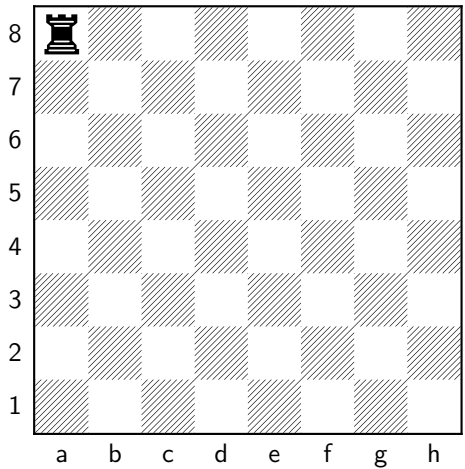
25. Knobelaufgabe #11

- Bewege den Turm von a8 nach h1, so dass jedes Feld *genau einmal* besucht wird.



- Ein Turm darf horizontal und vertikal, aber nicht diagonal ziehen.

25. Motivation

Parametrisierte Typen erhöhen — wie auch Funktionen — die Wiederverwendbarkeit von Programmen („one size fits all“).

Funktionen auf parametrisierten Typen halten mit dieser Entwicklung nicht Schritt. *Beispiel:* Länge einer Liste.

$$\textbf{\textit{let rec}}\ \mathit{length}\ (\mathit{list} : \mathit{List}\ \langle \mathit{Nat} \rangle) : \mathit{Nat} = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \mid \mathit{Nil} \to 0 \mid \mathit{Cons}\ (\_, \mathit{xs}) \to 1 + \mathit{length}\ \mathit{xs}$$

☞ Der Typ des Parameters, *List* ⟨*Nat*⟩, schränkt die Anwendung von *length* auf Listen von *natürlichen Zahlen* ein. Um die Länge einer Liste von *Personen* zu bestimmen, müssen wir eine zweite Funktion programmieren.

$$\textbf{\textit{let rec}}\ \mathit{length}\ (\mathit{list} : \mathit{List}\ \langle \mathit{Person} \rangle) : \mathit{Nat} = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \mid \mathit{Nil} \to 0 \mid \mathit{Cons}\ (\_, \mathit{xs}) \to 1 + \mathit{length}\ \mathit{xs}$$

☞ Die Definition ist baugleich zur ersten, nur die Typangabe hat sich geändert. Für das Ausrechnen der Listenlänge spielt der Typ der Elemente aber gar keine Rolle.

25. Motivation

☞ Für jeden Elementtyp einen neue *length* Funktion zu programmieren, ist *mühsam* und *unökonomisch*. Werden im gleichen Kontext Längenfunktionen für zwei verschiedene Elementtypen benötigt, müssen wir uns zudem unterschiedliche Namen für die Funktionen ausdenken.

☞ Das Problem ist ähnlich dem, das die Einführung parametrisierter Typen motivierte. Ähnliches Problem, ähnliche Lösung?

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

```
function lengthnats (p : natslink) : integer;
begin
   if p = nil then
      lengthnats := 0
   else
      lengthnats := lengthnats (p^.next) + 1
end;
```

Für die Länge von Lischten von Personen definiere isch:

```
function lengthpersons (p : personslink) : integer;
begin
   if p = nil then
      lengthpersons := 0
   else
      lengthpersons := lengthpersons (p^.next) + 1
end;
```

Für die Länge von Lischten von Adressen definiere isch ...

Ich glaub, ich hab ein déjà-vu ...

25. Motivation

Ähnliches Problem, ähnliche Lösung?

Ja! So wie wir Typdefinitionen mit einem Typ parametrisieren, so können wir auch Funktionsdefinitionen mit einem Typ parametrisieren.

$$\textbf{\textit{let rec}}\ \mathit{length}\,\langle'a\rangle(\mathit{list} : \mathit{List}\,\langle'a\rangle) : \mathit{Nat} =$$
$$\quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \mid \mathit{Nil} \to 0 \mid \mathit{Cons}\ (\_, xs) \to 1 + \mathit{length}\,\langle'a\rangle xs$$

☞ Die Funktion *length* hat jetzt zwei Parameter: einen Typparameter ($'a$) und einen Werteparameter (*list*). Der Typparameter wird verwendet, um den Typ des Werteparameters festzulegen.

