Fortgeschrittene Programmierung
Vorlesung
WS 09,10; SS 12–14, 16–19, 21

Johannes Waldmann, HTWK Leipzig

9. Juli 2021

Einleitung

Programmierung im Studium bisher

- 1. Sem: Modellierung (formale Spezifikationen (von konkreten und abstrakten Datentypen))
- 1./2. Sem Grundlagen der (AO) Programmierung
  - imperatives Progr. (Programm ist Folge von Anweisungen, bewirkt Zustandsänderung)
  - strukturiertes P. (genau ein Eingang/Ausgang je Teilp.)
  - objektorientiertes P. (Interface $=$ abstrakter Datentyp, Klasse $=$ konkreter Datentyp)
- 2. Sem: Algorithmen und Datenstrukturen (Spezifikation, Implementierung, Korrektheit, Komplexität)
- 3. Sem: Softwaretechnik (industrielle Softwareproduktion)

Worin besteht jetzt der Fortschritt?

- *deklarative* Programmierung

  (Programm *ist* ausführbare Spezifikation)

- insbesondere: *funktionale* Programmierung

  Def: Programm berechnet *Funktion*
  f : Eingabe $\mapsto$ Ausgabe,

  (kein Zustand, keine Zustandsänderungen)

- – Daten (erster Ordnung) sind Bäume
  – Programm ist Gleichungssystem
  – Programme sind auch Daten (höherer Ordnung)

- ausdrucksstark, sicher, effizient, parallelisierbar

Formen der deklarativen Programmierung

- funktionale Programmierung: `foldr (+) 0 [1,2,3]`

```
foldr f z l = case l of
    [] -> z ; (x:xs) -> f x (foldr f z xs)
```

- logische Programmierung: `append(A,B,[1,2,3]).`

```
append([],YS,YS).
append([X|XS],YS,[X|ZS]):-append(XS,YS,ZS).
```

- Constraint-Programmierung

```
(set-logic QF_LIA) (set-option :produce-models true
(declare-fun a () Int) (declare-fun b () Int)
(assert (and (>= a 5) (<= b 30) (= (+ a b) 20)))
(check-sat) (get-value (a b))
```

Definition: Funktionale Programmierung

- Rechnen $=$ Auswerten von Ausdrücken (Termbäumen)
- Dabei wird ein *Wert* bestimmt

  und es gibt keine (versteckte) *Wirkung*.

  (engl.: side effect, dt.: Nebenwirkung)
- Werte können sein:
  - "klassische" Daten (Zahlen, Listen, Bäume...)

    ```
    True :: Bool, [3.5, 4.5] :: [Double]
    ```
  - Funktionen (Sinus, ...)

    ```
    [sin, cos] :: [Double -> Double]
    ```
  - Aktionen (Datei lesen, schreiben, ...)

    ```
    readFile "foo.text" :: IO String
    ```

Softwaretechnische Vorteile

. . . der funktionalen Programmierung

- Beweisbarkeit: Rechnen mit Programmen
  wie in der Mathematik mit Termen

- Sicherheit: es gibt keine Nebenwirkungen
  und Wirkungen sieht man bereits am Typ

- Aussdrucksstärke, Wiederverwendbarkeit: durch
  Funktionen höherer Ordnung (sog. Entwurfsmuster)

- Effizienz: durch Programmtransformationen im Compiler,

- Parallelität: keine Nebenwirkungen $\Rightarrow$ keine *data races*,
  fktl. Programme sind *automatisch parallelisierbar*

Beispiel Spezifikation/Test

```
import Test.LeanCheck

append :: forall t . [t] -> [t] -> [t]
append [] y = y
append (h : t) y = h : (append t y)

associative f =
   \ x y z -> f x (f y z) == f (f x y) z
commutative f = \ x y -> ...

test = check
  (associative (append::[Bool]->[Bool]->[Boo
```

Übung: Kommutativität (formulieren und testen)

Beispiel Verifikation

```
app :: forall t . [t] -> [t] -> [t]
app [] y = y
app (h : t) y = h : (app t y)

Lemma: app x (app y z) .=. app (app x y) z

Proof by induction on List x
  Case []
    To show: app [] (app y z) .=. app (app [] y) z
  Case h:t
    To show: app (h:t) (app y z) .=. app (app (h:t) y
    IH: app t (app y z) .=. app (app t y) z
```

CYP `https://github.com/noschinl/cyp`,

ist vereinfachte Version

von Isabelle `https://isabelle.in.tum.de/`

Beispiel Parallelisierung (Haskell)

Klassische Implementierung von Mergesort

```
sort :: Ord a => [a] -> [a]
sort [] = [] ; sort [x] = [x]
sort xs =  let ( left,right ) = split xs
               sleft  = sort left
               sright = sort right
           in  merge sleft sright
```

wird parallelisiert durch *Annotationen*:

```
  sleft  = sort left
             `using` rpar `dot` spineList
  sright = sort right `using` spineList
```

vgl. `http://thread.gmane.org/gmane.comp.lang.haskell.parallel/181/focus=202`

Beispiel Parallelisierung (C#, PLINQ)

- Die Anzahl der 1-Bits einer nichtnegativen Zahl:

```
Func<int,int>f =
    x=>{int s=0; while(x>0){s+=x%2;x/=2;}return s;}
```

- $\sum_{x=0}^{2^{26}-1} f(x)$ `Enumerable.Range(0,1<<26).Select(f).Sum()`

- automatische parallele Auswertung, Laufzeitvergleich:

```
Time(()=>Enumerable.Range(0,1<<26).Select(f).Sum())
Time(()=>Enumerable.Range(0,1<<26).AsParallel()
                                  .Select(f).Sum())
```

vgl. *Introduction to PLINQ* `https://msdn.microsoft.com/en-us/library/dd997425(v=vs.110).aspx`

Softwaretechnische Vorteile

. . . der statischen Typisierung

> The language in which you write profoundly affects the design of programs written in that language.
>
> For example, in the OO world, many people use UML to sketch a design. In Haskell or ML, one writes type signatures instead. Much of the initial design phase of a functional program consists of writing type definitions.
>
> **Unlike UML, though, all this design is incorporated in the final product, and is machine-checked throughout.**

Simon Peyton Jones, in: Masterminds of Programing, 2009;

`http://shop.oreilly.com/product/9780596515171.do`

Deklarative Programmierung in der Lehre

- funktionale Programmierung: diese Vorlesung
- logische Programmierung: in *Angew. Künstl. Intell.*
- Constraint-Programmierung: als Master-Wahlfach

Beziehungen zu weiteren LV: Voraussetzungen

- Bäume, Terme (Alg.+DS, Grundlagen Theor. Inf.)
- Logik (Grundlagen TI, Softwaretechnik)

Anwendungen:

- Softwarepraktikum
- weitere Sprachkonzepte in *Prinzipien v. Programmiersprachen*
- *Programmverifikation* (vorw. f. imperative Programme)

Konzepte und Sprachen

Funktionale Programmierung ist ein *Konzept*.
Realisierungen:

- in prozeduralen Sprachen:
  - Unterprogramme als Argumente (in Pascal)
  - Funktionszeiger (in C)
- in OO-Sprachen: Befehlsobjekte
- Multi-Paradigmen-Sprachen:
  - Lambda-Ausdrücke in C#, Scala, Clojure
- funktionale Programmiersprachen (LISP, ML, Haskell)

Die Erkenntnisse sind sprachunabhängig.

- A good programmer can write LISP in any language.
- Learn Haskell and become a better Java programmer.

Gliederung der Vorlesung

- Terme, Termersetzungssysteme algebraische Datentypen, Pattern Matching, Persistenz
- Funktionen (polymorph, höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
- Typklassen zur Steuerung der Polymorphie
- Bedarfsauswertung, unendl. Datenstrukturen (Iterator-Muster)
- funktional-reaktive Programmierung (deklarative interaktive Programme)
- weitere Entwurfsmuster
- Code-Qualität, Code-Smells, Refactoring

Softwaretechnische Aspekte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme

  Scale: case class, Java: Entwurfsmuster Kompositum, immutable objects, das Datenmodell von Git

- Funktionen (höherer Ordnung), Lambda-Kalkül, Rekursionsmuster

  Lambda-Ausdrücke in C#, Entwurfsmuster Besucher Codequalität, code smells, Refaktorisierung

- Typklassen zur Steuerung der Polymorphie

  Interfaces in Java/C# , automatische Testfallgenerierung

- Bedarfsauswertung, unendl. Datenstrukturen

  Iteratoren, Ströme, LINQ

Literatur (allgemein)

- wissenschaftliche Quellen zur aktuellen Forschung und Anwendung der funktionalen Programmierung
  - Journal of Functional Programming (CUP) `https://www.cambridge.org/core/journals/journal-of-functional-programming`
  - Intl. Conference Functional Programming (ACM SIGPLAN) `https://www.icfpconference.org/`
  - Intl. Workshop Trends in Functional Programming in Education `https://wiki.tfpie.science.ru.nl/`

- `http://haskell.org/` (Sprachstandard, Werkzeuge, Bibliotheken, Tutorials),

Literatur (speziell diese VL)

- Skript aktuelles Semester `http://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
- How I Teach Functional Programming `https://imweb.imn.htwk-leipzig.de/~waldmann/talk/17/wflp/`
- Kriterium für Haskell-Tutorials und -Lehrbücher:
  - wo werden `data` (benutzerdefinerte algebraische Datentypen) und `case` (pattern matching) erklärt?

    Je später, desto schlechter!

Alternative Quellen

- – Q: Aber in Wikipedia/Stackoverflow steht, daß . . .
  – A: Na und.
- Es mag eine in Einzelfällen nützliche Übung sein, sich mit dem Halbwissen von Nichtfachleuten auseinanderzusetzen. (Aber `https://xkcd.com/386/`)
- In VL und Übung verwenden und diskutieren wir die durch Dozenten/Skript/Modulbeschreibung vorgegebenen Quellen (Lehrbücher, referierte Original-Artikel, Standards zu Sprachen und Bibliotheken)
- . . . gilt entsprechend für Ihre Bachelor- und Master-Arbeit.
- Wikipedia: benutzen—ja (um Primärquellen zu finden), zitieren—nein (ist keine wissenschaftliche Quelle).

