

7. Funktionen (Fortsetzung)

7.1 Mehrere Argumente, mehrere Ergebnisse

Bisher: Funktionen hatten ein Argument

Jetzt: Funktionen haben ein Argument, aber Argument kann vom „Tupeltyp“ oder „Verbundtyp“ sein. Mehrere Argumente werden in ein Tupel gepackt.

Beispiel:

Funktion dist zur Berechnung des euklidischen Abstands eines Punktes (x,y) vom Ursprung:

fun dist (x:real, y:real) : real = sqrt (x*x + y*y)

Mehrere Ergebnisse können ebenso in ein Tupel oder einen Verbund gepackt werden.

Neue Funktion: dist' soll auch ihre Argumente ausgeben:

fun dist' (x:real, y:real) : real * real * real = (x, y, sqrt (x*x + y*y))

7.2 Definition über Fälle

Argumentbereich einer Funktion aufteilen in mehrere Teilbereiche mit verschiedenen definierenden Ausdrücken.

$$\text{Mathe: } f(x) = \begin{cases} \dots\dots \text{falls } x > 0 \\ \dots\dots \text{falls } x = 0 \\ \dots\dots \text{falls } x < 0 \end{cases}$$

Die Teilbereiche werden charakterisiert durch Muster.

Beispiel:

Fakultätsfunktion n!

Argumentbereich $N = \{0\} \cup N \setminus \{0\}$

Funktionsdefinition enthält eine Folge von Fallregeln.

$p_i \Rightarrow e_i$ p_i ist Muster, e_i ist definierender Ausdruck

fn $\left. \begin{array}{l} p_1 \Rightarrow e_1 \\ | p_2 \Rightarrow e_2 \\ | \vdots \\ | p_n \Rightarrow e_n \end{array} \right\}$ Fallliste

Typforderungen:

- Alle p_i müssen den selben Typ t haben
- Alle e_i müssen den selben Typ t' haben
- Dann hat die Funktion den Typ $t \rightarrow t'$

Beispiel:

```
type inttelnr = {land:int , amt:int , anschl:int}
```

```
val praes:inttelnr = {land=49 , amt=681 , anschl=3021000}
```

```
val vonsb:inttelnr → string =
```

```
  fn = {land=49 , amt =681 , anschl=a} = Int.toString(a)
```

```
    | {land=49 , amt=am , anschl=a} => "0"^Int.toString(am)^Int.toString(a)
```

```
    | {land=l , amt=am , anschl=a} => "00"^Int.toString(l)^Int.toString(am)^Int.toString(a)
```

```
vonsb (praes) = "3021000"
```

```
vonsb {land=33 , amt=1 , anschl=11111111} => "003311111111"
```

Anwendung einer über Fälle definierten Funktion

Anwendung auf e

- 1) Auswertung von e zu Wert v
- 2) Fallauswahl: Musterabgleich zwischen p_1, \dots, p_n **in dieser Reihenfolge** und v, bis ein Muster passt oder bis alle erfolglos versucht wurden

Sei p_i das erste Muster, das passt. Ergebnis des Musterabgleichs ist eine Wertumgebung für die Variablen des Musters. Überschreibung der Original-Wertumgebung durch diese ergibt die Wertumgebung, in welcher e_i ausgewertet wird. Wert ist Funktionsergebnis.

Beispiel:

```
vonsb {land=33 , amt=1 , anschl=11111111} ⇓ "003311111111"
```

Wertumgebung $\left\{ \begin{array}{l} l \rightarrow 33 \\ am \rightarrow 1 \\ anschl \rightarrow 11111111 \end{array} \right\}$

Abkürzende fun-Notation

```
fun f  p1 = e1
      | p2 = e2
      | ...
      | pn = en
```

Case-Ausdruck

```
case e
of    p1 => e1
     | ...
     | pn => en
```

semantisch äquivalent zu:

```
(fn    p1 => e1
     | ...
     | pn => en) e
```

1. Auswertung von e zu v
2. Fallauswahl bzgl. v

Eigenschaften von Fallregeln bzw. Falllisten

1. Fallliste ist **ausschöpfend** (exhaustive), wenn auf jedes Argument mindestens ein Muster passt.

Beispiel: Fallliste von vonsb ist ausschöpfend (für internationale Telefonnummer des Typs inttelnr)

2. Fallregel ist **redundant**, wenn ihr Muster p_i durch ein Muster p_j mit $j < i$ „subsumiert“ wird, d.h. p_j passt immer, wenn p_i passt.

