Nebenbedingungen wie in ML. So gehen wir von Typkorrektheit (vgl. Abschnitt 3.9) aus. Muster müssen linear sein, d.h., in einem Muster darf keine Variable mehrfach auftreten. Außerdem darf in d keine Funktion mehr als einmal definiert werden. MML1 ist daher eine echte Teilsprache von SML.

3.3.2 Ausdrucksauswertung als Reduktion

Erststufige Funktionsdefinitionen definieren Ersetzungsregeln, mit denen Ausdrücke reduziert werden können. Betrachten wir beispielsweise die Funktion

```
fun append (nil,l) = l
  | append (hd::tl,l) = hd :: append (tl,l)
```

von früher. Sie definiert die beiden Reduktionsregeln

```
append (nil,l)     -> l
append (hd::tl,l) -> hd :: append(tl,l) .
```

Betrachten wir die Reduktion eines einfachen Ausdrucks:

```
   append(1::nil,2::3::nil)
-> 1::append(nil,2::3::nil)
-> 1::2::3::nil
```

Um eine Regel $l \to r$ in einem Teilausdruck anzuwenden, braucht es die Existenx einer Substitution, unter der l und dieser Teilausdruck gleich werden. Eine *Substitution* σ ist eine endliche Abbildung von (paarweise verschiedenen) Variablen x_i auf Ausdrücke e_i, geschrieben

$$\sigma = [e_1/x_1, \ldots, e_n/x_n] \ .$$

Die *Anwendung* einer Substitution σ auf einen Ausdruck e, bezeichnet mit $e\sigma$, liefert den Ausdruck e', der aus e durch Ersetzen eines jeden Auftreten eines x_i durch e_i entsteht. Eine Reduktionssregel ist ein Paar von Termen l und r, geschrieben $l \to r$. Die Regel *paßt* auf einen Teilausdruck e von $e_1 = e_1[e]$, wenn es eine Substitution σ für die Variablen in l gibt, so daß $l\sigma = e$. σ heißt die *Anpaßsubstitution* („matcher“) für l auf e. Dann nennen wir $e_2 = e[r\sigma]$ das Resultat der *Anwendung* der Regel am bezeichneten Auftreten von e in e_1. In diesem Fall schreiben wir auch $e_1 \to e_2$ und sprechen von der *Reduktion* von e_1 zu e_2. Das dabei reduzierte Auftreteten von e in e_1 heißt der *Redex* dieses Reduktionsschrittes.

[6] Bei der parallelen Komposition von Funktionsdeklarationen (mit `and`) sind in jeder rechten Seite einer jeden Regel alle deklarierten Funktionen sichtbar. In der SML-Programmierung, wo man auch sequentiell verknüpfte Definitionen schreiben kann, wird `and` insbesondere dann verwendet, wenn Definitionen verschränkt rekursiv sind.

Ein *Reduktionssystem* R ist eine Menge von Reduktionsregeln. Wir schreiben $e \rightarrow_R e'$, wenn e' aus e durch Anwendung einer Regel aus R auf e entsteht. Mit $\rightarrow_R^+$ bzw. $\rightarrow_R^*$ bezeichnen wir die transitive bzw. transitive und reflexive Hülle dieser durch R definierten *Redunktionsrelation* $\rightarrow_R$. Wir sagen, daß ein Ausdruck e *irreduzibel* ist bzgl. R, wenn für keinen Ausdruck e' $e \rightarrow_R e'$ gilt. e' heißt eine *Normalform* von t bzgl. R, wenn $e \rightarrow_R^* e'$ und wenn e' irreduzibel ist.

Im ersten Schritt der obigen Reduktionsfolge wurde die erste Regel, im zweiten Schritt die zweite Regel angewendet, und zwar jeweils an der Spitze des Ausdrucks. Folgendes Beispiel ist etwas komplizierter, weil die definierte Funktion `append` mehrfach vorkommt.

```
   append(1::nil,append(2::nil,2::3::nil))
-> append(1::nil,2::append(nil,2::3::nil))
-> append(1::nil,2::2::3::nil)
-> 1::append(nil,2::2::3::nil)
-> 1::2::2::3::nil
```

Im Prinzip wäre auch folgende Reduktionsfolge möglich gewesen:

```
   append(1::nil,append(2::nil,2::3::nil))
-> 1::append(nil,append(2::nil,2::3::nil))
-> 1::append(nil,2::append(nil,2::3::nil))
-> 1::append(nil,2::2::3::nil)
-> 1::2::2::3::nil
```

Wir sehen, daß das Ergebnis das gleiche ist, unabhängig von der Reduktionsstrategie. In SML ist die zweite Reduktionsfolge nicht erlaubt. SML-Reduktion erfolgt nach der Strategie "**innerste Redexe zuerst**". Der erste Schritt in der zweiten Reduktionsfolge wäre daher in SML nicht erlaubt, da es ein weiter innen liegendes Auftreten von `append` gibt, welches ebenfalls reduzierbar ist. Die Argumente einer Funktion müssen daher zuerst zu Konstruktortermen normalisiert werden, bevor irgendeine Regel der jeweiligen Funktionsdefinition angewendet wird. Unter einem *Konstuktorterm* verstehen wir einen Ausdruck, in dem nur Konstruktoren (die Konstanten der Typen `int`, `real`, `string`, und `bool` sowie die Konstruktoren für Tupel, Verbunde und Listen) vorkommen dürfen. Wir werden später sehen, daß der ML-Programmierer weitere Konstruktoren deklarieren darf, also nicht notwendig alle rekursiven, strukturierten Daten als Listen kodieren muß.

Zusammengenommen können wir für die MML1-Teilsprache von SML folgende einfache *Reduktionssemantik* definieren. Gegeben sei ein Programm $p = d\ \texttt{;}\ e$.

1. Aus den Funktionsdefinitionen in d der Form

   $$\texttt{fun}\ f\ m_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ f\ m_k\ \texttt{=>}\ e_k$$

   werden Termersetzungsregeln

   $$\begin{array}{rcl} (f\ m_1) & \rightarrow & e_1 \\ & \ldots & \\ (f\ m_k) & \rightarrow & e_k\ . \end{array}$$

2. Die Semantik von p ist das Ergebnis der Reduktion von e

- mit den Regeln aus d,
- mit den Regeln

$$\begin{array}{rcl} \texttt{if true then } e' \texttt{ else } e'' & \rightarrow & e' \\ \texttt{if false then } e' \texttt{ else } e'' & \rightarrow & e' \end{array}$$

  für Fallunterscheidungen und
- mit den Regeln zur Reduktion von Applikationen vordefinierter Funktionen, wie z.B.

$$\begin{array}{rcl} 3+5 & \rightarrow & 8 \\ 3<5 & \rightarrow & \texttt{true} . \end{array}$$

3. Die Anwendung der Regeln erfolgt unter Berücksichtigung der Strategie "innerste Redexe zuerst", d.h.,

   - immer der am weitesten links innen stehende Redex wird reduziert, **ausgenommen** Redexe im Ja- und Neinfall von Fallunterscheidungen, die **nicht** reduziert werden dürfen,
   - immer die erste (von links her) passende Regel einer Funktionsdefinition wird verwendet, wenn mehr als eine Regel paßt.

4. Die Auswertung ist fehlerhaft, wenn die sich ergebende Normalform kein Konstruktorterm ist, also noch Aufrufe deklarierter Funktionen enthält, die nicht reduziert werden können. In diesem Fall ist die jeweilige Funktionsdefinition unvollständig, die Muster in den Regeln decken nicht alle Fälle ab.

5. Mit $value(e)$ bezeichnen wir das Ergebnis dieser Reduktion von e, falls es existiert, d.h., falls die Reduktion terminiert und nicht mit Fehler endet.

Das Ergebnis von p ergibt sich also durch Reduktion von e mit der Strategie "innerste Redexe zuerst". Konsequenz dieser Reduktionsstrategie ist die *Striktheit* aller benutzerdefinierten Funktionen f. Es gilt

$$value(f(e_1, \ldots, e_n)) = value(f(value(e_1), \ldots, value(e_n))) \tag{1}$$

und als Konsequenz ist $value(f(e_1, \ldots, e_n))$ undefiniert, sofern eines der $value(e_i)$ undefiniert ist. Für Konstruktoren gibt es keine Reduktionsregeln, also gilt:

$$value(c(e_1, \ldots, e_n)) = c(value(e_1), \ldots, value(e_n)) \tag{2}$$

Natürlich ist die $value$-Funktion idempotent, d.h.

$$value(value(e)) = value(e) \tag{3}$$

Die Anwendung von Reduktionsregeln bewahrt die Semantik. Für jeden Redex gilt, wenn $l \rightarrow r$ die dort unter Substitution σ anwendbare Regel ist,

$$value(e[l\sigma]) = value(e[r\sigma]) . \tag{4}$$

Diese Gleichung gilt für jeden Teilausdruck $l\sigma$ von e, wenn die Definition der Funktion f, aus der die Regel $l \rightarrow r$ stammt, keine zwei Regeln mit sich überlappenden Mustern enthält. Ansonsten gilt diese Gleichung nur dann, wenn der bezeichnete Teilausdruck $l\sigma$ von e gemäß der angegebenen Strategie wirklich ein Redex ist, und wenn man mit der ersten passenden Regel reduziert.

Die SML-Semantik garantiert, daß Normalformen, sofern sie existieren, immer eindeutig bestimmt sind, die Reduktionsrelation ist *konfluent*. Dies ist der Fall, weil die Regeln geordnet sind, so daß, wenn mehr als eine Regel anwendbar ist, immer die erste anwendbare Regel verwendet wird. In jedem Ausdruck gibt es also höchstens einen Redex und genau eine Regel, um diesen zu reduzieren.

3.4 Typdefinitionen

Typabkürzungen
Hier werden neue Namen als Abkürzungen für komplexe, evtl. parametrisierte Typausdrücke eingeführt. Hier gibt es keinen speziellen semantischen Gehalt. Diese Typen werden auch *transparente Typen* genannt. Alle Auftreten des neuen Typnamens sind äquivalent zu seinem definierenden Ausdruck.

```
- type intpair = int * int;
type  intpair = int * int
- fun f(x:intpair) = let val (l,r)=x in l end;
val f = fn : intpair -> int
- f(3,2);
val it = 3 : int
- type 'a pair = 'a * 'a;
type  'a pair = 'a * 'a
- type boolpair = bool pair;
type  boolpair = bool pair
```

Neue Datentypen
In Datentypdeklarationen können neue (rekursive, polymorphe) Typen von Konstruktortermen definiert werden. Folgende Definitionen führen 3 nullstellige Konstruktoren für einen neuen Typ `color` ein bzw. zeigen, wie man den vordefinierten Typ `bool` selbst definieren könnte.

```
- datatype color = red | green | blue;
datatype  color
  con blue : color
  con green : color
  con red : color
```

Durch das Symbol `con` zeigt der SML-Interpreter an, daß mit der Typdedinition nun auch neue Konstuktoren (in diesem Fall nullstellige) definiert wurden, die nun auch in Konstuktorapplikationen zur Konstruktion von Termen verwendet werden dürfen.

```
- red;
val it = red : color
- datatype bool = true | false;
datatype  bool
  con false : bool
  con true : bool
```

```
val like = fn : color -> bool
- fun like red = true
=    | like green = false
=    | like blue = true;
val like = fn : color -> bool
- like blue;
val it = true : bool
```

Wir sehen, daß Typdefinitionen dieser Art den Aufzählungstypen von Pascal entsprechen. Konstruktoren müssen nicht nullstellig sein:

```
- type  rbg_color = color;
type  rbg_color = color
- datatype cmy_color = cyan | magenta | yellow;
datatype  cmy_color
  con cyan : cmy_color
  con magenta : cmy_color
  con yellow : cmy_color
- datatype color = rbg of rbg_color | cmy of cmy_color;
datatype  color
  con cmy : cmy_color -> color
  con rbg : rbg_color -> color
```

Hier ist `rbg` ein Konstruktor vom Typ `rbg_color -> color` und `cmy` ein Konstruktor vom Typ `cmy_color -> color`. Zusammen bilden sie alle Möglichkeiten, Terme vom (redefinierten) Typ `color` zu konstrieren.

Wir kommen nun zu *rekursiven Typen* und definieren als erstes (unmarkierte) Binärbäume.

```
- datatype btree = empty | leaf | node of btree * btree;
datatype  btree
  con empty : btree
  con leaf : btree
  con node : btree * btree -> btree
```

Hier wird ein neuer Typ `btree` eingeführt, und zwar als die Menge der Terme, die mit den Konstruktoren `empty`, `leaf`, und `node` bilden kann. `node` nimmt als Argumente zwei Terme vom Typ `btree`. Weil bei der Typdefinition des Konstruktors `btree` wieder Bezug auf den zu definierenden Typ genommen wird, spricht man von einem rekursiven Datentyp. Die Lösung dieser Rekursionsgleichung ist die kleinste Menge **endlicher** Terme $BTREE$, die die Gleichung erfüllt, also im obigen Beispiel die Konstanten `empty` und `leaf` enthält sowie jeden term, dessen Wurzel den Konstruktor `node` trägt und zwei $BTREE$-Unterterme besitzt.

Es ist nur natürlich, daß Funktionen auf rekursiven Datentypen selbst rekursiv sind und daß die Rekursionen von Typ- und Funktionsdefinition oft parallel verlaufen, siehe das folgende Beispiel. Mathematisch gesehen steht hier das Prinzip der *strukturellen Induktion* im Raum, welches wir im Abschnitt 3.5 rekapitulieren wollen.

```
- fun countleaves empty = 0
    | countleaves leaf = 1
```

```
    | countleaves (node(subtree1,subtree2)) =
                 countleaves (subtree1)+countleaves (subtree2);
val countleaves = fn : btree -> int
- countleaves (node(node(leaf,empty),leaf));
val it = 2 : int
```

Die im Abschnitt 3.3 definierte Reduktionssemantik für MML1 ist direkt auf MML1 mit Datentypdefinitionen anwendbar. Durch die Datentypdefinitionen in einem Programm ändert sich nur die Menge der Konstruktoren, die man zur Bildung von Konstruktortermen verfügbar hat. So zeigt

```
   countleaves (node(node(leaf,empty),leaf))
-> (countleaves (node(leaf,empty))) + (countleaves leaf)
-> ((countleaves leaf) + (countleaves empty))+ (countleaves leaf)
-> (1 + (countleaves empty))+ (countleaves leaf)
-> (1 + 0)+ (countleaves leaf)
-> 1 + (countleaves leaf)
-> 1 + 1
-> 2 .
```

daß die obige Ausgabe `val it = 2 : int` des SML-Interpreters korrekt war.

Als nächstes Beispiel zeigen wir, wie wir die Listen selbst definieren können.

```
- datatype 'a list = nil | :: of 'a * 'a list;
datatype  'a list
  con :: : 'a * 'a list -> 'a list
  con nil : 'a list
```

Daß es sich hier um einen polymorphen Typ handelt, wird durch den Typparameter `'a` auf der linken Seite der Typdeklaration angezeigt. `list` ist also ein *Typkonstruktor*, der aus Argumenttypen α Listentypen α *list* konstruiert. Hier, als weiteres Beispiel, eine polymorphe Variation der Binärbäume so, daß sie an Blättern Information tragen.

```
- datatype 'a tree = empty | leaf of 'a | node of 'a tree * 'a tree;
datatype  'a tree
  con empty : 'a tree
  con leaf : 'a -> 'a tree
  con node : 'a tree * 'a tree -> 'a tree
- fun frontier empty = []
    | frontier (leaf(x)) = [x]
    | frontier (node(t1,t2)) = append (frontier(t1),frontier(t2));
val frontier = fn : 'a tree -> 'a list
- frontier (node(leaf("a"),node(node(leaf("b"),empty),leaf("a"))));
val it = "a" :: "b" :: "a" :: nil : string list
```

Datentypdefinitionen können verschränkt rekursiv sein, wie in folgendem Beispiel, in dem eine Datenstruktur für arithmetische Ausdrücke definiert wird, wie sie in einem Übersetzer Verwendung finden würde (vgl. auch Abschnitt 3.8.1).

```
- datatype aexp = sum  of aexp * aexp | summand of mexp
       and mexp = prod of mexp * mexp | faktor of fexp
```

```
       and fexp = var of string       | exp of aexp;
datatype  aexp
  con sum : aexp * aexp -> aexp
  con summand : mexp -> aexp
datatype  mexp
  con faktor : fexp -> mexp
  con prod : mexp * mexp -> mexp
datatype  fexp
  con exp : aexp -> fexp
  con var : string -> fexp
```

Ein arithmetischer Ausdruck, wie $x * (y + z)$, wäre durch den Konstruktorterm

```
- summand(prod(faktor(var("x")),
               faktor(exp(sum(summand(faktor(var("y"))),
                              summand(faktor(var("z"))))))));
val it = summand (prod (faktor (var #),faktor (exp #))) : aexp
```

repräsentiert. In solchen Fällen findet man oft auf der Seite der Funktionen verschränkt rekursive Funktionen zur Manipulation solcher Terme. Wie im Falle von durch and parallel zusammengesetzten Funktionsdefinitionen, sind alle Typnamen aller linken Seiten in allen rechten Seiten der Typdefinitionen sichtbar.

Abstrakte Datentypen

Abstrakte Datentypen verstecken die Implementierungsstruktur von Daten. Hierdurch werden Impementierungsaspekte unabhängig von der Verwendung des Typs. Insbesondere kann die Implementierung geändert werden, ohne daß Anwendungen reimplementiert werden müssen, solange die *exportierten Funktionen* des Typs, genannt die *Schnittstelle*, sich in ihren *beobachtbaren Eigenschaften* nicht ändern. Wir machen ein Beispiel, in dem 4-Bit RGB-Farben definiert werden.

```
abstype color = blend of int*int*int
with val black = blend(0,0,0)
      and   red = blend(15,0,0)
      and green = blend(0,15,0)
      and  blue = blend(0,0,15)
      fun mix(parts:int,blend(r,g,b),parts':int,blend(r',g',b')) =
      if parts<= 0 orelse parts'<= 0 then black
      else let val tp=parts+parts'
               val rp = ((parts*r+parts'*r') div tp) mod 16
               and gp = ((parts*g+parts'*g') div tp) mod 16
               and bp = ((parts*b+parts'*b') div tp) mod 16
              in blend(rp,gp,bp)
             end
end;
type color
val white = - : color
val red = - : color
val green = - : color
val blue = - : color
```

```
val mix = fn : int * color * int * color -> color
- val white = mix(1,red,1,mix(1,green,1,blue));
val white = - : color
```

Man beachte:

- Die Typgleichung hinter „`abstype`" hat dieselbe Syntax, wie die, die wir vorher kennengelernt haben. Also können hier auch neue Typkonstruktoren eingeführt werden.
- Die nach außen sichtbare Schnittstelle des abstrakten Datentyps besteht aus dem Namen des Typs und den Namen der zwischen `with` und `end` eingeführten Werte und Funktionen.
- Die Implementierung der Schnittstellenobjekte bleibt verborgen. Technisch bedeutet das insbesondere, daß der Typ verschieden ist von allen anderen Typen und daß seine Konstruktoren außen nicht sichtbar sind. Daher kann man insbesondere keine musterorientierten Funktionen auf einem abstrakten Type schreiben. Zudem ist die Gleichheit `=` nicht definiert für Werte eines abstrakten Datentyps. Soll es eine Gleichheitsfunktion geben, so muß man die implementieren und in der Schnittstelle zur Verfügung stellen.
- Innerhalb der Definition Abstrakten Datentyps sind die Konstruktoren natürlich sichtbar und können zu musterorientierten Definitionen verwendet werden.

