21. Knobelaufgabe #9: Die Magie der Zahlen

Sie sind Kandidat*in in der Spielshow „Die Magie der Zahlen“. In der ersten Runde wird ihnen eine Reihe von nummerierten Schachteln präsentiert, die jeweils eine für Sie nicht sichtbare ganze Zahl enthalten. Sie müssen mit möglichst wenigen Versuchen eine *magische Schachtel* finden, eine Schachtel, die ihre eigene Hausnummer enthält.

Die versteckten Zahlen sind alle unterschiedlich. Schachteln mit größeren Hausnummern enthalten größere Zahlen. Entwickeln Sie eine Strategie, um möglichst wenige Schachteln zu öffnen.

- Zum Beispiel:

1	2	3	4	5	6	7	8	9
−10	−5	0	4	7	11	27	65	99

- Es gibt nur eine magische Schachtel: #4.
- Gibt es immer eine magische Schachtel?

21. Motivation

Eine Person ist entweder weiblich oder männlich; eine männliche Person hat Attribute, die eine weibliche nicht hat (und vielleicht umgekehrt — aber nicht modelliert).

$$\textbf{type}\ \mathit{Woman} = \{\mathit{name} : \mathit{String}\}$$
$$\textbf{type}\ \mathit{Man} = \{\mathit{name} : \mathit{String}; \mathit{bald} : \mathit{Bool}\}$$

Daten, die unterschiedliche Ausprägungen besitzen, können wir in Mini–F# mit sogenannten *Varianten* modellieren.

$$\begin{aligned}&\textbf{type}\ \mathit{Person} = \\ &\quad \mid \mathit{Female}\ \textbf{of}\ \mathit{Woman} \\ &\quad \mid \mathit{Male}\ \ \textbf{of}\ \mathit{Man}\end{aligned}$$

21. Motivation

$$\textbf{\textit{type}}\ \mathit{Person} = \begin{array}{l} \mid \mathit{Female}\ \textbf{\textit{of}}\ \mathit{Woman} \\ \mid \mathit{Male}\ \ \ \ \textbf{\textit{of}}\ \mathit{Man} \end{array}$$

Eine Variantentypdefinition führt zwei verschiedene Dinge ein:

- einen Namen für den Variantentyp: *Person*,
- Namen um Elemente des Variantentyps zu konstruieren: *Female* und *Male*. Diese Bezeichner heißen auch *Datenkonstruktoren* oder kurz *Konstruktoren*.

21. Motivation

$$\textbf{type}\ \mathit{Person} = \begin{array}{|l} \mathit{Female}\ \textbf{of}\ \mathit{Woman} \\ \mathit{Male}\quad \textbf{of}\ \mathit{Man} \end{array}$$

Umgangssprachlich lässt sich die Definition wie folgt lesen: ein Element $p : \mathit{Person}$ ist entweder

- von der Form $\mathit{Female}\ e$ mit $e : \mathit{Woman}$ *oder*
- von der Form $\mathit{Male}\ e$ mit $e : \mathit{Man}$.

Ein Konstruktor ähnelt einer Funktion. Der Typ nach dem Konstruktor korrespondiert zum Argumenttyp, der deklarierte Variantentyp korrespondiert zum Ergebnistyp:

- Female hat im Prinzip den Typ $\mathit{Woman} \to \mathit{Person}$ und
- Male den Typ $\mathit{Man} \to \mathit{Person}$.

☞ Im Unterschied zu einer Funktion hat ein Konstruktor aber keine Definition; er steht sozusagen für sich selbst.

21. Motivation

Beispiele:

$$\mathit{Female}\;\{\,\mathit{name} = \texttt{"Lisa"}\,\}$$
$$\mathit{Male}\;\{\,\mathit{name} = \texttt{"Florian"};\ \mathit{bald} = \mathit{false}\,\}$$

☞ Beide Ausdrücke sind vom Typ *Person*.

Ausdrücke können wie immer an Bezeichner gebunden werden.

$$\begin{aligned}
&\textbf{let}\ \mathit{ralf} &&= \mathit{Male}\;\{\,\mathit{name} = \texttt{"Ralf"};\ \mathit{bald} = \mathit{true}\,\}\\
&\textbf{let}\ \mathit{melanie} &&= \mathit{Female}\;\{\,\mathit{name} = \texttt{"Melanie"}\,\}\\
&\textbf{let}\ \mathit{julia} &&= \mathit{Female}\;\{\,\mathit{name} = \texttt{"Julia"}\,\}\\
&\textbf{let}\ \mathit{andres} &&= \mathit{Male}\;\{\,\mathit{name} = \texttt{"Andres"};\ \mathit{bald} = \mathit{false}\,\}
\end{aligned}$$

21. Motivation

Ein Variantentyp beschreibt Alternativen; mit Hilfe der *Fallunterscheidung* ***match*** können wir feststellen, welche konkrete Alternative vorliegt.

```
let dear (person : Person) : String =
  match person with
    | Female female → "Liebe " ^ female.name
    | Male   male   → (if male.bald
                        then "Lieber glatzköpfiger "
                        else "Lieber ") ^ male.name
```

☞ Nach dem Schlüsselwort ***match*** steht der Ausdruck, der analysiert werden soll; die Zweige der Fallunterscheidung führen die verschiedenen Fälle des Variantentyps auf.

