Algorithmen und Datenstrukturen

Werner Struckmann

Wintersemester 2005/06

10. Funktionale und deduktive Algorithmen

Einführung

Grundidee:

- Ein Algorithmus wird durch eine Funktion f realisiert.
- Die Eingabe des Algorithmus ist ein Element w aus dem Definitionsbereich von f.
- Die Ausgabe des Algorithmus ist der Wert $f(w)$ aus dem Wertebereich von f.

Einführung

- Mathematische Funktionen sind häufig deklarativ definiert: Sie beinhalten keine Angaben zur Durchführung ihrer Berechnung.
- Beispiele: $f(n, m) = n \cdot m$, $f(n) = n!$.
- Wie kann ein Algorithmus den Funktionswert $f(w)$ berechnen?
- Können alle berechenbaren Probleme derart gelöst werden?

Partielle und totale Funktionen

Eine partielle Funktion

$$f : A \longrightarrow B$$

ordnet jedem Element x einer Teilmenge $D_f \subseteq A$ genau ein Element $f(x) \in B$ zu. Die Menge D_f heißt Definitionsbereich von f. f ist eine totale Funktion, wenn $D_f = A$ gilt.

Beispiel:

$$f : \mathbb{R} \longrightarrow \mathbb{R}, \quad D_f = \mathbb{R} \setminus \{0\}, \quad f(x) = \frac{1}{x}$$

Algorithmen können undefinierte Ausdrücke enthalten und müssen nicht in jedem Fall terminieren, d. h.:

Algorithmen berechnen partielle Funktionen!

Definition von Funktionen

- Wenn der Definitionsbereich einer Funktion endlich ist, lässt sie sich durch Angabe aller Funktionswerte in einer Tabelle definieren.
- Beispiel: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

Definition von Funktionen

- In den meisten Fällen wird eine Funktion $f : A \to B$ durch einen Ausdruck, der zu jedem Element aus D_f genau einen Wert von B liefert, beschrieben.
- Beispiel: $\max : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

$$\max(x, y) = \begin{cases} x & x \geq y \\ y & x < y \end{cases}$$

$$= \textit{if } x \geq y \textit{ then } x \textit{ else } y \textit{ fi}$$

Rekursive Definitionen (Wiederholung)

Die Funktion $f : \mathbb{N} \longrightarrow \mathbb{N}$ wird durch

$$f(n) = \begin{cases} 1 & n = 0, \\ 1 & n = 1, \\ f\left(\frac{n}{2}\right) & n \geq 2,\ n \text{ gerade}, \\ f\left(3n + 1\right) & n \geq 2,\ n \text{ ungerade}. \end{cases}$$

rekursiv definiert.

Auswertung von Funktionen (Wiederholung)

Funktionsdefinitionen können als Ersetzungssysteme gesehen werden. Funktionswerte lassen sich aus dieser Sicht durch wiederholtes Einsetzen berechnen. Die Auswertung von $f(3)$ ergibt

$$f(3) \to f(10) \to f(5) \to f(16) \to f(8) \to f(4) \to f(2) \to f(1) \to 1.$$

Terminiert der Einsetzungsprozess stets?

Formen der Rekursion (Wiederholung)

- Lineare Rekursion,
- Endrekursion,
- Verzweigende Rekursion (Baumrekursion),
- Geschachtelte Rekursion,
- Verschränkte Rekursion (wechselseitige Rekursion).

Funktionen höherer Ordnung

Funktionen können selbst Argumente oder Werte sein. In diesem Fall spricht man von Funktionen höherer Ordnung oder Funktionalen.

$$f : (A_1 \to A_2) \times A_3 \to B$$

$$g : A \to (B_1 \to B_2)$$

$$h : (A_1 \to A_2) \to (B_1 \to B_2)$$

Funktionen höherer Ordnung

Beispiele:

- Summe: $\sum_{i=a}^{b} f(i)$
- Komposition von Funktionen: $f \circ g$
- Auswahl zwischen Funktionen: *if p then f else g fi*
- Bestimmtes Integral: $\int_a^b f(x)\, dx$

Funktionale Algorithmen

- Ein Algorithmus heißt funktional, wenn die Berechnungsvorschrift mittels einer Sammlung (partieller) Funktionen definiert wird.
- Die Funktionsdefinitionen dürfen insbesondere Rekursionen und Funktionen höherer Ordnung enthalten.

Funktionale Algorithmen

Beispiel:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= nf(n-2) \end{aligned}$$

Wenn wir als Datenbereich die Menge der ganzen Zahlen zugrunde legen, berechnet dieser Algorithmus die Funktion $f : \mathbb{Z} \to \mathbb{Z}$ mit $D_f = \mathbb{N}$ und

$$f(n) = \begin{cases} 0 & n \text{ gerade} \\ \prod_{i=0}^{\frac{n-1}{2}} (2i+1) & n \text{ ungerade} \end{cases}$$

Funktionale Programmiersprachen

Programmiersprachen, die in erster Linie für die Formulierung funktionaler Algorithmen gedacht sind, heißen funktional.
Funktionale Programmiersprachen sind beispielsweise

- Lisp,
- Scheme,
- ML, SML und
- Haskell.

Man kann in vielen imperativen/objektorientierten Programmiersprachen funktional programmieren – und umgekehrt!

Lisp und Scheme

- Lisp wurde Ende der 50er Jahre von John McCarthy entwickelt.
- Im Laufe der Jahre wurden viele Lisp-Dialekte, u. a. Common Lisp und Scheme definiert.
- Die erste Version von Scheme stammt aus dem Jahre 1975. Autoren waren Guy Lewis Steele Jr. und Gerald Jay Sussman.
- Lisp und Scheme werden in der Regel interpretiert, nicht compiliert.