Fallregel i ist dann überflüssig.

Beispiel: alle Fallregeln von vonsb sind nicht redundant

Vertauschung der Reihenfolge: $\underline{fn} \{ \text{land} = l, \text{amt} = am, \text{anschl} = a \}$ – passt auf alle
 \vdots

produziert zwei redundante Fallregeln

Im Interpreter:

- nicht ausschöpfend: Warnung
- redundant: Fehlermeldung

Allgemeine Definition über Fälle

val f = fn	atp ₁₁ atp _{1n} => e ₁		f (a ₁) (a ₂) ... (a _n)
	...		
	atp _{m1} atp _{mn} => e _m		

7.3 Rekursion

f ist rekursiv, wenn eine Anwendung von t (eventuell) zu einer weiteren Anwendung von f führt.

- **direkt:** f tritt in einem der definierenden Ausdrücke in der Definition von f auf.
- **indirekt:** f wird angewendet durch eine Funktion, die eventuell indirekt von f angewendet wurde.

7.3.1 Einleitendes Beispiel

Fakultätsfunktion $n!$

$$0! = 1$$

$$n! = 1 \cdot 2 \cdot \dots \cdot n, n > 0$$

Rekursionsschema für $n > 0$:

$$n! = 1 \cdot 2 \cdot \dots \cdot \underbrace{(n-1)}_{(n-1)!} \cdot n$$

Rekursionsformel für $n > 0$:

$$n! = n * (n - 1)!$$

$$3! \quad 3 > 0$$

$$= 3 \cdot (3 - 1)! = 3 \cdot 2! \quad 2 > 0$$

$$= 3 \cdot 2 \cdot (2 - 1)! = 3 \cdot 2 \cdot 1! \quad 1 > 0$$

$$= 3 \cdot 2 \cdot 1 \cdot (1 - 1)! = 3 \cdot 2 \cdot 1 \cdot 0! \quad \text{Basisfall}$$

$$= 3 \cdot 2 \cdot 1 \cdot 1 = 6$$

Muster hinter dieser rekursiven Definition:

- **Basisfälle:** definiert über Bedingung und nicht rekursiven Ausdruck
im Beispiel: $n = 0$ 1
- **Rekursionsfälle:** Definition der Funktion durch rekursive Anwendung auf „kleinere“ Argumente
im Beispiel: $n > 0$ $(n - 1)!$
- **Terminierung:**
Rekursionsfall benutzt „kleinere“ Argumente. Jedes Argument kann nur endlich oft „kleiner“ werden.
im Beispiel: jede natürliche Zahl kann nur endlich oft um 1 verringert werden.

Programmierung in SML:

```
val rec fact: int → int =
  fn 0 => 1
  | n => n * fact (n - 1)
```

oder

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

Gültigkeitsbereich des eingeführten Funktionsnamens umfasst alle definierenden Ausdrücke

Typüberprüfung: $\text{val rec } f: t \rightarrow t' = p \Rightarrow e$

Forderung: e hat Typ t' unter der Annahme, dass $f: t \rightarrow t'$

Beispiel:

```
val rec fact : int → int =
  0 => 1
  | n => n * fact (n - 1)
```

Fälle: $0 \Rightarrow 1$: $\text{int} \rightarrow \text{int}$

$$n \Rightarrow \underbrace{\underbrace{\underbrace{n}_{\text{int}} * \underbrace{\text{fact}(\underbrace{n}_{\text{int}} - 1)}_{\text{int}}}_{\text{int}}}_{\text{int}}$$

Auswertung von Anwendungen rekursiver Funktionen

Was ist anders als bei der Anwendung nicht rekursiver Funktionen?

Vorkommen von Anwendungen der Funktion in definierenden Ausdrücken. Deshalb müssen ihre Namen in der Wertumgebung (an entsprechende funktionale Objekte) gebunden sein.

Intuitive Vorstellung (etwas ungenau):

Wir ersetzen die Funktionsanwendung durch Definition und setzen Argumente für formale Parameter ein.