21. Variantentypen versus Baumsprachen

Variantentypen ähneln der Notation, mit der wir die abstrakte Syntax unserer Programmiersprache beschreiben.

- Baumsprachen sind Bestandteil der Sprache, mit der wir *über* die Sprache Mini–F# reden.

$$
\begin{aligned}
e \in \mathrm{Expr} ::= {} & \mathit{false} \\
\mid {} & \mathit{true} \\
\mid {} & \mathit{num}\ (\mathbb{N}) \\
\mid {} & \ldots
\end{aligned}
$$

- Variantentypen sind Bestandteil von Mini–F#.

$$
\begin{aligned}
\textbf{\textit{type}}\ \mathit{Expr} = {} & \mid \mathit{False} \\
& \mid \mathit{True} \\
& \mid \mathit{Num}\ \textbf{\textit{of}}\ \mathit{Nat} \\
& \mid \ldots
\end{aligned}
$$

☞ Fachjargon: Variantentypen *internalisieren* Baumsprachen.

21. Abstrakte Syntax

Ein Variantentyp (engl. union type) wird durch eine *Definition* eingeführt.

$C \in \text{Con}$	*Datenkonstruktoren*
$d ::= \cdots$	*Deklarationen:*
$\mid\ \textbf{type}\ T = \mid C_1\ \textbf{of}\ t_1$ $\qquad\qquad\quad \mid C_2\ \textbf{of}\ t_2$	Variantentypdefinition $(C_1 \neq C_2)$

☞ Der Bezeichner T wird durch die Definition neu eingeführt, ebenso die Konstruktoren C_1 und C_2.

☞ Wie Recordtypen dürfen auch Variantentypen weder redefiniert noch lokal definiert werden.

☞ Der Bereich der Konstruktoren Con wird in Teil VI festgelegt.

Für's erste: Ein Konstruktor fängt mit einem *großen* Buchstaben an. Danach können weitere Buchstaben, kleine und große, Ziffern, und Sonderzeichen wie ein Unterstrich oder ein Apostroph folgen.

21. Abstrakte Syntax

Wir erweitern Ausdrücke um Sprachkonstrukte, die Varianten konstruieren bzw. analysieren.

$e ::= \cdots$	*Ausdrücke:*
$\mid\ C\ e$	Konstruktion
$\mid\ \textbf{\textit{match}}\ e\ \textbf{\textit{with}}\ \mid C_1\ x_1 \to e_1$	Fallunterscheidung ($C_1 \neq C_2$)
$\qquad\qquad\qquad\quad \mid C_2\ x_2 \to e_2$	

☞ Der Ausdruck e zwischen den Schlüsselwörtern ***match*** und ***with*** heißt *Diskriminatorausdruck*; $C_1\ x_1 \to e_1$ und $C_2\ x_2 \to e_2$ sind *Zweige* der Fallunterscheidung.

21. Statische Semantik

Die folgenden Typregeln setzen voraus, dass der Variantentyp

$$\textbf{\textit{type}}\ T = \mid C_1\ \textbf{\textit{of}}\ t_1 \mid C_2\ \textbf{\textit{of}}\ t_2$$

bekannt ist.

Typregeln:

$$\frac{\Sigma \vdash e : t_i}{\Sigma \vdash C_i\ e : T}$$

$$\frac{\Sigma \vdash e : T \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash e_1 : t \qquad \Sigma, \{x_2 \mapsto t_2\} \vdash e_2 : t}{\Sigma \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_1\ x_1 \to e_1 \mid C_2\ x_2 \to e_2) : t}$$

☞ Alle Zweige der Fallunterscheidung müssen den gleichen Typ besitzen; dieser ist auch der Typ des gesamten Ausdrucks.

21. Dynamische Semantik

Konstruktoren konstruieren Werte, entsprechend müssen wir den Bereich der Werte erweitern.

$\nu ::= \cdots$	*Werte:*
$\mid \; C\ \nu$	Konstruktion╲Injektion in einen Variantentyp

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash C\ e \Downarrow C\ \nu}$$

$$\frac{\delta \vdash e \Downarrow C_i\ \nu_i \qquad \delta, \{x_i \mapsto \nu_i\} \vdash e_i \Downarrow \nu}{\delta \vdash (\textbf{\textit{match}}\ e\ \textbf{\textit{with}} \mid C_1\ x_1 \to e_1 \mid C_2\ x_2 \to e_2) \Downarrow \nu}$$

21. Varianten mit n Alternativen

Alle Konstrukte verallgemeinern sich in natürlicher Weise auf Varianten mit n Alternativen.

