

Teil IV

Datentypen

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

17. Knobelaufgabe #8

Ein *Fleißiger Biber* ist ein Mini-F# Ausdruck, der zu der größten Zahl auswertet — unter allen Ausdrücken der gleichen textuellen Länge.

Länge	Fleißiger Biber / Wert
1	9 / 9
2	99 / 99
⋮	⋮
50	let rec f n:Nat=if n=0 then 9 else n*f(n-1)in f 99 / 8399359389954973741352931497064003044164437143794 3459321733667505695839993906924048047317578540866 4576283281287445013824260666898251776000000000000 0000000000
⋮	⋮

Füllen Sie die fehlenden Einträge (korrigieren Sie ggf. die obigen)! Wann werden die Werte zu groß? Lässt sich die Funktion, die jeder Länge den Wert des Fleißigen Bibers zuordnet, in Mini-F# programmieren?

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

17. Gliederung

- 18 Tupel
- 19 Unwiderlegbare Muster
- 20 Records
- 21 Varianten
- 22 Rekursive Varianten
- 23 Widerlegbare Muster
- 24 Parametrisierte Typen
- 25 Polymorphie
- 26 Arrays

Tupel

Unwiderlegbare
Muster

Records

Varianten

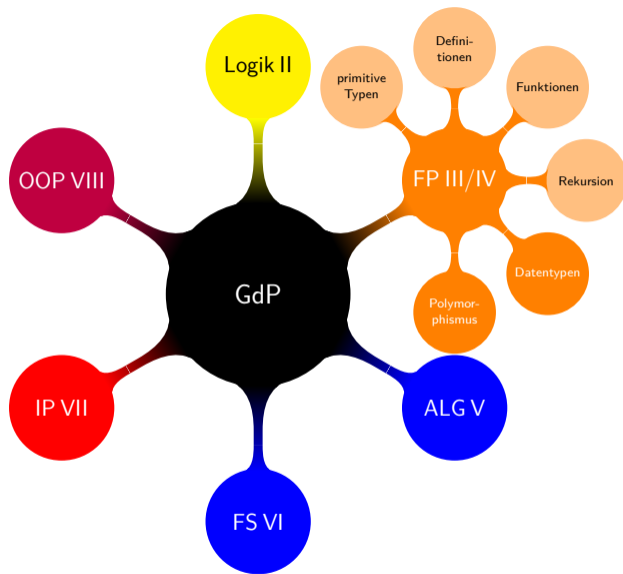
Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays



Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

17. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- ▶ die beiden grundsätzlichen Strukturierungselemente für Daten kennen: Tupel bzw. Records und Varianten,
- ▶ Mini-F# Sprachkonstrukte für Tupel bzw. Records und Varianten kennen und verwenden können,
- ▶ mit der Verwendung von Mustern vertraut sein,
- ▶ rekursive und parametrisierte Typdefinitionen lesen und selbst definieren können,
- ▶ einfache Datenstrukturen wie Listen kennen,
- ▶ das Konzept der Polymorphie verstanden haben,
- ▶ Arrays verwenden können.

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

17. Überblick

- ▶ Informatiker*innen bilden Modelle der Wirklichkeit.
- ▶ Einen wesentlichen Teil dieser Modelle machen Daten aus.
- ▶ *Bisher*: bescheidenes Repertoire an Datentypen:
 - ▶ Boolesche Werte: *Bool*,
 - ▶ natürliche Zahlen: *Nat*,
 - ▶ Funktionen: $t_1 \rightarrow t_2$.
- ▶ Was uns fehlt, sind Möglichkeiten
 - ▶ mehrere Daten zu einem Datum zusammenzufassen: etwa
 - ▶ einen Straßennamen,
 - ▶ eine Postleitzahl *und*
 - ▶ einen Ortsnamenzu einer Adresse.
 - ▶ mehrere alternative Angaben als Einheit zu behandeln: etwa den Familienstand mit den Alternativen
 - ▶ ledig,
 - ▶ verheiratet mit Angabe des Datums der Trauung *oder*
 - ▶ geschieden ebenfalls mit Datumsangabe.

Tupel

Unwiderlegbare
Muster

Records

Varianten

Rekursive
VariantenWiderlegbare
MusterParametrisierte
Typen

Polymorphie

Arrays



Schau mal, Lisa. Ich habe zwei Funktionen programmiert, die von zwei Zahlen die kleinere bzw. die größere bestimmen.

let *minimum* ($a : \text{Nat}, b : \text{Nat}$) = **if** $a \leq b$ **then** a **else** b
let *maximum* ($a : \text{Nat}, b : \text{Nat}$) = **if** $a \leq b$ **then** b **else** a

Ja?



Wenn ich jetzt sowohl die kleinere als auch die größere Zahl brauche, dann muss ich beide Funktionen aufrufen. Aber dann wird der Vergleich $a \leq b$ zweimal durchgeführt. Unnötigerweise.

