Funktionale Programmierung in Lean

Burkhardt Renz

Rev 0.9.8 [16.01.2025]

Inhaltsverzeichnis

1 Erste Begegnung mit Lean

Lean ist eine *funktionale Sprache* und gleichzeitig ein *interaktives Beweissystem*. Das Besondere an Lean besteht darin, dass wir nicht nur funktionale Programme implementieren können, sondern dass man auch ihre Eigenschaften spezifizieren und beweisen kann — und dies in der Sprache Lean selbst!

In diesem ersten Kapitel geht es um eine erste Begegnung mit Lean. Wir werden sehen, wie man in Lean Ausdrücke auswertet, Funktionen definiert, eingebaute Typen verwendet, Funktionen höherer Ordnung und rekursive Funktionen definiert. Und natürlich werfen wir einen Blick darauf, wie man in Lean Eigenschaften von Funktionen spezifiziert und beweist. Dabei geht es noch nicht darum, jedes Detail genau zu analysieren, sondern darum, einen Eindruck zu bekommen, wie man in Lean entwickelt.

In dieser Veranstaltung konzentrieren wir uns auf die *funktionale Programmierung in Lean*. Die Erarbeitung des Themas orientiert sich an dem Buch „Functional Programming in Lean“ von David Thrane Christiansen sowie am „Lean Manual“.

Mehr Literatur zu Lean:

- Jeremy Avigad, Leonardo de Moura, Soonho Kong und Sebastian Ullrich: „Theorem Proving in Lean 4“
- Sebastian Ullrich, David Thrane Christiansen et al: „The Lean Language Reference“
- Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl und Jannis Limperg: „The Hitchhiker’s Guide to Logical Verification“
- Arthur Paulino, Damiano Testa, Edward Ayers, Evgenia Karunus, Henrik Böving, Jannis Limperg, Siddhartha Gadgil, Siddharth Bhat: „Metaprogramming in Lean 4“

Weiterführende Literatur zum Zusammenhang von funktionaler Programmierung und Kategorientheorie:

- Benjamin C. Pierce: „Basic Category Theory for Computer Scientists“, MIT Press 1991
- Benedikt Ahrens und Kobe Wullaert: „Category Theory for Programming“
- Bartosz Milewski: „Category Theory For Programmers“

Um die Konzepte der funktionalen Programmierung in Lean zu verstehen, ist diese weiterführende Literatur nicht *notwendig*. Es gibt recht unterschiedliche (und jeweils individuelle) Herangehensweisen an ein neues Thema, eine neue Programmiersprache, neue Konzepte. Einer dieser Wege kann auch darin bestehen, sich die Konzepte *hinter* der neuen Sprache anzusehen — dazu diese Literaturhinweise.

Installation von Lean:

- siehe „Lean Manual — Quickstart“
- Man kann Lean auch Online im „Playground“ verwenden.

Das Skript der Vorlesung wird aus Lean-Quellen generiert. In der Veranstaltung werden wir diese Quellen live verwenden. Ich empfehle, dass Sie schon in der Veranstaltung die Quellen live in Visual Studio Code mitverfolgen und auch direkt damit experimentieren.

Installieren Sie zunächst `elan`, den Versionsmanager von Lean, siehe elan. Für MacOs können Sie dazu das Kommando `brew install elan-init` einsetzen, sofern Sie `brew` als Paketmanager auf Ihrem Rechner installiert haben. Dann legen Sie ein neues Lean-Projekt an mit dem Befehl `lake new fpl math`. Danach kopieren Sie die Quelldateien von meiner Webseite in das Unterverzeichnis `Fpl` des Projektverzeichnisses. Im Projektverzeichnis lädt das Kommando `lake update` die benötigte Bibliothek. Danach können Sie das Projekt in Visual Studio Code öffnen.

Doch beginnen wir zunächst mit ein paar Vorbemerkungen zu Paradigmen von Programmiersprachen und wie sich Lean da einordnet.

1.1 Paradigmen von Programmiersprachen

Paradigma kommt aus dem Griechischen und bedeutet „Lehrbeispiel“, „Vorbild“, „Vorzeigestück“. Verwendet wird der Begriff oft im Sinne von „grundlegende Denkweise“, „Leitbild“, „Weltanschauung“.

Bezüglich Programmiersprachen kann man sich Fragen stellen wie:

- wie sieht man den Prozess einer Berechnung?
- wie definiert man eine Berechnung?
- wie definiert und verwendet man Daten?

Wichtigste Paradigmen von Programmiersprachen sind:

- *imperative* Programmierung:
  - ein Programm ist eine detaillierte Beschreibung, wie der Speicher der Maschine sukzessive verändert werden soll → *imperativ* von lat. *imperare* = befehlen
- *funktionale* Programmierung:
  - ein Programm ist die Berechnung einer Funktion.
  - Funktionale Programmierung basiert auf Alonzo Churchs λ-Kalkül.
  - Funktionen sind „Bürger erster Klasse“, d.h. sie können selbst Parameter oder Rückgabewert einer Funktion sein
  - Konstruktion von Funktionen mittels anderer Funktionen: Funktionen *höheren Typs*
  - *Referenzielle Transparenz*: Ein Ausdruck entspricht seinem Wert und hat immer denselben Wert — dies gilt auch für (*reine*) Funktionen
  - Werte (*immutable data*), keine zustandsbehafteten Objekte
  - In Lean hat jeder Ausdruck einen Typ, der auch von Werten abhängen kann (*dependent types*)
- *logische*, bzw. relationale Programmierung:
  - Programmausführung ↔ Ableitungsprozess, Beweis
  - Programm spezifiziert die Fragestellung, nicht einen Weg zur Lösung
  - Keine lineare Berechnung: System sucht eine Bindung logischer Variablen, die die in der Spezifikation gestellten Bedingungen erfüllen
  - Mehrere Lösungen (sogar unendlich viele) möglich
- *objektorientierte* Programmierung:
  - Programm zur Laufzeit ↔ Geflecht von Objekten, die durch Nachrichten interagieren
  - Objekte kapseln ihren Zustand, der durch Methoden/Nachrichten geändert wird
  - Klassen sind Vorlagen für Objekte, die zur Laufzeit instanziiert werden

- Vererbung von Schnittstellen und Verhalten
- Polymorphismus

1.2 Charakteristika von Lean

Aus dem Lean Manual:

> Lean is a functional programming language that makes it easy to write correct and maintainable code. You can also use Lean as an interactive theorem prover.

> Lean programming primarily involves defining types and functions. This allows your focus to remain on the problem domain and manipulating its data, rather than the details of programming.

> Lean has numerous features, including:

> - Type inference
> - First-class functions
> - Powerful data types
> - Pattern matching
> - Type classes
> - Monads
> - Extensible syntax
> - Hygienic macros
> - Dependent types
> - Metaprogramming
> - Multithreading
> - Verification: you can prove properties of your functions using Lean itself

Im ersten Teil der Veranstaltung wollen wir die Grundelemente der funktionalen Programmierung in Lean kennenlernen, aber auch einen Blick auf das Beweissystem riskieren.

1.3 Ausdrücke und ihre Auswertung

Im Unterschied zu einer *imperativen* Sprache, die eine Folge von Anweisungen vorgibt, um den Zustand der Variablen eines Programms zu ändern, hat Lean keine

Anweisungen, sondern nur Ausdrücke, die ausgewertet werden können.

Das Kommando `#eval` wertet den angegebenen Ausdruck aus. Das Ergebnis wird in Visual Studio Code im Tab „Lean Infoview“ angezeigt.

```
-- Auswertung eines Ausdrucks
#eval 41 + 1 -- ergibt 42
```

Wir definieren simple Funktionen, die ein Argument vom Typ `Nat` und einen Funktionswert vom Typ `Nat` haben:

```
def timesTwo (n : Nat) : Nat := n * 2
-- oder auch:
def timesThree : Nat → Nat := fun n => n * 3

-- und wenden sie an
#eval timesTwo 21 -- ergibt 42
#eval timesTwo (timesThree 7) -- ergibt 42
```

Wir könnten übrigens hier schon das Beweissystem von Lean verwenden, um zu prüfen, dass wir tatsächlich das erwartete Ergebnis erhalten:

```
example : timesTwo (timesThree 7) == 42 := by
  rfl
```

Das Schlüsselwort `example` steht für eine Beweisverpflichtung. In unserem Fall besteht sie darin, dass wir nachweisen sollen, dass die beiden Ausdrücke identisch sind. Hinter `by` schreiben wir den Beweis. Dafür dürfen wir Beweistaktiken verwenden, von denen es sehr viele in Lean gibt. Wir werden später eine ganze Reihe davon kennenlernen. In diesem Fall genügt die Taktik `rfl`, die erfolgreich ist, wenn die beiden Ausdrücke qua Einsetzen ihrer Definitionen identisch sind.

Im Unterschied zu anderen Programmiersprachen werden in Lean keine Klammern für die Angabe der Argumente der Funktion verwendet, also nicht `f(x)`, sondern einfach `f x`.

In Lean gibt es keine Anweisungen, was bedeutet, dass „if then else“ auch ein Ausdruck ist:

```
#eval if true then 1 else 0 -- ergibt 1

def timesTwo' (n: Nat) : Nat :=
  if n == 42 then n else timesTwo n

#eval timesTwo' 21
#eval timesTwo' 42
#eval timesTwo' 21 == timesTwo' 42
```

Lean kennt die Vergleichsoperatoren = und ==. Es handelt sich bei = um die Gleichheit von Termen in Aussagen der Prädikatenlogik, während == die Gleichheit von Booleschen Werten betrifft. Wir verwenden in der funktionalen Programmierung mit Lean ==.

Übungen:

[01.1] Vollziehen Sie nach, wie Lean die folgenden Ausdrücke auswertet:

```
#eval 42 + 19
#eval String.append "A" (String.append "B" "C")
#eval String.append (String.append "A" "B") "C"
#eval if 3 == 3 then 5 else 7
#eval if 3 == 4 then "equal" else "not equal"
```

1.4 Typen

Lean hat ein Typsystem, das extrem ausdrucksstark ist. Es verwendet *dependent type theory*, was nicht nur erlaubt, dass man Typen definieren kann wie die natürlichen Zahlen, Strings, Listen von Strings usw., sondern auch Typen folgenden Inhalts:

- Diese Sortierfunktion hat eine Permutation der eingegebenen Liste zum Ergebnis.
- Die Ergebnisse dieser Funktion haben verschiedenen Typ abhängig vom Wert der Argumente.

- Der Typ dieser Funktion ist eine Aussage, die gerade den Satz von Feit-Thompson beinhaltet, der aussagt, dass jede Gruppe ungerader Ordnung auflösbar ist.

Dass Lean eine funktionale Sprache und zugleich ein Beweissystem ist, rührt daher, dass Lean dieses Typsystem hat und die Typüberprüfung von Typen der Sorte `Prop` (für *Proposition*) nichts anderes ist als das Beweisen von logischen Aussagen. Man nennt diesen Zusammenhang gerne den *Curry-Howard-Isomorphismus*. Da in diesem Kurs die funktionale Programmierung in Lean im Vordergrund steht, wird es bis ins letzte Kapitel des Kurses dauern, bis wir diese Korrespondenz von logischen Aussagen mit Typen im Detail ansehen werden.

Wir befassen uns jetzt aber zuerst mit „einfachen“ Typen.

Entscheidend ist zunächst, dass jeder Ausdruck in Lean, und damit jedes Programm einen Typ haben muss. Nur dann kann der Ausdruck ausgewertet werden.

Das Kommando `#check` ermittelt den Typ eines Ausdrucks:

```
#check 1 + 41 -- ergibt Nat
```

In vielen Situationen kann Lean den Typ eines Ausdrucks selbst ermitteln (Typinferenz). Wenn dies jedoch nicht der Fall ist, kann man den Typ explizit angeben:

```
-- Die Subtraktion kann bei natürlichen Zahlen 0 ergeben
#eval 2 - 1   -- ergibt 1
#eval 1 - 2   -- ergibt 0 (alles unter 0 wird abgeschnitten!)
#check 1 - 2  -- ergibt Nat
```

Wenn wir jedoch die ganzen Zahlen haben wollen, müssen wir dies explizit angeben, da bei einer literalen Zahl Lean als Typ automatisch `Nat` annimmt:

```
#eval (1 - 2 : Int)   -- ergibt -1
#check (1 - 2 : Int)  -- ergibt Int
```

Falsche Typen haben Fehlermeldungen zur Folge:

```
#eval 1 + "A"
#check String.append "Hello" [" ", "world"]
```

Lean hat folgende (einfache) Datentypen in der Bibliothek `Init.Data` bereits definiert:

`Nat`	Natürliche Zahlen

`Int`	Ganze Zahlen
`Float`	Gleitkommazahlen
`Bool`	Boolesche Werte
`Char`	Zeichen (Unicode = gültiger Unicode-Skalar-Wert)
`String`	Zeichenketten

Es gibt in der `mathlib4` natürlich auch die rationalen Zahlen, die wir aber später selbst in Lean definieren und implementieren werden (jedenfalls eine einfache Variante).

Übungen:

[01.2] Gegeben sei die Funktion `twice`:

```
def twice (n: Nat) := n * 2
```

Ermitteln Sie den Typ folgender Ausdrücke und erläutern Sie die Ergebnisse:

```
25 - 30
twice 21
twice 10 + 22
21 + `a`
twice
```

1.4.1 Natürliche Zahlen

Der Type `Nat` repräsentiert die natürlichen Zahlen, sie können also beliebig groß sein. Es gibt keine Overflows wie in Java etwa, wo `2147483647 + 1` den Wert `-2147483648` ergibt.

```
#eval 2147483647 + 1 -- ergibt 2147483648
#eval 100000000000000000 * 200000000000000000000 + 100000000000000000000
```

Ein Literal, das eine natürliche Zahl bezeichnet, wird intern stets als vom Typ `Nat` behandelt:

```
#check 10             -- ergibt Nat
def irgendwas := 32
#check irgendwas + 10 -- ergibt Nat
#eval irgendwas + 10  -- ergibt 42
```

Subtraktion ergibt 0, wenn das Ergebnis als ganze Zahl negativ würde. Die Division natürlicher Zahlen ist die Division mit Rest. Ganz abweichend vom mathematischen Standpunkt ist die Division durch 0 erlaubt und ergibt 0[1].

```
#eval 42 - 43 -- ergibt 0
#eval 42 / 5  -- ergibt 8
#eval 42 % 5  -- ergibt 2, den Rest
-- Was passiert bei Division durch 0?
#eval 42 / 0 -- ergibt 0!!
```

1.4.2 Ganze Zahlen

Der Typ `Int` repräsentiert ganze Zahlen beliebiger Größe, es gibt wie bei natürlichen Zahlen keine Overflows.

Die Division ist Division mit Rest.

```
#check 42        -- ergibt Nat
#check -42       -- ergibt Int
#check (42: Int) -- ergibt Int

#eval (100000000000000000 : Int) * 200000000000000000000 +
 ↪  100000000000000000000

#eval 10 / -3    -- ergibt -3
#eval 10 % -3    -- ergibt 1
```

1.4.3 Gleitkommazahlen

```
#check 0.42   -- ergibt Float
#check 42e-2  -- ergibt Float
```

[1] Division von natürlichen Zahlen ist in Lean eine Funktion, und Funktionen in Lean sind total. Deshalb haben sich die Entwickler (wie auch in anderen Beweissystemen) dafür entscheiden, den Fall der Division durch 0 einfach so zu behandeln, dass sie definitionsgemäß 0 ergibt (siehe auch Division by zero in type theory: a FAQ).

```
#eval 0.42 + 42e-2 - 4.2e-1 -- ergibt 0.420000
#eval 1.0/3 + 1.0/3 + 1.0/3 -- ergibt 1.000000
#eval 1.0/3                 -- ergibt 0.333333
```

1.4.4 Boolesche Werte

Die beiden Booleschen Werte sind `true` und `false`[2].

```
#check true    -- ergibt Bool
#check false   -- ergibt Bool

#eval 2 == 2   -- ergibt true
#check 2 == 2  -- ergibt Bool

#eval true && false    -- ergibt false
#eval and true false   -- ergibt false
#eval true || false    -- ergibt true
#eval or true false    -- ergibt true
#eval xor false true   -- ergibt true
#eval xor true true    -- ergibt false
#eval ¬true            -- ergibt false
#eval not false        -- ergibt true
```

1.4.5 Zeichen

Ein Zeichen ist ein Unicode-Skalarwert, repräsentiert intern als unsigned 32-bit integer.

Zeichen werden in der Syntax mit einfachen Anführungszeichen dargestellt.

```
#eval 'a'               -- ergibt 'a'
#eval isValidChar 0xe4  -- ergibt true
#eval 'a' < 'b'         -- ergibt true
#eval 'a' == 'ä'        -- ergibt false
```

[2] Es gibt auch `True` und `False`, die die Wahrheit respektive den Widerspruch als logische Aussagen repräsentieren. In der funktionalen Programmierung verwendet man den Typ `Bool`, wie in anderen Programmiersprachen auch. Wenn wir aber Eigenschaften von Funktionen beweisen wollen, dann geht es um die Wahrheit logischer Aussagen vom Typ `Prop`.

1.4.6 Strings

Zeichenketten (Strings) sind Listen von Zeichen.

Syntaktisch werden sie mit doppelten Anführungszeichen dargestellt.

Es gibt sehr viele Funktionen für Strings, von denen einige Beispiele folgen. Da Strings Listen von Zeichen sind, lernen wir dabei auch gleich viele Listenfunktionen kennen:

```
#check "Lean"                    -- ergibt String
#eval "Lean"                     -- ergibt "Lean"

#eval "lean".length              -- ergibt 4
#eval "LƎ∀N".length              -- ergibt 4

#eval "lean".front               -- ergibt 'l'
#eval "lean".back                -- ergibt 'n'

#eval "abc".push 'd'             -- ergibt "abcd"
#eval String.push "abc" 'd'      -- ergibt "abcd"

#eval "abc".append "def"         -- ergibt "abcdef"
#eval String.append "abc" "def"  -- ergibt "abcdef"
#eval "abc" ++ "def"             -- ergibt "abcdef"

#eval "abc".toList               -- ergibt ['a', 'b', 'c']

#eval "abc" < "abd"              -- ergibt true
#eval "abc" < "abc "             -- ergibt true
#eval "abc" == "abc "            -- ergibt false

#eval String.splitOn "here is some text"
-- ergibt ["here", "is", "some", "text"]
#eval String.splitOn "here is some text" "some"
-- ergibt ["here is ", " text"]
#eval String.splitOn "here is some text" ""
-- ergibt ["here is some text"]

#eval String.join ["abc", "def", "ghi" ]
-- ergibt "abcdefghi"
#eval String.intercalate  ", " ["abc", "def", "ghi" ]
-- ergibt "abc, def, ghi"
```

```
#eval "abc".contains 'a'          -- ergibt true
#eval "abc".contains 'ä'          -- ergibt false
#eval "abcdef".startsWith "abc"   -- ergibt true
#eval " abcdef ".trim             -- ergibt "abcdef"

#eval String.toUpper "abc"        -- ergibt "ABC"
#eval "abc".capitalize            -- ergibt "Abc"
#eval "äbc".capitalize            -- ergibt "äbc"
```

Man kann Strings auch mit Ergebnissen von Auswertungen mixen (*String interpolation*):

```
def triple x := x * 3

#eval s!"Das Dreifache von 5 ist {triple 5}"
-- ergibt "Das Dreifache von 5 ist 15"
```

Übungen:

[01.3] Machen Sie sich mit den numerischen Typen und dem Booleschen Datentyp vertraut, indem Sie Ausdrücke auswerten, etwa

```
21 + 3 * 7
(21 + 3) * 7
2^10
21 + 3 * 7.0
1 + 2 + 3 + 4 + 5 == 5 * 6 / 2
1 + 2 + 3 + 4 + 5 + 6 == 6 * 7 / 2
6 * 7 / 2
6 * (7 / 2)
42 - 84
42 + (-84)
(42 : Int) - 84
(42 - 84) : Int
```

[01.4] Experimentieren Sie mit Strings. Gegeben sei

```
def hall := "Hall"
def echo := "Echo"
```

```
def ohho := "ohoh"
def lean := "Lean"
def rufz := '!'
def leer := ' '
```

Erzeugen Sie aus diesen Definitionen die beiden folgenden Strings:

```
"Echo macht Hall, Hall, Hall"
"Hallo Lean!"
```

1.5 Definitionen und Funktionen

`def` definiert ein Symbol. Es kann für einen Wert, eine Funktion oder auch einen Typ stehen.

```
def hello := "Hallo"
#check hello
#eval hello

def adams := 42
#check adams
#eval adams

def addOne x := x + 1
#check addOne
#eval addOne (adams - 1)
```

Funktionen können mehrere Argumente haben. Bei ihrer Auswertung wird per *Currying*[3] bei einer Funktion der Arität n mit dem ersten Argument eine Funktion der Arität (n-1) ermittelt usw. bis die letzte einstellige Funktion zum Ergebnis ausgewertet werden kann.

Haben wir eine Funktion `f` mit drei Variablen des Typs `Nat`, dann hat sie folgenden Typ

[3] Der Begriff *Currying* kommt daher, dass Haskell Brooks Curry die Umwandlung einer Funktion mit mehreren Variablen in eine Folge von unären Funktionen 1958 beschrieben hat. Tatsächlich könnte man auch von *Schönfinkeln* sprechen, denn Moses Schönfinkel hat dies schon 1924 beschrieben.

```
Nat → Nat → Nat → Nat
```

und → ist rechts-assoziativ, d.h. der Typ ist explizit

```
Nat → (Nat → (Nat → Nat))
```

Also ist `f n₁` eine Funktion mit 2 Variablen, d.h. vom Typ `Nat → (Nat → Nat)` und `f n₁ n₂` hat den Typ `Nat → Nat`, ist also eine unäre Funktion. Es ist also

```
f n₁ n₂ n₃ = ((f n₁) n₂) n₃
```

Beispiel einer zweistelligen Funktion:

```
def swapSign (n: Nat) (b: Bool) : Int :=
  if b then n else -n

-- check zeigt die Signatur der Funktion
#check swapSign
-- ergibt swapSign (n : ℕ) (b : Bool) : ℤ

-- Man kann aber auch den Typ anzeigen
#check (swapSign) -- ergibt swapSign : ℕ → Bool → ℤ
#check @swapSign  -- ergibt swapSign : ℕ → Bool → ℤ

#check swapSign 42       -- ergibt Bool → Int
#check swapSign 42 true  -- ergibt Int

#eval swapSign 42 false  -- ergibt -42
```

Die Auswertung von Funktionen in Lean ist strikt. Das bedeutet, dass zunächst die Ausdrücke der übergebenen Argumente ausgewertet werden und dann erfolgt die Anwendung der Funktion.

```
#eval swapSign (39 + 3) (2 == 3) -- ergibt -42
```

Übungen:

[01.5] Definieren Sie eine Funktion `joinStringsWith` vom Typ `String → String → String → String`, die einen String erstellt, bei dem das erste Argument zwischen

die beiden folgenden Argumente platziert wird. `joinStringsWith ", " "one" "and another"` sollte zu `"one, and another"` auswerten.

[01.6] Was ist der Typ von `joinStringsWith ": "`?

[01.7] Definieren Sie eine Funktion, die das Volumen eines Quaders mit Kantenlängen in ℕ berechnet.

[01.8] Sei `c` eine Temperatur in Grad Celsius. Die Temperatur `f` in Grad Fahrenheit ergibt sich als: `f = 32 + c * 1.8`. Schreiben Sie Funktionen `celcius2Fahrenheit` und `fahrenheit2Celcius`, die die jeweilige Umrechnung vornehmen.

[01.9] Schreiben Sie eine Funktion, die drei natürliche Zahlen als Argumente nimmt und die Summe der Quadrate der beiden größeren Zahlen zurückgibt.

1.6 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind solche, die Funktionen als Argumente haben oder zu Funktionen auswerten.

Als Beispiel nehmen wir definieren wir die Funktion comp für die Komposition zweier Funktionen Nat → Nat: g ∘ f, „g nach f“

```
def comp (g: Nat → Nat) (f: Nat → Nat): (Nat → Nat) :=
  λ x => g (f x)

def f x := x + 1
def g x := x * 2

def g_kringel_f := comp g f
#check g_kringel_f  -- ergibt g_kringel_f : ℕ → ℕ
#eval g_kringel_f 1 -- (1 + 1) * 2 = 4
```

Hierbei haben wir gleich gesehen, wie man anonyme Funktionen in Lean definiert: durch einen Lambda-Ausdruck beginnend mit λ oder fun, wobei der Körper der Funktion hinter => kommt.

```
#eval (λ x => x * 2) 21        -- ergibt 42
#eval (fun x y => x + y) 41 1  -- ergibt 42
```

Es gibt auch eine andere Ausdrucksweise für einfache Lambdas: (Dabei wird · durch die Tastenkombination von \ mit . erzeugt)

```
#check (· + 1)  -- fun x => x + 1 : Nat → Nat
#eval (2 - ·) 1 -- 1

def h (x y z : Nat) :=
  x + y + z

#check (h · 1 ·)
-- fun x x_1 => h x 1 x_1 : Nat → Nat → Nat
#eval (h · 1 ·) 3 3 -- 7
```

Übungen:

[01.10] Schreiben Sie eine Funktion `addN` mit einem Argument `n` aus ℕ, die eine Funktion zurückgibt, die zu einer natürlichen Zahl `n` addiert.

[01.11] Schreiben Sie eine Funktion `swap`, die als Argument eine Funktion `α → β → γ` hat und eine Funktion zurückgibt, die die Reihenfolge der beiden Argumente der übergebenen Funktion vertauscht.

`swap` sollte also zum Beispiel so wirken:

```
#eval (swap (· - · ) 3 (2 : Int )) -- ergibt -1
```

Anmerkung: Die Funktion, die die Argumente vertauscht heißt in Lean `flip`.

[01.12] Definieren Sie eine Funktion `nfach` die als Argumente eine einstellige Funktion natürlicher Zahlen, eine Zahl `n` sowie einen Ausgangswert `k` hat, also

```
def nfach (f: Nat → Nat): Nat → Nat → Nat
```

`nfach` soll die Funktion `f` `n`-mal auf den Wert `k` anwenden.

Beispiel:

Ist `mal2` definiert als

```
def mal2 := (· * 2)
```

dann soll gelten:

```
#eval mal2 (mal2 (mal2 5)) == nfach mal2 3 5  -- ergibt true
```

Anmerkungen:

- Diese Funktion heißt in Lean `Nat.iterate`.
- Für die Definition dieser Funktion brauchen Sie Pattern Matching, das im nächsten Kapitel der Vorlesung genauer erläutert wird, das wir aber in den Beispielen schon gesehen haben.

1.7 Verketten von Funktionen

Man kann die Auswertung von Funktionen auch durch *Pipelining* machen, was Klammern sparen hilft und vielleicht auch das Ziel der Berechnung leichter überschaubar macht.

```
def add1 x := x + 1
def times2 x := x * 2

#eval times2 (add1 100)       -- 202
#eval 100 |> add1 |> times2   -- vorwärts 202
#eval times2 <| add1 <| 100   -- rückwärts 202
```

1.8 Rekursion und Endrekursion

Es ist in Lean ganz natürlich, wie man rekursive Funktionen schreibt.

Wir nehmen als Beispiel die Berechnung der Fakultätsfunktion:

```
def fact : Nat → Nat
  | 0 => 1
  | n + 1 => (n+1) * fact n
```

Bei der Definition wird *Pattern matching* verwendet: Es gibt zwei mögliche Fälle bezüglich des Arguments der Funktion:

- das Argument ist `0`: in diesem Fall haben wir das Ende der Rekursion erreicht und `0!` ist per definitionem 1.
- das Argument hat das Muster `n + 1` ist also größer als `0`: in diesem Fall wird diese Zahl mit dem Ergebnis von `n!` multipliziert.

Diese Definition erlaubt es Lean automatisch zu ermitteln, dass die Rekursion terminiert: denn mit jedem Schritt wird das Argument beim rekursiven Aufruf kleiner bis wir schließlich beim Basisfall `0` angekommen sind.

```
#eval fact 100
```

Wenn wir jedoch eine große Zahl als Argument von `fact` nehmen, passiert Folgendes:

```
-- #eval fact 100000
-- FPL01.lean crashed, likely due to a stack overflow or a bug.
```

Warum bekommen wir einen Überlauf des Stacks? Sehen wir uns etwas genauer an, wie der Aufruf der Funktion ausgewertet wird:

Untersuchen wir mal, wir die Berechnung geht:

```
fact 6 ->
6 * fact 5 ->
6 * 5 * fact 4 ->
6 * 5 * 4 * fact 3 ->
6 * 5 * 4 * 3 * fact 2 ->
6 * 5 * 4 * 3 * 2 *  fact 1 ->
6 * 5 * 4 * 3 * 2 * 1 *  fact 0 ->
6 * 5 * 4 * 3 * 2 * (1 * 1) ->
6 * 5 * 4 * 3 * (2 * 1) ->
6 * 5 * 4 * (3 * 2) ->
6 * 5 * (4 * 6) ->
6 * (5 * 24)) ->
6 * 120 ->
720
```

Diese Art der Berechnung nennt man „linear rekursiv“.

Bei jedem Schritt wird ein Ergebnis gemerkt und dann die Funktion rekursiv aufgerufen => Folge: der Stack bläht sich auf!

Geht das auch anders?

Statt so zu klammern:

```
(6 * (5 * (4 * (3 * (2 * (1 * 1))))))
```

kann man doch auch so klammern:

```
(((((((1 * 6) * 5) * 4) * 3) * 2) * 1) * 1)
```

Diese Art der Berechnung nennt man auch „linear iterativ“.

Was hat sich außer der Klammerung noch geändert: wir starten mit `(1 * n)`. Man nennt den linken Operanden den *Akkumulator*, der mit `1` startet, dann `6` wird, dann `30` usw.

Programmieren wir das mal:

```
def facthelper (n acc : Nat) : Nat :=
  match n with
  | 0 => acc
  | n + 1 => facthelper n (acc * (n+1))
#eval facthelper 6 1  -- ergibt 720


def fact2 (n : Nat) : Nat := facthelper n 1
#eval fact2 6  -- ergibt 720
-- #eval fact2 100000

def factorial (n : Nat) : Nat :=
  let rec helper : Nat → Nat → Nat
      | 0, acc => acc
      | n + 1, acc => helper n (acc * (n+1))
  helper n 1

#eval factorial 6 --ergibt 720
--#eval factorial 100000
```

Eine rekursive Funktion heißt *endrekursiv*, wenn der rekursive Aufruf der Funktion der letzte Akt der Berechnung der Funktion ist. Dann braucht kein zusätzlicher Stackframe für den rekursiven Aufruf angelegt zu werden.

Bei unserem Beispiel `fact` ist der rekursive Aufruf `(n+1) * fact n`, d.h. es wird mit dem Wert des rekursiven Aufrufs selbst eine Multiplikation durchgeführt. Deshalb muss für die Berechnung von `fact n` ein neuer Stackframe angelegt werden.

Bei `factorial` ist der letzte Ausdruck der rekursive Aufruf von `helper` selbst: `helper n (acc * (n+1))`. In diesem Fall kann Lean den bisherigen Stackframe wiederverwenden, weil ja keine weitere Berechnung mehr durchgeführt werden muss.

1.9 Beweise von Eigenschaften von Funktionen

Eine erste Begegnung mit Lean muss natürlich auch das Bemerkenswerteste an der funktionalen Sprache Lean ansprechen. In Lean kann man nicht nur Funktionen definieren, sondern auch Eigenschaften dieser Funktionen *beweisen*, und zwar in der Sprache selbst.

Die Grundidee besteht darin, dass man logische Aussagen und Typen identifizieren kann. Diese Identifizierung wird auch als *Curry-Howard-Isomorphismus* bezeichnet. Sie bedeutet in der Konsequenz, dass der Typchecker von Lean auch die Korrektheit von Beweisen verifizieren kann. Für uns mag es zunächst für den *Eye catcher* genügen, an einem Beispiel zu sehen, wie man solche Beweise führt.

Wer jetzt gleich mehr über das Beweisen in Lean erfahren möchte, dem sei folgende Lektüre empfohlen:

- Burkhardt Renz: Natürliches Schließen in Lean (als erste Einführung), aber natürlich insbesondere
- Jeremy Avigad, Leonardo de Moura, Soonho Kong und Sebastian Ullrich: Theorem Proving in Lean 4

Es gibt für das Beweisen in Lean *Taktiken*. Dies sind Instruktionen, wie ein Beweis zu konstruieren ist. Wir werden den Taktikmodus von Lean in unseren Beispielen verwenden.