- $n = 0$:
  - keine Alternative: leerer Typ;

    type *Empty* $=$ $|$

  - Fallunterscheidung ohne Zweige: ***match*** *e* ***with***;
  - die leere Fallunterscheidung signalisiert *toten Code*:

    you-cannot-call-me $($*x* : *Empty*$)$: *Nat* $=$ ***match*** *x* ***with***

☞ F# kennt keine leeren Varianten.

21. Varianten mit n Alternativen

- $n = 1$:
  - eine Alternative:

$$\begin{aligned} &\textbf{\textit{type}}\ \mathit{Price} &&= \mid \mathit{Cent}\ \textbf{\textit{of}}\ \mathit{Nat} \\ &\textbf{\textit{type}}\ \mathit{Postcode} &&= \mid \mathit{Code}\ \textbf{\textit{of}}\ \mathit{Nat} \end{aligned}$$

  - Fallunterscheidung hat genau einen Zweig:

$$\begin{aligned} &\textbf{\textit{let}}\ \mathit{double}\ (\mathit{price} : \mathit{Price}) : \mathit{Price} = \\ &\quad \textbf{\textit{match}}\ \mathit{price}\ \textbf{\textit{with}} \mid \mathit{Price}\ n \to \mathit{Price}\ (2 * n) \end{aligned}$$

  - 1-Varianten sehr nützlich: sind *Price* und *Postcode* wie oben definiert, so stellt die statische Semantik sicher, dass wir in einem Programm nicht aus Versehen Preise und Postleitzahlen addieren.
  - Auch lässt sich ein Preis p nicht mit $2 * p$ verdoppeln. Zu diesem Zweck muss *double* verwendet werden.
  - Der Gewinn an Sicherheit wird mit einem Verlust an Bequemlichkeit erkauft.
- $n = 3$:
  - drei Alternativen und Fallunterscheidungen;
- ...

21. Vertiefung

Der Typ *Bool* kann durch einen Variantentyp implementiert werden.

$$\textbf{type}\ \mathit{Bool} = \mid \mathit{False}\ \textbf{of}\ \mathit{Unit} \mid \mathit{True}\ \textbf{of}\ \mathit{Unit}$$

- *false* wird durch *False* () repräsentiert,
- *true* wird durch *True* () repräsentiert,
- $\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$ wird durch die Fallunterscheidung $\textbf{match}\ e_1\ \textbf{with} \mid \mathit{False}\ () \to e_3 \mid \mathit{True}\ () \to e_2$ realisiert.

„Nullstellige“ Konstruktoren wie *False* oder *True* sind relativ häufig. Aus diesem Grund erlauben wir, das Dummyargument auch wegzulassen.

21. Fallstudie: Ganze Zahlen — da capo

Idee: eine ganze Zahl wird durch das Vorzeichen und den Betrag repräsentiert.

$$\textbf{\textit{type}}\ \mathit{Int} = \ \mid \mathit{Neg}\ \textbf{\textit{of}}\ \mathit{Nat} \mid \mathit{Pos}\ \textbf{\textit{of}}\ \mathit{Nat}$$

☞ $\mathit{Neg}\ n$ bezeichnet die Zahl $-n$; $\mathit{Pos}\ n$ entsprechend die Zahl $+n$.

☞ *Invariante:* 0 wird durch $\mathit{Pos}\ 0$ repräsentiert; für $\mathit{Neg}\ n$ gilt stets $n > 0$.

Cleverer Konstruktor:

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{neg}\ (n : \mathit{Nat}) : \mathit{Int} = \\ \quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \mathit{Pos}\ 0\ \textbf{\textit{else}}\ \mathit{Neg}\ n\end{array}$$

☞ *neg* etabliert die Invariante.

21. Ganze Zahlen — Klassifikation

$$
\begin{array}{l}
\textbf{let } \mathit{is\text{-}negative}\ (i : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{match } i \textbf{ with} \\
\qquad \mid \mathit{Neg}\ n \to \mathit{true} \\
\qquad \mid \mathit{Pos}\ n \to \mathit{false} \\
\textbf{let } \mathit{is\text{-}zero}\ (i : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{match } i \textbf{ with} \\
\qquad \mid \mathit{Neg}\ n \to \mathit{false} \\
\qquad \mid \mathit{Pos}\ n \to n = 0 \\
\textbf{let } \mathit{is\text{-}positive}\ (i : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{match } i \textbf{ with} \\
\qquad \mid \mathit{Neg}\ n \to \mathit{false} \\
\qquad \mid \mathit{Pos}\ n \to \mathit{true}
\end{array}
$$