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- Hausaufgaben
  - gruppenweise: schriftliche Aufgaben: anmelden (Wiki), diskutieren (Issue-Tracker), vorrechnen (in der jeweils nächsten Übung)
  - individuell (jeweils 2 Wochen Bearbeitungszeit)

    `https://autotool.imn.htwk-leipzig.de/new/`
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten der Übungsaufgaben.
  - Vorrechnen: 3 mal,
  - Autotool: 50 Prozent der Pflicht-Aufgaben,
- Prüfung: digitale Hausarbeit (autotool)

Hinweise zu Fernlehre

- allgemeines: `https://imweb.imn.htwk-leipzig.de/~waldmann/etc/fernlehre/`
- Software (Compiler) im Pool (Z423) installiert, von außen erreichbar (ssh, tmux); private Installation empfohlen (Details in Gitlab-Projekt), aber nicht notwendig
- Schrift in Präsent.: groß (Ctrl-Plus), Schwarz auf Weiß.
- Datenschutz:
  - vor Screen-Sharing: nachdenken und aufräumen
  - vor Benutzung von: Browser, Suchmaschine, sogenannten sozialen Netzwerken: nachdenken!!

vgl. `https://imweb.imn.htwk-leipzig.de/~waldmann/edu/ws20/inf/folien/#(11)`

Übungen

- Benutztung Rechnerpool (ssh, tmux, ghci)
- Auf wieviele Nullen endet die Fakultät von 100?
  - Benutze `foldr` zum Berechnen der Fakultät.
  - Beachte polymorphe numerische Literale. (Auflösung der Polymorphie durch Typ-Annotation.) Warum ist 100 Fakultät als `Int` gleich 0?
  - Welches ist der Typ der Funktion `takeWhile`? Beispiel:
    ```
    odd 3 ==> True ; odd 4 ==> False
    takeWhile odd [3,1,4,1,5,9] ==> [3,1]
    ```
  - ersetze in der Lösung `takeWhile` durch andere Funktionen des gleichen Typs (suche diese mit Hoogle), erkläre Semantik

- typische Eigenschaften dieses Beispiels (nachmachen!)
  statische Typisierung, Schachtelung von Funktionsaufrufen, Funktion höherer Ordnung, Benutzung von Funktionen aus Standardbibliothek (anstatt selbstgeschriebener).
- schlechte Eigenschaften (vermeiden!)
  Benutzung von Zahlen und Listen (anstatt anwendungsspezifischer Datentypen) vgl.
  `http://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`

- Haskell-Entwicklungswerkzeuge

  - Compiler, REPL: ghci (Fehlermeldungen, Holes)
  - API-Suchmaschine `http://www.haskell.org/hoogle/`

  - Editor: Emacs `http://xkcd.org/378/`,
    IDE? gibt es, brauchen wir (in dieser VL) nicht
    `https://hackage.haskell.org/package/haskell-language-server`

- Softwaretechnik im autotool: `http://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/se/`

Aufgaben

(im SS21: Aufgaben 2, 3 bis KW 15)

1. zu: E. W. Dijkstra: *Answers to Questions from Students of Software Engineering* (Austin, 2000) (EWD 1035)

   - „putting the cart before the horse“
     - übersetzen Sie wörtlich ins Deutsche,
     - geben Sie eine entsprechende idiomatische Redewendung in Ihrer Muttersprache an,
     - wofür stehen *cart* und *horse* hier konkret?

2. sind die empfohlenen exakten Techniken der Programmierung für große Systeme anwendbar?

   Erklären Sie „lengths of . . . grow not much more than linear with the lengths of . . .“.

- Welche Längen werden hier verglichen?

Modellieren Sie das System als Graph, die Knoten sind die Komponenten, die Kanten sind deren Beziehungen (direkte Abhängigkeiten).

- Welches asymptotische Wachstum ist bei undisziplinierter Entwicklung des Systems zu befürchten?
- Welche Graph-Eigenschaft impliziert den linearen Zusammenhang?
- Wie gestaltet man den System-Entwurf, so daß diese Eigenschaft tatsächlich gilt? Welchen Nutzen hat das für Entwicklung und Wartung?

3. Über ein Monoid $(M, \circ, 1)$ mit Elementen $a, b \in M$ (sowie

eventuell weiteren) ist bekannt: $a^2 = b^2 = (ab)^2 = 1$.

Dabei ist ab eine Abkürzung für $a \circ b$ und a^2 für aa, usw.

- Geben Sie ein Modell mit $1 \neq a \neq b \neq 1$ an.
- Überprüfen Sie $ab = ba$ in Ihrem Modell.
- Leiten Sie $ab = ba$ aus den Monoid-Axiomen und gegebenen Gleichungen ab.

Das ist eine Übung zur Wiederholung der Konzepte *abstrakter* und *konkreter* Datentyp sowie *Spezifikation*.

4. im Rechnerpool live vorführen:

- ein Terminal öffnen
- `ghci` starten (in der aktuellen Version), Fakultät von 100 ausrechnen

- Datei `F.hs` mit Texteditor anlegen und öffnen, Quelltext `f = ...` (Ausdruck mit Wert `100!`) schreiben, diese Datei in `ghci` laden, `f` auswerten

Dabei wg. Projektion an die Wand:

Schrift 1. groß genug und 2. schwarz auf weiß.

Vorher Bildschirm(hintergrund) aufräumen, so daß bei Projektion keine personenbezogenen Daten sichtbar werden. Beispiel: `export PS1="$ "` ändert den Shell-Prompt (versteckt den Benutzernamen).

Wer (unsinnigerweise) eigenen Rechner im Pool benutzt:

- Aufgaben wie oben *und*
- `ssh`-Login auf einen Rechner des Pools (damit wird die Ausrede *GHC (usw.) geht auf meinem*

Rechner nicht hinfällig)

- `ssh`-Login oder remote-Desktop-Zugriff *von* einem Rechner des Pools auf Ihren Rechner (damit das projiziert werden kann, *ohne* den Beamer umzustöpseln)

(falls das alles zu umständlich ist, dann eben doch einen Pool-Rechner benutzen)

5. welcher Typ ist zu erwarten für die Funktion,

- (wurde bereits in der Übung behandlelt) die das Spiegelbild einer Zeichenkette berechnet?
- die die Liste aller (durch Leerzeichen getrennten) Wörter einer Zeichenkette berechnet?

```
f "foo bar" = [ "foo", "bar" ]
```

Suchen Sie nach Funktionen dieses Typs mit `https://www.haskell.org/hoogle/`, erklären Sie einige der Resultate, welches davon ist das passende, rufen Sie diese Funktion auf (in ghci).

Daten

Wiederholung: Terme

- (Prädikatenlogik) *Signatur* Σ ist Menge von Funktionssymbolen mit Stelligkeiten

  ein Term t in Signatur Σ ist
  - Funktionssymbol $f \in \Sigma$ der Stelligkeit k
    mit Argumenten $(t_1, \ldots, t_k)$, die selbst Terme sind.

  $\mathrm{Term}(\Sigma) =$ Menge der Terme über Signatur Σ
- (Graphentheorie) ein Term ist ein gerichteter, geordneter, markierter Baum
- (Datenstrukturen)
  - Funktionssymbol $=$ Konstruktor, Term $=$ Baum

Beispiele: Signatur, Terme

- Signatur: $\Sigma_1 = \{Z/0, S/1, f/2\}$
- Elemente von $\mathrm{Term}(\Sigma_1)$:

  $Z(), \quad S(S(Z())), \quad f(S(S(Z())), Z())$

- Signatur: $\Sigma_2 = \{E/0, A/1, B/1\}$
- Elemente von $\mathrm{Term}(\Sigma_2)$: ...

Algebraische Datentypen

```
data Foo = Foo { bar :: Int, baz :: String }
    deriving Show
```

Bezeichnungen (benannte Notation)

- `data Foo` ist Typname
- `Foo { .. }` ist Konstruktor
- `bar, baz` sind Komponenten

```
x :: Foo
x = Foo { bar = 3, baz = "hal" }
```

Bezeichnungen (positionelle Notation)

```
data Foo = Foo Int String
y = Foo 3 "bar"
```

Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

```
data T = A { foo :: Int }
       | B { bar :: String, baz :: Bool }
    deriving Show
```

Bespiele (in Prelude vordefiniert)

```
data Bool = False | True
data Ordering = LT | EQ | GT
```

Mehrsortige Signaturen

- (bisher) einsortige Signatur

  Abbildung von Funktionssymbol nach Stelligkeit

- (neu) mehrsortige Signatur
  - Menge von Sortensymbolen $S = \{S_1, \ldots\}$
  - Abb. von F.-Symbol nach Typ
  - *Typ* ist Element aus $S^* \times S$
    Folge der Argument-Sorten, Resultat-Sorte

Bsp.: $S = \{Z, B\}, \Sigma = \{0 \mapsto ([], Z), p \mapsto ([Z, Z], Z),$
$e \mapsto ([Z, Z], B), a \mapsto ([B, B], B)\}$.

- $\mathrm{Term}(\Sigma)$: konkrete Beispiele, allgemeine Definition?