Algorithmus von Euklid

Funktional geschrieben hat der

Algorithmus von Euklid

die Form:

$$\begin{aligned} \text{ggT}(a, 0) &= a \\ \text{ggT}(a, b) &= \text{ggT}(b, a \bmod b) \end{aligned}$$

Beispiel: $\text{ggT}(36, 52) \rightarrow \text{ggT}(52, 36) \rightarrow \text{ggT}(36, 16) \rightarrow \text{ggT}(16, 4) \rightarrow \text{ggT}(4, 0) \rightarrow 4$

Scheme: Algorithmus von Euklid

Der funktionale

Algorithmus von Euklid

lautet beispielsweise als Scheme-Programm:

```
(define (ggT a b)
    (if (= b 0)
        a
        (ggT b (remainder a b))))

(ggT 36 52)
4
```

Terme

Terme sind aus

- Konstanten,
- Variablen,
- Funktions- und
- Relationssymbolen

zusammengesetzte Zeichenketten. Terme besitzen einen Typ.

Beispiele:

- Die Konstanten $\ldots, -2, -1, 0, 1, 2, \ldots$ sind int-Terme.
- $13 - \sqrt{2} + 3$ ist ein real-Term.
- $4 \cdot (3 - 2) + 3 \cdot i$ ist ein int-Term, falls i eine Variable vom Typ int ist.
- Ist b ein bool-Term und sind t, u int-Terme, so ist auch if b then t else u fi ein int-Term.

Funktionsdefinitionen

Die Gleichung

$$f(x_1, \ldots, x_n) = t(x_1, \ldots, x_n)$$

heißt Definition der Funktion f vom Typ τ, wenn gilt:

- f ist der Funktionsname.
- $x_1, \ldots, x_n$ sind Variable, die formale Parameter genannt werden. Die Typen von $x_1, \ldots, x_n$ seien $\tau_1, \ldots, \tau_n$.
- t ist ein Term, der die Variablen $x_1, \ldots, x_n$ enthält. Der Typ von t sei τ.
- $f : \tau_1 \times \cdots \times \tau_n \to \tau$ heißt Signatur von f.

Der Fall $n = 0$ ist zugelassen. In diesem Fall liefert die Auswertung stets das gleiche Ergebnis. Die Funktion entspricht somit einer Konstanten.

Funktionsdefinitionen

Beispiele:

$\mathrm{ZylVol}(h, r)$	$= h \cdot \pi \cdot r^2$
	(Signatur ZylVol: real×real→real)
π	$= 3.1415926535\ldots$
	(Signatur: $\pi \rightarrow$real)
$\max(x, y)$	$=$ `if` $x > y$ `then` x `else` y `fi`
	(Signatur max: int×int→int)
$f(p, q, x, y)$	$=$ `if` $p \vee q$ `then` $2 \cdot x + 1$ `else` $3 \cdot y - 1$ `if`
	(Signatur f: bool×bool×int×int→int)
$g(x)$	$=$ `if` $\mathit{even}(x)$ `then` $\frac{x}{2}$ `else` $3 \cdot x + 1$ `fi`
	(Signatur g: int→int)
$h(p, q)$	$=$ `if` p `then` q `else` false `fi`
	(Signatur h: bool×bool→bool)

Funktionsanwendungen

- Unter einer Anwendung (Applikation) einer Funktion $f(x_1, \ldots, x_n) = t(x_1, \ldots, x_n)$ versteht man einen Ausdruck

$$f(a_1, \ldots, a_n).$$

- Für die formalen Parameter werden Ausdrücke (aktuelle Parameter) eingesetzt, die den richtigen Typ besitzen müssen.
- Die Auswertung liefert eine Folge

$$f(a_1, \ldots, a_n) \rightarrow t(a_1, \ldots, a_n) \rightarrow \cdots .$$

- Es muss genau festgelegt werden, wie und in welcher Reihenfolge auftretende (Teil-)Ausdrücke ausgewertet werden.
- Diese Folge muss nicht terminieren.

Funktionsanwendungen

$$
\begin{aligned}
f(p,q,x,y) &= \texttt{if } p \wedge q \texttt{ then } 2 \cdot x + 1 \texttt{ else } 3 \cdot y - 1 \texttt{ fi} \\
f(\mathit{false},\mathit{false},3,4) &= \texttt{if } \mathit{false} \wedge \mathit{false} \texttt{ then } 2 \cdot x + 1 \\
&\qquad \texttt{else } 3 \cdot y - 1 \texttt{ fi} \\
&= \texttt{if } \mathit{false} \texttt{ then } 2 \cdot 3 + 1 \texttt{ else } 3 \cdot 4 - 1 \texttt{ fi} \\
&= 3 \cdot 4 - 1 = 11
\end{aligned}
$$

$$
\begin{aligned}
g(x) &= \texttt{if } \mathit{even}(x) \texttt{ then } \frac{x}{2} \texttt{ else } 3 \cdot x + 1 \texttt{ fi} \\
g(1) &= \texttt{if } \mathit{even}(1) \texttt{ then } \frac{1}{2} \texttt{ else } 3 \cdot 1 + 1 \texttt{ fi} \\
&= 3 \cdot 1 + 1 = 4
\end{aligned}
$$

Funktionsanwendungen

$$h(p, q) = \texttt{if } p \texttt{ then } 8 \texttt{ else } 8/q \texttt{ fi}$$

$$\begin{aligned}
h(\mathit{true}, 2) &= \texttt{if } \mathit{true} \texttt{ then } 8 \texttt{ else } 8/2 \texttt{ fi} && = 8 \\
h(\mathit{false}, 2) &= \texttt{if } \mathit{false} \texttt{ then } 8 \texttt{ else } 8/2 \texttt{ fi} && = 4 \\
h(\mathit{false}, 0) &= \texttt{if } \mathit{false} \texttt{ then } 8 \texttt{ else } 8/0 \texttt{ fi} && \mathit{undefiniert} \\
h(\mathit{true}, 0) &= \texttt{if } \mathit{true} \texttt{ then } 8 \texttt{ else } 8/0 \texttt{ fi} && = 8
\end{aligned}$$

Bei der Auswertung des Terms `if` b `then` t_1 `else` t_2 `fi` wird zunächst der boolesche Term b ausgewertet, und dann, abhängig vom Ergebnis, t_1 oder t_2.