21. Ganze Zahlen — arithmetische Operationen

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{negate}\ (i : \mathit{Int}) : \mathit{Int} = \\
\quad \textbf{\textit{match}}\ i\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Neg}\ n \rightarrow \mathit{Pos}\ n \\
\qquad \mid \mathit{Pos}\ n \rightarrow \mathit{neg}\ n \\
\textbf{\textit{let}}\ \mathit{add}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Int} = \\
\quad \textbf{\textit{match}}\ (i, j)\ \textbf{\textit{with}} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \rightarrow \mathit{Neg}\ (m + n) \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \rightarrow \textbf{\textit{if}}\ m \leqslant n\ \textbf{\textit{then}}\ \mathit{Pos}\ (n \dot{-} m) \\
\qquad \qquad \qquad \qquad \qquad \qquad \quad \textbf{\textit{else}}\ \mathit{Neg}\ (m \dot{-} n) \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \rightarrow \textbf{\textit{if}}\ m < n\ \textbf{\textit{then}}\ \mathit{Neg}\ (n \dot{-} m) \\
\qquad \qquad \qquad \qquad \qquad \qquad \qquad \quad \textbf{\textit{else}}\ \mathit{Pos}\ (m \dot{-} n) \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \rightarrow \mathit{Pos}\ (m + n) \\
\textbf{\textit{let}}\ \mathit{sub}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Int} = \\
\quad \mathit{add}\ (i, \mathit{negate}\ j)
\end{array}
$$

21. Ganze Zahlen — arithmetische Operationen

$$
\begin{array}{l}
\textbf{\textit{let}}\ mul\ (i : Int, j : Int) : Int = \\
\quad \textbf{\textit{match}}\ (i, j)\ \textbf{\textit{with}} \\
\qquad \mid (Neg\ m, Neg\ n) \to Pos\ (m * n) \\
\qquad \mid (Neg\ m, Pos\ n) \to neg\ (m * n) \\
\qquad \mid (Pos\ m, Neg\ n) \to neg\ (m * n) \\
\qquad \mid (Pos\ m, Pos\ n) \to Pos\ (m * n)
\end{array}
$$

☞ *mul* implementiert die Vorzeichenregel: $- * - = +$ etc.

☞ *div* und *mod* zur Übung.

21. Ganze Zahlen — Konversion und Betrag

$$\textbf{\textit{let}}\ \mathit{int}\ (n : \mathit{Nat}) : \mathit{Int} = \mathit{Pos}\ n$$

$$\begin{aligned}
&\textbf{\textit{let}}\ \mathit{abs}\ (i : \mathit{Int}) : \mathit{Nat} = \\
&\quad \textbf{\textit{match}}\ i\ \textbf{\textit{with}} \\
&\qquad \mid \mathit{Neg}\ n \to n \\
&\qquad \mid \mathit{Pos}\ n \to n
\end{aligned}$$

21. Ganze Zahlen — Vergleichsoperationen

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{less}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ (i, j)\ \textbf{\textit{with}} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \rightarrow m > n \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \rightarrow \mathit{true} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \rightarrow \mathit{false} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \rightarrow m < n \\
\textbf{\textit{let}}\ \mathit{less\text{-}equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ (i, j)\ \textbf{\textit{with}} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \rightarrow m \geqslant n \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \rightarrow \mathit{true} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \rightarrow \mathit{false} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \rightarrow m \leqslant n \\
\textbf{\textit{let}}\ \mathit{equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ (i, j)\ \textbf{\textit{with}} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \rightarrow m = n \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \rightarrow \mathit{false} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \rightarrow \mathit{false} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \rightarrow m = n
\end{array}
$$

21. Ganze Zahlen — Vergleichsoperationen

$$
\begin{array}{l}
\textbf{let } \mathit{not\text{-}equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{match } (i, j) \textbf{ with} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \to m <> n \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \to \mathit{true} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \to \mathit{true} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \ \to m <> n \\
\textbf{let } \mathit{greater\text{-}equal}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{match } (i, j) \textbf{ with} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \to m \leqslant n \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \to \mathit{false} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \to \mathit{true} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \ \to m \geqslant n \\
\textbf{let } \mathit{greater}\ (i : \mathit{Int}, j : \mathit{Int}) : \mathit{Bool} = \\
\quad \textbf{match } (i, j) \textbf{ with} \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Neg}\ n) \to m < n \\
\qquad \mid (\mathit{Neg}\ m, \mathit{Pos}\ n) \to \mathit{false} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Neg}\ n) \to \mathit{true} \\
\qquad \mid (\mathit{Pos}\ m, \mathit{Pos}\ n) \ \to m > n
\end{array}
$$

21. Ganze Zahlen — Demo

Mini⟩ *int* 4711
Pos 4711
Mini⟩ *add* (*negate* (*int* 4711), *int* 815)
Neg 3896
Mini⟩ *add* (*int* 4711, *negate* (*int* 4711))
Pos 0
Mini⟩ *is-zero* (*add* (*int* 4711, *negate* (*int* 4711)))
true
Mini⟩ *mul* (*negate* (*int* 2), *negate* (*int* 3))
Pos 6
Mini⟩ *abs* (*mul* (*negate* (*int* 2), *int* 3))
6
Mini⟩ *div* (*negate* (*int* 4), *int* 3)
Neg 2
Mini⟩ *mod* (*negate* (*int* 4), *int* 3)
Pos 2

☞ Für $b \neq 0$ gilt weiterhin $a = (a \div b) * b + (a \,\%\, b)$. *Aber:* die Eigenschaft legt $\div$ und % nicht länger eindeutig fest, siehe Übung.