Rekursive Datentypen

```
data Tree = Leaf {}
     | Branch { left :: Tree
              , right :: Tree }
```

Übung: Objekte dieses Typs erzeugen

(benannte und positionelle Notation der Konstruktoren)

Daten mit Baum-Struktur

- mathematisches Modell: Term über Signatur
- programmiersprachliche Bezeichnung: *algebraischer Datentyp* (die Konstruktoren bilden eine Algebra)
- praktische Anwendungen:
  - Formel-Bäume (in Aussagen- und Prädikatenlogik)
  - Suchbäume (in VL Algorithmen und Datenstrukturen, in `java.util.TreeSet<E>`)
  - DOM (Document Object Model) `https://www.w3.org/DOM/DOMTR`
  - JSON (Javascript Object Notation) z.B. für AJAX `http://www.ecma-international.org/publications/standards/Ecma-404.htm`

Übung Terme

- Geben Sie die Signatur des Terms $\sqrt{a \cdot a + b \cdot b}$ an.
- Geben Sie ein Element $t \in \mathrm{Term}(\{f/1, g/3, c/0\})$ an mit $|t| = 5$ und $\mathrm{depth}(t) \leq 2$.

mit `ghci`:

- `data T = F T | G T T T | C deriving Show`

  erzeugen Sie o.g. Term t (durch Konstruktoraufrufe)

Die *Größe* eines Terms t ist definiert durch $|f(t_0, \ldots, t_{k-1})| = 1 + \sum_{i=0}^{k-1} |t_i|$.

- Bestimmen Sie $|\sqrt{a \cdot a + b \cdot b}|$.

Hausaufgaben

SS21: für KW15: Aufgaben 2 und 3

1. autotool-Aufgabe Data-1

   Allgemeine Hinweise zur Bearbeitung von Haskell-Lückentext-Aufgaben:

   - Schreiben Sie den angezeigten Quelltext (vollständig! ohne zusätzliche Leerzeichen am Zeilenanfang!) in eine Datei mit Endung `.hs`, starten Sie ghci mit diesem Dateinamen als Argument
   - ändern Sie den Quelltext: ersetzen Sie `undefined` durch einen geeigneten Ausdruck, hier z.B.
     `solution = S.fromList [ False, G ]`
     im Editor speichern, in ghci neu laden (`:r`)

- reparieren Sie Typfehler, werten Sie geeignete Terme aus, hier z.B. `S.size solution`
- werten Sie `test` aus, wenn `test` den Wert True ergibt, dann tragen Sie die Lösung in autotool ein.

2. Geben Sie einen Typ (eine `data`-Deklaration) mit genau 100 Elementen an. Sie können andere Data-Deklarationen benutzen (wie in autotool-Aufgabe). Minimieren Sie die Gesamtzahl aller deklarierten Konstruktoren.

3. Vervollständigen Sie die Definition der *Tiefe* von Termen:

$$\mathrm{depth}(f()) = 0$$

$$k > 0 \;\Rightarrow\; \mathrm{depth}(f(t_0, \ldots, t_{k-1})) = \ldots$$

- Bestimmen Sie $\mathrm{depth}(\sqrt{a \cdot a + b \cdot b})$
- Beweisen Sie $\forall \Sigma : \forall t \in \mathrm{Term}(\Sigma) : \mathrm{depth}(t) \leq |t| - 1$.
- Für welche Terme gilt hier die Gleichheit?

Programme

Plan

- wir haben: für Baum-artige Daten:
  - mathematisches Modell: Terme über einer Signatur
  - Realisierung als: algebraischer Datentyp (`data`)
- wir wollen: für Programme, die diese Daten verarbeiten:
  - mathematisches Modell: Termersetzung
  - Realisierung als: Pattern matching (`case`)

Bezeichnungen für Teilterme

- *Position*: Folge von natürlichen Zahlen
  (bezeichnet einen Pfad von der Wurzel zu einem Knoten)
  Beispiel: für $t = S(f(S(S(Z())), Z()))$
  ist $[0, 1]$ eine Position in t.
- $\mathrm{Pos}(t) =$ die Menge der Positionen eines Terms t
  Definition: wenn $t = f(t_0, \ldots, t_{k-1})$, d.h., k Kinder
  dann $\mathrm{Pos}(t) = \{[]\} \cup \{[i]{+\!\!+}p \mid 0 \leq i < k \wedge p \in \mathrm{Pos}(t_i)\}$.

dabei bezeichnen:

- $[]$ die leere Folge,
- $[i]$ die Folge der Länge 1 mit Element i,
- $+\!\!+$ den Verkettungsoperator für Folgen

Operationen mit (Teil)Termen

- $t[p] =$ der Teilterm von t an Position p

  Beispiel: $S(f(S(S(Z())), Z()))[0, 1] = \ldots$

  Definition (durch Induktion über die Länge von p): ...

- $t[p := s]$: wie t, aber mit Term s an Position p

  Beispiel: $S(f(S(S(Z())), Z()))[[0, 1] := S(Z)] = \ldots$

  Definition (durch Induktion über die Länge von p): ...

Operationen mit Variablen in Termen

- $\mathrm{Term}(\Sigma, V) =$ Menge der Terme über Signatur Σ mit Variablen aus V

  Beispiel: $\Sigma = \{Z/0, S/1, f/2\}, V = \{y\}$,
  $f(Z(), y) \in \mathrm{Term}(\Sigma, V)$.

- Substitution σ: partielle Abbildung $V \to \mathrm{Term}(\Sigma)$

  Beispiel: $\sigma_1 = \{(y, S(Z()))\}$

- eine Substitution auf einen Term anwenden: $t\sigma$:

  Intuition: wie t, aber statt v immer $\sigma(v)$

  Beispiel: $f(Z(), y)\sigma_1 = f(Z(), S(Z()))$

  Definition durch Induktion über t

Termersetzungssysteme

- Daten $=$ Terme (ohne Variablen)
- Programm $R =$ Menge von Regeln

  Bsp: $R = \{(f(Z(), y), y),\ (f(S(x), y), S(f(x, y)))\}$
- Regel $=$ Paar (l, r) von Termen mit Variablen
- Relation $\to_R$ ist Menge aller Paare (t, t') mit
  - es existiert $(l, r) \in R$
  - es existiert Position p in t
  - es existiert Substitution $\sigma : (\mathrm{Var}(l) \cup \mathrm{Var}(r)) \to \mathrm{Term}(\Sigma)$
  - so daß $t[p] = l\sigma$ und $t' = t[p := r\sigma]$.

Termersetzungssysteme als Programme

- $\to_R$ beschreibt *einen* Schritt der Rechnung von R,
- transitive und reflexive Hülle $\to_R^*$
  beschreibt *Folge* von Schritten.
- *Resultat* einer Rechnung ist Term in R-Normalform
  ($:=$ ohne $\to_R$-Nachfolger)

dieses Berechnungsmodell ist im allgemeinen

- *nichtdeterministisch* $R_1 = \{C(x, y) \to x, C(x, y) \to y\}$
  (ein Term kann mehrere $\to_R$-Nachfolger haben,
  ein Term kann mehrere Normalformen erreichen)
- *nicht terminierend* $R_2 = \{p(x, y) \to p(y, x)\}$
  (es gibt eine unendliche Folge von $\to_R$-Schritten,
  es kann Terme ohne Normalform geben)

Konstruktor-Systeme

Für TRS R über Signatur Σ: Symbol $s \in \Sigma$ heißt

- *definiert*, wenn $\exists (l, r) \in R : l[] = s(\dots)$
  (das Symbol in der Wurzel ist s)
- sonst *Konstruktor*.

Das TRS R heißt *Konstruktor-TRS*, falls:

- definierte Symbole kommen links *nur* in den Wurzeln vor

Übung: diese Eigenschaft formal spezifizieren

Beispiele: $R_1 = \{a(b(x)) \to b(a(x))\}$ über $\Sigma_1 = \{a/1, b/1\}$,
$R_2 = \{f(f(x, y), z) \to f(x, f(y, z))$ über $\Sigma_2 = \{f/2\}$:
definierte Symbole? Konstruktoren? Konstruktor-System?

Funktionale Programme sind ähnlich zu Konstruktor-TRS.

Funktionale Programme (Bsp. und Vergleich)

- Termersetzungssystem:
  - Signatur: $\{(S,1),(Z,0),(f,2)\}$, Variablenmenge $\{x',y\}$
  - Ersetzungssystem
    $\{f(Z,y) \to y, f(S(x'),y) \to S(f(x',y))\}$.
  - ist Konstruktor-System, definierte Symbole: $\{f\}$,
    Konstruktoren: $\{S,Z\}$,
  - Startterm $f(S(S(Z)),S(Z))$.
- funktionales Programm:

```
data N = Z | S N -- Signatur für Daten
f :: N -> N -> N -- Signatur für Funktion
f Z y = y ; f (S x') y = S (f x' y) -- Gleichungen
f (S (S Z)) (S Z) -- Benutzung der definierten Fkt.
```

Alternative Notation f. Gleichungssystem

- für die Definition einer Funktion f mit diesem Typ

```
data N = Z | S N ; f :: N -> N -> N
```

- (eben gesehen) *mehrere* Gleichungen

```
f Z y = y
f (S x') y = S (f x' y)
```

- äquivalente Notation: *eine* Gleichung,
  in der rechten Seite: Verzweigung (erkennbar an `case`)
  mit zwei Zweigen (erkennbar an `->`)

```
f x y = case x of
    { Z   -> y  ;  S x' -> S (f x' y) }
```

Pattern Matching

```
data N = Z | S N ; data Bool = False | True
positive :: N -> Bool
positive x = case x of { Z -> False ; S x' -> True }
```

- Syntax: `case <Diskriminante> of`
  `{ <Muster> -> <Ausdruck> ; ... }`
- `<Muster>` enthält Konstruktoren und Variablen, entspricht linker Seite einer Term-Ersetzungs-Regel, `<Ausdruck>` entspricht rechter Seite
- statische Semantik:
  - jedes `<Muster>` hat gleichen Typ wie `<Diskrim.>`,
  - alle `<Ausdruck>` haben übereinstimmenden Typ.
- dynamische Semantik:
  - Def.: t *paßt* zum Muster l: es existiert σ mit $l\sigma = t$
  - für das erste passende Muster wird $r\sigma$ ausgewertet

Eigenschaften von Case-Ausdrücken

ein `case`-Ausdruck heißt

- *disjunkt*, wenn die Muster nicht überlappen

  (es gibt keinen Term, der zu mehr als 1 Muster paßt)

- *vollständig*, wenn die Muster den gesamten Datentyp abdecken

  (es gibt keinen Term, der zu keinem Muster paßt)

Bespiele (für `data T = A | B T | F T T`)

```
-- nicht disjunkt:
case t of { F (B x) y -> .. ; F x (B y) -> .
-- nicht vollständig:
case t of { F x y -> .. ; A -> .. }
```

`data` und `case`

typisches Vorgehen beim Verarbeiten algebraischer Daten vom Typ `T`:

- Für jeden Konstruktor des Datentyps

```
data T = C1 ...
       | C2 ...
```

- schreibe einen Zweig in der Fallunterscheidung

```
f x = case x of
    { C1 ... -> ...
    ; C2 ... -> ... }
```

- Argumente der Konstruktoren sind Variablen $\Rightarrow$ Case-Ausdruck ist disjunkt und vollständig.