Du willst die beiden Zahlen also sortieren?



Genau!

Wie wär's, wenn Du einfach ein Paar zurückgibst?



$$\text{let sort2 } (a : \text{Nat}, b : \text{Nat}) : \text{Nat} * \text{Nat} = \\ \text{if } a \leq b \text{ then } (a, b) \text{ else } (b, a)$$

Cool! Darauf hätte ich auch kommen können. Aber wie kriege ich das Paar wieder auseinanderklamüsert?



Stimmt, wir brauchen noch zwei zusätzliche Konstrukte. Wie wär's mit *fst e* und *snd e*, um an die erste bzw. zweite Komponente zu kommen?

Dann müsste ich also schreiben:

$$\text{let } x = \text{sort2 } (\dots, \dots) \text{ in } \dots \text{fst } x \dots \text{snd } x \dots$$


18. Motivation

- ▶ Paare erlauben es, Daten zu aggregieren; zwei verschiedene Daten als Einheit zu behandeln.
- ▶ Die zwei Komponenten eines Paares müssen nicht den gleichen Typ besitzen:
 - ▶ ("Lisa", 9)
 - ▶ (7, **fun** ($i : \text{Nat}$) $\rightarrow i$)
- ▶ Eigentlich sind Paare kein neues Konzept; die Funktion *minimum* nimmt ein Paar als *Argument*.
 - ▶ *Bisherige Sichtweise*: *minimum* hat zwei Argumente,
 - ▶ *Jetzt*: *minimum* hat *ein* Argument, nämlich ein Paar.

[Tupel](#)**Motivation**[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische](#)[Semantik](#)[Vertiefung](#)[Unwiderlegbare](#)[Muster](#)[Records](#)[Varianten](#)[Rekursive](#)[Varianten](#)[Widerlegbare](#)[Muster](#)[Parametrisierte](#)[Typen](#)[Polymorphie](#)[Arrays](#)

18. Abstrakte Syntax

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

$e ::= \dots$	<i>Paarausdrücke:</i>
(e_1, e_2)	Konstruktion \setminus Paarbildung
$fst\ e$	Projektion auf die erste Komponente
$snd\ e$	Projektion auf die zweite Komponente

 Die Ausdrücke e_1 und e_2 heißen *Komponenten* des Paares (e_1, e_2) .

18. Statische Semantik

Der Typ eines Paares ist ein Paar von Typen, das sogenannte kartesische Produkt der Typen.

$$t ::= \dots$$
$$| t_1 * t_2$$

Typen:
Paartyp

Typregeln:

$$\frac{\Sigma \vdash e_1 : t_1 \quad \Sigma \vdash e_2 : t_2}{\Sigma \vdash (e_1, e_2) : t_1 * t_2}$$

$$\frac{\Sigma \vdash e : t_1 * t_2}{\Sigma \vdash fst\ e : t_1}$$

$$\frac{\Sigma \vdash e : t_1 * t_2}{\Sigma \vdash snd\ e : t_2}$$

[Tupel](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische](#)[Semantik](#)[Vertiefung](#)[Unwiderlegbare
Muster](#)[Records](#)[Varianten](#)[Rekursive](#)[Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

18. Dynamische Semantik

Wir erweitern den Bereich der Werte um Paare von Werten.

$\nu ::= \dots$
| (ν_1, ν_2)

Werte:
Paare

Auswertungsregeln:

$$\frac{\delta \vdash e_1 \Downarrow \nu_1 \quad \delta \vdash e_2 \Downarrow \nu_2}{\delta \vdash (e_1, e_2) \Downarrow (\nu_1, \nu_2)}$$

$$\frac{\delta \vdash e \Downarrow (\nu_1, \nu_2)}{\delta \vdash \text{fst } e \Downarrow \nu_1}$$

$$\frac{\delta \vdash e \Downarrow (\nu_1, \nu_2)}{\delta \vdash \text{snd } e \Downarrow \nu_2}$$

Tupel

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

18. Tupel

Alle Konstrukte verallgemeinern sich in natürlicher Weise auf *Tupel*, Aggregationen von n verschiedenen Komponenten.

- ▶ $n = 0$:
 - ▶ keine Komponente, keine Projektionsfunktion;
 - ▶ der sogenannte *Unit* Typ umfasst genau ein Element, nämlich `()`;
 - ▶ später: nützlich als „Dummytyp“.
- ▶ $n = 1$:
 - ▶ eine Komponente, eine Projektionsfunktion;
 - ▶ wenig sinnvoll, da anstelle des 1-Tupels stets die einzige Komponente treten kann;
 - ▶ wird von der konkreten Syntax *nicht* unterstützt, da `'(e)'` zur Gruppierung von Ausdrücken dient. (Mehr dazu in Teil VI.)
- ▶ $n = 3$:
 - ▶ drei Komponenten, drei Projektionsfunktionen.
- ▶ ...