21. Summen und Produkte

Ist t ein endlicher Typ, so bezeichnet $|t|$ die Anzahl der Elemente von t, die *Kardinalität* von t.

1. Der Paartyp $t_1 * t_2$ korrespondiert zu einem *Produkt*, da

$$|t_1 * t_2| = |t_1| * |t_2|$$

2. Der Typ *Unit* korrespondiert zu der 1, da

$$|\mathit{Unit}| = 1$$

3. Der Variantentyp T mit ***type*** $T = \mid C_1$ ***of*** $t_1 \mid C_2$ ***of*** t_2 korrespondiert zu einer *Summe*, da

$$|T| = |t_1| + |t_2|$$

4. Der Variantentyp *Empty* mit ***type*** *Empty* $= \mid$ korrespondiert zu der 0, da

$$|\mathit{Empty}| = 0$$

☞ Und Funktionen?

21. Summen und Produkte — Eigenschaften

Für Paar- und Variantentypen gelten ähnliche Gesetze wie für die natürlichen Zahlen. Zum Beispiel:

$$0 + t \cong t \cong t + 0$$

$$0 \times t \cong 0 \cong t \times 0$$

$$1 \times t \cong t \cong t \times 1$$

$$t_1 \times (t_2 + \tau_3) \cong (t_1 \times t_2) + (t_1 \times \tau_3)$$

$$(t_1 + t_2) \times \tau_3 \cong (t_1 \times \tau_3) + (t_2 \times \tau_3)$$

☞ Im Unterschied zu den natürlichen Zahlen sind die beiden oder die drei Seiten nicht gleich, sondern nur *isomorph*: $t_1 \cong t_2$ bedeutet, dass sich jedes Element aus t_1 eineindeutig einem Element aus t_2 zuordnen lässt.

☞ Und Funktionen?

21. Rechnen mit Typen

Der Typ *Person* in arithmetischer Notation:

$$Person \cong String + (String \times Bool)$$

Jetzt können wir rechnen:

$$\begin{aligned} & String + (String \times Bool) \\ \cong \quad & \{ \text{1 ist das neutrale Element von ‘}\times\text{’} \} \\ & (String \times 1) + (String \times Bool) \\ \cong \quad & \{ \text{Distributivgesetz} \} \\ & String \times (1 + Bool) \end{aligned}$$

☞ Der Typ $String \times (1 + Bool)$ ist eine alternative Implementierung von *Person*.

☞ Für das Rechnen mit Typen (!) ist die arithmetische Notation sehr bequem — es ist aber nur eine Notation!

21. Rechnen mit Typen

Übersetzen wir den Typ $\mathit{String} \times (1 + \mathit{Bool})$ in Mini–F# Notation, so erhalten wir

$$\textbf{type}\ \mathit{Person}' = \{\, \mathit{name} : \mathit{String}; \mathit{gender} : \mathit{Gender} \,\}$$
$$\textbf{type}\ \mathit{Gender} = \mathit{Female}' \mid \mathit{Male}'\ \textbf{of}\ \{\, \mathit{bald} : \mathit{Bool} \,\}$$

☞ Das Geschlecht umfasst die trennenden Merkmale, die gemeinsamen sind in Person' zusammengefasst.

22. Motivation

- Mit den Konstrukten, die wir bisher eingeführt haben, können wir nur eine beschränkte Anzahl von Daten zusammenfassen:
  - ein 7-Tupel aggregiert 7 Daten,
  - ein 128-Tupel 128 Daten.

  ☞ Beide Typen sind ungeeignet um 6, 8, 127 oder 129 Daten aufzunehmen.
- Zum Zeitpunkt des Programmierens kennt man häufig die genaue Anzahl von Daten nicht:
  - Wieviele Personen immatrikulieren sich im WS 2019/2020?
  - Wieviele Unternehmen sind an der Börse notiert?
  - Wieviele Mitarbeiter*innen hat eine Abteilung?
  - usw.
- Wie können wir eine beliebige Anzahl von Daten aggregieren?

22. Motivation

Konkret: wie können wir eine Folge von natürlichen Zahlen repräsentieren?

Ein erster Versuch (in arithmetischer Notation):

$$
\begin{aligned}
\mathit{Nats} \cong\ & 1 \\
& + \mathit{Nat} \\
& + \mathit{Nat} \times \mathit{Nat} \\
& + \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \\
& + \ \ldots
\end{aligned}
$$

☞ Eine Folge von natürlichen Zahlen ist entweder die leere Folge (ein 0-Tupel), oder eine einelementige Folge (ein „1-Tupel“), oder eine zweielementige Folge (ein 2-Tupel) usw.