Pattern Matching in versch. Sprachen

- Scala: case classes `http://docs.scala-lang.org/tutorials/tour/case-classes.html`
- C#: `https://github.com/dotnet/csharplang/blob/master/proposals/csharp-8.0/patterns.md`
- Javascript?

Nicht verwechseln mit *regular expression matching* zur String-Verarbeitung. Es geht um algebraische (d.h. baum-artige) Daten!

Rechnen mit Wahrheitswerten

- der Datentyp

```
import qualified Prelude
data Bool = False | True
  deriving Prelude.Show
```

- die Negation

```
not :: Bool -> Bool
not x = case x of { False -> _ ; True -> _
```

- die Konjunktion (als Operator geschrieben)

```
(&&) :: Bool -> Bool -> Bool
x && y = case x of { False -> _ ; True -> _
```

Syntax für Unterprogramm-Aufrufe

- die Syntax eines Namens bestimmt, ob er als Funktion oder Operator verwendet wird:
  - Name aus Buchstaben (Bsp.: `not`, `plus`)
    steht als Funktion vor den Argumenten
  - Name aus Sonderzeichen (Bsp.: `&&`)
    steht als Operator zw. erstem und zweitem Argument

- zwischen Funktion und Operator umschalten:
  - in runden Klammern: Operator als Funktion
    `(&&)::Bool->Bool->Bool`, `(&&) False True`
  - in Backticks: Funktion als Operator
    `3 ` `plus` ` 4`

Syntax für Fallunterscheidungen

- `not x = case x of { False -> _; True -> _ }`

  Alternative Notation (links), Übersetzung (rechts)

```
not x = case x of
  False -> _
  True -> _
```

```
not x = case x of
  {False -> _
  ;True -> _
  }
```

- Abseitsregel (offside rule): wenn das nächste (nicht leere) Zeichen nach `of` kein `{` ist, werden eingefügt:
  - `{` nach `of`
  - `;` nach Zeilenschaltung bei gleicher Einrückung
  - `}` nach Zeilenschaltung bei höherer Einrückung

Übung Term-Ersetzung

Für die Signatur $\Sigma = \{f/1, g/3, c/0\}$:

- geben Sie ein $t \in \mathrm{Term}(\Sigma))$ an mit $t[1] = c$.
- Beweisen Sie $\forall \Sigma : \forall t \in \mathrm{Term}(\Sigma) : |t| = |\mathrm{Pos}(t)|$.

Für die Signatur $\Sigma = \{Z/0, S/1, f/2\}$:

- für welche Substitution σ gilt $f(x, Z)\sigma = f(S(Z), Z)$?
- für dieses σ: bestimmen Sie $f(x, S(x))\sigma$.

Notation für Termersetzungsregeln: anstatt (l, r) schreibe $l \to r$.

Abkürzung für Anwendung von 0-stelligen Symbolen: anstatt $Z()$ schreibe Z. (Vorsicht: dann kann man Variablen

nicht mehr von 0-stelligen Symbolen unterscheiden. Man muß dann immer die Signatur explizit angeben oder auf andere Weise vereinbaren, wie man Variablen erkennt, z.B. „Buchstaben am Ende das Alphabetes ($\ldots, x, y, \ldots$) sind Variablen“, das ist aber riskant)

- Für $R = \{f(S(x), y) \to f(x, S(y)), f(Z, y) \to y\}$

  bestimme alle R-Normalformen von $f(S(Z), S(Z))$.

- für $R_d = R \cup \{d(x) \to f(x, x)\}$

  bestimme alle R_d-Normalformen von $d(d(S(Z)))$.

- Bestimme die Signatur Σ_d von R_d.

  Bestimme die Menge der Terme aus $\mathrm{Term}(\Sigma_d)$, die R_d-Normalformen sind.

Abkürzung für mehrfache Anwendung eines einstelligen Symbols: $A(A(A(A(x)))) = A^4(x)$

- für $\{A(B(x)) \to B(A(x))\}$

  über Signatur $\{A/1, B/1, E/0\}$:

  bestimme Normalform von $A^k(B^k(E))$

  für $k = 1, 2, 3$, allgemein.

- für $\{A(B(x)) \to B(B(A(x)))\}$

  über Signatur $\{A/1, B/1, E/0\}$:

  bestimme Normalform von $A^k(B(E))$

  für $k = 1, 2, 3$, allgemein.

Übung Pattern Matching, Programme

- Für die Deklarationen

```
-- data Bool = False | True    (aus Prelude)
data T = F T | G T T T | C
```

  entscheide/bestimme für jeden der folgenden Ausdrücke:

  – syntaktisch korrekt?
  – statisch korrekt?
  – Resultat (dynamische Semantik)
  – disjunkt? vollständig?

```
1. case False of { True -> C }
2. case False of { C -> True }
```

```
3. case False of { False -> F F }
4. case G (F C) C (F C) of { G x y z -> F z
5. case F C of { F (F x) -> False }
6. case F C of { F x -> False ; True -> Fal
7. case True of { False -> C ; True -> F C
8. case True of { False -> C ; False -> F C
9. case C of { G x y z -> False; F x -> Fal
```

- Listen von Wahrheitswerten:

```
data List = Nil | Cons Bool List deriving P

and :: List -> Bool
and l = case l of ...
```

entsprechend `or :: List -> Bool`

- (Wdhlg.) welche Signatur beschreibt binäre Bäume

  (jeder Knoten hat 2 oder 0 Kinder, die Bäume sind; es gibt keine Schlüssel)
- geben Sie die dazu äquivalente `data`-Deklaration an:
  `data T = ...`
- implementieren Sie dafür die Funktionen

```
size  :: T -> Prelude.Int
depth :: T -> Prelude.Int
```

  benutze `Prelude.+` (das ist Operator),
  `Prelude.min, Prelude.max`
- für Peano-Zahlen `data N = Z | S N`

  implementieren Sie *plus, mal, min, max*

Hausaufgaben

im SS 21: Aufgaben 1 bis 4

1. für die Signatur $\{A/2, D/0\}$:
   - definiere Terme $t_0 = D, t_{i+1} = A(t_i, D)$.
     Zeichne t_3. Bestimme $|t_i|, \mathrm{depth}(t_i)$.
   - für $S = \{A(A(D, x), y) \to A(x, A(x, y))\}$
     bestimme S-Normalform(en), soweit existieren, der Terme $t_0, t_1, \ldots, t_4$.
     Geben Sie für t_2 die ersten Ersetzungs-Schritte explizit an.
   - Normalform von t_i allgemein.

2. Für die Deklarationen

```
-- data Bool = False | True    (aus Prelude)
```

```
data S = A Bool | B | C S S
```

entscheide/bestimme für jeden der folgenden Ausdrücke:

- syntaktisch korrekt?
- Resultat-Typ (statische Semantik)
- Resultat-Wert (dynamische Semantik)
- Menge der Muster ist: disjunkt? vollständig?

```
1. case False of { True -> B }
2. case False of { B -> True }
3. case C B B of { A x -> x }
4. case A True of { A x -> False }

5. case A True of { A x -> False ; True ->
6. case True of { False -> A ; True -> A Fa
```

```
7. case True of { False -> B ; False -> A F
8. case B of { C x y -> False; A x -> x; B
```

weitere Beispiele selbst herstellen und dann in der Übung die anderen Teilnehmer fragen.

3. für selbst definierte Wahrheitswerte

   deklarieren, implementieren und testen Sie die einstellige Negation, die zweistellige Antivalenz.

```
import qualified Prelude
data Bool = False | True deriving Prelude.S
not :: ...
xor :: ...
```

   Definieren Sie

- `not` mit *einer* Gleichung (evtl. mit `case`)
- `xor` *ohne* `case` (evtl. mehrere Gleichungen)

4. für binäre Bäume ohne Schlüssel

```
data Tree = Leaf | Branch Tree Tree
```

deklarieren, implementieren und testen Sie ein einstelliges Prädikat über solchen Bäumen, das genau dann wahr ist, wenn das Argument eine gerade Anzahl von Blättern enthält.

Diese Anzahl *nicht* ausrechnen, sondern direkt den Wahrheitswert!

Bemerkung zur Bezeichnung: man könnte auch

```
data Node = Leaf | Branch Tree Tree
```

schreiben, aber bitte niemals:

```
data Tree = Leaf | Node Tree Tree
```

denn *auch Blätter sind Knoten* (nodes)! Ein `Branch` ist ein *Verzweigungsknoten* oder *innerer Knoten*, ein Blatt ist ein *äußerer Knoten* des Baumes.