Diese Art von abstrakten Datentypen in SML ist historisch bedingt und sollte angesichts der erweiterten Möglichkeiten von Abstraktion mit Hilfe der Konzepte von Strukturen und Funktoren eigentlich nicht mehr verwendet werden.

3.5 Strukturelle Induktion

Gegeben sei eine Menge von Funktionssymbolen F. Mit T_F bezeichnen wir die Menge der Terme mit Funktionssymbolen aus F. Strukturelle Induktion erlaubt, Eigenschaften aller Terme in T_F zu beweisen, indem folgende Schlußregel verwendet wird.

Theorem 1 (Prinzip der strukturellen Induktion) *Sei $P(x)$ eine Eigenschaft über Termen x in T_F. Wenn man für alle Funktionssymbole f in F die Implikation*

$$\forall t_1, \ldots, t_n : \mathsf{T}_F \ . \ P(t_1) \wedge \ldots \wedge P(t_n) \supset P(f(t_1, \ldots, t_n))$$

zeigen kann, so gilt $P(t)$ für alle t in T_F.

Wir geben ein Beispiel für seine Anwendung, welches die zentrale Bedeutung dieses Prinzips zur Verifikation funktionaler Programme zeigt. Betrachte folgende Funktion, die Binärbäume rekursiv spiegelt.

```
- fun swap empty = empty
=   | swap leaf = leaf
=   | swap(node(t1,t2))  = node(swap(t2),swap(t1));
val swap = fn : btree -> btree
- countleaves(swap(node(node(leaf,leaf),leaf)));
val it = 3 : int
```

Wir beweisen, daß für alle Konstruktorterme t vom Typ `btree`

$$value(\texttt{countleaves(swap(t))}) = value(\texttt{countleaves(t)}).$$

In diesem Fall besteht also unser F aus den Funktionen `empty`, `leaf` und `node`. Wir haben daher drei Fälle zu untersuchen:

(i) $f =$ `empty` : Weil `empty` ein nullstelliger Konstruktor ist, ist das n im Antezedent der zu beweisenden Implikation 0, und es bleibt zu zeigen, daß

$$value(\texttt{countleaves(swap(empty))}) = value(\texttt{countleaves(empty)}).$$

welches sich direkt durch Reduktion von `swap(empty)` zu `empty` mit der ersten Regel von `swap` ergibt.

(ii) $f =$ `leaf`: Hier ist n ebenfalls 0 und es gilt

$$value(\texttt{countleaves(swap(leaf))}) = value(\texttt{countleaves(leaf)}).$$

durch Reduktion von `swap(leaf)` zu `leaf` mit der zweiten Regel von `swap`.

(iii) $f =$ `node : btree * btree -> btree`: Hier ist $n = 2$. Wir nehmen an, um die Implikation zu zeigen, daß

$$value(\texttt{countleaves(swap(}t_i\texttt{))}) = value(\texttt{countleaves(}t_i\texttt{)}).$$

für beliebige t_i in T_F, $i = 1, 2$. (Diese Annahme nennt man die *Induktionsannahme.*) Nun berechnen wir (wobei wir *value* zu v abkürzen)

$$\begin{array}{lll}
& v(\texttt{countleaves(swap(node(}t_1, t_2\texttt{))))} & \\
= & v(\texttt{countleaves(node(swap(}t_2\texttt{), swap(}t_1\texttt{))))} & \text{(Regel 3 von \texttt{swap})} \\
= & v(\texttt{countleaves(node(}v\texttt{(swap(}t_2\texttt{)),}v\texttt{(swap(}t_1\texttt{)))}) & \text{(Regeln für } v) \\
= & v(\texttt{countleaves(}v\texttt{(swap(}t_2\texttt{)))+countleaves(}v\texttt{(swap(}t_1\texttt{))))} & \text{(Regel 3 \texttt{countleaves})} \\
= & v(v(\texttt{countleaves(swap(}t_2\texttt{)))} + v(\texttt{countleaves(swap(}t_1\texttt{))))} & \text{(Regeln für } v) \\
= & v(v(\texttt{countleaves(}t_2\texttt{))} + v(\texttt{countleaves(swap(}t_1\texttt{))))} & \text{(Ind.-Annahme für } t_2) \\
= & v(v(\texttt{countleaves(}t_2\texttt{))} + v(\texttt{countleaves(}t_1\texttt{)))} & \text{(Ind.-Annahme für } t_1) \\
= & v(v(\texttt{countleaves(}t_1\texttt{))} + v(\texttt{countleaves(}t_2\texttt{)))} & \text{(+ ist kommutativ)} \\
= & v(\texttt{countleaves(}t_1\texttt{)} + \texttt{countleaves(}t_2\texttt{))} & \text{(+ ist strikt)} \\
= & v(\texttt{countleaves(node(}t_1, t_2\texttt{)))} & \text{(Regel 3 \texttt{countleaves})}
\end{array}$$

und erhalten die Gleichung, die zu zeigen war. Bzgl. der Regeln für *value* vgl. Gleichungen (1)–(4) in Abschnitt 3.3.2. In diesem Beispiel überlappen sich keine zwei verschiedenen Reduktionsregeln. Damit bewahren beliebige Regelanwendungen gemäß Gleichung (4) die Semantik eines Ausdrucks.

Das oben angegebene Schema von struktureller Induktion ist nicht praktikabel, wenn die Menge der Konstruktoren unendlich ist und dann es normalerweise auch mehr als einen Termtyp gibt. Nehmen wir daher an, daß Ty eine Menge von Termtypen ist und daß wir eine Eigenschaft P für die Terme dieser Typen per Induktion zeigen wollen. Wegen der Typisierung ist es sinnvoll anzunehmen, daß es für jeden Typ τ in Ty eine Spezialisierung $P_\tau(x_\tau)$ von P gibt, die für Terme x_τ vom Typ τ zuständig ist.

Theorem 2 (Prinzip der strukturellen Induktion: typisierte Variante) *Wenn man für jedes Funktionssymbol*

$$f : \tau_1 * \ldots * \tau_n * \sigma_1 * \ldots * \sigma_m \rightarrow \tau_0$$

in F mit Typen τ_i in Ty und Typen σ_j nicht in Ty die Implikation

$$\forall t_1 : \tau_1, \ldots, t_n : \tau_n,\ x_1 : \sigma_1, \ldots, x_m : \sigma_m\ .\qquad P_{\tau_1}(t_1) \wedge \ldots \wedge P_{\tau_n}(t_n) \ \supset\ P_{\tau_0}(f(t_1, \ldots, t_n, x_1, \ldots, x_m))$$

zeigen kann, so gilt für jedes τ in Ty die Eigenschaft $P_\tau(t)$ für alle Terme t vom Typ τ.

Diese Variante wendet man an, wenn `datatype`-Deklarationen mehr als einen Typ einführen, und wenn Konstuktorterme Blätter enthalten, die aus unendlichen Typen (Zahlen, Zeichenreihen, Funktionstypen) stammen. Als Beispiel sei folgende (etwas synthetische) Typdefinition betrachtet:

```
datatype S = f1 of S * int * T
     and T = f2 of S * T | g of (S -> T) *S;
```

Um Eigenschaften P solcher Terme zu beweisen, nimmt man zweckmäßigerweise $Ty = \{S, T\}$, spaltet P in P_S und P_T gemäß der 2 möglichen Termtypen auf, und somit belaufen sich die Beweisverpflichtungen gemäß Theorem 2 auf

$$\begin{array}{lclcl}
\forall s : S, n : int, t : T & . & P_S(s) \wedge P_T(t) & \supset & P_S(f1(s, n)) \\
\forall s : S, t : T & . & P_S(s) \wedge P_T(t) & \supset & P_T(f2(s, t)) \\
\forall h : (S \to T), s : S & . & P_S(s) & \supset & P_T(g(h, s))\ .
\end{array}$$

3.6 Ausnahmebehandlung

Ausnahmebehandlung unterstützt die Programmierung partieller Funktionen. Das Problem ist, sicherzustellen, daß auch im Falle des Aufrufs einer solchen Funktion mit einem Parameter außerhalb des Definitionsbereichs eine wohldefinierte, typsichere Berechnung erfolgt.

```
- exception Head;
exception Head
- fun head nil = raise Head
=   | head (x::l) = x;
val head = fn : 'a list -> 'a
- head nil;

uncaught exception Head
```

Ausnahmeindikatoren, wie `Head`, werden durch eine Deklaration der Form

`exception <Id>` ≪ `of <Typ>` ≫ [7]

eingeführt. Will man im Ausnahmefall neben dem Indikator `<Id>` auch noch einen Wert vom Typ `<Typ>` mitgeben, so verwdndet man die erweiterte Variante der Deklaration. Bei Auswertung von Durch

`raise <Id>` ≪ `<Ausdruck>` ≫

wird eine Ausnahmewert von der Form `<Id>` bzw. `<Id>`(x) erzeugt, wenn x der Wert des Ausdrucks `<Ausdruck>` ist.

[7] Mit ≪ und ≫ kennzeichnen wir optionale Syntax.

```
- head nil;

uncaught exception Head
```

Der Effekt ist, falls keine andere Behandlung definiert ist, daß die Auswertung mit einer entsprechenden Fehlermeldung abgebrochen wird. Eine Ausnahmebehandlung kann explizit programmiert werden. Die zugehörige Konstruktion hat die Form

$$e \texttt{ handle } M_1 \texttt{ => } e_1 \mid \ldots \mid M_k \texttt{ => } e_k$$

mit Ausdrücken e und e_i sowie Mustern M_i. Deren Auswertung ist wie folgt definiert:

(i) Werte e aus; ergibt sich ein Wert, so ist dies der Wert der gesamten Konstruktion.

(ii) Tritt eine Ausnahmesituation mit einem Ausnahmewert a ein, der auf eines der Muster paßt, so nehme das erste unter den passenden Mustern, sagen wir M_i, erstelle die durch die Anpaßsubstitution gegebenen Bindungen für die Variablen in M_i und werte den zugehörigen Ausdruck M_i aus. Das Ergebnis dieser Auswertung ist das Ergebnis der gesamten Konstruktion.

(iii) Andernfalls liefere den Ausnahmewert a als Ergebnis ab, der dann vielleicht in einer äußeren Ausnahmebehandlung behandelt wird.

```
- fun head2 l = head(l) handle Head => 0;
val head2 = fn : int list -> int
- head2 [2,3,4];
val it = 2 : int
- head2 [];
val it = 0 : int
```

In obigem Beispiel sieht man die Typsicherheit dieser Konstruktion. Während `head` den polymorphen Typ $\tau\,list \to \tau$ hat, hat `head2` wegen der speziellen Ausnahmebehandlung den Typ $int\,list \to int$.

In Ausnahmeindikatoren sollte man nicht nur eine Notation für spezielle Fehlerwerte sehen, sondern ein Mittel, daß es dem Programmierer gestattet, auf das explizite Einpacken von Werten zu verzichten — eine Technik, mit der man ansonsten Fehlerbehandlung programmieren würde. Hierbei würde man für eine partielle Funktion $f : t' \to t$ zunächst einen neuen Typ

```
datatype tt = success of t | failure
```

definieren müssen und dann, wenn $f(x)$ definiert ist, `success(`x`)` zurückliefern, andernfalls `failure`. Dies bedeutet aber, daß bei jeder gewünschten Komposition von f mit einer Funktion $g : t \to \tilde{t}$, etwas in einem Ausdruck der Form $g(f(x))$, zunächst mal die Typisierung nicht paßt. Also muß g auf den Typ tt erweitert werden, was bedeutet, daß im Kode für g zunächst das Ergebnis von f ausgepackt werden muß, bevor es an die eigentliche Berechnung von g übergeben werden kann. `failure` müßte in irgendeinen „speziellen“ Wert aus t kodiert werden, oder aber es müßte auch der Ergebnistyp von g entsprechend um Fehlerwerte erweitert werden, selbst dann, wenn g eine totale Funktion ist. Möchte man g nicht ändern, muß man an jeder Aufrufstelle von f eine Spezialbehandlung der Fehlersituation vornehmen. In SML schreibt man einfacher `raise Ex`, wenn $f(x)$ undefiniert ist, und kann sich eine geeignete Fortsetzung für f an der Stelle x, sagen wir zu x', aus dem Aufrufkontext durch Ausnahmebehandlung `f(`x`)`

`handle Ex =>` x' konstruieren. In jedem Fall ist aber `g (f` x`)`, auch bei Nichtbehandlung der möglichen Fehlersituation, ein sinnvoller Ausdruck, bei dem jeder eventuell auftretende Ausnahmewert einfach automatisch soweit nach außen propagiert wird, bis eine Ausnahmebehandlung definiert ist. Wir werden nichttriviale Verwendungen dieses Mechanismus in Abschnitt 3.8.1 bei der Programmierung eines Simplifikators für arithmetische Ausdrücke kennenlernen.

3.7 Funktionen höherer Ordnung

3.7.1 Partiell applizierte Funktionen

Ein wesentliches Konzept funktionaler Programmiersprachen ist, daß Funktionen wie Werte behandelt werden können. Man sagt, Funktionen sind in diesen Sprachen „Bürger erster Klasse“.

Folgendes Beispiel definiert eine Funktion `map`, die eine Funktion f als Argument nimmt und als Ergebnis die Funktion liefert, die die elementweise Fortsetzung von f auf Listen darstellt.

```
- fun map _ nil = nil | map f (x::l) = (f x)::(map f l)
val map = fn : ('a -> 'b) -> 'a list -> 'b list
- fun succ n : int = n+1;
val succ = fn : int -> int
- map succ [2,4,6];
val it = [3,5,7] : int list
```

Der Typ von `map` ist also $(\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$. Oben haben wir die Instanz für $\alpha = \beta = \texttt{int}$ aufgerufen und die Funktion `succ` auf die Liste $[2,4,6]$ fortgesetzt.

Der Typ aller Funtionen von α nach β wird mit $\alpha \to \beta$ bezeichnet. Wir nehmen an, daß $\to$ rechtsassoziativ ist, d.h. daß $(\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$ für $(\alpha \to \beta) \to (\alpha\,\texttt{list} \to \beta\,\texttt{list})$ steht, wie dies für `map` gewünscht ist.

Funktionen, die Funktionen als Ergebnis liefern, werden i.allg. so definiert:

$$\begin{array}{lll} \texttt{fun} & & f\ m_{11} \ldots m_{1n}\ \texttt{=>}\ e_1 \\ & | & f\ m_{21} \ldots m_{2n}\ \texttt{=>}\ e_2 \\ \ldots & & \\ & | & f\ m_{k1} \ldots m_{kn}\ \texttt{=>}\ e_k \end{array}$$

Der Typ von f ist dann $t_1 \to \ldots \to t_n \to t_{n+1}$, falls die Muster m_{ij} den Typ t_j haben, $1 \leq j \leq n$, und die e_i vom Typ t_{n+1} sind. Hinsichtlich Funktionsapplikation erinnern wir daran, daß diese in SML die Form $(e_1\ e_2)$ haben, wobei e_1 ein **beliebiger** Ausdruck (nicht nur eine Funktionsvariable) sein darf, der eine Funktion als Ergebnis liefert, für die e_2 ein typmäßig passendes Argument ist, Für *voll applizierte* Aufrufe gibt es zunächst keinen Unterschied zwischen einer Funktion f der Form

$$\texttt{fun}\ f\ m_{11} \ldots m_{1k_1}\ \texttt{=>}\ e_1 \mid \ldots$$

und einer Funktion $\hat{f}$ der Form

$$\texttt{fun}\ \hat{f}\ (m_{11}, \ldots, m_{1k_1})\ \texttt{=>}\ e_1 \mid \ldots\ ;$$

$((\ldots(f\ e_1)\ \ldots)\ e_n)$ und $\hat{f}(e_1,\ldots,e_n)$ liefern dasselbe Ergebnis.[8] Diesem Umstand liefert eine einfache Technik, Funktionen mit n Argumenten als Funktionen mit einem Argument, die $n-1$-stellige Funktionen als Ergebnis nehmen, zu programmieren. Man nennt diese Technik *Curryfizieren*:[9] f heißt die curryfizierte Form von $\hat{f}$. Deswegen braucht es in einer funktionalen Programmiersprache auch nicht unbedingt Tupeltypen, so wie in SML.

Die wirkliche Mächtigkeit von Funktionen höherer Ordnung liegt in der Möglichkeit der partiellen Applikation. Betrachte

```
- fun plus x y = x+y:int;
val plus = fn : int -> int -> int
```

Folgendes Beispiel zeigt den Unterschied zwischen `plus` und `+`:

```
-  map (plus 1) [2,4,6];
val it = [3,5,7] : int list
```

Die Successorfunktion haben wir hier einfach durch die partielle Applikation `(plus 1)` repräsentiert. (Wir nennen eine Applikation *partiell*, dann wenn das Ergebnis von funktionalem Typ ist, also noch auf weitere Argumente appliziert werden darf.) Hingegen wäre die Applikation $(+\ 1)$ typmäßig unzulässig: $+$ erwartet Paare von Zahlen als Argumente.

Was die operationale Semantik anbelangt, so sind dann, wenn man nur globale Definitionen vorsieht, also nach wie vor λ-Ausdrücke nicht überall zuläßt, die Konzepte aus Abschnitt 3.3 ausreichend. Der einzige Unterschied ist, daß Klauseln in Funktionsdefinitionen mehr als ein Muster haben, und daß Applikationen allgemenier sind: der Funktionsteil darf ein beliebiger Ausdruck sein. Eine Klausel der Form

$$f\ \ m_{11}\ldots m_{1k_1}\ \ \texttt{=>}\ \ e_1$$

definiert eine Reduktionsregel

$$((\ldots(f\ m_{11})\ \ \ldots)\ m_{1k_1}) \to e_1$$

die also erst dann anwendbar wird, wenn alle k Argumente für f bereits vorliegen. Normalformen sind jetzt, zusätzlich zu Konstruktortermen, alle nicht voll applizierten Aufrufe, sofern ihre Teilterme in Normalform sind. Hier ein Reduktionsbeispiel:

```
map (plus 1) [2]
= map (plus 1) (2::nil)
→ ((plus 1) 2)::(map (plus 1) nil)
→ (1+2)::(map (plus 1) nil)
→ 3::(map (plus 1) nil)
→ 3::nil
= [3]
```

Im nächsten Beispiel ergibt sich ein funktionales Ergebnis:

```
map (plus (1+3))
→ map (plus 4))
```

[8] Beachte, daß Applikation, dual zu $\to$, linksassoziativ ist, d.h. daß man für $((\ldots(f\ e_1)\ \ldots)\ e_n)$ einfach $f\ e_1\ \ldots\ e_n$ schreiben kann.

[9] benannt nach dem Mathematiker und Logiker H. B. Curry

3.7.2 β-Reduktion

Funktionen müssen nicht unbedingt Namen haben, Namenlose Funktionen werden durch λ-Ausdrücke definiert. Die einfachste Form eines λ-Ausdrucks in SML ist

$$\texttt{(fn } x \texttt{ => } e\texttt{)} \ ,$$

wobei x eine Variable und e ein beliebiger Ausdruck ist. (In der Logik schreibt mein für solche Ausrücke $\lambda x.e$, daher der Name. Im folgenden werden wir diese Schreibweise auch immer wieder verwenden.)

λ-Ausdrücke dürfen überall dort stehen, wo Ausdrücke, die Werte liefern, stehen dürfen, insbesondere dürfen sie als erster Operand einer Applikation auftreten. In unserer operationalen Semantik haben wir dort bisher nur Funktionsvariablen und partielle applizierte Funktionsvariablen zugelassen. Jetzt lassen wir dort jeden Ausdruck von funktionalem Typ zu. In diesem Abschnitt wird erklärt, wie solche allgemeinen Applikationen reduziert werden.

