22. Knobelaufgabe #10

Ministerpräsident Sörkus Mader möchte im Rahmen seines Raumfahrtprogramms „Failaria One“ einen Roboter auf dem Mars absetzen. Dazu wird der Roboter aus der Trägerrakete auf den Planeten fallen gelassen. Hierbei ist die Fallhöhe von entscheidender Bedeutung: Wird sie zu hoch angesetzt, dann zerschellt der Roboter an der Oberfläche des Planeten. Eine zu niedrige Höhe birgt das Risiko, dass die Trägerrakete dem Planeten zu nahe kommt und explodiert.

In der Forschungsstation „Dahoam“ wurde ein Turm errichtet, der die Fallbedingungen exakt nachstellt. Es stehen zwei Imitate des Roboters zur Verfügung, die die gleichen Aufpralleigenschaften wie der echte Roboter aufweisen. Das Roboter-Imitat wird jeweils aus einem der n Stockwerke des Versuch-Turms fallen gelassen, um das höchste Stockwerk zu ermitteln, bei dem der Roboter unversehrt am Boden ankommt. Da jeder Versuch mit hohen Kosten für den Steuerzahler verbunden ist, soll die Anzahl der Fall-Versuche minimiert werden. Allerdings stehen auch nur die genannten beiden Imitate zur Verfügung — beim Fall aus zu großer Höhe werden sie zerstört und sind unbrauchbar.

Helfen Sie Sörkus eine Strategie zu entwickeln, um das höchste Stockwerk, aus dem der Roboter unversehrt fallen gelassen werden kann, exakt zu ermitteln bei minimaler Anzahl von Fall-Versuchen.

22. Vertiefung

Der Typ *Bool* kann mit einem Variantentyp implementiert werden; *Nat* lässt sich mit einem rekursiven Variantentyp implementieren!

Eine natürliche Zahl ist entweder 0 oder der Nachfolger einer natürlichen Zahl $(n + 1)$.

Denken wir uns Namen für die 0 und die Nachfolgerfunktion aus.

```
type Peano =
    | Zero
    | Succ of Peano
```

22. Natürliche Zahlen — Peano

Wir zählen

Zero
Succ Zero
Succ (*Succ Zero*)
Succ (*Succ* (*Succ Zero*))
Succ (*Succ* (*Succ* (*Succ Zero*)))
...

☞ Die Zahlendarstellung unserer Urahnen: für jedes erlegte Bison ein Strich.

22. Addition — Peano

Die Addition lässt sich auf die Nachfolgerfunktion zurückführen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{add}\ (m : \mathit{Peano}, n : \mathit{Peano}) : \mathit{Peano} = \\
\quad \textbf{\textit{match}}\ m\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Zero} \quad\ \to \ldots \\
\qquad \mid \mathit{Succ}\ m' \to \ldots \mathit{add}\ (m', n) \ldots
\end{array}
$$

22. Addition — Peano

- *Rekursionsbasis*: $0 + n = n$.

```
let rec add (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → n
    | Succ m' → ... add (m', n) ...
```

- *Rekursionsschritt*: $(m' + 1) + n = (m' + n) + 1$.

```
let rec add (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → n
    | Succ m' → Succ (add (m', n))
```

22. Multiplikation — Peano

Die Multiplikation lässt sich auf die Addition zurückführen.

```
let rec mul (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → ...
    | Succ m' → ... mul (m', n) ...
```

22. Multiplikation — Peano

- *Rekursionsbasis*: $0 * n = 0$.

```
let rec mul (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → Zero
    | Succ m' → ... mul (m', n) ...
```

- *Rekursionsschritt*: $(m' + 1) * n = m' * n + n$.

```
let rec mul (m : Peano, n : Peano) : Peano =
  match m with
    | Zero    → Zero
    | Succ m' → add (mul (m', n), n)
```

22. Peano Entwurfsmuster — da capo

Das Peano Entwurfsmuster entspricht dem Struktur Entwurfsmuster für den Typ *Peano*.

Peano Entwurfsmuster:

```
let rec f (n : Nat) : t =          Peano Entwurfsmuster:
  if n = 0
    then ...                       Rekursionsbasis
    else ... f (n ÷ 1) ...         Rekursionsschritt
```

Struktur Entwurfsmuster für den Typ *Peano*:

```
let rec f (n : Peano) : t =        Struktur Entwurfsmuster:
  match n with
    | Zero    → ...                Rekursionsbasis
    | Succ n' → ... f n' ...       Rekursionsschritt
```

22. Natürliche Zahlen — Leibniz

Eine natürliche Zahl ist entweder 0, gerade $(2n)$ oder ungerade $(2n+1)$.