Beispiel:

$\text{val rec fact : int} \rightarrow \text{int} = \text{fn } (0 \Rightarrow 1 \mid n \Rightarrow n * \text{fact } (n-1))$
└──────────────────────────────────┘
factdef

fact (3)
 3 * (factdef (2))
 3 * (2 * (factdef (1)))
 3 * (2 * (1 * (factdef (0))))
 3 * (2 * (1 * 1)) = 6

Bei Anwendung fact (*k*) warten bei der letzten Anwendung fact (0) *k* vorherige Anwendungen auf die rechten Operanden der Multiplikation.

Das bedeutet, dass der Speicherplatzbedarf in der Größe von *k* wächst. Gleich kommt eine Funktion, die konstanten Speicherplatz braucht.

7.3.2 Programmieretechnik

akkumulierende Parameter

Fakultätsfunktion, die konstanten Speicherplatz braucht

Technik: zusätzlicher (akkumulierender) Parameter, in dem bei Aufrufen **Teilresultate** berechnet und weitergegeben werden.

Beispiel: akkumulierend
 $\text{fun fact1 } (0, a:\text{int}) = a$ ↓
└──────────────────────────────────┘
 $\mid (n:\text{int}, a:\text{int}) = \text{fact1 } ((n-1), n * a)$

$\text{fun facta } (n:\text{int}) = \text{fact1 } (n, 1)$

Anwendungsbeispiel:

facta (3, 1)
 fact1 (3, 1) → fact1 (2, 3*1) → fact1 (1, 3*2*1) → fact1 (0, 3*2*1*1) = 6

Platzbedarf ist konstant, d.h. unabhängig von der Größe des Arguments.

Weshalb gibt es hier keine „wartenden“ Anwendungen wie in fact?

Definition von fact1 ist **endrekursiv** (tail recursive), d.h. die letzte Operation im definierenden Ausdruck des rekursiven Falls ist die rekursive Anwendung.

Entrekursive Funktionen entsprechen Schleifen in imperativen Programmiersprachen.

Zum Programmierstil:

fact1 wird nur von facta benutzt. Deshalb macht man fact1 zu einer lokalen Funktion von facta:

```
local fun fact1 (0 , a:int) = a
      | fact1 (n:int , a:int) = fact1 (n-1 , n*a)
in fun facta (n:int) = fact1 (n , 1)
end
```

7.3.3 Syntax, Typen, Semantik

Syntax:

Siehe Skript Jörg Bauer:

<http://rw4.cs.uni-sb.de/~joba/Info1/Material/semantik.pdf>

Typen:

$$[\text{valdec2}] \quad \frac{\Gamma \oplus \{x \mapsto (\text{val}, t)\} \vdash e : t}{\Gamma \vdash \text{val } \text{rec } x : t = e \triangleright \Gamma'} \quad \begin{array}{l} x \in \text{vid} \\ t \text{ Funktionstyp} \end{array}$$

implizite Typberechnung durch Mehrfachauftreten von t!

Typumgebung $\Gamma \oplus \{x \mapsto (\text{val}, t)\}$ kann e mit seinem Vorkommen von x typen.

[fundec] braucht neue Deklarationen:

- 4, $T_r \subseteq TU \times \text{mrule} \times ty$ für **Fallregeln** (mrules) der Form $\text{pat} \Rightarrow \text{exp}$

Notation: $\Gamma \vdash \text{mr} \succ t$

„In der Typumgebung Γ hat die Fallregel mr den Typ t“

- 5, $T_m \subseteq TU \times \text{match} \times ty$ für **Falllisten** (matches)

folgender Form: $p_1 \Rightarrow e_1$

$p_n \Rightarrow e_n$

Notation: $\Gamma \vdash m \succeq t$

„In der Typumgebung Γ hat Fallliste den Typ t“

t Funktionstyp

$$[\text{match}] \quad \frac{\Gamma \vdash \text{mr} \succ t \quad \wedge \quad \Gamma \vdash m \succeq t}{\Gamma \vdash \text{mr} \mid m \succeq t}$$

$$[\text{mrule}] \quad \frac{\Gamma \vdash p \succeq (\Gamma', t) \quad \wedge \quad \Gamma \oplus \Gamma' \vdash e : t'}{\Gamma \vdash p \Rightarrow e \succ t \rightarrow t'}$$

$$[\text{fn}] \quad \frac{\Gamma \vdash m \succeq t}{\Gamma \vdash \text{fn } m : t}$$

Semantik:

$$Fkt = \underbrace{(match \times WU)}_{\text{für fn}} \cup \underbrace{(var \times match \times WU)}_{\text{für fun}}$$