Zunächst machen wir ein sehr einfaches Beispiel:

```
∀ n : Nat, fact n > 0
```

Wie kann man eine solche Aussage beweisen? Wir wollen die Aussage für alle natürlichen Zahlen zeigen und erinnern uns, dass man dafür das Prinzip der *vollständigen Induktion* verwendet: man zeigt die Aussage für die Induktionsbasis, hier also für `n = 0` und dann setzt man voraus, dass die Induktionsvoraussetzung gilt, dass also für `n` die Aussage `fact n > 0` richtig ist. Wenn man nun zeigen kann, dass dies auch für `n + 1` der Fall ist, dann gilt die Aussage für alle natürlichen Zahlen[4].

Wir werden im nächsten Kapitel sehen, dass die natürlichen Zahlen in Lean als *induktiver Datentyp* definiert sind:

[4] Gelegentlich wird die vollständige Induktion mit dem Umfallen von Dominosteinen veranschaulicht: „Wenn der erste Dominostein fällt und durch jeden fallenden Dominostein der nächste umgestoßen wird, wird schließlich jeder Dominostein der unendlich lang gedachten Kette irgendwann umfallen.“ (Wikipedia). Allerdings würde es *unendlich lange* dauern, bis alle Dominosteine umgefallen wären. Der eigentlich Clou ist also: sie fallen alle *gleichzeitig* um!

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Eine natürliche Zahl ist die 0 oder sie ist die Nachfolgerin einer natürlichen Zahl. Diese Definition der natürliche Zahlen entspricht genau dem Beweisprinzip der vollständigen Induktion.

Im Folgenden kommt es mir nicht auf die Details der Beweisführung an. Diese werden wir später in der Vorlesung etwas genauer erläutern. Es geht jetzt nur um einen ersten Eindruck. In der Vorlesung können wir in Visual Studio im Fenster *Lean Infoview* verfolgen, wie der Beweis abläuft. Dort wird das Ziel des Beweises angezeigt und alle Aussagen, die wir für das Erreichen des Ziels einsetzen können: Lean ist ein *interaktives* Beweissystem!

Wir beginnen einfach mit dem Prinzip der Induktion und setzen dann Taktiken ein[5]:

```
lemma fact_gt_0: ∀ n : Nat, fact n > 0 := by
  intro n             -- nehme ein beliebiges n
  induction n with
  | zero =>           -- Induktionsbasis
    unfold fact       -- Definition von fact einsetzen
    decide            -- Entscheide die Ungleichung
  | succ n ih =>      -- Induktionsschritt
                         -- ih ist die Induktionsvoraussetzung
    unfold fact       -- Definition von fact einsetzen
    positivity        -- kann Ziele der Form term > 0 lösen

#print fact_gt_0
-- ergibt einen Term, dessen Typ gerade die Aussage des Lemmas ist
#check fact_gt_0 -- ergibt fact_gt_0 (n : ℕ) : fact n > 0
```

Zum Schluss dieser ersten Begegnung mit Lean wollen wir noch eine interessantere Aussage beweisen. Wir haben zwei Varianten für die Fakultätsfunktion definiert, die eine ist endrekursiv, die andere nicht. Man mag sich durch genaues Hinsehen und etwas Überlegen davon überzeugen, dass die beiden Funktionen für jede natürliche Zahl das identische Ergebnis liefern. Viel besser ist es jedoch, wenn wir Lean überprüfen lassen, dass dies tatsächlich so ist.

Dazu formulieren wir die zu beweisende Aussage:

[5] Ein *cheatsheet* zu Taktiken in Lean findet man im Kurs Kevin Buzzard: Formalising Mathematics.

```
∀ n : Nat, fact n = factorial n
```

```
example: ∀ n : Nat, fact n = factorial n := by
  intro n                    -- nehme ein beliebiges n
  induction n with
  | zero =>                  -- Induktionsbasis
    unfold fact              -- vereinfache durch die Definitionen
    unfold factorial            -- von fact, factorial und
    unfold factorial.helper     -- factorial.helper
    rfl                      -- rfl beweist definitorische Gleichheit
  | succ n ih =>             -- Induktionsschritt
    unfold fact              -- vereinfache durch die Definition
    rw [ih]                  -- setze die Induktionsvoraussetzung ein
    unfold factorial         -- vereinfache durch die Definition
    rw [mul_comm]            -- vertausche die Operanden von *
    sorry                    -- was bleibt zu zeigen?
```

Nun bleibt zu zeigen:

```
⊢ (factorial.helper n 1) * (n + 1) = factorial.helper n (1 * (n + 1))
```

Überlegen wir kurz: Der `factorial.helper` nimmt den Startwert `1`, multipliziert ihn mit `n`, dann das Ergebnis mit `n-1` usw. Wenn man nun das Gesamtergebnis (nämlich `n!`) mit `n + 1` multipliziert, dann hätte man genauso gut den Startwert mit `n + 1` multiplizieren können.

Diese Aussage erheben wir zu einem Lemma, das wir zuerst zeigen, dann können wir den Beweis abschließen:

```
lemma fh_mul (n a m : ℕ) :
  (factorial.helper n a) * m = factorial.helper n (a * m) := by
  induction n generalizing a with  -- Induktion für beliebiges a
  | zero =>                        -- Induktionsbasis
    unfold factorial.helper        -- setze Definition ein
    rfl                            -- definitorische Gleichheit
  | succ n ih =>                   -- Induktionsschritt
    unfold factorial.helper        -- setze Definition ein
    rw [ih]      -- Induktionsvoraussetzung einsetzen
    ac_rfl       -- Gleichheit mod Akkusativ- und Kommutativgesetz

theorem fact_eq_factorial: fact = factorial := by
  funext n                   -- Gleichheit von Funktionen
  induction n with           -- wie oben
```

```
| zero =>
  unfold fact
  unfold factorial
  unfold factorial.helper
  rfl
| succ n ih =>
  unfold fact
  rw [ih]
  unfold factorial
  rw [Nat.mul_comm]
  rw [factorial.helper]   -- setze ein
  rw [fh_mul]             -- wir verwenden das Lemma
```

Übungen:

[01.13] Entwickeln Sie eine endrekursive Funktion `myListSum`, die die Elemente einer Liste natürlicher Zahlen addiert. (Sie können gerne den Abschnitt Tail Recursion aus dem Buch von David Thrane Christiansen zu Rate ziehen, denn Sie benötigen Konzepte, die bisher nicht in der Vorlesung vorgekommen sind.)

[01.14] Die Fibonacci[6]-Zahlen sind definiert durch

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)  für n > 1
```

- Setzen Sie diese Definition in eine rekursive Funktion um, die `fib n` berechnet.
- Programmieren Sie eine endrekursive Variante nach folgender Idee:
  - Initialisiere `a = 1, b = 0`.
  - Wiederhole gleichzeitig die Transformationen `(a = a + b, b = a)` n-mal.
- Zusatzaufgabe: In welcher europäischen Stadt gibt es ein Museum, dessen Fassade die Fibonacci-Folge ziert?

[6] Leonardo da Pisa, genannt Fibonacci, etwa 1180 - 1241

1.10 Zusammenfassung

- **Auswertung:** Lean wertet Ausdrücke strikt aus: Jeder Subausdruck wird durch seinen Wert ersetzt bis der Wert des Ausdrucks ermittelt ist. Wenn eine Variable einen Wert hat, dann kann sich dieser niemals mehr ändern.
- **Typen:** Jeder Ausdruck in Lean hat einen Typ. Wir haben einfache Typen kennengelernt wie die natürlichen Zahlen, die ganzen Zahlen sowie Strings, aber auch Funktionstypen wie `Nat → Nat`.
- **Funktionen** können als anonyme Funktionen (Lambda-Ausdrücke) oder mit einem Funktionsnamen definiert werden. Auch sie haben natürlich einen Typ, zum Beispiel hat eine einstellige Funktion über den natürlichen Zahlen den Typ `Nat → Nat`.
- **Funktionen höherer Ordnung** können Funktionen als Argumente und/oder als Rückgabewerte haben.
- **Rekursive Funktionen** sind solche, die sich selbst in der Definition der Funktion verwenden. Rekursive Funktionen in Lean müssen so definiert werden, dass Lean erkennt, dass die Rekursion terminiert — oder es muss ein Beweis geführt werden, dass dies der Fall ist. Dies haben wir noch nicht genauer untersucht. Es ergibt sich aus der Tatsache, dass Lean ja auch ein logisches Beweissystem ist. Wir werden das später noch genauer sehen. Für die Definition rekursiver Funktionen verwendet man *Pattern Matching*: auch dieses Konzept haben wir bisher nur an Beispielen kennengelernt, wir werden es im nächsten Kapitel genauer erläutern. Um einen Überlauf des Stacks bei rekursiven Funktionen zu vermeiden, wird *Endrekursion* verwendet. Auch dafür haben wir Beispiele gesehen.
- **Beweise:** wir haben einen kurzen Blick auf Lean als Beweissystem geworfen und im *Lean Infoview* von Visual Studio Code verfolgt, wie man mit Taktiken die Aussage zeigt, dass die „normale“ Version der Fakultät mit der endrekursiven Version übereinstimmt. Wir konzentrieren uns in dieser Veranstaltung auf Lean als funktionale Sprache. Erst in Kapitel 7 werden wir uns etwas eingehender mit dem Beweisen in Lean beschäftigen.

---

2 Datentypen definieren und verwenden

Wir haben bereits gesehen, wie man Funktionstypen in Lean definieren kann. Und wir haben eine ganze Reihe von einfachen Datentypen kennengelernt, die Lean mitbringt. Dabei wurde erwähnt, dass die natürlichen Zahlen `Nat` zum Beispiel in Lean tatsächlich ein *induktiver* Typ sind. Diese Typen wollen wir nun näher betrachten und sehen, wie man eigene Typen definiert und verwendet.

2.1 Enumerationen

Die einfachste Art eines induktiven Typs ist ein Typ mit einer endlichen, aufgezählten Menge von Elementen. Das Standardbeispiel dafür ist der Typ `Weekday`:

```
inductive Weekday where
  | sunday    : Weekday
  | monday    : Weekday
  | tuesday   : Weekday
  | wednesday : Weekday
  | thursday  : Weekday
  | friday    : Weekday
  | saturday  : Weekday
```

Das Schlüsselwort `inductive` definiert einen induktiven Typ, indem die verschiedenen Konstruktoren für den Typ angegeben werden. Man spricht auch von einem *Summentyp*, weil er die disjunkte Vereinigung der angegebenen Elemente ist.

In dem einfachen Fall einer Enumeration sind die Konstruktoren einfach die verschiedenen Elemente des Typs, die sonst keine weiteren unterschiedlichen Eigenschaften haben. Die Konstruktoren gehören zum Namensraum `Weekday`, der durch die Definition eingeführt wurde.

In diesem einfachen Fall könnte man die Angabe des Typs bei den Konstruktoren auch weglassen, also

```
inductive Weekday where
```

```
  | sunday
  | usw.

#check Weekday -- Type
#check Weekday.monday -- Weekday
```

Funktionen für Enumerationen definiert man entsprechend der Definition des Typs durch Pattern Matching:

```
def natOfWeekday (d : Weekday) : Nat :=
  match d with
  | Weekday.sunday    => 1
  | Weekday.monday    => 2
  | Weekday.tuesday   => 3
  | Weekday.wednesday => 4
  | Weekday.thursday  => 5
  | Weekday.friday    => 6
  | Weekday.saturday  => 7

#check natOfWeekday
#check (natOfWeekday)

#eval natOfWeekday Weekday.tuesday -- 3
```

Wenn wir explizit Funktionen im Namespace `Weekday` definieren, dann können wir die Symbole der Enumeration ohne die Angabe des Namensraums verwenden:

```
namespace Weekday

def next (d : Weekday) : Weekday :=
  match d with
  | sunday    => monday
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => saturday
  | saturday  => sunday

#check next sunday
```

Die Typklasse `Repr` wird von Lean verwendet, wenn Werte von Typen ausgegeben werden sollen. Wir machen den Typ `Weekday` zu einer Instanz von `Repr`, indem wir die Funktion `reprPrec` für Weekday implementieren. (Mehr über Typklassen in Teil 6 der Veranstaltung.)

```
#check Repr.reprPrec
```

Die Dokumentation in `Init.Data.Repr.lean` beschreibt diese Funktion so:

> Turn a value of type α into `Format` at a given precedence. The precedence value can be used to avoid parentheses if they are not necessary. reprPrec : α → Nat → Format

Wenn unsere Enumeration eine Instance von `Repr` ist, dann können wir die Werte per `#eval` anzeigen lassen.

Man kann in Lean diese Funktion auch bei der Definition der Enumeration automatisch erzeugen lassen. Wir werden dies am Beispiel `Month` sehen.

```
instance: Repr Weekday where
  reprPrec
    | sunday, _    => "Sunday"
    | monday, _    => "Monday"
    | tuesday, _   => "Tuesday"
    | wednesday, _ => "Wednesday"
    | thursday, _  => "Thursday"
    | friday, _    => "Friday"
    | saturday, _  => "Saturday"

#eval sunday           -- ergibt Sunday
#eval next sunday      -- ergibt Monday
#eval sunday.next      -- ergibt Monday

def previous : Weekday -> Weekday
  | sunday    => saturday
  | monday    => sunday
  | tuesday   => monday
  | wednesday => tuesday
  | thursday  => wednesday
  | friday    => thursday
  | saturday  => friday

#eval sunday.next.previous --ergibt Sunday
```

Lean ist nicht nur eine funktionale Sprache, sondern auch ein Beweissystem. Wir formulieren die Aussage, dass für ein `d` vom Typ `Weekday` stets gilt: `next (previous d) = d`. Wir wollen dafür einen Beweis führen. Er besteht darin, dass wir für die Aussage eine Funktion angeben, die diese Gleichheit für alle Ausprägungen der Enumeration ergibt. Dabei verwenden wir `rfl` (für *reflexivity*), eine Funktion im

Beweissystem von Lean, die ausdrückt, dass zwei Ausdrücke definitorisch gleich sind.

```
theorem nextOfPrevious: ∀ d: Weekday, next (previous d) = d := by
  intro d
  match d with
  | sunday    => rfl
  | monday    => rfl
  | tuesday   => rfl
  | wednesday => rfl
  | thursday  => rfl
  | friday    => rfl
  | saturday  => rfl
  done

end Weekday
```

Noch ein Beispiel:

```
inductive Month where
  | january
  | february
  | march
  | april
  | may
  | june
  | july
  | august
  | september
  | october
  | november
  | december
deriving Repr
```

`deriving Repr` sorgt dafür, dass die Instanz der Typklasse `Repr`, die wir oben explizit erstellt haben, automatisch von Lean generiert wird.

```
#eval Month.may  --ergibt Month.may
```

Ein weiteres Beispiel für eine Enumeration ist

```
inductive Bool where
  | false : Bool
  | true : Bool
deriving Repr
```

Man könnte dann für `Bool` die Operatoren der Aussagenlogik definieren:

```
def and (p q: Bool): Bool :=
  match p with
    | Bool.true => q
    | Bool.false => Bool.false

def or (p q: Bool): Bool :=
  match p with
    | Bool.true => Bool.true
    | Bool.false => q

def not (p: Bool): Bool :=
  match p with
    | Bool.true => Bool.false
    | Bool.false => Bool.true

#eval and Bool.true Bool.true
#eval and Bool.true Bool.false
#eval and Bool.false Bool.true
#eval and Bool.false Bool.false

#eval or Bool.true Bool.true
#eval or Bool.true Bool.false
#eval or Bool.false Bool.true
#eval or Bool.false Bool.false

#eval not Bool.true
#eval not Bool.false
```

Übungen:

[02.1] Definieren Sie einen Datentyp `Season` für die vier Jahreszeiten und definieren Sie eine Funktion, die zu einer Jahreszeit einen String zurückgibt, der die Jahreszeit beschreibt.

2.2 Rekursive induktive Typen

Rekursive induktive Typen haben die Eigenschaft, dass der definierte Typ bei seiner Definition selbst verwendet wird. Das kanonische Beispiel für rekursive induktive Typen sind die natürlichen Zahlen:

```
inductive Nat where
  | zero: Nat
  | succ: Nat → Nat
deriving Repr
```

Dieser Typ hat zwei Konstruktoren: `zero` hat keine Argumente, während der zweite Konstruktor bereits einen Wert vom Typ `Nat` benötigt. Darin besteht das „Rekursive“ dieser Definition.

```
open Nat

#check zero
#eval zero -- ergibt 0

#check succ (succ (succ zero))
#eval succ (succ (succ zero))          -- ergibt 3
#eval zero |> succ |> succ |> succ     -- ergibt 3
```

Man kann nun auch hier Pattern Matching und Rekursion verwenden, um Funktionen zu definieren. Hier sehen wir unmittelbar, dass der rekursive induktive Datentyp gewissermaßen zur Rekursion einlädt, weil er das Schema, wie die Rekursion zu programmieren ist, vorgibt.

```
def add (m n : Nat) : Nat :=
  match n with
  | Nat.zero   => m
  | Nat.succ n => Nat.succ (add m n)

def sum_n (n: Nat) : Nat :=
  match n with
  | Nat.zero => Nat.zero
  | Nat.succ n => add (Nat.succ n) (sum_n n)

#eval sum_n (Nat.succ (Nat.succ Nat.zero))
#eval Nat.add (succ (succ zero)) (succ zero) -- ergibt 3
```

Ein weiteres Beispiel für eine rekursive induktive Definition ist der Typ der generischen Liste mit Elementen eines Typs α:

```
-- Definition der Liste
inductive List (α : Type u) where
  | nil  : List α
  | cons : α → List α → List α

namespace List

def append (as bs : List α) : List α :=
  match as with
  | nil       => bs
  | cons a as => cons a (append as bs)

end List
```

Wir werden in Kapitel 4 sehen, wie man mit Listen in Lean arbeiten kann.

Übungen:

[02.2] Definieren Sie einen Datentyp für binäre Bäume, deren Knoten mit einer natürlichen Zahl bezeichnet sind.

[02.3] Schreiben Sie eine Funktion `countNodes`, die die Zahl der Knoten des Baums ermittelt.

[02.4] Schreiben Sie eine Funktion `treeHeight`, die die Höhe des Baums ausgibt.

2.3 Strukturen

Strukturen (Verbünde) sind Datentypen, die aus anderen Datentypen zusammengesetzt sind. Man bezeichnet Strukturen auch als *Produkttypen*, weil die Elemente des Typ das kartesische Produkt der Komponenten sind. In Lean haben sie einen Konstruktor, der Werte für die Komponenten der Struktur erwartet und per Default `mk` heißt.

Für ihre Definition gibt es das Schlüsselwort `structure`, weil die ganze Infrastruktur zum Umgang mit Strukturen von Lean bei der Definition automatisch erzeugt wird.

Wir beginnen mit dem Beispiel `Point`, einer Struktur von Punkten mit generischen Koordinaten vom Typ α:

```
structure Point (α: Type u) where
  x : α
  y : α
  deriving Repr

open Point
```

Bei der Definition der Struktur wird (wenn man nichts anderes angibt) automatisch eine Funktion namens `mk` erzeugt, der Konstruktor für Punkte.

```
def p := Point.mk 10 20

#check p.x  -- ergibt p.x : Nat
#eval p.x   -- ergibt 10
#eval p.y   -- ergibt 20
```

Man kann aber auch Objekte der Struktur durch Angabe der Komponenten erstellen:

```
def p1 := { x := 1, y:= 1 : Point Nat}
#check p1
#eval p1  -- ergibt { x := 1, y := 1 }

-- Neue Punkte aus bisherigen erstellen:
def p2 := { p1 with y := 2 }
#eval p2  -- ergibt { x := 1, y := 2 }
```

Man kann Strukturen auch durch weitere Felder erweitern. Dies ergibt eine Form der *Vererbung* von Datentypen.

```
inductive Color where
  | red | green | blue
  deriving Repr

structure ColorPoint (α : Type u) extends Point α where
  c : Color
  deriving Repr

def cp1: ColorPoint Nat := { p1 with c := Color.red}
#eval cp1
-- ergibt { toPoint := { x := 1, y := 1 }, c := Color.red }

def cp2: ColorPoint Nat := { x := 1, y:= 1, c:= Color.blue }
#eval cp2
-- ergibt { toPoint := { x := 1, y := 1 }, c := Color.blue }
```

Übungen:

[02.5] Definieren Sie eine Struktur `Segment`, die ein Segment einer Linie durch ihre Endpunkte definiert und definieren Sie eine Funktion `length : Segment → Float`, die die Länge des Segments berechnet.

[02.6] Definieren Sie einen Typ `Rational`, der rationale Zahlen repräsentiert.

[02.7] Definieren Sie die Funktion `Rational.add` für die Addition rationaler Zahlen.

[02.8] Definieren Sie Funktion `Rational.mul` für die Multiplikation rationaler Zahlen.

[02.9] Was muss man beachten, um die Gleichheit rationaler Zahlen zu überprüfen? Definieren Sie eine Funktion, die beim Überprüfen der Gleichheit rationaler Zahlen benötigt wird.

2.4 Zusammenfassung

- Es gibt **Datentypen**, die aus einer fixen Zahl von Komponenten zusammengesetzt sind. Diese nennt man auch *Produkttypen*, denn der Typ besteht aus dem kartesischen Produkt seiner Komponenten. Es gibt auch Datentypen, die verschiedenartige Elemente enthalten können, diese nennt man dann *Summentypen*, denn der Typ besteht aus der disjunkten Vereinigung der verschiedenen möglichen Typen. Wir haben gesehen, wie diese beiden Arten von Datentypen in Lean definiert und verwendet werden:
- **Enumerationen** sind Summentypen mit einer fixen Anzahl von Elementen. Bei ihrer Verwendung setzt man *Pattern Matching* ein: die Muster beschreiben, welcher der Konstruktoren der Enumeration das übergebene Argument gebildet hat. Musterbeispiel einer Enumeration in Lean ist der Datentyp `Bool`.
- **Rekursive induktive Datentypen** sind Summentypen mit Konstruktoren, die auf den konstruierten Typ selbst referenzieren, die also *rekursiv* sind. Musterbeispiel eines induktiven Datentyps sind die natürlichen Zahlen `Nat`. Induktive Datentypen ergeben gewissermaßen durch ihre Definition, wie man *rekursive Funktionen* über den Typ schreibt.

- **Strukturen** sind Produkttypen in Lean. Strukturen kann man sehen als einen Spezialfall von Typen mit einem Konstruktor, der Werte für die Komponenten der Struktur erwartet. Man kann Strukturen durch weitere Komponenten erweitern, dies ergibt eine Form der Vererbung von Datenstrukturen.

---

3 Weitere eingebaute Typen

3.1 Option

Der Typ `Option` kommt ins Spiel, wenn es vorkommen kann, dass Werte fehlen. Eine Funktion, die das n-te Element einer Liste ergeben soll, wird das nicht tun können, wenn die Liste gar kein n-tes Element hat.

In vielen Programmiersprachen wird in dieser Situation der sehr spezielle Wert `null` als Rückgabewert zum Einsatz kommen oder die Funktion wird eine Exception werfen.

In Lean verwendet man in dieser Situation den Type `Option`, der so definiert ist:

```
inductive Option (α : Type) : Type where
  | none : Option α
  | some (val : α) : Option α
```

Ein Beispiel für die Verwendung von `Option` ist die folgende Funktion `List.head?`, die das erste Element einer Liste ausgibt.

```
def List.head? {α : Type} (xs : List α) : Option α :=
  match xs with
  | [] => Option.none
  | y :: _ => Option.some y
```

Dabei steht `[]` für die leere Liste und `y :: _` für eine Liste, die mit `y` beginnt. (Mehr über verkettete Listen im nächsten Kapitel der Veranstaltung.)

```
def primesUnder10: List Nat := [2, 3, 5, 7]
#eval primesUnder10.head? -- some 2

#eval [].head?
```

ergibt eine Fehlermeldung, weil Lean den Typ des Ausdrucks nicht herleiten konnte. Man muss ein bisschen helfen:

```
#eval ([]: List Nat).head? -- none
```

Der Type `Option` ist also Leans Weise mit fehlenden Werten umzugehen. Er verwendet das Typsystem der Sprache zu einem disziplinierten Vorgehen bei Fehlern.

`Option` ist ein Exemplar der Sorte „Monade“. Wir werden Monaden später noch genauer behandeln und dabei wieder auf `Option` zu sprechen kommen.

3.2 Prod

`Prod` (Produkt) ist eine Struktur für geordnete Paare (2-Tupel). Wir hätten also den Typ `Point` aus Kapitel 2 auch als `Prod` definieren können.

Die Struktur definiert das kartesische Produkt mit den Selektoren `fst` und `snd`:

```
structure Prod (α : Type) (β : Type) : Type where
  fst : α
  snd : β
```

Für `Prod` gibt es eine spezielle Notation:

- Statt `Prod α β` kann man `α × β` schreiben. (Man gibt × ein, indem man backslash gefolgt von x tippt.)
- Man kann dann Werte des Typs angeben, indem man die Komponenten mit Komma getrennt in Klammern schreibt.

Mit `Prod` kann man auch n-Tupel erstellen: `α × β × γ` ist rechts-assoziativ, also `α × (β × γ)`.

Beispiele:

```
def point2d : Nat × Nat := (42, 42)
#eval point2d           -- ergibt (42, 42)
#eval point2d.fst       -- ergibt 42

def point3d : Nat × Nat × Nat := (41, 42, 43)
#eval point3d                 -- ergibt (41, 42, 43)
#eval point3d.fst             -- ergibt 41
#eval point3d.snd             -- ergibt (42, 42)
#eval point3d.snd.fst == 42   -- ergibt true

def sevens1 : String × Int × Nat := ("VII", 7, 4 + 3)
def sevens2 : String × (Int × Nat) := ("VII", (7, 4 + 3))
#eval sevens1 == sevens2           -- ergibt true
```

3.3 Sum

Der Typ `Sum` ergibt die disjunkte Vereinigung der beiden angegebenen Typen. Ein Wert vom Typ `Sum String Nat` ist also ein String *oder* eine natürliche Zahl. Die beiden Konstruktoren heißen `inl` und `inr` für „Injektion links“ bzw. „Injektion rechts“.

```
inductive Sum (α : Type) (β : Type) : Type where
  | inl : α → Sum α β
  | inr : β → Sum α β
  deriving Repr

def five_l : Sum String Nat := Sum.inl "five"
def five_r : Sum String Nat := Sum.inr 5
#eval five_l   -- ergibt "five"
#eval five_r   -- ergibt 5
```

Man kann diesen Typ auch mit zwei identischen Typen definieren. Im Buch von Christiansen kommt das Beispiel von Tiernamen vor. Der linke String enthält Hundenamen, der rechte Katzennamen. (Man gibt das Symbol ⊕ für `Sum` durch die Zeichenfolge backslash o + ein.)

```
def PetName : Type := String ⊕ String

def animals : List PetName :=
  [Sum.inl "Spot", Sum.inr "Tiger", Sum.inl "Fifi", Sum.inl "Rex", Sum.inr
↪  "Floof"]

def howManyDogs (pets : List PetName) : Nat :=
  match pets with
  | [] => 0
  | Sum.inl _ :: morePets => howManyDogs morePets + 1
  | Sum.inr _ :: morePets => howManyDogs morePets

#eval howManyDogs animals   -- ergibt 3
```

Man sieht an dem Beispiel aber auch, dass es besser ist, einen eigenen induktiven Datentyp zu definieren, so dass man den Konstruktoren aussagekräftige Namen geben kann.

3.4 Unit

`Unit` ist ein Typ mit einem Konstruktor ohne Argumente namens `unit`. Mit anderen Worten `Unit` steht für einen einzelnen Wert.

```
inductive Unit : Type where
  | unit : Unit
```

In Lean haben alle Funktionen Argumente. Braucht man eine Funktion ohne Argument, etwa eine Funktion mit Seiteneffekt, dann kann man sie definieren als eine Funktion mit einem Argument vom Typ `Unit`. Funktionen, deren Returnwert in anderen Sprachen als `void` gekennzeichnet wird, haben in Lean den Rückgabewert `Unit`.

Wir werden `Unit` brauchen, sobald wir Programme schreiben, die etwas ausgeben. Die Funktion `IO.println` etwa hat den Typ `String → IO Unit`, weil sie den Seiteneffekt (markiert durch `IO`) erzeugt, aber nichts Interessantes zurückgibt.

3.5 Empty

`Empty` hat überhaupt keinen Konstruktor. Man wird `Empty` für Programme kaum benötigen, bedeutet der Typ ja unerreichbaren Code.

Christiansen nennt eine Situation, in der man `Empty` einsetzen kann:

> However, it is useful in some specialized contexts. Many polymorphic datatypes do not use all of their type arguments in all of their constructors. For instance, Sum.inl and Sum.inr each use only one of Sum's type arguments. Using Empty as one of the type arguments to Sum can rule out one of the constructors at a particular point in a program. This can allow generic code to be used in contexts that have additional restrictions.

3.6 Algebraische Datentypen

Die Datentypen `Prod`, `Sum`, `Empty` und `Unit` sind nicht nur für sich alleine interessant, sondern werden erst so recht mächtig durch ihre *Kombination*. Und wenn man diese Kombinationen betrachtet, ergibt sich eine interessante Analogie.

Wir können aus einem Summentyp `α ⊕ β` auch einen weiteren Summentyp konstruieren: `(α ⊕ β) ⊕ γ` und dabei spielt die Klammerung offensichtlich keine Rolle.

D.h. wir können die Summe als Addition auf Typen sehen und der Typ `Empty` spielt dabei die Rolle der Null in der Addition. Denn in der Tat `α ⊕ Empty` ist isomorph zu `α`, denn `Empty` hat ja kein Element, das in einer disjunkten Vereinigung zu den Elementen von `α` hinzukommen könnte.

Ganz analog können wir beim Produkttyp die Analogie zur Multiplikation finden mit `Unit` als neutralem Element. Natürlich identifizieren wir dabei die Typen via Isomorphie und nicht per Gleichheit wie bei Zahlen.

Auf diese Weise bekommen wir folgende Korrespondenz

Zahlen	Typen
`0`	`Empty`
`1`	`Unit`
`a + b`	`Sum α β`
`a × b`	`Prod α β`
`2 = 1+1`	`Bool = true | false`
`1 + a`	`Option = none | some a`

Wegen dieser Analogie spricht man von *algebraischen Datentypen*. Die Analogie geht sogar noch weiter, wenn wir Funktionstypen mit in die Betrachtung einbeziehen: `α → β` kann man als Exponential sehen und man schreibt auch deshalb für diesen Funktionstyp auch β^α

Warum kann man diese Analogie machen: Stellen wir uns einen endlichen Typ `α` mit 3 Elementen vor und den Typ `Bool` mit 2 Elementen, dann gibt es 2^3 Möglichkeiten Funktionen von `α` mit Werten `true` oder `false` zu bilden: Jedes Element von `α` kann entweder auf `true` oder auf `false` abgebildet werden.

Nimmt man diese Analogie hinzu, erhält man zum Beispiel:

Zahlen	Typen
$\alpha^0 = 1$	`Empty → α`
$1^\alpha = 1$	`α → Unit`
$\alpha^1 = \alpha$	`Unit → α` isomorph `α`

$a^{b+c} = a^b \times a^c$ sagt für Lean aus, dass eine Funktion einer Summe zweier Typen äquivalent ist zu einem Paar von Funktionen der beiden Typen.

$(a^b)^c = a^{b \times c}$ entspricht dem Currying: Eine Funktion, die eine Funktion zurückgibt ist äquivalent zu einer Funktion mit 2 Argumenten.

$(a \times b)^c = a^c \times b^c$ sagt in Lean aus: Eine Funktion, die ein Paar zurückgibt, ist äquivalent zu einem Paar von Funktionen, die jeweils ein Element des Paars ergeben.

Wir wollen im nächsten Kapitel den Datentyp der Liste genauer betrachten. Man kann die Liste als algebraischen Datentyp herleiten:

Eine Liste ist entweder die leere Liste oder eine Liste mit einem zusätzlichen Element `a` als Kopf der Liste. Bezeichnen wir `List α` mal kurz mit `l`, dann können wir diese Definition der Liste so ausdrücken:

```
l = 1 + a × l
```

Wenn wir nun immer wieder das `l` auf der rechten Seite durch diese Gleichung ersetzen, erhalten wir:

```
l = 1 + a × l
l = 1 + a × (1 + a × l) = 1 + a + a×a × l
l = 1 + a + a × a × (1 + a × l) = 1 + a + a×a + a×a×a × l
usw.
```

Wir bekommen so eine unendliche Summe von Tupeln von Elementen in `α`. Also: eine Liste ist entweder leer oder ein Einertupel oder ein Zweiertupel oder ein Dreiertupel oder usw.

3.7 Zusammenfassung

In diesem Kapitel haben wir Typen kennengelernt, die Lean bereits vordefiniert hat. Wenn wir präzise sprechen, haben wir eigentlich **Typkonstruktoren** kennengelernt, die aus gegebenen Typen neue konstruieren wie Produkte oder Summen. Diese Typkonstruktoren zusammen mit den Typen `Empty` und `Unit` haben ganz analoge

Eigenschaften zu Zahlen mit Addition, Multiplikation und Exponentialfunktion: **algebraische Datentypen**[1].

Für Lean besonderes interessant ist der Zusammenhang des Produkt- und des Summentyps zur Logik, was wir in den Übungen etwas beleuchten werden.

Es gibt zwei weitere, sehr wichtige und interessante eingebaute Typen: verkettete Listen und Arrays. Diese betrachten wir im nächsten Kapitel.

Übungen:

[03.1] Schreiben Sie eine Funktion, die ein Produkt von Options zu einer Option eines Produkts transformiert (Signatur: `pairOpt2optPair : Option α × Option β → Option (α × β)`). Probieren Sie die Funktion am Paar der Option des ersten und letzten Elements einer Liste aus.

[03.2] Was bedeutet `Prod` für logische Aussagen in `Prop`?

[03.3] Was bedeutet `Sum` für logische Aussagen?

[03.4] Was ist die Besonderheit der Funktion `f : Empty → α` in Lean? (Lean wertet bekanntlich strikt aus.)

---

[1] Alle Beispiele kommen auch in der Kategorientheorie vor. Wer sich damit genauer beschäftigen möchte, dem empfehle ich das Buch von Bartosz Milewski, insbesondere aber auch seine Vorlesungsvideos.

4 Verkettete Listen und Arrays

4.1 Listen

Verkettete Listen sind eine grundlegende Datenstruktur in der funktionalen Programmierung.

Lean kennt im Unterschied zu objektorientierten Sprachen keine zustandsbehafteten Objekte, sondern alle Werte sind (einmal erzeugt) unveränderlich. Damit dieses Konzept effizient umgesetzt werden kann, muss man bei der „Veränderung“ von Werten vermeiden, dass größere Datenstrukturen kopiert werden müssen.

Bei Listen ist die typische Art des Wachsens der Liste, dass mit `cons` ein neues Element an den Kopf der Liste gesetzt wird. Die bisherige Referenz auf die Liste wird dadurch nicht verändert, sie referenziert nach wie vor die bisherige Liste. Auf diese Weise ist die Liste das einfachste Beispiel von *structure sharing* in sogenannten *persistent data structures* (siehe *Making Data Structures Persistent* von James R. Driscoll, Neil Sarnak, Daniel D. Sleator und Robert E. Tarjan in: *Journal of Computer and System Sciences* **38**, 86–124 (1989) und *Purely Functional Data Structures* von Chris Okasaki).

Wir haben schon gesehen, wie verkettete Listen in Lean als induktiver Typ definiert sind:

```
inductive List (α : Type u) where
  | nil : List α
  | cons (head : α) (tail : List α) : List α
```

Listen sind in Lean generisch, alle Element sind vom selben Typ α.

Der Konstruktor `nil` repräsentiert die leere Liste und der Konstruktor `cons` setzt ein Kopfelement vorne an die bisherige Liste.

In Lean gibt es eine spezielle Syntax für Listen: sie werden in eckige Klammern geschrieben.