Tupel

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Vertiefung

Unwiderlegbare
Muster

Records

Varianten

Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

18. Vertiefung

Die Funktion *sort2* ordnet zwei natürliche Zahlen; wie lassen sich drei natürliche Zahlen sortieren?

```
let sort3 (a : Nat, b : Nat, c : Nat) : Nat * Nat * Nat =  
  if a ≤ b then  
    if b ≤ c then (a, b, c)  
    else if a ≤ c then (a, c, b) else (c, a, b)  
  else  
    if a ≤ c then (b, a, c)  
    else if b ≤ c then (b, c, a) else (c, b, a)
```

☞ Die Implementierung ist *optimal*: Drei Zahlen können auf 3 Fakultät Arten angeordnet werden (als Formel: $3! = 6$). Mit zwei ineinander geschachtelten Alternativen können aber nur $2^2 = 4$ Fälle unterschieden werden.

[Tupel](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische](#)[Semantik](#)[Vertiefung](#)[Unwiderlegbare
Muster](#)[Records](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

18. Vertiefung

Die Funktion *sort3* lässt sich etwas kompakter aufschreiben, indem wir auf *sort2* zurückgreifen.

```
let sort3 (a : Nat, b : Nat, c : Nat) : Nat * Nat * Nat =  
  let x = sort2 (a, b)  
  in if snd x ≤ c then (fst x, snd x, c)  
      else if fst x ≤ c then (fst x, c, snd x)  
      else (c, fst x, snd x)
```

☞ Diese Version macht die Vorgehensweise deutlich: zunächst werden *a* und *b* geordnet, dann wird die Position von *c* bestimmt.

[Tupel](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische Semantik](#)[Vertiefung](#)[Unwiderlegbare Muster](#)[Records](#)[Varianten](#)[Rekursive Varianten](#)[Widerlegbare Muster](#)[Parametrisierte Typen](#)[Polymorphie](#)[Arrays](#)

19. Motivation

Binden wir ein Paar an einen Bezeichner, so ist es bequem, nicht nur einen Namen für das Paar selbst, sondern auch Namen für die beiden Komponenten vergeben zu können.

```
let sort3 (a : Nat, b : Nat, c : Nat) : Nat * Nat * Nat =  
  let (min, max) = sort2 (a, b)  
  in if max ≤ c then (min, max, c)  
      else if min ≤ c then (min, c, max)  
      else (c, min, max)
```

☞ Selbst vergebene Namen für Komponenten, hier *min* und *max*, sind in der Regel prägnanter als Projektionen wie *fst x* und *snd x*.

[Tupel](#)[Unwiderlegbare
Muster](#)[Motivation](#)[Abstrakte Syntax](#)[Dynamische
Semantik](#)[Vertiefung](#)[Records](#)[Varianten](#)[Rekursive](#)[Varianten](#)[Widerlegbare](#)[Muster](#)[Parametrisierte](#)[Typen](#)[Polymorphie](#)[Arrays](#)


19. Abstrakte Syntax

Bezeichner in Bindungspositionen werden verallgemeinert zu sogenannten *Mustern* (engl. patterns).

$d ::= \dots$	<i>Deklarationen:</i>
let $p = e$	verallgemeinerte Wertedefinition

Muster:

$p \in \text{Pat}$	<i>Muster:</i>
$p ::= x$	Bezeichner
$-$	anonymer Bezeichner \ „don't care“ Muster
$p_1 \ \& \ p_2$	konjunktives Muster
(p_1, p_2)	Paarmuster

 Statische Semantik, siehe Skript/zur Übung.

Tupel

Unwiderlegbare
Muster

Motivation

Abstrakte Syntax

Dynamische
Semantik

Vertiefung

Records

Varianten

Rekursive
VariantenWiderlegbare
MusterParametrisierte
Typen

Polymorphie

Arrays

19. Dynamische Semantik

Beispiele: wir nehmen an, dass der Ausdruck $e = \text{sort2}(e_1, e_2)$ zu dem Wert $\nu = (\nu_1, \nu_2)$ ausgewertet.

Definition

let $_ = e$

let $(min, _) = e$

let $(min, max) = e$

let $x \& (min, _) = e$

let $x \& (min, max) = e$

Umgebung

\emptyset

$\{min \mapsto \nu_1\}$

$\{min \mapsto \nu_1, max \mapsto \nu_2\}$

$\{x \mapsto \nu, min \mapsto \nu_1\}$

$\{x \mapsto \nu, min \mapsto \nu_1, max \mapsto \nu_2\}$

☞ Eine verallgemeinerte Wertedefinition bindet mehrere Bezeichner oder auch keine.

☞ Dynamische Semantik, siehe Skript/zur Übung.

Tupel

Unwiderlegbare
Muster

Motivation

Abstrakte Syntax

Dynamische
Semantik

Vertiefung

Records

Varianten

Rekursive

Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

19. Lösung Knobelaufgabe #1

Mit wievielen Vergleichen lassen sich 5 Zahlen sortieren?