Funktionsdefinitionen

Eine Funktion kann auch in mehreren Gleichungen definiert werden, jeweils für einen Teil der Argumente.

Beispiel:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 2 \\
f(-1) &= 2 \\
f(x) &= \texttt{ if } x > 1 \texttt{ then } x(x-1) \texttt{ else } -x(x-1) \texttt{ fi}
\end{aligned}
$$

Die Auswertung erfolgt dabei von oben nach unten, wobei die erste passende linke Seite für die Berechnung angewendet wird. Es kommt daher auf die Reihenfolge der Gleichungen an.

Funktionsdefinitionen

Folgendes Gleichungssystem definiert eine andere Funktion.

$$
\begin{aligned}
f(-1) &= 2 \\
f(x) &= \texttt{ if } x > 1 \texttt{ then } x(x-1) \texttt{ else } -x(x-1) \texttt{ fi} \\
f(1) &= 2 \\
f(0) &= 0
\end{aligned}
$$

Hier sind die letzten beiden Gleichungen überflüssig.

Man kann mehrere Definitionsgleichungen immer in einer Gleichung zusammenfassen, indem man geschachtelte if-then-else-fi Konstrukte verwendet.

Funktionsdefinitionen

Das erste Beispiel oben lässt sich in einer Gleichung schreiben:

$$
\begin{aligned}
f(x) = \;& \texttt{if } x = 0 \texttt{ then } 0 \\
& \texttt{else if } x = 1 \texttt{ then } 2 \\
& \texttt{else if } x = -1 \texttt{ then } 2 \\
& \texttt{else if } x > 1 \texttt{ then } x(x-1) \\
& \texttt{else } -x(x-1) \\
& \texttt{fi fi fi fi}
\end{aligned}
$$

Die Schreibweise mit mehreren Gleichungen ist in der Regel übersichtlicher.

Funktionsdefinitionen

Ein Wächter (guard) ist eine Vorbedingung, die für die Anwendung einer Definitionsgleichung erfüllt sein muss.

Beispiel:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 2 \\
f(-1) &= 2 \\
x > 1 : f(x) &= x(x-1) \\
f(x) &= -x(x-1)
\end{aligned}
$$

Funktionsdefinitionen

Eine Funktionsdefinition kann unvollständig sein.

Beispiel:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 2 \\ f(-1) &= 2 \\ x > 1 : f(x) &= x(x-1) \end{aligned}$$

Gegenüber dem vorigen Beispiel fehlt hier die letzte Gleichung. Es gibt daher keine Berechnungsvorschrift für Werte < -1. D. h., die Funktion ist dort nicht definiert.

Funktionsdefinitionen

Funktionen können unter Verwendung von Hilfsfunktionen definiert werden.

Beispiel:

$$\begin{aligned}
\text{Volumen}(h, r, a, b, c) &= \text{ZylVol}(h, r) + \text{QuadVol}(a, b, c) \\
\text{ZylVol}(h, r) &= h \cdot \text{KreisFl}(r) \\
\text{KreisFl}(r) &= \pi r^2 \\
\text{QuadVol}(a, b, c) &= a \cdot b \cdot c
\end{aligned}$$

Einsetzen führt zu einer einzeiligen Definition:

$$\mathit{Volumen}(h, r, a, b, c) = h\pi r^2 + a \cdot b \cdot c$$

Auswertung von Funktionen

$$\begin{aligned}
\text{Volumen}(3,2,5,1,5) &= \text{ZylVol}(3,2) + \text{QuadVol}(5,1,5) \\
&= 3 \cdot \text{KreisFl}(2) + \text{QuadVol}(5,1,5) \\
&= 3\pi 2^2 + \text{QuadVol}(5,1,5) \\
&= 3\pi 2^2 + 5 \cdot 1 \cdot 5 \\
&= 12\pi + 25 \\
&\approx 62.699111843
\end{aligned}$$

Alternativ kann einem Term ein Name gegeben werden, der dann (mehrfach) verwendet werden kann:

$$f(a,b) = x \cdot x \text{ where } x = a + b$$

ist gleichbedeutend mit

$$f(a,b) = (a+b) \cdot (a+b).$$

Applikative Algorithmen

Ein applikativer (funktionaler) Algorithmus ist eine Liste von Funktionsdefinitionen

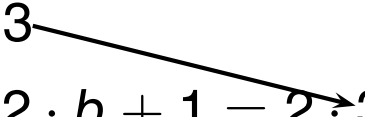
$$
\begin{aligned}
f_1(x_{1,1},\ldots,x_{1,n_1}) &= t_1(x_{1,1},\ldots,x_{1,n_1}),\\
f_2(x_{2,1},\ldots,x_{2,n_2}) &= t_2(x_{2,1},\ldots,x_{2,n_2}),\\
&\vdots = \vdots\\
f_m(x_{m,1},\ldots,x_{m,n_m}) &= t_m(x_{m,1},\ldots,x_{m,n_m}).
\end{aligned}
$$

Die erste Funktion ist die Bedeutung (Semantik) des Algorithmus. Die Funktion wird für eine Eingabe $(a_1,\ldots,a_{n_1})$ wie beschrieben ausgewertet. Die Ausgabe ist $f_1(a_1,\ldots,a_{n_1})$.