```
def empty_list : List Nat := []
#eval empty_list          -- ergibt []

def primesUnder10 : List Nat := [2, 3, 5, 7]
#eval primesUnder10       -- ergibt [2, 3, 5, 7]
```

4.1.1 Grundlegende Funktionen für Listen:

```
-- Anzahl der Elemente einer Liste
#check List.length
#eval primesUnder10.length  -- ergibt 4
#eval ([]: List Nat).length -- ergibt 0

-- Element an einer bestimmten Position der Liste
#check List.get?
#eval primesUnder10.get? 0  -- ergibt some 2
#eval primesUnder10.get? 10 -- ergibt none

-- Element der Liste "ändern"
#check List.set
#eval primesUnder10.set 0 3 -- ergibt [3, 3, 5, 7]
#eval primesUnder10.set 5 3 -- ergibt [2, 3, 5, 7]

-- Liste verlängern
#check List.concat
#eval List.concat primesUnder10 11 -- ergibt [2, 3, 5, 7, 11]

#check List.append
#eval primesUnder10 ++ [11]
#eval List.append primesUnder10 [11, 13, 17, 19]
#eval primesUnder10 ++ [11, 13, 17, 19]

-- Liste reduzieren
-- foldl: Faltet eine Liste durch eine Funktion von links:
-- foldl f init [a, b, c] = f (f (f init a) b) c
#check List.foldl
#eval List.foldl Nat.add 0 primesUnder10
#eval primesUnder10.foldl Nat.add 0
-- foldr: Wendet die Funktion f auf alle Elemente der Liste an,
-- von rechts nach links
-- foldr f init [a, b, c] = f a (f b (f init c))
#eval List.foldr Nat.add 0 primesUnder10   -- ergibt 17
```

```
def intList: List Int := [2]
#eval intList.foldl Int.sub 1 -- = 1 - 2
#eval intList.foldr Int.sub 1 -- = 2 - 1

-- Listen vergleichen
#check List.beq
#eval [1, 2].beq [1, 2]     -- ergibt true
#eval [1, 2].beq [1, 2, 3]  -- ergibt false
#eval ([]: List Nat).beq [] -- ergibt true

-- Lexikalische Ordnung
#check List.lt
#eval [] < [1, 3]
#eval [1] < [1, 3]
#eval [1, 2] < [1, 3]
#eval [1, 4] < [1, 3]
#eval [1, 2, 3] < [1, 3]

def natUp10: List Nat := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

-- Map wendet eine Funktion auf die Elemente der Liste an
#check List.map
#eval natUp10.map (λ n => n * 2)
#eval natUp10.map (·  * 2)

-- Filter filtert Elemente der Liste mittels eines Prädikats
#check List.filter
#eval natUp10.filter (. > 2)
#eval natUp10.filter (. % 2 == 0)

-- Filter mit Option
#check List.filterMap
#eval List.filterMap (λ n => if n > 5 then some (n * 2) else none) natUp10

-- Join macht aus einer Liste von Listen eine flache Liste
#eval List.join [[1], [], [2, 3], [4, 5, 6]]

-- Replicate
#eval List.replicate 5 1   -- ergibt [1, 1, 1, 1, 1]
#check List.replicate 5 1
#eval List.replicate 5 (-1: Int)
#check List.replicate 5 (-1: Int)
```

```
-- Reverse dreht die Liste um
#check List.reverse
#eval natUp10.reverse
-- ergibt [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
#eval "Hallo".toList.reverse.asString -- ergibt "ollaH"

-- Leer?
#check List.isEmpty
#eval natUp10.isEmpty
#eval List.isEmpty ([]: List Nat)

-- Gibt es ein bestimmtes Element in der Liste?
#eval List.elem 3 [1, 4, 2, 3, 3, 7] -- ergibt true
#eval List.elem 5 [1, 4, 2, 3, 3, 7] -- ergibt false

#eval natUp10.contains 1             -- ergibt true
#eval natUp10.contains 11            -- ergibt false
```

4.1.2 Funktionen mit Teillisten

```
-- take
-- Returns the first n elements of xs, or the whole list if n is too large.
#check List.take
#eval List.take 3 natUp10
#eval List.take 13 natUp10

-- drop
-- Removes the first n elements of xs.
#eval List.drop 3 natUp10

-- takeWhile
-- Returns the longest initial segment of xs for which p returns true.
#eval natUp10.takeWhile (. < 5)

-- dropWhile
-- dropWhile p l removes elements from the list until it finds the first
↪  element for which p returns false; this element and everything after it is
↪  returned.
#eval natUp10.dropWhile (. < 5)

-- partition
```

```
-- teilt die Liste entsprechend dem übergebenen Filter
#eval natUp10.partition (. % 2 == 0)
-- ergibt ([0, 2, 4, 6, 8, 10], [1, 3, 5, 7, 9])

-- dropLast
-- entfernt das letzte Element
#eval natUp10.dropLast

-- isPrefixOf, isSuffixOf
#eval List.isPrefixOf [0, 1] natUp10
#eval List.isPrefixOf [1, 2] natUp10
#eval List.isSuffixOf [10] natUp10
#eval List.isSuffixOf [10, 9] natUp10

-- rotateLeft, rotateRight
#eval natUp10.rotateLeft 1
-- ergibt [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0]
#eval natUp10.rotateLeft 5
-- ergibt [5, 6, 7, 8, 9, 10, 0, 1, 2, 3, 4]
#eval natUp10.rotateRight 2
--ergibt [9, 10, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

4.1.3 Listen „verändern“

```
-- replace
-- replace l a b replaces the first element in the list equal to a with b.
#eval natUp10.replace 2 12

-- insert
-- Inserts an element into a list without duplication
#eval List.insert 11 natUp10
#eval List.insert 0 natUp10

-- erase
-- removes the first occurrence of a from l.
#eval natUp10.erase 1
#eval natUp10.erase 12

-- eraseIdx
-- removes the i'th element of the list l.
#eval natUp10.eraseIdx 1
#eval natUp10.eraseIdx 12
```

4.1.4 Suchen in Listen

```
-- find?, findSome?
-- returns the first element for which p returns true, or none if no such
 ↪  element is found.
#eval natUp10.find? (. > 2)
#eval natUp10.find? (. > 12)
#eval natUp10.findSome? (λ x => if x > 9 then some 111 else none)
#eval natUp10.findSome? (λ x => if x > 12 then some 111 else none)

-- lookup
-- treats l : List (α × β) like an association list, and returns the first β
 ↪  value corresponding to an α value in the list equal to a.
#eval List.lookup 3 [(1, 2), (3, 4), (3, 5)] -- some 4
#eval List.lookup 2 [(1, 2), (3, 4), (3, 5)] -- none
```

4.1.5 Logische Operatoren für Listen

```
-- any list p
-- Returns true if p is true for any element of l
#eval natUp10.any (. > 5)
#eval natUp10.any (. > 10)

-- all list p
-- Returns true if p is true for every element of l.
#eval natUp10.all (. < 5)
#eval natUp10.all (. < 12)

-- or list bool-list
-- Returns true if true is an element of the list of booleans l.
#eval List.or [1 == 2, 3 == 4, 5 == 6]
#eval List.or [1 == 2, 3 == 4, 5 == 6, 7 == 7]

-- and list bool-list
-- Returns true if every element of l is the value true.
#eval List.and [1 == 1, 3 <= 4, 5 < 6]
#eval List.and [1 == 1, 3 <= 4, 5 > 6]
```

4.1.6 Zippers

```
-- List.zip
-- Combines the two lists into a list of pairs, with one element from
-- each list. The longer list is truncated to match the shorter list.
#eval List.zip [1, 2 , 3, 4] [1, 2]
-- ergibt [(1, 1), (2, 2)]
#eval [1, 2].zip ['a', 'b']
-- ergibt [(1, 'a'), (2, 'b')]

-- list.zipWith
-- Eine binäre Funktion f und zwei Listen ergibt eine Liste mit den
-- Funktionswerten von f
#eval List.zipWith Nat.add [1, 1, 1] [1, 2, 3]
-- ergibt [2, 3, 4]
#eval [2, 2, 2].zipWith Nat.add [1, 2, 3]
-- ergibt [3, 4, 5]

-- List.unzip
-- Separates a list of pairs into two lists containing the first
-- components and second components.
#eval List.unzip [(1, 1), (2, 2), (3, 3)]
-- ergibt ([1, 2, 3], [1, 2, 3])
```

4.1.7 Bereiche und Enumerationen

```
-- List.range
-- Range n sind die Zahlen 0 bis n-1
#eval List.range 11
#eval (List.range 11 == natUp10)

-- List.iota
-- Bereich n ... 1 absteigend
#eval List.iota 10
#eval (0 :: List.reverse (List.iota 10) == natUp10)

-- List.enum, List enumFrom
-- Indexiert eine Liste
#eval List.enum "abcde".toList
-- ergibt [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
#eval List.enumFrom 1 "abcde".toList
-- ergibt [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

4.1.8 Minimum, Maximum

```
-- List.minimum?, List.maximum?
#eval ([]: List Nat).minimum?
#eval [1, 1].minimum?
#eval (List.iota 10).minimum?
#eval (List.iota 10).maximum?
#eval (List.range 10).maximum?
```

4.1.9 Weitere Funktionen für Listen

```
-- List.intersperse sep list
-- Liste, in der sich die Werte von list mit sep abwechseln
#eval List.intersperse 0 [1, 2, 3, 4]
-- ergibt [1, 0, 2, 0, 3, 0, 4]

-- List.intercalate
-- wie intersperse, aber mit einer Liste, die zwischen Elemente einer Liste von
↪  Listen geschoben wird
#eval List.intercalate ([0, 42]: List Nat) [[1, 2], [3, 4], [5, 6]]
-- ergibt [1, 2, 0, 42, 3, 4, 0, 42, 5, 6]

-- List.eraseDups
-- löscht Duplikate in einer Liste
#eval [1, 1, 1, 2].eraseDups
#eval [1, 1, 2, 1].eraseDups

-- List.eraseReps
-- löscht sich wiederholende benachbarte Werte
#eval [1, 1, 1, 2].eraseReps
#eval [1, 1, 2, 1].eraseReps

-- List.span pred list
-- zerlegt die Liste in zwei Teile: Teil 1 ist das längste Initialsegment, das
↪  pred erfüllt, Teil 2 der Rest
#eval [1, 1, 1, 2, 1, 1].span (. = 1)
-- ergibt ([1, 1, 1], [2, 1, 1])
#eval [1, 1, 1, 2, 1, 1].span (. <= 2)
-- ergibt ([1, 1, 1, 2, 1, 1], [])
#eval [1, 1, 1, 2, 3, 1].span (. <= 2)
-- ergibt ([1, 1, 1, 2], [3, 1])
```

```
-- List.groupBy
#eval [1, 1, 2, 2, 3, 2].groupBy (. == .)
-- ergibt [[1, 1], [2, 2], [3], [2]]
#eval [1, 1, 2, 2, 3, 2].groupBy (. < .)
-- ergibt [[1], [1, 2], [2, 3], [2]]

-- List.removeAll list otherList
-- entfernt alle Elemente in otherList aus list
#eval List.removeAll [1, 1, 5, 1, 2, 4, 5] [1, 2, 2]
-- ergibt [5, 4, 5]
```

4.2 Strings und Listen

Konzeptionell kann man Strings als Listen von Characters sehen.

Viele der Funktionen für Listen, die wir behandelt haben, gibt es auch für Strings und noch viele spezielle Funktionen für Strings (siehe Doku zu String).

Wir können Strings in Listen verwandeln und List von Characters in Strings:

```
-- List.asString
#check List.asString
#eval ['l', 'e', 'a', 'n'].asString

-- String.toList
#check String.toList
#eval "lean".toList
```

4.3 Arrays

Ein Array in Lean ist eigentlich nur ein Wrapper um List, genauer gesagt:

```
structure Array (α : Type u) where
  data : List α
```

Aber dies ist gewissermaßen die logische, die konzeptionelle Sicht. Der eigentliche Unterschied liegt in der Implementierung:

Eine verkettete Liste hat pro Element jeweils eine Indirektion per Pointer auf das folgende Element. Folglich ist die Zugriffszeit auf ein bestimmtes Element proportional zum Index in der Liste. Andererseits können viele verschiedene Referenzen

auf den hinteren Teil einer Liste gleichzeitig vorhanden sein (Stichwort: *persistente Datenstruktur*).

Im Unterschied dazu ist ein Array in Lean als fortlaufende Speicherstrecke implementiert (analog zu einem C++ `std::vector`). In rein funktionalen Sprachen wie Lean ist es nicht möglich, eine bestimmte Position in einer Datenstruktur *in place* zu verändern. Stattdessen wird eine Kopie erstellt, die die gewünschten Änderungen enthält. Für die Verwendung eines Arrays hat der Lean-Compiler und die Laufzeitumgebung eine Optimierung, die es ermöglicht, das Array intern als mutierbare Datenstruktur zu behandeln, wenn es nur eine einzige Referenz auf das Array gibt.

Syntaktisch werden Arrays wie Listen geschrieben, jedoch mit einem führenden #:

```
def hessisch : Array String :=
  #["Frankfurt", "Kassel", "Darmstadt", "Wiesbaden", "Gießen"]

#eval hessisch[1]
#eval hessisch.size
-- #eval hessisch[42] -- führt zu Fehler zur Compile-Zeit

-- Array.push
-- Push an element onto the end of an array. This is amortized O(1) because
 ↪  Array α is internally a dynamic array.
#eval hessisch
#eval hessisch.push "Limburg"

-- Arrays und Lists
def hessischList : List String :=
  ["Frankfurt", "Kassel", "Darmstadt", "Wiesbaden", "Gießen"]
#eval hessischList.toArray
#eval hessisch.data
#check hessisch.data
#check hessisch.toList
#eval hessisch.toList
```

Es gibt wie bei Listen und Strings eine Unmenge von Funktionen für Arrays (siehe Doku von Array.

Einige Beispiele:

```
#eval hessisch.qsort (. < . )
#eval hessisch.map String.length
```

```
#eval hessisch.filter (λ s => s.length > 6)
#eval hessisch.find? (. == "Gießen")
#eval hessisch.contains "Gießen"
#eval hessisch.contains "Marburg"
#eval hessisch.reverse
#eval (hessisch.erase "Gießen").size

#eval Array.flatten #[#[1, 2], #[3, 4]]
```

4.4 Zusammenfassung

In diesem Kapitel haben wir eine Menge von Funktionen für **Listen** (damit auch Strings = Listen von Buchstaben) kennengelernt. Die Liste ist eine der wichtigsten Datenstrukturen in der funktionalen Programmierung. Und wir werden in den Übungen Beispiele kennenlernen, wie man Aufgaben mit Faltungen von Listen erledigen kann, für die man in der imperativen Programmierung Schleifen verwenden würde.

In Lean gibt es auch **Arrays**, denn es gibt viele Situationen in denen die Performance eines Arrays der der verketteten Liste überlegen ist. Aber wenn man Arrays verwendet hat man (1) beim Zugriff über den Index auch eine Beweisverpflichtung, dass er möglich ist und (2) auch zu zeigen, dass Funktionen, die ein Array durchlaufen wirklich terminieren. Dies wollen wir im Rahmen der Vorlesung nicht weiter behandeln. Wer sich dafür interessiert, kann im Kapitel „Arrays and Termination“ in „Functional Programming in Lean“ über diese Fragestellungen nachlesen.

Übungen:

[04.1] In den folgenden Übungen wollen wir uns etwas genauer mit dem „Schweizer Messer“ für Listen befassen, mit `foldr`. Hier die Definition:

```
def foldr (f : α → β → β) (init : β) : List α → β
  | []     => init
  | a :: l => f a (foldr f init l)
```

Das Ergebnis von `foldr` ist ein Wert vom Typ β. Die Parameter sind:

- Eine Funktion `f` mit zwei Parametern: einem Wert vom Typ der Elemente der Liste auf die `foldr` angewendet wird sowie einen Wert vom Typ β – wird also angewandt auf den aktuellen Kopf der Liste und das Ergebnis des bereits bearbeiteten Rests der Liste,
- ein Initialwert vom Typ β, – gibt also das Ergebnis am Ende der Rekursion, bei der leeren Liste und
- die Liste mit Werten vom Typ α.

Die Funktion arbeitet rekursiv. Die Abbruchbedingung ist die leere Liste und es wird der Initialwert zurückgegeben. Ist die Liste nicht leer, dann wird die Funktion f angewandt auf das erste Element der Liste und das Ergebnis des rekursiven Aufrufs auf den Rest der Liste.

Beispiel: Sei

```
def rev := λ (a : Int) (as : List Int) => as ++ [a]

#eval List.foldr rev [] [1, 2, 3]
-- [3, 2, 1]
```

Wie sieht der Ablauf aus:

```
foldr rev [] [1,2,3]
= rev 1 (foldr rev [] [2,3]
= rev 1 (rev 2 (foldr rev [] [3]))
= rev 1 (rev 2 (rev 3 (foldr rev [] [])))
= (rev 2 (rev 3 (foldr rev [] []))) ++ [1]
= ((rev 3 (foldr rev [] [])) ++ [2]) ++ [1]
= (((foldr rev [] []) ++ [3]) ++ [2]) ++ [1]
= ((([]) ++ [3]) ++ [2]) ++ [1]
= (([3]) ++ [2]) ++ [1]
= ([3,2]) ++ [1]
= [3,2,1]
```

Definieren Sie folgende Funktionen mit Hilfe von `foldr`:

[04.1a] `mySum : List Int → Int`

[04.1b] `myProduct : List Int → Int`

[04.1c] `myLength : (List α) → Nat`

[04.1d] `myFilter : (α → Bool) → List α → List α`

[04.1e] `myReverse : List α → List α`

[04.2] Definieren Sie einen Datentyp `Stack` mit den Funktionen `stackNew`, `stackPush`, `stackPeek`, `stackPop`, `stackSize` und `stackIsEmpty`. Berücksichtigen Sie dabei die Namenskonventionen von Lean.

[04.3] Schreiben Sie eine Funktion `isSorted : List Nat → Bool`, die ausgibt, ob eine Liste natürlicher Zahlen sortiert ist.

[04.4] Project Euler Problem 1: Berechnen Sie die Summe aller Vielfachen von 3 oder 5 im Bereich `0...1000`.

[04.5] Project Euler Problem 6: Berechnen Sie die Differenz zwischen der Summe der Quadrate der ersten 100 natürlichen Zahlen und dem Quadrat der Summe dieser Zahlen.

[04.6] Die Collatz-Folge[1] mit dem Startwert einer natürliche Zahl `n` wird folgendermaßen gebildet:

- Ist `n` gerade, dann ist der nächste Wert `n/2`
- Ist `n` ungerade, dann ist der nächste Wert `3 × n + 1`.

Alle bekannten Collatz-Folgen enden mit einem Zyklus der Ziffern `4, 2, 1`. D.h. hat man einmal die `1` erreicht, dann braucht man keine weiteren Folgenglieder mehr auszurechnen.

Ob dies allerdings für alle natürlichen Zahlen als Startwert gilt, ist weder beweisen noch widerlegt!

Schreiben Sie eine Funktion

```
collatzSequence (steps : Nat) (start : Nat) : List Nat,
```

die zum Startwert `start` mit maximal `steps` Schritten die Collatz-Folge als Liste natürlicher Zahlen ausgibt. Sie können natürlich mit der Folge auch schon aufhören, wenn die `1` erreicht wurde.

Zusatzfrage: Warum wird in der Aufgabenstellung eine maximale Zahl von Schritten vorgegeben? Könnte man die Collatz-Folge nicht einfach solange berechnen, bis man den Folgenwert `1` erreicht hat?

---

[1] Lothar Collatz Mathematiker, 1910 - 1990.

5 Ausführbare Programme erzeugen

Bisher haben wir Ausdrücke in Lean in Visual Studio Code interaktiv ausgewertet. Lean hat aber auch die komplette Infrastruktur, um ausführbare Programme zu erzeugen.

Dies wollen wir an zwei Beispielen tun:

1. Ein Hello World
2. Ein `cat` in Lean

(siehe Kapitel 2 in Functional Programming in Lean)

5.1 Hello, World!

5.1.1 Projekt anlegen und bauen

Zunächst erzeugen wir ein neues Projekt mit dem Befehl `lake new hello exe`.

Der Befehl erzeugt ein neues Projektverzeichnis namens `hello` mit folgendem Inhalt:

```
├── Main.lean
├── README.md
├── hello
│   └── Basic.lean
├── hello.lean
├── lakefile.toml
└── lean-toolchain
```

Außerdem wurde bereits ein lokales git-Repository für das neue Projekt angelegt.

Die Datei `lakefile.toml`[1] hat folgenden Inhalt:

```
name = "hello"
version = "0.1.0"
defaultTargets = ["hello"]

[[lean_exe]]
name = "hello"
root = "Main"
```

Die Datei `Main.lean` wurde mit folgendem Inhalt erzeugt:

```
def main : IO Unit :=
  IO.println s!"Hello, world!"
```

Wir können nun die Main-Funktion direkt aus dem Terminal heraus bauen und starten mit

```
lean --run Main.lean
```

Wir können aber auch erst ein ausführbares Programm erzeugen und dann dieses aufrufen: Das Kommando `lake build` erzeugt ein ausführbares Programm namens `hello`.

Das Programm wird im Unterverzeichnis `.lake/build/bin` abgelegt. Wenn man es dort startet, erhält man folgende Ausgabe:

```
Hello, world!
```

5.1.2 Analyse des generierten Codes

Das Symbol `main` hat den Typ `IO Unit`. `IO α` ist in Lean der Typ eines Programms, das bei seiner Ausführung Seiteneffekte der Ein- bzw. Ausgabe erzeugt und einen Wert vom Typ α zurückgibt. Eventuell kann auch eine Exception geworfen werden. Da unser Programm nichts zurückgibt, hat es den Typ `IO Unit` – wie in Abschnitt 3.4 diskutiert.

[1] In der Datei `lakefile.toml` wird die Konfiguration für `lake`, das Build- und Abhängigkeitstool von Lean („lean make“) festgelegt. Das Dateiformat `toml` steht für „Tom's Obvious Minimal Language“. Es gibt auch die Möglichkeit, die Konfiguration in der DSL für `lake` in einer Datei namens `lakefile.lean` festzulegen. Dies war bis Herbst 2024 der Standard, jetzt erstellt `lake new <project> exe` automatisch die Datei `lakefile.toml`.

> Lean distinguishes between *evaluation* of expressions, which strictly adheres to the mathematical model of substitution of values for variables and reduction of sub-expressions without side effects, and *execution* of IO actions, which rely on an external system to interact with the world. (Running a Program)

`IO.println` ist eine Funktion vom Typ `String → IO Unit`, die bei Ausführung den String ausgibt.

Das funktionale Paradigma von Lean besteht in der Auswertung von Ausdrücken in denen einmal mit einem Ausdruck belegte Symbole niemals ihren Zustand ändern. Eine Auswertung eines Ausdrucks wird immer denselben Wert ergeben.

Dieses Konzept allein würde aus Lean eine „Without output machine“ und eine „Without input machine“ machen (Umberto Eco: *Wie man mit eine Lachs verreist und andere nützliche Ratschläge*, dtv 1995 S.50ff) und Lean wäre somit gänzlich unnütz. Deshalb besteht Lean tatsächlich aus zwei Komponenten:

- das Laufzeitsystem von Lean (geschrieben in C) ist für die Interaktion mit der Welt durch `IO actions` zuständig
- der Lean Compiler erzeugt den Code bestehend aus reinen Funktionen, an die Aktionen des Laufzeitsystems Berechnungen delegieren.

Christiansen bebildert diese Arbeitsteilung so:

> Imagine a café that sells coffee and sandwiches. This café has two employees: a cook who fulfills orders, and a worker at the counter who interacts with customers and places order slips. The cook is a surly person, who really prefers not to have any contact with the world outside, but who is very good at consistently delivering the food and drinks that the café is known for. In order to do this, however, the cook needs peace and quiet, and can’t be disturbed with conversation. The counter worker is friendly, but completely incompetent in the kitchen. Customers interact with the counter worker, who delegates all actual cooking to the cook. If the cook has a question for a customer, such as clarifying an allergy, they send a little note to the counter worker, who interacts with the customer and passes a note back to the cook with the result.

> In this analogy, the cook is the Lean language. When provided with an order, the cook faithfully and consistently delivers what is requested. The counter worker is the surrounding run-time system that interacts with the world and can accept payments, dispense food, and have conversations with customers. Working

> together, the two employees serve all the functions of the restaurant, but their responsibilities are divided, with each performing the tasks that they're best at. Just as keeping customers away allows the cook to focus on making truly excellent coffee and sandwiches, Lean's lack of side effects allows programs to be used as part of formal mathematical proofs. It also helps programmers understand the parts of the program in isolation from each other, because there are no hidden state changes that create subtle coupling between components. The cook's notes represent IO actions that are produced by evaluating Lean expressions, and the counter worker's replies are the values that are passed back from effects.

Für die sukzessive Ausführung von IO-Aktionen hat Lean eine spezielle DSL, die *do*-Notation. Wir setzen diese ein, um unser erstes Programm etwas auszubauen:

5.1.3 IO-Aktionen kombinieren

```
def main : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout

  stdout.putStrLn "Bitte geben Sie Ihren Namen ein"
  let input ← stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace

  stdout.putStrLn s!"Hallo {name}!"
```

In der *do*-Notation werden die *Methoden* (Funktionen, die IO-Effekte produzieren) aufgerufen. Zur Anwendung von Methoden sagt man in Lean *Aktion*.

In einem *do*-Block werden die Aktionen jeweils in einer eigenen Zeile angegeben. Man kann stattdessen auch eine Notation wie in anderen Programmiersprachen verwenden. Beispiele:

```
def main : IO UInt32 := do {
  IO.println "hello";
  IO.println "world";
  return 0;
}
```

```
def main : IO UInt32 := do
  IO.println "hello"; IO.println "world"
  return 0
```

Gehen wir nun die Aktionen unseres Programms durch.

Mit `let` werden lokale Symbole mit einem Wert belegt. Man beachte, dass dafür nicht wie im funktionalen Teil der Sprache `:=` verwendet wird, sondern `←`, was ausdrückt, dass hier ein Wert verwendet wird, der durch die Ausführung einer IO-Aktion zustande kommt.

Mit `let` zugewiesene Werte sind im gesamten *do*-Block gültig.

Wir besorgen uns also zuerst die Handles auf `stdin` und `stdout`.

Dann geben wir die Frage nach dem Namen aus und lesen die Antwort in die Variable `input` ein.

Die Bereinigung des Inputs wird als Wert dem Symbol `name` zugewiesen. Hier wird eine reine Funktion verwendet.

Danach wird der interpolierte String ausgegeben.

Soweit unser erstes Programm in Lean.

Mehr über die *do*-Notation findet man im Lean Manual: The do notation. Wir werden in Kapitel 8 näher auf die *do*-Notation eingehen.

Mehr über das IO-System steht in der Dokumentation von Init.System.IO.

5.2 lcat

Wir wollen eine einfache Variante von `cat` in Lean implementieren. Man startet `cat`, in dem man Dateien als Argumente angibt. `cat` liest diese Dateien und „verkettet“ sie (con_cat_enate) und gibt sie dann auf `stdout` aus.

Wir beginnen damit, ein neues Projekt anzulegen: `lake new lcat exe` und wir erzeugen gleich ein ausführbares Programm durch `lake build` im Verzeichnis `lcat`.

Es lässt sich dann starten mit `.lake/build/bin/lcat` und gibt `Hello, world!` aus.

5.2.1 Behandlung des Aufrufs von lcat

Wir beginnen mit der Behandlung der Kommandozeile für `lcat`. Zunächst geben wir einfach die übergebenen Argumente aus. Wenn keine Argumente angegeben wurden, soll `lcat` von `stdin` lesen, was wir durch ein `-` markieren.

```
def main (args: List String) : IO Unit :=
  match args with
  | [] => IO.println "-"
  | _  => IO.println args
```

Es gibt drei Möglichkeiten für die Kommandozeile:

- `main: IO Unit` ist ein Programm, das keine Argumente hat und immer den Exit-Code `0`.
- `main:  IO UInt32` ist ein Programm ohne Argument mit einem Exit-Code.
- `main (args: List String) : IO UInt32` ist ein Programm mit Argumenten und einem Exit-Code.

In unserem Fall brauchen wir die dritte Möglichkeit.

5.2.2 Lesen von Dateien

Wir brauchen einen Zugriff auf den Inhalt der Dateien, deren Name als Argument übergeben wurde. Dazu definieren wir die Methode `fileStream`, die ein Handle auf einen POSIX-Stream zum Zugriff auf den Inhalt der Datei erzeugt.