Beweise

Motivation

- Programmierer (Software-Ingenieur) muß beweisen, daß Programm (Softwareprodukt) die Spezifikation erfüllt

  vgl. (Maschinen)Bau-Ingenieur: Brücke, Flugzeug

- vgl. Dijkstra (EWD 1305) zum Verhältnis von *Programmieren* (Wagen) und *Beweisen* (Pferd)

- für funktionale Programmierung: direkte Entsprechung zw. Konstruktion/Ausführung von Programm und Beweis:
  - Auswertungs-Schritte: Gleichungskette
  - Verzweigung (case): Fallunterscheidung
  - strukturelle Rekursion: vollständige Induktion

Formale Beweise

- verschiedene Formen des Beweises:
  - Beweis durch *hand waving*, durch Autorität
  - formaler Beweis (handschriftlich, LaTeX)
  - formaler Beweis *mit maschineller Prüfung*
- statische Programm-Eigenschaften:
  - als Typ-Aussagen formuliert (Bsp: `f x :: Bool`)
  - und durch Compiler bewiesen
- für Eigenschaften, die sich (in Haskell) nicht als Typ formulieren lassen (Bsp: `f x == True`),
  Benutzung anderer Notation und Werkzeuge.
  wir verwenden CYP (Noschinski et al.)
- ausdrucksstärkere Programmiersprachen ist z.B. Agda

CYP: Gleichungsketten

- ```
  data Bool = False | True

  not :: Bool -> Bool
  not False = True
  not True  = False

  Lemma nnf: not (not False) .=. False
  Proof by rewriting   not (not False)
     (by def not) .=. not True
     (by def not) .=. False
  QED
  ```

- vgl. Definition/Autotool-Aufgabe Term-Ersetzung
  ```

CYP: Fallunterscheidung

```
Lemma nnx:  not (not x) .=. x
Proof by case analysis on x :: Bool
  Case False
    Assume XF :  x .=. False
    Then Proof by rewriting  not (not x)
      (by XF) .=. not (not False)
      ...
    QED
  ...
QED
```

- vollständige Menge der Muster in der Fallunterscheidung
- Notation . . . für Lücken auch in Autotool-Aufgaben

Peano-Zahlen

- Axiome von G. Peano: $0 \in \mathbb{N}, \forall x : x \in \mathbb{N} \Rightarrow (1 + x) \in \mathbb{N}$
- realisiert als algebraischer Datentyp

```
data N = Z     -- Null, Zero
       | S N   -- Nachfolger, Successor
```

  Zahl $n \in \mathbb{N}$ dargestellt als $S^n(Z)$, Bsp: $2 = S(S(Z))$

- Ableitung der Implementierung der Addition

```
plus :: N -> N -> N
plus x y = case x of Z -> y ; S x' -> ...
```

  benutze Assoziatitität $x + y = (1 + x') + y = \ldots$

Spezifikation und Test

Bsp: Addition von Peano-Zahlen

- Spezifikation:
  - Typ: `plus :: N -> N -> N`
  - Axiome (Bsp): `plus` ist kommutativ
- Test der Korrektheit durch
  - Aufzählen einzelner Testfälle

    ```
    plus(S (S Z))(S Z) == plus(S Z)(S (S Z))
    ```
  - Notieren von Eigenschaften (*properties*)

    ```
    plus_comm :: N -> N -> Bool
    plus_comm x y = plus x y == plus y x
    ```

    und automatische typgesteuerte Testdatenerzeugung

    ```
    Test.LeanCheck.checkFor 10000 plus_comm
    ```

Spezifikation und Verifikation

Beweis für: Addition von Peano-Zahlen ist assoziativ

- zu zeigen ist

```
plus a (plus b c) == plus (plus a b) c
```

- Beweismethode: Induktion (nach `a`)
  und Umformen mit Gleichungen (äquiv. zu Implement.)

```
plus  Z     y = y
plus (S x') y = S (plus x' y)
```

- Anfang: `plus Z (plus b c) == ..`
- Schritt: `plus (S a') (plus b c) ==`

```
== S (plus a' (plus b c)) == ..
```

Bezeichnungen in Beweisen durch Induktion

- Es ist $\forall t \in \mathrm{Term}(\Sigma) : P(t)$ zu zeigen

  P gilt für alle Terme der Signatur Σ.

- Beweis durch strukturelle Induktion
  - (IA) Induktions-Anfang:
    wir zeigen $P(t)$ für alle Terme $t = f()$, d.h., Blätter
  - (IS) Induktions-Schritt
    wir zeigen $P(t)$ für alle Terme $t = f(t_1, \ldots, t_n)$ mit $n \geq 1$,
    d.h., innere Knoten
    - (IV) Induktions-Voraussetzung:
      $P(t_1) \wedge \cdots \wedge P(t_n)$, d.h., P gilt für alle Kinder von t
    - (IB) Induktions-Behauptung: $P(t)$
      gezeigt wird die Implikation IV $\Rightarrow$ IB.

CYP: Induktion

```
Lemma plus_assoc :
  plus a (plus b c) .=. plus (plus a b) c
Proof by induction on a :: N
Case Z
  Show : plus Z (plus b c) .=. plus (plus Z b) c
  Proof ... QED
Case S a'
  Fix a' :: N
  Assume IV :
    plus a' (plus b c).=. plus (plus a' b) c
  Then Show :
    plus(S a')(plus b c) .=. plus (plus (S a') b) c
  Proof ... QED
QED
```

- ausführliche Notation erforderlich — das ist Absicht

Beispiel Konstruktion/Induktion: even

- ```
  even :: N -> Bool
  even Z = True ; even (S x) = not (even x)
  ```

- suchen `f :: Bool -> Bool -> Bool`, so daß

  ```
  Lemma : even (plus a b) .=. f (even a) (even b)
  ```

- Vorgehensweise:

  wir sehen beim Beweisen, welche Eigenschaften `f` haben muß,

  und definieren `f` dann genau so.

  (Pferd — Wagen!)
  ```

Beispiel Konstruktion/Induktion: even (IA)

- ```
  Lemma : even (plus a b) .=. f (even a) (even b)
  Proof by induction on a :: N
  Case Z
    Show : even (plus Z b) .=. f (even Z) (even b)
    Proof by rewriting    even (plus Z b)
      (by def plus) .=. even b
      (by def f)    .=.  f True (even b)
      (by def even) .=. f (even Z) (even b)
    QED
  ...
  QED
  ```

- damit `(by def f)` stimmt, schreiben wir

  ```
  f :: Bool -> Bool -> Bool
  f _ _ = _
  ```
  ```

Beispiel Konstruktion/Induktion: even (IS)

- entscheidenden Stelle im Induktionsschritt ist

```
Proof by rewriting  even (plus (S a') b)
   (by def plus) .=. even (S (plus a' b))
   (by def even) .=. not (even (plus a' b))
   (by IH) .=. not (f (even a') (even b))
   (by fnot) .=. f (not (even a')) (even b)
   (by def even) .=. f (even (S a')) (even b)
QED
```

- wir postulieren den Hilfssatz

```
Lemma fnot : not (f x y) .=. f (not x) y
Proof by case analysis on x :: Bool ... QED
```

Induktion über Bäume (IA)

- gegeben sind:

```
data Tree = Leaf | Branch Tree Tree
leaves :: Tree -> N
leaves Leaf = S Z
leaves (Branch l r) = plus (leaves l) (leaves r)
```

- gesucht ist: `g :: Tree -> Bool` mit

```
Lemma : even (leaves t) .=. g t
Proof by induction on t :: Tree
Case Leaf
  Show : even (leaves Leaf) .=. g Leaf
  Proof by rewriting ...  QED
...
QED
```

Induktion über Bäume (IS)

- ```
  Case Branch l r
    Fix l :: Tree, r :: Tree
    Assume
      IH1: g l .=. even (leaves l)
      IH2: g r .=. even (leaves r)
    Then Show :
      g (Branch l r) .=. even (leaves (Branch l r))
    Proof by rewriting
    ...
    QED
  ```

- zwei Teile der Induktionsvoraussetzung (IH1, IH2)
  ```

Multiplikation von Peano-Zahlen

- ```
  times :: N -> N -> N
  times x y = case x of
    Z -> _  ; S x' -> _
  ```

  vervollständigen durch Umformen der Spezifikation,

  Bsp. $(1 + x') \cdot y = y + x' \cdot y$

- Eigenschaften formulieren, testen (leancheck),

  beweisen (auf Papier, mit CYP)

  – Multiplikation mit 0, mit 1,
  – Distributivität (mit Plus), Assoziativität, Kommutativität

- ähnliche für Potenzierung
  ```

Subtraktion

- `minus :: N -> N -> N`

  modifizierte Subtraktion, Bsp: $5 \ominus 3 = 2, 3 \ominus 5 = 0$

- Spezifikation: eigentlich $a \ominus b = \max(a - b, 0)$,

  vollst. Spez. ohne Verwendung von Hilfsfunktionen:

  - $\forall a, b \in \mathbb{N} : (a + b) \ominus b = a$
  - $\forall a, b \in \mathbb{N} : a \ominus (a + b) = 0$

- Implementierung (Muster disjunkt? vollständig?)

```
minus Z b = _ ; minus a Z = _
minus (S a') (S b') = _
```

Hausaufgaben

im SS21: Aufgaben 2 bis 5

1. (autotool) Für die Funktion

```
f :: N -> N
f Z = Z ; f (S x) = S (S (f x))
```

beweisen Sie (erst auf Papier, dann mit CYP)

```
f (plus x y) = plus (f x) (f y)
```

durch Induktion nach `x`.

Papier: Verwenden Sie die angegebenen Bezeichnungnen für die Beweis-Schritte, geben Sie IA, IV, IB explizit an.

2. Implementieren Sie Peano-Multiplikation und -Potenz.

   Formulieren, testen (leancheck) und beweisen (Papier, CYP) Sie einige Eigenschaften.

   CYP: formulieren Sie ggf. Hilfssätze als Axiome, d.h., ohne Beweis—aber mit Tests.

3. Für zweistelliges min und max auf $\mathbb{N}$:

   (a) Geben Sie eine äquivalente vollständige Spezifikation an, die keine Fallunterscheidung benutzt.
   (b) Implementieren Sie min und max nur durch Addition und Subtraktion ($\ominus$).
   (c) testen (optional: und beweisen) Sie, daß Ihre Implementierung die Spezifikation erfüllt

4. Implementieren Sie nur mit min und max:

   - den Median von drei Argumenten
   - den Median von fünf Argumenten

   Geben Sie Tests an (optional: Beweis)

5. für das TRS $R = \{A(A(D, x), y) \to A(x, A(x, y))\}$ über der Signatur $\Sigma = \{D/0, A/2\}$, vgl. frühere Aufgabe,

   (a) die Menge (Folge) aller R-Normalformen ist $N_0 = D, N_1 = A(D, N_0), \ldots, N_{k+1} = A(D, N_k), \ldots$.
   warum gibt es keine anderen R-Normalformen?

   (b) Die R-Normalform von $A(N_l, N_r)$ ist N_k mit $k = 2^l + r$.

   i. Geben Sie Beispiele an (auf Papier oder maschinell)

   ii. beweisen Sie durch vollständige Induktion nach l.
   (Auf Papier, aber mit korrekten Bezeichnungen.)

Polymorphie

Definition, Motivation

- Beispiel: binäre Bäume mit Schlüssel vom Typ `e`