WU für Bindung von freien Variablen!

neue Relationen für die Fallauswahl:

$$4. \quad W_r \subseteq WU \times Wert \times mrule \times (WU \cup \{\perp\})$$

für **Fallregeln** $p \Rightarrow e$

Notation: $\rho, v \vdash mr \downarrow v'$

„In der Wertumgebung ρ passt das Muster p der Fallregel nur auf v , und der Ausdruck e der Fallregel mr wertet sich in der entstandenen Wertumgebung zu v' aus.“

oder $\rho, v \vdash mr \downarrow \perp$

„In der Wertumgebung ρ schlägt der Musterabgleich zwischen p und v fehl.“

$$5. \quad W_m \subseteq WU \times Wert \times match \times (Wert \cup \{\perp\})$$

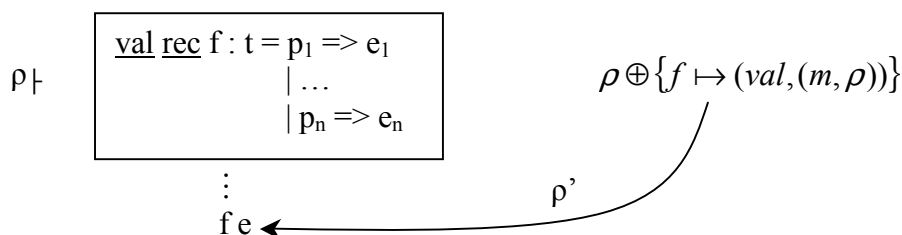
für Falllisten

Notation: $\rho, v \vdash m \searrow v'$

„In ρ trifft eine der Fallregeln aus m auf v zu und der korrespondierende Ausdruck e wertet sich aus zu v' “

oder $\rho, v \vdash m \searrow \perp$

„die Fallauswahl schlägt fehl“

Zusammenspiel zwischen Deklaration und Anwendung einer Funktion

1. e wird **ausgewertet** zu v
2. **Fallauswahl** zwischen m und v
Musterabgleich zwischen p_1 und v_1 , p_n und v_n bis ein Muster p_i passt
3. Auswertung von e_i in der neuen Wertumgebung nach Musterabgleich zwischen p_i und v_i

7.4 Nichtlineare Rekursion

Fakultätsfunktionen: je **eine** rekursive Anwendung der Funktion im definierenden Ausdruck des rekursiven Falls. Deshalb: **lineare Rekursion**

\Rightarrow lineare Zahl von Aufrufen

Berechnung von $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Zahl der Möglichkeiten, k -elementige Teilmengen aus einer n -elementigen Menge zu bilden.

Intuition:

Wir wollen k Elementen aus n auswählen:

Möglichkeiten für die Auswahl des

1. Elements:	n
2. Elements:	$n - 1$
k . Elements:	$n - k + 1$

Insgesamt: $n \cdot (n-1) \cdots (n-k+1)$ Möglichkeiten $= \frac{n!}{(n-k)!}$

Reihenfolge der k Elemente spielt keine Rolle. Es gibt $k!$ verschiedene Reihenfolgen.

Also: $\frac{n!}{k!(n-k)!}$

Rekursionsformel für $\binom{n}{k}$

Basis:

$k = 0$ Eine Möglichkeit, 0 Elemente aus n zu wählen

$n = k$ Eine Möglichkeit, n Elemente aus n zu wählen

Rekursiver Schritt:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Ein Element wird
verworfen und k
Elemente aus dem
Rest ausgewählt

Ein Element wird
ausgewählt und $k-1$
aus dem Rest

Programm in SML:

fun komb ($n:\text{int}$, $k:\text{int}$)

definiert durch:

fun komb ($n:\text{int}$, $k:\text{int}$) =

if $k = 0$ orelse $n = k$

then 1

else komb ($n - 1$, k) + komb ($n - 1$, $k - 1$)

exponentiell viele Anwendungen

komb arbeitet unter der Annahme, $0 \leq k \leq n$, sonst Ausnahme (exception).

7.5 Korrektheitsbeweis für rekursive Funktionen

Ein- / Ausgabe-Spezifikation

- Vorbedingung (precondition) für die Argumente
- Nachbedingung (postcondition) für die Ergebnisse

Behauptung:

Für alle Argumente, welche die Vorbedingung erfüllen, produziert die Funktion Ergebnisse, welche die Nachbedingung erfüllen.