```
def fileStream (filename : System.FilePath) : IO (Option IO.FS.Stream) := do
  let fileExists ← filename.pathExists
  if not fileExists then
    let stderr ← IO.getStderr
    stderr.putStrLn s!"File not found: {filename}"
    pure none
  else
    let handle ← IO.FS.Handle.mk filename IO.FS.Mode.read
    pure (some (IO.FS.Stream.ofHandle handle))
```

Wir übergeben der Methode eine Instanz von `System.FilePath`, die die Datei namens `filename` repräsentiert. Als Ergebnis werden wir einen `IO IO.FS.Stream` verpackt in

eine `Option` bekommen. Hier sehen wir, wie man `Option` zu sauberem Fehlerhandling einsetzt.

`System.FilePath` hat eine Reihe von Funktionen, die zum Teil in `Init.System.FilePath` definiert sind, aber auch in `Init.System.IO`, so auch `System.FilePath.pathExists`.

Wenn die Datei nicht existiert, wird eine Fehlermeldung erzeugt und ein `IO (Option IO.FS.Stream)`. Der Wert, den `fileStream` zurückgibt, ist vom Typ `Option IO.FS.Stream`. Die Funktion `pure` von `IO` erstellt zu einem übergebenen Wert eine `Option`. Der übergebene Wert `none` ist der Konstruktor von `Option` für den fehlenden Wert.

Wenn die Datei existiert, brauchen wir den Zugriff auf sie im lesenden Modus. Dies geschieht in der Zeile

```
    let handle ← IO.FS.Handle.mk filename IO.FS.Mode.read
```

`handle` hat den Typ `IO IO.FS.Handle` und `IO.FS.Stream.ofHandle` fertigt daraus einen POSIX-Stream, mit dem man auf den Inhalt der Datei zugreifen kann. Auch hier müssen wir die Funktion `pure` von `IO` verwenden.

(`pure` ist eine Funktion einer Monade. Wir werden später noch ausführlich auf Monaden zu sprechen kommen.)

Konzeptionell ist der POSIX-Stream in Lean durch die folgende Struktur repräsentiert:

```
structure Stream where
  flush   : IO Unit
  read    : USize → IO ByteArray
  write   : ByteArray → IO Unit
  getLine : IO String
  putStr  : String → IO Unit
```

Die Felder sind die IO-Aktionen, die den Operationen eines POSIX-Streams entsprechen.

Wir schreiben jetzt eine IO-Aktion, die die Dateien tatsächlich verarbeitet. Die Signatur der Funktion ist:

```
process (exitCode : UInt32) (args : List String) : IO UInt32
```

Das erste Argument ist der Exit-Code, der später den Exit-Code des gesamten Programms ergeben soll und das zweite Argument ist die Liste der Namen der zu bearbeitenden Dateien.

In der Funktion erfolgt die eigentliche Ausgabe des Inhalts einer Datei durch die Funktion `dump`, die wir gleich definieren werden.

Die Funktion `process`:

```
def process (exitCode : UInt32) (args : List String) : IO UInt32 := do
  match args with
  | [] => pure exitCode
  | "-" :: args =>
    let stdin ← IO.getStdin
    dump stdin
    process exitCode args
  | filename :: args =>
    let stream ← fileStream ⟨filename⟩
    match stream with
    | none =>
      process 1 args
    | some stream =>
      dump stream
      process exitCode args
```

Die Funktion unterscheidet drei Möglichkeiten:

- Die Argumentliste ist leer, dann endet das Programm und gibt den Exit-Code als `IO UInt32` zurück.
- Die Argumentliste beginnt mit dem String `"-"`. In diesem Fall soll `stdin` ausgegeben werden. Danach wird die Funktion `process` mit dem bisherigen Exit-Code und den restlichen Dateinamen in der Argumentliste aufgerufen.
- Die Argumentliste beginnt mit einem Dateinamen. Hier kommt jetzt die Funktion `fileStream` ins Spiel. Wir unterscheiden die beiden Möglichkeiten der Option:
  - Bei `none` tun wir gar nichts und fahren mit `process` fort, aber jetzt mit dem ExitCode 1, weil ja ein Fehler aufgetreten ist
  - Gibt es einen Stream, dann verwenden wir ihn und fahren dann fort.

Die Funktion hat eine wichtige Eigenschaft: sie ist *endrekursiv*. Das sieht man daran, dass beim rekursiven Aufruf `process` stets an letzter Stelle der Verarbeitung steht und einfach nur den Aufruf ohne weitere Berechnung oder Manipulation enthält. Das bedeutet, dass der Compiler Code erzeugen kann, bei dem der Funktionsaufruf durch einen Sprung erfolgen kann und keinen Stack benötigt. Deshalb müssen wir keine Sorge haben, dass bei sehr großen Dateien Probleme durch die begrenzte Größe des Stacks entstehen könnten.

Nun brauchen wir noch die Funktion `dump`:

```
def bufsize : USize := 20 * 1024

partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    let stdout ← IO.getStdout
    stdout.write buf
    dump stream
```

Die Funktion liest jeweils `bufsize` Bytes aus dem übergebenen Stream und schreibt sie auf `stdout`. Wenn der gelesene Puffer leer ist, endet die Funktion.

Wir beobachten, dass auch diese Funktion endrekursiv ist.

Außerdem ist sie mit dem Schlüsselwort `partial` gekennzeichnet. Dies bedeutet, dass Lean keinen Beweis benötigt, dass die Funktion auch wirklich terminiert. Das ließe sich in diesem Fall aber auch nicht beweisen, denn man kann nicht sicher sein, dass der gelesene Stream sukzessive kleiner wird. (Würde man zum Beispiel aus `/dev/random` lesen, wäre der Stream unendlich!)

Nun haben wir alle Bestandteile für `lcat` beieinander und wir können `main` anpassen:

```
def main (args : List String) : IO UInt32 :=
  match args with
  | [] => process 0 ["-"]
  | _ =>  process 0 args
```

In Lean müssen wir den Quellcode so organisieren, dass Symbole bereits definiert sind, wenn man sie verwenden möchte. Also erst `bufsize`, dann `dump` usw.

`lake build` erzeugt das Programm.

Wir erstellen Dateien, mit denen wir das Programm verwenden können.

```
echo "Hallo, dies ist unser erstes" > lcat1.txt

echo "Programm in Lean. Cool." > lcat2.txt

lorem -l 200 > lcat3.txt
```

Nun können wir das Programm mit diesen Dateien verwenden:

```
.lake/build/bin/lcat lcat1.txt
.lake/build/bin/lcat lcat1.txt lcat2.txt
echo "und äußerst interessantes" | .lake/build/bin/lcat lcat1.txt - lcat2.txt
.lake/build/bin/lcat lcat3.txt | wc -l
.lake/build/bin/lcat lcat3.txt lcat3.txt | wc -l
.lake/build/bin/lcat lcat4.txt
```

5.3 Aufbau komplexerer Programme

Unsere Beispiele von ausführbaren Programmen sind so einfach, dass der gesamte Code in einem File ist. Bei komplexeren Programmen ist dies nicht der Fall.

Wir wollen an einem Beispiel sehen, wie komplexere Programme strukturiert sind. Wir nehmen als Beispiel doc-gen4, den Dokumentationsgenerator für Lean4.

5.3.1 Organisation des Quellcodes

- `Main.lean` enthält die Main-Funktion. `Main.lean` verwendet die Bibliothek Cli (lean4-cli) für die Auswertung der Argumente für `doc-gen4` und die Weiterleitung zu den eigentlichen Funktionen.
- `DocGen4.lean` importiert die Namensräume von `doc-gen4`.
- Das Verzeichnis `DocGen4` enthält die eigentliche Implementierung:

- `Load.lean`: Lader für eine Liste der Module, die dokumentiert werden sollen.
- `Output.Lean`: zuständig für die Ausgabe der Dokumentation, verwendet die Quelldateien im Verzeichnis `Output`.
- `Process.lean`: importiert die Dateien im Verzeichnis `Process`.

5.3.2 Definition des Programms für `lake`

Die Datei `lakefile.lean` definiert die Abhängigkeiten des Programms und wird von `lake` verwendet, um das Programm zu compilieren und zu linken.

In der Datei wird das Ziel angegeben:

```
@[default_target]
lean_exe «doc-gen4» {
  root := `Main
  supportInterpreter := true
}
```

Es werden Abhängigkeiten definiert und angegeben, wie sie erfüllt werden können, zum Beispiel:

```
require Cli from git
  "https://github.com/mhuisi/lean4-cli" @ "nightly"
```

`lake` hat eine ausführliche Dokumentation, in der der Aufbau von Packages beispielhaft gezeigt wird, die Begriffe (Package, Module etc) definiert werden und die verschiedenen Konfigurationsmöglichkeiten in einem lakefile angegeben werden.

5.4 Zusammenfassung

Wir haben am Beispiel der ersten Vorstellung einer Programmiersprache, an „Hello World“ sowie an einer vereinfachten Implementierung von `cat` gesehen, wie man in Lean ausführbare Programme erzeugt. Dabei haben wir mit `IO` diese Monade mit der *do*-Notation verwendet, die wir später noch ausführlich erläutern werden.

Übungen:

[05.1] Erweitern Sie das Programm `lcat`: Es soll die Option `-h` oder `--help` auf der Kommandozeile unterstützen und wenn angegeben eine Usage-Meldung ausgeben.

[05.2] Schreiben Sie ein Programm `ltree`, das zu einem gegebenen Namen eines Dateiverzeichnisses den Baum der Einträge in diesem Verzeichnis ausgibt. Etwa so:

```
$ ltree path/to/folder/
path/to/folder/
├── a-first.html
├── b-second.html
├── subfolder
│   ├── readme.html
│   ├── code.cpp
│   └── code.h
└── z-last-file.html
```

---

6 Typklassen

6.1 Polymorphismus und Überladung

Bisher haben wir Polymorphismus in folgender Form kennengelernt:

`List α` ist ein *generischer Datentyp*, weil er ja Listen für Elemente jedes beliebigen Typs α definiert. Die Funktion `List.append` ist eine *generische Funktion*, die denselben Algorithmus anwendet, ganz unabhängig vom Typ der Elemente der Liste. Bei dieser Form des Polymorphimus spricht man auch von *parametrischer Polymorphie*, weil die Liste einen Typparameter hat. Der Begriff der *Generizität* drückt aber präziser aus, worum es sich handelt.

Es gibt eine weitere Form, bei der Funktionen polymorph verwendet werden. Nämlich dann, wenn dasselbe Symbol für eine Funktion für verschiedene Typen von Parametern eingesetzt werden kann und dabei je nach Typ der Parameter ein *spezieller* Algorithmus angewendet wird. Diese Form wird auch *Ad-hoc-Polymorphie* genannt oder auch *constrained polymorphism*, besser sollte man aber von *Überladung* sprechen.[1]

Lean kennt nur Funktionen, wie etwa `List.append`. Man kann aber auch Listen aneinanderhängen, in dem man den Operator `++` verwendet. Dabei wird tatsächlich die Funktion verwendet. `++` ist eine *Notation* in Lean. In `Init.Notation` wird dem Parser von Lean mitgeteilt, dass die Funktion `hAppend` der Typklasse `HAppend` (für *heterogenous append*) auch infix als binärer Operator verwendet werden kann.

```
@[inherit_doc] infixl:65 " ++ "  => HAppend.hAppend
macro_rules | `($x ++ $y)  => `(binop% HAppend.hAppend $x $y)
```

[1] Die Begriffe „parametrisierte Polymorphie“ und „Ad-hoc-Polymorphie“ stammen von Christopher Strachey aus seinem Skript *Fundamental Concepts in Programming Languages* von 1967. Er spricht von „Ad hoc“, weil es bei der Überladung im Unterschied zur parametrischen Polymorphie keine einfache systematische Weise gibt, die tatsächlich zu verwendende Funktion zu ermitteln. Lean hat das Konzept der Typklassen für „Ad-hoc-Polymorphie“ und verwendet intern ein Dictionary von Instanzen von Typklassen, in dem bei der *typeclass resolution* die tatsächlich anzuwendende Funktion ermittelt wird.

Der Parser ersetzt dann den Operator durch den Funktionsausfruf. Aus der Definition der Notation sieht er, dass der Infix-Operator ++ links-assoziativ ist und die Priorität 65 bei der Präzedenz von Operatoren hat. Deshalb spricht man beim Überladen von Funktionen auch vom Überladen von *Operatoren* in Lean.

In Lean wird das Konzept der Überladung durch *Typklassen* realisiert, ein Muster, das von Haskell herrührt. Der Ausdruck „Klasse“ steht für „Art“ oder „Sorte“ und hat nichts zu tun mit „Klasse“ in objektorientierten Sprachen. Eine Typklasse kann man eher mit einem Interface in der objektorientierten Welt vergleichen, wie wir gleich sehen werden.

6.2 Typklassen

Typklassen sind benamte Mengen von Funktionssignaturen (syntaktisch aufgebaut wie eine Struktur)[2]. Im Jargon von Lean nennt man die Funktionssignaturen auch *Methoden*, was bedeutet, dass die Signatur für das Überladen definiert wurde. Zum Beispiel

```
class Add (α : Type) where
  add : α → α → α

#check @Add.add
-- @Add.add : {α : Type} → [self : Add α] → α → α → α
```

Dabei bedeutet

- `{α : Type}` ein implizites Argument und
- `[self : Add α]`, dass das Argument vom Typ `Add α` ein implizites Instanzargument ist.

Um für einen bestimmten Typ α die in der Typklasse definierten Methoden zu überladen, erstellt man in Lean eine *Instanz* der Typklasse mit Implementierungen deren Methoden für den Typ ’α‘.[3].

[2] Diese Definition der Typklasse ist nur die halbe Wahrheit. Typklassen können auch logische Aussagen enthalten, die Eigenschaften der Methoden der Typklasse beschreiben. Bei solchen Typklassen muss für eine Instanz auch bewiesen werden, dass diese Eigenschaften erfüllt sind. Wir werden in Kapitel 9 das Beispiel der Typklasse `LawfulFunctor` sehen, in dem Gesetze definiert sind, die ein Funktor erfüllen muss.

[3] Man sieht hier, dass sich Typklassen in Lean und Interfaces in einer objektorientierten Sprache für Java bei aller Ähnlichkeit unterscheiden. Der wichtigste Unterschied was die Wiederverwendung von Code angeht, besteht darin, dass Typklassen die Definition, dass ein Typ eine Instanz ist, trennt von der Definition des Typs selbst. In Java gibt man für eine Klasse die Interfaces, die sie

Lean wird also bei der Auflösung der Überladung eine Instanz der Typklasse finden müssen und diese übergeben.

Zum Unterschied zwischen normalen impliziten Argumenten und impliziten Instanzargumenten sagt Christiansen:

> The most important difference between ordinary implicit arguments and instance implicits is the strategy that Lean uses to find an argument value. In the case of ordinary implicit arguments, Lean uses a technique called *unification* to find a single unique argument value that would allow the program to pass the type checker. This process relies only on the specific types involved in the function's definition and the call site. For instance implicits, Lean instead consults a built-in table of instance values.

Nun definieren wir `Nat`, `Int` und `Float` als Instanzen der Typklasse `Add`:

```
instance : Add Nat where
  add := Nat.add

instance : Add Int where
  add := Int.add

instance : Add Float where
  add := Float.add
```

Wenn wir nun eine Funktion definieren, die `Add.add` verwendet, dann können wir den Effekt des Überladens beobachten:

```
def double [Add α] (x : α) : α :=
  Add.add x x

#check @double
-- @double : {α : Type} → [inst : Add α] → α → α

#eval double 21
#eval double (-21)
#eval double (21 : Float)
```

Instanzen können auch von anderen Instanzen abhängen. In folgendem Beispiel wird eine Instanz der Typklasse `Add` definiert für den Typ `Array a`, wobei der Typ `a` selbst eine Instanz der Typklasse `Add` sein muss:

erfüllt, bei der Definition der Klasse an und man kann dies nicht später tun. In Lean kann man für einen Typ, den man gar nicht selbst erstellt hat, später eine Instanz einer Typklasse erstellen. Es gibt weitere Unterschiede, die in Difference between OOP interfaces and FP type classes gut erläutert werden.

```
instance [Add a] : Add (Array a) where
  add x y := Array.zipWith x y (· + ·)

#eval Add.add #[1, 2] #[3, 4]
-- #[4, 6]

#eval #[1, 2] + #[3, 4]
-- #[4, 6]
```

6.3 Beispiele von Typklassen

6.3.1 Gleichheit

Es gibt in Lean zwei Gleichheitsoperatoren = und ==. Das liegt daran, dass Lean nicht nur eine funktionale Sprache ist, sondern insbesondere auch ein Beweissystem.

- == steht für Boolesche Gleichheit, die zwei Werte auf Gleichheit überprüft und den entsprechenden Booleschen Wert zurückgibt. Boolesche Gleichheit wird via der Typklasse `BEq` definiert.
- = steht für Gleichheit von Termen in logischen Aussagen. D.h. es gibt einen Beweis, dass die beiden Terme, die verglichen werden, gleich sind. Nicht alle Aussagen sind (in der intuitionistischen Logik) entscheidbar, deshalb gibt es in Lean eine Typklasse `Decidable`, die für eine Aussage einen Beweis für die Wahrheit der Aussage oder den Nachweis des Widerspruchs braucht. Und `DecidableEq` ist die Typklasse für die Gleichheit von Termen, d.h. `a = b` ist für alle `a b: α` entscheidbar.

Wir befassen uns im Folgenden mit der Booleschen Gleichheit ==, weil für uns Lean als funktionale Sprache im Vordergrund steht.

Wir definieren eine Struktur `Person` und definieren die Gleichheit von Werten des Typs:

```
structure Person where
  name: String
  vorname: String

def Person.eq (p1 : Person) (p2 : Person) : Bool :=
  p1.name == p2.name && p1.vorname == p2.vorname

def p1 : Person := { name := "Curry", vorname := "Haskell"}
```

```
def p2 : Person := { name := "Curry", vorname := "Haskell"}
def p3 : Person := { name := "Moses", vorname := "Schönfinkel"}

#eval Person.eq p1 p2
#eval Person.eq p1 p3

#eval  p1 == p2 -- failed to synthesize BEq Person
```

Die Fehlermeldung macht uns darauf aufmerksam, dass wir den Operator == für den Typ `Person` nur verwenden können, wenn wir ihn auch für `Person` überladen haben. Tun wir das und auch gleich für die Typklasse `Repr`, damit wir Werte des Typs ausgeben können. (Für `Repr` haben wir schon im Kapitel 2 ein Beispiel gesehen.)

```
instance: BEq Person where
  beq := Person.eq

instance: Repr Person where
  reprPrec := λ x _ => x.vorname ++ " " ++ x.name

#eval p1         -- ergibt Haskell Curry

#eval p1 == p2   -- ergibt true
#eval p1 == p3   -- ergibt false
```

Beide Überladungen müssen wir nicht selbst programmieren. Lean stellt uns Implementierungen bereit, wenn wir bei der Definition des Typs angeben, dass die Überladungen automatisch erstellt werden sollen. Dies sieht man am Beispiel `Buch`:

```
structure Buch where
  titel: String
  autor: Person
deriving BEq, Repr

def b1 : Buch := {titel := "Theory of formal deducibility", autor := p1}
def b2 : Buch := {titel := "Theory of formal deducibility", autor := p2}
def b3 : Buch := {titel := "Theory of formal deducibility", autor := p3}

#eval b1
-- ergibt { titel := "Theory of formal deducibility", autor := Haskell Curry }

#eval b1 == b2  -- ergibt reu
#eval b2 == b3  -- ergibt false
```

6.3.2 Vergleichsoperatoren

Die Vergleichsoperatoren führen in Lean zu logischen Aussagen:

```
#check 41 < 42 -- hat Typ Prop
```

Es handelt sich dabei um *entscheidbare* Aussagen, was bedeutet dass sie wie Boolesche Funktionen verwendet werden können:

```
#eval 41 < 42 -- ergibt true
```

Wenn wir < im Quellcode von Lean verfolgen, finden wir in `Init.Notation`:

```
@[inherit_doc] infix:50 " < "  => LT.lt
macro_rules | `($x < $y)  => `(binrel% LT.lt $x $y)
```

Der Operator wird also durch die Funktion `LT.lt` der Typklasse `LT` realisiert. Diese wird in `Init.Prelude` so definiert:

```
class LT (α : Type u) where
  lt : α → α → Prop
```

Die Typklasse hat viele Instanzen, unter anderem

```
instance instLTNat : LT Nat where
  lt := Nat.lt
```

wobei `Nat.lt` definiert wird durch

```
protected inductive Nat.le (n : Nat) : Nat → Prop
  | refl     : Nat.le n n
  | step {m} : Nat.le n m → Nat.le n (succ m)

protected def Nat.lt (n m : Nat) : Prop :=
  Nat.le (succ n) m
```

Eine weitere Instanz der Typklasse `LT` ist

```
instance [LT α] : LT (List α) := ⟨List.lt⟩
```

wobei `List.lt` die lexikographische Ordnung auf Listen ist.

6.3.3 Arithmetische Operatoren

```
#check 41 + 1 -- Nat
#check 41 + (1.0/3 * 3) -- Float

#eval 41 + 1 -- 42
#eval 41 + (1.0/3 * 3) -- 42.00000
```

Die Addition funktioniert nicht nur mit identischen Typen, sondern auch mit heterogenen Typen. Das liegt an der Typklasse `HAdd`, die folgendermaßen definiert ist:

```
class HAdd (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hAdd : α → β → γ
```

Bei dieser Definition tritt ein neues Schlüsselwort auf: `outParam`. Normalerweise benötigt Lean für die Auflösung der Instanz der Typklasse alle Typen der Definition. Man kann die beiden ersten Typen als Input-Parameter sehen und γ als Output-Parameter. Durch die Kennzeichnung als `outParam` sagen wir, dass für die Suche der Instanz der Typklasse dieser Typ nicht bekannt sein muss, sondern sich aus der Instanz ergibt.

Wir probieren dies an einem eigenen Beispiel aus:

```
-- Typklasse ohne Outputparameter
class HPlus' (α : Type u) (β : Type v) (γ : Type w) where
  hPlus : α → β → γ

instance: HPlus' Nat Nat Nat where
  hPlus := Nat.add

#eval HPlus'.hPlus 39 3
-- ergibt:  typeclass instance problem is stuck, it is often due to
 ↪  metavariables
--          HPlus' Nat Nat ?m.3999

-- Typklasse mit Outputparameter
class HPlus (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hPlus : α → β → γ

instance: HPlus Nat Nat Nat where
  hPlus := Nat.add
```

```
#eval HPlus.hPlus 39 3

#eval HPlus.hPlus (43 : Int) (-1)
-- failed to synthesize HPlus Int ?m.4405 ?m.4407

instance: HPlus Int Int Int  where
  hPlus := Int.add

#eval HPlus.hPlus (43 : Int) (-1)
-- ergibt 42

-- Auch so etwas geht:
instance: HPlus Nat (Array Nat) (Array Nat) where
  hPlus n as := as.map (λ a => Nat.add n a)

#eval HPlus.hPlus 4 #[2, 3, 4]
-- #[6, 7, 8]
```

6.3.4 Hashing

Lean hat eine Typklasse `Hashable`, die so definiert ist:

```
class Hashable (α : Type) where
  hash : α → UInt64
```

Die Standardbibliothek hat eine Funktion `mixHash : UInt64 → UInt64 → UInt64` mit der man Hashes für verschiedene Konstruktoren induktiver Typen oder Felder von Strukturen zusammenbauen kann.

```
instance [Hashable String] : Hashable Person where
  hash p := mixHash (hash p.name) (hash p.vorname)

#eval Hashable.hash p1
#eval Hashable.hash p2

-- Werte von Person, die in Bezug auf `BEq` gleich sind, haben auch den
-- gleichen Hashwert
#eval Hashable.hash p1 == Hashable.hash p2 -- ergibt true
#eval Hashable.hash p1 == Hashable.hash p3 -- ergibt false
```

6.3.5 Inhabited

Die Typklasse `Inhabited α` definiert, dass der Typ α ein ausgezeichnetes Element namens `default` hat.

Definition der Typklasse:

```
class Inhabited (α : Sort u) where
  default : α
```

Wozu wird dies eingesetzt: Wenn Funktionen in Sondersituationen einen Wert zurückgeben sollen, auch wenn sie nicht korrekt aufgerufen wurden. Ein Beispiel dafür ist die Funktion `Array.get! arr i`, die mit einem Index `i` aufgerufen werden kann, der nicht im Bereich des Arrays liegt. In diesem Fall wird Lean eine Panik-Meldung (`PANIC at outOfBounds `) ausgeben, aber das Programm nicht beenden, sondern den Defaultwert von `Array`, der als `Array.empty` definiert ist zurückgeben.

Definition der Instanz für `Array α`:

```
instance : Inhabited (Array α) where
  default := Array.empty
```

```
-- Beispiel für PANIC
#def arr := #[1, 2]
#eval arr[0]!
#eval arr[1]!
-- #eval arr[2]!
```

6.3.6 Ord

Die Typklasse `Ord α` verlangt eine Vergleichsfunktion `compare` des Typs `compare: α → α → Ordering`. Dabei ist `Ordering` ein Summentyp mit `lt`, `eq` und `gt`.

Die Typklasse hat einen sogenannten „Derive-Handler“, d.h. für einen induktiven Datentyp oder eine Struktur führt die Anweisung `deriving Ord` dazu, dass Lean versucht automatisch eine Instanz von `Ord` zu definieren.

Als Beispiel betrachten wir, wie im Code von Lean die Instanz von `Ord` für die natürlichen Zahlen definiert ist:

```
instance : Ord Nat where
  compare x y := compareOfLessAndEq x y
```

Dabei ist `compareOfLessAnd Eq` folgendermaßen definiert:

```
def compareOfLessAndEq {α} (x y : α)
          [LT α] [Decidable (x < y)] [DecidableEq α] : Ordering :=
  if x < y then Ordering.lt
  else if x = y then Ordering.eq
  else Ordering.gt
```

6.3.7 ToString

Die Typklasse:

```
class ToString (α : Type u) where
  toString : α → String
```

Ein Beispiel einer Instanz:

```
protected def List.toString [ToString α] : List α → String
  | [] => "[]"
  | [x] => "[" ++ toString x ++ "]"
  | x::xs => xs.foldl (· ++ ", " ++ toString ·) ("[" ++ toString x) |>.push ']'

instance {α : Type u} [ToString α] : ToString (List α) :=
  ⟨List.toString⟩
```

```
-- Beispiele
#eval toString 12
#eval toString [1, 12, 42]
#eval toString (-12 : Int)
#eval toString true
#eval toString ()
#eval toString (() : Unit)
```

6.3.8 Automatisches Erzeugen von Instanzen für Standardklassen

Lean kann automatisch Instanzen für bestimmte Typklassen erstellen. Es funktioniert dies für die Typklassen

- `BEq`

- `Repr`
- `Hashable`
- `Ord`
- `Inhabitated`

Dies kann man auf zwei Arten tun. Man kann bei der Definition des Typs am Ende der Definition des Typ `deriving ...` anhängen. Oder man kann nachträglich Instanzen erzeugen lassen, wie in folgendem Beispiel:

```
deriving instance Hashable for Buch

#eval Hashable.hash b1
#eval Hashable.hash b2
#eval Hashable.hash b3
```

6.4 Typ-Konversionen

In der Programmierung sind Typkonversionen üblich, so kann in Java etwa ein `byte` an einer Stelle verwendet werden, an der ein `int` erwartet wird, ohne dass explizit eine Typumwandlung angegeben werden muss. In Lean wird dazu ein Mechanismus verwendet, der *coercions* genannt wird und der durch Typklassen gesteuert wird.

Als Beispiel definieren wir uns einen Typ von positiven natürlichen Zahlen:

```
inductive Pos : Type where
  | one : Pos
  | succ : Pos → Pos
deriving Repr

def Pos.toNat : Pos → Nat
  | Pos.one => 1
  | Pos.succ n => n.toNat + 1

-- Definition der Instanz von ofNat
instance : OfNat Pos (n + 1) where
  ofNat :=
    let rec natPlusOne : Nat → Pos
      | 0 => Pos.one
      | k + 1 => Pos.succ (natPlusOne k)
    natPlusOne n
```

```
#eval (5 : Pos)
#eval [1, 2, 3, 4].drop (2 : Pos) -- geht nicht

-- wir definieren die implizite Typumwandlung
instance : Coe Pos Nat where
  coe x := x.toNat

#eval [1, 2, 3, 4].drop (2 : Pos) -- ergibt [3, 4]
#check [1, 2, 3, 4].drop (2 : Pos)
```

Neben der Typklasse `Coe` gibt es auch `CoeSort` und `CoeFun`.

`CoeSort` ist im Prinzip wie `Coe`, nur ist das Ziel der Umwandlung ein *sort* in Lean, also ein Universum von Typen wie `Type` oder `Prop`. `CoeSort` wird gerne im Beweissystem verwendet, wo man zum Beispiel mit einer Gruppe zu tun hat und manchmal mit dem Ausdruck „Gruppe“ die *algebraische Struktur* meint, manchmal aber auch nur die *Menge* der Elemente der Gruppe. Mit `CoeSort` kann man Definitionen in dieser Situation vereinfachen. In Functional Programming in Lean 4.6 Coercions wird dies am Beispiel eines Monoids im Detail diskutiert.

Ein anderes Beispiel, das wir in der Programmierung immer wieder verwenden ist

```
instance : CoeSort Bool Prop where
  coe b := b = true
```

Hier wird der Boolesche Wert `b` umgewandelt in die logische Aussage `b = true`, also vom Typ `Prop`. Diese Typumwandlung macht es in Lean möglich, dass das `if` sowohl für Boolesche Werte als auch für logische Aussagen verwendet werden kann.

In der Programmierung mit Lean werden wir eher eine Typumwandlung von der Art `CoeFun` gelegentlich einsetzen. Hierbei wird ein Wert in eine Funktion gewandelt.

Als Beispiel nehmen wir wie im eben erwähnten Kapitel von Functional Programming in Lean eine Struktur:

```
structure Adder where
  howMuch : Nat

def addFive : Adder := ⟨5⟩

#eval addFive 42
-- ergibt function expected at
```

```
--           addFive
--         term has type
--           Adder

-- mit CoeFun

instance : CoeFun Adder (fun _ => Nat → Nat) where
  coe a := (· + a.howMuch)

#eval addFive 42
```

Übungen:

[06.1] Gegeben sei die Typklasse

```
class YesNo (α : Type u) where
  yesno : α → Bool
```

Definieren Sie Instanzen für `Bool`, `Nat`, `Int` und `Option α` mit vernünftigen Zuordnungen der Booleschen Werte. Verwenden Sie die Definitionen in Beispielen.

[06.2] Gegeben sei die Typklasse

```
class OrElse (α : Type u) where
  orElse  : α → (Unit → α) → α
```

Definieren sie eine Instanz für `Option α` und machen Sie ein paar Beispiele für die Verwendung der Instanz.

Zusatzfrage: Warum wird als zweites Argument von `orElse` `Unit → α` genommen und nicht α?

[06.3] Definieren Sie Instanzen der Typklasse `Inhabited` für den Produkttyp, den Summentyp und `List`. Beim Produkt und der Summe setzen Sie voraus, dass die beteiligten Typen Instanzen der Typklasse `Inhabited` sind.

Werten Sie Beispiele aus.

---

7 Logische Verifikation

7.1 Das Konzept zertifizierter Programmierung

In der funktionalen Programmierung verwenden wir, wo immer es geht, *reine* Funktionen, d.h. solche, deren Ausgabewerte ausschließlich durch die übergebenen Argumente bestimmt sind. (Das geht natürlich nicht immer, schließlich wollen wir mit der „Welt“ kommunizieren. Dazu mehr im nächsten Kapitel der Veranstaltung.)

Reine Funktionen sind softwaretechnisch von Vorteil: sie sind leichter zu verstehen und zu überprüfen, weil man nur sie selbst analysieren muss und nicht den Kontext, denn sie haben ja keine *Seiteneffekte*. Dies ist besonders wertvoll in verteilten und nebenläufigen Systemen. Mit der zunehmenden Relevanz solcher Systeme ist in der letzten Zeit auch das Interesse an funktionaler Programmierung wieder erwacht.

Es kommt aber noch ein weiterer Aspekt hinzu: mit einer Sprache wie Lean kann man die Korrektheit von reinen Funktionen *beweisen*. Lean ist eine funktionale Sprache und *zugleich* ein (interaktives) Beweissystem. Wenn man zum Beispiel die grundlegende Datei von Lean: `Init.Prelude.lean` betrachtet, dann enthält sie nicht nur Definitionen von Funktionen, sondern auch Beweise für die Eigenschaften dieser Funktionen. Man findet (Stand November 2024) etwas über 300 Definitionen in dieser Datei und knapp 70 Beweise. In der gesamten Codebasis von Lean dürfte der Anteil der Beweise noch viel größer sein.

Das Konzept *zertifizierter Programmierung* besteht darin, dass wir für unsere Programme nicht nur formale Spezifikationen zugrundelegen, sondern auch beweisen, dass sie diese Spezifikationen erfüllen, also *logisch verifizieren*.

Diese Veranstaltung konzentriert sich auf die funktionale Programmierung in Lean. Deshalb werden wir die logische Verifikation von Funktionen nicht in der ganzen Tiefe beleuchten können. In diesem Kapitel kommen Beispiele vor, die demonstrieren sollen, was das Konzept der zertifizierten Programmierung bedeutet.

Wir werden Eigenschaften von Funktionen für Listen beweisen (gewissermaßen als „Aufwärmung“), die Korrektheit eines Sortieralgorithmus zeigen und schließ-

lich am Beispiel des MergeSort vorstellen, wie man beweist, dass eine Funktion terminiert.

Alle Beweise werden wir im *Taktikmodus* von Lean ausführen und en passent die verwendeten Taktiken kurz erläutern. Die meisten der Taktiken werden auf der Seite von Kevin Buzzard: Tactics vorgestellt, man kann auf dieser Seite also wie in einer Art Kurzdokumentation nachschauen. Es ist unbedingt zu empfehlen, die Beweise selbst schrittweise in Visual Studio Code auszuführen und immer im Fenster *Lean Infoview* zu beobachten, welche Aussagen wir als Voraussetzungen verwenden dürfen und welches Beweisziel wir anstreben.

Genug der Vorrede.

7.2 Eigenschaften von Funktionen für Listen

Die erste Eigenschaft, die wir zeigen wollen, sagt aus, dass die leere Liste die Länge `0` hat.