Wird die Funktion *length* aufgerufen, müssen für die formalen Parameter entsprechend aktuelle Parameter angegeben werden:

- ein Typ und
- eine Liste mit Elementen des angegebenen Typs.

25. Motivation — polymorphe Funktionen

Wenden wir *length* auf einen konkreten Typ an, erhalten wir spezielle Längenfunktionen — Funktionen, die wir bisher mühsam per Hand programmieren mussten.

- *length* ⟨*Nat*⟩ für Listen von natürlichen Zahlen,
- *length* ⟨*Bit*⟩ für Listen von Binärziffern,
- *length* ⟨*Person*⟩ für Listen von Personen,
- *length* ⟨*Address*⟩ für Listen von Adressen,
- *length* ⟨*List* ⟨*Nat*⟩⟩ für Listen von Listen von natürlichen Zahlen,
- usw.

☞ Funktionen, die mit einem oder mehreren Typen parametrisiert sind, heißen auch *polymorphe* Funktionen nach dem griechischen Wort $\pi o \lambda \upsilon \mu o \rho \phi \iota \alpha$ für Vielgestaltigkeit.

25. Motivation — polymorphe Funktionen

Labels parametrisierter Recordtypen und Konstruktoren parametrisierter Variantentypen sind im Prinzip polymorphe Funktionen bzw. Werte:

- *Nil* hat den Typ $\mathit{List}\,\langle'a\rangle$;
- *Cons* hat den Typ $'a * \mathit{List}\,\langle'a\rangle \to \mathit{List}\,\langle'a\rangle$.

25. Motivation — Peano Entwurfsmuster, programmiert

Blicken wir zurück, erkennen wir, dass viele Funktionen allgemeiner sind, als die Typangaben es vermuten lassen. *Zum Beispiel:*

```
let peano-pattern ⟨'soln⟩(zero : 'soln,
                          succ : 'soln → 'soln) : Nat → 'soln =
  let rec f (n : Nat) : 'soln =
        if n = 0 then zero
                 else succ (f (n ∸ 1))
  in f
```

☞ Jetzt können wir in Mini–F# ausdrücken, dass die Funktion für jeden beliebigen Typ von Lösungen funktioniert (siehe frühere Diskussion).

Super, damit kann ich endlich meine Implementierung der Fakultät ausprobieren.

Mini⟩ *peano-pattern* ⟨*Nat* ∗ *Nat*⟩
((0, 1), ***fun*** (*n*, *s*) → (*n* + 1, *s* ∗ (*n* + 1))) 10
(10, 3628800)

Pass mal auf, Lisa, ich hab mich mittlerweile mit den anonymen Funktionen angefreundet. Jetzt kommt der Hammer: man kommt auch ohne Paare aus!

Mini⟩ *peano-pattern* ⟨*Nat* → *Nat*⟩
(***fun*** *n* → 1, ***fun*** *s* → ***fun*** *n* → *s* (*n* ∸ 1) ∗ *n*) 10 10
3628800

Clever, clever, Harry. Deine Funktion ist sogar noch allgemeiner: wenn Du sie mit *m* und *n* aufrufst ($m \leqslant n$), berechnet sie $n\,! \div (n - m)\,!$.

25. Motivation — Sortieren durch Einfügen

Auch die Funktion *sort-by* lässt sich verallgemeinern.

Sortieren durch Einfügen

```
let sort-by ⟨'a⟩(less-equal : 'a * 'a → Bool) : List ⟨'a⟩ → List ⟨'a⟩ =
  let rec insert (elem : 'a, list : List ⟨'a⟩) : List ⟨'a⟩ =
    match list with
      | Nil          → Cons (elem, Nil)
      | Cons (x, xs) → if less-equal (elem, x)
                       then Cons (elem, list)
                       else Cons (x, insert (elem, xs))
  let rec sort (list : List ⟨'a⟩) : List ⟨'a⟩ =
    match list with
      | Nil          → Nil
      | Cons (x, xs) → insert (x, sort xs)
  in sort
```

25. Motivation — Sortieren durch Einfügen

Jetzt können wir beliebige (!) Listen sortieren.