In einer Applikation $(f\ e)$ werden zunächst die beiden Operanden f und e reduziert. Ergibt sich für f eine partiell applizierte Funktionsvariable, so wird verfahren, wie im letzten Abschnitt beschrieben. Ergibt sich für f ein λ-Ausdruck obiger Form, wird die Applikation in folgender Weise *β-reduziert*:

$$\texttt{(fn } x \texttt{ => } e\texttt{)}\ e' \quad\longrightarrow\quad e[e'/x]$$

wobei $e[e'/x]$ aus e durch Substitution von e' für alle **freien Auftreten** von x in e entsteht. Der Substitutionsbegriff ist in diesem Kontext komplexer als wir ihn für Musteranpassung bei Konstruktortermen kennengelernt haben. Es gibt im wesentlichen zwei Komplikationen:

Erstens werden nur freie Auftreten von x in e substituiert. Ein Auftreten von x in einem Ausdruck e ist *frei*, wenn es nicht ein Auftreten in einem Teilausdruck der Form $\texttt{(fn } x \texttt{ => } \tilde{e}\texttt{)}$ von e ist. Ein solches Auftreten heißt *gebunden*. Enthält e also innere λ-Ausdrücke, die x binden, so wird x in deren Rümpfen nicht ersetzt:

$$\texttt{(fn } x \texttt{ => (}x\texttt{ (fn } x \texttt{ => } x\texttt{)))}\ e' \quad\longrightarrow\quad \texttt{(}e'\texttt{ (fn } x \texttt{ => } e\texttt{))}$$

Desweiteren ist immer vorausgesetzt, daß bei der Substitution keine der freien Variable in e' nach der Einsetzung für x im dortigen Kontext gebunden wird. Gegebenenfalls muß man sonst gebundene Variablen in e umbenennen. Also, nicht gemeint ist etwa

$$\texttt{(fn } x \texttt{ => (fn } y \texttt{ => } y\ x\texttt{))}\ y \quad\longrightarrow\quad \texttt{(fn } y \texttt{ => } y\ y\texttt{)} \ ,$$

sondern stattdessen

$$\texttt{(fn } x \texttt{ => (fn } y \texttt{ => } y\ x\texttt{))}\ y \quad\longrightarrow\quad \texttt{(fn } y' \texttt{ => } y'\ y\texttt{)} \ ,$$

wobei wir zuerst im innernen λ-Ausdruck y in y' umbenannt haben, um den Variablenkonflikt zu vermeiden. Die Bindungen der freien Variablen in e' verändern sich also nicht, sondern bleiben so, wie es dem statischen Auftreten von $(f\ e')$ in einem größeren Ausdruck oder Programm entspricht.

In SML ist die allgemeine Form von λ-Ausdrücken komplexer. Anstelle einer Variablen x kann allgemeiner ein Muster M stehen. Dann haben wir als allgemeinere Form der β-Reduktion die Regel

$$\texttt{(fn } M \texttt{ => } e\texttt{)}\ e' \quad\longrightarrow\quad e[e'_1/x_1, \ldots, e'_n/x_n]$$

wenn x_i die in M auftretenden Variablen sind und unter Substitution

$$\sigma = [x_1 \mapsto e'_1, \ldots, x_n \mapsto e'_n]$$

das Muster M auf die Normalform von e' paßt, d.h. $M\sigma = value(e')$ gilt. Entsprechend wird bei mehr als einem Muster die Applikation so reduziert, daß die erste passende Alternative j ausgewählt wird:

$$\texttt{(fn } M_1 \texttt{ => } e_1 \texttt{ | ... | } M_k \texttt{ => } e_k\texttt{)}\ e' \quad \longrightarrow \quad e_j[e'_1/x_1, \ldots, e'_n/x_n]\ .$$

Die Reduktionsstrategie für SML muß um folgende wesentliche Bedingung erweitert werden: Rümpfe von λ-Ausdrücken werden grundsätzlich niemals reduziert. In Normalformen von Ausdrücken dürfen daher beliebige λ-Ausdrücke vorkommen. Man spricht von schwachen *Kopfnormalformen.*

Hat man allgemeine λ-Ausdrücke, so sind Funktionsdefinitionen äquivalent mit Wertedeklarationen, wo auf der rechten Seite λ-Ausdrücke verwendet werden. Etwa erzeugt die Auswertung von `fun f x y => e` die Bindung von f an `(fn x=> (fn y => e))`,[10] was aber keiner Erweiterung des Konzeptes der β-Reduktion bedarf.

MERKE:

- Ausdrücke von funktionalem Typ werden nur soweit reduziert, bis sich ein λ-Ausdruck ergibt. Ergibt sich eine Funktionsvariable, so wird der an sie gebundene λ-Ausdruck eingesetzt.

- Die diversen, komplexen Formen von Funktionsdefinitionen in SML erzeugen Bindungen von (Funktions-)Variablen an λ-Ausdrücke nach SML-Bauart (Übung!).

- Applikationen mit einem λ-Ausdruck als Funktion werden β-reduziert, wobei die gebundenen Variablen durch die (Komponenten des) aktuellen Parameter(s) ersetzt werden.

- Bei diesen Substitutionen muß darauf geachtet werden, daß keine Namenskonflikte vorliegen, die aus einer freien Variablen eine gebundene machen könnten. Andernfalls müssen gebundene Variablen geeignet umbenannt werden.

- Die Bindungen von freien Variablen werden nicht verändert.

- Theoretisch betrachtet käme man auch ohne λ-Ausdrücke aus, wenn man partiell applizierte Funktionen höherer Ordnung im Sinne von Abschnitt 3.7.1 hat. Wo immer ein λ-Ausdruck im Programm vorkommt, kann man eine benamte äquivalente Funktion f stattdessen global deklarieren und deren Namen an die Stelle des λ-Ausdrucks einsetzen. Komplizierter und von Hand mühsam wird das Ganze dann, wenn es freie (globale) Variablen im λ-Ausdruck gibt. Diese müssen dann nämlich als zusätzliche Argumente der Funktion simuliert werden, die an der Stelle des λ-Ausdruck in Form einer partiellen Applikation an f übergeben werden.

[10] Man beachte, daß in SML der wesentliche Unterschied zwischen einer `val`-Deklaration und einer `fun`-Deklaration der ist, daß erstere (leider?) nicht rekursiv ist, d.h. die rechte Seite e in `val x =` e in der alten Umgebung ausgewertet wird. Bei Funktions- (und `datatype`-) Definitionen ist dies anders. In `fun` f x `=` e ist die neue Bindung $f \equiv$ `(fn` x `=>` e`)` bereits in e sichtbar. Damit ist nur, wenn f in e nicht frei auftritt, `fun` f x `=` e mit `val f = (fn x=> (fn y => e))` äquivalent.

Diese Technik ist als λ-lifting in der funktionalen Programmierung bekannt und wird z.B. als Implementierungstechnik propagiert (Kombinatorreduktion, Superkombinatoren). In der Logik entspricht es der Transformation von Formeln mit Quantoren auf pränexe Normalform.

Reduktionsbeispiele werden wir im Anschnitt über die verzögernde Implementierung von Strömen sehen.

3.8 Programmiertechniken

3.8.1 Symbolische Manipulation

Wir zeigen in diesem Abschnitt anhand von drei Beispielen, wie mittels der SML-Konzepte Konstruktorterme und regelorientierte Funktionsdefinition, symbolische Manipulation von Daten klar und knapp formuliert werden kann.

Übersetzerbau.
Wir schreiben einen Übersetzer für eine einfache Programmiersprache für while-Programme. Die abstrakte Syntax der Quellsprache sei wie folgt gegeben:

```
datatype
    ('id,'op,'type) prog =
    program of ('id,'op,'type) stat
and ('id,'op,'type) stat =
    & of ('id,'op,'type) stat * ('id,'op,'type) stat
  | decl of 'id * 'type
  | assign of 'id * ('id,'op) exp
  | ifstat of ('id,'op) exp * ('id,'op,'type) stat * ('id,'op,'type) stat
  | whilestat of ('id,'op) exp * ('id,'op,'type) stat
  | skip
and ('id,'op) exp =
    unop of 'op * ('id,'op) exp
  | binop of ('id,'op) exp * 'op * ('id,'op) exp
  | var of 'id
  | bconst of bool
  | const of int;

infix &;
```

Konkrete Instanziierungen für die oben noch offengelassenen Typen für arithmetische und logische Operatoren sowie für Typen in der Quellsprache könnten sein:

```
datatype types = intType | boolType;
datatype ops   = minus | not | plus | logicalOr | less | greater;
```

Damit wäre ein einfaches Quellprogramm, welches $y = 2 * x$ berechnet, durch

```
val p = program (decl("x",intType) & decl("y",intType) &
 whilestat(binop(var "x",greater, const 0),
 assign("x",binop(var "x",minus,const 1)) &
 assign("y",binop(var "y",plus, const 2))
  )
);
```

gegeben. Als Zielsprache wählen wir eine Kellermaschinensprache.

```
datatype 'op ins =
    ^ of 'op ins * 'op ins
  | store of int
  | load of int
  | loadConst of int
  | loadBConst of bool
  | uop of 'op
  | bop of 'op
  | define of int
  | goto of int
  | jumpF of int
  | nop;
```

^ ist die sequentielle Komposition von Befehlen. `load` i lädt den Wert der Zelle i in den Keller. `store` i speichert den Inhalt des obersten Kellerelementes in die Zelle i. `uop` o bzw. `bop` i wendet die Operation o auf das oberste (die obersten beiden) Kellerelement(e) an, löscht diese Kellerelemente und speichert das Ergebnis auf dem Keller. `loadConst` c bzw. `loadBConst` c bringen die Konstante c auf den Keller. `goto` l und `jumpF` l sind Sprungbefehle an die Marke l. Im letzteren Fall wird nur gesprungen, falls das oberste Kellerelement den Wert `false` enthält. `define` l dient zur Definition von Marken.

Es folgen einige Hilfsstrukturen für Symboltabellen und Typprüfung.

```
type 'id symtab = 'id * types * int list;

exception undeclared_id;

fun lookup(nil,x) = raise undeclared_id
  | lookup((y,ty,addr)::st,x)   = if x=y then (ty,addr) else lookup(st,x);

(* Funktionen zur Typpr"ufung bei Operatoren *)

exception type_error;

fun resultType1(minus,intType) = intType
  | resultType1(not,boolType) = boolType
  | resultType1(_,_) = raise type_error;

fun resultType2(plus,intType,intType)   = intType
  | resultType2(minus,intType,intType)   = intType
  | resultType2(less,intType,intType)   = boolType
  | resultType2(greater,intType,intType)  = boolType
  | resultType2(_,_,_)   = raise type_error;
```

Schließlich definieren wir den Übersetzer durch rekursiven Abstieg über das Quellprogramm. Dabei wird jedes Konstrukt in einer Umgebung übersetzt, die durch verschiedene zusätzliche Parameter gegeben wird. `st`, die Symboltabelle, enthält die jeweils sichtbaren Bindungen. Bindungsinformation für jede Variable ist ihre Speicheradresse. Speicheradressen werden durch den Zähler `az` gezählt. entsprechend zählt `lz` die bereits vergebenen Markennummern. Ergebnisparameter sind der erzeugte Code `code` sowie die veränderten Umgebungsbestandteile, die sich durch die Übersetzung eines Programmkonstruktes ergeben haben. Hinzukommt Typinformation für Ausdrücke.

```
fun compileExp(const i,st) = (intType, loadConst i)
  | compileExp(bconst b,st) = (boolType,loadBConst b)
  | compileExp(var x,st) =
let val (ty,addr) = lookup(st,x) in (ty,load addr) end
  | compileExp(unop(oper,e),st) =
let val (ty_e,code_e) = compileExp(e,st)
in (resultType1(oper,ty_e),code_e^uop(oper))
end
  | compileExp(binop(e1,oper,e2),st) =
let val (ty_e1,code_e1) = compileExp(e1,st);
    val (ty_e2,code_e2) = compileExp(e2,st)
in  (resultType2(oper,ty_e1,ty_e2),code_e1^code_e2^bop(oper))
end
and

    compileStat(whilestat (e,s),st,az,lz) =
let val (typ_e,code_e) = compileExp(e,st);
    val (st1,az1,lz1,code_s) = compileStat(s,st,az,lz+2)
in if typ_e = boolType then
      (st1,az1,lz1,define(lz+1)^code_e^jumpF(lz)^code_s^goto(lz+1)^define(lz))
   else raise type_error
end
  | compileStat(skip,st,az,lz) = (st,az,lz,nop)

  | compileStat(ifstat(e,p,q),st,az,lz) =
let val (typ_e,code_e) = compileExp(e,st);
    val (st1,az1,lz1,code_p) = compileStat(p,st,az,lz+2)
    val (st2,az2,lz2,code_q) = compileStat(q,st1,az1,lz1)
in  if typ_e = boolType then
    (st2,az2,lz2,code_e^jumpF(lz+1)^code_p^goto(lz)^
              define(lz+1)^code_q^define(lz))
    else raise type_error
end

  | compileStat(assign(v,e),st,az,lz) =
let val (ty_v,addr) = lookup(st,v);
    val (ty_e,code_e) = compileExp(e,st)
in  if ty_v = ty_e then (st,az,lz,code_e^store(addr))
    else raise type_error
end

  | compileStat(decl(x,ty),st,az,lz) = ((x,ty,az)::st,az+2,lz,nop)
```

```
  | compileStat(s1 & s2,st,az,lz) =
let val (st1,az1,lz1,code_s1) = compileStat(s1,st,az,lz);
    val (st2,az2,lz2,code_s2) = compileStat(s2,st1,az1,lz1)
in  (st2,az2,lz2,code_s1^code_s2)
end
and

    compileProgram(program  s) =
let val (_,_,_,code) = compileStat(s,[],0,0)
in code
end;
```

Wenn wir das Beispielprogramm p übersetzen, so erhalten wir das nachfolgende Zielprogramm.

```
- compileProgram(p);
val it =
  nop ^ nop ^
  (define 1 ^ (load 0 ^ loadConst 0 ^ bop greater) ^ jumpF 0 ^
   (load 0 ^ loadConst 1 ^ bop minus ^ store 0 ^
    (load 2 ^ loadConst 2 ^ bop plus ^ store 2)) ^ goto 1 ^ define 0) : ops ins
```

Symbolisches Differenzieren.
Der folgende Typ beschreibt die abstrakte Syntax der Klasse von Funktionen, die wir differenzieren können wollen.

```
dataype expr = constant of int
             | variable of string
             | sum of exp list
             | product of expr list
```

Es folgen ein paar Beispiele von Funktionsausdrücken:

```
val u = variable("u")
val v = variable("v")
val w = variable("w")
val e = product [v, sum [u,v,w]];
val zero = constant(0)
val one = constant(1);
```

Das Differenzieren erfolgt nach den üblichen Regeln:

```
fun d x (constant _) = zero
  | d (variable s) (variable t) = if s=t then one else zero
  | d x (sum Elist) = sum (map (d x) Elist)
  | d x (product []) = zero
  | d x (product [E]) = d x E
  | d x (product (E1::Elist)) =
    let val EP = product(Elist)
        val T1 = product [d x E1, EP]
```

```
        val T2 = product [E1, d x EP]
    in sum [T1, T2]
    end;
```

Da wir d als eine Funktion vom Typ $d : exp \to exp \to exp$ geschrieben haben, können wir sie partiell auf die Variable x applizieren, nach der differenziert werden soll. Es ergibt sich eine Funktion $(dx) : exp \to exp$, die Ausdrücke speziell nach x differenziert. In der Regel

```
d x (sum Elist) = sum (map (d x) Elist)
```

wird daher die Liste `Elist` der Summanden einer Summe in gewünschter Weise differenziert. Man beachte, daß `map (d x)` die Spezialisierung von `map` ist, die die Fortsetzung von (dx) auf Listen repräsentiert. Hier ein Beispiel für die Anwendung von `d`, wobei der Audruck `e` nach der Variablen `u` differenziert wird.

```
- val e' = d u e;
val e' =
  sum
    [product
       [constant 0,product [sum [variable "u",variable "v",variable "w"]]],
     product [variable "v",sum [constant 1,constant 0,constant 0]]] : expr
```

Da dies noch stark vereinfacht werden kann, schreiben wir noch einen Vereinfacher. Wir brauchen folgende Hilfsfunktionen:

```
fun member _ nil = false
  | member x (y::l) = if x=y then true else member x l;

exception Delete;
fun delete _ nil = raise Delete
  | delete x (y::l) = if x=y then l else y::(delete x l);
```

Dann schreiben wir zunächst alle Simplifikationsregeln als Funktion `rule` auf.

```
exception NotSimplifiable;

fun rule (sum [E]) = E
  | rule (product [E]) = E
  | rule (sum []) = zero
  | rule (product []) = one
  | rule (sum Elist) =
    (sum (delete (constant 0) Elist) handle Delete => raise NotSimplifiable)
  | rule (product Elist) =
    if member (constant 0) Elist
       then (constant 0)
       else (product (delete (constant 1) Elist)
            handle Delete => raise NotSimplifiable)
  | rule _ = raise NotSimplifiable;
```

Hierbei verwenden wir den Ausnahmeindikator `NotSimplifiable` um anzuzeigen, wenn keine Regel (an der Spitze) des gegebenen Ausdrucks anwendbar ist. Man beachte die Umsetzung des Ausnahmeindikators `Delete` zu `NotSimplifiable`, wie sie beispielsweise in der Regel

```
rule (sum Elist) =
 (sum (delete (constant 0) Elist) handle Delete => raise NotSimplifiable)
```

stattfindet. Auf die Summe (`sum Elist`) ist keine Simplifikationsregel anwendbar, wenn keiner der Summanden die Form (`constant 0`) hat. Dieses wird von der Funktion `delete` durch den Ausnahmewert `Delete` angezeigt. Andernfalls wird das erste Auftreten von (`constant 0`) gelöscht, was der Anwendung einer Simplifikationsregel $x+0 \rightarrow x$ oder $0 + x \rightarrow x$ entspricht.

`rule` wendet immer höchstens eine Regel auf einen Ausdruck an, und zwar nur an seiner Spitze. Um einen Ausdruck vollständig zu reduzieren, müssen wir `rule` iteriert und an allen Teiltermen aufrufen. Damit schreiben wir also:

```
fun simplify (sum Elist) =
    let val E1 = sum (map simplify Elist)
    in simplify (rule E1) handle NotSimplifiable => E1
    end
  | simplify (product Elist) =
    let val E1 = product (map simplify Elist)
    in simplify (rule E1) handle NotSimplifiable => E1
    end
  | simplify E = E;
```

Wie gewünscht erhalten wir damit

```
- simplify (d u e);
val it = variable "v" : expr
```

MERKE:

- Spezialisierte Varianten von mehrstelligen Funktionen erhält man durch partielle Applikation. Dazu muß die mehrstellige Funktion in currifizierter Form geschrieben werden. Spezialisierung ist dann besonders nützlich, wenn ein Parameter eine eher globale Bedeutung hat, also weniger dynamisch agiert, als andere Parameter.
- `map` verwendet man zur Fortsetzung von (spezialisierten) Funktionen auf Listen.
- Speziellere Regeln bei Funktionsdefinitionen werden zuerst geschrieben.
- Bei Reduktion durch Termersetzung durch die Verwendung von Ausnahmeindikatoren als Abbruchsbedingung für Termination sorgen.