- ▶ Mit weniger als 7 Vergleichen geht es *nicht*:
 - ▶ 5 Zahlen lassen sich auf $5! = 120$ Weisen anordnen.
 - ▶ Mit 6 geschachtelten Vergleichen können nur $2^6 = 64$ Fälle unterschieden werden.
 - ▶ Mit 7 Vergleichen können $2^7 = 128$ Fälle unterschieden werden.
 - ▶ 7 ist die sogenannte informationstheoretische Schranke.
- ▶ Aber, ist es auch möglich, mit 7 Vergleichen auszukommen? Ja!

19. Lösung Knobelaufgabe #1

```
let sort5 (a : Nat, b : Nat, c : Nat, d : Nat, e : Nat) : Nat * Nat * Nat * Nat * Nat =
  let (a, b, c, d) =
    if a ≤ b then
      if c ≤ d then if b ≤ d then (a, b, c, d) else (c, d, a, b)
      else if b ≤ c then (a, b, d, c) else (d, c, a, b)
    else
      if c ≤ d then if a ≤ d then (b, a, c, d) else (c, d, b, a)
      else if a ≤ c then (b, a, d, c) else (d, c, b, a)
  // a ≤ b ≤ d und c ≤ d
  in if b ≤ e then
    if d ≤ e then // a ≤ b ≤ d ≤ e
      if b ≤ c then (a, b, c, d, e)
      else if a ≤ c then (a, c, b, d, e) else (c, a, b, d, e)
    else // a ≤ b ≤ e ≤ d
      if b ≤ c then if c ≤ e then (a, b, c, e, d) else (a, b, e, c, d)
      else if a ≤ c then (a, c, b, e, d) else (c, a, b, e, d)
  else
    if a ≤ e then // a ≤ e ≤ b ≤ d
      if c ≤ e then if a ≤ c then (a, c, e, b, d) else (c, a, e, b, d)
      else if b ≤ c then (a, e, b, c, d) else (a, e, c, b, d)
    else // e ≤ a ≤ b ≤ d
      if a ≤ c then if b ≤ c then (e, a, b, c, d) else (e, a, c, b, d)
      else if c ≤ e then (c, e, a, b, d) else (e, c, a, b, d)
```

19. Lösung Knobelaufgabe #1

Vorgehensweise:

- ▶ Zunächst werden die ersten beiden Zahlen sortiert (1 Vergleich).
- ▶ Dann die zweiten beiden (1 Vergleich).
- ▶ Dann die beiden größeren Zahlen aus den ersten beiden Runden (1 Vergleich).
- ▶ Situation: $a \leq b \leq d$ und $c \leq d$.
- ▶ Dann wird e in die sortierte Folge $a \leq b \leq d$ eingefügt (2 Vergleiche).
- ▶ Schließlich wird c eingefügt (1 oder 2 Vergleiche).

☞ Siehe Donald E. Knuth, TAOCP, Band 3, Seite 183f.

19. Lösung Knobelaufgabe #3

Wieviele Möglichkeiten gibt es, eine Mauer der Breite n zu konstruieren?

- ▶ $n = 0$: eine Möglichkeit, die leere Mauer.



- ▶ $n = 1$: eine Möglichkeit, ein einzelner Quader.



- ▶ $n \geq 2$: ergibt sich als Summe aus der Anzahl der Möglichkeiten für $n - 1$ und der Anzahl der Möglichkeiten für $n - 2$.



19. Lösung Knobelaufgabe #3 — Programm

Als Mini-F# Programm:

```
let rec bob (w : Nat) : Nat =  
  if w ≤ 1 then 1  
    else bob (w ÷ 1) + bob (w ÷ 2)
```

 Das Programm folgt *nicht* dem Peano Entwurfsmuster.

19. Lösung Knobelaufgabe #3 — Demo

```
Mini> bob 4
5
Mini> bob 10
89
Mini> bob 20
10946
Mini> bob 30
1346269
Mini> bob 100
...
```


☞ Um *bob n* auszurechnen, werden mehr als *bob n* Funktionsaufrufe benötigt.
(Nachdenken!)

19. Lösung Knobelaufgabe #3 — Programm

Beobachtung: im Rekursionsschritt benötigen wir *bob* ($w \div 1$) und *bob* ($w \div 2$).

Idee: wir definieren eine Funktion, die beides auf einen Schlag berechnet:
two-bob $w = (\text{bob } w, \text{bob } (w + 1))$.

```
let rec two-bob (w : Nat) : Nat * Nat =
  if w = 0 then (1, 1)
  else let
    (a, b) = two-bob (w ÷ 1)
  in
    (b, a + b)
let fast-bob (w : Nat) : Nat = fst (two-bob w)
```

 Das Programm *folgt* dem Peano Entwurfsmuster.