- Listen von natürlichen Zahlen:

$$\mathit{sort\text{-}by}\,\langle \mathit{Nat}\rangle(\textbf{\textit{fun}}\ (m,n) \to m \leqslant n)$$
$$\mathit{sort\text{-}by}\,\langle \mathit{Nat}\rangle(\textbf{\textit{fun}}\ (m,n) \to m \geqslant n)$$

- Listen von Binärziffern:

$$\begin{aligned}\mathit{sort\text{-}by}\,\langle \mathit{Bit}\rangle(\textbf{\textit{fun}}\ (b,c) \to \textbf{\textit{match}}\ (b,c)\ \textbf{\textit{with}}\\ \mid (\mathit{One}, \_\ \ ) \to \mathit{true}\\ \mid (\mathit{Two}, \mathit{One}) \to \mathit{false}\\ \mid (\mathit{Two}, \mathit{Two}) \to \mathit{true}\end{aligned}$$

- Listen von Personen:

$$\mathit{sort\text{-}by}\,\langle \mathit{Person}\rangle(\textbf{\textit{fun}}\ (\mathit{ann}, \mathit{bob}) \to \mathit{ann.surname} \leqslant \mathit{bob.surname})$$

- Listen von Listen von natürlichen Zahlen:

$$\mathit{sort\text{-}by}\,\langle \mathit{List}\,\langle \mathit{Nat}\rangle\rangle(\textbf{\textit{fun}}\ (\mathit{ns}_1, \mathit{ns}_2) \to \mathit{length}\,\langle \mathit{Nat}\rangle\,\mathit{ns}_1 \leqslant \mathit{length}\,\langle \mathit{Nat}\rangle\,\mathit{ns}_2)$$

25. Pragmatik

Wir sehen davon ab, Syntax und Semantik polymorpher Funktionen formal zu definieren.

Stattdessen geben wir uns pragmatisch und erlauben *Typparameter auszulassen*: sowohl bei der Definition (keine große Sache)

$$\textbf{\textit{let rec}}\ \mathit{length}\ (\mathit{list} : \mathit{List}\ \langle' a\rangle) : \mathit{Nat} = \ldots$$
$$\textbf{\textit{let}}\ \mathit{sort\text{-}by}\ (\mathit{less\text{-}equal} : {'a} * {'a} \to \mathit{Bool}) : \mathit{List}\ \langle' a\rangle \to \mathit{List}\ \langle' a\rangle = \ldots$$

als auch bei der Anwendung (eine erhebliche Schreiberleichterung).

$$\mathit{sort\text{-}by}\ (\textbf{\textit{fun}}\ (m, n) \to m \leqslant n)$$
$$\mathit{sort\text{-}by}\ (\textbf{\textit{fun}}\ (ns_1, ns_2) \to \mathit{length}\ ns_1 \leqslant \mathit{length}\ ns_2)$$

25. Vertiefung

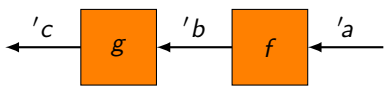
Polymorphen Funktionen ist gemeinsam, dass sie die Elemente eines polymorphen Typs nicht analysieren oder inspizieren (oder gar generieren), sondern nur transportieren (oder womöglich ignorieren).

Mini⟩ ***let*** *id x = x*
val *id : 'a → 'a*

Mini⟩ ***let*** *constant a b = a*
val *constant : 'a → 'b → 'a*

Mini⟩ ***let*** *compose g f x = g (f x)*
val *compose : ('b → 'c) → ('a → 'b) → ('a → 'c)*

☞ *compose* ist ebenfalls eine Funktion höherer Ordnung: sie definiert die Komposition (Hintereinanderschaltung) zweier Funktionen.



25. Vertiefung — Listenzugriff

Aufgabe: Bestimmung des n-ten Elements einer Liste.

let rec *nth* $(list : List\ \langle 'a \rangle, n : Nat) : 'a =$

☞ Was machen wir, wenn die Liste weniger als n Elemente hat?