Tautologietest.
In diesem Abschnitt schreiben wir ein Programm, welches aussagenlogische Formeln auf Tautologieeigenschaft testet. Hier zunächst die abstrakte Syntax von aussagenlogischen Ausdrücken mit Konjunktion, Disjunktion und Negation. Aussagenvariablen werden durch Zeichenreihen benannt.

```
datatype prop = Atom of string
  | Neg  of prop
  | Conj of prop * prop
  | Disj of prop * prop;
```

Hier eine Druckfunktion.

```
fun show (Atom a)     = a
  | show (Neg p)      = "(~" ^ (show p) ^ ")"
  | show (Conj(p,q)) = "(" ^ (show p) ^ " & " ^ (show q) ^ ")"
  | show (Disj(p,q)) = "(" ^ (show p) ^ " | " ^ (show q) ^ ")";
```

Der erste Schritt besteht darin, daß die Formeln in Negationsnormalform gebracht werden, wobei die Negationssymbole zu den Atomen geschoben werden. Grundlage hierfür sind im wesentlichen die De Morganschen Gesetze.

```
fun nnf (Atom a)           = Atom a
  | nnf (Neg (Atom a))     = Neg (Atom a)
  | nnf (Neg (Neg p))      = nnf p
  | nnf (Neg (Conj(p,q))) = nnf(Disj(Neg p, Neg q))
  | nnf (Neg (Disj(p,q))) = nnf(Conj(Neg p, Neg q))
  | nnf (Conj(p,q))        = Conj(nnf p, nnf q)
  | nnf (Disj(p,q))        = Disj(nnf p, nnf q);
```

Hier eine optimierte Variante, die durch zwei Funktionen für positive bzw. negierte Ausdrücke repräsentiert wird.

```
fun nnfpos (Atom a)     = Atom a
  | nnfpos (Neg p)      = nnfneg p
  | nnfpos (Conj(p,q)) = Conj(nnfpos p,nnfpos q)
  | nnfpos (Disj(p,q)) = Disj(nnfpos p,nnfpos q)
and nnfneg (Atom a)     = Neg (Atom a)
  | nnfneg (Neg p)      = nnfpos p
  | nnfneg (Conj(p,q)) = Disj(nnfneg p,nnfneg q)
  | nnfneg (Disj(p,q)) = Conj(nnfneg p,nnfneg q);
```

Ist eine Formel in Negationsnormalform, so muß nur noch das Distributivgesetz angewendet werden, um sie auf konjunktive Normalform zu bringen.

```
fun distrib (p, Conj(q,r)) = Conj(distrib(p,q), distrib(p,r))
  | distrib (Conj(q,r), p) = Conj(distrib(q,p), distrib(r,p))
  | distrib (p, q)         = Disj(p,q)    (*no conjunctions*);
```

so daß die konjunktive Normalform `cnf` (einer Formel in Negationsnormalform) nun wie folgt programmiert werden kann:

```
fun cnf (Conj(p,q)) = Conj (cnf p, cnf q)
  | cnf (Disj(p,q)) = distrib (cnf p, cnf q)
  | cnf p           = p   (*a literal*)
```

Diese Hilfsfunktionen filtern die positiven bzw. negativen Literale aus einer konjunktiven Normalform heraus:

```
exception NonCNF;
fun positives (Atom a)      = [a]
  | positives (Neg(Atom _)) = []
  | positives (Disj(p,q))   = positives p @ positives q
```

```
  | positives  _            = raise NonCNF;

fun negatives (Atom _)      = []
  | negatives (Neg(Atom a)) = [a]
  | negatives (Disj(p,q))   = negatives p @ negatives q
  | negatives  _            = raise NonCNF;
```

Wir brauchen noch eine Funktion `inter`, die den Durchschnitt zweier als Listen repräsentierter Mengen berechnet. Unsere Lösung verwendet Funktionen höherer Ordnung.

```
fun member nil _    = false
  | member (y::l) x = if x=y then true else member l x;

fun filter _ nil    = nil
  | filter p (x::l) = if p x then (x::(filter p l)) else (filter p l);


fun inter l1 l2 = (filter (member l1) l2);
```

Schließlich ist eine Formel in konjunktiver Normalform genau dann eine Tautologie, wenn in jeder ihrer Klauseln ein Atom sowohl positiv als auch negativ vorkommt.

```
fun taut (Conj(p,q)) = taut p andalso taut q
  | taut p           = ([] <> inter (positives p) (negatives p));
```

3.8.2 Ströme

In diesem Kapitel besprechen wir die Datenstruktur der *Ströme*, auch *Folgen* genannt. Vorläufig können wir uns Ströme einfach als Listen vorstellen, bis verschiedene Überlegungen uns eine verfeinerte Repräsentation von Strömen abverlangen werden.

Ströme tauchen in vielen Gebieten der Informatik in der einen oder anderen Form auf, z.B. bei der Datenflußprogrammierung oder im Übersetzerbau. In jedem Fall dient das Konzept der expliziten Formulierung von strukturellen Ähnlichkeiten in einer großen Klasse von Programmen. Zusammen mit Strömen beobachten wir folgende Grundgedanken:

1. Komplexe Mechanismen oder Module sollen als Kombination (funktionale Komposition, Hintereinanderschaltung, usw.) einfacherer Mechanismen oder Module entstehen.

2. Rekursion, ein unvermeidbares Konzept, immer dann, wenn Ein- und Ausgaben sowie Rechenprozesse unbeschränkte Länge haben dürfen, soll implizit bleiben. D.h. gewisse vorgegebene Standardbausteine sind intern rekursiv definiert. Die externe Kombination der Standardbausteine aber ist selbst nicht rekursiv.

3. Kommunikation zwischen Modulen verläuft auf der Basis des Austauschens von Folgen von Informationseinheiten. Diese Folgen von Daten sind die Ströme. Uniformität des Aufbaus eines Systems wird verbessert, weil die Eingaben und Ausgaben aller Module von derselben Struktur, Ströme eben, sind.

Wir finden diese Idee z.B. im Betriebssystem UNIX. Hier hat jeder Prozeß eine Standardein- und eine Standardausgabe. Auf diesen Leitungen fließen Ströme von Zeichen, die der Prozeß liest bzw. ausgibt. Durch Verwendung einer „pipe“ kann die Ausgabe eines Prozesses mit der Eingabe eines zweiten Prozesses verbunden werden. Komplexe Prozesse entstehen als Hintereinanderschaltung von einfacheren Prozessen. Wollten wir aus dem Systemwörterbuch alle mit „a“ beginnenden Wörter suchen, so daß Wörter nach ähnlichen Endungen sortiert werden, wäre eine Lösung

```
cat /usr/dict/words | rev | sort | rev | grep "^a" | pr -t -3 | more
```

„`|`“ ist der Kompositionsoperator „pipe“. Die primitiven Funktionen sind `cat` (zur Erzeugung (Aufzählung) des Stroms, der aus der Konkatenation aller Parameterdateien, hier `/usr/dict/words`, besteht), `rev` (zum zeichenweisen Spiegeln aller Zeilen des Stroms), `sort` (zum alphabetischen Sortieren von Zeilen), `grep` (zum Ausfiltern aller Zeilen, die ein bestimmtes Muster enthalten, hier mit „a“ beginnen), `pr -t -3` (zum 3-spaltigen, seitenweisen Tabellieren von Strömen) und `more` (zum seitenweisen Ausgeben eines Stroms auf dem Bildschirm). Das Ergebnis dieses Kommandos wäre so etwas wie

```
a           atrophic         anastomotic
amoeba      anamorphic       asymptotic
alba        anthropomorphic  analeptic
armada      automorphic      aseptic
addenda     anorthic         apocalyptic
agenda      acyclic          aspartic
anaconda    angelic          antagonistic
althea      alcoholic        anachronistic
azalea      apostolic        autistic
area        acrylic          atavistic
alfalfa     aerodynamic      agnostic
alga        academic         acoustic
aloha       algorithmic      attic
alpha       astronomic       aeronautic
apologia    autonomic        astronautic
alia        atomic           analytic
academia    anatomic         arc
ammonia     anthropogenic    anybody'd
armillaria  arsenic          ad
asteria     actinic          ahead
aphasia     anionic          arrowhead
amnesia     avionic          armload
```

was ein Amateurdichter zum Finden von Reimen, z.B. „apostolic“ auf „alcoholic“, benutzen könnte.

Bei Strömen spielt es keine Rolle, ob man sich bei Kombination `p|q` von `p` und `q` hintereinander oder parallel ausgeführt denkt. Üblicherweise benötigt `q` nicht seine gesamte Eingabe auf einmal, um Teilberechnungen anstellen zu können. Damit macht es Sinn, `p` und `q` parallel zu starten. Da `p` seinen Ausgabestrom elementweise produziert und `q` seine Eingabe elementweise und sequentiell liest, muß `q` aber dann warten, wenn es das nächste Element seines Eingabestromes lesen will, `p` aber dieses noch nicht produziert hat, d.h. die „pipe“ zwischen `p` und `q` leer ist. Diese Form der Kommunikation zwischen parallelen Prozessen ist grundlegend, und daher spielen Ströme von Kommunikationseinheiten eine wichtige Rolle bei der mathematischen Modellierung sowie bei Spezifikation und Programmierung kommunizierender, parallel ablaufender Prozesse.

Wie gesagt sind Programme hier meist nichtrekursive Kombinationen von primitiven Bausteinen und damit nur implizit rekursiv. Implizite Rekursion hat den Vorteil, daß wir weitgehend die Probleme *induktiven Schließens* von Programmeigenschaften vermeiden können und mit *algebraischen Gesetzen* auskommen. Desweiteren bekommt man garantierte Terminierung aller Programme dann, wenn die vorgegebenen primitiven Elemente garantiert terminieren.

Literatur.

Abelson, H. und Sussman, G.J.: Structure and Interpretation of Computer Programs, MPI Press, 1985.

Bird, R. und Wadler, Ph.: Introduction to Functional Programming, Prentice Hall, 1988.

Homomorphismen auf Strömen. Ströme sind immer endliche oder unendliche Folgen von Daten (von beliebigem Typ α). Die verschiedenen Ansätze unterscheiden sich darin, von welchen algebraischen Eigenschaften von Strömen ausgegangen wird. Wir wollen hier von einer sehr einfachen Struktur von Strömen ausgehen, wie sie durch die Operationen

$$\begin{array}{ll} \texttt{Nil:} & \alpha\ \texttt{stream} \\ \texttt{Cons:} & \alpha * \alpha\ \texttt{stream} \to \alpha\ \texttt{stream} \\ \texttt{head:} & \alpha\ \texttt{stream} \to \alpha \\ \texttt{tail:} & \alpha\ \texttt{stream} \to \alpha\ \texttt{stream} \end{array}$$

repräsentiert wird. Dann werden endliche Ströme $[e_1\, e_2 \ldots e_n]$ durch Ausdrücke der Form

$\texttt{Cons(}e_1\texttt{,Cons (}e_2\texttt{, ..., Cons(}e_n\texttt{,Nil)...))}$

berechnet.[11]

Zunächst nehmen wir, wie bereits gesagt, an, daß Ströme wie Listen realisiert sind.

```
datatype 'a stream = Nil
                   | Cons of 'a * 'a stream;
fun head(Cons(x,_)) = x;
fun tail(Cons(_,xs)) = xs;
```

Später wollen wir besser geeignete Realisierungen besprechen, wenn wir einmal gesehen haben, wie typischerweise mit Strömen programmiert wird.

Eine wichtige Beobachtung ist, daß die meisten Programme, die Ströme als Eingabe akzeptieren, in einfacher, syntaxorientierter Weise ihr Ergebnis berechnen. Algebraisch formalisiert bedeute „einfach“, daß das Programm einen *Homomorphismus* berechnet. Ein Homomorphismus h von Strömen in einen Typ β, repräsentiert durch eine Konstante $\texttt{a}$ vom Typ β und eine binären Funktion $\oplus : \alpha * \beta \to \beta$, ist eine Abbildung $h : \alpha\ \texttt{stream} \to \beta$, gegeben durch die Gleichungen

$$h(\texttt{Nil}) \quad = \quad \texttt{a}$$

[11] Für Ströme mit den Elementen $e_1, \ldots, e_n$ benutzen wir die Notation $[e_1 \ldots e_n]$, wie wir sie auch für Listen verwenden.

$$h(\texttt{Cons}(e,s)) \quad = \quad e \oplus h(s) \ .$$

Ein Homomorphismus h ist also durch Angabe von $\oplus$ und a eindeutig bestimmt. Da man sich die Arbeitsweise von h als ein Reduzieren der Ströme von rechts her vorstellen kann,

$$h([e_1 e_2 \ldots e_n]) = e_1 \oplus (e_2 \oplus \ldots \oplus (e_n \oplus \texttt{a}) \ldots)),$$

nennen wir die entsprechende Funktion `foldr` von „fold right" (in korrekter SML-Notation mit einem Identifikator f anstelle des Symbols $\oplus$):

```
- fun foldr _ a Nil         = a
    | foldr f a Cons(x,xs) = f(x, (foldr f a xs));
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a stream -> 'b
```

Als einfache, mit `foldr` berechnete Homomorphismen seien die folgenden genannt:

```
fun sum s = foldr op+ 0 s;
```

`sum` berechnet die Summe aller Elemente eines Stroms von Zahlen. Hier ist $\texttt{a} = \texttt{0}$ und $\oplus = +$. Analog wird das Produkt der Elemente durch

```
fun product s = foldr op* 1 s;
```

berechnet. Das Maximum der Elemente eines Stroms nichtnegativer Zahlen erhalten wir mit

```
fun max_element s = foldr max 0 s;
```

Die allen drei Anwendungen gemeinsame Rekursionsstruktur wird von derjenigen in `foldr` erfaßt. Die Definition von `sum`, `product` und `max-element` sind selbst nicht rekursiv.

Oftmals haben auch Funktionen auf Zahlen diese Homomorphismenstruktur, wenn man z.B. natürliche Zahlen `n` als Ströme $[1\ 2 \ldots n]$ repräsentiert. Hier wird die Funktion

```
fun interval a b =
    if a>b then Nil else Cons(a,interval (1+a) b);
```

gebraucht, die das Intervall $[a, b]$ als Strom der ganzen Zahlen zwischen a und b codiert. Jetzt können wir auch Funktionen, wie die Fakultät, als Funktion auf Strömen sehen,

```
fun fac n = product (interval 1 n);
```

also als das Produkt der ganzen Zahlen im Intervall $[1, n]$.

Um Programme auf Strömen wie in UNIX „pipe"-artig komponieren zu können, müssen häufig genug Ströme auch als Resultate von Programmen auftreten.

Eine wichtige Funktion ist `map`, die wir schon bei Listen kennengelernt haben. Bei Stömen wendet sie Funktionen f auf alle Elemente eines Stroms an liefert und den Strom der Ergebnisse zurück. Die Definition von `map` durch

```
- fun map f s = foldr (fn (x,xs) => Cons((f x),xs)) Nil s;
val map = fn : ('a -> 'b) -> 'a stream -> 'b stream
```

zeigt, daß auch (`map` f) ein Homomorphismus ist, für jedes f. `map` ist also eine *Homomorphismenfamilie.* $\oplus$ ist ein `Cons`, nur werden zusätzlich die ursprünglichen Stromelemente x in $f(x)$ übersetzt.

```
fun sum_squares n =
   let fun square x = x*x
    in sum (map square (interval 1 n))
   end;
```

berechnet die Summe $\sum_{i=1}^{n} i^2$ der Quadrate von 1 bis n.

```
- sum_squares 5;
val it = 55 : int
```

Neben `map`, der Fortsetzung einer Abbildung von Elementen auf Ströme, ist das Filtern von Strömen durch ein Prädikat eine häufig vorkommende Teilaufgabe. Um auch `filter` als Familie von Homomorphismen zu erkennen, brauchen wir noch eine Operation für Ströme, mit `++` bezeichnet, die, wie das `append` bei Listen, seine Argumente konkateniert:

```
fun ++ s1 s2 = foldr Cons s2 s1;
```

`++` für jedes feste `s2` ein Homomorphismus, also ebenfalls eine Homomorphismenfamilie.[12]

`filter` kann nun wie folgt definiert werden:

```
fun filter p s =
   foldr ++ Nil
      (map (fn x => if p x then singleton x else Nil) s);
```

wobei `singleton` aus `Cons` durch

```
fun singleton e = Cons(e, Nil);
```

abgeleitet werden kann. `filter` produziert aus dem gegebenen Strom `s` zunächst einen Strom `s'` von Strömen. Dieser enthält als Elemente die einelementigen Ströme `[x]`, für alle `x` in `s`, für die (`p x`) wahr ist. Die Elemente `y`, die `p` nicht erfüllen, werden durch die leeren Ströme unter den Elementen von `s'` repräsentiert. Konkatenieren wir alle Ströme von `s'`, so erhalten wir den Strom der Elemente von `s`, die `p` erfüllen. Letzteres geschieht durch ein `foldr` mit `++` als Kombinator.

Auch konventionelle Datenverarbeitungsprobleme sind durch stromorientiertes Programmieren oft einfach zu lösen.

---

[12] Algebraisch betrachtet spricht einiges dafür, Ströme anstatt durch `Cons` und `Nil` als Datentyp mit den Konstruktoren

`Nil :`	α`stream`
`++ :`	α`stream` $*$ α`stream` $\rightarrow$ α`stream`
`singleton :`	$\alpha \rightarrow \alpha$`stream`

einzuführen. Hierbei ist (`singleton e`) der Strom, der `e` als einziges Element besitzt. `++` und `Nil` bilden eine Monoidstruktur. Die Konstruktion von Strömen über `Cons` wiederum hat den Vorteil, daß die Sequentalität der Elemente in Strömen und der auf ihnen operierenden Prozesse direkter zum Ausdruck kommt. Beide Sichtweisen von Strömen können leicht wechselweise ineinander überführt werden.

„Finde die Note des bzw. der besten diesjährigen Absolventen unter den Studenten“

kann man durch

```
fun markOfBestStudent recordStream =
    foldr min
          4.3   ;; schlechteste Note, mit der man noch besteht
          (map mark
               (filter graduatedThisYear recordStream)))
```

berechnen, wenn `recordStream` der Strom der Daten über die Studenten ist, aus dem `graduatedThisYear` die diesjährigen Absolventen herausfindet, und `mark` deren Note liest.

Fassen wir die bisher besprochenen Funktionen auf Strömen noch einmal zusammen:

1. Aufzählen von Strömen: z.B. `interval`. Im übernächsten Paragraphen werden wir komplexere Mechanismen zu diesem Zweck besprechen.

2. Elementweises Abbilden von Strömen: `map`

3. Filtern von Strömen: `filter`

4. Die einzige rekursive Funktion ist `foldr`, das Konzept des Homomorphismus. Alle anderen Funktionen ergeben sich durch nichtrekursive Komposition (Hintereinanderschaltung) von Homomorphismen oder (instanziierten) Homomorphismenfamilien, wie `map`, `filter` und `++`.