Gültigkeit und Sichtbarkeit

Beispiel:

$$f(a,b) = g(b) + a$$
$$g(a) = a \cdot b$$
$$b = 3$$
$$f(1,2) = g(2) + 1 = 2 \cdot b + 1 = 2 \cdot 3 + 1 = 7$$



Die globale Definition von b wird in der Definition von f durch die lokale Definition verdeckt. Es treten also auch hier die Fragen nach dem Gültigkeits- und dem Sichtbarkeitsbereich von Variablen auf, wie wir sie in Kapitel 2 bei den imperativen Algorithmen angetroffen haben.

Undefinierte Funktionswerte

Die Fakultätsfunktion ist definiert durch:

$$\begin{aligned} \mathrm{Fac}(0) &= 1 \\ \mathrm{Fac}(n) &= n \cdot \mathrm{Fac}(n-1) \end{aligned}$$

Für negative Parameter terminiert die Berechnung nicht:

$$\mathrm{Fac}(-1) = -1 \cdot \mathrm{Fac}(-2) = -1 \cdot -2 \cdot \mathrm{Fac}(-3) = \cdots$$

Die Funktion *Fac* ist also partiell. Es gibt drei mögliche Ursachen für undefinierte Funktionswerte:

- Die Parameter führen zu einer nicht terminierenden Berechnung.
- Eine aufgerufene Funktion ist für einen Parameter undefiniert (zum Beispiel Division durch 0).
- Die Funktion ist unvollständig definiert.

Komplexe Datentyen

Komplexe Datentypen (Datenstrukturen) entstehen durch Kombination elementarer Datentypen und besitzen spezifische Operationen. Sie können vorgegeben oder selbstdefiniert sein.

Die grundlegenden Datentypen werden auch Atome genannt. Übliche Atome sind die Typen int, bool, float und char sowie Variationen davon.

Es gibt in Bezug auf das Anwendungsspektrum eine Unterscheidung in

- generische Datentypen: Sie werden für eine große Gruppe ähnlicher Problemstellungen entworfen und sind oft im Sprachumfang enthalten (Liste, Keller, Feld, Verzeichnis, . . .).
- spezifische Datentypen: Sie dienen der Lösung einer eng umschriebenen Problemstellung und werden im Zusammenhang mit einem konkreten Problem definiert (Adresse, Person, Krankenschein, . . .).

Generische Datentypen der funktionalen Programmierung

In der funktionalen Programmierung spielen die folgenden generischen Datentypen eine hervorgehobene Rolle:

- Listen,
- Texte (Liste von Zeichen),
- Tupel,
- Funktionen.

Listen

Die Datenstruktur funktionaler Sprachen und Programmierung. Die funktionale Programmierung wurde anfangs auch Listenverarbeitung genannt. Lisp steht für „List Processor“.

Beispiele:

- Liste von Messwerten, geordnet nach Aufzeichnungszeitpunkt, z. B. Zimmertemperatur (° C) im Informatikzentrum nach Ankunft: [17.0, 17.0, 17.1, 17.2, 17.4, 17.8].
- Alphabetisch geordnete Liste von Namen z. B. Teilnehmer der kleinen Übung: [„Baltus“, „Bergmann“, „Cäsar“].
- Alphabetisch geordnete Liste von Namen mit Vornamen, d. h. Liste von zweielementigen Listen mit Namen und Vornamen: [[„Kundera“, „M.“], [„Hesse“, „S.“], [„Einstein“, „A.“]].

Listen

Syntax und Semantik:

- $[D]$ ist der Datentyp der Listen, d. h. der endlichen Folgen, über D.
- Notation: $[x_1, x_2, \ldots, x_n] \in [D]$ für $x_1, x_2, \ldots, x_n \in D$.

Beispiele:

- **[real]**: Menge aller Listen von Fließkommazahlen, z. B. Messwerte, Vektoren über $\mathbb{R}$.
- **[char]**: Menge aller Listen von Buchstaben, z. B. Namen, Befunde, Adressen.
- **[[char]]**: Menge aller Listen von Listen von Buchstaben, z. B. Namensliste.

Typische Listenoperationen

- **[]**: leere Liste.
- **e: l**: Verlängerung einer Liste l nach vorn um ein Einzelelement e, z. B. $1 : [2, 3] = [1, 2, 3]$.
- **length(l)**: Länge einer Liste l, z. B. $\mathit{length}([4, 5, 6]) = 3$.
- **head(l)**: erstes Element e einer nichtleeren Liste $l = e : l'$, z. B. $\mathit{head}([1, 2, 3]) = 1$.
- **tail(l)**: Restliste l' einer nichtleeren Liste $l = e : l'$ nach Entfernen des ersten Elementes, z. B. $\mathit{tail}([1, 2, 3]) = [2, 3]$.
- **last(l)**: letztes Element einer nichtleeren Liste, z. B. $\mathit{last}([1, 2, 3]) = 3$.

Typische Listenoperationen

- **init(l)**: Restliste einer nichtleeren Liste nach Entfernen des letzten Elements, z. B. $\mathit{init}([1,2,3]) = [1,2]$.
- **l++l'**: Verkettung zweier Listen l und l', z. B. $[1,2] + +[3,4] = [1,2,3,4]$.
- **l!!n**: Das n-te Element der Liste l, wobei $1 \leq n \leq \mathit{length}(l)$, z. B. $[2,3,4,5]!!3 = 4$.