```
data Tree e = Leaf
            | Branch (Tree e) e (Tree e)
Branch Leaf  True Leaf :: Tree Bool
Branch Leaf (S Z) Leaf :: Tree N
```

- Definition:
  ein polymorpher Datentyp ist ein *Typkonstruktor*
  (= eine Funktion, die Typen auf einen Typ abbildet)
- unterscheide: `Tree` ist der Typkonstruktor,
  `Branch` ist ein Datenkonstruktor

Beispiele f. Typkonstruktoren (I)

- Kreuzprodukt: `data Pair a b = Pair a b`
- disjunkte Vereinigung:

```
data Either a b = Left a | Right b

data Maybe a = Nothing | Just a
```

- Haskell-Notation für Produkte: `(Z,True)::(N,Bool)`
  - das Komma `(,)` = Name des Typ-Konstruktors = Name des Daten-Konstruktors
  - ist gefährlich (wg. Verwechslung), aber nützlich und empfohlen, *falls* der Typ genau einen Konstruktor hat
  - Komma-Notation für Produkte mit $0, 2, 3, \ldots$ Komponenten
  - 0 Komponenten = die Einermenge: `data () = ()`

Beispiele f. Typkonstruktoren (II)

- binäre Bäume (Schlüssel in der Verzweigungsknoten)

```
data Bin a = Leaf
           | Branch (Bin a) a (Bin a)
```

- einfach (vorwärts) verkettete Listen

```
data List a = Nil
            | Cons a (List a)
```

- Bäume mit Knoten beliebiger Stelligkeit, Schlüssel in jedem Knoten

```
data Tree a = Node a (List (Tree a))
```

Polymorphe Funktionen

- Beispiele:
  - Spiegelbild einer Liste:

    ```
    reverse :: forall e . List e -> List e
    ```
  - Verkettung von Listen mit gleichem Elementtyp:

    ```
    append :: forall e .      List e -> List e
                           -> List e
    ```

    Knotenreihenfolge eines Binärbaumes:

    ```
    preorder :: forall e . Bin e -> List e
    ```

- Def: der Typ einer polymorphen Funktion beginnt mit All-Quantoren für Typvariablen.

- Bsp: Datenkonstruktoren polymorpher Typen.

Bezeichnungen f. Polymorphie

```
data List e = Nil | Cons e (List e)
```

- `List` ist ein *Typkonstruktor*
- `List e` ist ein *polymorpher* Typ

  (ein Typ-Ausdruck mit *Typ-Variablen*)
- `List Bool` ist ein *monomorpher* Typ

  (entsteht durch *Instantiierung*: Substitution der Typ-Variablen durch Typen)
- polymorphe Funktion:
  `reverse:: forall e . List e -> List e`

  monomorphe Funktion: `xor:: List Bool -> Bool`

  polymorphe Konstante: `Nil::forall e. List e`

Operationen auf Listen (I)

```
data List a = Nil | Cons a (List a)
```

- ```
  append xs ys = case xs of
        Nil          ->
        Cons x xs' ->
  ```
- Ü: formuliere, teste und beweise: `append` ist assoziativ.
- ```
  reverse xs = case xs of
        Nil          ->
        Cons x xs' ->
  ```
- Ü: beweise:

```
forall xs ys : reverse (append xs ys)
  == append (reverse ys) (reverse xs)
```

Von der Spezifikation zur Implementierung (II)

- Bsp: homogene Listen

  ```
  data List a = Nil | Cons a (List a)
  ```

  Aufgabe: implementiere `maximum :: List N -> N`

  Spezifikation:

  ```
  maximum (Cons x1 Nil) = x1
  maximum (append xs ys) = max (maximum xs) (
  ```

- – substitutiere `xs = Nil`, erhalte

  ```
    maximum (append Nil ys) = maximum ys
  = max (maximum Nil) (maximum ys)
  ```

  d.h. `maximum Nil` sollte das neutrale Element für `max` (auf natürlichen Zahlen) sein, also `0` (geschrieben `Z`).

– substitutiere `xs = Cons x1 Nil`, erhalte

```
    maximum (append (Cons x1 Nil) ys)
    = maximum (Cons x1 ys)
  = max (maximum (Cons x1 Nil)) (maximum ys)
    = max x1 (maximum ys)
```

Damit kann der aus dem Typ abgeleitete Quelltext

```
maximum :: List N -> N
maximum xs = case xs of
  Nil          ->
  Cons x xs' ->
```

ergänzt werden.

Vorsicht: für `min, minimum` funktioniert das nicht so, denn `min` hat für `N` kein neutrales Element.

Operationen auf Listen (II)

- Die vorige Implementierung von `reverse` ist (für einfach verkettete Listen) nicht effizient (sondern quadratisch, vgl. `https://accidentallyquadratic.tumblr.com/`)
- Besser ist Verwendung einer Hilfsfunktion

```
reverse xs = rev_app xs Nil
```

mit Spezifikation

```
rev_app xs ys = append (reverse xs) ys
```

- noch besser ist es, *keine* Listen zu verwenden
  `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`

Operationen auf Bäumen

```
data List e = Nil | Cons e (List e)
data Bin e = Leaf | Branch (Bin e) e (Bin e)
```

Knotenreihenfolgen

- `preorder :: forall e . Bin e -> List e`
  `preorder t = case t of ...`
- entsprechend `inorder`, `postorder`
- und Rekonstruktionsaufgaben

Adressierug von Knoten (`False` = links, `True` = rechts)

- `get :: Bin e -> List Bool -> Maybe e`
- `positions :: Bin e -> List (List Bool)`

Statische Typisierung und Polymorphie

- Def: dynamische Typisierung:
  - die Daten (zur Laufzeit des Programms, im Hauptspeicher) haben einen Typ
- Def: statische Typisierung:
  - Bezeichner, Ausdrücke (im Quelltext) haben einen Typ, dieser wird zur Übersetzungszeit (d.h., ohne Programmausführung) bestimmt
  - für *jede* Ausführung des Programms gilt: der statische Typ eines Ausdrucks ist gleich dem dynamischen Typ seines Wertes

Bsp. für Programm ohne statischen Typ

- Javascript

```
function f (x) {
  if (x > 0) {
    return function () { return 42; }
  } else { return "foobar"; } } }
```

  Dann: Auswertung von `f(1)()` ergibt 42, Auswertung von `f(0)()` ergibt Laufzeit-Typfehler.

- entsprechendes Haskell-Programm ist statisch fehlerhaft

```
f x = case x > 0 of
  True  -> \ () -> 42
  False -> "foobar"
```

Nutzen der stat. Typisierung und Polymorphie

- Nutzen der statischen Typisierung:
  - beim Programmieren: Entwurfsfehler werden zu Typfehlern, diese werden zur Entwurfszeit automatisch erkannt $\Rightarrow$ früher erkannte Fehler lassen sich leichter beheben
  - beim Ausführen: keine Lauzeit-Typfehler $\Rightarrow$ keine Typprüfung zur Laufzeit nötig, effiziente Ausführung
- Nutzen der Polymorphie:
  - Flexibilität, nachnutzbarer Code, z.B. Anwender einer Collection-Bibliothek legt Element-Typ fest (Entwickler der Bibliothek kennt den Element-Typ nicht)
  - gleichzeitig bleibt statische Typsicherheit erhalten

Konstruktion von Objekten eines Typs

Aufgabe (Bsp):

```
x :: Either (Maybe ()) (Pair Bool ())
```

Lösung (Bsp):

- der Typ `Either a b` hat Konstruktoren `Left a | Right b`. Wähle `Right b`.

  Die Substitution für die Typvariablen ist

  `a = Maybe (), b = Pair Bool ().`

  `x = Right y` mit `y :: Pair Bool ()`

- der Typ `Pair a b` hat Konstruktor `Pair a b`.

  die Substitution für diese Typvariablen ist

  `a = Bool, b = ().`

`y = Pair p q` mit `p :: Bool, q :: ()`

- der Typ `Bool` hat Konstruktoren `False | True`, wähle `p = False`. der Typ `()` hat Konstruktor `()`, also `q=()`

Insgesamt `x = Right y = Right (Pair False ())`

Vorgehen (allgemein)

- bestimme den Typkonstruktor
- bestimme die Substitution für die Typvariablen
- wähle einen Datenkonstruktor
- bestimme Anzahl und Typ seiner Argumente
- wähle Werte für diese Argumente nach diesem Vorgehen.

Bestimmung des Typs eines Bezeichners

Aufgabe (Bsp.) bestimme Typ von `x` (erstes Arg. von `get`):

