# Teil V

# Datentypen I





## 9. Quiz: Recordtypen

Gegeben sei die Typdefinition

***type*** $La = \{ li : Bool; lu : Nat \}$

Welche der folgenden Ausdrücke sind Elemente des Typs?

$\{ li = true \}$ ⟹ Nein





## 9. Quiz: Recordtypen

Gegeben sei die Typdefinition

***type*** $La = \{ li : Bool; lu : Nat \}$

Welche der folgenden Ausdrücke sind Elemente des Typs?

Ja ⟸ $\{ li = not\ true; lu = 4711 * 815 \}$





## 9. Quiz: Recordtypen

Gegeben sei die Typdefinition

***type*** $La = \{ li : Bool; lu : Nat \}$

Welche der folgenden Ausdrücke sind Elemente des Typs?

Ja ⟸ $\{ lu = 4711 * 815; li = not\ true \}$

# 9. Quiz: Recordtypen



Gegeben sei die Typdefinition

$\textbf{\textit{type}}\ La = \{\, li : Bool;\ lu : Nat \,\}$

Welche der folgenden Ausdrücke sind Elemente des Typs?

$\{\, la = \texttt{"Lisa"};\ li = not\ \ true;\ lu = 4711 * 815 \,\}$ ⟹ Nein



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

$\textbf{\textit{type}}\ La = \ \mid Li \mid Lu\ \textbf{\textit{of}}\ La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

Ja ⟸ $Li$



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

$\textbf{\textit{type}}\ La = \ \mid Li \mid Lu\ \textbf{\textit{of}}\ La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

$Lu\ \textbf{\textit{of}}\ Li$ ⟹ Nein



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

$\textbf{\textit{type}}\ La = \ \mid Li \mid Lu\ \textbf{\textit{of}}\ La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

$Lu\ La$ ⟹ Nein

# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

***type*** $La = \mid Li \mid Lu$ ***of*** $La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

Ja ⟸ $Lu\ Li$



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

***type*** $La = \mid Li \mid Lu$ ***of*** $La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

Ja ⟸ $Lu\ (Lu\ Li)$



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

***type*** $La = \mid Li \mid Lu$ ***of*** $La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

$Lu\ Lu\ Li$ ⟹ Nein



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

***type*** $La = \mid Li \mid Lu$ ***of*** $La$

Welche der folgenden Ausdrücke sind Elemente des Typs?

$Li\ (Li\ Lu)$ ⟹ Nein

# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

***type*** *La* = { *li* : *Bool*; *lu* : *byte* }

Wieviele Elemente besitzt der Typ?

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

512 ⟵ Anzahl



# 9. Quiz: Variantentypen



Gegeben sei die Typdefinition

***type*** *La* = | *Li* ***of*** *Bool* | *Lu* ***of*** *byte*

Wieviele Elemente besitzt der Typ?

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

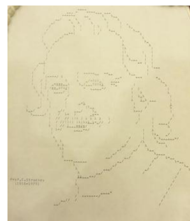
258 ↑ Anzahl



# 10. A tribute to Christopher Strachey (1916—1975)



*It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing ⟨. . .⟩ is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing.*
*— Christopher Strachey*



`#?@ !!` …so kann ich nicht arbeiten …`@#/- &@`

Was ist denn los Harry? So übellaunig kenne ich dich gar nicht.

F# und ich werden keine Freunde: kein **continue**, kein **break**, kein **goto**, …so kann man doch nicht programmieren.

Was willst du denn programmieren?

Die Funktion *product* aus der Vorlesung:

***let rec*** *product* = ***function***
  | [] → 1
  | *x* :: *xs* → *x* ∗ *product xs*

Wenn eins der Listenelemente $0$ ist, will ich die Berechnung abbrechen und sofort $0$ zurückgeben …

Mit meiner Auswertungsregel würde genau das passieren, du müsstest gar nichts machen!