22. Motivation

Beobachtung: alle Alternativen bis auf die erste haben eine *Nat* Komponente. Der gemeinsame Faktor kann „herausgezogen" werden.

$$
\begin{aligned}
\mathit{Nats} \cong\ & 1 \\
& + \mathit{Nat} \times \ (\ 1 \\
& \qquad\qquad + \mathit{Nat} \\
& \qquad\qquad + \mathit{Nat} \times \mathit{Nat} \\
& \qquad\qquad + \ldots)
\end{aligned}
$$

22. Motivation

Der Typausdruck in Klammern entspricht der ursprünglichen Definition von *Nats*.

$$Nats \cong 1 + Nat \times Nats$$

Erlauben wir bei der Definition eines Typs den Rückgriff auf den definierten Typ selbst (!), erhalten wir (in Mini–F# Notation):

$$\textbf{\textit{type}}\ Nats = \ |\ Nil \ |\ Cons\ \textbf{\textit{of}}\ Nat * Nats$$

☞ *Zur Erinnerung:* Greift man bei der Definition auf das definierte Objekt selbst zurück, spricht man von einer *rekursiven* Definition.

22. Listen

$$\textbf{type}\ \mathit{Nats} = \ \mid \mathit{Nil} \mid \mathit{Cons}\ \textbf{of}\ \mathit{Nat} * \mathit{Nats}$$

☞ Eine Folge von natürlichen Zahlen ist entweder die leere Folge *Nil* oder eine mindestens einelementige Folge $\mathit{Cons}\,(n, ns)$ bestehend aus einer natürlichen Zahl n und einer Folge von natürlichen Zahlen ns.

- *Nil* ist eine Verkürzung des lateinischen Wortes *nihil* für „nichts“.
- *Cons* kürzt das englische Wort *construct* ab.
- Statt von einer Folge von natürlichen Zahlen spricht man auch kurz von einer *Liste*:
  - n ist das *Kopfelement* der Liste $\mathit{Cons}\,(n, ns)$,
  - ns ist die *Restliste* der Liste $\mathit{Cons}\,(n, ns)$.
- Bei Listen ist wie bei Tupeln die Reihenfolge der Elemente signifikant.
- Listen sind die erste und einfachste *Datenstruktur*, die uns begegnet. Eine Datenstruktur verwaltet Daten und unterstützt Zugriff und Manipulation dieser Daten.

22. Listen — Programmierung

Wie gehen wir mit einem rekursiven Variantentyp um? Wir konstruieren und analysieren rekursive Varianten mit Hilfe rekursiver Funktionen!

Beispiel: Sortieren von Listen. Der Variantentyp gibt das folgende Skelett für *sort* vor.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{sort}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\ \to \ \ldots \\
\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \ \ldots
\end{array}
$$

☞ Wie füllen wir den *Cons* Zweig mit Leben?

22. Sortieren

Getreu dem Motto „rekursive Funktionen für rekursive Typen“ erlauben wir, *sort* auf die Restliste *ns* anzuwenden.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{sort}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\quad \to \ \ldots \\
\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \ \ldots \mathit{sort}\ \mathit{ns} \ldots
\end{array}
$$

22. Sortieren

- *Rekursionsbasis*: die leere Liste ist bereits geordnet.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{sort}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\ \ \to \mathit{Nil} \\
\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \ldots \mathit{sort}\ \mathit{ns} \ldots
\end{array}
$$

- *Rekursionsschritt*: *sort ns* ist eine geordnete Liste; wir müssen das Element n an die richtige Stelle einordnen. Wir geben dieser Teilaufgabe einen Namen: *insert*.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{sort}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\ \ \to \mathit{Nil} \\
\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \mathit{insert}\ (n, \mathit{sort}\ \mathit{ns})
\end{array}
$$

☞ Wenn sich ein Problem nicht mit dem bisherigen Vokabular lösen lässt, müssen wir das Vokabular erweitern. An dieser Stelle ist Kreativität gefragt!

22. Einfügen in eine geordnete Liste

Die Definition von *insert* gehen wir auf die gleiche Art und Weise an.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ insert\ (nat : Nat, nats : Nats) : Nats = \\
\quad \textbf{\textit{match}}\ nats\ \textbf{\textit{with}} \\
\qquad \mid Nil \qquad\qquad \rightarrow \ \ldots \\
\qquad \mid Cons\ (n, ns) \rightarrow \ \ldots insert\ (nat, ns) \ldots
\end{array}
$$

Vorbedingung: nats ist bereits geordnet. Das Typsystem stellt diese Eigenschaft nicht sicher, darum müssen wir uns kümmern.