Abarbeitung von links. Homomorphismen mit `foldr` arbeiten Ströme von rechts her ab. Dies ist algebraisch schöner, aber oftmals nicht sehr effizient. Für unendliche Ströme, wie wir sie noch besprechen werden, ist diese Methode natürlich ungeeignet. Selbst für endliche Ströme ist es oftmals effizienter, wenn man sie von links her abarbeitet, insbesondere dann, wenn für das Ergebnis nicht alle Elemente des Stroms inspiziert werden müssen. Die lineare Rekursion von `foldr` will immer erst das Ende des Stromes finden, bevor irgendwelche Bestandteile des Ergebnisses berechnet werden. Wir verwenden ab jetzt absichtlich auch keine `Cons`-Muster, sondern arbeiten stattdessen mit den Selektoren `head` und `tail`, um der späteren Realisierung unendlicher Ströme vorzubauen. Das sequentielle Analogon zum rekursiven `foldr` ist `foldl`. Es berechnet

$$\texttt{foldl} \otimes \texttt{a}\ [e_1 e_2 \ldots e_n] = (\ldots(a \otimes e_1) \otimes e_2) \otimes \ldots) \otimes e_n)$$

und wird wie folgt definiert (mit f für $\otimes$):

```
fun foldl f a Nil = a
  | foldl f a xs = foldl f (f(a,head xs)) (tail xs);
val foldl = fn : ('a * 'b -> 'a) -> 'a -> 'b stream -> 'a
```

Erstes Anwendungsbeispiel ist `pack`, welches den Wert der durch eine Ziffernfolge gegebenen Dezimalzahl berechnet:

```
fun pack xs = foldl (fn (n,x) => 10*n+x) 0 xs;
- pack (interval 1 3);
```

```
val it = 123 : int
```

Die endständige Rekursion von `foldl`, die sich in effizienter Weise (konstanter Speicherverbrauch) in Schleifen übersetzen läßt, wird im folgenden dafür sorgen, daß die auf `foldl` aufbauenden Funktionen selbst im wesentlichen endständig rekursiv sind.

Zwischen `foldl` und `foldr` bestehen Zusammenhänge, gennant *Dualitätsgesetze.*

Proposition 1 *Seien a, $\oplus$ und $\otimes$ so, daß*

$$\begin{aligned} x \oplus (y \otimes z) &= (x \oplus y) \otimes z \\ x \oplus a &= a \otimes x \end{aligned}$$

für alle x, y, z gilt. Dann ist

$$\texttt{foldr} \oplus a \ xs = \texttt{foldl} \otimes a \ xs \tag{5}$$

für alle endlichen Ströme xs.

Der Beweis dieser Aussage erfolgt durch Induktion über die Länge der Ströme und ist etwas trickreich, indem man zuerst von vorne und danach von hinten den Strom um ein Element kürzt, um dann auf den um jeweisl ein Element kürzeren Strom die Induktionsannahme anzuwenden.

Eine unmittelbare Folge dieser Proposition ist:

Proposition 2 *Bildet $\oplus$ und a ein Monoid, so ist*

$$\texttt{foldr} \oplus a \ xs = \texttt{foldl} \oplus a \ xs \tag{6}$$

Dieses klärt den Zusammenhang zwischen `foldl` und `foldr`:

Proposition 3 *Es gilt*

$$\texttt{foldr} \oplus a \ xs = \texttt{foldl} \oplus^{-} a \ (\texttt{reverse} \ xs) \tag{7}$$

für alle endlichen Ströme xs, wobei $\oplus^{-}$ durch $x \oplus^{-} y = y \oplus x$ gegeben sei.

Diese Gesetze können als Grundlage einer wichtigen Technik für *Entrekursivierung* von Programmen betrachtet werden: Die Definition einer Funktion als Homomorphismus, also mittels `foldr`, ist normalerweise die einfachere und intuitivere Alternative. Programme mit Hilfe von `foldl`, weil endständig rekursiv, sind aber effizienter ausführbar. Die Dualitätsgesetze helfen, die eine in die andere Form überzuführen.

Eine weitere Anwendung dieser Gesetze ist Programmverifikation ohne Induktion. Wir zeigen nun, als ein Beispiel, daß

$$\texttt{reverse}(\texttt{reverse}(xs)) = xs,$$

ohne dabei explizite Induktion zu verwenden. Sei dazu

```
fun prefix (xs,x) = Cons(x,xs);
val reverse = foldl prefix Nil;
```

Die so definierte `reverse`-Funktion entspricht der nichtnaiven Variante mit endständiger Rekursion. Es ist $\texttt{prefix}^{-} = \texttt{Cons}$, damit gilt wegen des zweiten Dualitätsgesetzes

$$\texttt{foldr Cons Nil} \ xs = \texttt{foldl prefix Nil (reverse} \ xs)$$

und die Behauptung folgt aus der Tatsache, daß

$$xs = \texttt{foldr Cons Nil } xs.$$

Die Induktion ist versteckt in der „kanonischen Induktion", die man verwendet hat, um die Dualitätsgesetze nachzuweisen.

Zum Abschluß einige Merksätze:

- `foldl` ist die endständig rekursive Variante von `foldr` und damit zumeist effizienter.
- Für Monoide $(\oplus, a)$ sind beide gleich. Die Verwendung von `foldl` entspricht einer automatischen Entrekursivierung, wobei der Speicherverbrauch konstant ist, unabhängig von der Rekursionstiefe.
- Algebraische Gesetze zwischen Kombinatoren wie z.B. die obigen Dualitätsgesetze, erlauben induktionsloses Verifizieren gewisser Programmeigenschaften.

Diese Aspekte ist besonders ausgearbeitet in der Backus-Sprache FP [?] und im sogenannten „Bird/Meertens-Formalismus" [?].

Aufzählen von Mengen und Multimengen. Eine weitere Anwendung der Ströme in der funktionalen Programmierung ist nahe verwandt zu einer wichtigen Technik des logischen Programmierens, bei der eine Menge (bzw. ein Prädikat) definiert wird, indem zunächst eine Grundmenge aufgezählt wird und dann die Elemente herausgefiltert werden, die das die Menge definierende Prädikat erfüllen. In der Mathematik schreiben wir ja genau nach dieser Methode *Komprehensionsausdrücke*

$$M = \{\, x \in G \mid p(x) \,\},$$

wenn wir eine Menge M aller Elemente x einer bereits gegebenen Grundmenge G, die die Eigenschaft $p(x)$ besitzen, definieren wollen. In Prolog könnten wir dann schreiben

```
'M'(X) :- 'G'(X), p(X).
```

`G` generiert die Grundmenge, `p` filtert diese nach der `M` charakterisierenden Eigenschaft. Wir sagen auch, `G` zählt den Suchraum auf, und `p` schränkt ihn auf die zu `M` gehörigen Werte ein.

Das Schema, welches wir hier anbieten wollen, ist etwas allgemeiner. Es basiert auf der Idee der *abhängigen Produkte* („dependent products") von *Multimengen.* Multimengen sind Strukturen wie Mengen, außer daß Elemente mehrfach auftreten können. Wie bei Mengen ist die Reihenfolge des Auftretens irrelevant. Mengen sind Spezialfälle von Multimengen. Wir wollen Multimengen[13] der Form

$$\begin{array}{lllll} M = [\ f(x_1, \ldots, x_n) & \mid & x_1 & \in & M_1,\ p_1(x_1), \\ & & x_2 & \in & M_2(x_1),\ p_2(x_1, x_2), \\ & & x_3 & \in & M_3(x_1, x_2),\ p_3(x_1, x_2, x_3), \\ & & & & \ldots \\ & & x_n & \in & M_n(x_1, \ldots, x_{n-1}),\ p(x_1, \ldots, x_n), \quad] \end{array}$$

betrachten, wobei

[13] Zur Unterscheidung der Multimengen von Mengen benutzen wir eckige Klammern zu ihrer Notation. Also, $M = [x \in G \mid p(x)]$ sei die Multimenge M aller Elemente x der Multimenge G, die $p(x)$ erfüllen.

- f eine n-stellige Funktion,
- M_1, $M_2(x_1)$, $M_3(x_1, x_2)$, ... , $M_n(x_1, \ldots, x_{n-1})$, n (bereits gegebene) Multimengenfamilien und
- die p_i i-stellige (gegebene) Prädikate sind, die weggelassen werden, wenn sie trivial sind.

(Wir nehmen hierbei an, daß gewisse elementare Multimengen auf andere Weise, z.B. durch Aufzählung ihrer Elemente, vgl. `interval`, definiert werden.)

Die Besonderheit der abhängigen Produkte ist, daß der i-te Faktor M_i von den Elementen der $i-1$ vorangehenden Faktoren abhängig sein darf. Beispiele für Multimengen sind

$$pairs(k) = [\,(i,j) \mid i \in [1,k], j \in [i+1,k]\,]$$

als die Menge aller Paare (i,j) mit $1 \leq i \leq k$ und $i < j \leq k$, oder

$$triples(k) = [\,(i,j,i+j) \mid i,j \in [1,k], j \in [1,i-1]\,]$$

als eine Menge von Tripeln, in der die dritte Komponente die Summe der beiden ersten ist. Daß das zweite Beispiel in die Klasse der hier betrachteten Multimengen fällt, ergibt sich mit $n = 2$, $f(i,j) = (i,j,i+j)$, $M_1 = [1,k]$, $M_2(i) = [1,i-1]$, $p = true$.

In einigen funktionalen Programmiersprachen, wie z.B. Haskell oder Miranda, gibt es vordefinierte Notationen für solche Multimengenkomprehensionen. In SML können wir uns diese durch folgendes einfache, rekursive Übersetzungsschema herstellen (wobei man die Regelanwendung solange iteriert, bis alle Komprehensionen eliminiert sind):

$$\begin{array}{rcl}
[\,x \mid x \in xs\,] & \Rightarrow & xs \\
[\,f\ x \mid x \in xs\,] & \Rightarrow & \texttt{map}\ f\ xs \\
[\,e \mid x \in xs,\ px, \ldots] & \Rightarrow & [\,e \mid x \in \texttt{filter}\ p\ xs, \ldots] \\
[\,e \mid x \in xs,\ y \in ys, \ldots] & \Rightarrow & \texttt{concat}\,[\,[\,e \mid y \in ys, \ldots] \mid x \in xs\,]
\end{array}$$

wobei `concat` definiert ist als

```
val concat = foldl op++ Nil
```

und alle Elemente eines Strom von Strömen konkateniert.

Die letzte dieser Regeln verdient eine Erläuterung. Die linke Seite definiert einen Strom von Elementen einer Familie e, die durch (mindestens) zwei Parameter x und y indiziert ist. Die Regel transformiert dieses zunächst in einen Strom von Strömen $[\,e \mid y \in ys, \ldots]$, für jedes x in xs. `concat`, auf diesen zweidimensionalen Strom angewendet, macht dann wieder einen eindimensionalen Strom daraus. Wesentlich ausgenutzt wird hierbei, daß die Reihenfolge der Elemente in den Strömen keine Rolle spielt, weil wir Multimengen implementieren.

Betrachten wir ein Beispiel, bei dem wir die Primzahlen unter den Summen von je zwei Zahlen im Intervall $[1,k]$ ausrechnen wollen:

$$\texttt{primeSum}\ (k) = [\,(u,v,u+v) \mid u \in [1,k], v \in [1,u-1],\ \texttt{prime}\ (u\texttt{+}v)\,]$$

Die Übersetzung hiervon ergibt

$[\,(u,v,u+v) \mid u \in [1,k], v \in [1,u-1],$ `prime` $(u\texttt{+}v)\,]$
$\Rightarrow$ `concat` $[\,[\,(u,v,u+v) \mid v \in [1,u-1],$ `prime` $(u\texttt{+}v)\,] \mid u \in [1,k]\,]$
$=$ `concat` $[\,[\,(u,v,u+v) \mid v \in [1,u-1],$ `(fn` v `=> prime` $(u\texttt{+}v)$ $\,] \mid u \in [1,k]\,]$
$\Rightarrow$ `concat` $[\,[\,(u,v,u+v) \mid v \in$ `(filter (fn` v `=> prime` $(u\texttt{+}v)$`)` $[1,u-1])\,] \mid u \in [1,k]\,]$
$\Rightarrow$ `concat` $([\,P\ u \mid u \in [1,k]\,])$
$\Rightarrow$ `concat (map` P `[1, k])`

wobei P die Funktion

```
fun P u = map (fn v => (u,v,u+v))
              (filter (fn v => prime (u+v)) [1, u-1])
```

Mittels `interval` können wir das Ganze wie folgt aufschreiben:

```
fun primeSum k =
let fun P u = map (fn v => (u,v,u+v))
                  (filter (fn v => prime (u+v)) (interval 1 (u-1)))
    in concat (map P (interval 1 k))
end;
```

```
- primeSum 6;
```
((2 1 3) (3 2 5) (4 1 5) (4 3 7) (5 2 7) (6 1 7) (6 5 11))

Wie wir den Primzahltest `prime` programmieren können, z.B. durch Aufzählung des Stromes aller Primzahlen, werden wir im nächsten Abschnitt sehen.

Daß es uns möglich ist, so komplexe Konstruktionen, wie die Multimengenkomprehensionen nur mit `map`, `filter`, `++` und `foldr` (für `concat`) zu simulieren, zeigt die Mächtigkeit, die man durch Kombination dieser elementaren Elemente erzielen kann. Andererseits kann man wohl feststellen, daß gerade die logische Komplexität der Mengenkomprehensionen eine gute Motivation dafür ist, daß eine geeignete Notation hierfür in jeder funktionalen Programmiersprachen vorgegeben sein sollte. Wer möchte schon `primeSum`, ein noch relativ harmloses Beispiel, so „zu Fuß“ programmieren müssen.

Implementierung von Strömen. Bisher haben wir die Vorstellung propagiert, Ströme seien Listen. Die Verwendung von Strömen in der Praxis zeigt, daß diese Implementierung oft sehr ineffizient ist. Oftmals fließen nur wenige Elemente eines Stromes in das Endergebnis einer Rechnung ein. Wenn dann der Strom sehr lang ist, werden viele Elemente umsonst berechnet. Ein zweites Problem ist, daß bei Verwendung der stromorientierten Programmiertechnik ein komplexes Programm zumeist eine Komposition von recht vielen einfachen Funktionen ist. An der Schnittstelle zwischen je zwei Funktionen entstehen aber Ströme als Zwischenergebnis, die bei der Listensichtweise vollständig im Listenspeicher des SML-Systems abgelegt werden müssen.

Ein einfaches Beispiel kann beide Probleme illustrieren.

```
fun secondPrimeSquareIn a b =
   head (tail (map square
                   (filter prime
                           (interval a b)))))
```

welches wir, unter Verwendung der polymorphen Komposition `o` von Funktionen

```
infix o;
fun o (f,g) = (fn x => f(g x));
```

die in dieser Weise bereits vordefiniert ist, kürzer und „kombinatorischer“ als

```
fun secondPrimeSquareIn a =
    head o (tail o ((map square) o ((filter prime) o (interval a))));
```

schreiben können. `secondPrimeSquareIn` berechnet das Quadrat der zweiten Primzahl im Intervall $[a, b]$, wenn $[a, b]$ mindestens 2 Primzahlen enthält. Der Aufruf

```
- secondPrimeSquareIn 34 100000
```
1681

produziert sein Ergebnis, nachdem 3 große Ströme als Zwischendatenstrukturen erstellt worden sind, wobei zusätzlich zumindest das Filtern auf Primzahleigenschaft sehr rechenaufwendig für jede große Zahl des ersten Stroms ist. Letztlich werden aber nur zwei Primzahlen wirklich gebraucht. Ad hoc hätte man das Beispielproblem vielleicht so programmiert:

```
fun secondPrimeSquareIn a b =
    let nthPrimeSquareIn a b n =
       if a>b then error "Intervall erschoepft"
       else if prime a
              then if n=1
                   then square a
                   else nthPrimeSquareIn (1+a) b (n-1)
              else nthPrimeSquareIn (1+a) b n)
     in nthPrimeSquareIn a b 2
end;
```

Dieses Programm braucht nur einen kleinen konstanten Speicher[14] und eine kleine Laufzeit, wenn `a` nicht zu groß ist.

Natürlich kann man eine Methode, wie die der Programmierung mit Strömen, nicht dann propagieren, wenn sie zu derart dramatischen Verschlechterungen im Laufzeitverhalten und Speicherverbrauch führt. Was kann man also tun, um Ströme so zu realisieren, daß der Überhang in bezug auf Komplexität nur einen konstanten Faktor ausmacht? Eine Idee ist die der *verzögerten Auswertung* von Strömen. Ströme sollen nur soweit berechnet werden müssen, wie ihre Elemente auch tatsächlich von den Funktionen, die den Strom als Eingabe lesen müssen, gebraucht werden. Im Beispiel haben wir die Folge

```
head o (tail o ((map square) o ((filter prime) o (interval a))))
```

von stromproduzierenden bzw. -konsumierenden Funktionen. Vom Strom, der Eingabe für `tail` ist, brauchen wir nur die ersten zwei Elemente. Um die von `map` berechnen zu lassen, braucht `map` seinerseits nur die ersten zwei Elemente seiner Eingabe. Also muß `filter` nur solange laufen, bis es in seiner Eingabe die ersten zwei Primzahlen gefunden hat, und mehr Eingaben muß `interval` nicht generieren.

[14] Man beachte, daß die Prozedur endständig rekursiv ist, also einen iterativen Prozeß erzeugt.

Im allgemeinen muß erst dann das erste Element, und erstmal nur dieses, eines Stroms berechnet werden, wenn mit `head` darauf zugegriffen wird oder wenn mit `tail` zum Reststrom übergegangen wird.

Ein Standardtrick, um in SML, wo immer alle Argumentausdrücke einer Applikation ausgewertet werden, bevor die Auswertung des Funktionsrumpfes stattfindet, die Auswertung zu verzögern, ist das Einpacken eines Ausdrucks in einen λ-Ausdruck. Ist also `e` ein Ausdruck, so vermeidet man seine Auswertung dadurch, daß man stattdessen

```
(fn () => e)
```

schreibt. Aus einem Wert wird hierdurch eine parameterlose Funktion, die genau diesen Wert als Ergebnis liefert. Mathematisch betrachet ist da kein Unterschied. Operational gesehen ist `(fn () => e)`, im Gegensatz zu `e` nur ein *Versprechen*, den Wert von `e` zu berechnen, wann immer man es möchte.

Wie wir schon gesehen haben, wird eine nur partiell applizierte Funktion, ein λ-Ausdruck ist ein Spezialfall hiervon, erst dann reduziert, wenn alle übrigen Argumente in einer Applikation vorliegen. Ein λ-Ausdruck wird also nur dann reduziert, wenn er linker Operand einer Applikation ist und die gesamte Applikation dann gemäß der im λ-Ausdruck spezifizierten Reduktionsregeln reduziert wird. Einen Auswertungswunsch äußert man also, indem man `(fn () => e)` aufruft, also z.B. `(fn () => e) ()` oder `x()` schreibt, wenn `x` eine Variable, z.B. Funktionsparameter, ist, an die `(fn () => e)` gebunden ist.

Dieses *Erzwingen der Auswertung* wollen wir augenfälliger machen, indem wir eine Funktion `force` definieren, die durch

```
fun force x = x ()
```

gegeben sei. Entsprechend sei `(delay e)` von nun an immer als eine *Abkürzung* für `(fn () => e)` gebraucht, so daß bereits der Name uns den Grund für das Einpacken von `e` andeutet. Wohlgemerkt, wir können *nicht*, im Gegensatz zu `force`, `delay` als eine Funktion, etwa durch

```
fun delay e = (fn () => e)
```

programmieren. Jeder Aufruf `(delay e)` würde ja gerade den mit `delay` beabsichtigten Zweck zunichtemachen und `e` auswerten, bevor der sich ergebende Wert dann überflüssigerweise in ein `lambda` eingepackt wird. `delay` ist also als ein *Makro* zu verstehen. In Lisp wäre sowas unproblematisch, da praktisch jedes Lisp-System ein Makro-Konzept hat, schon aus den Gründen, die wir gerade besprechen. Im Falle von SML müssen wir als Programmierer selbst den Makroexpander spielen, oder ein tool wie M4 verwenden.

