

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.

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

351

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
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

352

22. Natürliche Zahlen — Peano

Wir zählen

```
Zero  
Succ Zero  
Succ (Succ Zero)  
Succ (Succ (Succ Zero))  
Succ (Succ (Succ (Succ Zero)))  
...
```

Die Zahlendarstellung unserer Urahren: für jedes erlegte Bison ein Strich.

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

353

22. Addition — Peano

Die Addition lässt sich auf die Nachfolgerfunktion zurückführen.

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

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

354

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))
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

355

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) ...
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

356

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)
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

357

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
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

358

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 $n \geq 2$ 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.



IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

359

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
```

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

360

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))
...
```

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

361

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' ...
```

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

362

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')
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

363

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) ...
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

364

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') ...
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

365

22. Addition — Leibniz

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

```
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') ...
```

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Motivation

Vertiefung

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

366

22. Addition — Leibniz

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

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

```
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') ...
```

367

22. Addition — Leibniz

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

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

368

22. Addition — Leibniz

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

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

```
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')))
```

369

22. Leibniz Entwurfsmuster — da capo

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Motivation
Vertiefung
Widerlegbare Muster
Parametrisierte Typen
Polymorphie
Arrays

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

Leibniz Entwurfsmuster:

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

Leibniz Entwurfsmuster:
Rekursionsbasis
Rekursionsschritt

Struktur Entwurfsmuster für den Typ *Leibniz*:

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

Struktur Entwurfsmuster:
Rekursionsbasis
Rekursionsschritt
Rekursionsschritt

370

22. Dualsystem

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

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

Damit haben wir eine alternative Definition von *Leibniz*:

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

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare Muster

Records

Varianten

Rekursive Varianten

Motivation

Vertiefung

Widerlegbare Muster

Parametrisierte Typen

Polymorphie

Arrays

371

23. Motivation

- ▶ Die Zweige einer Fallunterscheidung binden Bezeichner:

$$Female\ f \rightarrow \dots f.name \dots$$

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

$$Female\ \{ name = n \} \rightarrow \dots n \dots$$

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

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare Muster

Records

Varianten

Rekursive Varianten

Motivation

Abstrakte Syntax

Dynamische Semantik

Vertiefung

Parametrisierte Typen

Polymorphie

Arrays

372

23. Abstrakte Syntax

Wir erweitern die Syntax von Fallunterscheidungen.

$$e ::= \dots \quad \text{Ausdrücke:}$$

$$| \text{ match } e \text{ with } m \quad \text{erweiterte Fallunterscheidung}$$

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

$$m ::= p \rightarrow e \quad \text{Regel}$$

$$| m_1 \mid m_2 \quad \text{Sequenz von Regeln}$$

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare Muster

Records

Varianten

Rekursive Varianten

Widerlegbare Muster

Motivation

Abstrakte Syntax

Dynamische Semantik

Vertiefung

Parametrisierte Typen

Polymorphie

Arrays

373


23. Abstrakte Syntax

Muster werden um disjunkte Muster und Konstruktoranwendungen erweitert.

$$p ::= \dots \quad \text{Muster:}$$

$$| p_1 \mid p_2 \quad \text{disjunktives Muster}$$

$$| C\ p \quad \text{Konstruktoranwendung}$$

 Statische Semantik, siehe Skript/zur Übung.

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare Muster

Records

Varianten

Rekursive Varianten

Widerlegbare Muster

Motivation

Abstrakte Syntax

Dynamische Semantik

Vertiefung

Parametrisierte Typen

Polymorphie

Arrays

374

23. Dynamische Semantik

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

| p | ν | Umgebung |
|-----------------------|-------------------------|---------------------------|
| Nil | Nil | ∅ |
| Nil | Cons (1, Nil) | ⊥ |
| Cons (a, x) | Nil | ⊥ |
| Cons (a, x) | Cons (1, Nil) | { a ↦ 1, x ↦ Nil } |
| Cons (a, Cons (b, x)) | Nil | ⊥ |
| Cons (a, Cons (b, x)) | Cons (1, Nil) | ⊥ |
| Cons (a, Cons (b, x)) | Cons (1, Cons (2, Nil)) | { a ↦ 1, b ↦ 2, x ↦ Nil } |

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

📖 Dynamische Semantik, siehe Skript/zur Übung.

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare

Muster

Records

Varianten

Rekursive

Varianten

Widerlegbare

Muster

Motivation

Abstrakte Syntax

Dynamische

Semantik

Vertiefung

Parametrisierte

Typen

Polymorphie

Arrays

375

23. Vertiefung

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

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

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

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare

Muster

Records

Varianten

Rekursive

Varianten

Widerlegbare

Muster

Motivation

Abstrakte Syntax

Dynamische

Semantik

Vertiefung

Parametrisierte

Typen

Polymorphie

Arrays

376

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 Datenkonstruktor ausdenken.

IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare

Muster

Records

Varianten

Rekursive

Varianten

Widerlegbare

Muster

Parametrisierte

Typen

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Polymorphie

Arrays

377

Mon dieu, ich verstehe die Aufregung nicht. Für Listen von Zahlen definiere ich:

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



Für Listen von Personen definiere ich:

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

Für Listen von Adressen definiere ich ...

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



IV Datentypen

Ralf Hinze

Tupel

Unwiderlegbare

Muster

Records

Varianten

Rekursive

Varianten

Widerlegbare

Muster

Parametrisierte

Typen

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Polymorphie

Arrays

378

24. Motivation

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

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

☞ *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: $List \langle Nat \rangle$,
- ▶ Listen von Binärziffern: $List \langle Bit \rangle$,
- ▶ Listen von Personen: $List \langle Person \rangle$,
- ▶ Listen von Adressen: $List \langle Address \rangle$,
- ▶ Listen von Listen von natürlichen Zahlen: $List \langle List \langle Nat \rangle \rangle$,
- ▶ usw.

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

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Widerlegbare Muster
Parametrisierte Typen
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Polymorphie
Arrays

379

24. Motivation

- ▶ Listen sind ein typisches Beispiel für Behälter (engl. container): eine Liste enthält Daten.
- ▶ Der Typ $List \langle t \rangle$ 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.

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Widerlegbare Muster
Parametrisierte Typen
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Polymorphie
Arrays

380

24. Abstrakte Syntax

Wir erweitern Definitionen um parametrisierte Recordtyp- und Variantentypdefinitionen.

$d ::= \dots$

Deklarationen:

- | **type** $T \langle 'a \rangle = \{ \ell_1 : t_1 ; \ell_2 : t_2 \}$ parametrisierter Recordtyp
- | **type** $T \langle 'a \rangle = | C_1 \text{ of } t_1 | C_2 \text{ of } t_2$ parametrisierter Variantentyp

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Widerlegbare Muster
Parametrisierte Typen
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Polymorphie
Arrays

381

24. Statische Semantik

Für den parametrisierten Variantentyp

type $T \langle 'a \rangle = | C_1 \text{ of } t_1 | C_2 \text{ of } t_2$

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

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

☞ 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:

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

IV Datentypen

Ralf Hinze

Tupel
Unwiderlegbare Muster
Records
Varianten
Rekursive Varianten
Widerlegbare Muster
Parametrisierte Typen
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Polymorphie
Arrays

382

24. Statische Semantik

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

Beispiele: t ist ein beliebiger Typ.

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

☞ Nil besitzt unendlich viele Typen.

24. Dynamische Semantik

Es ändert sich nichts.