```
lemma length_nil : List.length ([] : List α) = 0 := by
  constructor
```

Die Funktion `List.length` ist folgendermaßen definiert:

```
def List.length : List α → Nat
  | nil       => 0
  | cons _ as => HAdd.hAdd (length as) 1
```

Die Taktik `constructor` verwendet den ersten passenden Konstruktor eines induktiv definierten Typs, bei `List` in unserem Beweis also `nil` (`= []`), und man erhält damit, dass die Länge der leeren Liste in `List.length` als `0` definiert wurde.

Wir hätten auch Folgendes tun können:

```
example : List.length ([] : List α) = 0 := by
  rfl
```

Die Taktik `rfl` beweist Ziele der Form `⊢ x = y`, wenn `x` und `y` per Definition gleich sind (siehe Kevin Buzzard: Equality), was hier der Fall ist.

Folgendes Lemma sagt aus, dass eine Liste mit einem Element die Länge `1` hat. Beweisen Sie diese Aussage als Übung [07.1].

```
lemma length_singleton : List.length [a] = 1 := by
  sorry
```

Mit `sorry` kennzeichnen wir eine Beweisverpflichtung, die wir später noch einlösen wollen (und müssen). Wir können dann die Aussage zwar verwenden, aber sie wurde noch nicht bewiesen – und Lean weiß das!

`cons` *konstruiert* eine neue Liste aus einem Element und der bisherigen Liste. Listen „wachsen“ vorne, damit die bisherige Liste im Sinne des *structure sharings* ohne Kopieren für die neue Liste verwendet werden kann. Durch `cons` wird die Liste um 1 länger:

```
lemma length_cons :
  List.length (a :: as) = (List.length as) + 1 := by
    rfl
```

Wenn man zwei Listen aneinander hängt, dann addieren sich ihre Längen. Der Beweis dafür ist etwas aufwändiger:

```
lemma length_append (as bs : List α) :
  List.length (as ++ bs) = List.length as + List.length bs := by
  induction as with
  | nil => -- simp
    rw [length_nil]
    rw [Nat.zero_add]
    rewrite [List.nil_append]
    rfl
  | cons a as ih => -- simp [ih, Nat.succ_add]
    rw [List.cons_append]
    rw [length_cons]
    rw [ih]
    rw [length_cons]
    ac_rfl
```

In diesem Beweis haben wir die Taktik `induction` eingesetzt, die einen Induktionsbeweis über die beiden Konstruktoren `nil` und `cons` von `List` durchführt.

Die Taktik `rw [h]` setzt die (bereits bewiesene oder vorausgesetzte) Aussage `h` der Form `h : x = y`, also eine Gleichheitsaussage an allen passenden Stellen im Ziel ein. Zuerat verwenden wir als `h` das oben bewiesene Lemma `length_nil`. Danach das Theorem `Nat.zero_add`, das Lean bereits mitliefert. Danach `List.nil_append`.

Im Beweis werden die einzelnen Schritte durchgeführt, damit man sieht, was genau passiert. Wir sehen auch, dass `rw` nach einem `rewrite` automatisch `rfl` versucht.

Wie findet man bereits bewiesene Aussagen in Lean? Es gibt die Suchseite Loogle!. Geben wir dort zum Beispiel `"zero", "add"` ein, dann sind die beiden ersten Treffer

`Nat.add_zero` und `Nat.zero_add`. A propos: Man kann daran leicht erkennen, welche Namenskonvention in Lean für solche Aussagen verwendet wird und deshalb kann man oft auch erraten, wie die gesuchte Aussage wohl heißen wird.

Im Beweis für den zweiten Konstruktor `cons` von `List` nennen wir die Induktionshypothese `ih` und wir müssen den Beweis so konstruieren, dass wir sie anwenden können. Dazu verwenden wir `List.cons_append` sowie `length_cons`. Dann sind wir fast fertig.

Die Taktik `ac_rfl` ist wie `rfl` jedoch zusätzlich modulo Assoziativ- und Kommutativgesetz. Sie erspart uns also mühseliges Anwenden von `Nat.add_assoc` und `Nat.add_comm`.

Es folgt die kurze Variante des Beweises, die die Taktik `simp` einsetzt:

```
example:  List.length (as ++ bs) = List.length as + List.length bs := by
  induction as with
  | nil => simp
  | cons _ as ih =>
    simp [ih]
    ac_rfl
```

Die Taktik `simp` (für *simplifier*) verwendet eine Datenbank von bewiesenen Aussagen und versucht das Ziel (oder mit `simp at h` auch die Voraussetzung `h`) durch Einsetzen dieser Aussagen zu vereinfachen. Man kann selbst eigene Aussagen dieser Datenbank hinzufügen, indem man das Lemma oder Theorem mit dem Attribut `@simp` versieht (Mehr über `simp`).

```
lemma length_concat (as : List α) (a : α) :
  List.length (List.concat as a) = List.length as + 1 := by
  cases as with
  | nil => rfl
  | cons a as => simp [List.concat]
```

Die Taktik `cases` zerlegt das Beweisziel in Teilziele pro Konstruktor des Typs des Terms, hier für die beiden Konstruktoren von `List`. `cases` ist ähnlich zu `induction`, erstellt aber keine Induktionsvoraussetzung.

Nun wollen wir noch einige Eigenschaften von `map` beweisen. Die Funktion `List.map` wendet eine unäre Funktion auf jedes Element einer Liste an:

```
def map (f : α → β) : List α → List β
  | []    => []
  | a :: as => f a :: map f as
```

Wenn wir mittels der Identitätsfunktion „mappen“, dann erhalten wir die Eingabeliste als Ergebnis:

```
lemma map_id {α : Type} (xs : List α) :
  List.map (fun x => x) xs = xs := by
    cases xs
    . rfl
    . simp [List.map]
```

Beweisen Sie als Übungsaufgabe [07.2], dass `map` und `append` vertauschbar sind:

```
lemma map_append {α β : Type} (f : α → β) (xs ys : List α) :
  List.map f (xs ++ ys) = List.map f xs ++ List.map f ys := by
  sorry
```

In Abschnitt 7.4 dieses Kapitels werden wir uns mit dem MergeSort befassen. Wir werden die Funktion `mergeSort` entwickeln und wollen dann beweisen, dass diese Funktion terminiert. Dafür werden iwir einige Aussagen über Listen benötigen. Diese Aussagen wollen wir nun im Vorgriff betrachten.

Die Funktion `split` teilt eine Liste in der Mitte. Sie verwendet dazu die Listenfunktionen `take` und `drop`, die wir schon in Kapitel 4 kennengelernt haben.

```
def split (xs : List α) : List α × List α :=
  let m := (xs.length + 1)/2
  (xs.take m, xs.drop m)

#eval [1, 2, 3, 4].take 2
#eval [1, 2, 3, 4].drop 2

#eval [1, 2, 3, 4].split
#eval [1, 2, 3, 4, 5].split
#eval [1, 2, 3, 4, 5, 6].split
#eval [1].split
#eval ([] : List Nat).split
```

Wir wollen nun beweisen, dass beide Teile nach dem Teilen durch `split` kleiner sind als die Liste selbst, vorausgesetzt, sie hatte mindestens 2 Elemente.

Dazu beweisen wir zunächst zwei Aussagen über `(n + 1) / 2`:

```
lemma split_lt_n (n : ℕ) : n ≥ 2 → (n + 1) / 2 < n := by
  cases n
  . trivial
  . intro h
```

```
    rw [Nat.div_lt_iff_lt_mul]
    . rw [Nat.add_mul]
      simp
      simp at h
      linarith
    . trivial
```

Im Fall `n = 0` können wir die Taktik `trivial` anwenden, denn die Voraussetzung ist gar nicht erfüllt. `trivial` „erkennt“ diesen Widerspruch in den Voraussetzungen.

Im Fall `cons` führt `intro h` die Voraussetzung `n + 1 ≥ 2` als Voraussetzung (*hypotheses*) ein. Nun verwenden wir die Aussage

```
#check Nat.div_lt_iff_lt_mul
```

wodurch wir zwei Beweisverpflichtung erhalten, nämlich die Voraussetzungen von `Nat.div_lt_iff_lt_mul`:

```
n + 1 + 1 < (n + 1) * 2
0 < 2
```

Die zweite ist trivial, für die erste setzen wir das mächtige Geschütz der Taktik `linarith` ein, die lineare Gleichungen und Ungleichungen für die natürlichen Zahlen oft lösen kann.

Nun beweisen wir die zweite Aussage über `(n + 1) / 2`:

```
lemma split_gt_zero (n : ℕ) : n ≥ 2 → 0 < (n + 1) / 2 := by
  cases n
  . simp
  . rw [Nat.add_assoc]
    simp
```

Mit Hilfe dieser beiden Lemmata können wir beweisen, dass beim Aufspalten der Liste die Länge abnimmt.

```
lemma split_length_fst {xs : List α} :
  xs.length ≥ 2 → (split xs).fst.length < xs.length := by
  intro h
  dsimp [split]
  have hl : (xs.length + 1) / 2 < xs.length := by
    apply split_lt_n
    apply h
  rw [List.length_take_of_le]
  . exact hl
  . apply Nat.le_of_lt
```

```
    exact hl

lemma split_length_snd {xs : List α} :
  xs.length ≥ 2 → (split xs).snd.length < xs.length := by
  intro h
  dsimp [split]
  have hl : (xs.length + 1) / 2 < xs.length := by
    apply split_lt_n
    apply h
  rw [List.length_drop]
  simp [hl]
  apply And.intro
  . linarith
  . apply split_gt_zero
    apply h
```

In diesen beiden Beweisen kommen folgende Taktiken vor, die wir bisher nicht kennengelernt haben:

- `dsimp` wird in der Doku so erklärt:

> The `dsimp` tactic is the definitional simplifier. It is similar to simp but only applies theorems that hold by reflexivity. Thus, the result is guaranteed to be definitionally equal to the input.

- `have` führt eine neue Aussage ein, zusammen mit einer Beweisverpflichtung dafür. Erbringt man den Beweis, dann wird die Aussage in den Stack der Voraussetzungen übernommen, die man im Folgenden verwenden kann.
- `exact` kann man verwenden, wenn das Beweisziel exakt mit einer Voraussetzung übereinstimmt.
- `apply` wendet ein Funktion an. Zum Beispiel ist `split_lt_n (n : ℕ) : n ≥ 2 → (n + 1) / 2 < n` eine Funktion mit der Aussage `x ≥ 2` als Argument. Deshalb können wir sie auf das Ziel anwenden.

Damit beenden wir den Abschnitt über Listen. Die beiden zuletzt bewiesenen Aussagen werden später beim MergeSort verwendet.

Übungen:

[07.1] Beweisen Sie das Lemma aus der Vorlesung:

```
lemma length_singleton : List.length [a] = 1 := by
  sorry
```

[07.2] Beweisen Sie das Lemma aus der Vorlesung:

```
lemma map_append {α β : Type} (f : α → β) (xs ys : List α) :
  List.map f (xs ++ ys) = List.map f xs ++ List.map f ys := by
  sorry
```

7.3 Verifizierung des InsertionSorts

Der InsertionSort (Sortieren durch Einfügen) sortiert eine Folge natürlicher Zahlen, indem er sukzessive der Eingabe eine Zahl entnimmt und sie an der richtigen Stelle der Ausgabefolge einsortiert.

Der Algorithmus hat quadratische Laufzeit (bezüglich der Anzahl der zu sortierenden Zahlen), kann aber für kleine Folgen schneller sein als andere Sortieralgorithmen.

Wir wollen diesen Algorithmus implementieren und seine Korrektheit beweisen. Dabei folgen wir der Diskussion des Algorithmus in Software Foundations Volume 3: Verified functional Algorithms.

Zunächst implementieren wir den Algorithmus:

```
namespace InsertionSort

def insert (i : Nat) (l : List Nat) :=
  match l with
  | [] => [i]
  | h :: t => if i ≤ h then i :: h :: t else h :: insert i t

def sort (l : List Nat) : List Nat :=
  match l with
  | [] => []
  | h :: t => insert h (sort t)

#eval sort [3, 5, 1, 7, 1, 25, 4] -- [1, 1, 3, 4, 5, 7, 25]
def listPi := [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
#eval sort listPi
```

Bemerkung zum Algorithmus:

Die Funktion `insert` verwendet den Ausdruck `if ... then ... else ...`. Man kann in Lean eine logische Aussage (aus dem Universum `Prop`) für die Entscheidung verwenden. In unserem Beispiel ≤

```
#check 1 ≤ 3 -- 1 ≤ 3 : Prop
```

Das ist jedoch recht problematisch. Könnte man hier eine beliebige Aussage einsetzen, dann könnten wir ja jede beliebige Aussage der Prädikatenlogik entscheiden. Dies ist aber nicht möglich! Wieso erlaubt dann Lean, dass wir mit ≤ einen Ausdruck aus der Welt von `Prop` einsetzen dürfen, wo doch eigentlich ein Ausdruck aus der Welt von `Bool` hingehört? (Wir erinnern uns an Teil 1 und 6 der Vorlesung, wo wir erfahren haben, dass es in Lean zwei Vergleichsoperatoren gibt: = in der Welt von `Prop` und == in der Welt von `Bool`.) Lean erlaubt hier nur solche logischen Aussagen, die *entscheidbar* sind, was über die Typklasse `Decidable` geregelt wird. Aus der Dokumentation:

> Decidable p is a data-carrying class that supplies a proof that p is either true or false. It is equivalent to Bool (and in fact it has the same code generation as Bool) together with a proof that the Bool is true iff p is.

> Decidable instances are used to infer "computation strategies" for propositions, so that you can have the convenience of writing propositions inside if statements and executing them (which actually executes the inferred decidability instance instead of the proposition, which has no code).

Da ≤ entscheidbar ist, können wir den Vergleichsoperator in der Definition von `insert` unbeschadet übernehmen.

Es gäbe noch eine Alternative: `Nat.ble`, den Booleschen Vergleichsoperator

```
#check Nat.ble 1 2 -- Nat.ble 1 2 : Bool
```

Wir könnten diesen Operator für die Definition von `insert` verwenden. Es ist in Lean jedoch durchaus üblich, entscheidbare Prädikate in `if ... then ...else ...` zu verwenden.

Zurück zum eigentlichen Thema. Was müssen wir tun, um diesen Algorithmus zu verifizieren?

1. Wir müssen zeigen, dass er terminiert. Das ist in diesem Beispiel keine große Sache, denn bei der Rekursion wird die Funktion `sort` auf einer kleineren Liste aufgerufen. In diesem Fall ermittelt Lean automatisch, dass die Rekursion terminiert. Wir werden am Beispiel des MergeSort sehen, dass Lean dies nicht immer automatisch kann, dann müssen wir das Terminieren des Algorithmus beweisen.
2. Die Ausgabeliste muss eine Permutation der Eingabeliste sein, d.h. alle Zahlen der Eingabe müssen in ihr vorkommen — und nur diese.
3. Und natürlich: die Ausgabeliste muss aufsteigend sortiert sein.

Wie definieren wir in Lean die Eigenschaft der Liste sortiert zu sein?

```
inductive sorted : List Nat → Prop :=
  | sorted_nil : sorted []
  | sorted_singleton : ∀ x, sorted [x]
  | sorted_cons : ∀ x y l, x ≤ y → sorted (y :: l) → sorted (x :: y :: l)
```

Diese Art von Definition nennt man ein *induktives Prädikat*. Es bedeutet:

1. Die leere Liste ist sortiert
2. Jede Liste, die nur ein Element hat, ist sortiert
3. Je zwei aufeinander folgende Elemente sind in der richtigen Reihenfolge in der Liste

und außerdem: jede andere Liste ist nicht sortiert.

Es gibt natürlich noch andere Möglichkeiten ein Prädikat zu definieren, das aussagt, dass eine Liste `l` sortiert ist.

Man könnte etwa formulieren, dass für alle `i` und `j` (< der Länge der Liste) gilt: `l[i] ≤ l[j]`.

Oder man könnte die Boolesche Funktion `isSorted` verwenden, die wir in Übung [04.3] entwickelt haben.

Ein induktives Prädikat ist eine Funktion `... → Prop`, die alles als zutreffend charakterisiert, was sich aus den angegebenen Konstruktoren ableiten lässt und *nur das*. Es eignet sich besonders gut, um Beweise auszuführen.

Wir sehen uns zuerst ein Paar Beispiele an:

```
example : sorted [1, 2, 3] := by
    constructor
    decide
    constructor
    decide
    constructor
```

Wenn das Ziel ein induktives Prädikat ist, dann versucht `constructor` es mit einem der Konstruktoren zu lösen.

`decide` kann Ungleichheiten beweisen, vorausgesetzt, das Ziel hat keine lokalen Variablen oder Metavariablen. In unserem Beispiel würde auch `simp` funktionieren.

Man sieht: wir rollen den Beweis rekursiv auf. Das geht auch kürzer:

```
example : sorted [1,2,3] := by
    repeat (constructor; decide)
    constructor
```

Wie sieht es mit einem negativen Fall aus?

```
example : ¬ sorted [2, 1] := by
  intro h
  cases h with
  | sorted_cons _ _ _ contra => simp at contra
```

In diesem Beispiel verwenden wir die Taktik `cases`, die für einen induktiven Typ Beweisziele für jeden Konstruktor erzeugt. In unserem Fall wird jedoch kein Beweisziel für `sorted_nil` und `sorted_singleton` erzeugt. Die Taktik `cases` ist so clever, dass sie unmögliche Fälle entdeckt und gleich schließt.

Nun folgt der Beweis, dass die Ausgabe des Algorithmus sortiert ist. Der Beweis wird in zwei Schritten ausgeführt: Zuerst zeigen wir, dass das Einsortieren eines Elements in eine Liste eine sortierte Liste ergibt. Danach dass der Algorithmus sortiert.

Zum Verstehen ist es am besten, den Beweis im *Lean Infoview* mitzuverfolgen.

Welche neuen Taktiken benötigen wir?

`cases Classical.em (a ≤ x)`:[1] Das Axiom `Classical.em` ist der sogenannte „Satz vom ausgeschlossenen Dritten“. Er erweitert unsere bisherige Logik, denn er ist nur in der klassischen Logik gültig und sagt an unserem Beispiel aus, dass `a ≤`

[1] In diesem Beispiel steht hinter `cases` ein zusammengesetzter Ausdruck. Die Fälle, die `cases` hier erstellt, sind die Ergebnisse dieses Ausdrucks. Hier werden also aus dem Oder in `a ≤ x ∨ ¬ a ≤ x` die beiden Fälle `a ≤ x` sowie `¬ a ≤ x`.

`x ∨ ¬ a ≤ x` gilt. Mit `cases` wird gleich die Elimination des logischen Oder durch Fallunterscheidung begonnen: Wir nehmen einmal den Fall an, die linke Seite des Oder (`inl`) gilt und dann nehmen wir den Fall an, dass die rechte Seite des Oder (`inr`) gilt. Für beide Fälle müssen wir das Beweisziel zeigen.

Dabei verwenden wir bereits in der Bibliothek von Lean bewiesene Aussagen:

```
not_le.{u} {α : Type u} [LinearOrder α] {a b : α} : ¬a ≤ b ↔ b < a
le_of_lt.{u} {α : Type u} [Preorder α] {a b : α} (hab : a < b) : a ≤ b
```

Außerdem setzen wir `unfold` ein, was die Definition eines Terms „entfaltet“.

```
#check not_le
#check le_of_lt

lemma insert_sorted: ∀ a l, sorted l → sorted (insert a l) := by
  intro a l s_l
  induction s_l with
  | sorted_nil =>
    unfold insert
    constructor
  | sorted_singleton x =>
    unfold insert
    cases Classical.em (a ≤ x) with
    | inl a_le_x =>
      simp only [a_le_x]
      constructor
      exact a_le_x
      constructor
    | inr not_a_le_x =>
      simp only [not_a_le_x]
      rw [not_le] at not_a_le_x
      constructor
      . exact Nat.le_of_lt not_a_le_x
      . constructor
  | sorted_cons x y t x_le_y s_y_t ih =>
    unfold insert
    cases Classical.em (a ≤ x) with
    | inl a_le_x =>
      simp only [a_le_x]
      constructor
      . exact a_le_x
      . constructor
```

```
        . exact x_le_y
        . exact s_y_t
    | inr not_a_le_x =>
      simp only [not_a_le_x]
      unfold insert
      cases Classical.em (a ≤ y) with
      | inl a_le_y =>
        simp only [a_le_y]
        rw [not_le] at not_a_le_x
        constructor
        . exact le_of_lt not_a_le_x
        . unfold insert at ih
          simp only [a_le_y] at ih
          exact ih
      | inr not_a_le_y =>
        simp only [not_a_le_y]
        constructor
        . exact x_le_y
        . unfold insert at ih
          simp only [not_a_le_y] at ih
          exact ih
```

Uff!!

Jetzt ist es aber recht einfach, zu zeigen, dass unser Algorithmus tatsächlich sortiert:

```
theorem sort_sorted: ∀ l, sorted (sort l) := by
    intro l
    induction l with
    | nil =>
      unfold sort
      constructor
    | cons a t ih =>
      unfold sort
      apply insert_sorted
      apply ih
```

Bemerkung zum Beweis:

Weil wir für die Definition des Einfügens im `if ... then ... else ...` den Vergleichsoperator ≤ vom Typ `α → α → Prop` verwendet haben, bot es sich an, für die

Fallunterscheidung im `if` das Axiom `Classical.em` zu verwenden. Und wenn man den Hitchhiker's Guide to Logical Verification S.71 zu Rate zieht, ist das auch angemessen.

Natürlich setzt man dabei ein starkes Geschütz ein, aber andererseits ist der Satz vom ausgeschlossenen Dritten in diesem Fall recht offensichtlich. Aber: wir haben die Welt von `Bool` verlassen, in der klar ist, dass es nur die Fälle `true` und `false` geben kann.

Wie könnte man vorgehen, ohne `Classical.em` einzusetzen? Man kann die beiden Fälle im `if` mit `Nat.bel a x` einführen und dann die Beweisziele für die Fälle `true` und `false` anstreben. Dabei helfen eine Menge von Aussagen in der Bibliothek, die für die benötigten Vergleichsoperatoren die Äquivalenz der Welt von `Bool`und `Prop` beweisen, etwa `Nat.le_of_ble_eq_true`.

Aber: unser Beweis mit `Classical.em` verwendet nur die Eigenschaften der Totalordnung. Er kann also verallgemeinert werden für alle Arten von Listen, die eine solche Ordnung haben.

Man könnte nun denken, wir sind mit der Verifikation des Algorithmus fertig. Aber es fehlt noch ein wichtiger Teil. Bisher haben wir nur beweisen, dass die Ausgabeliste sortiert ist. Das wäre aber auch der Fall, wenn sie einfach die leere Liste wäre.

Wir müssen also auch noch zeigen, dass alle Elemente der Eingabeliste in der Ausgabeliste vorkommen, kurz: dass die Ausgabeliste eine *Permutation* der Eingabeliste ist.

Dazu verwenden wir das induktive Prädikat für die Permutation von Listen in Lean:

```
#print List.Perm
```

Aus der Doku:

> `Perm l₁ l₂` or `l₁ ~ l₂` asserts that `l₁` and `l₂` are permutations of each other. This is defined by induction using pairwise swaps.

```
inductive Perm : List α → List α → Prop
  | nil : Perm [] []
  | cons (x : α) {l₁ l₂ : List α} : Perm l₁ l₂ → Perm (x :: l₁) (x :: l₂)
  | swap (x y : α) (l : List α) : Perm (y :: x :: l) (x :: y :: l)
  | trans {l₁ l₂ l₃ : List α} : Perm l₁ l₂ → Perm l₂ l₃ → Perm l₁ l₃
```

Also:

1. Die leere Liste ist eine Permutation der leeren Liste.
2. Kommt ein Element x zu zwei modulo Permutation identischen Listen dazu, dann bleiben sie Permutationen.
3. Vertauscht man die beiden vorderen Elemente einer Liste, dann hat man sie permutiert.
4. Das Permutieren von Listen ist transitiv.

Damit wird unsere Spezifikation der Korrektheit des InsertionSort zu:

```
def insertionSort_korrekt (sort: List Nat → List Nat) :=
  ∀ l, List.Perm l (sort l) ∧ sorted (sort l)
```

Wir beweisen wieder zuerst ein Lemma für `insert`:

```
theorem insert_perm: ∀ x l, List.Perm (x :: l) (insert x l) := by
  intro x l
  induction l with
  | nil =>
    unfold insert
    repeat (constructor)
  | cons head tail ih =>
    unfold insert
    cases Classical.em (x ≤ head) with
    | inl x_le_h =>
      simp only [x_le_h]
    | inr not_x_le_h =>
      simp only [not_x_le_h]
      have h1 : List.Perm (x :: head :: tail) (head :: x :: tail) := by
        apply List.Perm.swap
      have h2 : List.Perm (head :: x :: tail) (head :: insert x tail) := by
        apply List.Perm.cons
        apply ih
      apply List.Perm.trans h1 h2
```

Und nun der Beweis, dass unser Algorithmus tatsächlich die Eingabeliste permutiert:

```
theorem sort_perm: ∀ l, List.Perm l (sort l) := by
  intro l
  induction l with
  | nil =>
    unfold sort
```

```
    constructor
  | cons head tail ih =>
    unfold sort
    have h1 : List.Perm (head :: (sort tail)) (insert head (sort tail)) := by
      apply insert_perm
    have h2 : List.Perm (head :: tail) (head :: sort tail) := by
      apply List.Perm.cons
      apply ih
    apply List.Perm.trans h2 h1
```

Damit sind wir am Ziel:

```
theorem sort_korrekt : insertionSort_korrekt sort := by
  unfold insertionSort_korrekt
  intro l
  exact ⟨sort_perm l, sort_sorted l⟩
  -- kurz für
  -- apply And.intro (sort_perm l) (sort_sorted l)
```

Neu verwendet werden hier die *constructor brackets* `⟨` und `⟩`, die man durch die Tastenfolge `\<` bzw. `\>` eingibt. Wir müssen ein logisches Und einführen. Ein Beweis für `And` ist ein Paar von Beweisen: einer für die linke und einer für die rechte Seite. Der Konstruktor hat also genau diese beiden Argumente und kann mit der *anonymous constructor notation* aufgerufen werden.

```
end InsertionSort
```

Übungen:

[07.3] Schreiben Sie eine geradlinig rekursive Funktion `sum0n` und eine endrekursive Funktion `sum0nTR`, die die natürlichen Zahlen von `0` bis `n` aufaddiert.

[07.4] Über Carl Friedrich Gauß wird erzählt: „Gauß' Lehrer ließ die Schüler die Zahlen von 1 bis 100 addieren. Während nun seine Mitschüler fleißig zu addieren begannen, stellte Gauß fest, dass sich die 100 zu addierenden Zahlen zu 50 Paaren gruppieren lassen, die jeweils die Summe `101` haben: `1 + 100`, `2 + 99`, … `50 + 51`. Also musste das gesuchte Ergebnis gleich dem Produkt `50 · 101 = 5050` sein." (Wikipedia)

Definieren Sie eine Funktion `littleGauss`, die diese Berechnung ausführt.

[07.5] Beweisen Sie, dass die beiden Varianten `sum0n` und `sum0nTR` der Summenformel für alle natürlichen Zahlen denselben Funktionswert ergeben.

[07.6] Beweisen Sie, dass „der kleine Gauß“ für alle natürlichen Zahlen denselben Funktionswert hat wie die Funktion `sum0n`.

[07.7] Verwenden Sie die in Übung [04.1a] definierte Funktion `mySum` und die in Übung [07.3] definierte Funktion `sum0n` und zeigen Sie:

```
example: mySum [0, 1, 2, 3] = sum0n 3 := by
```

7.4 MergeSort: Beweis des Terminierens

In diesem Abschnitt geht es um einen weiteren Sortieralgorithmus: den MergeSort. Die Idee ist einfach: Angenommen wir haben eine Liste in zwei Hälften geteilt und jede ist bereits sortiert, dann müssen wir sie nur noch sortiert mischen.

Zunächst definieren wir `merge`, zuständig für das Mischen, und dann die rekursive Sortierfunktion `mergeSort`. Für die Sortierung geben wir eine Vergleichsfunktion `comp` mit.

```
namespace MergeSort

def merge (xs ys : List Nat) : List Nat :=
  match xs, ys with
  | [], ys => ys
  | xs, [] => xs
  | x :: xs, y :: ys =>
    if x ≤ y then
      x :: merge xs (y :: ys)
    else
      y :: merge (x :: xs) ys
```

Wenn wir nun die Sortierfunktion selbst definieren:

```
def mergeSortv1 (xs : List Nat) : List Nat :=
  match xs with
  | []  => []
  | [a] => [a]
  | _   => let listPair := split xs
           merge (mergeSortv1 listPair.fst) (mergeSortv1 listPair.snd)
```

Dann erhalten wir folgende Fehlermeldung:

```
fail to show termination for
  MergeSort.mergeSortv1
with errors
failed to infer structural recursion:
Cannot use parameter xs:
  failed to eliminate recursive application
    mergeSortv1 listPair.1

failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded relation
  - Use `decreasing_by` to specify your own tactic for discharging this kind of
↪  goal

xs x✝ : List ℕ
listPair : List ℕ × List ℕ := split xs
⊢ sizeOf (split xs).1 < sizeOf x✝
```

Im Unterschied zum InsertionSort kann Lean in diesem Fall nicht selbst herleiten, dass die Rekursion terminiert.

Was tun?

- Eine Möglichkeit besteht darin, die Funktion als `partial` zu definieren. Mit dem Schlüsselwort `partial` schalten wir die Überprüfung aus, ob die Funktion terminiert. Wie wir an `mergeSort'` sehen, können wir dann die Funktion aufrufen.
- Besser ist es natürlich, dass wir beweisen, dass die Funktion terminiert. Warum ist das der Fall? In der Rekursion rufen wir die Funktion jeweils mit einer Teilliste auf. Beide Teillisten sind kürzer als die Liste selbst, also nähert sich der Aufruf sukzessive dem Fall `[a]` bzw `[]`: die Rekursion terminiert.