22. Einfügen in eine geordnete Liste

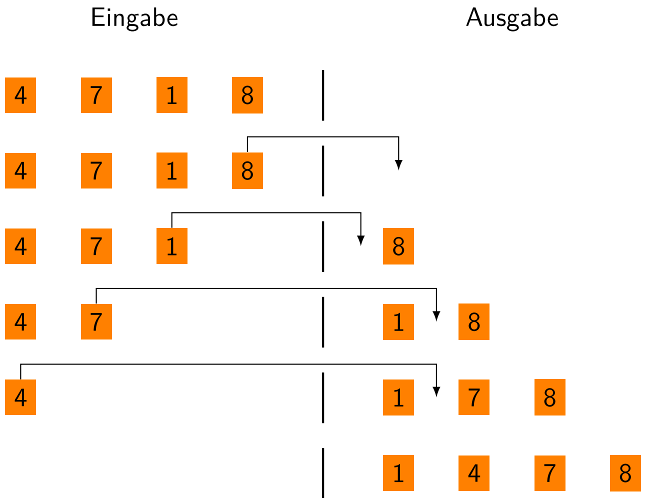
- *Rekursionsbasis*: Ist die Liste leer, so geben wir die einelementige Liste $\mathit{Cons}\,(\mathit{nat}, \mathit{Nil})$ zurück.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{insert}\ (\mathit{nat} : \mathit{Nat}, \mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\quad\ \to \mathit{Cons}\,(\mathit{nat}, \mathit{Nil}) \\
\qquad \mid \mathit{Cons}\,(n, \mathit{ns}) \to \ldots
\end{array}
$$

- *Rekursionsschritt*: gilt $\mathit{nat} \leqslant n$, so stellen wir nat an den Anfang der Liste; anderenfalls wird nat in die Restliste ns eingefügt.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{insert}\ (\mathit{nat} : \mathit{Nat}, \mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\quad\ \to \mathit{Cons}\,(\mathit{nat}, \mathit{Nil}) \\
\qquad \mid \mathit{Cons}\,(n, \mathit{ns}) \to \textbf{\textit{if}}\ \mathit{nat} \leqslant n \\
\qquad \qquad\qquad\qquad\quad\ \ \textbf{\textit{then}}\ \mathit{Cons}\,(\mathit{nat}, \mathit{nats}) \\
\qquad \qquad\qquad\qquad\quad\ \ \textbf{\textit{else}}\ \ \mathit{Cons}\,(n, \mathit{insert}\,(\mathit{nat}, \mathit{ns}))
\end{array}
$$

22. Sortieren durch Einfügen



22. Sortieren — Verallgemeinerung

Die Funktion *sort* sortiert eine Liste aufsteigend. Was machen wir, wenn wir die Liste absteigend ordnen wollen?

- Programmcode duplizieren und '$\leqslant$' systematisch durch '$\geqslant$' ersetzen. Unökonomisch!
- Aufsteigend sortieren und das Ergebnis umdrehen, siehe Skript.
- Wir verallgemeinern die Aufgabenstellung und abstrahieren von einer speziellen Ordnungsrelation.

$$\mathit{sort\text{-}by}\;(\mathit{less\text{-}equal} : \mathit{Nat} * \mathit{Nat} \to \mathit{Bool}) : \mathit{Nats} \to \mathit{Nats}$$

22. Sortieren — Verallgemeinerung

Sortieren durch Einfügen

$$
\begin{array}{l}
\textbf{let } \mathit{sort\text{-}by}\ (\mathit{less\text{-}equal} : \mathit{Nat} * \mathit{Nat} \to \mathit{Bool}) : \mathit{Nats} \to \mathit{Nats} = \\
\quad \textbf{let rec } \mathit{insert}\ (\mathit{nat} : \mathit{Nat}, \mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad\quad \textbf{match } \mathit{nats} \textbf{ with} \\
\quad\quad\quad \mid \mathit{Nil} \to \mathit{Cons}\ (\mathit{nat}, \mathit{Nil}) \\
\quad\quad\quad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \textbf{if } \mathit{less\text{-}equal}\ (\mathit{nat}, n) \\
\quad\quad\quad\quad\quad \textbf{then } \mathit{Cons}\ (\mathit{nat}, \mathit{nats}) \\
\quad\quad\quad\quad\quad \textbf{else } \mathit{Cons}\ (n, \mathit{insert}\ (\mathit{nat}, \mathit{ns})) \\
\quad \textbf{let rec } \mathit{sort}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nats} = \\
\quad\quad \textbf{match } \mathit{nats} \textbf{ with} \\
\quad\quad\quad \mid \mathit{Nil} \to \mathit{Nil} \\
\quad\quad\quad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \mathit{insert}\ (n, \mathit{sort}\ \mathit{ns}) \\
\quad \textbf{in } \mathit{sort}
\end{array}
$$

☞ die Definition von *sort-by* verwendet Layout (Abseitsregel).

22. Sortieren — Verallgemeinerung

Die ursprünglichen Sortierfunktionen sind jetzt hausbackene Spezialfälle:

$$\textbf{let } \mathit{increasing\text{-}sort} = \mathit{sort\text{-}by}\,(\textbf{fun}\,(m, n) \rightarrow m \leqslant n)$$
$$\textbf{let } \mathit{decreasing\text{-}sort} = \mathit{sort\text{-}by}\,(\textbf{fun}\,(m, n) \rightarrow m \geqslant n)$$

☞ Die Funktion *sort-by* ist ein weiteres Beispiel für eine Funktion höherer Ordnung: *sort-by* nimmt eine Funktion als Argument (*less-equal*) und gibt eine Funktion (*sort*) als Ergebnis zurück.