Vergleichsoperationen $=$ und $\neq$:

$$(e_1 : t_1) = (e_2 : t_2) \Leftrightarrow e_1 = e_2 \wedge t_1 = t_2,$$
$$l_1 \neq l_2 \Leftrightarrow \neg(l_1 = l_2).$$

$$[i,\ldots,j] = \begin{cases} [i, i+1, i+2, \ldots, j-1, j] & \text{falls } i \leq j, \\ [] & \text{falls } i > j. \end{cases}$$

Typische Listenoperationen

Die folgende Funktion berechnet rekursiv das Spiegelbild einer Liste.

$$
\begin{aligned}
mirror &: [int] \rightarrow [int] \\
mirror([]) &= [] \\
mirror(l) &= last(l) : mirror(init(l))
\end{aligned}
$$

$$
\begin{aligned}
mirror([1,2,3,4]) &= 4 : mirror(init([1,2,3,4])) \\
&= 4 : mirror([1,2,3]) = 4 : (3 : mirror([1,2])) \\
&= 4 : (3 : (2 : mirror([1]))) \\
&= 4 : (3 : (2 : (1 : mirror([])))) \\
&= 4 : (3 : (2 : (1 : []))) \\
&= 4 : (3 : (2 : [1])) = 4 : (3 : [2,1]) \\
&= 4 : [3,2,1] = [4,3,2,1]
\end{aligned}
$$

Typische Listenoperationen

Die folgende Funktion berechnet rekursiv das Produkt der Elemente einer Liste.

$$
\begin{aligned}
prod &: [int] \rightarrow int \\
prod([]) &= 1 \\
prod(l) &= head(l) \cdot prod(tail(l))
\end{aligned}
$$

Die folgende Funktion konkateniert rekursiv eine Liste von Listen.

$$
\begin{aligned}
concat &: [[t]] \rightarrow [t] \\
concat([]) &= [] \\
concat(l) &= head(l) + +concat(tail(l)) \\
concat([[1,2],[],[3],[4,5,6]]) &= [1,2,3,4,5,6]
\end{aligned}
$$

Sortierverfahren

Alle Algorithmen aus den vorhergehenden Kapiteln lassen sich auch funktional beschreiben, häufig sehr viel eleganter. Als Beispiel betrachten wir zwei Sortierverfahren.

Wiederholung: Es sei eine Ordungsrelation $\leq$ auf dem Elementdatentyp D gegeben.

- Eine Liste $l = (x_1, \ldots, x_n) \in [D]$ heißt sortiert, wenn $x_1 \leq x_2 \leq \cdots \leq x_n$ gilt.
- Eine Liste $l' = [D]$ heißt Sortierung von $l \in [D]$, wenn l und l' die gleichen Elemente haben und l' sortiert ist.
- Eine Sortierung l' von l heißt stabil, wenn sie gleiche Listenelemente nicht in ihrer Reihenfolge vertauscht. $l = [5, 9, 3, 8, \underline{8}]$, $l' = [3, 5, 8, \underline{8}, 9]$ (nicht stabil wäre $l'' = [3, 5, \underline{8}, 8, 9]$)
- Ein Sortieralgorithmus heißt stabil, wenn er stabile Sortierungen liefert

Sortieren durch Einfügen

Beim Sortieren durch Einfügen wird die Ordnung hergestellt, indem jedes Element an der korrekten Position im Feld eingefügt wird.

$$
\begin{aligned}
& insert(x, []) && = [x] \\
x \leq y : & insert(x, y : l) && = x : y : l \\
& insert(x, y : l) && = y : insert(x, l)
\end{aligned}
$$

Für das Sortieren einer unsortierten Liste gibt es zwei Varianten:

$$
\begin{aligned}
sort1([]) &= [] \\
sort1(l) &= insert(head(l), sort1(tail(l))) \\
sort2([]) &= [] \\
sort2(l) &= insert(last(l), sort2(init(l)))
\end{aligned}
$$

Welche dieser Algorithmen sind stabil?

Sortieren durch Auswählen

Beim Sortieren durch Auswählen wird das kleinste (größte) Element an den Anfang (das Ende) der sortierten Liste angefügt. Die folgende Funktion löscht ein Element aus einer Liste:

$$\begin{aligned} & delete(x, []) && = [] \\ x = y : & delete(x, y : l) && = l \\ & delete(x, y : l) && = y : delete(x, l) \end{aligned}$$

Für das Sortieren einer unsortierten Liste gibt es wieder zwei Varianten:

$$sort3(l) = x : sort3(delete(x, l)) \qquad \text{where } x = min(l)$$
$$sort4(l) = sort4(delete(x, l)) + +[x] \text{ where } x = max(l)$$

Wie lauten *min* und *max*? Was lässt sich über die Stabilität dieser beiden Algorithmen aussagen?

Extensionale und intensionale Beschreibungen

Bisher wurden Listen durch Aufzählung oder Konstruktion beschrieben. Man spricht von einer extensionalen Beschreibung.

Mengen werden implizit durch einen Ausdruck der Form $\{t(x) \mid p(x)\}$ angegeben.

Beispiel: $\{x^2 \mid x \in \mathbb{N} \wedge x \bmod 2 = 0\} = \{4, 16, \ldots\}$

Analog hierzu bedeutet

$$[t(x) \mid x \leftarrow l, p(x)]$$

die Liste aller Werte $t(x)$, die man erhält, wenn x die Liste l durchläuft, wobei nur die Elemente aus l ausgewählt werden, die der Bedingung $p(x)$ genügen.

$[t(x) \mid x \leftarrow l, p(x)]$ ist eine intensionale Definition. $t(x)$ ist ein Term. $x \leftarrow l$ heißt Generator und $p(x)$ ist eine Auswahlbedingung.