19. Lösung Knobelaufgabe #3 — Demo

```
Mini> two-bob 10
(89, 144)
Mini> two-bob 20
(10946, 17711)
Mini> two-bob 30
(1346269, 2178309)
Mini> fast-bob 100
573147844013817084101
```

☞ Um *fast-bob* n auszurechnen, werden ungefähr n Funktionsaufrufe benötigt.

Ich habe mir überlegt, wie man die Fakultät mit Hilfe von *peano-pattern* programmieren kann.



$$\text{let } n\text{-and-factorial} : \text{Nat} \rightarrow \text{Nat} * \text{Nat} = \\ \text{peano-pattern } ((0, 1), \text{fun } (n, s) \rightarrow (n + 1, s * (n + 1)))$$

Wir konstruieren ein Paar: 1. Komponente: aktueller Wert von n , 2. Komponente: Fakultät n .

Das geht aber nicht durch den Typechecker!

This expression was expected
to have type 'Nat'
but here has type 'Nat * Nat'



Ich weiß. Das hatten wir doch schon besprochen: der Typ von *peano-pattern* ist zu speziell.

20. Motivation

- ▶ Bei Paaren und allgemein bei Tupeln spielt die Reihenfolge der Komponenten eine Rolle:
 - ▶ (12, 1, 2018) versus (1, 12, 2018);
 - ▶ ("Stefan", "Thomas") versus ("Thomas", "Stefan").
- 👉 Die Rolle der Komponenten ist nur implizit festgelegt: Programmierkonvention.
- ▶ Alternative: *Records* statt Tupel.
 - ▶ { *day* = 12; *month* = 1; *year* = 2018 };
 - ▶ { *forename* = "Stefan"; *surname* = "Thomas" }.
- 👉 Die sogenannten *Labels* machen die Rolle der verschiedenen Komponenten explizit.
- ▶ Die Reihenfolge, in der die benannten Komponenten aufgeschrieben werden, ist irrelevant.
 - ▶ { *month* = 1; *day* = 12; *year* = 2018 };
 - ▶ { *surname* = "Thomas"; *forename* = "Stefan" }.
- ▶ Mit Hilfe der Labels können Komponenten auch extrahiert werden: *date.year* oder *person.surname*.

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische
Semantik](#)[Vertiefung](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

20. Motivation

Bevor Records verwendet werden können, müssen die Labels zunächst mit einer sogenannten *Typdefinition* bekannt gemacht werden.

```
type Date = { day : Nat; month : Nat; year : Nat }
```

```
type Name = { forename : String; surname : String }
```

Eine Recordtypdefinition führt zwei verschiedene Dinge ein:

- ▶ einen Namen für den Recordtyp: *Date* und *Name*,
- ▶ Namen um Komponenten des Recordtyps zu extrahieren: *day*, *month*, *year*, *forename* und *surname*. Diese Bezeichner heißen auch *Recordlabels* oder kurz *Labels*.

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

- ▶ *year* hat im Prinzip den Typ *Date* \rightarrow *Nat* und
- ▶ *surname* den Typ *Name* \rightarrow *String*.

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

☞ An die Stelle der Funktionsanwendung tritt die Punktnotation: *date.year* oder *person.surname*.

20. Abstrakte Syntax

☞ Der Übersichtlichkeit halber formalisieren wir nur Records mit genau 2 Komponenten.

Ein Recordtyp wird durch eine Definition eingeführt.

$T \in \text{TyId}$	<i>Typbezeichner</i>
$\ell \in \text{Lab}$	<i>Labels</i>
$d ::= \dots$	<i>Deklarationen:</i>
type $T = \{\ell_1 : t_1; \ell_2 : t_2\}$	Recordtypdefinition ($\ell_1 \neq \ell_2$)

☞ Der Bezeichner T wird durch die Definition neu eingeführt, ebenso die Labels ℓ_1 und ℓ_2 .

20. Abstrakte Syntax

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

$e ::= \dots$

| $\{l_1 = e_1; l_2 = e_2\}$

| $e.l$

Recordausdrücke:

Konstruktion ($l_1 \neq l_2$)

Projektion \ Extraktion

20. Statische Semantik: Vorüberlegungen★

- ▶ *Zur Erinnerung:* ein Bezeichner kann redefiniert werden.

```
let s = false
```

```
let s = 4711
```

☞ Die zweite Definition verschattet die erste.

- ▶ Sollen wir zulassen, dass auch Typen redefiniert können?

```
type Oh = {je : Bool}
```

```
type Oh = {je : Nat }
```

☞ Die zweite Definition verschattet die erste.

- ▶ Aber, was passiert, wenn der Typbezeichner in Typangaben verwendet wird?