## 10. Beispiel für eine Auswertung



$$
\begin{aligned}
& \mathit{product}\ [4;7;11] \\
= & \quad \{ \text{Definition von } \mathit{product} \} \\
& 4 * \mathit{product}\ [7;11] \\
= & \quad \{ \text{Definition von } \mathit{product} \} \\
& 4 * (7 * \mathit{product}\ [11]) \\
= & \quad \{ \text{Definition von } \mathit{product} \} \\
& 4 * (7 * (11 * \mathit{product}\ [])) \\
= & \quad \{ \text{Definition von } \mathit{product} \} \\
& 4 * (7 * (11 * 1)) \\
= & \quad \{ \text{Definition von '}*\text{'} \} \\
& 4 * (7 * 11) \\
= & \quad \{ \text{Definition von '}*\text{'} \} \\
& 4 * 77 \\
= & \quad \{ \text{Definition von '}*\text{'} \} \\
& 308
\end{aligned}
$$



## 10. Akkumulator: Idee

Datentypen I
Ralf Hinze
Repetitorium
Rechnungen mit Fortsetzungen

- Beobachtung: beim rekursiven Abstieg wird ein Turm von Multiplikationen aufgebaut; beim rekursiven Aufstieg wird der Turm abgebaut/abgearbeitet.
- Speicherbedarf von *product*: linear zur Länge der Liste.
- *Idee:* die gesamte Arbeit auf dem Hinweg erledigen.
- *Spezifikation:*

$$\mathit{worker}\ a\ \mathit{list} = a * \mathit{product}\ \mathit{list} \tag{10}$$

- Die Funktion *worker* erledigt zwei Aufgaben auf einmal; sie verallgemeinert *product*. (Die Spezifikation ähnelt stark dem Rekursionsschritt von *product* — das ist kein Zufall.)

$$\mathit{product}\ \mathit{list} = \mathit{worker}\ 1\ \mathit{list}$$

- Allgemeine Programmiertechnik: Verallgemeinerung — Verstärkung der Rekursionsannahme — Rekursionsparadoxon.



## 10. Akkumulator: Herleitung

Datentypen I
Ralf Hinze
Repetitorium
Rechnungen mit Fortsetzungen

Wir leiten die Definition von *worker* aus der Spezifikation her.

**Fall** $\mathit{list} = []$**:**

$$
\begin{aligned}
& \mathit{worker}\ a\ [] \\
= & \quad \{ \text{Spezifikation von } \mathit{worker}\ (10) \} \\
& a * \mathit{product}\ [] \\
= & \quad \{ \text{Definition von } \mathit{product} \} \\
& a * 1 \\
= & \quad \{ 1 \text{ ist das neutrale Element von '}*\text{'} \} \\
& a
\end{aligned}
$$



## 10. Akkumulator: Herleitung

Datentypen I
Ralf Hinze
Repetitorium
Rechnungen mit Fortsetzungen

**Fall** $\mathit{list} = x :: xs$**:**

$$
\begin{aligned}
& \mathit{worker}\ a\ (x :: xs) \\
= & \quad \{ \text{Spezifikation von } \mathit{worker}\ (10) \} \\
& a * \mathit{product}\ (x :: xs) \\
= & \quad \{ \text{Definition von } \mathit{product} \} \\
& a * (x * \mathit{product}\ xs) \\
= & \quad \{ \text{'}*\text{' ist assoziativ} \} \\
& (a * x) * \mathit{product}\ xs \\
= & \quad \{ \text{Spezifikation von } \mathit{worker}\ (10) \} \\
& \mathit{worker}\ (a * x)\ xs
\end{aligned}
$$

## 10. Akkumulator: Programm



Wir erhalten das folgende Programm.

```
let product =
  let rec worker a = function
    | []      → a
    | x :: xs → worker (a ∗ x) xs
  worker 1
```

Der Parameter *a* ist ein sogenannter *akkumulierender Parameter*, die Funktion *worker* ist *endrekursiv*.