Idee: die beiden möglichen Resultate des Zugriffs, erfolglos und erfolgreich, mit einem Datentyp darstellen.

type *Option* $\langle 'a \rangle =$ | *None* | *Some* ***of*** $'a$

Mit diesem neuen Typ können wir die Signatur von *nth* verfeinern.
Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec nth (list : List ⟨'a⟩, n : Nat) : Option ⟨'a⟩ =
  match list with
  | Nil          → ...
  | Cons (x, xs) → ... nth (xs, ...) ...
```

25. Vertiefung — Listenzugriff

- *Rekursionsbasis*: der Index ist zu groß.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : \mathit{Option}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad\ |\ \mathit{Nil} \qquad\quad \to \mathit{None} \\ \quad\ |\ \mathit{Cons}\ (x, xs) \to \ \ldots \mathit{nth}\ (xs, \ldots) \ldots\end{array}$$

- *Rekursionsschritt*: zusätzliche Fallunterscheidung über n.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{nth}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle, n : \mathit{Nat}) : \mathit{Option}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad\ |\ \mathit{Nil} \qquad\quad \to \mathit{None} \\ \quad\ |\ \mathit{Cons}\ (x, xs) \to \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ \mathit{Some}\ x \\ \qquad\qquad\qquad\qquad\quad\ \ \textbf{\textit{else}}\ \ \mathit{nth}\ (xs, n \dot{-} 1)\end{array}$$

☞ Die Funktion *nth* sollte nicht *missbraucht* werden, um über eine Liste zu iterieren!

25. Über den Tellerrand: F#

Listen sind in F# vordefiniert. Allerdings unterscheidet sich die Notation etwas:

- T *list* statt *List* $\langle T \rangle$ (Postfix- statt Präfixnotation),
- [] statt *Nil*, und
- $x :: xs$ statt *Cons* (x, xs) (Infix- statt Präfixnotation).

☞ $[x_1; x_2; x_3]$ kürzt $x_1 :: (x_2 :: (x_3 :: []))$ ab.

Beispiel: Bestimmung des n-ten Elements einer Liste.

```
let rec nth (list : 'a list, n : int) : 'a option =
  match list with
  | []      → None
  | x :: xs → if n = 0 then Some x
                       else nth (xs, n - 1)
```

25. Über den Tellerrand: F#

F# erlaubt, die Typen von Funktionsparametern und -ergebnissen auszulassen — die fehlenden Informationen werden automatisch inferiert (Typinferenz).

Beliebige Grade an Flexibilität:

let rec *length* (*xs* : *'a list*) : *int* = ***match*** *xs* ***with*** ...

let rec *length* (*xs* : *'a list*) = ***match*** *xs* ***with*** ...

let rec *length* (*xs*) = ***match*** *xs* ***with*** ...

let rec *length* = ***function*** ...

☞ ***function*** *m* ist eine beliebte Abkürzung für den Ausdruck ***fun*** $x \rightarrow$ ***match*** x ***with*** m. Beliebt, weil die Einführung eines Parameters vermieden wird (neue Namen zu erfinden ist schwer).

25. Über den Tellerrand: F#

Beispiel: enthält eine Liste ein gegebenes Element?

```
let rec contains key = function
    | []      → false
    | x :: xs → key = x || contains key xs
```

Beispiel: Konkatenation zweier Listen.

```
let rec append list y =
  match list with
    | []      → y
    | x :: xs → x :: append xs y
```

☞ *append x y* ist als Infixoperator vordefiniert: $x \mathbin{@} y$, z.B. $[4;7] \mathbin{@} [1;1] = [4;7;1;1]$.
(Was ist der Typ von *append* aka @? Und von *contains*?)

26. Motivation

Wir haben verschiedene Möglichkeiten kennengelernt, Daten zu aggregieren.

- Tupel und Records:
  - *feste* Anzahl von Daten *verschiedenen* Typs,
  - Zugriff in konstanter Zeit.
- Listen:
  - *beliebige* Anzahl von Daten des *gleichen* Typs,
  - Zugriff in linearer Zeit.
- Fehlt:
  - *beliebige* Anzahl von Daten des *gleichen* Typs,
  - Zugriff in *konstanter* Zeit.

26. Motivation

☞ Diese Lücke schließen Arrays (auch: Felder oder Reihungen).

Konstruktion:

$$[|2;3;5;7;11|]$$

ist ein Array der Größe 5. Mit *a.Length* wird die Größe bzw. Länge eines Arrays ermittelt, z.B. $[|2;3;5;7;11|].\mathit{Length} = 5$.

Zugriff: ist e_2 ein Ausdruck, der zu einem Array auswertet, und e_1 ein arithmetischer Ausdruck, dann kann mit

$$e_2.[e_1]$$

auf die entsprechende Komponente zugegriffen werden.
☞ Nummerierung ab 0, z.B. $[|2;3;5;7;11|].[3] = 7$.

26. Motivation

Arrays können nicht nur durch Aufzählung der Elemente, sondern auch mittels einer Bildungsvorschrift erzeugt werden.

$$[\,|\,\textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 0\,..\,9 \to i * i\,|\,]$$

ist das Array der ersten zehn Quadratzahlen.

26. Demo