```
partial def mergeSort' (xs : List Nat) : List Nat :=
  match xs with
  | []  => []
  | [a] => [a]
  | _   => let listPair := split xs
           merge (mergeSort' listPair.fst) (mergeSort' listPair.snd)

#eval mergeSort' [1, 2, 4, 5]
#eval mergeSort' [6, 2, 4, 5]
#eval mergeSort' [6, 2, 7, 5]
```

Die Fehlermeldung hat uns einen Hinweis gegeben, dass wir `terminating_by` ver-

wenden könnten, um das Terminieren der Funktion zu beweisen. Wir müssen eine *well_founded relation* angeben. Worum geht es dabei?

In der Doku steht über eine Relation `r : α → α → Prop`:

> A relation `r` is `WellFounded` if all elements of `α` are accessible within `r`. If a relation is `WellFounded`, it does not allow for an infinite descent along the relation.

> If the arguments of the recursive calls in a function definition decrease according to a well founded relation, then the function terminates. Well-founded relations are sometimes called *Artinian* or said to satisfy the "descending chain condition".

Nun erfüllt < für die natürlichen Zahlen offensichtlich die *descending chain condition*, und in der Tat ist unsere Überlegung ja, dass die Rekursion terminiert, weil die Länge der übergebenen Listen immer kleiner wird.

Mit diesen Überlegungen können wir in der Definition von `mergeSort` selbst den Beweis führen, dass `xs.length`, die Länge der Liste, bei jedem rekursiven Aufruf kleiner wird. Im Abschnitt dieses Kapitels über Eigenschaften von Listen haben wir schon vorgearbeitet und die Lemmata `split_length_fst` und `split_length_snd` bewiesen. Diese können wir nun verwenden.

```
def mergeSort (xs : List Nat) : List Nat :=
  match xs  with
  | []  => []
  | [a] => [a]
  | a :: b :: rest =>
    let xs := a :: b :: rest
    let listPair := split xs
    have xs_length_2 : xs.length ≥ 2 := by
      simp [List.length]
    have : listPair.fst.length < xs.length := by
      apply split_length_fst xs_length_2
    have : listPair.snd.length < xs.length := by
      apply split_length_snd xs_length_2
    merge (mergeSort listPair.fst) (mergeSort listPair.snd)
termination_by xs.length

#eval mergeSort [1, 2, 4, 5]
#eval mergeSort [6, 2, 4, 5]
```

```
#eval mergeSort [6, 2, 7, 5]
#eval mergeSort [42, 6, 2, 7, 5]

end MergeSort
```

Um die Korrektheit von `mergeSort` zu zeigen, gilt es wie beim InsertionSort zu zeigen, dass der Algorithmus die Eingabe permutiert und sortiert.

Bei jedem Aufteilen der Liste fällt kein Element weg, wie man sich leicht überlegen kann, da `split` mittels `take` und `drop` definiert wurde. Und dass das Ergebnis sortiert ist, liegt an der Definition von `merge`.

Aber natürlich sind wir nicht mit dieser Argumentation ganz zufrieden, wir wollen sie gerne in Lean selbst formulieren – und damit durch Lean auch überprüfen. Im Code von Lean findet sich der Beweis für beide Punkte und wir überlassen es als Übung, sich mit diesen beiden Beweisen zu befassen.

Übungen:

[07.8] Analysieren Sie den Code in `Init.Data.List.Sort.Lemmas` und versuchen Sie die Beweisführung in `theorem List.mergeSort_perm` und `theorem List.sorted_mergeSort` nachzuvollziehen.

[07.9] Erläutern Sie, wie in Lean Manual – Example Palindromes die Korrektheit der Funktion `isPalindrome` bewiesen wird.

7.5 Zusammenfassung

In diesem Kapitel haben wir einen ersten Blick auf **logische Verifikation** mittels Lean geworfen: Lean ist nicht nur eine funktionale Programmiersprache, sondern auch ein interaktives Beweissystem und man kann deshalb das Terminieren und die Korrektheit von Funktionen *beweisen*.

- Wir haben zunächst ein paar Eigenschaften von Listenfunktionen bewiesen,
- und uns dann mit dem InsertionSort beschäftigt und seine Korrektheit gezeigt.
- Im letzten Abschnitt des Kapitels wurde am Beispiel des MergeSorts das Terminieren dieses Algorithmus in Lean bewiesen.

8 Programmierung mit Effekten

8.1 Das Konzept der Monade

Funktionale Programmierung und Lean im Speziellen basiert auf

- referenzieller Transparenz: Reine Funktionen ergeben für dieselben Argumente immer das gleiche Ergebnis.
- Komposition von Funktionen: Funktionen können zu komplexeren Funktionen kombiniert werden.
- Funktionen höherer Ordnung: Funktionen können selbst Argumente und/oder Resultate von Funktionen sein.
- Typsystem: Jeder Ausdruck der Sprache hat einen exakten Typ.

Mit diesen Prinzipien allein kann man keine Programme schreiben, denn keine sogenannten Seiteneffekte wären möglich. Seiteneffekte bestehen darin, dass über die Berechnung eines Werts hinaus der Zustand unseres Programms verändert wird. Beispiele für Seiteneffekte sind Ein- und Ausgaben oder das Werfen einer Exception. Solche Effekte muss es aber in einem Programm geben, sonst könnte es nicht mit seiner Aufrufumgebung kommunizieren. Lean (inspiriert von Haskell) kennt deshalb ein Konzept, das Seiteneffekte durch spezielle Typkonstruktoren ausdrückt und somit im Typsystem explizit unterscheidet zwischen Ausdrücken mit und ohne Seiteneffekte.

Reine Funktionen in Lean haben eine Signatur wie zum Beispiel

```
def double : Nat → Nat := λ n => n + n

#eval double 21   -- ergibt 42
#eval double 21.0 -- Fehler
```

Reine Funktionen können zusammengebaut werden, vorausgesetzt der Typ des Ergebnisses der einen ist der Typ des Arguments der anderen.

In Lean ist die Komposition von Funktionen so definiert:

```
def Function.comp {α : Type} {β : Type} {γ : Type}
  (f : β → γ) (g : α → β) : α → γ :=
    fun x => f (g x)
```

das heißt bei der Komposition von `f` mit `g` wird erst `g` und dann `f` ausgeführt. Man sagt auch `f ∘ g` ist „f *nach* g“

```
def add7: Nat → Nat := λ n => n + 7

#eval double 1 -- ergibt 2
#eval add7 1   -- ergibt 8

def doubadd: Nat → Nat := double ∘ add7
def doubadd2 (n : Nat) : Nat := (n + 7) * 2

#eval doubadd 1
#eval doubadd2 1
#eval Function.comp double add7 1
-- f ∘ g ist infix für Function.comp f g
```

8.1.1 Funktionen mit einem gewissen Extra

Wir brauchen aber auch Funktionen, die Seiteneffekte machen oder bei denen spezielle Situationen auftreten.

Wir nehmen als Beispiele:

- Lesen bzw. Schreiben eines Strings vom bzw auf das Terminal wie `getLine` oder `putStr`
- Fehlende Werte für eine Funktion, wie zum Beispiel die Funktion `List.head?` , die das erste Element einer beliebigen Liste zurückgeben soll.

Lean macht die Besonderheit solcher Funktionen explizit, in dem sie Werte eines Typs zurückgeben, der dieses „gewisse Extra“, also den Seiteneffekt oder die Möglichkeit des fehlenden Werts „markiert“.

Die Signaturen für unsere Beispiele sind:

```
getLine : IO String
putStr  : String → IO Unit
List.head? : List α → Option α
```

Der Typ des Ergebnisses dieser Funktionen ist markiert:

- `getLine` hat als Typ nicht einfach `String`, sondern `IO String` und `putStr` hat den Returntyp `IO Unit`. Diese mit dem Typkonstruktor `IO` gebildeten Typen geben also explizit an, dass die beiden Funktionen einen Seiteneffekt ausgelöst haben, sie haben gewissermaßen die Instanz der „Welt“ in eine andere Instanz verändert. `getline` gibt einen `String` zurück und hat gleichzeitig einen Seiteneffekt erzeugt, nämlich das Lesen einer Eingabe. `putstr` hat als Returntyp `Unit`, gibt also nichts zurück, hat aber einen Efeekt erzeugt, die Ausgabe.
- `List.head?` hat als Ergebnis nicht einfach einen Wert vom Typ `α`, sondern vom Typ `Option α`, wodurch explizit wird, dass der Zugriff auf das erste Element der Liste scheitern kann, wenn die Liste leer ist.

`IO` und `Option` sind Beispiele von Monaden und wir haben eine *erste Auffassung* von Monaden: es sind „Markierungen“ von Typen, die angeben, dass zusätzlich zum Ergebnis noch etwas Weiteres passiert ist: an unseren Beispielen ein Seiteneffekt oder das Fehlen eines Werts. Man kann von Funktionen mit solchen Typen des Zielbereichs auch als *monadischen* Funktionen sprechen.

Die Sprechweise „Markierung“ ist natürlich eher bildlich. Tatsächlich ist eine Monade ein *Typkonstruktor*: eine Funktion, die einen Typ als Argument hat und einen Typ zurückgibt. Alle Monaden sind Typkonstruktoren, aber nicht umgekehrt. Denn für Monaden müssen bestimmte Gesetze erfüllt sein.

Monaden erlauben es nicht nur, dass auf kontrollierte Weise (also via des Typsystems) mit Seiteneffekten umgegangen werden kann, sondern man kann sie auch dafür einsetzen, bestimmte Berechnungen sehr elegant zu programmieren, wie wir später sehen werden.

8.1.2 Komposition von monadischen Funktionen

Monadische Funktionen kann man nicht mit `Function.comp` „kringeln“: die Typen passen nicht: Eine monadische Funktion `g` habe den Typ `α → m β` und die Funktion `f` den Typ `β → m γ` für eine Monade `m`, dann hätten wir gerne, dass die Komposition `h = f ∘ g` den Typ `α → m γ` hätte. Das geht aber nicht, weil der Returntyp von `g` nicht der Typ des Arguments von `f` ist.

Statt `Function.comp` bräuchten wir eine Funktion `monadcomp` (oder so) mit der Signatur

```
monadcomp : (α → m β) → (β → m γ) → α → m γ
```

Diese Art der Komposition von monadischen Funktionen können wir machen, wenn es für die Monade `m` die Funktion `bind` gibt:

```
bind : m β → (α → m γ) → m γ
```

Denn gegeben

```
g vom Typ α → m β
f vom Typ β → m γ
```

dann bekommen wir eine Funktion `h` mit

```
h vom Typ α → m γ
```

durch

```
h a = bind (g a) f
```

denn `g a` hat den Typ `m β` und `f` den Typ `β → m γ`, also hat `h` den Typ `α → m γ`.

Es gibt für bind auch einen Infix-Operator, nämlich >>=. In unserem Beispiel könnte man also auch schreiben

```
h a = g a >>= f
h := λ x => (g x >>= f)
```

Gegeben `bind` für eine Monade `m` kann man also die monadische Funktionskomposition definieren. Es gibt dafür auch einen Infix-Operator in Lean: >=>, d.h.

```
g >=> f = λ x => (g x >>= f)
```

Es gibt noch einen weiteren Fall, den wir beachten müssen: es kann vorkommen, dass man eine monadische Funktion mit einer regulären Funktion komponieren möchte:

Gegeben

```
g vom Typ α → m β
f vom Typ β → γ
```

wollen wir auch in diesem Fall eine Funktion `h` haben mit

```
h: α → m γ
```

Dazu verwenden wir die Funktion `pure` der Monade:

```
pure : α → m α
```

Denn gegeben `pure` ergibt sich

```
h = g >=> (pure ∘ f)
```

Überprüfung der Typen:

```
g          hat Typ α → m β
f          hat Typ β → γ
pure       hat Typ γ → m γ
pure ∘ f   hat Typ β → m γ
also
h          hat Typ α → m γ
denn
>==>       hat Typ (α → m β) → (β → m γ) → α → m γ
```

Wir können also monadische Funktionen miteinander und mit regulären Funktionen komponieren, wenn wir für eine Monade die Funktionen `pure` und `bind` zur Verfügung haben.

Dazu gibt es die Typklasse

```
class Monad (m : Type → Type) where
  pure : α → m α
  bind : m α → (α → m β) → m β
```

Wie die Funktionen `pure` und `bind` für die jeweilige Monade zu implementieren sind, bzw. in Lean bereits implementiert sind, hängt natürlich vom jeweiligen Anwendungsfall für die Monade ab, ist also spezifisch je Monade. Damit aber die gewünschten Eigenschaften der Komposition tatsächlich erreicht werden, muss man die Gesetze der Monade befolgen.

8.1.3 Die Gesetze der Monade

Für die Funktionen `pure` und `bind` müssen folgende Gesetze gelten.

- `pure` ist die linke Identität von `bind`, d.h. `bind (pure a) g = g a`, oder kurz `pure >=> g = g`
- `pure` ist die rechte Identität von `bind`, d.h. `bind mv pure = mv` für den monadischen Wert `mv`, oder kurz `g >=> pure = g`.
- `bind` ist assoziativ, d.h. `bind (bind mv g) f = bind mv (λ x => bind (g x) f)`, oder kurz `(g >=> f) >=> h = g >=> (f >=> h)`

Die Begründung für die Gesetze:

- `pure` hat keine Effekte, d.h. die Kombination mit bind sollte nichts ändern.
- Assoziativität wird verlangt, weil es nur auf die Reihenfolge der Funktionen ankommt, egal welches `bind` man zuerst ausführt.

Es gibt in Lean das Konzept der `LawfulMonad`. Diese Typklasse verlangt, dass `pure`und `bind`einer Monade folgende Gesetze erfüllt (siehe `Init.Control.Lawful.Basic`):

```
pure x >>= f = f x
x >>= pure = x
x >>= f >>= g = x >>= (fun x => f x >>= g)
```

Die Typklasse `LawfulMonad` in Lean verlangt also nicht nur dass eine Instanz die Methoden der Typklasse bereitstellt, sondern auch die Beweise für die drei Gesetze einer Monade.

8.2 Beispiel Option

Wir haben schon als Beispiel für fehlende Werte gesehen, dass das Ermitteln des ersten Elements einer Liste zwei mögliche Ergebnisse hat: einen Wert oder das Fehlen eines Werts. Dies regelt `Option`.

Wir wollen nun an einem Beispiel (siehe Christiansen: Kap. 5 Monads) sehen, wie man `Option` einsetzen kann, ohne die Typklasse `Monad` zu verwenden. Daran kann man gut sehen, welche Vorteile die Funktionen `pure` und `bind` bringen.

Beginnen wir damit das erste Element einer Liste zu ermitteln:

```
def first (as : List α) : Option α :=
  as[0]?
```

```
#eval first [1, 2, 3, 4, 5, 6, 7]  -- ergibt some 1
#eval first ([] : List Nat)        -- ergibt none
```

Angenommen wir wollen das erste, dritte und fünfte Element der Liste haben, dann müssen wir mehr arbeiten:

```
def firstThirdFifth (xs : List α) : Option (α × α × α) :=
  match xs[0]? with
  | none => none
  | some first =>
    match xs[2]? with
    | none => none
    | some third =>
      match xs[4]? with
      | none => none
      | some fifth =>
        some (first, third, fifth)

#eval firstThirdFifth [1, 2, 3, 4, 5, 6, 7]  -- ergibt some (1, 3, 5)
#eval firstThirdFifth [1, 2, 3]              -- ergibt none
```

Es ist offensichtlich, dass der Code für jedes `match` redundant ist und dass er durch die Monaden-Funktionen ersetzt werden kann. Im Prinzip sieht die Instanz der Typklasse so aus:

```
instance : Monad Option where
  pure x := some x
  bind opt next :=
    match opt with
    | none => none
    | some x => next x
```

Damit lässt sich die Funktion so formulieren:

```
def firstThirdFifth' (xs : List α) : Option (α × α × α) :=
  xs[0]? >>= fun first =>
  xs[2]? >>= fun third =>
  xs[4]? >>= fun fifth =>
  pure (first, third, fifth)
```

Zur Erinnerung: `>>=` ist der Infix-Operator für `bind : m α → (α → m β) → m β`, d.h. wir erhalten genau die Schachtelung der `matches` von oben.

```
#eval firstThirdFifth' [1, 2, 3, 4, 5, 6, 7, 8, 9]
  -- ergibt some (1, 3, 5)
#eval firstThirdFifth' [1, 2, 3, 4,]
  -- ergibt none
```

8.3 *do*-Notation für Monaden

Bei der Verwendung von Monaden kann der Code leicht unübersichtlich werden. Deshalb gibt es die *do*-Notation, die den Code durchsichtiger macht. Wir haben diese Notation schon am Beispiel des Programms `lcat` eingesetzt. Im Grunde führt die *do*-Notation dazu, dass man funktionalen Code schreiben kann, als wäre er imperativer Code.

Um die _do_Notation zu verstehen, müssen wir etwas mehr über `let` wissen.

Man kann *lokale* Definitionen von Symbolen mit `let ... := ...` machen. Dies ist hilfreich, wenn in einer Funktion eine Berechnung wiederholt werden müsste. Mit einem `let` kann ihr Ergebnis wiederverwendet werden.

Speziell für die *do*-Notation kann `let` auch so verwendet werden: `let x ← Expr; Action` was syntaktischer Zucker ist für `bind Expr fun x => Action`.

Beispiel:

```
def isGreaterThan0 (x : Nat) : IO Bool := do
  IO.println s!"value: {x}"
  return x > 0
```

In der *do*-Notation gibt es das Schlüsselwort `return`. `return` beendet den `do`-Block und gibt den Wert eingepackt in `pure` zurück (siehe Functional Programming in Lean: More do Features).

```
def f (x : Nat) : IO Unit := do
  let c <- isGreaterThan0 x
  if c then
    IO.println s!"{x} is greater than 0"
  else
    pure ()
```

(dabei ist `()` eine Notation für `Unit.unit`.)

```
#eval f 10
-- value: 10
```

```
-- 10 is greater than 0
#eval f 0
-- value: 0
```

Folgende Fälle können bei der *do*-Notation auftreten:

1. `do` hat nur einen einzigen Ausdruck

```
do Expr
```

wird übersetzt zu

```
Expr
```

2. `do` beginnt mit einem `let ←`

```
do let x ← Expr1
   Stmt
   ...
   Exprn
```

wird übersetzt zu

```
Expr1 >>= fun x =>
do Stmt
   ...
   Exprn
```

3. `do` beginnt mit einem Ausdruck und weiteren Statements

```
do Expr1
   Stmt
   ...
   Exprn
```

wird übersetzt zu

```
Expr1 >>= fun () =>
do Stmt
   ...
   Exprn
```

4. `do` beginnt mit einem `let` mit `:=`

```
do let x := Expr1
   Stmt
   ...
   Exprn
```

wird übersetzt zu

```
let x := Expr1
do Stmt
   ...
   Exprn
```

Mehr zur *do*-Notation findet man im Lean Manual.

Mit der *do*-Notation kann man unsere Funktion, die Elemente in einer Liste sucht und ausgibt, so schreiben — und es sieht fast wie imperativer Code aus:

```
def firstThirdFifth''
  (xs : List α) : Option (α × α × α) := do
  let first ← xs[0]?
  let third ← xs[2]?
  let fifth ← xs[4]?
  pure (first, third, fifth)

#eval firstThirdFifth'' [1, 2, 3, 4, 5, 6, 7, 8, 9]
  -- ergibt some (1, 3, 5)
#eval firstThirdFifth'' [1, 2, 3, 4]
  -- ergibt none
```

8.4 Beispiel Except

`Except` ist die Monade in Lean, die dem Konzept des Exception-Handlings in anderen Programmiersprachen entspricht.

Wir nehmen das Beispiel für das Ausgeben des ersten, dritten etc Elements einer Liste, das oben bereits als Motivation für die Monade `Option` verwendet wurde. Siehe wieder Christiansen: Monads.

Man könnte definieren

```
inductive Except (ε : Type) (α : Type) where
  | error : ε → Except ε α
  | ok : α → Except ε α
deriving BEq, Hashable, Repr
```

Dabei steht ε für den Typ des Fehlers und α für den Typ des regulären Werts.

Beim Suchen nach Einträgen in einer Liste könnte man also verwenden:

```
def getOrExcept (xs : List α) (i : Nat) : Except String α :=
  match xs[i]? with
  | none => Except.error s!"Index {i} not found (maximum is {xs.length - 1})"
  | some x => Except.ok x
```

und wieder könnte man eine Kaskade von matches machen, wenn man nach mehreren Einträge sucht:

```
def firstThirdFifth (xs : List α) : Except String (α × α × α) :=
  match get xs 0 with
  | Except.error msg => Except.error msg
  | Except.ok first =>
    match get xs 2 with
    | Except.error msg => Except.error msg
    | Except.ok third =>
      match get xs 4 with
      | Except.error msg => Except.error msg
      | Except.ok fifth =>
        Except.ok (first, third, fifth)
```

Aber wir machen das schöner mit einer passenden Monade, siehe `Init.Prelude`:

```
inductive Except (ε : Type u) (α : Type v) where
  | error : ε → Except ε α
  | ok    : α → Except ε α
```

und die Dokumentation dazu:

> `Except ε α` is a type which represents either an error of type ε, or an “ok” value of type α. The error type is listed first because `Except ε : Type → Type`

> is a `Monad`: the pure operation is `ok` and the bind operation returns the first encountered `error`.

Wir verallgemeinern unsere Funktion `firstThirdFifth` aus dem Abschnitt über `Option` so dass wir die Suchfunktion übergeben können:

```
def firstThirdFifth''' [Monad m]
  (lookup : List α → Nat → m α) (xs : List α) : m (α × α × α) := do
  let first ← lookup xs 0
  let third ← lookup xs 2
  let fifth ← lookup xs 4
  pure (first, third, fifth)

#eval firstThirdFifth''' getOrExcept [1, 2, 3, 4, 5, 6, 7, 8, 9]
  -- ergibt Except.ok (1, 3, 5)

#eval firstThirdFifth''' getOrExcept [1, 2, 3, 4]
  -- ergibt Except.error "Index 4 not found (maximum is 3)"
```

Ein weiteres Beispiel aus Lean Manual:

```
def divide (x y : Float) : Except String Float :=
  if y == 0 then
    throw "can't devide by zero"
  else
    pure (x / y)
```

`throw` ist syntaktischer Zucker für den Fehlerfall von `Except`.

```
#eval divide 42 2 -- ergibt: Except.ok 21.00000
#eval divide 42 0 -- ergibt: Except.error "can't devide by zero"
```

Man kann nun `Except` verketten. Schreiben wir erst eine weitere Funktion:

```
def square (x : Float) : Except String Float :=
  if x > 100 then
    throw s!"{x} is too huge!"
  else
    pure (x * x)
```

Nun können wir mit der *do*-Notation die beiden Funktionen verketten:

```
def chainFuncs (x y : Float):= do
  let r ← divide x y
  square r
```

```
#eval chainFuncs 300 2 -- Except.error "150.000000 is too huge!"
#eval chainFuncs 42 5  -- Except.ok 70.560000
#eval chainFuncs 42 0  -- Except.error "can't devide by zero"
```

8.5 Beispiel Reader

Reine Funktionen können nur Werte verwenden, die ihnen als Argumente übergeben wurden. Im Prinzip bedeutet das, dass man keine globalen Variablen haben kann.

Man kann globale Definitionen in Funktionen verwenden, aber sie werden zur Compile-Zeit als Closure fixiert. Beispiel:

```
def global := 2

def specialAdd: ℕ → ℕ := λ n => n + global

#eval specialAdd 0

def global := 3
  -- 'global' has already been declared
```

Wie man mit diesem Thema in Lean umgeht, wird im Lean Manual: 12.4 Readers an folgendem Beispiel erläutert:

Stellen wir uns vor, wir haben Umgebungsvariablen, die wir laden wollen. Dann müssen wir die IO-Monade einsetzen. Etwa so:

```
structure Environment where
  path : String
  home : String
  user : String
  deriving Repr

-- Lese Umgebungsvariable aus aktuellem Prozess
def getEnvDefault (name : String) : IO String := do
  let val? ← IO.getEnv name
  pure <| match val? with
    | none => ""
    | some s => s

#eval getEnvDefault "USER"

def loadEnv : IO Environment := do
```

```
  let path ← getEnvDefault "PATH"
  let home ← getEnvDefault "HOME"
  let user ← getEnvDefault "USER"
  pure { path, home, user }

#eval loadEnv

-- Wir fangen an, eine Art Hash für das Env zu bilden
def func1 (env : Environment) : Float :=
  let pl := env.path.length
  let hl := env.home.length * 2
  let ul := env.user.length * 3
(pl + hl + ul).toFloat * 2.1

def func2 (env: Environment) : Nat :=
  2 + (func1 env).floor.toUInt32.toNat

def func3 (env: Environment) : String :=
  "Result: " ++ (toString (func2 env))

def main1 : IO Unit := do
  let env ← loadEnv
  let str := func3 env
  IO.println str

#eval main1
```

Die einzige der drei Funktionen, die die Umgebung tatsächlich verwendet ist `func1`. Aber diese Funktion kann `loadEnv` nicht direkt verwenden, weil `loadEnv` eine `IO`-Funktion ist, also eine monadische Funktion.

Dies führt dazu, dass man `env` durch die ganzen Funktionen durchreichen muss. Bei komplexeren Programmen kann das dazu führen, dass man viele Parameter hat, die man bloß deshalb braucht.

Die Lösung in Lean ist die Monade `ReaderM`:

```
def readerFunc1 : ReaderM Environment Float := do
  let env ← read
  let pl := env.path.length
  let hl := env.home.length * 2
  let ul := env.user.length * 3
  return (pl + hl + ul).toFloat * 2.1
```

In der Typklasse `MonadReader`, die von `ReaderM` erfüllt wird, ist `read` definiert als das Lesen des aktuellen Zustands von `ReaderM`.

```
def readerFunc2 : ReaderM Environment Nat :=
  readerFunc1 >>= (λ x => return 2 + (x.floor.toUInt32.toNat))

def readerFunc3 : ReaderM Environment String := do
  let x ← readerFunc2
  return "Result: " ++ toString x

def main2 : IO Unit := do
  let env ← loadEnv
  let str := readerFunc3.run env
  IO.println str

#eval main2
```

Hier gibt es Einiges zu erläutern:

1. `ReaderM` hat eine Methode `run`, die die Umgebung (hier `env`) übernimmt. Dadurch bekommt `ReaderM` diesen Kontext, der dann in allen Funktionen zur Verfügung steht. Man könnte also sagen, dass `run` zu einer Art `Dependency Injection` führt.
2. Wir sehen die erneut die Verwendung von `return`. `return` erzeugt wie `pure` aus einem Wert einen Monadenwert. Aber `return` ist ein Schlüsselwort, d.h. wir brauchen keine Klammern für den Ausdruck wie bei `pure`. Insbesondere aber kann `return` wie in einer imperativen Sprache zu einem frühen Return in einer monadischen Funktion eingesetzt werden. Mehr zu „frühem Return“ siehe Christiansen 7.4 More do Features.

Was wird mit `ReaderM` erreicht?

> The important difference here to the earlier code is that `readerFunc3` and `readerFunc2` no longer have an *explicit* Environment input parameter that needs to be passed along all the way to `readerFunc1`. Instead, the `ReaderM` monad is taking care of that for you, which gives you the illusion of something like global context where the context is now available to all functions that use the `ReaderM` monad.

8.6 Beispiel State

Die Monade `StateM` erlaubt es nicht nur Daten im Kontext zu lesen, sondern sie auch zu ändern. Wir nehmen als Beispiel einen Stack.

Wenn man einen Stack mit den Operationen `push` und `pop` darauf hat, würde man das in einer objektorientierten Sprache wie Java etwa so machen:

```
Deque<Integer> stack= new ArrayDeque<>();
boolean result = stack.empty();
System.out.println("Ist der Stack leer? " + result);
// Werte auf den Stack legen
stack.push(1);
stack.push(2);
stack.push(3);
stack.push(42);
stack.push(43);
stack.pop;
// Stack ausgeben
System.out.println("Werte im Stack: " + stack);
```

ergibt folgende Ausgabe

```
Ist der Stack leer? true
Werte im Stack: [1, 2, 3, 42]
```

Da das Objekt in Java zustandsbehaftet ist, änderm die Funktionen `push` und `pop` den Zustand des Stacks.

Wenn wir nun dasselbe Beispiel in Lean machen wollen, sind wir mit der Tatsache konfrontiert, dass Lean keine zustandsbehafteten Objekte kennt.

Wenn wir den Stack durch eine Liste repräsentieren, dann könnten wir folgende Implementierung für den Stack machen.

```
structure Stack (α : Type) where
  st : List α
  deriving Repr

def Stack.new (α : Type) : Stack α :=
  { st := (List.nil : List α) }
```

```
def Stack.push (stack : Stack α) (a : α) : Stack α :=
  let l := stack.st
  Stack.mk (a :: l)

def Stack.pop? (stack : Stack α) : (Option α × Stack α) :=
  match stack.st with
  | [] => (none, stack)
  | a :: as => (some a, Stack.mk as)

def Stack.empty (stack: Stack α) : Bool :=
  stack.st.length == 0
```

Mit dieser Implementierung könnte man das obige Beispiel folgendermaßen in Lean nachvollziehen:

```
namespace Stack

def stackExample : Stack Nat :=
  let stack := new Nat
  let stack1 := stack.push 1
  let stack2 := stack1.push 2
  let stack3 := stack2.push 3
  let stack4 := stack3.push 42
  let stack5 := stack4.push 43
  stack5.pop?.snd

#eval stackExample

end Stack

namespace Stack'
```

Wie wir sehen, brauchen wir für jeden Schritt immer den vorherigen Zustand und geben auch einen neuen Zustand zurück. Warum also nicht `push` und `pop` gleich so definieren, dass der Zustand des Stacks als Argument den Funktionen selbst steckt?

```
def push (stack : Stack Nat) (a : Nat) : Unit × Stack Nat :=
  ((), Stack.push stack a)

def pop? (stack : Stack Nat): Option Nat × Stack Nat :=
  let res := Stack.pop? stack
  match res.fst with
```

```
  | none => (none, res.snd)
  | some a => (a, res.snd)
```

Damit können wir das Beispiel so formulieren:

```
def stackExample : Stack Nat :=
  let stack1 := (push (Stack.new Nat) 1).snd
  let stack2 := (push stack1 2).snd
  let stack3 := (push stack2 3).snd
  let stack4 := (push stack3 42).snd
  let stack5 := (push stack4 43).snd
  (pop? stack5).snd

#eval stackExample
```

oder so:

```
def stackExample' : Stack Nat :=
  (pop?
    (push
      (push
        (push
          (push
            (push (Stack.new Nat) 1).snd
           2).snd
         3).snd
       42).snd
     43).snd
  ).snd
#eval stackExample'
end Stack'
```

Wie man sieht, muss man immerzu den Stack als Argument an die nächste Funktion weitergeben. Hier kommt nun die Monade `StateM` ins Spiel: sie gibt den Stack jeweils als neuen Wert an die nächste Funktion weiter.

Eine zustandsändernde Aktion ist eine Funktion, die den aktuellen Zustand bekommt und einen Ergebniswert zusammen mit den neuen Zustand zurückgibt. Die Monade `StateM` kann das Weitergeben des Zustands, das wir eben selbst gemacht haben für uns übernehmen.

Wir wenden das auf unser Beispiel mit dem Stack an und ersparen uns das mühsame Weitergeben des jeweiligen Zustands des Stacks:

```
namespace StateStack
```

```
def push (a : Nat) : StateM (Stack Nat) Unit := do
  modifyGet fun stack => ((), Stack.push stack a)
```

`modifyGet` nimmt den bisherigen Zustand und wendet eine Funktion darauf an, die das Paar von Ergebniswert (hier vom Typ `Unit`) sowie den neuen Zustand zurückgibt.

```
def pop: StateM (Stack Nat) (Option Nat) := do
  let stack ← get
  let res := Stack.pop? stack
  let stack' := res.snd
  set stack'
  return res.fst
```

`get` und `set` verwenden den Zustand, den die Monade weiterreicht, lesend bzw. schreibend.