## 10. Akkumulator: Beispiel für eine Auswertung



$$
\begin{aligned}
& \textit{worker}\ 1\ [4;7;11] \\
= & \quad \{\ \text{Definition von } \textit{worker}\ \} \\
& \textit{worker}\ 4\ [7;11] \\
= & \quad \{\ \text{Definition von } \textit{worker}\ \} \\
& \textit{worker}\ 28\ [11] \\
= & \quad \{\ \text{Definition von } \textit{worker}\ \} \\
& \textit{worker}\ 308\ [\,] \\
= & \quad \{\ \text{Definition von } \textit{worker}\ \} \\
& 308
\end{aligned}
$$

Speicherbedarf von *worker*: konstant.



## 10. Akkumulator: Diskussion



Die Herleitung verwendet algebraische Eigenschaften der Multiplikation: '$*$' ist assoziativ und $1$ ist das neutrale Element von '$*$'.

$$
\begin{aligned}
\textit{product}\ [a;b;c;d] &= a * (b * (c * (d * 1))) \\
\textit{worker}\ 1\ [a;b;c;d] &= ((((1 * a) * b) * c) * d)
\end{aligned}
$$

Die gleiche Herleitung funktioniert auch für die Addition, '$+$' und $0$; *nicht* aber für die Subtraktion, '$-$' und $0$.



## 10. Akkumulator: spring raus!



Da *worker* endrekursiv ist, die gesamte Arbeit auf dem Hinweg erledigt, können wir die Rekursion einfach beenden (ohne dass noch ein Turm von Multiplikationen abgearbeitet werden muss).

```
let product =
  let rec worker a = function
    | []      → a
    | x :: xs → if x = 0 then 0
                         else worker (a ∗ x) xs
  worker 1
```

## 10. Akkumulator: Beispiel für eine Auswertung



$$
\begin{aligned}
& \textit{worker}\ 1\ [4; 0; 7; 11] \\
= & \quad \{\ \text{Definition von } \textit{worker}\ \} \\
& \textit{worker}\ 4\ [0; 7; 11] \\
= & \quad \{\ \text{Definition von } \textit{worker}\ \} \\
& 0
\end{aligned}
$$



## 10. Fortsetzungen: Idee



- Beobachtung: beim rekursiven Abstieg wird ein Turm von Multiplikationen aufgebaut; beim rekursiven Aufstieg wird der Turm abgebaut/abgearbeitet.
- Allgemeiner: *product* wird in einem bestimmten Kontext aufgerufen, *cont* (*product list*), der die Berechnung fortsetzt und die Rückgabe von *product* verarbeitet.
- *Idee:* die restliche Arbeit, die sogenannte *Fortsetzung*, der Funktion mit auf den Weg geben!
- *Spezifikation:*

$$\textit{worker list cont} = \textit{cont}\ (\textit{product list}) \qquad (11)$$

  Im Normalfall wird *worker* sein Ergebnis nicht zurückgeben, sondern an die die Fortsetzung weiterreichen.
- (NACA computer room: Wer erhält das Ergebnis einer Zwischenrechnung? Die Auftraggeber*in oder eine Kolleg*in …)
- Die Funktion *worker* verallgemeinert *product*:

$$\textit{product list} = \textit{worker list}\ (\textbf{fun}\ r \rightarrow r)$$



## 10. Fortsetzungen: Herleitung



Wir leiten die Definition von *worker* aus der Spezifikation her.

**Fall** $\textit{list} = [\,]$**:**

$$
\begin{aligned}
& \textit{worker}\ [\,]\ \textit{cont} \\
= & \quad \{\ \text{Spezifikation von } \textit{worker}\ (11)\ \} \\
& \textit{cont}\ (\textit{product}\ [\,]) \\
= & \quad \{\ \text{Definition von } \textit{product}\ \} \\
& \textit{cont}\ 1
\end{aligned}
$$



## 10. Fortsetzungen: Herleitung



**Fall** $\textit{list} = x :: \textit{xs}$**:**

$$
\begin{aligned}
& \textit{worker}\ (x :: \textit{xs})\ \textit{cont} \\
= & \quad \{\ \text{Spezifikation von } \textit{worker}\ (11)\ \} \\
& \textit{cont}\ (\textit{product}\ (x :: \textit{xs})) \\
= & \quad \{\ \text{Definition von } \textit{product}\ \} \\
& \textit{cont}\ (x * \textit{product xs}) \\
= & \quad \{\ \text{Funktionsaufruf}\ \} \\
& (\textbf{fun}\ r \rightarrow \textit{cont}\ (x * r))\ (\textit{product xs}) \\
= & \quad \{\ \text{Spezifikation von } \textit{worker}\ (11)\ \} \\
& \textit{worker xs}\ (\textbf{fun}\ r \rightarrow \textit{cont}\ (x * r))
\end{aligned}
$$

☞ Im dritten Schritt wird der Kontext von *product xs* in eine Funktion überführt.