Die Fälle sind aber nicht exklusiv: 0 ist auch eine gerade Zahl.

Na ja, wie ich das verstanden habe, brauchen wir die 0 als Rekursionsbasis.

Ich will nicht die 0 streichen, sondern gerade Zahlen auf $n \geqslant 2$ beschränken $(2n+2)$.

Vom höchsten Ordnungssinn ist es nur ein Schritt zur Pedanterie. — Eine natürliche Zahl ist entweder 0, ungerade $(2n+1)$ oder positiv gerade $(2n+2)$.

Nur als Warnung: wir weichen an dieser Stelle vom Skript ab.

22. Natürliche Zahlen — Leibniz

Denken wir uns Namen für die 0, ungerade Zahlen $(2n + 1)$ und gerade Zahlen größer als 0 $(2n + 2)$ aus.

```
type Leibniz =
    | Null
    | Odd of Leibniz
    | Even of Leibniz
```

22. Natürliche Zahlen — Leibniz

Wir zählen

Null
Odd Null
Even Null
Odd (*Odd Null*)
Even (*Odd Null*)
Odd (*Even Null*)
Even (*Even Null*)
Odd (*Odd* (*Odd Null*))
Even (*Odd* (*Odd Null*))
Odd (*Even* (*Odd Null*))
...

22. Nachfolgerfunktion — Leibniz

Um das Bildungsgesetz zu sehen, ist es am einfachsten, wenn wir die Nachfolgerfunktion programmieren.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null     → ...
    | Odd  n'  → ... succ n' ...
    | Even n'  → ... succ n' ...
```

- *Rekursionsbasis*: $0 + 1 = 1$.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null     → Odd Null
    | Odd  n'  → ... succ n' ...
    | Even n'  → ... succ n' ...
```

22. Nachfolgerfunktion — Leibniz

- *Rekursionsschritt*: $(2 * n' + 1) + 1 = 2 * n' + 2$.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null    → Odd Null
    | Odd  n' → Even n'
    | Even n' → ... succ n' ...
```

- *Rekursionsschritt*: $(2 * n' + 2) + 1 = 2 * (n' + 1) + 1$.

```
let rec succ (n : Leibniz) : Leibniz =
  match n with
    | Null    → Odd Null
    | Odd  n' → Even n'
    | Even n' → Odd (succ n')
```

22. Addition — Leibniz

Addition:

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
    | Null      → ...
    | Odd  m'   → ... add (m', n) ...
    | Even m'   → ... add (m', n) ...
```

- *Rekursionsbasis*: $0 + n = n$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
    | Null      → n
    | Odd  m'   → ... add (m', n) ...
    | Even m'   → ... add (m', n) ...
```

22. Addition — Leibniz

- *Rekursionsschritt*: Können wir die Summe berechnen, ohne n zu kennen? Nein, wir müssen eine geschachtelte Fallunterscheidung über n vornehmen.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m' → match n with
                | Null    → ...
                | Odd  n' → ... add (m', n') ...
                | Even n' → ... add (m', n') ...
  | Even m' → match n with
                | Null → ...
                | Odd  n' → ... add (m', n') ...
                | Even n' → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $n = \mathit{Null}$: $m + 0 = m$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null     → m
                | Odd  n'  → ... add (m', n') ...
                | Even n'  → ... add (m', n') ...
  | Even m'   → match n with
                | Null     → m
                | Odd  n'  → ... add (m', n') ...
                | Even n'  → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $m = \mathit{Odd}\ m'$ und $n = \mathit{Odd}\ n'$: $(2m' + 1) + (2n' + 1) = 2(m' + n') + 2$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null      → m
                | Odd  n'   → Even (add (m', n'))
                | Even n'   → ... add (m', n') ...
  | Even m'   → match n with
                | Null      → m
                | Odd  n'   → ... add (m', n') ...
                | Even n'   → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $m = \mathit{Odd}\ m'$ und $n = \mathit{Even}\ n'$: $(2m' + 1) + (2n' + 2) = 2((m' + n') + 1) + 1$.
- Fall $m = \mathit{Even}\ m'$ und $n = \mathit{Odd}\ n'$: analog.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null      → m
                | Odd  n'   → Even (add (m', n'))
                | Even n'   → Odd (succ (add (m', n')))
  | Even m'   → match n with
                | Null      → m
                | Odd  n'   → Odd (succ (add (m', n')))
                | Even n'   → ... add (m', n') ...
```