Intensionale Beschreibungen

$$[x \mid x \leftarrow [1 \ldots 5]] = [1, 2, 3, 4, 5]$$
$$[x^2 \mid x \leftarrow [1 \ldots 5]] = [1, 4, 9, 16, 25]$$
$$[x^2 \mid x \leftarrow [1 \ldots 5], odd(x)] = [1, 9, 25]$$

Eine intensionale Beschreibung kann auch mehrere Generatoren enthalten:

$$[x^2 - y \mid x \leftarrow [1 \ldots 3], y \leftarrow [1 \ldots 3]] = [0, -1, -2, 3, 2, 1, 8, 7, 6]$$
$$[x^2 - y \mid x \leftarrow [1 \ldots 3], y \leftarrow [1 \ldots x]] = [0, 3, 2, 8, 7, 6]$$
$$[x^2 - y \mid x \leftarrow [1 \ldots 4], odd(x), y \leftarrow [1 \ldots x]] = [0, 8, 7, 6]$$
$$[x^2 - y \mid x \leftarrow [1 \ldots 4], y \leftarrow [1 \ldots x], odd(x)] = [0, 8, 7, 6]$$

Man vergleiche die Effizienz der beiden letzten Beschreibungen.

Intensionale Beschreibungen

$$
\begin{aligned}
teiler(n) &= [i \mid i \leftarrow [1 \ldots n], n \bmod i = 0] \\
teiler(18) &= [1, 2, 3, 6, 9, 18] \\
ggT(a, b) &= max([d \mid d \leftarrow teiler(a), b \bmod d = 0]) \\
ggT(18, 8) &= max([d \mid d \leftarrow [1, 2, 3, 6, 9, 18], 8 \bmod d = 0]) \\
&= max([1, 2]) = 2 \\
primzahl(n) &= (teiler(n) = [1, n]) \\
primzahl(17) &= (teiler(17) = [1, 17]) = true \\
concat(l) &= [x \mid l' \leftarrow l, x \leftarrow l'] \\
concat([[1, 2, 3], [4, 5, 6]]) &= [1, 2, 3, 4, 5, 6]
\end{aligned}
$$

Tupel

Tupel sind Listen fester Länge.

Beispiele:

- $(1.0, -3.2)$ als Darstellung für die komplexe Zahl $1 - 3.2i$.
- $(4, 27)$ als Abbildung eines Messzeitpunkts $(4\,ms)$ auf einen Messwert $(27\,V)$.
- $(2, 3.4, 5)$ als Darstellung eines Vektors im $\mathbb{R}^3$.

Der Typ t eines Tupels ist das kartesische Produkt der seiner Elementtypen: $t = t_1 \times t_2 \times \ldots \times t_n$

Schreibweise für Elemente des Typs $t : (x_1, x_2, \ldots, x_n)$ Man nennt $(x_1, x_2, \ldots, x_n)$ ein n-Tupel.

Tupel sind grundlegende Typen aller funktionalen Sprachen.

Tupel

Auf der Basis von Tupeln lassen sich spezifische Datentypen definieren:

- **date**: $int \times text \times int$. Datumsangaben mit Werten wie (2, „Mai“, 2001). Es dürfen nur gültige Werte aufgenommen werden.
- **rat**: $int \times int$. Rationale Zahlen mit Werten wie (2,3) für $\frac{2}{3}$. Das 2-Tupel (Paar) $(1, 0)$ stellt keinen gültigen Wert dar.

Beispiele für Funktionen auf rat:

$$ratAdd, ratMult : rat \times rat \to rat$$
$$kuerze : rat \to rat$$

$$kuerze(z, n) = (z \text{ div } g, n \text{ div } g) \text{ where } g = ggT(z, n)$$
$$ratAdd((z_1, n_1), (z_2, n_2)) = kuerze(z_1 n_2 + z_2 n_1, n_1 n_2)$$
$$ratMult((z_1, n_1), (z_2, n_2)) = kuerze(z_1 z_2, n_1 n_2)$$

Funktionen höherer Ordnung

- Funktionen als Datentypen machen es möglich, Funktionen auf Funktionen anzuwenden.
- Eine Funktion $f : A \to B$ ist vom Typ $A \to B$.
- Die Operation $\to$ sei rechtsassoziativ, d. h.

$$A \to B \to C = A \to (B \to C)$$

Currying

- Das Currying vermeidet kartesische Produkte: Eine Abbildung

$$f : A \times B \to C$$

kann als eine Abbildung

$$f : A \to (B \to C) = A \to B \to C$$

gesehen werden.

- Beispiel: $f : int \times int$ mit $f(x, y) = x + y$ entspricht $f_g : int \to int \to int$ mit $f(x) = g_x : int \to int$ und $g_x(y) = x + y$. Hintereinanderausführung: $(f_g(x))(y) = g_x(y) = x + y = f(x, y)$

Funktionen höherer Ordnung

Funktionen können als Werte und Argumente auftreten.

Beispiel: Ein Filter, der aus einer Liste diejenigen Elemente auswählt, die einer booleschen Bedingung genügen.