## 10. Fortsetzungen: Programm



Wir erhalten das folgende Programm.

```
let product list =
  let rec worker xs cont =
    match xs with
     | []      → cont 1
     | x :: xs → worker xs (fun r → cont (x ∗ r))
  worker list (fun r → r)
```

Die Funktion *worker* ist *endrekursiv*.

☞ Wir haben keinerlei Annahmen gemacht, keine Eigenschaften der beteiligten Operationen ausgenutzt.



## 10. Fortsetzungen: Beispiel für eine Auswertung

Datentypen I
Ralf Hinze
Repetitorium
Rechnungen mit Fortsetzungen

$$
\begin{aligned}
& \mathit{worker}\ [4;7;11]\ (\textbf{fun}\ r_1 \to r_1) \\
= \quad & \{\ \text{Definition von } \mathit{worker}\ \} \\
& \mathit{worker}\ [7;11]\ (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2)) \\
= \quad & \{\ \text{Definition von } \mathit{worker}\ \} \\
& \mathit{worker}\ [11]\ (\textbf{fun}\ r_3 \to (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3)) \\
= \quad & \{\ \text{Definition von } \mathit{worker}\ \} \\
& \mathit{worker}\ []\ (\textbf{fun}\ r_4 \to (\textbf{fun}\ r_3 \to (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3))\ (11 * r_4)) \\
= \quad & \{\ \text{Definition von } \mathit{worker}\ \} \\
& (\textbf{fun}\ r_4 \to (\textbf{fun}\ r_3 \to (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3))\ (11 * r_4))\ 1 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\textbf{fun}\ r_3 \to (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ (7 * r_3))\ 11 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2))\ 77 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& (\textbf{fun}\ r_1 \to r_1)\ 308 \\
= \quad & \{\ \text{Funktionsaufruf}\ \} \\
& 308
\end{aligned}
$$



## 10. Fortsetzungen: spring raus!

Datentypen I
Ralf Hinze
Repetitorium
Rechnungen mit Fortsetzungen

Da *worker* endrekursiv ist, können wir die Rekursion einfach beenden, indem wir die Fortsetzung ignorieren.

```
let product list =
  let rec worker xs cont =
    match xs with
     | []      → cont 1
     | x :: xs → if x = 0 then 0   // Fortsetzung wird ignoriert
                         else worker xs (fun r → cont (x ∗ r))
  worker list (fun r → r)
```



## 10. Fortsetzungen: Beispiel für eine Auswertung

Datentypen I
Ralf Hinze
Repetitorium
Rechnungen mit Fortsetzungen

$$
\begin{aligned}
& \mathit{worker}\ [4;0;7;11]\ (\textbf{fun}\ r_1 \to r_1) \\
= \quad & \{\ \text{Definition von } \mathit{worker}\ \} \\
& \mathit{worker}\ [0;7;11]\ (\textbf{fun}\ r_2 \to (\textbf{fun}\ r_1 \to r_1)\ (4 * r_2)) \\
= \quad & \{\ \text{Definition von } \mathit{worker}\ \} \\
& 0
\end{aligned}
$$

## 10. Fortsetzungen: Puzzle

Was macht die folgende Funktion?

```
let puzzle list =
  let rec worker xs cont =
    match xs with
    | []      → cont (cont 1)   // Fortsetzung wird 2-mal verwendet
    | x :: xs → if x = 0 then 0   // Fortsetzung wird ignoriert
                          else worker xs (fun r → cont (x * r))
  worker list (fun r → r)
```





## 10. Fortsetzung folgt ...