```
Mini⟩ [| |]
[| |]
Mini⟩ [|2; 3; 5; 7; 11|]
[|2; 3; 5; 7; 11|]
Mini⟩ [|for i in 0 .. 9 → 4711|]
[|4711; 4711; 4711; 4711; 4711; 4711; 4711; 4711; 4711; 4711|]
Mini⟩ [|for i in 0 .. 9 → i|]
[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
Mini⟩ [|for i in 0 .. 9 → i * i|]
[|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
Mini⟩ [|for i in 0 .. 7 → [|for j in 1 .. i → i * j|]|]
[|[| |];
  [|1|];
  [|2; 4|];
  [|3; 6; 9|];
  [|4; 8; 12; 16|];
  [|5; 10; 15; 20; 25|];
  [|6; 12; 18; 24; 30; 36|];
  [|7; 14; 21; 28; 35; 42; 49|]|]
```

26. Arrays versus Funktionen

Arrays sind Funktionen recht ähnlich.

Typ	$t_1 \to t_2$	$\mathit{Array}\,\langle t_2 \rangle \quad (t_1 = \mathit{Int})$
Konstruktion	—	$[\!\mid e_0; \ldots; e_{n-1} \mid\!]$
	$\textbf{\textit{fun}}\ x \to e_3$	$[\!\mid \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 0\,..\,n-1 \to e_3 \mid\!]$
Elimination	$e_2\ e_1$	$e_2.[e_1]$
	—	$e.\mathit{Length}$

☞ Der Definitionsbereich eines Arrays ist stets ein Anfangsstück der natürlichen Zahlen. Haben wir das schon einmal gesehen?

26. Arrays versus Sequenzen

Zur Erinnerung: endliche Abbildungen des Typs $\mathbb{N} \to_{\text{fin}} A$ heißen Sequenzen (Folie 60).

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

☞ Fachjargon: Arrays *internalisieren* Sequenzen. Umgekehrt verwenden wir Sequenzen, um die Semantik von Arrays und Operationen auf Arrays präzise zu beschreiben.

26. Abstrakte Syntax

$e ::= \cdots$	*Arrays:*
$\mid [\mid e_0; \ldots; e_{n-1} \mid]$	Konstruktion durch Aufzählung ($n \in \mathbb{Z}$)
$\mid [\mid \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1 .. e_2 \to e_3 \mid]$	Konstruktion durch Bildungsvorschrift
$\mid e_2.[e_1]$	Subskription
$\mid e.\mathit{Length}$	Größe eines Arrays

☞ Eckige Klammern, [| und |], sind das Markenzeichen der Sprachkonstrukte, die Arrays konstruieren.

☞ Das Konstrukt $[\mid \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1 .. e_2 \to e_3 \mid]$ führt den Bezeichner x neu ein; x ist in e_3 sichtbar.

☞ Der Ausdruck e_1 in $e_2.[e_1]$ heißt *Index*.

26. Statische Semantik

Der Typ eines Arrays ist mit dem Typ der Elemente parametrisiert.

$t ::= \cdots$	*Typen:*
$\quad \mid \mathit{Array}\ \langle t \rangle$	Arraytyp

Typregel:

$$\frac{\Sigma \vdash e_i : t \quad \mid \quad i \in \mathbb{N}_n}{\Sigma \vdash [\mid e_0; \ldots; e_{n-1} \mid] : \mathit{Array}\ \langle t \rangle}$$

☞ Die Notation $\phi_i \mid i \in \mathbb{N}_n$ ist eine kompakte Schreibweise für eine Regel mit den n Voraussetzungen ϕ_0, ..., ϕ_{n-1}.

26. Statische Semantik

Typregel:

$$\frac{\Sigma \vdash e_1 : \mathit{Int} \quad \Sigma \vdash e_2 : \mathit{Int} \quad \Sigma, \{x \mapsto \mathit{Int}\} \vdash e_3 : t}{\Sigma \vdash [|\ \textbf{for}\ x\ \textbf{in}\ e_1 \,..\, e_2 \to e_3\ |] : \mathit{Array}\ \langle t \rangle}$$

$$\frac{\Sigma \vdash e_2 : \mathit{Array}\ \langle t \rangle \qquad \Sigma \vdash e_1 : \mathit{Int}}{\Sigma \vdash e_2.[e_1] : t}$$

$$\frac{\Sigma \vdash e : \mathit{Array}\ \langle t \rangle}{\Sigma \vdash e.\mathit{Length} : \mathit{Int}}$$

☞ ***for*** x ***in*** $e_1 \,..\, e_2$ arbeitet auch mit anderen Zahlentypen, nicht nur mit *Int*.

26. Statische Semantik — Extremfälle

- $n = 0$: Felder dürfen leer sein.
  - $[|\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 1\,..\,0 \to x\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ l\,..\,r \to e\,|]$ mit $l > r$,
  - dann gilt: $e.\mathit{Length} = 0$.
  - Das leere Feld $[\![]\!]$ hat den Typ $\mathit{Array}\,\langle t \rangle$ für einen beliebigen Typ t.
- $n = 1$: Felder können genau ein Element enthalten.
  - $[|\,4711\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ 0\,..\,0 \to 4711\,|]$,
  - $[|\,\textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ l\,..\,r \to e\,|]$ mit $l = r$,
  - dann gilt: $e.\mathit{Length} = 1$.
- $n = 2$: ...

26. Dynamische Semantik

Der Wert eines Arrayausdrucks ist eine Sequenz, eine endliche Abbildung $\mathbb{N} \to_{\text{fin}} \text{Val}$; jedem Index wird die entsprechende Komponente zugeordnet.

Beispiele:

- $[|\,|]$ wertet zu ϵ aus;
- $[|4711|]$ wertet zu $\{0 \mapsto 4711\}$ aus;