Spezifikation:

$$filter(p, l) = [x \mid x \leftarrow l, p(x)]$$

Definition:

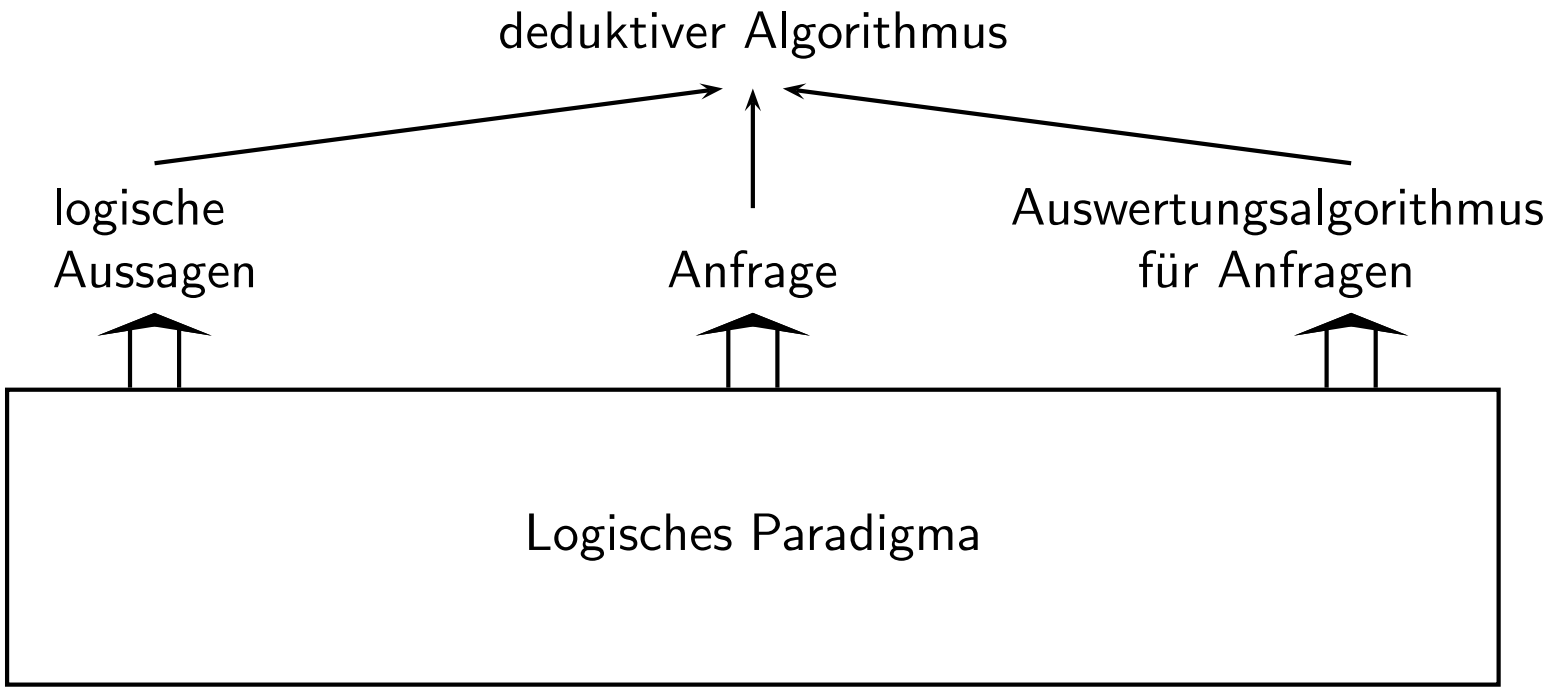
$$\begin{aligned} & filter && : (t \to bool) \times [t] \to [t] \\ & filter(p, []) && = [] \\ p(x) : & filter(p, x : l) && = x : filter(p, l) \\ & filter(p, x : l) && = filter(p, l) \end{aligned}$$

Funktionen höherer Ordnung

Fortsetzung zum Filter, Anwendung:

$$
\begin{aligned}
& p : int \rightarrow bool \\
even(i) : & p(i) && = true \\
& p(i) && = false \\
& filter(p, [1 \ldots 5]) && = [2, 4]
\end{aligned}
$$

Deduktive Algorithmen



Die wichtigste logische Programmiersprache ist Prolog.

Prädikatenlogik

- Grundlage des logischen Paradigmas ist die Prädikatenlogik.
- Beispiel einer Aussage: „Susanne ist Tochter von Petra“.
- Eine Aussageform ist eine Aussage mit Unbestimmten: x ist Tochter von y.
- Durch eine Belegung der Unbestimmten kann eine Aussageform in eine Aussage transformiert werden: $x \leftarrow \mathit{Susanne}, y \leftarrow \mathit{Petra}$.
- Statt natürlichsprachiger Aussagen und Aussageformen, werden in deduktiven Algorithmen atomare Formeln verwendet: $\mathit{Tochter}(x, y)$.

Prädikatenlogik

Alphabet:

- Konstante: $a, b, c, \ldots$.
- Unbestimmte/Variable: $x, y, z, \ldots$.
- Prädikatssymbole: $P, Q, R, \ldots$ mit Stelligkeit.
- Logische Verknüpfungen: $\wedge, \Rightarrow, \ldots$.

Atomare Formeln: $P(t_1, \ldots, t_n)$.

Fakten: Atomare Formeln ohne Unbestimmte.

Regeln haben die Form (α_i ist atomare Formel):

$$\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n \Rightarrow \alpha_0$$

$\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$ wird als Prämisse, α_0 als Konklusion der Regel bezeichnet.

Beispiel

Zwei Fakten:

- $\mathit{Tochter}(\mathit{Susanne}, \mathit{Petra})$
- $\mathit{Tochter}(\mathit{Petra}, \mathit{Rita})$

Eine Regel mit Unbestimmten:

- $\mathit{Tochter}(x, y) \wedge \mathit{Tochter}(y, z) \Rightarrow \mathit{Enkelin}(x, z)$

Die Ableitung neuer Fakten erfolgt analog zur Implikation:

1. Finde eine Belegung der Unbestimmten einer Regel, so dass auf der linken Seite (Prämisse) bekannte Fakten stehen.
2. Die rechte Seite ergibt den neuen Fakt.