```
def stackExample : StateM (Stack Nat) (Stack Nat):= do
  push 1
  push 2
  push 3
  push 42
  push 43
  let _ ← pop
  return ← get

#eval stackExample.run (Stack.new Nat)
#eval stackExample (Stack.new Nat)
#eval (stackExample (Stack.new Nat)).fst
#eval (stackExample (Stack.new Nat)).fst.st
```

Wie bei `ReaderM` gibt es auch hier die Funktion `run`, mit der der Startzustand übergeben wird und dann die Funktion ausgeführt wird. In unserem beispiel starten wir mit einem leeren Stack natürlicher Zahlen.

Übungen:

[08.1] Eine Funktion zum Aufstellen der Wahrheitstafel für eine Boolesche Funktion findet man m Internet, siehe Lean 99: Ninety-Nine Lean Problems, Problem 48 (Man kann dort nicht nur die Aufgabe sehen, sondern mit einem Link ganz oben rechts auch die vorgeschlagene Lösung). In der Aufgabe wird `List` eine Instanz der Typklasse `Monad`, wir können aber auch einfach `List` aus der `MathLib` nehmen:

```
--import MathLib (brauchen wir am Anfang der Lean-Datei)
namespace tt
def Arity : (n : Nat) → Type
  | 0 => Bool
  | n + 1 => Bool → Arity n

 def tt (n : Nat) (p : Arity n) : List (List Bool) :=
  match n with
  | 0 => [[p]]
  | n + 1 => do
    let b ← [true, false]
    let result ← tt n (p b) |>.map (b :: ·)
    return result

-- Beispiele
#eval tt 1 (fun a => !a)
#eval tt 2 (fun a b => a || b)
#eval tt 3 (fun a b c => a && b && c)

end tt
```

Diese Implementierung verwendet die Monade `List` aus `MathLib`. Entschlüsseln Sie, was da genau passiert und schreiben Sie ein Definition der Funktion ohne `do` und ohne `>>=`, sondern nur mit Funktionen von `List` aus `Init.Data.List.Basic`.

[08.2] Vollziehen Sie das Beispiel für den Einsatz der Monade `StateM` am Beispiel des Spiels Tic Tac Toe nach: Erläutern Sie genau, was in diesem Programm passiert. Das Beispiel war Teil des Lean-Manuals, ist aber dort nicht mehr zu finden. Eine Kopie finden Sie hier. (Hinweis: An einigen Stellen benötigen Sie Kenntnisse, die in Kapitel 9.3 besprochen werden.)

8.7 Zusammenfassung

In diesem Kapitel haben wir Beispiele von *Monaden* kennengelernt, Typkonstruktoren, die es erlauben auf elegante Weise Seiteneffekte, „globale Variablen“, Ausnahmesituationen und Ähnliches in der funktionalen Sprache Lean zu behandeln. Wir haben Beispiele gesehen:

- **`Option`**: Lean hat nur totale Funktionen, d.h. die Situation, dass zu einem Argument kein Funktionswert vorhanden ist, muss explizit behandelt werden. Dies geschieht mit der Monade `Option`, die fehlende Funktionswerte „markiert“.
- **`Except`**: Treten Ausnahmesituationen aus, ist es in imperativen und/oder objektorientierten Sprachen üblich, mit Exceptions zu arbeiten. In Lean kann man dafür `Except` verwenden.
- **`ReaderM`** erlaubt es, einen Kontext in Funktionen lesend zu verwenden, ohne dass man in jeder von ihnen dafür einen eigenen Parameter braucht. Der Zugriff auf die Werte im Kontext erfolgt über `ReaderM`.
- **`StateM`** erlaubt es einen Kontext lesend und schreibend zu verwenden.

Im nächsten Kapitel werden wir uns mit der Systematik im Aufbau der Monaden befassen: Vom Funktor über den Applicative Funktor zur Monade.

---

9 Mehr über Monaden

In diesem Kapitel wollen wir unsere Kenntnisse über Monaden in Lean vertiefen:

- Man kann die systematische Verwendung von Typklassen in Lean (wie auch in Haskell) als die Art betrachten, wie man in einer funktionalen Sprache softwaretechnisch vorgeht. Dies wollen wir nun näher betrachten in der Systematik, wie die Typklassen `Functor`, `Applicative` und `Monad` aufeinander aufbauen.
- Monaden kann man kombinieren, um neue Monaden zu konstruieren. Deshalb werfen wir einen kurzen Blick mit zwei Beispielen auf das Konzept der *Monad Transformer*.
- Schließlich lernen wir weitere Möglichkeiten der *do*-Notation kennen, die es im Kontext von Monaden erlauben, in der funktionalen Sprache Lean quasi im imperativen Stil zu programmieren.

9.1 Funktoren, Applikative Funktoren und Monaden

9.1.1 Funktoren

Bildlich gesprochen kann man sich einen Funktor vorstellen als einen Container von Elementen zusammen mit einer Funktion, die normalerweise `map` genannt wird, die die Elemente des Containers abbildet, und dabei die Struktur des Containers erhält.

Genau genommen ist ein Funktor also ein Typkonstruktor `F : Type u → Type v` zusammen mit einer Funktion `map`, so dass für eine Funktion `f : α → β` `map f` den Typ `F α → F β` hat. Man sagt, dass `map` die Funktion `f` „liftet“.

Nicht nur das: natürlich will man auch, dass `map` sich ordentlich verhält, d.h mit der Identitätsfunktion `id: α → α` und der Komposition von Funktionen verträglich ist. Man nennt das auch *Funktorialität* oder die *Gesetze des Funktors* – mehr dazu nach den Beispielen.

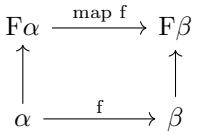

Abbildung 9.1: Funktor

9.1.1.1 Beispiele

Das Standardbeispiel ist `List`, aber auch `Option` ist ein Funktor.

```
#check List.map
#eval [1, 2, 3].map (λ _ =>  2)  -- ergibt [2, 2, 2]
```

Beispiele mit `List`:

```
#eval List.map (λ x => toString x) [1, 2 ,3] --  ["1", "2", "3"]

-- oder mit dot-Notation
#eval [1, 2, 3].map (λ x => toString x)

#eval ["lean", "haskell"].map (λ s => s.length) -- [4, 7]
#eval ["lean", "haskell"].map (λ s => s.capitalize) -- ["Lean", "Haskell"]
```

Beispiele mit `Option`:

```
#eval (some 5).map (λ x => x + 1) -- some 6

-- oder kürzer
#eval (some 5).map (· + 1) -- some 6
#eval (none).map (· + 1)   -- none
```

Es gibt auch einen Infix-Operator für `map`, nämlich `<$>` mit `map f l = f <$> l`, d.h. die obigen Beispiel kann man auch so schreiben:

```
#eval (λ _ =>  2) <$> [1, 2, 3]
#eval (λ x =>  toString x) <$> [1, 2, 3]
#eval (λ s => s.length) <$> ["lean", "haskell"]
#eval (λ s => s.capitalize) <$> ["lean", "haskell"]
#eval String.length <$> ["lean", "haskell"]
#eval String.capitalize <$> ["lean", "haskell"]
```

9.1.1.2 Gesetze für Funktoren

Die Typklasse `LawfulFunctor` gibt die beiden Gesetze für Funktoren an:

1. `id <$> x = x`, d.h. der Funktor erhält die Identitätsfunktion
2. `(h ∘ g) <$> x = h <$> g <$> x`, d.h. der Funktor erhält die Komposition von Funktionen

Beispiele:

```
def list_ex := [1, 2, 3]

#eval List.map id list_ex
#eval id <$> list_ex == list_ex -- true

def double : ℕ → ℕ := (· * 2)
def square : ℕ → ℕ := λ n => n * n
#eval double <$> square <$> list_ex
#eval (double ∘ square) <$> list_ex
#eval double <$> square <$> list_ex == (double ∘ square) <$> list_ex -- true

def option_ex1 := some 5
def option_ex2: Option Nat := none

#eval id <$> option_ex1
#eval id <$> option_ex2
#eval ((. + 2) ∘ (. * 3)) <$> option_ex1 -- some 17
#eval (. + 2) <$> (. * 3) <$> option_ex1 -- some 17
```

Übungen:

[09.1] Beweisen Sie, dass `List` die Gesetze des Funktors erfüllt. (Verwenden Sie die Taktik `simp` möglichst nicht.)

[09.2] Beweisen Sie, dass `Option` die Gesetze des Funktors erfüllt. (Verwenden Sie die Taktik `simp` möglichst nicht.)

9.1.1.3 Komposition von Funktoren

Wir können Funktoren auch komponieren:

Beispiel `Option List α`:

```
def mySq : Int → Int := λ i => i * i

def safeTail : List α → Option (List α)
  | [] => none
  | _ :: as => some as

#eval List.map mySq [1, 2, 3]
#eval List.map mySq []

-- Wir komponieren map von List und Option
#eval (Option.map ∘ List.map) mySq (safeTail [1, 2, 3] )
-- some [4, 9]
#eval (Option.map ∘ List.map) mySq (safeTail [])
-- none
```

Beispiel `List (List α)`:

Wir verwenden `List.map ∘ List.map` für verschachtelte Listen.

```
#eval List.map mySq [1, 2, 3]      -- [1, 4, 9]
#eval List.map mySq [40, 41, 42]   -- [1600, 1681, 1764]

#eval (List.map ∘ List.map) mySq [[1, 2, 3], [40, 41, 42]]
-- [[1, 4, 9], [1600, 1681, 1764]]

def nestedMap (f : α → β ):= (List.map ∘ List.map) f
#eval nestedMap mySq [[1, 2, 3], [40, 41, 42]]
-- [[1, 4, 9], [1600, 1681, 1764]]
```

9.1.2 Applikative Funktoren

Der Funktor liftet mit `map` eine Abbildung in den Funktortyp. Man kann sich aber auch den Fall vorstellen, dass der Funktortyp selbst einen Funktionstyp „enthält“. Das führt zum *applicativen Funktor*, d.h. der Typklasse `Applicative`.

Das Konzept besteht darin, dass der applikative Funktor zwei Funktionen hat, die sich „ordentlich verhalten“:

```
pure: α → F α
seq:  F (α → β) → F α → F β
```

Als Infix-Operator wird `seq` so geschrieben: `<*>`.

Die Funktion `pure` erlaubt es ein Objekt zu einem Objekt des Typs `Applicative` zu machen. Und mit `seq` kann man Funktionen verketten. Der Name „applikativer Funktor“ kommt daher, dass man Funktionen „innerhalb“ des Funktortyps anwendet.

9.1.2.1 Beispiele

Wir sehen uns wieder Beispiele mit `List` und `Option` an:

Beispiel `List`:

Wir können nun also eine Liste von Funktionen auf eine Liste von Werten durch `seq` anwenden:

```
#eval [(· + 2)] <*> [1, 2, 3] -- [3, 4, 5]
-- dies ergibt also dasselbe wie map
#eval (· + 2) <$> [1, 2, 3] -- [3, 4, 5]
#eval [(· + · )] <*> [1, 2, 3] <*> [3, 4, 5]
-- ergibt [4, 5, 6, 5, 6, 7, 6, 7, 8]

#eval [(· + 2), (· * 5)] <*> [1, 2, 3]
-- ergibt [3, 4, 5, 5, 10, 15]
#eval [(· + 2), (· * 5), (· - 1)] <*> [1, 2, 3]
-- ergibt [3, 4, 5, 5, 10, 15, 0, 1, 2]
```

Man sieht, dass jeder der Funktionen auf jeden Wert der Liste angewendet wird.

Damit kann man auch ein bisschen Kombinatorik betreiben:

```
#eval Prod.mk <$> (List.range 2) <*> (List.range 3)
-- ergibt [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
#eval (· , ·) <$> (List.range 2) <*> (List.range 3)
-- ergibt [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
#eval (· , · , ·) <$> [1, 2] <*> [3, 4] <*> [5]
-- ergibt [(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)]
```

Beispiel `Option`:

```
#eval pure (· * ·) <*> some 2 <*> some 21 -- some 42
#eval (· * ·) <$> some 21 <*> some 2      -- some 42
#eval pure (· * ·) <*> some 2 <*> none    -- none
#eval pure (· * ·) <*> none <*> some 21   -- none
```

`Applicative` hat auch noch zwei weitere Sequenzoperatoren `SeqLeft` (infix: <*) und `SeqRight` (infix: *>), die folgendermaßen funktionieren:

`x <* y` evaluiert `x` und `y` und hat als Ergebnis die Auswertung von `x`. Analog hat `x *> y` nach der Evaluierung von `x` und `y` den Wert von `y`.

Diese Operatoren werden viel in Parser-Bibliotheken verwemdet.

9.1.2.2 Die Gesetze des applikativen Funktors

Zunächst wollen wir prüfen, dass es sich tatsächlich um einen Funktor handelt. Das bedeutet, dass man `map` aus den beiden Funktionen `pure` und `seq` komponieren kann. Und wir haben das schon in Beispielen gesehen:

```
map (f : α → β) : F α → F β = seq (pure f) : F α → F β
```

Nun die Gesetze von `Applicative`:

```
pure id <*> v = v                                    -- Identität
pure f <*> pure x = pure (f x)                       -- Homomorphismus
u <*> pure y = pure (· y) <*> u                      -- Vertauschung
pure (· ∘ ·) <*> u <*> v <*> w = u <*> (v <*> w) -- Komposition
```

Beispiel `List`:

```
-- Identität

#eval pure id <*> [1, 2, 3] -- [1, 2, 3]


-- Homomorphismus

def f : Nat → Nat := (· + 2)

#eval (pure f : List (Nat → Nat)) <*> pure 1       -- ergibt [3]
#eval (pure (f 1) : List Nat)                      -- ergibt [3]
#eval pure f <*> pure 1 == (pure (f 1) : List Nat) -- true

-- Vertauschung

def g : List (Nat → Nat) := [(· + 2)]

#eval g <*> pure 40                    -- ergibt [42]
#eval pure (· 40) <*> g                -- ergibt [42]
```

```
#eval g <*> pure 40 == pure (· 40) <*> g -- true

-- dazu beobachten wir:
#eval (· 40) (· + 2)          -- ergibt 42
#eval (fun f => f 40) (· + 2) -- ohne syntaktischen Zucker

--  Komposition

def u := [(· + 2)]    -- eine Liste von Funktionen
def v := [(· + 3)]    -- eine Liste von Funktionen
def w := [5, 6]       -- eine Liste von Werten

#eval pure (· ∘ ·) <*> u <*> v <*> w   -- [10, 11]
-- löst sich so auf:
#eval pure ((· + 2) ∘ .) <*> v <*> w   -- [10, 11]
#eval pure ((· + 2) ∘ (· + 3)) <*> w   -- [10, 11]

#eval u <*> (v <*> w)                  -- [10, 11]
-- löst sich so auf:
#eval v <*> w                          -- [8, 9]
#eval u <*> [8, 9]                     -- [10, 11]
```

Beispiel `Option`:

```
-- Identität

#eval pure id <*> some 42             -- some 42
#eval pure id <*> (none : Option Nat) -- none

-- Homomorphismus

#print f -- fun x => x + 2

#eval (pure f : Option (Nat → Nat)) <*> some 1  -- ergibt some 3
#eval (pure (f 1) : Option Nat)                 -- ergibt some 3
#eval pure f <*> pure 1 == (pure (f 1) : Option Nat) -- true

-- Vertauschung

def add2 : Option (Nat → Nat) := some (· + 2)

#eval add2 <*> pure 40                   -- ergibt some 42
#eval pure (· 40) <*> add2               -- ergibt some 42
#eval add2 <*> pure 40 == pure (· 40) <*> add2 -- true
```

```
-- dazu beobachten wir:
#eval (· 40) (· + 2)          -- ergibt 42
#eval (fun f => f 40) (· + 2) -- ohne syntaktischen Zucker

--  Komposition

def uo := some (· + 2)    -- eine optionale Funktion
def vo := some (· + 3)    -- eine optionale Funktion
def wo := some 5          -- eine optionaler Wert

#eval pure (· ∘ ·) <*> uo <*> vo <*> wo  -- some 10
-- löst sich so auf:
#eval pure ((· + 2) ∘ .) <*> vo <*> wo   -- some 10
#eval pure ((· + 2) ∘ (· + 3)) <*> wo    -- some 10

#eval uo <*> (vo <*> wo)                 -- some 10
-- löst sich so auf:
#eval vo <*> wo                          -- some 8
#eval uo <*> some 8                     -- some 10

#eval pure (· ∘ ·) <*> uo <*> vo <*> none  -- none
#eval uo <*> (vo <*> none)                 -- none
```

9.1.2.3 Verzögerte Auswertung

Im Code der Typklasse `Seq` hat die Funktion `seq` eine etwas andere Signatur als wir oben angegeben haben:

```
class Seq (f : Type u → Type v) where
  seq : {α β : Type u} → f (α → β) → (Unit → f α) → f β
```

Das bedeutet, dass `seq` gar nicht direkt `F α` als zweites Argument hat, sondern eine Funktion `Unit → F α`. Warum?

Lean wertet strikt (*eager evaluation*) aus. Möchte man jedoch verzögerte Auswertung (*lazy evaluation*), dann verwendet einen sogenannten *Thunk*, eine Funktion `Unit → α`.

Wird eine Funktion `f` mit dem Argument `x` ausgewertet, dann wird zunächst `x` ausgewertet und dann der Wert als Argument an die Funktion `f` übergeben. Will

man diese Auswertung nicht, dann verwendet man als Argument für `f` die Funktion `λ _ => x`. Da Funktionen Werte sind, ist dieses Argument also schon evaluiert.

Dieses Vorgehen ist bei `seq` sinnvoll, wie man am Beispiel von `Option` sieht. Dort ist `seq` so definiert:

```
seq f x :=
    match f with
    | none => none
    | some g => g <$> x () -- map
```

Eigentlich ist `seq` für eine Funktion *in* der `Option` definiert wie `map`. Aber `map` verlangt als Argument eine Funktion. Die gibt es aber bei `none` nicht:

Im Fall `none` gibt es gar keine Funktion, die `seq` auf `x` anwenden könnte, deshalb wird `x` in `seq f x` in diesem Fall nicht ausgewertet, also auch nie aufgerufen.

```
#eval some (· + 2) <*> some 2  -- ergibt some 4
#eval (· + 2) <$> some 2       -- ergibt some 4
#eval (none : Option (Nat → Nat)) <*> some 2 -- ergibt none
#eval some (· + 2) <*> none    -- ergibt none
```

Wenn man den Infix-Operator <*> verwendet, dann wird automatisch aus dem rechten Operanden eine Funktion der Unit gemacht:

```
#check [(· + 2)] <*> [1, 2 ,3]
-- Seq.seq [fun x => x + 2] fun x => [1, 2, 3] : List ℕ
#check Seq.seq [(· + 2)] [1, 2 ,3]
-- expected to have type Unit → List ℕ : Type
#check Seq.seq [(· + 2)] (λ _ => [1, 2 ,3])
-- Seq.seq [fun x => x + 2] fun x => [1, 2, 3] : List ℕ
```

9.1.3 Monaden

Eine reine Funktion hängt nur von ihren Aufrufargumenten ab und ergibt für dieselben Argumentwerte stets das identische Ergebnis. Oft ist man aber in einer Situation, wo zusätzlich ein Berechnungskontext im Spiel ist: etwa ein Protokoll des Aufrufs der Funktion, eine Ausgabe oder der Zugriff auf einen globalen Zustand.

Wir reichern dazu eine reine Funktion `f : α → β` um diesen Berechnungskontext an und erhalten eine monadische Funktion `f: α → m β`. Der Typkonstruktor `m` wird dann als Monade bezeichnet.

Wir haben im vorherigen Kapitel schon gesehen, dass wir uns nun natürlich um die Komposition von solchen monadischen Funktionen kümmern müssen, wozu der Infix-Operator >=> dient. (Dieser Operator heißt auch „Fisch“-Operator wegen seines Aussehens oder K eisli-Operator nach dem Schweizer Mathematiker Heinrich Kleisli, nachndem die Kleisli-Kategorie benannt ist, die dem Konzept der Komposition von monadischen Funktionen in der Kategorientheorie entspricht. Ich meine gehört zu haben, dass bei der Entwicklung des Konzepts der Monade diese zunächst als (Kleisli-)Tripel bezeichnet wurde.)

9.1.3.1 Die Typklasse `Monad`

Den Kleisli-Operator kann man auch ausdrücken, wenn man die Funktion `bind` gegeben hat und so wird eine Monade in Lean (und auch in Haskell) definiert:

```
class Functor (f : Type u → Type v) : Type (max (u+1) v) where
  map : {α β : Type u} → (α → β) → f α → f β

class Pure (f : Type u → Type v) where
  pure {α : Type u} : α → f α

class Seq (f : Type u → Type v) : Type (max (u+1) v) where
  seq : {α β : Type u} → f (α → β) → (Unit → f α) → f β

class SeqLeft (f : Type u → Type v) : Type (max (u+1) v) where
  seqLeft : {α β : Type u} → f α → (Unit → f β) → f α

class SeqRight (f : Type u → Type v) : Type (max (u+1) v) where
  seqRight : {α β : Type u} → f α → (Unit → f β) → f β

class Applicative (f : Type u → Type v) extends
  Functor f, Pure f, Seq f, SeqLeft f, SeqRight f where
  map      := fun x y => Seq.seq (pure x) fun _ => y
  seqLeft  := fun a b => Seq.seq (Functor.map (Function.const _) a) b
  seqRight := fun a b => Seq.seq (Functor.map (Function.const _ id) a) b

class Bind (m : Type u → Type v) where
  bind : {α β : Type u} → m α → (α → m β) → m β
```

```
class Monad (m : Type u → Type v) extends Applicative m, Bind m :
  Type (max (u+1) v) where
  map      f x := bind x (Function.comp pure f)
  seq      f x := bind f fun y => Functor.map y (x ())
  seqLeft  x y := bind x fun a => bind (y ()) (fun _ => pure a)
  seqRight x y := bind x fun _ => y ()
```

Man kann eine Monade auch definieren durch

```
m ist ein Funktor
pure : α → m α
join : m (m α) → m a
```

Dabei ist `join` definiert als

```
join (a : m (m α)) : m α := bind a id
```

9.1.3.2 Die Gesetze der Monade

Wir haben die Gesetze der Monade schon im vorherigen Kapitel besprochen. Hier zur Erinnerung:

```
-- Linksidentität
pure x >>= g = g x
-- Rechtsidentität
m >>= pure = m
-- Assoziativität
(m >>= g) >>= h = m >>= (λ x => g x >>= h)
```

Beispiele für diese Gesetze mit `Option`:

```
def plus2 : Nat → Option Nat := fun n => some (n + 2)
def mal3  : Nat → Option Nat := fun n => some (n * 3)

#eval pure 1 >>= plus2    -- some 3
#eval plus2 1             -- some 3

#eval (some 42) >>= pure  -- some 42

#eval ((some 1) >>= plus2) >>= mal3            -- some 9
#eval (some 1) >>= (fun x => plus2 x >>= mal3) -- some 9
```

Die folgende Diskussion eines anderen Blicks auf Monaden beginnen wir mit Beispielen mit `Option`.

Wir definieren eine Funktion `join` (hier speziell `joino` für `Options`). `join` hat als Argument eine „Doppelmonade“ und macht sie „flach“.

```
def joino (o2 : Option (Option α)) : Option α := bind o2 id
#check @joino
-- joino : Option (Option α) → Option α

#eval joino (some (some 42))        -- some 42
#eval joino (some (some (some 42))) -- some (some 42)
```

Wir wollen nun überprüfen wie es sich verhält, wenn wir bei einer „Dreifachmonade“ erste die innere Schicht und dann die äußere oder umgekehrt mit `join` „flach“ machen:

```
#check joino ∘ Option.map joino
#check joino ∘ joino

-- wir beobachten
#eval joino (some (some 42))
-- join macht aus einer Doppelmonade eine Monade
#eval Option.map joino (some (some (some 42)))
-- map join macht aus einer Dreifachmonade eine Doppelmoande
#eval joino (some (some (some 42)))
-- joino auch

#eval (joino ∘ Option.map joino) (some (some (some 42)))  -- some 42
#eval (joino ∘ joino) (some (some (some 42)))             -- some 42
```

Es stellt sich heraus, dass in unserem Beispiel

```
join ∘ map join = join ∘ join
```

gilt, gewissermaßen die Assoziativität des „Flachmachens“.

Nun wollen wir am Beispiel sehen, wie `join` und `map` die Identität bilden können:

```
#eval (joino ∘ Option.map pure) (some 42)  -- some 42
#eval (joino ∘ pure) (some 42)             -- some 42
```

Sowohl `join ∘ map pure` als auch `join pure` sind die Identität.

Nun wollen wir noch sehen, ob sich `join` mit `map` „gut verträgt“:

```
def add1 := (· + 1)

#eval joino (some (some 41))                      -- some 41
#eval Option.map add1 (some 41)                   -- some 42
#eval (Option.map add1 ∘ joino) (some (some 41)) -- some 42

#eval Option.map (Option.map add1) (some (some 41))           -- some (some 42)
#eval joino (some (some 42))                                  -- some 42
#eval (joino ∘ Option.map (Option.map add1)) (some (some 41)) -- some 42
```

Am Beispiel sehen wir, dass es

```
join ∘ map (map f) = map f ∘ join
```

erfüllt ist.

Dies ist natürlich nicht nur in unserem Beispiel so, sondern gilt allgemein.

Mit `map`, `pure` und `join` können wir die Gesetze der Monade so ausdrücken:

```
join ∘ map join = join ∘ join
join ∘ map pure = join ∘ pure = id
join ∘ map (map f) = map f ∘ join
```

Dies erläutert die Verbindung zur mathematischen Definition der Monade:

Eine Monade T auf der Kategorie C ist ein Endofunktor $T : C \to C$ zusammen mit zwei natürlichen Transformationen $\eta : 1_C \to T$ und $\mu : T \circ T \to T$, die folgende Eigenschaften erfüllen:

- $\mu \circ T\mu = \mu \circ \mu T$ (als natürliche Transformationen $T^3 \to T$)
- $\mu \circ T\eta = \mu \circ \eta T = 1_T$ (als natürliche Transformationen $T \to T$)

Was sagt die Wikipedia dazu:

> The first axiom is akin to the associativity in monoids if we think of μ as the monoid's binary operation, and the second axiom is akin to the existence of an identity element (which we think of as given by η). Indeed, a monad on C can alternatively be defined as a monoid in the category $\mathbf{End}_C$ whose objects are the endofunctors of C and whose morphisms are the natural transformations between them, with the monoidal structure induced by the composition of endofunctors.

Dabei entspricht $\eta : 1_C \to T$ gerade `pure` und $\mu : T^2 \to T$ entspricht `join`.

`join ∘ map join = join ∘ join` verlangt die Assoziativität: Hat man 3 Schichten von Monaden, dann ist es egal, ob man erst die äußere oder die innere Schicht mit join flach macht oder umgekehrt.

`join ∘ map pure = join ∘ pure = id` bedeutet, dass startend mit einer Schicht der Monade es egal ist, ob man eine weitere Schicht innen oder außen bildet und dann erneut flach macht.

Und schließlich sagt `join ∘ map (map f) = map f ∘ join`, dass `join` eine natürliche Transformation ist.

Erläuterung:

Unter einer natürlichen Transformation versteht man eine strukturerhaltende Abbildung von Funktoren. Genauer:

Seien **C** und **D** Kategorien und F und G Funktoren von **C** nach **D**. Eine natürliche Transformation $\mu : F \to G$ ist eine Funktion, die jedem Objekt α einen Morphismus μ_α zuordnet, so dass für alle $f : \alpha \to \beta$ in **C** das folgende Diagramm kommutiert (das heißt, dass die Abbildung eines Objekts erst nach unten und dann rechts identisch ist mit erst nach rechts und dann nach unten):

$$\begin{array}{ccc} \alpha & & F\alpha \xrightarrow{\mu_\alpha} G\alpha \\ \downarrow f & & \downarrow Ff \qquad\quad \downarrow Gf \\ \beta & & F\beta \xrightarrow{\mu_\beta} G\beta \end{array}$$

Abbildung 9.2: Natürliche Transformation

In unserer Situation handelt es sich um die Kategorie **Lean**, deren Objekte Typen und deren Morphismen Funktionen sind. Eine Monade ist ein Funktor (mit den genannten zusätzlichen Eigenschaften), den wir in folgender Abbildung als T bezeichnen. Gegeben sei eine Funktion $f : \alpha \to \alpha$, sowie `join`, die Funktion $: T(T\alpha) \to T\alpha$.

Die Eigenschaft `join ∘ map (map f) = map f ∘ join` können wir dann durch das kommutierende Diagramm in Abb. 9.3 ausdrücken.

Daran sehen wir unmittelbar, dass es sich bei `join` um eine natürliche Transformation von $T \circ T \to T$ handelt.

Dies war sehr knapp! Wenn Sie an der mathematischen Definition der Monade interessiert sind, so kann ich wieder auf Bartosz Milewski: Category Theory for Programmer verweisen. Für den Einsatz von Monaden in Lean muss man freilich

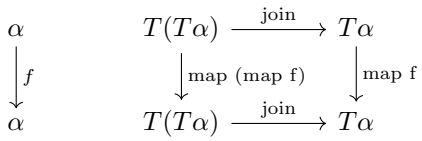

Abbildung 9.3: `join` ist eine natürliche Transformation

den Zusammenhang zur Kategorientheorie nicht unbedingt in aller Tiefe studiert haben.

Übungen:

[09.3] wir haben gesehen, dass wir mit `map`, `pure` und `join` die Gesetze der Monade so ausdrücken können:

```
join ∘ map join = join ∘ join
join ∘ map pure = join ∘ pure = id
join ∘ map (map f) = map f ∘ join
```

Machen Sie einige Beispiel für `List` analog zu den Beispielen für `Option` im Skript. Zusätzlich können Sie die Gleichungen mit Lean beweisen.

9.2 Monaden-Transformierer

Man kann Monaden natürlich selbst von Grund auf neu schreiben. Aber meist ist es viel besser, bereits vorhandene Monaden aus der Bibliothek zu modifizieren, so dass sie den eigenen Anforderungen entsprechen. Dazu dienen *Monad Transformer*.

Ein *Monad Transformer* `T` hat als einen Parameter eine Monade, also typischerweise den Typ `(Type u → Type v) → Type u → Type v` und eventuell zusätzliche Argumente vor der Monade. Eine Instanz für `T m` benötigt eine Instanz von `Monad m`, dadurch kann `Monad T m` selbst als Monade genutzt werden. Außerdem sollte es möglich sein, dass Aktionen vom Typ `m α` zu Aktionen des Typs `T m α` gemacht werden können, damit Aktionen von `m` auch mit `T m` genutzt werden können. Dafür ist die Typklasse `MonadLift` zuständig.

Die Monad-Transformer heißen alle wie die entsprechende Monade mit einem `T` dahinter. Also zB. `ReaderT` oder `StateT`.

In diesem Abschnitt werfen wir nur einen kurzen Blick auf Monad-Transformierer. Ziel ist es nur, an wenigen Beispielen einen ersten Eindruck zu bekommen.

Zuerst betrachten wir ein Beispiel, bei dem Argumente und Optionen auf der Kommandozeile verarbeitet werden.

```
abbrev Arguments := List String

-- ermittelt den Index von s in xs
def indexOf? [BEq α] (xs: List α) (s : α) (start := 0) : Option Nat :=
  match xs with
  | [] => none
  | a :: tail => if a == s then some start else indexOf? tail s (start+1)

def myArguments := ["--input", "--help"]

#eval indexOf? myArguments "--input"   -- some 0
#eval indexOf? myArguments "--help"    -- some 1
#eval indexOf? myArguments "--nothing" -- none

-- eine Option, die ein Argument braucht
def requiredArgument (name : String) : ReaderT Arguments (Except String) String
 ↪  := do
  let args ← read
  let value := match indexOf? args name with
    | some i => if i + 1 < args.length then args[i + 1]! else ""
    | none => ""
  if value == "" then throw s!"Command line argument {name} missing"
  return value

#eval (requiredArgument "--input").run ["--input", "filename"]
-- Except.ok "filename"
#eval (requiredArgument "--input").run ["--input"]
-- Except.error "Command line argument --input missing"
#eval (requiredArgument "--input").run ["--output", "filename"]
-- Except.error "Command line argument --input missing"

#check @requiredArgument
-- requiredArgument : String → ReaderT Arguments (Except String) String
```

`requiredArgument` hat als Argument einen String, nämlich die Option auf der Kom-

mandozeile, von der wir erwarten, dass sie ein zusätzliches Argument hat. Der Typ des Ergebnisses ist eine Monade `ReaderM`, die eine Liste von Strings, die Argumente lesen kann. Der Reader ist entstanden durch eine Erweiterung der Monade `Except ε α`, deren Typ entweder einen Fehler vom Typ `ε` oder ein `Except.ok` vom Typ `α` repräsentiert.