```
at :: List N -> Tree a -> Maybe a
at p t = case t of
  Node f ts -> case p of
    Nil -> Just f
    Cons x p' -> case get x ts of
      Nothing -> Nothing
      Just t' -> at p' t'
```

Lösung:

- bestimme das Muster, durch welches `x` deklariert wird.
  Lösung: `Cons x p' ->`
- bestimme den Typ diese Musters

Lösung: ist gleich dem Typ der zugehörigen *Diskriminante* `p`

- bestimme das Muster, durch das `p` deklariert wird
  Lösung: `at p t =`
- bestimme den Typ von `p`
  Lösung: durch Vergleich mit Typdeklaration von `at` (`p` ist das erste Argument) `p :: Position`, also `Cons x p' :: Position = List N`, also `x :: N`.

Vorgehen zur Typbestimmung eines Namens:

- finde die Deklaration (Muster einer Fallunterscheidung oder einer Funktionsdefinition)
- bestimme den Typ des Musters (Fallunterscheidung: Typ der Diskriminante, Funktion: deklarierter Typ)

Übung Polymorphie

- Geben Sie alle Elemente dieser Datentypen an:
  - `Maybe ()`
  - `Maybe (Bool, Maybe ())`
  - `Either ((),Bool) (Maybe (Maybe Bool))`
- Operationen auf Listen:
  - append, reverse, rev_app
- Operationen auf Bäumen:
  - preorder, inorder
  - get, (positions)

Hausaufgaben

1. für die folgenden Datentypen: geben Sie einige Elemente an (ghci), geben Sie die Anzahl aller Elemente an (siehe auch autotool-Aufgabe)

   (a) `Maybe (Maybe Bool)`

   (b) `Either (Bool, ()) (Maybe ())`

   (c) `Foo (Maybe (Foo Bool))` mit

   ```
   data Foo a = C a | D
   ```

   (d) geben Sie einen Typ mit 100 (Zusatz: 1000) Elementen an und (insgesamt) wenig Daten-Konstruktoren. (Vgl. frühere Aufgabe, die Frage ist jetzt, ob die Polymorphie hilft, eine kleinere Lösung zu erhalten)

2. Implementieren Sie die Post-Order Durchquerung von

Binärbäumen.

(Zusatz: Level-Order. Das ist schwieriger.)

Verwenden Sie nur die in der VL definierten Typen (`data List a = ...`, nicht `Prelude.[]`)

und Programmbausteine (`case _ of`)

3. Beweisen Sie (auf Papier, Zusatz: mit Cyp)

```
forall xs . reverse (reverse xs) == xs
```

Verwenden Sie ggf.

```
rev (app xs ys) = app (rev ys) (rev xs)
```

oder andere Hilfssätze ohne Beweis (aber mit Beispielen und Tests)

4. (von voriger Woche, Typ ist nicht polymorph)

   Definitionen ergänzen; Eigenschaften testen (Einzelfälle, Leancheck), beweisen (Papier oder Cyp)

```
data Tree = Leaf | Branch Tree Tree
size :: Tree -> N
leaves :: Tree -> N
branches :: Tree -> N
odd :: N -> Bool
Lemma :
  size t .=. plus (leaves t) (branches t)
Lemma : odd (size t)
```

   Folien *Induktion über Bäume* benutzen.

Lambda-Kalkül: Syntax und dynamische Semantik (Auswertung)

Funktionen als Daten

- bisher: Fkt. definiert d. Gleichg. `dbl x = plus x x`
- jetzt: durch Lambda-Term `dbl = \ x -> plus x x`
  λ-Terme: mit lokalen Namen (hier: x)
- Funktionsanwendung: $(\lambda x.B)A \rightarrow B[x := A]$
  freie Vorkommen von x in B werden durch A ersetzt
- Funktionen sind Daten (Bsp: `Cons dbl Nil`)
- λ-Kalkül: Alonzo Church 1936, Henk Barendregt 198*
  Henk Barendregt: *The Impact of the Lambda Calculus in Logic and Computer Science*, Bull. Symbolic Logic, 1997.

https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.9348

Der Lambda-Kalkül

- ist ein Berechnungsmodell,
  vgl. Termersetzungssysteme, Turingmaschine,
  Random-Access-Maschine ($=$ Goto-Programme)
- *Syntax*: die Menge der Lambda-Terme Λ:
  - jede Variable ist ein Term: $v \in V \Rightarrow v \in \Lambda$
  - Funktionsanwendung (Applikation):
    $F \in \Lambda, A \in \Lambda \Rightarrow @(F, A) \in \Lambda$
    abstrakte Syntax: 2-stell. Operator $@$,
    konkrete Syntax: Leerzeichen (!)
  - Funktionsdefinition (Abstraktion):
    $v \in V, B \in \Lambda \Rightarrow (\lambda v.B) \in \Lambda$
- *Semantik*: eine Relation $\rightarrow_\beta$ auf Λ
  (vgl. $\rightarrow_R$ für Termersetzungssystem R)

Freie und gebundene Variablen(vorkommen)

- Das Vorkommen von $v \in V$ an Position p in Term t heißt *frei*, wenn „darüber kein $\lambda v. \ldots$ steht“
- Def. $\mathrm{fvar}(t) =$ Menge der in t frei vorkommenden Variablen (definiere durch strukturelle Induktion)
- Eine Variable x heißt in A *gebunden*, falls A einen Teilausdruck $\lambda x.B$ enthält.
- Def. $\mathrm{bvar}(t) =$ Menge der in t gebundenen Variablen
- Bsp: $\mathrm{fvar}(x(\lambda x.\lambda y.x)) = \{x\}$, $\mathrm{bvar}(x(\lambda x.\lambda y.x)) = \{x, y\}$,

Semantik des Lambda-Kalküls: Reduktion $\rightarrow_\beta$

Relation $\rightarrow_\beta$ auf Λ (ein Reduktionsschritt)

Es gilt $t \rightarrow_\beta t'$, falls

- $\exists p \in \mathrm{Pos}(t)$, so daß
- $t[p] = (\lambda x.B)A$ mit $\mathrm{bvar}(B) \cap \mathrm{fvar}(A) = \emptyset$
- $t' = t[p := B[x := A]]$

  dabei bezeichnet $B[x := A]$ ein Kopie von B, bei der jedes freie Vorkommen von x durch A ersetzt ist

Ein (Teil-)Ausdruck der Form $(\lambda x.B)A$ heißt *Redex*.
(Dort kann weitergerechnet werden.)

Ein Term ohne Redex heißt *Normalform*.
(Normalformen sind Resultate von Rechnungen.)

Falsches Binden lokaler Variablen

- dieser Ausdruck hat den Wert 15:

```
(\x->(((\f->\x->x + f 8) (\y-> x+y)) 4)) 3
```

- Redex $(\lambda f.B)A$ mit $B = \lambda x.x + f8$ und $A = \lambda y.x + y$:
- dort keine $\rightarrow_\beta$-Reduktion, $\mathrm{bvar}(B) \cap \mathrm{fvar}(A) = \{x\} \neq \emptyset$.
- falls wir die Nebenbedingung ignorieren, erhalten wir

```
(\x->((      \x->x + (\y-> x+y) 8)     4)) 3
```

  mit Wert 16.

- dieses Beispiel zeigt, daß die Nebenbedingung semantische Fehler verhindert

Semantik . . . : gebundene Umbenennung $\rightarrow_\alpha$

- falls wir einen Redex $(\lambda x.B)A$ reduzieren möchten, für den $\mathrm{bvar}(B) \cap \mathrm{fvar}(A) = \emptyset$ nicht gilt,

  dann vorher dort die lokale Variable x umbenennen (hinter dem λ und jedes freie Vorkommen von x in B)
- Relation $\rightarrow_\alpha$ auf Λ, beschreibt *gebundene Umbenennung* einer lokalen Variablen.
- Beispiel $\lambda x.fxz \rightarrow_\alpha \lambda y.fyz$.

  (f und z sind frei, können nicht umbenannt werden)
- Definition $t \rightarrow_\alpha t'$:
  - $\exists p \in \mathrm{Pos}(t)$, so daß $t[p] = (\lambda x.B)$
  - $y \notin \mathrm{bvar}(B) \cup \mathrm{fvar}(B)$
  - $t' = t[p := \lambda y.B[x := y]]$

Umbenennung von lokalen Variablen

- ```
  int x = 3;
  int f(int y) { return x + y; }
  int g(int x) { return (x + f(8)); }  // g(4) => 15
  ```
- Darf `f(8)` ersetzt werden durch $f[y := 8]$? - Nein:

  ```
  int x = 3;
  int g(int x) { return (x + (x+8)); } // g(4) => 16
  ```

  Das freie x in $(x + y)$ wird fälschlich gebunden.
- Lösung: lokal umbenennen

  ```
  int g(int z) { return (z + f(8)); }
  ```

  dann ist Ersetzung erlaubt

  ```
  int x = 3;
  int g(int z) { return (z + (x+8)); } // g(4) => 15
  ```
  ```

Lambda-Terme: verkürzte Notation

- Applikation ist links-assoziativ, Klammern weglassen:

$$(\ldots((FA_1)A_2)\ldots A_n) \sim FA_1A_2\ldots A_n$$

Beispiel: $((xz)(yz)) \sim xz(yz)$

Wirkt auch hinter dem Punkt:
$(\lambda x.xx)$ bedeutet $(\lambda x.(xx))$ — und nicht $((\lambda x.x)x)$

- geschachtelte Abstraktionen unter ein Lambda schreiben:

$$(\lambda x_1.(\lambda x_2.\ldots.(\lambda x_n.B)\ldots)) \sim \lambda x_1x_2\ldots x_n.B$$

Beispiel: $\lambda x.\lambda y.\lambda z.B \sim \lambda xyz.B$

Ein- und mehrstellige Funktionen

- eine einstellige Funktion zweiter Ordnung:

```
f = \ x -> ( \ y -> ( x*x + y*y ) )
```

  Anwendung dieser Funktion:

```
(f 3) 4 = ...
```

  Kurzschreibweisen (Klammern weglassen):

```
f = \ x y -> x * x + y * y ; f 3 4
```

- Übung:

  gegeben `t = \ f x -> f (f x)`

  bestimme `t succ 0, t t succ 0,`
  `t t t succ 0, t t t t succ 0, ...`

Lambda-Ausdrücke in C#

- Beispiel (Fkt. 1. Ordnung)

```
Func<int,int> f = (int x) => x*x;
f (7);
```

- Übung (Fkt. 2. Ordnung) — ergänze alle Typen:

```
???  t = (??? g) => (??? x) => g (g (x));
t (f)(3);
```