22. Struktur Entwurfsmuster für Listen

Haben wir die Aufgabe eine Funktion $f : Nats \to t$ zu erstellen, dann sieht ein erster Entwurf folgendermaßen aus.

$$
\begin{array}{ll}
\textbf{\textit{let rec}}\ f\ (nats : Nats) : t = & \textit{Struktur Entwurfsmuster} \\
\quad \textbf{\textit{match}}\ nats\ \textbf{\textit{with}} & \\
\qquad \mid Nil \qquad\quad\ \to \ldots & \textit{Rekursionsbasis} \\
\qquad \mid Cons\ (n, ns) \to \ldots n \ldots f\ ns \ldots & \textit{Rekursionsschritt}
\end{array}
$$

Die Ellipsen müssen mit Leben gefüllt werden:

- *Rekursionsbasis*: ein Ausdruck des Typs t.
- *Rekursionsschritt*: ein Ausdruck, der die Teillösung $f\ ns$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert.

22. Summe einer Liste von Zahlen

Aufgabe: sum nats soll die Elemente der Liste *nats* aufaddieren (die Verallgemeinerung von $+$ auf Listen).

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ sum\ (nats : Nats) : Nat = \\
\quad \textbf{\textit{match}}\ nats\ \textbf{\textit{with}} \\
\qquad \mid Nil \qquad\qquad\quad \to \ \ldots \\
\qquad \mid Cons\ (n, ns) \to \ \ldots sum\ ns \ldots
\end{array}
$$

22. Summe einer Liste von Zahlen

- *Rekursionsbasis*: *sum Nil* $= 0$. Warum?

$$
\begin{aligned}
&\textbf{\textit{let rec}}\ \mathit{sum}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nat} = \\
&\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
&\qquad \mid \mathit{Nil} \qquad\qquad\ \ \to 0 \\
&\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \ \ldots \mathit{sum}\ \mathit{ns} \ldots
\end{aligned}
$$

- *Rekursionsschritt*: Wir addieren n zur Summe der Restliste.

$$
\begin{aligned}
&\textbf{\textit{let rec}}\ \mathit{sum}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nat} = \\
&\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
&\qquad \mid \mathit{Nil} \qquad\qquad\ \ \to 0 \\
&\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to n + \mathit{sum}\ \mathit{ns}
\end{aligned}
$$

22. Produkt einer Liste von Zahlen

Aufgabe: product nats soll die Elemente der Liste *nats* miteinander multiplizieren (die Verallgemeinerung von $*$ auf Listen).

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{product}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nat} = \\
\quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\qquad\quad \to \ldots \\
\qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \ldots \mathit{product}\ \mathit{ns} \ldots
\end{array}
$$

22. Produkt einer Liste von Zahlen

- *Rekursionsbasis*: *product Nil* $= 1$. Warum?

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{product}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nat} = \\ \quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\ \qquad \mid \mathit{Nil} \qquad\qquad\quad \to 1 \\ \qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to \ \ldots \mathit{product}\ \mathit{ns} \ldots \end{array}$$

- *Rekursionsschritt*: Wir multiplizieren n mit dem Produkt der Restliste.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{product}\ (\mathit{nats} : \mathit{Nats}) : \mathit{Nat} = \\ \quad \textbf{\textit{match}}\ \mathit{nats}\ \textbf{\textit{with}} \\ \qquad \mid \mathit{Nil} \qquad\qquad\quad \to 1 \\ \qquad \mid \mathit{Cons}\ (n, \mathit{ns}) \to n * \mathit{product}\ \mathit{ns} \end{array}$$

22. Konstruktion von Listen

Die bisherigen Funktionen verarbeiten Listen; wie können wir Listen erzeugen?

Beispiel: $between\ (l, u)$ erzeugt die Liste aller Elemente in dem gegebenen Intervall.

Wir wenden das Peano Entwurfsmuster auf die Intervallgröße an.

$$
\begin{array}{l}
\textbf{let rec}\ between\ (l : Nat, u : Nat) : Nats = \\
\quad \textbf{if}\ \ l > u\ \textbf{then}\ Nil \\
\qquad\qquad\ \textbf{else}\ \ Cons\ (l, between\ (l + 1, u))
\end{array}
$$

☞ Im Basisfall geben wir die leere Liste zurück; im Rekursionsfall setzen wir die linke Intervallgrenze vor die rekursiv erzeugte Liste.

22. Alte Funktionen neu

Wir haben unser Vokabular beträchtlich erweitert. Mit den neuen Vokabeln können wir zum Beispiel *factorial* kürzer definieren.

$$\textbf{let}\ \mathit{factorial}\ (n : \mathit{Nat}) : \mathit{Nat} = \mathit{product}\ (\mathit{between}\ (1, n))$$

☞ Wie lässt sich *power* auf *product* zurückführen?