20. Statische Semantik: Vorüberlegungen★

```
type Oh = {je : Bool}  
let na-und (oh : Oh) : Bool = not (oh.je)  
type Oh = {je : Nat }  
let egal = na-und {je = 4711 }
```

- ▶ Der Typ *Oh* wird definiert.
- ▶ Die Funktion *na-und* erhält den Typ $Oh \rightarrow Bool$.
- ▶ Der Typ *Oh* wird redefiniert.
- ▶ Die Funktion wird mit einem Element des neuen Typs aufgerufen: $na-und : Oh \rightarrow Bool$ und $\{je = 4711\} : Oh$...
- ▶ ...und das Unglück nimmt seinen Lauf.
- ▶ *Konsequenz*: Typen dürfen *nicht* redefiniert werden. Aus ähnlichen Gründen sind keine lokalen Typdefinitionen erlaubt.

20. Statische Semantik

Die folgenden Typregeln setzen voraus, dass die Typdefinition

$$\text{type } T = \{l_1 : t_1; l_2 : t_2\}$$

bekannt ist.

Typregeln:

$$\frac{\Sigma \vdash e_1 : t_1 \quad \Sigma \vdash e_2 : t_2}{\Sigma \vdash \{l_1 = e_1; l_2 = e_2\} : T}$$

$$\frac{\Sigma \vdash e : T}{\Sigma \vdash e.l_j : t_j}$$

☞ Ähnlich den Regeln für Paare: an die Stelle des anonymen Typs $t_1 * t_2$ tritt der benannte Typ T .

20. Dynamische Semantik

Wir erweitern den Bereich der Werte um Records, deren Komponenten Werte sind.

$\nu ::= \dots$
| $\{l_1 = \nu_1; l_2 = \nu_2\}$

Werte:
Records ($l_1 \neq l_2$)

Auswertungsregeln:

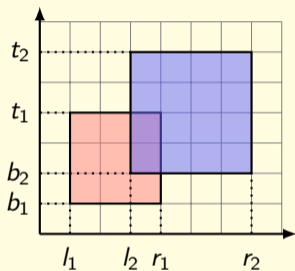
$$\frac{\delta \vdash e_1 \Downarrow \nu_1 \quad \delta \vdash e_2 \Downarrow \nu_2}{\delta \vdash \{l_1 = e_1; l_2 = e_2\} \Downarrow \{l_1 = \nu_1; l_2 = \nu_2\}}$$

$$\frac{\delta \vdash e \Downarrow \{l_1 = \nu_1; l_2 = \nu_2\}}{\delta \vdash e.l_i \Downarrow \nu_i}$$

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische
Semantik](#)[Vertiefung](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

20. Beispiel: Berechnung der Wohnfläche — da capo

Problem: Die Fläche einer Wohnung soll berechnet werden.



Abstraktes Problem: Gesamtfläche zweier gegebener Rechtecke berechnen.

Lösung: Die Gesamtfläche zweier Rechtecke ist die Summe der Einzelflächen minus der Fläche des Durchschnitts. *Ziel:* Vokabular einführen, um die umgangssprachliche Lösung möglichst direkt umzusetzen.

Tupel

Unwiderlegbare
Muster

Records

Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik

Vertiefung

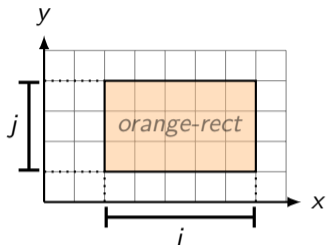
Varianten

Rekursive
VariantenWiderlegbare
MusterParametrisierte
Typen

Polymorphie

Arrays

20. Repräsentation der Daten



```
let orange-rect =  
  let i = { lo = 2; hi = 7 }  
  let j = { lo = 1; hi = 4 }  
  { x = i; y = j }
```

Repräsentation von Intervallen:

```
type Interval = { lo : Nat; hi : Nat } // low und high
```

Repräsentation von Rechtecken:

```
type Rectangle = { x : Interval; y : Interval }
```

☞ Viele alternative Darstellungen von Rechtecken sind denkbar. Welche? Welche Vor- und Nachteile haben die verschiedenen Darstellungen?

20. Rechnen mit Intervallen

```
type Interval = { lo : Nat; hi : Nat }           // low und high
```

Länge eines Intervalls:

```
let length (i : Interval) : Nat = i.hi ÷ i.lo
```

Durchschnitt zweier Intervalle:

```
let intersection (i : Interval, j : Interval) : Interval =  
  { lo = max i.lo j.lo; hi = min i.hi j.hi }
```

 Lassen sich Intervalle auch vereinigen?

20. Rechnen mit Rechtecken

```
type Rectangle = { x : Interval; y : Interval }
```


Flächeninhalt eines Rechtecks:

```
let area (r : Rectangle) = length r.x * length r.y
```

Durchschnitt zweier Rechtecke:

```
let Intersection (r : Rectangle, s : Rectangle) =  
  { x = intersection (r.x, s.x); y = intersection (r.y, s.y) }
```

```
let (^&&^) (r : Rectangle) (s : Rectangle) =  
  { x = intersection (r.x, s.x); y = intersection (r.y, s.y) }
```

 F# erlaubt es, eigene Infixoperatoren zu definieren: statt `Intersection (red-rect, blue-rect)` können wir kurz `red-rect ^&&^ blue-rect` schreiben.