Nichtleere Ströme sollen wieder durch `Cons` erzeugt werden. Eine verzögernde Berechnung des Stroms erhalten wir, wenn unser Datentyp `stream` so implementiert wird:

```
datatype 'a stream = Nil
                   | Cons of 'a * (unit -> 'a stream);
```

Zudem legen wir fest, daß im folgenden

```
Consq(h,t)
```

als Abkürzung für

```
Cons(h,delay t)
```

und damit für

```
Cons(h,(fn () => t))
```

steht. Damit ergibt sich klarerweise

```
fun head Cons(h,_) = h;
fun tail Cons(_,t) = force t;
```

Mit dieser neuen Realisierung der Operationen auf Strömen werden immer nur soviele Elemente eines Stromes wirklich ausgerechnet werden, wie durch aufeinanderfolgende `tail`-Aufrufe erzwungen wird.[15]

`map` und `filter` können jetzt neu programmiert werden, so daß sie auch im verzögernden Berechnungsmodus arbeiten:

```
fun map f Nil = Nil
  | map f xs  = Consq(f (head xs),map f (tail xs));
fun filter p Nil = Nil
  | filter p xs =
    let val x = head xs
    in
       if p x
          then Consq(x,filter p (tail xs))
          else filter p (tail xs)
    end;
```

Die Intervallfunktion wäre jetzt entsprechend wie folgt definiert:

```
fun interval a b = if a>b then Nil else Consq(a,interval a+1 b);
```

Nachfolgend zeigen wir ein Beispiel für eine Reduktion mit den Abkürzungen `inter` für `interval`, `sq` für `square`, `hd` für `head`, `tl` für `tail` und ⟦e⟧ für `(fn () =>` e`)`.

```
hd(tl(map sq (inter 1 1000)))
→   hd(tl(map sq (Cons(1, ⟦(inter 1+1 1000)⟧))))
→   hd(tl(Cons(sq(hd(Cons(1,...))), ⟦map sq (tl(Cons(1,...)))⟧)))
→   hd(tl(Cons(sq(1), ⟦map sq (tl(Cons(1,...)))⟧)))
→*  hd(tl(Cons(1, ⟦map sq (tl(Cons(1,...)))⟧)))
→   hd(⟦map sq (tl(Cons(1,...)))⟧ ())
→   hd(map sq (tl(Cons(1, ⟦(inter 1+1 1000)⟧))))
→   hd(map sq (⟦(inter 1+1 1000)⟧ ()))
→   hd(map sq (inter 1+1 1000))
→   hd(map sq (inter 2 1000))
→   hd(map sq Cons(2,⟦inter 2+1 1000⟧))
→   hd(Cons(sq(hd(Cons(2,⟦inter 2+1 1000⟧))),⟦map sq (tl(Cons(2,...)))⟧)
→   hd(Cons(sq(2),⟦map sq (tl(Cons(2,...)))⟧))
→*  hd(Cons(4,⟦map sq (tl(Cons(2,...)))⟧))
→   4
```

Einen erheblichen Nachteil unserer Neuimplementierung von Strömen wollen wir nicht verschweigen. Es kann jetzt vorkommen, daß ein Element eines Stromes mehr-

[15] Tatsächlich ist es bei nichtleeren Strömen immer ein Element mehr als es aufeinanderfolgende `tail`-Aufrufe gibt, das berechnet wird.

fach berechnet wird, dann nämlich, wenn mehrfach mit `tail` auf dasselbe Reststück zugegriffen wird. Jedesmal erfolgt dann ja ein `force`. Dies kann durch Verwendung der *Memotechnik* vermieden werden, die mithilfe von Zeigervariablen[16] einmal berechnete Werte abspeichert und bei jedem weiteren Aufruf auf diese zurückgreift. Damit sieht die nochmals verbesserte Implementierung wie folgt aus:

```
datatype 'a stream = Nil
                   | Cons of 'a * (unit -> 'a stream) ref;
fun tail(Cons(_,f)) =
    let val res = (!f)()
     in (f:= (fn () => res); res)
    end;
```

```
Consq(h,t)
```

ist nun ein Makro für

```
Cons(h,ref(fn () => t))
```

Die übrigen Implementierungen bleiben unverändert.

Damit diese Implementierung greift, müssen wir einige der grundlegenden Funktionen neu schreiben. Für `concat`, welches wir bei der Realisierung von Multimengenkomprehensionen verwendet haben, schreiben wir

```
fun concat Nil = Nil
  | concat(Cons(Nil,ss)) = concat (ss ())
  | concat(Cons(Cons(x,xs),ss)) =
           Consq(x,concat(Cons(xs(),ss)));
```

Man beachte, daß `concat` bei einem Strom ss mit unendlichen Strömen (vgl. nächstes Kapitel) als Elementen unfair konkateniert: Elemente eines Stromes die hinter einen unendlichen Element in ss stehen, können mit keiner endlichen Folge von `tail`-Aufrufen aus `concat(`ss`)` extrahiert werden. Man sollte stattdessen `concat` so schreiben, daß das bekannte Zickzack-Schema, mit dem man z.B. zeigt, daß $N * N$ abzählbar ist, geeignet verallgemeinert wird (Übung!). Da hierbei die Reihenfolge der Stromelemente, je nach Implementierung von `concat`, unvorhersehbar ist, haben wir im Zusammenhang mit Komprehension von Multimengen, und nicht von Listen gesprochen.

Unendliche Ströme. Interessanterweise vermeidet unsere neue Realisierung von Strömen nicht nur, daß Elemente, die zur Berechnung des Endergebnisses einer Anwendung nicht gebraucht werden, nicht berechnet werden, sie erweitert auch das logische Konzept der Ströme. Wenn ein Strom potentiell unendlich ist, aber in jeder Verwendung des Stromes nur endlich viele Anfangselemente inspiziert werden müssen, so haben wir jetzt keine Terminierungsprobleme. Der Strom der Zahlen im rechtsoffenen Intervall $[a, \infty[$ ist durch

[16] Zu den SML-Konstruktionen für Zeigerobjekte: Mit **`ref`** e wird ein Zeiger auf den Wert von e erzeugt. $!x$ liefert den Wert, auf den der Zeiger x zeigt. Zuweisungen von Zeigern e an Zeigervariablen x sind von der Form x `:=` e. Eine Zuweisung hat () als Ergebnis. Ausdrucksfolgen $(e_1; \ldots; e_n)$, die im Zusammenhang mit Zuweisungen häufig auftreten, haben als Wert den Wert des letzten Ausdrucks. Zeigervariablen dürfen keinen polymorphen Typ haben.

```
fun integersFrom a = Consq(a,integersFrom (1+a))
```

in Lisp nun wohldefiniert, jeder Aufruf, etwa

```
- integersFrom 2
```
val it = Cons (2,fn) : int stream

produziert ja zunächst nur das erste Element des unendlichen Stroms, zusammen mit der Vorschrift zur Berechnung der weiteren Elemente. Im Beispiel steht `fn` für einen λ-Ausdruck mit dem Rumpf `integersFrom (1+a)` in dem die freie Variable a an 2 gebunden ist.

Auf unendlichen Strömen können wir auch Funktionen programmieren, die unendliche Ströme als Ergebnisse liefern. Die elementweise Addition von zwei Strömen ist ein Beispiel.

```
fun addStreams(Nil,s2) = s2
  | addStreams(s1,Nil) = s1
  | addStreams(Cons(x1,s1), Cons(x2,s2)) =
              Consq(x1+x2:int,addStreams(s1(),s2()));
```

Eine Anwendung hiervon wäre eine Neudefinition der Fibonnacci-Zahlen, diesmal als Strom.

```
val fibs =
  let fun fibs1 () =
    Consq(0,
          Consq(1,
                addStreams(fibs1(),tail(fibs1()))))
  in fibs1()
end;
```

(Leider können wir nicht einfach

```
val fibs =
    Consq(0,
          Consq(1,
                 addStreams(fibs,tail fibs)));
```

schreiben, weil `val`-Definitionen nicht rekursiv sind.)

```
- fibs;
val it = Cons (0,fn) : int stream
- tail it;
val it = Cons (1,fn) : int stream
- tail it;
val it = Cons (1,fn) : int stream
- tail it;
val it = Cons (2,fn) : int stream
- tail it;
val it = Cons (3,fn) : int stream
- tail it;
val it = Cons (5,fn) : int stream
```

Hier wird in der Terminologie von Strömen ausgesagt, daß, bis auf die Anfangselemente 0 und 1, jedes Element der Folge der Fibonnacci-Zahlen sich als die Summe der beiden vorhergehenden berechnet. Addieren wir den Strom der Fibonnacci-Zahlen mit sich selbst um eins nach links verschoben, so erhalten wir die Folge der Fibonnacci-Zahlen ab ihrem dritten Element aufwärts. Daß das SML-Programm funktioniert liegt daran, daß immer, wenn `addStreams(f,(tail f)` mit irgendeinem Rest `f` der Fibonnacci-Zahlen ausgeführt wird, man die ersten beiden Elemente von `f`, und damit auch das erste Element von `(tail f)`, bereits berechnet hat. Nur diese beiden Elemente werden gebraucht, um das erste Element von `addStreams(f,(tail f)` zu berechnen.

Als abschließenden Beispiel wollen wir den Primzahltest nach der Methode des Siebs des Eratosthenes stromorientiert programmieren. Wir definieren dazu die Folge aller Primzahlen. Diese ist rekursiv so aufgebaut, daß jedes Endstück der Folge möglichst viele ganze Zahlen in aufsteigender Reihenfolge enthält, die nicht durch ihr erstes Element teilbar sind, dieses natürlich ausgenommen. Erfüllt wirklich jedes Endstück einer Folge ansteigender Zahlen diese Eigenschaft und soll der Strom mit 2 beginnen, so muß der Strom tatsächlich gerade alle Primzahlen enthalten. Wir benötigen hierzu die Standardfunktion `remainder`, die den Rest bei ganzzahliger Division liefert.

```
fun sieve n =
    Consq(n,filter (fn x => not(x mod n = 0))
                   (sieve (1+n)));
val primes = sieve 2;
```

Der Nachteil dieser Definition ist, daß jede Zahl mit allen kleineren Primzahlen auf Unteilbarkeit getestet wird. Folgende Lösung kommt mit erheblich weniger solchen Tests aus. Hier sind die Primzahlen `primes` diejenigen ganzen Zahlen ≥ 2, die prim sind. Die Primzahleigenschaft `prime` einer Zahl testet man, indem man sie durch alle kleineren Primzahlen zu teilen versucht. Wichtig ist nun hierbei zu wissen, daß man für jede Zahl n nur Teiler kleiner oder gleich $\sqrt{n}$ testen muß. Denn ist n eine Primzahl, so ist sie kleiner als das Quadrat der nächstkleineren Primzahl. Mit dieser Erkenntnis kann man `primes1` und den Test `prime` in gegenseitiger Abhängigkeit definieren.

```
fun primes1 () = Consq(2,filter prime (integersFrom 3))
and    prime n =
       let fun iter ps =
           if (square (head ps)) > n then true
           else if (n mod (head ps)) = 0 then false
           else (iter (tail ps))        (*)
       in iter (primes1 ())
       end;
val primes = primes1 ();
```

Daß diese gegenseitige Rekursion wohlfundiert ist, liegt also an dem zahlentheoretischen Zusammenhang $p_{n+1} < p_n^2$ für die Folge p_i der Primzahlen. Man kann sich daher sicher sein, daß man nicht mehr Primzahlen aus `primes` in `(*)` anfordert, als es solche unterhalb von n gibt.

3.9 Typinferenz

In diesem Abschnitt beschreiben wir Typinferenz für eine sehr kleine Teilsprache von SML, die allerdings groß genug ist, um ganz SML darin mit wenig Aufwand abbzubilden. Die abstrakte Syntax[17] der Ausdrücke e dieser Sprache MML (von „Mini-ML“) ist

$$e ::= x \mid (e\ e) \mid \lambda x \,.\, e \mid \texttt{let}\ x = e\ \texttt{in}\ e'$$

mit Variablen x aus einer Menge X von Variablen. Das `let` soll, wie in SML, nicht rekursiv sein. Für rekursive Funktionsdefinitionen werden wir später noch die Existenz eines Fixpunktoperators voraussetzen, wie wir überhaupt, um Beispiele machen zu können, die Existenz verschiedener globaler Variablenbindungen voraussetzen werden. Wenn das `let` also nicht rekursiv ist, so fragt man sich gleich, warum man es in so einer theoretischen Sprache überhaupt betrachtet, wo doch operational $\texttt{let}\ x = e\ \texttt{in}\ e'$ äquivalent mit $(\lambda x.e')e$ ist. Der Grund wird sich aus dem Typsystem ergeben.

3.9.1 Typausdrücke, Typannahmen

Unter *Typinferenz* verstehen wir das Ableiten von Aussagen der Form

$$\Gamma \vdash e : \tau \ ,$$

die ausdrücken sollen, daß im Kontext Γ der Ausdruck e den Typ τ hat. τ ist ein Term aus der Menge der Typausdrücke, die unten definiert wird. Ein *Kontext* Γ ist eine Folge $x_1 : \tau_1, \ldots, x_n : \tau_n$ von *Typannahmen* über die Variablen x_i, wobei für eine Variable höchstens eine Typannahme vorkommen darf. Kontexte werden verwendet, um die jeweils gültigen Bindungen der freien Variablen zu repräsentieren. Dies bedeutet im Falle der Typinferenz, daß wir wissen wollen, was der Typ des Ausdrucks ist, an den die Variable gebunden ist. Wären wir auch an Auswertung interessiert, so würden wir den Wert der Variablen im Kontext spezifizieren. Es bezeichne $\Gamma, x : \tau$, für jeden Kontext Γ, den Kontext $(\Gamma \setminus \{x : \tau'\}) \cup \{x : \tau\}$, bzw. $\Gamma \cup \{x : \tau\}$, wenn $x : \tau'$ in Γ ist, bzw. kein $x : \tau'$ in Γ vorkommt.

Typen werden als Ausdrücke (Terme) über $F = \{\rightarrow\}$ und einer Menge V von Typvariablen gebildet. Ein Typ τ ist also ein Ausdruck der Syntax

$$\tau ::= \alpha \mid \tau \rightarrow \tau \ .$$

mit α in V. Nachfolgend werden wir α, β, γ, usw. als Metavariablen für Typvariablen verwenden. Wie immer wird es von dem Kontext abhängen, was die Bedeutung einer Typvariablen in einem Typausdruck ist. Sie kann die Rolle einer Typkonstanten, wie `int`, `bool`, oder die Rolle eines Typparameters spielen. Dann, wenn wir explizit machen wollen, daß wir einen polymorphen Typ meinen, der mit einer Variablen parametrisiert ist, werden wir explizite Quantoren verwenden. Explizit quantifizierte Typausdrücke nennen wir *Typschemata*. Sie werden folgendermaßen gebildet:

$$\tau_1 ::= \tau_0 \mid \forall \alpha \,.\, \tau_1$$

Wir verwenden im folgenden τ_0, σ_0, usw. als Metavariablen für Typausdrücke ohne Quantoren, τ_1 oder τ usw. für Typschemata. Wir verwenden die Abkürzung $\forall \alpha_1 \ldots \alpha_n$ für $\forall \alpha_1 \ldots \forall \alpha_n$.

[17] Bzgl. der konkreten Syntax werden wir die üblichen Konventionen für das Setzen von Klammern oder das Verwenden von Infixnotation gebrauchen.

Von Typschemata können generische Typinstanzen gebildet werden, indem eine gebundene Typvariable instanziiert wird. Seien

$$\tau = \forall \alpha_1 \dots \alpha_m \, . \, \tau_0$$

und

$$\tau' = \forall \beta_1 \dots \beta_n \, . \, \tau_0'$$

Typschemata, $m, n \geq 0$. Dann heißt τ' eine *Instanz* von τ, i.Z. $\tau' \leq \tau$,[18] falls es eine Substitution σ gibt, so daß $\tau_0' = \tau_0 \sigma$, $dom(\sigma) \subseteq \{\alpha_1, \dots, \alpha_m\}$,[19], und die β_i nicht frei in τ auftreten. Beispielsweise ist

$$\forall \beta, \delta \, . \, ((\texttt{int} \rightarrow \delta) \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta)$$

eine generische Instanz von

$$\forall \alpha, \beta \, . \, (\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta),$$

nicht aber

$$\forall \gamma \, . \, (\texttt{int} \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta),$$

da wir in diesem Fall nach der Instanziierung die vorher freie Variable γ polymorph machen. Typinstanziierung elimiert also eine Reihe von Qantoren, substitutiert die zugehörigen Variablen durch konkrete Typausdrücke und quantifiziert hinterher evtl. über einige der durch die Substitution neu eingeführten Variablen. Im ersten Beispiel ist $\sigma : \alpha \mapsto (\texttt{int} \rightarrow \delta)$ und der neue Quantor bindet die neu eingeführte Variable δ.

3.9.2 Das Damas/Milner-Typinferenzsystem

Das Damas/Milner Typinferenzsystem für MML besteht aus folgenden Regeln:

$$\text{(Taut)} \qquad \Gamma,\ x : \tau \vdash x : \tau$$

$$\text{(Inst)} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau'} \qquad [\tau' \leq \tau]$$

$$\text{(Gen)} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \forall \alpha . \tau} \qquad [\alpha \notin FV(\Gamma)]$$

$$\text{(App)} \qquad \frac{\Gamma \vdash M : \tau_0' \rightarrow \tau_0 \qquad \Gamma \vdash N : \tau_0'}{\Gamma \vdash (MN) : \tau_0}$$

$$\text{(Abs)} \qquad \frac{\Gamma, x : \tau_0' \vdash M : \tau_0}{\Gamma \vdash (\lambda x . M) : \tau_0' \rightarrow \tau_0}$$

$$\text{(Let)} \qquad \frac{\Gamma \vdash M : \tau' \qquad \Gamma, x : \tau' \vdash N : \tau}{\Gamma \vdash (\texttt{let}\ x = M\ \texttt{in}\ N) : \tau}$$

[18] τ' ist kleiner (weniger allgemein) als τ

[19] $dom(\sigma)$ den *Definitionsbereich* („domain") einer Substitution, nennt man diejenige Teilmenge von Variablen x, für die $x\sigma \neq x$ gilt.

Hierbei enthält Γ Typannahmen über beide Arten von Variablen, die, die durch λ und die, die durch *let* deklariert wurden. Die ersten sind nicht polymorph, die zweiten dürfen polymorph sein und spielen daher die gleiche Rolle wie Konstanten. Eine explizite syntaktische Unterscheidung zwischen Konstanten und Variablen braucht es daher nicht. Ihre Typen können diese Unterscheidung ausdrücken. Das Standardbeispiel für eine polymorphe Funktion ist die Identität. Sie kann auf sich selbst angewendet werden, und diese Selbstanwendung ist typisierbar. Es ist ableitbar:

$$\vdash \texttt{let } i \texttt{ = } \lambda x.x \texttt{ in } (i\ i) \ :\ \alpha \to \alpha$$

3.9.3 Erweiterung mit Produkten, Konditional, Rekursion

SML kann zu einer richtigen Programmiersprache erweitert werden, wenn man das Typinferenzsystem geeignet eweitert.