22. Addition — Leibniz

- Fall $m = \mathit{Even}\ m'$ und $n = \mathit{Even}\ n'$: $(2m' + 2) + (2n' + 2) = 2((m' + n') + 1) + 2$.

```
let rec add (m : Leibniz, n : Leibniz) : Leibniz =
  match m with
  | Null      → n
  | Odd  m'   → match n with
                | Null      → m
                | Odd  n'   → Even (add (m', n'))
                | Even n'   → Odd (succ (add (m', n')))
  | Even m'   → match n with
                | Null      → m
                | Odd  n'   → Odd (succ (add (m', n')))
                | Even n'   → Even (succ (add (m', n')))
```

22. Leibniz Entwurfsmuster — da capo

Das Leibniz Entwurfsmuster entspricht dem Struktur Entwurfsmuster für den Typ *Leibniz*.

Leibniz Entwurfsmuster:

```
let rec f (n : Nat) : t =                    Leibniz Entwurfsmuster:
  if n = 0 then ...                          Rekursionsbasis
           else ... n % 2 ... f (n ÷ 2) ...  Rekursionsschritt
```

Struktur Entwurfsmuster für den Typ *Leibniz*:

```
let rec f (n : Leibniz) : t =    Struktur Entwurfsmuster:
  match n with
    | Null    → ...              Rekursionsbasis
    | Odd  n' → ... f n' ...     Rekursionsschritt
    | Even n' → ... f n' ...     Rekursionsschritt
```

22. Dualsystem

Der Typ *Leibniz* implementiert übrigens das Binär- oder *Dualsystem*. Der Zusammenhang wird deutlich, wenn wir den Typ etwas umschreiben.

$$
\begin{aligned}
& \mathit{Leibniz} \\
\cong \quad & \{ \text{ Definition von } \mathit{Leibniz} \ \} \\
& 1 + \mathit{Leibniz} + \mathit{Leibniz} \\
\cong \quad & \{ \ 1 \text{ ist das neutrale Element von } `\times' \ \} \\
& 1 + (1 \times \mathit{Leibniz}) + (1 \times \mathit{Leibniz}) \\
\cong \quad & \{ \text{ Distributivgesetz } \} \\
& 1 + (1 + 1) \times \mathit{Leibniz}
\end{aligned}
$$

Damit haben wir eine alternative Definition von *Leibniz*:

$$
\begin{aligned}
&\textbf{\textit{type}}\ \mathit{Bit} \ = \ |\ \mathit{One} \ |\ \mathit{Two} \\
&\textbf{\textit{type}}\ \mathit{Bits} = \ |\ \mathit{Nil} \ |\ \mathit{Cons}\ \textbf{\textit{of}}\ \mathit{Bit} * \mathit{Bits}
\end{aligned}
$$

23. Motivation

- Die Zweige einer Fallunterscheidung binden Bezeichner:

  $\mathit{Female}\ f \to \ldots f.\mathit{name} \ldots$

- Wie bei Wertedefinitionen können wir anstelle eines einzelnen Bezeichners auch ein Muster zulassen.

  $\mathit{Female}\ \{\mathit{name} = n\} \to \ldots n \ldots$

- Wir können auch die gesamte linke Seite als Muster lesen: $\mathit{Female}\ p$ ist ein Muster, auf das ein Wert der Form $\mathit{Female}\ \nu$ passt, sofern ν auf p passt.
- Der Musterabgleich kann auch fehlschlagen: der Wert $\mathit{Male}\ \nu$ passt nicht auf das Muster $\mathit{Female}\ x$.
- Man sagt auch, das Muster $\mathit{Female}\ x$ ist *widerlegbar*.
- Die Fallunterscheidung lässt sich als sukzessives Durchprobieren von Mustern deuten.

23. Abstrakte Syntax

Wir erweitern die Syntax von Fallunterscheidungen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ ***match*** e ***with*** m	erweiterte Fallunterscheidung

Der Rumpf einer Fallunterscheidung ist eine Folge von sogenannten *Regeln* der Form $p \to e$.

$m ::= p \to e$	Regel
$\mid \; m_1 \; ! \; m_2$	Sequenz von Regeln

23. Abstrakte Syntax

Muster werden um disjunkte Muster und Konstruktoranwendungen erweitert.