- $[|2;3;5;7;11|]$ wertet zu $\{0 \mapsto 2, 1 \mapsto 3, 2 \mapsto 5, 3 \mapsto 7, 4 \mapsto 11\}$ aus.

26. Dynamische Semantik

$s \in \mathsf{Val}^*$

$\nu ::= \cdots$	*Werte:*
$\mid\ s$	Array (-wert)

Auswertungsregeln:

$$\frac{\delta \vdash e_i \Downarrow \nu_i \quad \mid \quad i \in \mathbb{N}_n}{\delta \vdash [\!| e_0, \ldots, e_{n-1} |\!] \Downarrow \{ i \mapsto \nu_i \mid i \in \mathbb{N}_n \}}$$

$$\frac{\begin{array}{ll}\delta \vdash e_1 \Downarrow l & \\ \delta \vdash e_2 \Downarrow r & \delta, \{x \mapsto i\} \vdash e_3 \Downarrow \nu_i \quad \mid \quad i \in \{l \,..\, r\}\end{array}}{\delta \vdash [\!| \textbf{\textit{for}}\ x\ \textbf{\textit{in}}\ e_1 \,..\, e_2 \to e_3 |\!] \Downarrow \{ i - l \mapsto \nu_i \mid i \in \{l \,..\, r\} \}}$$

☞ Es werden zunächst die Arraygrenzen ausgerechnet, dann wird der Bezeichner x nacheinander an die Werte l, ..., r gebunden und bezüglich jeder Bindung wird der Rumpf e_3 ausgerechnet.

26. Dynamische Semantik

Auswertungsregeln:

$$\frac{\delta \vdash e_2 \Downarrow s \qquad \delta \vdash e_1 \Downarrow i}{\delta \vdash e_2.[e_1] \Downarrow s(i)} \quad i < \mathit{len}\ s$$

$$\frac{\delta \vdash e \Downarrow s}{\delta \vdash e.\mathit{Length} \Downarrow \mathit{len}\ s}$$

☞ Ein Array ist eine endliche Abbildung; Subskription ist entsprechend Funktionsapplikation.
☞ Die Subskription $e_2.[e_1]$ ist nur definiert, wenn der Index e_1 im Definitionsbereich der endlichen Abbildung liegt.

26. Vertiefung — Spiegelung

Arraytransformationen lassen sich oftmals mit dem ***for***-Konstrukt (Konstruktion durch Bildungsvorschrift) programmieren.

Beispiel: Spiegelung eines Arrays.

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{reverse}\ \langle'a\rangle(a : \mathit{Array}\ \langle'a\rangle) = \\ \quad \textbf{\textit{let}}\ n = a.\mathit{Length}\ \textbf{\textit{in}} \\ \quad\quad [|\ \textbf{\textit{for}}\ i\ \textbf{\textit{in}}\ 1\,..\,n \to a.[n-i]\ |];\end{array}$$

☞ *reverse* ist eine polymorphe Funktion.

Mini⟩ *reverse*[| ***for*** *i* ***in*** $0\,..\,9 \to i * i$ |]
[|81; 64; 49; 36; 25; 16; 9; 4; 1; 0|]

26. Vertiefung — Konkatenation

Beispiel: Konkatenation zweier Arrays.

$$\textbf{let rec } \mathit{append}\ \langle'a\rangle(a : \mathit{Array}\ \langle'a\rangle, b : \mathit{Array}\ \langle'a\rangle) : \mathit{Array}\ \langle'a\rangle =$$
$$[\!|\ \textbf{for } i \textbf{ in } 0\,..\,a.\mathit{Length} + b.\mathit{Length} - 1 \to$$
$$\quad \textbf{if } i < a.\mathit{Length} \textbf{ then } a.[i]$$
$$\qquad\qquad\qquad \textbf{else } \ b.[i - a.\mathit{Length}]\ |\!]$$

☞ *append* ist eine polymorphe Funktion.

☞ Vergleiche mit der Konkatenation $s_1 \cdot s_2$ von Sequenzen (Folie 61; Arrays internalisieren Sequenzen).

Mini⟩ $\mathit{append}\ ([\!|\ \textbf{for } i \textbf{ in } 0\,..\,9 \to i\ |\!], \mathit{reverse}[\!|\ \textbf{for } i \textbf{ in } 0\,..\,9 \to i\ |\!])$
$[\!|\,0;1;2;3;4;5;6;7;8;9;9;8;7;6;5;4;3;2;1;0\,|\!]$

26. Vertiefung — Entwurfsmuster

Der „Definitionsbereich“ des Arrays a ist durch das Intervall $(0, a.Length - 1)$ gegeben. Somit können wir die Programmierung von arrayverarbeitenden Funktionen so ähnlich angehen, wie die von Funktionen auf Suchintervallen.

Entwurfsmuster für Arrays (à la Peano):

$$
\begin{array}{l}
\textbf{\textit{let}}\ f\ (a : \mathit{Array}\ \langle t_1 \rangle) : t_2 = \\
\quad \textbf{\textit{let rec}}\ g\ (i : \mathit{Int}) : t_2 = \\
\qquad \textbf{\textit{if}}\ i = a.\mathit{Length}\ \textbf{\textit{then}} \ldots \\
\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ldots g\ (i + 1) \ldots \\
\quad \textbf{\textit{in}}\ g\ 0
\end{array}
$$

26. Entwurfsmuster — Summe

Beispiel: Elemente eines Arrays aufaddieren (das Pendant zur Funktion $sum : List\ \langle Nat \rangle \to Nat$).

Das Entwurfsmuster für Arrays gibt vor:

$$
\begin{array}{l}
\textbf{\textit{let}}\ sum\ (a : Array\ \langle Nat \rangle) : Nat = \\
\quad \textbf{\textit{let rec}}\ s\ (i : Int) : Nat = \\
\qquad \textbf{\textit{if}}\ \ i = a.Length\ \textbf{\textit{then}} \ldots \\
\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ \ldots s\ (i + 1) \ldots \\
\quad \textbf{\textit{in}}\ s\ 0
\end{array}
$$

26. Entwurfsmuster — Summe

- *Rekursionsbasis*: $sum[\,|\,] = 0$!