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau'}{\Gamma \vdash (M, N) : \tau * \tau'} \quad \text{(Prod)}$$

$$\Gamma \vdash \texttt{fst} : \forall\alpha\forall\beta.(\alpha * \beta) \to \alpha \quad \text{(fst)}$$

$$\Gamma \vdash \texttt{snd} : \forall\alpha\forall\beta.(\alpha * \beta) \to \beta \quad \text{(snd)}$$

$$\Gamma \vdash \texttt{if} : \forall\alpha.\ \texttt{bool} \ \to \alpha \to \alpha \to \alpha \quad \text{(if)}$$

$$\Gamma \vdash \ = : \forall\alpha.(\alpha * \alpha) \to \texttt{bool} \quad \text{(eq)}$$

$$\Gamma \vdash \texttt{Y} : \forall\alpha.(\alpha \to \alpha) \to \alpha \quad \text{(fix)}$$

$$\Gamma \vdash \texttt{nil} : \forall\alpha.\ \alpha\,\texttt{list} \quad \text{(nil)}$$

$$\Gamma \vdash :: :\ \forall\alpha.\alpha \to \texttt{list}\ \alpha \to \alpha\,\texttt{list} \quad \text{(cons)}$$

$$\Gamma \vdash \texttt{hd} : \forall\alpha.\ \alpha\,\texttt{list} \to \alpha \quad \text{(hd)}$$

$$\Gamma \vdash \texttt{tl} : \forall\alpha.\ \alpha\,\texttt{list} \to \alpha\,\texttt{list} \quad \text{(tl)}$$

Bemerkungen:

- `Y` ist der Fixpunktkombinator zur Definition von rekursiven Funktionen, vgl. nächstes Kapitel. In diesem Zusammenhang wollen wir auch folgende Notation gebrauchen und

  $$\texttt{letrec } x \texttt{ = } M \texttt{ in } N$$

  als Abkürzung für

  $$\texttt{let } x \texttt{ = Y}(\lambda x.M) \texttt{ in } N$$

  verwenden.

- Was wir exemplarisch für Listen vorgeführt haben, läßt sich in analoger Weise für alle Konstruktortypen machen.

Mit dem erweiterten System leitet man beispielsweise für

```
letrec map = λf. λl. if l = nil then nil else (f(hd l)).(map f (tl l))
```

den Typ

$$\texttt{map} : (\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$$

ab.

Theorem 3 (Milner) *Jeder typisierbare ML-Ausdruck hat einen bzgl. $\geq$ allgemeinsten Typ.*

Der Beweis liefert ein Typinferenzverfahren für ML-Terme. Dies rechtfertigt in Nachhinein, warum Variablentypen nicht deklariert werden brauchen.

Für reine λ-Ausdrücke ohne `let` ergibt sich jede mögliche Typisierung als die Lösung eines Gleichungssystems, wobei für jeden Teilausdruck e' eines Ausdruckes e eine Typmetavariable $\tau_{e'}$ verwendet wird. Die Gleichungen ergeben sich nach folgendem Schema

$$\begin{array}{lcll} \tau_e & = & (\tau_{e'} \to \tau_{(ee')}) & \text{(Applikation)} \\ \tau_{\lambda x.e} & = & (\alpha^x \to \tau_e) & \text{(Abstraktion)} \\ \tau_x & = & \alpha^x & \text{(Var)} \end{array}$$

Wir nehmen an, daß gebundene Variablen x eindeutig benannt sind, d.h. nur in jeweils einem λ-Ausdruck deklariert werden. α^x ist dann eine x eineindeutig zugeordnete Typvariable. Für `if`-Ausdrücke gibt es 3 Gleichungen:

$$\left.\begin{array}{lcl} \tau_{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)} & = & \tau_{e_2} \\ \tau_{e_3} & = & \tau_{e_2} \\ \tau_{e_1} & = & \texttt{bool} \end{array}\right\} \quad \text{(if)}$$

Schließlich gibt es für jedes <u>Auftreten</u> x^a einer polymorphen freien Variablen x vom Typ $\forall \alpha_1 \cdots \alpha_n.\ \tau_o$ eine Gleichung

$$\tau_{x^a} = \tau_0[\alpha_1^a/\alpha_1, \ldots, \alpha_n^a/\alpha_n] \qquad \text{(Konst)}$$

wobei die α_i^a paarweise verschiedene, neue Variablennamen sind.

Wir machen ein Beispiel. Sei folgender Ausdruck gegeben, wobei die Superskripte Namen für die jeweiligen Teilausdrücke darstellen sollen:

$$\begin{array}{lcl} e & = & \{\lambda map.\lambda f.\lambda l. \\ & & \qquad [\quad \texttt{if}\ (l =^6 nil^1)^{15} \\ & & \qquad\qquad \texttt{then}\ nil^2 \\ & & \qquad\qquad \texttt{else}\ [(f(hd^3\ l)^7)^8 ::^5 (map\ f\ (tl^4\ l)^9)^{10})^{11}]^{12} \\ & & \qquad]^{13} \\ & & \}^{14} \end{array}$$

Es ergibt sich das Gleichungssytem

$$\begin{array}{lcl} \tau_1 & = & \alpha^1\,\texttt{list} \\ \tau_2 & = & \alpha^2\,\texttt{list} \\ \tau_3 & = & \alpha^3\,\texttt{list} \to \alpha^3 \end{array}$$

$$
\begin{aligned}
\tau_4 &= \alpha^4\,\texttt{list} \to \alpha^4\,\texttt{list}\\
\tau_5 &= \alpha^5 \to \alpha^5\,\texttt{list} \to \alpha^5\,\texttt{list}\\
\tau_6 &= \alpha^6 \to \alpha^6 \to \texttt{bool}\\
\tau_6 &= \tau_1 \to \tau_1 \to \tau_{15}\\
\tau_{15} &= \texttt{bool}\\
\tau_3 &= \tau_l \to \tau_7\\
\tau_f &= \tau_7 \to \tau_8\\
\tau_4 &= \tau_l \to \tau_9\\
\tau_f &= \tau_9 \to \tau_{10}\\
\tau_{map} &= \tau_{10} \to \tau_{11}\\
\tau_5 &= \tau_8 \to \tau_{11} \to \tau_{12}\\
\tau_1 &= \tau_{12}\\
\tau_{13} &= \tau_2\\
\tau_{14} &= \tau_{map} \to \tau_f \to \tau_l \to \tau_{12}
\end{aligned}
$$

aus dem sich die allgemeinste Lösung

$$\tau_{16} = ((\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}) \to ((\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list})$$

ergibt.

Wir suchen Lösungen, in denen nur Typvariablen und Typausdrücke ohne $\forall$ vorkommen. Typausdäucke sind terme erster Stufe, die aus den Typkonstuktoren ($\to$, $*$, `list`, usw.) gebildet werden.

Ein allgemeines Resultat aus der Logik über Terme erster Stufe besagt, daß solche Gelichungssysteme entweder keine oder eine allgemeinste Lösung (bzgl. Subsumptionsordnung) besitzen. Die Subsumptionsordnung auf Typtermen entspricht im wesentlichen der Instanziierungsordnung $\leq$ auf Typschemata. Dies rechtfertigt die Aussage von Satz 3.

Für `let`-Ausdrücke kann Typinferenz wie folgt implementiert werden: Sei $l \equiv$ `let` $x = e'$ `in` e''.

1. Finde die allgemeinste Lösung des Typgleichungssystems für e'.

2. Quantifiziere über alle freien Variablen in τ_0 (Generalisierung): dies liefert $\tau_1 = \forall\alpha_1 \cdots \alpha_n,\ \tau_0$.

3. Inferiere den allg. Typ von e'' in einer um die polymorphe Konstante $x : \tau_1$ erweiterten Umgebung.

4. Das Ergebnis ist der allgemeinste Typ von e.

Betrachte als Beispiel

```
let map = Y(e) in map id 1::nil
```

mit e wie im letzten Beispiel (Übung), oder die bereits erwähnte Selbstanwendung der Identität

$$\texttt{let id = } \lambda x.x \texttt{ in id}^1\texttt{(id}^2\texttt{)}.$$

Hier ergibt sich folgendes:

Schritt 1

$$\begin{array}{lcll} \tau_{\lambda x.x} & = & \alpha^x \rightarrow \tau_x & \text{(Abs)} \\ \tau_x & = & \alpha^x & \text{(Var)} \end{array}$$

Damit ist $\tau_{\lambda x.x} = \alpha^x \rightarrow \alpha^x$ die allgemeinste Lösung

Schritt 2

$$\tau_1 = \quad \forall \alpha^x . \alpha^x \rightarrow \alpha^x \equiv \quad \forall \alpha . \alpha \rightarrow \alpha$$

Schritt 3

$$\begin{array}{lcll} \tau_{id^1} & = & \alpha^1 \rightarrow \alpha^1 & \text{(Konst)} \\ \tau_{id^2} & = & \alpha^2 \rightarrow \alpha^2 & \text{(Konst)} \\ \tau_{id^1} & = & (\tau_{id^2} \rightarrow \tau_{id^1(id^2)}) & \text{(Appl)} \end{array}$$

Die Lösung muß

$$(\alpha^1 \rightarrow \alpha^1) = ((\alpha^2 \rightarrow \alpha^2) \rightarrow \tau_{id^1(id^2)})$$

erfüllen. Als Lösungssubstitution ergibt sich daher

$$\alpha^1 \mapsto (\alpha^2 \rightarrow \alpha^2)$$

zusammen mit

$$\tau_{id^1(id^2)} \mapsto \alpha^1 \mapsto (\alpha^2 \rightarrow \alpha^2).$$

Dieses ist der allgemeinste Typ im betrachteten Beispiel.

3.10 Reduktionssemantik für MML

Um die Strategie „innen zuerst“ darzustellen, verwenden wir eine erweiterte Ausdrucksmenge

$$e_1 ::= e \mid e!$$

wobei $e!$ anzeigen wird, daß e in Normalform ist. Mit $erase(e_1)$ sei dann der Ausdruck bezeichnet, der aus e_1 durch Streichen aller „!“ entsteht.

$(f\ e_1! \cdots e_n!) \rightarrow e_0!$ (Konst)
falls f vordef. Konstante $f : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0$,
τ_0 nicht funktional und $f : (e_1, \ldots, e_n) \mapsto e_0$

$(\lambda x.e) \rightarrow (\lambda x.e)!$ (lambda)

$(\lambda x.e')\ e! \rightarrow e'[e!/x]$ (beta)

$\texttt{if true then } e_1 \texttt{ else } e_2 \rightarrow e_1$ (if1)

$\texttt{if false then } e_1 \texttt{ else } e_2 \rightarrow e_2$ (if2)

$$\frac{e \rightarrow e' \quad e' \rightarrow e''}{e \rightarrow e''} \quad \text{(Trans)}$$

$$\frac{e \rightarrow e'}{(e_1\ e) \rightarrow (e_1\ e')} \quad \text{(Appl1)}$$

$$\frac{e \rightarrow e'}{(e\ e_1) \rightarrow (e'\ e_1)} \quad \text{(App2)}$$

$$\frac{e \rightarrow e'}{\texttt{if } e \cdots \rightarrow \texttt{if } e' \cdots} \quad \text{(if3)}$$

$$\frac{e \rightarrow e'}{\texttt{let } x = e \cdots \rightarrow \texttt{let } x = e' \cdots} \quad \text{(let1)}$$

$\texttt{let } x = e! \texttt{ in } e' \rightarrow e'[e!/x]$ (letbeta)

$Y(\lambda f.e) \rightarrow e[Y(\lambda f.e)/f]$ (fix)

Theorem 4 (Subjektreduktion) *Sei $e \rightarrow e'$ und $\vdash erase(e) : \tau$. Dann gilt auch $\vdash erase(e') : \tau$.*

Wohlgetypte Programme liefern also keinen Typfehler zur Laufzeit. Typinformation wird zur Laufzeit nicht benötigt, in den Ausdrücken der Reduktionssemantik kommt keine Typinformation vor. Das SML-Typsystem ist daher ein *statisches Typsystem*.

3.11 Elementare Konzepte zur Datenabstraktion

Folgende Gesichtspunkte sind motivierend für die Konzepte von Datenabstraktion in Programmiersprachen.

- Softwaresysteme haben mindestens zwei Ebenen. Auf der untersten Ebene werden die Datenstrukturen definiert. Auf der obersten Ebene befinden sich die Applikationsprogramme, die diese Datenstrukturen verwenden.

- Die Repräsentation der Daten soll von ihrer Verwendung unabhängig sein. Insbesondere soll erstere änderbar sein, ohne daß man die Applikationen umschreiben muß.

- *Datentypen* sind Datenstrukturen **zusammen** mit den darauf definierten und für Anwendungsprogrammen exportierten Operationen. Bei *abstrakten Datentypen* sind nur diese Operationen sichtbar. Auf die konkrete Datenstruktur darf von außen (etwa durch Verwendung von Termmustern) nicht zugeggriffen werden.

- Abstrakte Datentypen sind aus der Anwendungsansicht wie mathematische Strukturen (ein Grund für die Bezeichnung „abstrakt“). Ein Beispiel wäre der Datentyp der endlichen Mengen mit den Operationen $\emptyset, \cup, \cap, \{.\}$. Unabhängig von der internen Realisierung, etwa durch Listen oder Hashtabellen, sind für die externen Applikationen nur die Eigenschaften der abstrakten Struktur relevant, etwa daß $s \cup s = s$, auch wenn die interne Realisierung von s und $s \cup s$ verschieden sind. So könnten im Falle $s \cup s$, ander als bei s, die interne Listenstruktur Elemente mehrfach enthalten. Daß für das Verhalten einer Applikation nur die Gesetzmäßigkeiten der abstrakten Struktur relevant sind, ist der zweite Grund für die Bezeichnung „abstrakt“. Diese Einsichten, die auf Hoare und Zilles zurückgehen, ist Grundlage für das Fachgebiet der algebraischen Spezifikationen. In SML finden sich die zentralen Konzepte Signatur, Struktur (Algebra) und Funktor dieses Gebeites als Sprachkonstruktionen.

3.11.1 Module in SML

Literatur für SML-Konzepte für Datenabstraktion: Paulson-Buch, Kapitel 7.

Methodisch gesehen repräsentieren Signaturen und Strukturen in SML folgende Aspekte von Modularisierung und strukturierter Programmierung:

- Module in SML sind Strukturen (oder Funktoren) und verkörpern Ansammlungen von Deklarationen von Größen, die in irgendeinem Sinn zusammengehören.

- Signaturen spezifizieren die Schnittstellen von Modulen. In SML werden nur Namen und Art und/oder Typ einer Größe spezifiziert. Idealerweise sollten auch semantische Eigenschaften spezifiziert werden können.

- Datenabstraktion entsteht in SML durch das explizite Versehen von Strukturen mit Signaturconstraints. Dann werden Objekte, die in der Struktur deklariert aber nicht in der Signatur erwähnt werden, nicht exportiert. Etwaige Konstruktoreigenschaften exportierter Funktionen c werden dann ebenfalls nicht exportiert, so daß extern c nicht in einem Muster auftreten kann.

- In SML is keine perfekte Datenabstraktion gewährleistet, weil man extern auf Werte von modulinternen (Gleichheits-)Typen das Gleichheitsprädikat anwenden kann.

- Signaturen wie Stukturen sind in nichttrivialen Fällen stukturiert, d.h. enthalten andere Signaturen oder Strukturen als Komponenten oder benutzen bereits definierte Signaturen oder Stukturen.

- Die Strukturierung von Schnittstellen (Spezifikation) und Strukturen (Implementierung) sind i.allg. verschieden. Im Paulson-Buch findet man als ein einfaches Beispiel mit dieser Eigenschaft die arrays (Kap. 7.10): Die SIgnatur (ARRAY verwendet nur vordefinierte Typen (`int`) und sonst keine weiteren freien Variablen. Die Struktur `Array` hingegegen basiert auf der Struktur `ArrayOps` und diese wiederum hat die Stuktur `Tree` als Substruktur. Softwareentwicklungsmethoden, die diese Verschiedenheit nicht unterstützen, sind in der Praxis ungeeignet.

Im folgenden wollen wir für einen einfachen Spezialfall von Signaturen und Stukturen (keine Hierarchien, nur monomorphe Typen und Werte dieser Typen als Größen in

Stukturen) eine algebraische Semantik geben und das Problem der Verifikation von Implementierungsbeziehungen zwischen Spezifikation und Implementierung betrachten,

3.11.2 Algebraische Sichtweise von Signaturen und Strukturen

Im Kontext sogenannter algebraischer Spezifikationen sind *Signaturen* Paare $\Sigma = (S, \Omega)$, wobei S eine Menge (von *Sorten*) und Ω eine Menge von typisierten Funktionssymbolen

$$f : s_1 * \cdots * s_n \rightarrow s_0, \ n \geq 0, \ s_i \in S$$

sind. Dies entspricht der speziellen Form von ML-Signaturen

```
signature Σ =
  sig
    ...
    type s[20]
    ...
    ...
    val f : s1 * ··· * sn -> s0[21]
    ...
end;
```

Die Sorten sind also als **Typnamen** zu verstehen, die für nicht näher spezifizierte Typen stehen.

Gegeben eine Signatur Σ, eine Σ-*Algebra* A besteht dann aus je einer Menge A_s, für jedes $s \in S$, sowie Funktionen

$$f_A : \ A_{s_1} * \cdots * A_{s_n} \rightarrow A_{s_0}$$

für jedes $f : \ s_1 * \cdots * s_n \rightarrow s_0$ `in` Ω.

3.11.3 Implementierung zwischen zwei Σ-Algebren

Σ-Algebren sind das mathematische Konzept zur Formalisierung von Datentypen. Bei Datentypen gibt es immer zwei Sichtweisen, die des abstrakten Datentyps, wie ihn das Anwendungsprogramm sieht, und den konkreten Datentyp, wie er intern realisiert ist. Den Zusammenhang zwischen diesen beiden Ebenen bildet der in diesem Abschnitt zu besprechende Implementierungsbegriff zwischen Algebren.

Seien A und B zwei Σ-Algebren und $s \in S$ eine Sorte. A *implementiert* B auf s, wenn die nachfolgenden 4 Eigenschaften erfüllt sind:

1. $A_{s'} = B_{s'}$, für alle $s' \neq s$ in S
2. Es gibt eine partielle Funktion $a : A_s \rightarrow B_s$, genannt *Abstraktionsfunktion.*
3. Es gibt ein Prädikat $I \subseteq A_s$, genannt *Repräsentationsinvariante*, so daß für alle x in A_s gilt: ist $I(x)$ erfüllt, so ist $a(x)$ definiert.

[20] `für jedes` $s \in S$ `eine Deklaration`
[21] `für jedes` $f \in \Omega$ `eine solche Deklaration`

4. Für jedes Funktionssymbol

$$f : \underbrace{s * \cdots * s}_{n \geq 0} * \underbrace{s_1 * \cdots s_m}_{\neq s} \to s_0$$

in Ω gilt folgende Eigenschaft:

a) falls $s_0 \neq s$:
$\forall x_1, \ldots, x_n : A_s. \quad \forall_{y_1} : A_{s_1} \; \ldots \; \forall y_m : A_{s_m}.$
$I(x_1) \wedge \ldots \wedge I(x_n) \supset$
$f_A(x_1, \ldots, x_n, y_1, \ldots, y_m) = f_B(a(x_1), \ldots, a(x_n), y_1, \ldots, y_m)$

b) falls $s_0 = s$
$\forall x_1, \ldots, x_n : A_s.\forall y_1 : A_{s_1} \ldots \forall y_m : A_{s_m}.$
$I(x_1) \wedge \ldots \wedge I(x_n) \supset$
$a(f_A(x_1, \ldots, x_n, y_1, \ldots, y_m)) = f_B(a(x_1), \ldots, a(x_n), y_1, \ldots y_m)$
$\wedge I(a(f_A(x_1, \ldots, x_n, y_1, \ldots, y_m)))$

Die Verifikationsbedingungen 1) - 4) garantieren, daß A im intuitiven Sinn B implementiert, wobei letzteres durch die unten formulierte Aussagen formalisiert werden kann. Hierbei brauchen wir den Begriff der Auswertung t_A eines Σ-Terms t in einer Σ-Algebra A, der induktiv wie folgt definiert ist:

a) Ist $t = c : s$, $s \in S$ Konstante, so ist $t_A = c_A$

b) Ist $t = f(t_1, \ldots, t_n)$, so ist

$$t_A = \begin{cases} f_A(t_{1_A}, \ldots, t_{n_A}), & \text{falls alle } t_{i_A} \text{ definiert} \\ \text{undefiniert, sonst} \end{cases}$$

Theorem 5 *Falls die Algebra A die Algebra B auf s implementiert, dann gilt für jeden Grundterm $t \in T_\Omega$ in Operatoren aus Ω:*

a) *falls t von einer Sorte $s' \neq s$ ist:*

$$t_A = t_B$$

b) *falls t von der Sorte s ist:*

$$a(t_A) = t_B$$

Damit sind dann die abstrakt in B geforderten Ergebnisse von Programmen, diejenigen, die von der Implementierung A geliefert wurden, wobei die Entsprechung durch die Abstraktionsfunktion a hergestellt wird.