```
-- optionale Argumente
def optionalSwitch (name : String) : ReaderT Arguments (Except String) Bool :=
↪  do
  let args ← read
  return match (indexOf? args name) with
    | some _ => true
    | none => false

#eval optionalSwitch "--help" |>.run myArguments
-- Except.ok true
#eval optionalSwitch "--h" |>.run myArguments
-- Except.ok false
```

Dabei ist `|>.run` ist der Infix-Operator für `(requiredArgument args).run myArguments`.

Das Ergebnis eines Monad-Transformers ist selbst wieder eine Monade, also kann man das mit einem weiteren Monad-Transformer kombinieren. Das obige Beispiel geht weiter:

```
structure Config where
  help    : Bool := false
  verbose : Bool := false
  input   : String := ""
  deriving Repr

abbrev CliConfigM := StateT Config (ReaderT Arguments (Except String))
```

Hier haben wir also zuerst `Except`, darauf `ReaderT` und darauf `StateT`. dies ergibt eine Monade, die wir `m Bool` verwenden:

```
def parseArguments : CliConfigM Bool := do
  let mut config ← get -- mut definiert veränderliche Variablen, siehe unten
  if (← optionalSwitch "--help") then
    throw "Usage: example [--help] [--verbose] [--input <input file>]"
  config := {
      config with
        verbose := (← optionalSwitch "--verbose")
```

```
        input   := (← requiredArgument "--input")
  }
  set config
  return true

def main (args: List String) : IO Unit := do
  let config : Config := { input := "default" }
  match parseArguments |>.run config |>.run args with
  | Except.ok (_, c) => do
    IO.println s!"Processing input '{c.input}' with verbose: {c.verbose}"
  | Except.error s => IO.println s

#eval main ["--help"]
-- Usage: example [--help] [--verbose] [--input <input file>]

#eval main ["--input", "filename"]
-- Processing input 'filename' with verbose: false

#eval main ["--input", "filename", "--verbose"]
-- Processing input 'filename' with verbose: true
#eval main ["--input", "--verbose", "filename"]
-- Processing input 'filename' with verbose: true

-- Anmerkung: Das Beispiel ist nicht gerade robust bezüglich
-- fehlerhafter Eingaben. Es dient hier lediglich der Illustration von
-- Monad Transformer!!
```

9.2.1 Lifting von Monaden

Man kann in Lean Funktionen kombinieren, die unterschiedliche Monaden verwenden durch automatisches Lifting von Monaden. Das kam in obigem Beispiel schon vor, weil wir `optionalSwitch` in `parseArguments` aufgerufen haben, obwohl diese Funktion einen erweiterten Typ hat.

Das folgende Beispiel soll diesen Mechanismus erläutern (siehe auch Christiansen: Monad Transformers - Exceptions):

```
-- Wir können eine Division durch Null mit `Except` absichern:
def divide : Float → Float → Except String Float :=
  λ x y =>
    if y == 0 then throw "Keine Division durch 0"
      else pure (x / y)
```

```
#eval divide 6 3     -- Except.ok 2.000000
#eval divide 6 0     -- Except.error "Keine Division durch 0"
```

Nun wollen wir zählen, wie oft die Funktion aufgerufen wurde. Dazu verwenden wir als Zustand die Zahl, wie oft die Funktion aufgerufen wurde. Wir müssen also dem `run` der Zustandsmonade den aktuellen Zustand übergeben:

```
def divideCounter (x y: Float) : StateT Nat (Except String) Float := do
  modify λ s => s + 1
  divide x y

#eval divideCounter 6 3 |>.run 0 -- Except.ok (2.000000, 1)
#eval divideCounter 6 3 |>.run 4 -- Except.ok (2.000000, 5)
#eval divideCounter 6 0 |>.run 4 -- Except.error "Keine Division durch 0"
```

Das Interessante daran: `divideCounter` gibt den Wert der Division zurück, aber die Funktion hat einen anderen Typ, nämlich noch den Wrapper mit `StateT Nat`.

Das liegt an der Typklasse `MonadLift`:

```
class MonadLift (m : Type u → Type v) (n : Type u → Type w) where
  monadLift : {α : Type u} → m α → n α
```

`monadLift` liftet eine Berechnung von der inneren Monade `m` zur äußeren Monade `n`. Und weil `StateT` eine Instanz dieser Typklasse definiert hat, funktioniert das.

9.3 Mehr an *do*-Notation

In diesem Abschnitt folgen wir der Darstellung der Thematik in David Christiansen: Functional Programming 7.4, aber eteas kürzer. Es ist also sinnvoll, für Details dort nachzuschlagen.

9.3.1 Bedingte Ausdrücke

Normalerweise ist `if ... then ... else` in Lean ein Ausdruck, weshalb der `else`-Zweig erforderlich ist. In der _do_Notation ist auch reines `if` bzw. `unless` möglich.

```
def count (p : α → Bool) (l : List α) : StateM Nat Unit :=
  match l with
  | [] => pure ()
```

```
  | x :: xs => do
    if p x then
      modify (· + 1)
    count p xs

#eval count (fun n => n % 2 == 0) [1, 2] 0
#eval count (fun n => n % 2 == 0) [2, 2] 0
```

9.3.2 Early Return

In imperativen Sprachen gibt es in der Regel das Schlüsselwort `return`, das bewirkt, dass eine Funktion unmittelbar zurückgibt. Das ist mit der *do*-Notation in Lean auch möglich.

Als Beispiel betrachten wir die Funktion `find?`, die in einer Liste nach einem Wert sucht, der ein bestimmtes Prädikat erfüllt.

Diese Funktion könnte man so schreiben:

```
def find?' (p: α → Bool) : List α → Option α
  | [] => none
  | a :: as =>
    if p a then some a else find?' p as

#eval find?' (fun x => x % 2 == 0) [1, 1, 1, 1] -- none
#eval find?' (fun x => x % 2 == 0) [1, 1, 2, 1] -- some 2
```

Mit ‘return ist auch folgende Definition der Funktion möglich:

```
def find? (p: α → Bool) : List α → Option α
  | [] => none
  | a :: as => do
    if p a then return a
    find?' p as

#eval find? (fun x => x % 2 == 0) [1, 1, 1, 1] -- none
#eval find? (fun x => x % 2 == 0) [1, 1, 2, 1] -- some 2
```

Early return kann man auch mit Gewinn bei der Validierung der Kommandozeile von Programmen einsetzen. Das Beispiel aus dem Buch von Christiansen:

```
def main' (argv : List String) : IO UInt32 := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
```

```
  let stderr ← IO.getStderr

  unless argv == [] do
    stderr.putStrLn s!"Expected no arguments, but got {argv.length}"
    return 1

  stdout.putStrLn "How would you like to be addressed?"
  stdout.flush

  let name := (← stdin.getLine).trim
  if name == "" then
    stderr.putStrLn s!"No name provided"
    return 1

  stdout.putStrLn s!"Hello, {name}!"

  return 0

#eval main' []          -- okay
#eval main' ["defalt"]  -- nicht okay
```

9.3.3 Schleifen

Mit `ForM` kann man über eine Schleife über einen Container ausführen. Wir nehmen auch hier ein Beispiel von Christiansen, in dem Vokale und Konsonanten in einem String gezählt werden:

```
structure LetterCounts where
  vowels :     Nat
  consonants : Nat
deriving Repr

inductive Err where
  | notALetter : Char → Err
deriving Repr

def vowels :=
  let lowerVowels := "aeiuoy"
  lowerVowels ++ lowerVowels.map (·.toUpper)

def consonants :=
  let lowerConsonants := "bcdfghjklmnpqrstvwxz"
```

```
  lowerConsonants ++ lowerConsonants.map (·.toUpper )

def countLetters (str : String) : StateT LetterCounts (Except Err) Unit :=
  forM str.toList fun c => do
    if c.isAlpha then
      if vowels.contains c then
        modify fun st => {st with vowels := st.vowels + 1}
      else if consonants.contains c then
        modify fun st => {st with consonants := st.consonants + 1}
    else throw (.notALetter c)

#eval countLetters "alpha" ⟨0, 0⟩
#eval countLetters "1lpha" ⟨0, 0⟩
#eval countLetters "aaaaa" ⟨0, 0⟩
```

Mit `for` kann man sehr bequem Schleifen verwenden quasi wie in einer imperativen Sprache:

```
structure AllLessThan where
  num : Nat
deriving Repr

def AllLessThan.forM [Monad m] (coll : AllLessThan) (action : Nat → m Unit) : m
↪  Unit :=
  let rec countdown : Nat → m Unit
    | 0 => pure ()
    | n + 1 => do
      action n
      countdown n
  countdown coll.num

instance : ForM m AllLessThan Nat where
  forM := AllLessThan.forM

instance : ForIn m AllLessThan Nat where
  forIn := ForM.forIn

def countToThree (n : Nat) : IO Unit := do
  let nums : AllLessThan := ⟨n⟩
  for i in nums do
    if i < 3 then break
    IO.println i
```

```
#eval countToThree 21
#eval countToThree 7
#eval countToThree 2
```

9.3.4 Veränderliche Variablen

In der Funktion `parseArguments`haben wir schon eine veränderliche Variable gesehen, die mit dem Schlüsselwort `mut` definiert wird. Innerhalb eines `do`-Blocks kann man lokale veränderliche Variablen einsetzen. Unter der Hand werden sie zu einer Zustandsmonade, sie sind also syntaktischer Zucker, der das Arbeiten mit lokalen Zustandsvariablen vereinfacht.

Einfaches Beispiel:

Wir können die *do*-Notation nur im Kontext einer Monade verwenden. Wollen wir dies in einer reinen Funktion tun, dann meldet der Lean-Compiler:

> You can use the `do` notation in pure code by writing `Id.run do` instead of `do`, where `Id` is the identity monad.

Deshalb definieren wir

```
def two : Nat := Id.run do
...
```

10

```
def two : Nat := Id.run do
  let mut x := 0
  x := x + 1
  x := x + 1
  return x

#eval two   -- ergibt 2
```

Daraus wird folgender Code:

```
def two : Nat :=
  let block : StateT Nat Id Nat := do
    modify (· + 1)
    modify (· + 1)
    return (← get)
  let (result, _finalState) := block 0
  result

#eval two
```

Veränderliche Variabeln kann man zum Beispiel dafür verwenden, in einer Liste die Elemente zu zählen, die ein bestimmtes Prädikat erfüllen:

```
def cond_count (p : α → Bool) (xs : List α) : Nat := Id.run do
  let mut found := 0
  for x in xs do
    if p x then found := found + 1
  return found

#eval cond_count (fun n => n % 2 == 0) [1, 2, 3, 4, 5, 6]
#eval cond_count (fun n => n % 2 == 0) [1, 3, 5]
#eval cond_count (fun n => n % 2 == 0) [2, 4, 6]
```

Übungen:

[09.4] Schreiben Sie eine Funktion `sfrequencies`, die zu einer Liste von Strings eine `Std.HashMap` erstellt, in der zu den Strings ihre Häufigkeit in der Liste verzeichnet ist. Verwenden Sie dazu `StateM` und `for`. Erstellen Sie außerdem eine Variante `sfrequencies'`, bei der Sie `mut` und `for` einsetzen.

[09.5] Schreiben Sie eine zweite Variante `sfrequencies'`, bei der weder `StateM` noch `for` benötigt wird.

[09.6] Verallgemeinern Sie die Funktion der vorherigen Übung für beliebige Typen zu `frequencies`. Inwieweit muss dabei das „beliebig“ relativiert werden und wie spiegelt sich dies in der Definition der Funktion wieder?

10.1 Zusammenfassung

In diesem Kapitel haben wir gesehen,

- wie Typklassen in Lean verwendet werden, um die Signaturen und Gesetze von **Funktor**, **Applicative Funktor** und schließlich **Monade** aufeinander aufbauend zu spezifizieren. Mit dieser Analyse haben wir unser Verständnis von Monaden vertieft.
- wie man Monaden durch *Monad Transformer* miteinander kombinieren kann und auf diese Weise in Lean bereits vorhandene Monaden dazu nutzen kann, eigene zu entwickeln. Zur Vertiefung dieser Thematik empfiehlt es sich, David Thrane Christiansen: Functional Programming in Lean, Kapitel 7 zu Rate zu ziehen. Denn das Skript gibt nur einen ersten, ganz kurzen Blick darauf, was man mit Monaden-Transformierer erreichen kann.
- wie man mit `forM`, `for` und `mut` in Lean Code schreiben kann, der sich liest wie der einer imperativen Sprache, der aber tatsächlich unter der Hand in funktionale Konzepte umgesetzt wird.

---

11 Abhängige Typen

Ganz allgemein kann man von abhängigen Typen sprechen, wenn es sich um Funktionen von Typen handelt. So kann man das Produkt `Prod α β` als abhängigen Typ sehen. Wir sprechen aber in diesem Fall eher von einem Typkonstruktor. Üblicherweise wird der Begriff „abhängiger Typ“ (*dependent type*) verwendet, wenn der Typ nicht nur von anderen Typen, sondern von *Ausdrücken* abhängig ist.

11.1 Vektor

Das typische Beispiel für einen abhängigen Typ ist ein Vektor mit einer spezifizierten Länge, er hat also einen Typ `α → Nat`.

In der folgenden Definition ist `Vect` als Subtyp von `List` definiert mit der zusätzlichen Bedingungen, dass die Länge der Liste `n` ist.

Wenn man dann einen Vektor definiert, dann handelt es sich um eine Struktur und man muss die Konstruktorklammern für die Definition von Werten verwenden. Angegeben werden dann die Liste und ein Beweis, dass die Länge korrekt ist.

```
def Vect (α : Type u) (n : Nat) :=
  { l : List α // l.length = n }

#print Vect

def v1 : Vect Nat 3 := ⟨[1, 2, 3], rfl⟩
#eval v1
```

Ein großer Vorteil von abhängigen Typen besteht darin, dass man für bestimmte Funktionen Vorbedingungen in den Typ packen kann.

So ist zum Beispiel die Funktion `head` nur definiert für Vektoren der Längen > 1. Dies kann man nun in der Signatur der Funktion ausdrücken:

```
def head : Vect α (Nat.succ n) → α
  | ⟨a :: _, _⟩ => a
```

```
#eval head v1 -- ergibt 1

def nil : Vect α 0 :=
  ⟨[], rfl⟩

#eval head nil
--   application type mismatch ...

def map (f : α → β) : Vect α n → Vect β n
  | ⟨l, h⟩ => ⟨List.map f l, by simp [*]⟩

#eval map (· + 1) v1
```

Eine Funktion, bei der sich die Länge der Vektors nicht ändert, ist für die Typüberprüfung kein großes Problem.

Etwas schwieriger ist es dann schon bei Funktionen, bei denen sich die Länge ändert. Betrachten wir das Aneinanderfügen von Vektoren fixer Länge:

```
def append {n m : Nat} : Vect α n → Vect α m → Vect α (n + m)
  | ⟨l₁, h₁⟩, ⟨l₂, h₂⟩ => ⟨l₁ ++ l₂, by simp [*]⟩

def v2 : Vect Nat 2 := ⟨[4, 5], rfl⟩

#eval append v1 v2
-- [1, 2, 3, 4, 5]
```

Man sieht, dass das in Lean sehr schön gelöst ist: Denn die Beweisverpflichtung über die Länge, für die wir den Vektoren immer einen Beweis der Korrektheit der Länge mitgeben, erlaubt es, dass für die Typprüfung in Lean der Mechanismus des Beweisens in Lean selbst verwendet werden kann.

Es gibt aber auch noch komplexere Fälle: Stellen wir uns eine Funktion `filter` vor. Also zum Beispiel für

```
filter (λ x => Nat.mod x 2 == 0) v1
```

Das geht mit Listen problemlos:

```
#def even : Nat → Bool :=
  λ x => Nat.mod x 2 == 0
#eval List.filter even [1, 2, 3, 4, 5] -- [2, 4]
```

Das Problem ist nun: Versucht man eine solche Funktion für Vektoren zu definieren, dann muss man ja den Typ des Ergebnisses angeben, also etwa

```
def filter (p : α → Bool) : Vect α n → Vect α ??
```

Die Länge des Ergebnisses hängt von dem Prädikat ab, das wir beim Filtern verwenden. Hier wir die Sache kompliziert, siehe Carlo Angiuli, Daniel Gratzer.

Ich weiß nicht, wie man das in Lean lösen kann.

Mit diesen knappen Bemerkungen wollen wir es erst mal bewenden lassen. Mehr zum Programmieren mit abhängigen Typen findet man auch in David Thrane Christiansen: Functional Programming in Lean, Kapitel 7.

Aber wir wollen noch den Zusammenhang von abhängigen Typen zur Logik betrachten:

11.2 Abhängige Typen

11.2.1 Abhängiger Funktionstyp (Pi-Typ)

Eine Funktion, bei der der Typ der Funktionswerte abhängig ist von ihrem Argument ist ein abhängiger Funktionstyp und wird üblicherweise Pi-Typ genannt:

Gegeben eine Familie β von Typen `β : α → u`, wobei `u` ein Universum von Typen ist. (Tatsächlich hat Lean eine Hierarchie von Universen, das ist aber für das Verstehen des Konzepts der abhängigen Funktionstyp nicht wesentlich.) Man konstruiert den Typ der abhängigen Funktionen, der in der Literatur so geschrieben wird: `Π x : α, β x` – daher der Name „Pi-Typ“.

Ein Pi-Typ ist also eine Familie von Typen `β a` für alle `a : α`. Wenn β gar nicht von `a` abhängt, dann handelt es sich einfach um den Typ `α → β`. Der Pi-Typ ist also eine Verallgemeinerung einer Funktion.

Nehmen wir zur Veranschaulichung an, dass der Typ α endlich ist, etwa $\{a_1, a_2, a_3\}$ und eine Funktion `f` bildet ab: $f\ a_1 = (b_1 : \beta\ a_1)$, $f\ a_2 = (b_2 : \beta\ \alpha_2)$ und $f\ a_3 = (b_3 : \beta\ \alpha_3)$. Dann kann man die Funktion `f` sehen als das Tupel (b_1, b_2, b_3) und somit $f \in \beta\ a_1 \times \beta\ a_2 \times \beta\ a_3$. Andererseits ist jeder Wert auf der rechten Seite für eine Funktion dieses Typs möglich, d.h. `Π x : α, β x` kann man identifizieren mit $\beta\ a_1 \times \beta\ a_2 \times \beta\ a_3$.

Für den Vektor aus dem vorherigen Abschnitt ist `Π n : Nat, Vect α n` also ein Pi-Typ, denn für alle `n : Nat` haben wir den Typ `Vect α n` von Vektoren mit Elementen vom Typ `α` und Länge `n`.

11.2.2 Abhängiger Paartyp (Sigma-Typ)

Dual zum Pi-Typ gibt es den Sigma-Typ: ein geordnetes Paar, in dem der Typ der zweiten Komponente abhängig ist vom Typ der ersten.

Wieder gegeben eine Familie `β` von Typen `β : α → u`, wobei `u` ein Universum von Typen ist. Dann ist `Sigma β`, auch geschrieben als `Σ a : α, β a`, der Typ der Paare deren erste Komponente `a : α` und deren zweite Komponente `b : β a` ist – der Typ der zweiten Komponente hängt vom Wert der ersten Komponente ab.

Nehmen wir wieder zur Veranschaulichung an, dass `α` endlich ist wie oben, dann ist `Sigma β` gerade `{(a₁, β a₁), (a₂, β a₂), (a₃, b a₃)}`. Diese Menge kann man identifizieren mit der disjunkten Vereinigung der zweiten Komponenten, also mit `{β a₁} ⊕ {β a₂} ⊕ {β a₃}`.

Die Veranschaulichung zeigt, das der Pi-Typ die Verallgemeinerung des Produkttyps ist und der Sigma-Typ dual dazu die Verallgemeinerung des Summentyps.

11.3 Abhängige Typen und Logik

Der *Curry-Howard-Isomorphimus*, der auch gerne als *PAT-Prinzip* für *PAT = Propositions as Types* bezeichnet wird, besteht darin, dass wir eine logische Aussage `p : Prop` als Typ auffassen und ein Element dieses Typs `t : p` als Beweis für die logische Aussage.

In dieser Korrespondenz sind die Operatoren der Aussagenlogik so zu interpretieren:

Die *Implikation* `p → q` bedeutet, dass die Aussage `q` aus `p` folgt. Gibt es nun eine Funktion `p → q`, die Elemente des Typs `p` auf solche des Typs `q` abbilden, dann können wir das interpretieren als: Die Funktion leitet aus einem Beweis für `p` einen Beweis von `q` her. Die Implikation in der Logik entspricht also gerade der Anwendung einer Funktion.

Die *Konjunktion* `p ∧ q` ist genau dann wahr, wenn beide Aussagen wahr sind. Übertragen in das Typsystem entspricht das gerade dem Produkttyp: Gibt es ein Paar von

Elementen `(t₁, t₂) ∈ p × q`, dann ist die korrespondieren Aussage `p ∧ q` wahr und vice versa.

Die *Disjunktion* `p ∨ q` ist genau dann wahr, wenn mindestens eine der beiden Aussagen wahr ist. Übertragen in das Typsystem entspricht das gerade dem Summentyp `p ⊕ q`, der als disjunkte Vereinigung von `p` und `q` genau dann ein Element hat, wenn mindestens einer der Summanden eines hat.

Die *Negation* `¬ p` definieren wir als `p → False`, d.h. eine Aussage ist genau dann nicht wahr, wenn sich der Widerspruch aus ihr ableiten lässt. Damit ist die Negation auf die Implikation zurückgeführt.

In der Prädikatenlogik kommen die Quantoren und die Gleichheit von Termen hinzu und hier kommen die abhängigen Typen ins Spiel:

Der Zusammenhang zur Logik besteht darin, dass für logische Aussagen der Pi-Typ gerade dem Allquantor entspricht und der Sigma-Typ dem Existenzquantor.

Ist `α` ein beliebiger Typ, dann kann man ein unäres Prädikat `p` über `α` repräsentieren durch ein Objekt vom Typ `α → Prop`. Der *Allquantor* `∀ x : α, p x` entspricht also genau dem Pi-Typ für `Prop`.

Und somit: Finden wir einen Ausdruck `t` vom Typ `p x` für ein beliebiges `x : α`, dann haben wir eine Funktion `λ x => t : (x : α) → p x`, d.h. einen Beweis für die All-Aussage.

Wir können den Allquantor als Verallgemeinerung der Konjunktion (von ∧) sehen. Dies entspricht genau dem, dass der Pi-Typ die Verallgemeinerung des Produkttyps ist.

Zum *Existenzquantor*: Wenn wir einen Term vom Typ `Σ (x : α), p a` finden können, dann haben wir ein Paar, dessen erste Komponente `x` ist und die zweite das Prädikat `p` erfüllt: `x` ist also ein Zeuge für `p x` und wir haben beweisen, dass `∃ x : α, p x` gilt.

Wir können den Existenzquantor als Verallgemeinerung der Disjunktion (von ∨) sehen. Dies entspricht genau dem, dass der Sigma-Typ die Verallgemeinerung des Summentyps ist.

Ferner haben wir in der Prädikatenlogik die *Gleichheit*, die in Lean über die Typklasse `Eq` definiert ist und über die in der Dokumentation zu lesen ist:

> [Eq:] The equality relation. It has one introduction rule, Eq.refl. We use a = b as notation for Eq a b. A fundamental property of equality is that it is an

> equivalence relation.

> Equality is much more than an equivalence relation, however. It has the important property that every assertion respects the equivalence, in the sense that we can substitute equal expressions without changing the truth value. That is, given h1 : a = b and h2 : p a, we can construct a proof for p b using substitution: Eq.subst h1 h2.

Abhängige Funktionstypen machen also die Curry-Howard-Korrespondenz zwischen Logik und Typtheorie perfekt:

Logik	Typtheorie
∧	`Prod`
∨	`Sum`
→	→
¬ p	p → False
∀	Pi-Typ
∃	Sigma-Typ
=	Eq, =

11.4 Zusammenfassung

In diesem letzten Kapitel der Veranstaltung haben wir einen ganz kurzen Blick auf abhängige Typen geworfen. Es ging dabei

- am Beispiel des **Vektor**s darum, zu sehen, wie man abhängige Typen in Lean definieren kann und wie man diese Typen einsetzen kann. Wir haben dabei auch gesehen, dass dies recht komplex werden kann. Wie erwähnt: Mehr zum Programmieren mit abhängigen Typen findet man auch in David Thrane Christiansen: Functional Programming in Lean, Kapitel 7.
- um den Zusammenhang zwischen Typtheorie und Logik. Dazu haben wir die Verallgemeinerungen des Produkt- und des Summentyps zu **Pi-Typ** und den **Sigma-Typ** eingeführt und gesehen, dass diese gerade dem All- und dem Existenzquantor in der Logik entsprechen.

12 Anhang

12.1 Einige Infix-Operatoren

Operator	Bedeutung
α × β	Prod α β
α ⊕ β	Sum α β
f <| x	Pipelining: f x
x |> f	Pipelining: f x
oa <|> ob	orElse oa () → ob
f <$> l	map: map f l
`ma >>= f`	bind: bind ma f
`f <<= ma`	bind: bind f ma
(mf >=> mg) a	kleisli: Komposition monadischer Funktionen
mf <*> mx	seq: seq mf (λ _ => mx)
ma *> mb	seqRight
ma >* mb	seqLeft

```
-- Prod
#check Nat × Int
#check (2, -2)

-- Sum
#check Nat ⊕ String
#check (Sum.inl 42 : Nat ⊕ String)
#check (Sum.inr "one" : Nat ⊕ String)
```

```
-- pipelining
#eval (· * 2) ((· + 1) 4)      -- ergibt 10
#eval (· * 2) <| (· + 1) <| 4  -- ergibt 10
#eval 4 |> (· * 2) |> (· + 1)  -- ergibt 9

-- orelse
#eval (some 1) <|> (some 2)    -- ergibt some 1
#eval none <|> (some 2)        -- ergibt some 1
#eval (some 1) <|> none        -- ergibt some 1

-- map
#eval List.map (· * 2) [1, 2, 3]
#eval [1, 2, 3].map (· * 2)
#eval (· * 2) <$> [1, 2, 3]

-- bind
def sum2 (ns: List Nat) : Option Nat :=
  bind (List.get? ns 0)
    (fun n₁ => bind (List.get? ns 1)
      (fun n₂ => pure (n₁ + n₂)))
#eval sum2 [1] -- ergibt none
#eval sum2 [1, 1] -- some 2

def sum2' (ns: List Nat) : Option Nat :=
  (List.get? ns 0) >>=
    (fun n₁ => (List.get? ns 1) >>=
      (fun n₂ => pure (n₁ + n₂)))
#eval sum2' [1] -- ergibt none
#eval sum2' [1, 1] -- some 2

def sum2'' (ns: List Nat) : Option Nat := do
  let n₀ ← (List.get? ns 0)
  let n₁ ← (List.get? ns 1)
  pure (n₀ + n₁)
#eval sum2'' [1] -- ergibt none
#eval sum2'' [1, 1] -- some 2

-- kleisli
def first (s: String) : Option String :=
  if (s != "") then some s else none
def second (s: String) : Option Int :=
  if (s != "hallo") then some (s.length) else none
```

```
#eval Bind.kleisliRight first second "hallo"    -- ergibt none
#eval Bind.kleisliRight first second ""         -- ergibt none
#eval Bind.kleisliRight first second "kleisli"  -- ergibt some 7

#eval (first >=> second) "hallo"                -- ergibt none
#eval (first >=> second) "kleisli"              -- ergibt none

-- seq
#eval Seq.seq (Seq.seq (pure (· * ·)) (λ _ => (some 2))) (λ _ => some 21)
#eval pure (· * ·) <*> some 2 <*> some 21       -- ergibt 42

-- Beispiele für sinnvollen Einsatz von seq
-- Paarweise Kombination zweier Listen
#eval [Prod.mk] <*> [1, 2, 3] <*> [11, 12, 13]
-- Summen der Paare aus zwei Listen
#eval [(· + · )] <*> [1, 2, 3] <*> [11]
-- Tripel aus drei Listen
#eval (· , · , · ) <$> [1, 2] <*> [3, 4] <*> [5, 6]
-- Summen der Tripel
#eval (· + · + · ) <$> [1, 2] <*> [3, 4] <*> [5, 6]
-- Addiere Komponenten 2 und 3, und multipliziere mit Komponanten1
#eval (· * · + · ) <$> [1, 2] <*> [3, 4] <*> [5, 6]
```

12.2 Spezielle Symbole in Beweisen

Symbol	Bedeutung
t_1 ; t_2	Taktik t_1 gefolgt von t_2
t_1 <;> t_2	t_1 und danach t_2 auf alle Subziele
first | t_1 | t_2 …	t_1, t_2 etc bis eine Taktik klappt
h1 ▸ h2	Substitution von h1 in h2 (Eingabe \ gefolgt von t)
‹e›	kurz für (by assumption : e)

```
-- ;
example (p q : Prop) (hp : p) : p ∨ q := by
  apply Or.inl; assumption

-- <;>
```

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor <;> assumption

-- |
example (p q : Prop) (hp : p) : p ∨ q := by
  first | apply Or.inl; assumption | apply Or.inr; assumption

example (p q : Prop) (hq : q) : p ∨ q := by
  first | apply Or.inl; assumption | apply Or.inr; assumption

-- subst (triangle)
example (α : Type) (a b : α) (p : α → Prop) :
  (a = b) ∧ p a → p b := by
    intro h
    apply h.left ▸ h.right
```

12.3 Einige Taktiken

Taktik	Kurzbeschreibung
ac_rfl	beweist Gleichheit modulo des Assoziativ- und des Kommutativgesetzes (Doku)
apply	wendet eine Funktion an (Doku, siehe auch)
assumption	versucht das Beweisziel durch eine der gegebenen Voraussetzungen zu zeigen (Doku, siehe auch)
by_cases	erstellt aus einem Ziel `P` zwei mit der Annahme `P` und `¬ P` – ist also klassische Logik (Doku, siehe auch)
cases	erstellt bei einem induktiven Typ ein Subbeweisziel für jeden Konstruktor, schließt dabei unmögliche Fälle automatisch (Doku)
contradiction	findet einen offensichtlichen Widerspruch in den Voraussetzungen (Doku)
constructor	löst bei einem induktiven Typ das Beweisziel mit dem ersten passenden Konstruktor (Doku, siehe auch)

Taktik	Kurzbeschreibung
decide	versucht das Beweisziel durch eine Instanz von `Decidable` zu reduzieren, geht nur ohne lokale oder Metavariablen (Doku)
dsimp	Simplifier, der nur Sätze anwendet, die per definitorischer Gleichheit gelten (Doku)
exact	beweist das Ziel, wenn dessen Typ exakt mit der angegebenen Voraussetzung übereinstimmt (Doku, siehe auch)
funext	wendet *function extensionality* an und erzeugt neue Voraussetzungen (Doku, siehe auch)
have	führt eine neue Voraussetzung ein (die dann auch bewiesen werden muss) (Doku, siehe auch)
induction	beginnt einen Induktionsbeweis (Doku, siehe auch)
intro	zieht aus dem Beweisziel Voraussetzungen heraus (Doku, siehe auch)
linarith	versucht einen Widerspruch in den Voraussetzungen zu finden, die lineare (Un)gleichungen sind (Doku, siehe auch)
positivity	kann Ziele lösen, die behaupten, dass ein Wert positiv ist (Doku, siehe auch)
rename	bennent eine Voraussetzung um, die einen bestimmten Typ hat (Doku)
revert	das Inverse zu `intro`, setzt Voraussetzung als linke Seite einer Implikation vor das Beweisziel (Doku)
rewrite	wendet eine Identität/Äquivalenz auf das Ziel an und schreibt dieses entsprechend um (Doku, siehe auch)
rfl	beweist Gleichheit (Doku, siehe auch)
rw	wie rewrite, jedoch mit automatischen `rfl` danach (Doku, siehe auch und Theorem Proving in Lean)

Taktik	Kurzbeschreibung
simp	Simplifier, verwendet Lemmas und Voraussetzungen an, um das Ziel zu vereinfachen (Doku, siehe auch, Theorem Proving in Lean und Lean Community Doku)
split	zerlegt if-then-else und match in einzelne Fälle (Doku)
subst	substituiert einen Ausdruck durch einen der = ist (Doku)
trivial	versucht Taktiken wie `rfl`, `contradiction` uÄ, um das Ziel zu zeigen (Doku)
unfold	setzt Definitionen ein = entfaltet sie (Doku)