```
let sum (a : Array ⟨Nat⟩) : Nat =
  let rec s (i : Int) : Nat =
    if i = a.Length then 0
                    else ... s (i + 1) ...
  in s 0
```

- *Rekursionsschritt*: Wir addieren $a.[i]$ zur Summe des Restarrays.

```
let sum (a : Array ⟨Nat⟩) : Nat =
  let rec s (i : Int) : Nat =
    if i = a.Length then 0
                    else a.[i] + s (i + 1)
  in s 0
```

26. Entwurfsmuster — Produkt

Beispiel: Elemente eines Arrays miteinander multiplizieren (das Pendant zur Funktion $product : List\,\langle Nat\rangle \to Nat$).

Das Entwurfsmuster für Arrays gibt vor:

$$
\begin{array}{l}
\textbf{\textit{let}}\ product\ (a : Array\,\langle Nat\rangle) : Nat = \\
\quad \textbf{\textit{let rec}}\ p\ (i : Int) : Nat = \\
\qquad \textbf{\textit{if}}\ i = a.Length\ \textbf{\textit{then}} \ldots \\
\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ldots p\ (i+1) \ldots \\
\quad \textbf{\textit{in}}\ p\ 0
\end{array}
$$

26. Entwurfsmuster — Produkt

- *Rekursionsbasis*: $product[\,|\,|] = 1$!