Beispiel
Wir definieren eine Signatur für Queues, zusammen mit zwei Algebren, von denen die zweite die erste implementiert. Die erste Algebra ist die Variante von Warteschlangen, wie man sie sich konzeptuell vorstellt, nämlich als Listen wo neue Elemente hinte

angefügt werden und alte Elemente vorne entfernt werden. Die Implementierung soll gewissen Effizienzansprüchen genügen und alle Operationen in im Mittel konstanter Zeit realisieren. Zunächst die Signatur $\Sigma = (S, \Omega)$ von Warteschlangenmodulen, wobei

$$
\begin{aligned}
S &= \{\alpha,\ \alpha\ \texttt{queue},\ \texttt{bool}\} \\
\Omega &= \{\texttt{empty}\ :\ \alpha\ \texttt{queue}, \\
&\quad \texttt{enq}:\ \alpha\ \texttt{queue} * \alpha \to \alpha\ \texttt{queue} \\
&\quad \texttt{deq}:\ \alpha\ \texttt{queue} \to \alpha\ \texttt{queue} \\
&\quad \texttt{hd}:\ \alpha\ \texttt{queue} \to \alpha \\
&\quad \texttt{null}:\ \alpha\ \texttt{queue} \to \texttt{bool}\}\ .
\end{aligned}
$$

„Abstrakte" Algebra B B formalisiert unsere Intuition über Warteschlangen als first-in first-out Listen. Partialität von Operationen wird durch die Verwendung eines Fehlerwertes `error` angezeigt.

$$
\begin{aligned}
B_\alpha &= \alpha \cup \{\texttt{error}\} \\
B_{\alpha\texttt{queue}} &= \alpha\ \texttt{list} \cup \{\texttt{error}\} \\
B_{\texttt{bool}} &= \texttt{bool} \cup \{\texttt{error}\} \\
\texttt{empty}_B &= [] \\
\texttt{enq}_B([x_1, \ldots, x_n], x) &= [x_1, \ldots, x_n, x], \qquad x_i, x \in \alpha \\
\texttt{deq}_B([x, x_1, \ldots, x_n]) &= x, \qquad x_i, x \in \alpha \\
\texttt{deq}_B([]) &= \texttt{error} \\
\texttt{null}_B([]) &= \texttt{true} \\
\texttt{null}_B([x, x_1, \ldots, x_n]) &= \texttt{false}, \qquad x_i, x \in \alpha \\
\texttt{hd}_B([x, x_1, \ldots, x_n]) &= x, \qquad x_i, x \in \alpha \\
\texttt{hd}_B([]) &= \texttt{error}
\end{aligned}
$$

für alle anderen Argumente ergebe sich immer `error` als Ergebnis.

Konkrete Algebra A: Trägermengen Die konkrete Realisierung verwendet zwei Listen, um sowohl `enq` als auch `deq` in konstanter Zeit zu realisieren.

$$
\begin{aligned}
A_\alpha &= \alpha \cup \{\texttt{error}\} \\
A_{\alpha\texttt{queue}} &= (\alpha\ \texttt{list} * \alpha\ \texttt{list}) \cup \{\texttt{error}\} \\
A_{\texttt{bool}} &= \texttt{bool} \cup \{\texttt{error}\}
\end{aligned}
$$

Nachfolgend seien mit x, x_i, y_i immer Elemente aus α bezeichnet.

Abstraktionsfunktion:

$$a:\ A_a\texttt{queue} \to B_a\texttt{queue}$$

sei gegeben durch

$$a([x_1 \cdots x_n],\ [y_1 \cdots y_m]) = x_1 \cdots x_n\ y_m \cdots y_1,$$

falls nicht $n = 0$ und $m > 0$, durch

$$a(\texttt{error}) \;= \texttt{error}$$

(letzteres modelliert, daß Partialität richtig implementiert wird) und a undefiniert, sonst.

Repräsentationsinvariante:

$$I \subseteq A_\alpha\texttt{queue}$$

sei definiert als

$$I([x_1 \cdots x_n],[y_1 \cdots y_m]) \Leftrightarrow (n = 0 \supset m = 0)$$

und

$$I(\texttt{error}) \;\Leftrightarrow \top \;.$$

Definition der Funktionen:
Für beliebige x, x_i und y_i in $A_\alpha\texttt{queue}$ gelte:

$$\texttt{empty}_A = ([],[])$$

$$\texttt{enq}_A(([x_1,\ldots,x_n],[y_1,\ldots,y_m]),x) = \texttt{norm}(([x_1,\ldots,x_n],[x,y_1,\ldots,y_m])) \;,$$

$$\texttt{deq}_A\;(([x_1,\ldots,x_n],[y_1,\ldots,y_m])) = \begin{cases} (([x_2,\ldots,x_n],[y_1,\ldots,y_m])), & \text{falls } n \geq 1 \\ \texttt{error}, & \text{sonst} \end{cases}$$

$$\texttt{null}_A\;(([x_1,\ldots,x_n],[y_1,\ldots,y_m])) = \begin{cases} \texttt{true}, & \text{falls } n = 0 \\ \texttt{false}, & \text{sonst} \end{cases}$$

$$\texttt{hd}_A\;(([x_1,\ldots,x_n],[y_1,\ldots,y_m])) = \begin{cases} x_1, & \texttt{falls } n \geq 1 \\ \texttt{error}, & \texttt{sonst} \end{cases}$$

In allen anderen Fällen ergebe sich `error` als Ergebnis. Die Hilfsfunktion `norm` sei hierbei durch

$$\texttt{norm}\;(([x_1,\ldots,x_n],[y_1,\ldots,y_m])) = \begin{cases} ([x_1,\ldots,x_n,y_m,\ldots,y_1],[]), & \text{falls } n = 0 \\ ([x_1,\ldots,x_n],[y_1,\ldots,y_m)], & \text{sonst} \end{cases}$$

gegeben.

Wir müssen nun die einzelnen Bedingungen aus 4. verifizieren, was keine Probleme macht. Z.B. für `deq` haben wir zu zeigen

$$\forall q \in A_a\texttt{queue}: \; I(q) \supset \; a(\texttt{deq}_A(q)) = \texttt{deq}_B(a(q)) \;\wedge\; I(a(\texttt{deq}_A(q)))$$

Sei also zunächst $q = ([x_1,\ldots,x_n],[y_1,\ldots,y_m])$. Ist $I(q)$ wahr, so gilt entweder $n = m = 0$ oder $n \geq 1$. Ist $n = m = 0$, so ist

$$a(\texttt{deq}_A(q)) = a(\texttt{error}) = \texttt{error} = \texttt{deq}_B([]) = \texttt{deq}_B(a(q))$$

und

$$I(a(\texttt{error})) \Leftrightarrow I(\texttt{error})$$

was man leicht sieht. Ist $n = 1$, so ist

$$\begin{aligned}
a(\texttt{deq}_A(q)) &= a(\texttt{norm}(([], [y_1, \ldots, y_m]))) \\
&= a(([y_m, \ldots, y_1], [])) \\
&= [y_m, \ldots, y_1] \\
&= \texttt{deq}_B([x_1, y_m, \ldots, y_1]) \\
&= \texttt{deq}_B(a(q)).
\end{aligned}$$

Ist $n > 1$, so ist

$$\begin{aligned}
a(\texttt{deq}_A(q)) &= a(\texttt{norm}(([x_2, \ldots, x_n], [y_1, \ldots, y_m]))) \\
&= a(([x_2, \ldots, x_n], [y_1, \ldots, y_m])) \\
&= [x_2, \ldots x_n, y_m, \ldots, y_1] \\
&= \texttt{deq}_B([x_1, \ldots, x_n, y_m, \ldots, y_1]) \\
&= \texttt{deq}_B(a(q)).
\end{aligned}$$

Außerdem gilt $I(a(\texttt{deq}_A(q))$, da $I(\texttt{deq}_A(q)) \Leftrightarrow I(\texttt{norm}((x_2 \ldots x_n,\ y_1 \ldots y_m))) = \top$, was man leicht durch eine entsprechende Fallunterscheidung verifiziert.

Zusammenfassung der wichtigsten Aspekte

Der Begriff der Implementierung formalisiert folgende Aspekte von Datenabstraktion:

- Abstrakte und konkrete (implementierende) Algebra haben dieselbe Signatur.
- Konkrete Daten repräsentieren abstrakte Größen. Verschiedene konkrete Daten können denselben abstrakten Wert repräsentieren. Daher geht die Abstraktionsfunktion a von konkret nach abstrakt und sie braucht nicht injektiv zu sein. Die Gleichheit auf den konkreten Werten sagt also nicht über die Gleichheit der abstrakten Werte aus. Wird die konkrete Gleichheit, wie das in SML der Fall ist, dennoch exportiert, so geht die Datenabstraktion verloren.
- Nicht alle konkreten Daten repräsentieren vernünftige abstrakte Werte. Die Repräsentationsinvariante gibt an, welche konkreten Daten tatsächlich repräsentieren. Die Abstraktionsfunktion ist i.allg. nur partiell.
- Die konkreten Realisierungen der abstrakt geforderten Operationen brauchen nicht auf allen konkreten Daten definiert zu sein, sondern nur auf denjenigen, die auch etwas repräsentieren. Im Idealfall ist der Schutz der konkreten Daten vollständig, so daß man sie nur durch die exportierten Operationen erzeugen kann. Diese erfüllen dann aber die in der Repräsentationsinvariante geforderten Eigenschaften (vgl. Forderung 4 in der Definition von Implementierung).

3.12 Funktoren

3.12.1 Verwendung von Funktoren

Funktoren sind Funktionen die Strukturen als Argument nehmen und Strukturen liefern. Parameterstrukturen. Hauptanwendungen von Funktoren sind:

Generische Module: Betrachte einen Sortiermodul, der unabhängig von der <-Relation auf Listenelementen sein soll. Er kann verstanden werden als eine Abbildung von allen Strukturen der Signatur

```
signature ORDER =     sig    type T
                             val less: T * T → bool
                      end;
```

auf Strukturen der Form

```
structure SORT =
    struct      type L = ORDER.T list
                fun sort l = ··· if T.less(x, y) ···
    end;
```

Die Argumentstrukturen einer solchen Konstruktion können völlig unterschiedlicher Natur sein, solange `less` eine lineare Ordnung auf T definiert.

Spezifische Implementierung als Parameter: Der Modul `Array` stützte sich auf den `ArrayOps`-Modul ab. In der Funktionalität muß `ArrayOps` (bis auf Isomorphie) eindeutig bestimmt sein, damit der `Array`-Modul in der gewünschten Weise funktioniert. Dennoch ist der `Array`-Modul unabhängig von der konkreten Implementierung der `ArrayOps`. Stehen in einer Anwendungssituation meherere Implementierungen zur Wahl, so schreibt man `Array` besser als einen Funktor, der beliebige konkrete `ARRAYOPS`-Strukturen als Argument nimmt. Dann können beispielsweise von Fall zu Fall Komplexitätsargumente den Ausschlag für die Wahl einer konkreten Implementierung geben.

Separate Entwicklung von Systemteilen Strukturen S brauchen i.allg. andere Strukturen $S1$, auf denen sie aufbauen. Wie bei normalen Funktionen hat man die Wahl, $S1$ entweder direkt als freie (globale) Größe zu referenzieren, oder aber $S1$ als Parameter von S zu betrachten. Selbst wenn man niemals vorhat, für diesen Parameter etwas anderes als die Struktur $S1$ einzusetzten, so macht die Parametervariante doch großen Sinn. Dann kann nämlich S eigenständig programmiert, kompiliert und mit einem $S1$-Prototyp getestet werden. Die Unabhängigkeit in der Entwicklung von Systemteilen ist dann in praktikabler Weise gewährleistet.

Beachte: Die Anwendung 2) ist ein Spezialfall von Anwendung1) wo die Klasse der zulässigen Strukturen stärker eingeschränkt wird, z.B. auf die Isomorphieklassse einer Algebra.

3.12.2 Warum braucht es ein neues Sprachkonzept?

Polymorphismus als Parametrisierungskonzept für Typen ist nicht ausreichend für die Realisierung der oben beschriebenen Zwecke. Im Sortierbeispiel beschreibt der Typ

$$\texttt{sort: } \alpha\texttt{list} \rightarrow \alpha\texttt{list}$$

nicht, daß `sort` eine Ordnung auf α braucht. Technisch gesehen ist nicht klar, wie man auf das `less`-Prädikat, sofern es für ein konkretes α überhaupt existiert, Bezug nehmen kann. Eine Lösung dieses Problems verwendet Typklassen, d.h. Typangaben für Typen:

$$\texttt{sort} : \forall\alpha : \texttt{LinearOrder}.\ \alpha\ \texttt{list} \to \alpha\ \texttt{list}$$

Typklassen werden wir am Beispiel der Sprache Haskell noch kennenlernen.

In SML sind Typen sind keine Objekte erster Klasse. Funktionen können keine Typen als Argumente haben. Sonst könnte man Strukturen auch als abhängige Produkte (Verbunde) schreiben, z.B. als

$$\begin{array}{lllll}\texttt{val}\ \alpha\ \mathit{queue} & = \{ & \mathit{qty} & = & \alpha\ \texttt{list}; \\ & & \mathit{enq}(q : \mathit{qty},\ x : \alpha) & = & \cdots; \\ & & \cdots & & \\ & \} & & & \end{array}$$

und Funktoren wären einfach „normale“ Funktionen. Tatsächlich gibt es Sprachen auf der Grundlage stärkerer Typtheorien, die eine solche Formulierung von Funktionen zulassen, vgl. Burstall; Vorlesung Typtheorie.

3.13 Polymorphismus, Überladung und Typklassen

Als *Überladung* bezeichnet man die Situation, in der, im selben Gültigkeitsbereich, derselbe Name für verschiedene Funktionen gleichzeitig verwendet wird. Zwischen Überladung und Polymorphie sind die Grenzen fließend.

Überladung in SML

- 1, 2, ... sind Konstanten für ganze Zahlen beschränkter oder beliebiger Länge.
- $+$, $*$, ... sind für für alle Zahlentypen definiert.
- $=$ ist für alle „Gleichheitstypen“ (dies sind insbesondere keine funktionalen Typen) definiert.

In diesen Beispielen werden gleiche Namen für **verschiedene** Funktionen verwendet. Die Gleichheit auf Zahlen und die Gleichheit auf Listen sind völlig verschieden realisierte Funktionen. Überladene Symbole sind zumeist nicht auf allen Typen definiert.

Polymorphismus
Der Typ α in α `list` für `append` spielt keine Rolle, `append` ist immer diesselbe Funktion

$$\texttt{append}:\ ([x_1, \ldots, x_n], [y_1, \ldots, y_m]) \mapsto [x_1, \ldots, x_n, y_1, \ldots, y_m]$$

Der Übergang zwischen Überladung und Polymorphismus ist fließend. „Dieselbe“ Funktion könnte allgemeiner bedeuten, daß nur bestimmte Eigenschaften einer polymorphen Funktion für alle Typen gleich sind. Beispielsweise sollte „=“ eine Funktion sein, die für jeden Typ eine Kongruenzrelation darstellt. Unter diesem Verständnis könnte „=“ als eine polymorphe Funktion betrachtet werden. $+$ ist bei allen Typen kommutativ, d.h. es gilt $\forall x, y : T \quad x + y = y + x$, für jeden Typ T, auf dem $+$ definiert ist. Eine

Sortierfunktion arbeitet unabhängig vom Typ der Listenelemente. Daß sie die ihr üblicherweise zugedachte Eigenschaft erfüllt, hängt nur davon ab, daß des <-Prädikat auf den Listenelementen eine lineare Ordnung definiert.

Typklassen sind ein Ansatz, um dieses Spektrum zwischen der Extremen Überladung und Polymorphismus auszufüllen. Typklassen erlauben auch, die Unterscheidung zwischen Strukturen (Algebren) und Typen, die es in ML gibt, verschwinden zu lassen. Die Grundideen sind die folgenden:

1. Typen sind Strukturen (Algebren) einer gegebenen Signatur.

2. Strukturen sind klassifiziert oder typisiert. Der Typ eines Typs heißt eine *Typklasse*. Eine gröbste sinnvolle Klassifikation ist die Signatur der Algebra. Sie gibt an, welche Funktionen auf den Werten des Typs definiert sein sollen. Für ein mehr befriedigendes Konzept sollten auch die **Gesetze** bekannt sein, die die Funktionen in allen Typen der Klasse erfüllen sollen. In Haskell und in den meisten objektorientierten Programmiersprachen können Typklassen prototypische Implementierung von Operationen enthalten. Deren Eigenschaften kann man als die Gesetze verstehen, die jede Reimplementierung in Unterklassen erfüllen sollte. Typklassen sind also Mengen von Typen (Strukturen, Algebren), die Anforderungen einer irgendwie spezifizierten Art befriedigen.

3. Wenn Typklassen Mengen von Typen sind, so ist durch die Teilmengenbeziehung eine natürliche *Klassifikationshierarchie* gegeben.

4. Der Zusammenhang zu den Konzepten bei objektorientierten Prorgammiersprachen ist der, daß Objekte den Typen, Klassen den Typklassen und Objektzustände den Werten eines Typs entsprechen. Bei der reinen funktionalen Programmierung gibt es keine Möglichkeit, diesen Zustand zu ändern. Zwischen Wert/Zustand und Objektidentifikation gibt es dann keinen Unterschied.

In der Vorlesung besprechen wir an dieser Stelle das Konzept von Typklassen in Haskell nach dem ACM Sigplan Notes - Heft vom Mai 1992. Folgende Aspekte des Typsystems mit Typklassen sind besonders interessant:

- Es braucht in Haskell nur das Konzept von Typklassen mit ihren Typinstanzen, um die verschiedenen SML-Konzepte Typen, Typkonstruktoren, SIgnaturen, Strukturen und Funktoren zu realisieren.

- Default-Implementierungen von Funktionen in Typklassen können als eine Art Spezifikation aufgefaßt werden. Jede Reimplementierung in Unterklassen sollte sich möglichst an die Eigenschaften der Default-Implementierung halten.

- Das Problem der Verschiedenheit von konkreter Gleichheit und abstrakter Gleichheit bei konkreten bzw. abstrakten Datentypen, dann wenn mehrere konkrete Daten denselben abstrakten Wert repräsentieren, verschwindet, weil man in Haskell die Gleichheitsrelation in Unterklassen und Typinstanzen von `Eq` geeignet (re-)definieren kann.

4 Logische Programmierung

5 Anhang