$p ::= \cdots$	*Muster:*
$\quad \mid \; p_1 \mid p_2$	disjunktives Muster
$\quad \mid \; C\ p$	Konstruktoranwendung

☞ Statische Semantik, siehe Skript/zur Übung.

23. Dynamische Semantik

Beispiele: Wir betrachten die Fallunterscheidung ***match*** e ***with*** $p \to \ldots$ und nehmen an, dass e zu ν auswertet.

p	ν	Umgebung
Nil	Nil	$\varnothing$
Nil	$\mathit{Cons}\ (1, \mathit{Nil})$	$\lightning$
$\mathit{Cons}\ (a, x)$	Nil	$\lightning$
$\mathit{Cons}\ (a, x)$	$\mathit{Cons}\ (1, \mathit{Nil})$	$\{a \mapsto 1, x \mapsto \mathit{Nil}\}$
$\mathit{Cons}\ (a, \mathit{Cons}\ (b, x))$	Nil	$\lightning$
$\mathit{Cons}\ (a, \mathit{Cons}\ (b, x))$	$\mathit{Cons}\ (1, \mathit{Nil})$	$\lightning$
$\mathit{Cons}\ (a, \mathit{Cons}\ (b, x))$	$\mathit{Cons}\ (1, \mathit{Cons}\ (2, \mathit{Nil}))$	$\{a \mapsto 1, b \mapsto 2, x \mapsto \mathit{Nil}\}$

☞ Der Blitz signalisiert, dass das Muster *nicht* auf den Wert passt.

☞ Dynamische Semantik, siehe Skript/zur Übung.

23. Vertiefung

Die Addition lässt sich mit Hilfe geschachtelter Muster sehr viel natürlicher formulieren.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{add}'\ (m : \mathit{Leibniz}, n : \mathit{Leibniz}) : \mathit{Leibniz} = \\
\quad \textbf{\textit{match}}\ (m, n)\ \textbf{\textit{with}} \\
\qquad \mid (\mathit{Null}, \quad k \quad) \\
\qquad \mid (k, \quad \mathit{Null} \quad) \to k \\
\qquad \mid (\mathit{Odd}\ \ m', \mathit{Odd}\ \ n') \to \mathit{Even}\ (\mathit{add}'\ (m', n')) \\
\qquad \mid (\mathit{Odd}\ \ m', \mathit{Even}\ n') \\
\qquad \mid (\mathit{Even}\ m', \mathit{Odd}\ \ n') \to \mathit{Odd}\ \ (\mathit{succ}\ (\mathit{add}'\ (m', n'))) \\
\qquad \mid (\mathit{Even}\ m', \mathit{Even}\ n') \to \mathit{Even}\ (\mathit{succ}\ (\mathit{add}'\ (m', n')))
\end{array}
$$

☞ Wir unterscheiden vier Fälle: ein Argument ist Null, beide Argumente sind ungerade, ein Argument ist ungerade, beide sind gerade.

24. Motivation

Wir haben bisher zwei verschiedene Listentypen eingeführt:

- Listen von natürlichen Zahlen *Nats* und
- Listen von Binärziffern *Bits*.

Viele andere Listentypen sind denkbar:

- Listen von Personen,
- Listen von Adressen,
- Listen von Listen von natürlichen Zahlen um 2-dimensionale Tabellen zu repräsentieren,
- usw.

☞ Für jeden Elementtyp einen neuen Listentyp einzuführen ist *mühsam* und *unökonomisch*. Werden im gleichen Kontext zwei verschiedene Listentypen benötigt, müssen wir uns zudem unterschiedliche Namen für den Typ und insbesondere für die Datenkonstruktoren ausdenken.

Mon dieu, isch verschtehe die Aufregung nischt. Für Lischten von Zahlen definiere isch:

```
type natslink = ^nats;
     nats     = record
                  elem : nat;
                  next : natslink
                end;
```

Für Lischten von Personen definiere isch:

```
type personslink = ^persons;
     persons     = record
                     elem : person;
                     next : personslink
                   end;
```

Für Lischten von Adressen definiere isch ...

Ja, ja, schon gut Blaise. Wir befinden uns mittlerweile im 21. Jahrhundert.