```
let product (a : Array ⟨Nat⟩) : Nat =
  let rec p (i : Int) : Nat =
    if i = a.Length then 1
                    else ... p (i + 1) ...
  in p 0
```

- *Rekursionsschritt*: Wir multiplizieren $a.[i]$ mit dem Produkt des Restarrays.

```
let product (a : Array ⟨Nat⟩) : Nat =
  let rec p (i : Int) : Nat =
    if i = a.Length then 1
                    else a.[i] * p (i + 1)
  in p 0
```

26. Alte Funktionen neu

Eine weitere Implementierung der Fakultät und der Potenzfunktion:

let *factorial* $(n : Nat) : Nat =$
$\quad product[|$ ***for*** i ***in*** $1 \,..\, n \rightarrow i|]$

let *power* $(x : Nat, n : Nat) : Nat =$
$\quad product[|$ ***for*** i ***in*** $1 \,..\, n \rightarrow x|]$

26. Zusammenfassung

Wir haben das Repertoire von Mini–F# beträchtlich erweitert. *Teil III:* Werte, die von Werten abhängen. *Teil IV:* Typen, die von Typen abhängen und Werte, die von Typen abhängen.

---

hängt ab von	Wert	Typ
Wert	Funktion	polymorphe Funktion
Typ	?	parametrisierter Typ

Macht man die Größe von Arrays zum Bestandteil des Typs, dann erhält man ein Beispiel für einen Typ, der von Werten abhängt (sogenannter „dependent type“).

26. Zusammenfassung

Wir haben

- verschiedene Möglichkeiten kennengelernt, Daten zu aggregieren:
  - Tupel und Records,
  - Containertypen wie Listen,
  - Arrays;
- eine Möglichkeit kennengelernt, Daten zu vereinigen:
  - Variantentypen;
- Verschiedene Formen der Abstraktion kennengelernt:
  - parametrisierte Record- und Variantentypen,
  - polymorphe Funktionen;
- gesehen, dass jeder Typ ein Entwurfsmuster mitbringt;
- verschiedene Datenstrukturen kennengelernt:
  - Listen,
  - Arrays;
- und verschiedene Funktionen auf diesen Datenstrukturen programmiert.

26. Paradigmen: funktionale Programmierung

calculare
den Wert bestimmen; rechnen; zählen

functio
Verrichtung; Ausführung

fungi
ausüben