Arten von Korrektheitsaussagen:

- **totale Korrektheit:** Funktion terminiert für alle Argumente, die die Vorbedingung erfüllen und die Ergebnisse erfüllen die Nachbedingung
- **partielle Korrektheit:** für alle Argumente, die die Vorbedingung erfüllen, gilt: wenn die Funktion terminiert, dann erfüllen die Ergebnisse die Nachbedingung.

Ein Beispiel für Ein- / Ausgabespezifikation für eine Funktion vom Typ $t \rightarrow t'$ ist:

Für Argumente vom Typ t kann die Funktion nur Ergebnisse vom Typ t' produzieren.

Wird automatisch durch Typüberprüfer bewiesen.

Was ist das besondere Problem beim Korrektheitsbeweis von rekursiven Funktionen?

Der Beweis benötigt Eigenschaften der rekursiven Aufrufe

Deshalb: **Induktionsbeweise!**

Induktionsanfang: Beweis der zu zeigenden Eigenschaft für Basisfälle

Induktionsschritt: Eigenschaft für die rekursiven Fälle zu zeigen unter Annahme, dass sie für die rekursiven Anwendungen (auf „kleinere“ Argumente) gilt.

Zusammengefasst: Bestandteile eines Korrektheitsbeweises

1. Ein- / Ausgabespezifikation
2. Induktionsbeweis

Beispiel:

Korrektheitsbeweis für die zwei Versionen der Fakultätsfunktion:

Vorbedingung: $n \in \mathbb{N}$

Nachbedingung: $\text{fact}(n) \Downarrow n!$

$\text{facta}(n) \Downarrow n!$

Behauptung: Für $n \in \mathbb{N}$ terminieren $\text{fact}(n)$ und $\text{facta}(n)$ und ergeben $n!$

1. Beweis für fact

Vollständige Induktion über n

Ind.Anf.: $n = 0 : \text{fact}(0) = 1 = 0!$

Ind.Schritt: $n + 1 > 0 : \text{fact}(n + 1) \underset{\text{Def. von fact}}{=} (n + 1) \cdot \text{fact}(n)$

$$\underset{\text{Ind. Ann.}}{=} (n + 1) \cdot n!$$

$$= (n + 1)! \quad \text{q.e.d.}$$

2. Beweis für fact

Eigenschaft: Für $n \in \mathbb{N}$: $\text{facta}(n) \Downarrow n!$ beruht auf der folgenden Eigenschaft von

fun fact1 (0, a:int) = a

| fact1 (n, a:int) = fact1 (n-1, n*a)

Für $n \in \mathbb{N}$ berechnet fact1 (n, r) $n! \cdot r$

Denn facta (n) = fact1 (n, 1)

$\Downarrow n! \cdot 1 = n!$

Beweis der Eigenschaften von fact1 über vollständige Induktion über n:

Ind.Anf.: $n = 0$: fact1 (0, r) = r = 1 · r = 0! · r

Ind.Schritt: $n \rightarrow n+1$

Ind. Annahme: fact1 (n, r) = $n! \cdot r$

$$n+1 > 0: \text{fact1}(n+1, r) \underset{\text{Def. von fact1}}{=} \text{fact1}(n, (n+1) \cdot r) \underset{\text{Ind. Ann.}}{\Downarrow} n! \cdot (n+1) \cdot r = (n+1)! \cdot r \quad \text{q.e.d.}$$

7.6 Simultane Rekursion

Aufgabe: Definiere zwei Funktionen, die feststellen, ob eine gegebene natürliche Zahl gerade bzw. ungerade ist..

0 ist gerade und für $n > 0$ gilt: n ist gerade, wenn $n-1$ ungerade ist

0 ist nicht ungerade und für $n > 0$ gilt: n ist ungerade, wenn $n-1$ gerade ist

fun even 0 = true

| even (n:int) = odd (n-1)

simultan rekursiv

and

odd 0 = false

| odd (n:int) = even (n-1)

Eine Folge von durch **and** verknüpften Deklarationen für Namen v_1, \dots, v_n führt v_1, \dots, v_n mit dem gleichen Gültigkeitsbereich ein, welcher insbesondere die definierenden Ausdrücke für alle v_1, \dots, v_n einschließt.