24. Motivation

Idee: wir abstrahieren von dem Elementtyp und machen ihn zum Parameter der Typdefinition.

$$\textbf{\textit{type}}\ \mathit{List}\ \langle {}'a \rangle = \ |\ \mathit{Nil}\ |\ \mathit{Cons}\ \textbf{\textit{of}}\ {}'a * \mathit{List}\ \langle {}'a \rangle$$

☞ *Typvariablen* fangen mit einem Apostroph an, um sie von Record- und Variantentypen unterscheiden zu können.

Wenden wir *List* auf einen konkreten Typ an, so erhalten wir einen speziellen Listentyp:

- Listen von natürlichen Zahlen: $\mathit{List}\ \langle \mathit{Nat} \rangle$,
- Listen von Binärziffern: $\mathit{List}\ \langle \mathit{Bit} \rangle$,
- Listen von Personen: $\mathit{List}\ \langle \mathit{Person} \rangle$,
- Listen von Adressen: $\mathit{List}\ \langle \mathit{Address} \rangle$,
- Listen von Listen von natürlichen Zahlen: $\mathit{List}\ \langle \mathit{List}\ \langle \mathit{Nat} \rangle \rangle$,
- usw.

☞ *Typparameter* werden in spitze Klammern gesetzt, um sie von Werteparametern auf den ersten Blick unterscheiden zu können.

24. Motivation

- Listen sind ein typisches Beispiel für Behälter (engl. container): eine Liste enthält Daten.
- Der Typ $\mathit{List}\,\langle t \rangle$ ist ein Containertyp.
- Da *List* einen Typparameter hat, entspricht *List* im Prinzip einer Funktion auf Typen.
- *Zur Erinnerung:* Listen sind eine einfache *Datenstruktur*. Eine Datenstruktur verwaltet Daten und unterstützt Zugriff und Manipulation dieser Daten.
- Listen ordnen Daten linear an.

24. Abstrakte Syntax

Wir erweitern Definitionen um parametrisierte Recordtyp- und Variantentypdefinitionen.

$d ::= \cdots$		*Deklarationen:*
$\mid$	$\textbf{\textit{type}}\ T\ \langle {}'a \rangle = \{\ell_1 : t_1; \ell_2 : t_2\}$	parametrisierter Recordtyp
$\mid$	$\textbf{\textit{type}}\ T\ \langle {}'a \rangle = \mid C_1\ \textbf{\textit{of}}\ t_1 \mid C_2\ \textbf{\textit{of}}\ t_2$	parametrisierter Variantentyp

24. Statische Semantik

Für den parametrisierten Variantentyp

$$\textbf{\textit{type}}\ T\ \langle {}'a \rangle = \ \mid C_1\ \textbf{\textit{of}}\ t_1 \mid C_2\ \textbf{\textit{of}}\ t_2$$

muss zum Beispiel die Typregel für die Konstruktoranwendung wie folgt angepasst werden.

$$\frac{\Sigma \vdash e : t_i\{{}'a \mapsto t\}}{\Sigma \vdash C_i\ e : T\ \langle t \rangle}$$

☞ In t_i wird die Typvariable ${}'a$ durch den Typ t ersetzt. Notation der *Typsubstitution*: $t_i\{{}'a \mapsto t\}$. *Analogie:* ${}'a$ ist der formale Typparameter, t ist der aktuelle Typparameter.

Beispiel:

$${}'a * List\ \langle {}'a \rangle \{{}'a \mapsto Nat\} = Nat * List\ \langle Nat \rangle$$

24. Statische Semantik

type $List\ \langle 'a \rangle = \ |\ Nil \ |\ Cons$ ***of*** $'a * List\ \langle 'a \rangle$

Beispiele: t ist ein beliebiger Typ.

Ausdruck	Typ
Nil	$List\ \langle t \rangle$
$Cons\ (4711, Nil)$	$List\ \langle Nat \rangle$
$Cons\ (false, Nil)$	$List\ \langle Bool \rangle$
$Cons\ (Nil, Nil)$	$List\ \langle List\ \langle t \rangle \rangle$
$Cons\ (4711, Cons\ (false, Nil))$	nicht wohlgetypt
$Cons\ (Nil, Cons\ (Nil, Nil))$	$List\ \langle List\ \langle t \rangle \rangle$
$Cons\ (Cons\ (Nil, Nil), Nil)$	$List\ \langle List\ \langle List\ \langle t \rangle \rangle \rangle$

☞ Nil besitzt unendlich viele Typen.

24. Dynamische Semantik

Es ändert sich nichts.