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische
Semantik](#)[Vertiefung](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

20. Berechnung der Gesamtfläche

Jetzt haben wir das Vokabular zusammen, um die umgangssprachliche Lösung in Mini-F# zu transliterieren.

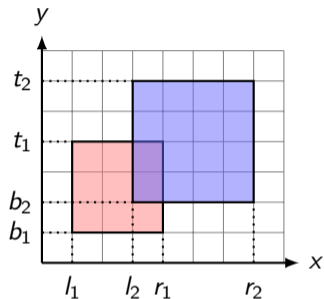
Gesamtfläche zweier Rechtecke:

```
let area2 (r : Rectangle, s : Rectangle) =  
    area r + area s ÷ area (r ^&&^ s)
```

☞ Man erkennt die Fortschritte, die wir erzielt haben, wenn man das obige Programm mit dem Code aus Teil III vergleicht. *Modularer Aufbau*: Das Programm besteht aus vielen kleinen Bausteinen, die separat ausprobiert, verstanden, getestet und verifiziert werden können.

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische
Semantik](#)[Vertiefung](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

20. Beispiel



```
let shift (d : Nat, i : Interval) =  
  { lo = i.lo + d; hi = i.hi + d }
```

```
let red-rect =  
  let i = { lo = 1; hi = 4 }  
  { x = i; y = i }
```

```
let blue-rect =  
  let i = { lo = 2; hi = 6 }  
  { x = shift (1, i); y = i }
```

Der Durchschnitt der Quadrate ergibt das kleine, violette Rechteck in der Mitte.

```
Mini> red-rect ^&&^ blue-rect  
{ x = { lo = 3; hi = 4 }; y = { lo = 2; hi = 4 } }
```

```
Mini> area2 (red-rect, blue-rect)
```

23

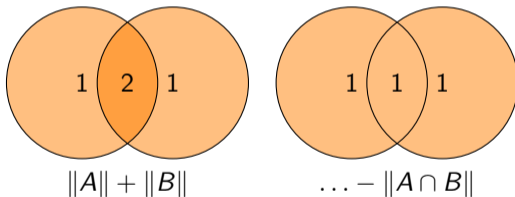
```
Mini> area2 (blue-rect, red-rect)
```

23

20. Gesamtfläche von drei Rechtecken

Werden wir etwas ambitionierter: Wie können wir die Gesamtfläche von *drei* Rechtecken bestimmen?

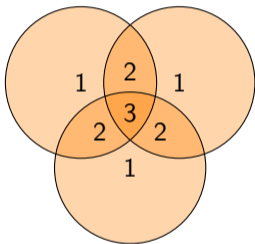
Dem Programm *area2* liegt das *Prinzip der Einschließung und Ausschließung* zugrunde.



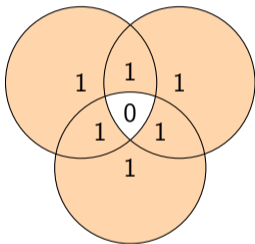
☞ $\|A\|$ ist die Fläche der Punktmenge A.

20. Gesamtfläche von drei Rechtecken

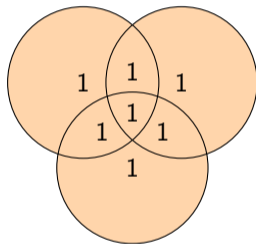
Wenn wir drei Flächen addieren, dann werden die Schnitte von zwei Flächen doppelt, der Schnitt von allen drei Flächen wird dreifach gezählt.



$$\|A\| + \|B\| + \|C\|$$



$$\dots - \|A \cap B\| - \|A \cap C\| - \|B \cap C\|$$



$$\dots + \|A \cap B \cap C\|$$

☞ Die Flächen werden alternierend ein- und ausgeschlossen, daher der Name des Prinzips.

20. Gesamtfläche von drei Rechtecken

Damit ergibt sich das folgende Mini-F# Programm.

```
let area3 (r : Rectangle, s : Rectangle, t : Rectangle) =
  area r + area s + area t
  ÷ area (r ^&&^ s) ÷ area (r ^&&^ t) ÷ area (s ^&&^ t)
  + area (r ^&&^ s ^&&^ t)
```

☞ (Die Infixnotation bietet Vorteile, wenn drei oder mehr Rechtecke geschnitten werden. Welche?)

☞ Das Prinzip der Ein- und Ausschließung lässt sich auch anwenden, um die Gesamtfläche von vier oder mehr Rechtecken auszurechnen. Aber ist das auch eine gute Idee?