Beispiel

Belegung der Unbestimmten der Regel:
$x \leftarrow \mathit{Susanne}, y \leftarrow \mathit{Petra}, z \leftarrow \mathit{Rita}$

Anwendung der Regel ergibt neuen Fakt: $\mathit{Enkelin}(\mathit{Susanne}, \mathit{Rita})$

(Erste) Idee deduktiver Algorithmen:

1. Definition einer Menge von Fakten und Regeln sowie einer Anfrage in Form einer zu prüfenden Aussage; z. B. $\mathit{Enkelin}(\mathit{Susanne}, \mathit{Rita})$.
2. Prüfen und Anwenden der Regeln, bis keine neuen Fakten mehr erzeugt werden können.
3. Prüfen, ob Anfrage in Faktenmenge enthalten ist.

Deduktive Algorithmen

- Ein deduktiver Algorithmus D besteht aus einer Menge von Fakten und Regeln.
- Aus einem deduktiven Algorithmus sind neue Fakten ableitbar. Die Menge aller Fakten $F(D)$ enthält alle direkt oder indirekt aus D ableitbaren Fakten.
- Ein deduktiver Algorithmus definiert keine Ausgabefunktion wie applikative oder imperative Algorithmen. Erst die Beantwortung von Anfragen liefert ein Ergebnis.
- Eine Anfrage γ ist eine Konjunktion von atomaren Formeln mit Unbestimmten: $\gamma = \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$

Beispiel: Addition zweier Zahlen

Fakten:

- $suc(n, n+1)$ für alle $n \in \mathbb{N}$

Regeln:

1. $true \Rightarrow add(x, 0, x)$
2. $add(x, y, z) \wedge suc(y, v) \wedge suc(z, w) \Rightarrow add(x, v, w)$

Anfrage: $add(3, 2, 5)$ liefert true.

Auswertung:

- Regel 1 mit der Belegung $x = 3$: $add(3, 0, 3)$
- Regel 2 mit der Belegung $x = 3, y = 0, z = 3, v = 1, w = 4$: $add(3, 1, 4)$
- Regel 2 mit der Belegung $x = 3, y = 1, z = 4, v = 2, w = 5$: $add(3, 2, 5)$

Beispiel: Addition zweier Zahlen

- $add(3, 2, x)$ liefert $x = 5$.
- $add(3, x, 5)$ liefert $x = 2$.
- $add(x, y, 5)$ liefert
  $(x, y) \in \{(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)\}$.
- $add(x, y, z)$ liefert eine unendliche Ergebnismenge.
- $add(x, x, 4)$ liefert $x = 2$.
- $add(x, x, x)$ liefert $x = 0$.
- $add(x, x, z) \wedge add(x, z, 90)$ liefert $(x, z) = (30, 60)$.

Deduktive Algorithmen sind deklarativ (s. oben). Im Vergleich zu applikativen und imperativen Algorithmen sind sie sehr flexibel – und häufig ineffizient.

Auswertungsalgorithmus

Dieser informelle nichtdeterministische Algorithmus wertet Anfragen aus:

1. Starte mit der Anfrage γ (anzusehen als Menge atomarer Formeln).
2. Suche Belegungen, die entweder
   - einen Teil von γ mit Fakten gleichsetzen (Belegung von Unbestimmten von γ) oder
   - einen Fakt aus γ mit einer rechten Seite einer Regel gleichsetzen (Belegungen von Unbestimmten in einer Regel).

   Setze diese Belegung ein.
3. Wende passende Regeln rückwärts an, ersetze also in γ die Konklusion durch die Prämisse.
4. Entferne gefundene Fakten aus der Menge γ.
5. Wiederhole diese Schritte bis γ leer ist.

Beispiel: Addition zweier Zahlen

1. $add(3, 2, 5)$.
2. $add(3, y', z'), suc(y', 2), suc(z', 5)$.
3. $y' = 1$, dadurch Fakt $suc(1, 2)$ streichen.
4. $add(3, 1, z'), suc(z', 5)$.
5. $z' = 4$, dadurch Fakt $suc(4, 5)$ streichen.
6. $add(3, 1, 4)$.
7. $add(3, y'', z''), suc(y'', 1), suc(z'', 4)$.
8. $y'' = 0, z'' = 3$ beide Fakten streichen.
9. $add(3, 0, 3)$ streichen, damit zu bearbeitende Menge leer, also
10. *true*.

Beispiel: Addition zweier Zahlen

1. $add(3, 2, x)$.
2. $add(3, y', z'), suc(y', 2), suc(z', x)$.
3. $y' = 1$, dadurch Fakt $suc(1, 2)$, streichen.
4. $add(3, 1, z'), suc(z', x)$.
5. $add(3, y'', z''), suc(y'', 1), suc(z'', z'), suc(z', x)$.
6. $y'' = 0$, dadurch Fakt $suc(0, 1)$, streichen.
7. $add(3, 0, z''), suc(z'', z'), suc(z', x)$.
8. $z'' = 3$, dadurch Regel 2 erfüllt, streichen.
9. $suc(3, z'), suc(z', x)$.
10. $z' = 4$, dadurch Fakt $suc(3, 4)$ streichen.
11. $suc(4, x)$.
12. $x = 5$, die zu bearbeitende Menge ist leer und eine Lösung für x bestimmt.

Deduktive Algorithmen

- Für eine Anfrage können unendlich viele Lösungen existieren: $add(x, y, z)$.
- Die Bestimmung aller möglichen Berechnungspfade kann durch Backtracking erfolgen.
- Das angegebene Verfahren ist sehr vereinfacht:
  - Wie wird verhindert, dass ein unendlich langer Weg eingeschlagen wird?
  - Was ist mit Negationen?