20. Fallstudie: Ganze Zahlen

Idee: wir repräsentieren eine ganze Zahl durch einen positiven und einen negativen Summanden.

```
type Int = { pos : Nat; neg : Nat }
```

☞ Die Bedeutung von $\{pos = p; neg = n\}$ ist $p - n$. (Hier meint '−' die *mathematische* Subtraktion auf den ganzen Zahlen.)

20. Ganze Zahlen — Normalisierung

Eine ganze Zahl hat viele verschiedene Repräsentationen: -4 wird zum Beispiel durch $0 - 4$, $5 - 9$ oder $4711 - 4715$ dargestellt.

Normalisierung einer ganzen Zahl:

```
let normalize (i : Int) : Int =
  if i.pos ≥ i.neg then
    { pos = i.pos ÷ i.neg; neg = 0 }
  else
    { pos = 0; neg = i.neg ÷ i.pos }
```

Alternative Definition:

```
let normalize (i : Int) : Int =
  { pos = i.pos ÷ i.neg; neg = i.neg ÷ i.pos }
```

☞ Hier bezeichnet ‘ $-$ ’ die Subtraktion auf den natürlichen Zahlen.

Tupel

Unwiderlegbare
Muster

Records

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Varianten

Rekursive
Varianten

Widerlegbare
Muster

Parametrisierte
Typen

Polymorphie

Arrays

20. Ganze Zahlen — Klassifikation

```
let is-negative (i : Int) : Bool =  
  i.pos < i.neg
```

```
let is-zero (i : Int) : Bool =  
  i.pos = i.neg
```

```
let is-positive (i : Int) : Bool =  
  i.pos > i.neg
```

☞ $i.pos - i.neg > 0$ gdw. $i.pos > i.neg$.

20. Ganze Zahlen — arithmetische Operationen

```
let negate (i : Int) : Int =  
  { pos = i.neg; neg = i.pos }  
let add (i : Int, j : Int) : Int =  
  normalize { pos = i.pos + j.pos; neg = i.neg + j.neg }  
let sub (i : Int, j : Int) : Int =  
  normalize { pos = i.pos + j.neg; neg = i.neg + j.pos }  
let mul (i : Int, j : Int) : Int =  
  normalize { pos = i.pos * j.pos + i.neg * j.neg;  
            neg = i.pos * j.neg + i.neg * j.pos }
```

☞ Subtraktion: $(i.pos - i.neg) - (j.pos - j.neg) = (i.pos + j.neg) - (i.neg + j.pos)$.

☞ *div* und *mod* zur Übung.

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische
Semantik](#)[Vertiefung](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

20. Ganze Zahlen — Konversion und Betrag

```
let int (n : Nat) : Int =  
  { pos = n; neg = 0 }  
let abs (i : Int) : Nat =  
  if i.pos < i.neg then  
    i.neg ÷ i.pos  
  else  
    i.pos ÷ i.neg
```

Alternative Definition:

```
let abs (i : Int) : Nat =  
  (i.neg ÷ i.pos) + (i.pos ÷ i.neg)
```

 Ist die Definition wirklich korrekt?

20. Ganze Zahlen — Vergleichsoperationen

```
let less (i : Int, j : Int) : Bool =  
    i.pos + j.neg < i.neg + j.pos  
let less-equal (i : Int, j : Int) : Bool =  
    i.pos + j.neg ≤ i.neg + j.pos  
let equal (i : Int, j : Int) : Bool =  
    i.pos + j.neg = i.neg + j.pos  
let not-equal (i : Int, j : Int) : Bool =  
    i.pos + j.neg <> i.neg + j.pos  
let greater-equal (i : Int, j : Int) : Bool =  
    i.pos + j.neg ≥ i.neg + j.pos  
let greater (i : Int, j : Int) : Bool =  
    i.pos + j.neg > i.neg + j.pos
```

☞ Echt kleiner: $i.pos - i.neg < j.pos - j.neg$ gdw. $i.pos + j.neg < i.neg + j.pos$.

[Tupel](#)[Unwiderlegbare
Muster](#)[Records](#)[Motivation](#)[Abstrakte Syntax](#)[Statische Semantik](#)[Dynamische
Semantik](#)[Vertiefung](#)[Varianten](#)[Rekursive
Varianten](#)[Widerlegbare
Muster](#)[Parametrisierte
Typen](#)[Polymorphie](#)[Arrays](#)

20. Ganze Zahlen — Demo

```
Mini> int 4711
{pos = 4711; neg = 0}
Mini> add (negate (int 4711), int 815)
{pos = 0; neg = 3896}
Mini> add (int 4711, negate (int 4711))
{pos = 0; neg = 0}
Mini> is-zero (add (int 4711, negate (int 4711)))
true
Mini> mul (neg (int 2), negate (int 3))
{pos = 6; neg = 0}
Mini> abs (mul (negate (int 2), int 3))
6
Mini> div (negate (int 4), int 3)
{pos = 0; neg = 2}
Mini> mod (negate (int 4), int 3)
{pos = 2; neg = 0}
```

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