- Anwendungen bei Streams, später mehr

```
(new int[]{3,1,4,1,5,9}).Select(x => x * 2);
(new int[]{3,1,4,1,5,9}).Where(x => x > 3);
```

- Übung: Diskutiere statische/dynamische Semantik von

```
(new int[]{3,1,4,1,5,9}).Select(x => x > 3);
(new int[]{3,1,4,1,5,9}).Where(x => x * 2);
```

Lambda-Ausdrücke in Java

- *funktionales* Interface (FI): hat genau eine Methode
- Lambda-Ausdruck („burger arrow“) erzeugt Objekt einer anonymen Klasse, die FI implementiert.

```
interface I { int foo (int x); }
I f = (x)-> x+1;
System.out.println (f.foo(8));
```

- vordefinierte FIs:

```
import java.util.function.*;
Function<Integer,Integer> g = (x)-> x*2;
   System.out.println (g.apply(8));
Predicate<Integer> p = (x)-> x > 3;
   if (p.test(4)) { System.out.println ("foo"); }
```

Lambda-Ausdrücke in Javascript

```
$ node

> let f = function (x){return x+3;}
undefined

> f(4)
7

> ((x) => (y) => x+y) (3) (4)
7

> ((f) => (x) => f(f(x))) ((x) => x+1) (0)
2
```

Übung Lambda-Kalkül

- abstrakten Syntaxbaum und Normalform von $SKKc$, wobei $S = \lambda xyz.xz(yz), K = \lambda ab.a$,

- (mit `data N=Z|S N`) bestimme Normalform von $ttSZ$ für $t = \lambda fx.f(fx)$,

- definiere Λ als algebraischen Datentyp

```
data L = Var String | App L L | Abs String L
```

  implementiere `size :: L -> Int`,
  `depth :: L -> Int`.

  implementiere `bvar :: L -> S.Set String`,
  `fvar :: L -> S.Set String`,

siehe Folie mit Definitionen und dort angegebene Testfälle

benutze `import qualified Data.Set as S`,
API-Dokumentation: `https://hackage.haskell.org/package/containers/docs/Data-Set.html`

Teillösung:

```
bvar :: L -> S.Set String
bvar t = case t of
  Var v   -> S.empty v
  App l r -> S.union (bvar l) (bvar r)
  Abs v b -> S.insert v (bvar b)
```

Hausaufgaben

1. `fvar` implementieren (vgl. Übungsaufgabe `bvar`), Testfälle vorführen (aus Skript und weitere)

2. für $t = \lambda f x.f(fx)$: AST von $ttSZ$ zeichnen, Normalform bestimmen: 1. auf Papier, 2. mit `ghci`.

   ```
   let t = \ f x -> f (f x)
   ...
   ```

   möglichst kleine Lambda-Ausdrücke (vom Typ `N`) angeben (Variablen, Applikation, Abstraktion, Konstruktoren von `data N`, *keine* Funktionen wie plus, mal), die eine große Normalform haben (mit `ghci` ausrechnen)

3. für $s = \lambda xyz.xz(yz)$ und $k = \lambda ab.a$:

   Auswertung von $skk0$ in Haskell, in Javascript (`node`), optional: in C# (`csharp`), Java (`jshell`), oder anderer Sprache

   - (autotool) Reduktionen im Lambda-Kalkül

L-K.: statische Semantik (Typisierung)

Motivation, Plan

- (Mathematik) Funktion $f : A \to B$ als Relation zw. Definitionsbereich A und Wertebereich B
- (Programmiersprache) Typ-Deklaration `f :: A -> B`
- (Softwaretechnik) der Typ eines Bezeichners ist seine beste Dokumentation— denn sie wird bei *jeder* Code-Änderung maschinell überprüft.
  andere (schlechtere) Varianten: 1. niemals, 2. nur in Entwurfsphase (bei separater Entwurfssprache)
- funktionale Programmierung (Funktionen als Daten): A und B können selbst Mengen von Funktionen sein

Grundlagen zur Typisierung

- – *statische* Typisierung: jeder Bezeichner, jeder Ausdruck (im Quelltext) hat einen Typ
  – *dynamische* Typisierung: jeder Wert (im Hauptspeicher) hat einen Typ
- statische Typisierung verhindert Laufzeit(typ)fehler
  $\Rightarrow$ alle wichtigen Laufzeiteigenschaften sollen als Typ-Aussagen formuliert werden
- flexible (wiederverwendbare) *und* sichere Software durch generische Polymorphie (Typ-Argumente)
- für statische Typisierung unterscheiden wir:
  – *Typ-Deklaration* (der Typ steht sichtbar im Quelltext)
  – *Typ-Inferenz* (der Typ wird vom Compiler bestimmt)
- C#/Java: polym. deklariert, ML/Haskell: polym. inferiert

Einfache (monomorphe) Typen

- benutzt diese Typisierungsregeln:
  - Abstraktion *erzeugt* Funktions-Typ:
    wenn $x :: T_1$ und $B :: T_2$, dann $\lambda x.B :: T_1 \to T_2$
  - Applikation *benutzt* Funktions-Typ:
    wenn $f :: (T_1 \to T_2)$ und $a :: T_1$, dann $fa :: T_2$
    das für f deklarierte T_1 muß genau der Typ von a sein
- damit kein sicherer wiederverwendbarer Code möglich:
  denn Bibliotheksfunktionen (Bsp. f) können
  anwendungsspezifische Typen (Bsp: von a) nicht kennen
- (schlechte) Auswege: keine statische Typisierung
  - für die Bibliothek (z.B. `Object` in Mittelalter-Java)
  - für die Sprache (Bsp: Sprachen mit P (nicht Pascal))
  oder keine anwendungsspez. Typen (sondern `int`)

Typ-Abstraktion und -Applikation

- Fkt. mit monomorphem Typ (Bsp):

```
f :: Bool -> N; f x = S Z ;  f False
-- Deklaration, Definition,  Anwendung
```

- Fkt. mit polymorphem Typ (Bsp), mit Typ-Abstraktion

```
g :: forall (t :: Type) . t -> t; g x = x
```

- Typ-Applikation: `g @N` ergibt monomorphes `N -> N`

  dann (Daten-)Applikation: `(g @N) Z`

- Beispiele: `g @Bool (g @Bool False)`

```
data List a = Nil | Cons a (List a)
Cons :: forall (a :: Type) . a -> List a -> List a
Cons @N Z (Nil @N)
```

Polymorphie: Notation

- volle Notation in Haskell leider umständlich:

```
{-# language ExplicitForall, KindSignatures #-}
import Data.Kind (Type)
g :: forall (t :: Type) . t -> t ; g x = x

{-# language TypeApplications #-}  g @N Z
```

- in Java

```
class C { static <A> A id (A x) {return x;}}
C.<Integer>id(9);  // Integer = der Box-Typ von int
```

- in C#

```
class C { public static A id<A> (A x){return x;}}
C.id<int> (9)
```

Polymorphie: Notation: Abkürzungen

- (Haskell) in der Typ-Abstraktion:

  Deklaration (Quantor) der Typvariablen weglassen

```
g :: forall (t :: Type) . t -> t ; g x = x
h ::                      t -> t ; h x = x
```

- (Haskell, Java, C#) unsichtbare Typ-Applikation:

  Typ-Argumente werden durch Compiler inferiert

```
g Z  ;   C.id(9) ; C.id(9)
```

- wie bei jeder Abkürzung: das kann
  - nützlich sein
  - aber auch irreführend

Verkürzte Notation für Typen

- Der Typ-Pfeil ist *rechts-assoziativ*:
  $T_1 \to T_2 \to \cdots \to T_n \to T$ bedeutet
  $(T_1 \to (T_2 \to \cdots \to (T_n \to T) \cdots))$
- das paßt zu den Abkürzungen für mehrstellige Funkt.:
  $\lambda(x :: T_1).\lambda(x :: T_2).(B :: T)$ hat den Typ $(T_1 \to (T_2 \to T))$,
  mit o.g. Abkürzung $T_1 \to T_2 \to T$.
- Beispiel: wir schreiben

```
plus :: N -> N -> N; plus x y = case x of ...
a = plus 2 3
```

  es bedeutet

```
plus :: N -> (N -> N); plus = \ x -> \ y -> case ..
a = (plus 2) 3
```

Nützliche Funktionen höherer Ordnung

- ```
  compose :: (b -> c) -> ((a -> b) -> (a -> c))
  --      :: (b -> c) -> (a -> b) -> a -> c
  compose f g x = f (g x)
  ```
  diese Funktion in der Standard-Bibliothek:
  der Operator `.` (Punkt)

- `flip :: (a -> b -> c) -> b -> a -> c`
  aus dem Typ folgt schon die Implementierung!

  ```
  flip f x y = ...
  ```

- `apply f x = f x`
  das ist der Operator `$` (Dollar), benutzt zum Einsparen von Klammern: `f (g (h x)) = f $ g $ h x`
  denn `($)` ist *rechts-assoziativ*
  ```

Der allgemeinste Typ eines Ausdrucks

- Haskell benutzt *Algorithmus W* von Damas/Milner, der zu jedem Lambda-Ausdruck A
  - bestimmt, ob A typisierbar ist (ob ein T exist. mit $A :: T$)
  - wenn ja, einen *allgemeinsten Typ* T_p von A inferiert (für jedes T mit $A :: T$ exists. Subst. σ mit $T_p\sigma = T$)
- Luis Damas, Robin Milner: *Principal Type Schemes for Functional Programs*, 1982; Roger Hindley: *The Principal Type Scheme of an object in Combinatory Logic*, 1969; Mark P. Jones: *Typing Haskell in Haskell*, 2000

  `http://web.cecs.pdx.edu/~mpj/thih/`
- der deklarierte Typ muß Instanz des inferierten Typs sein
- deswegen könnte man Typ-Deklarationen weglassen, sollte man aber nicht (Typ-Dekl. ist Dokumentation)

Beispiel für Typ-Bestimmung

Aufgabe: bestimme den allgemeinsten Typ von $\lambda f x.f(fx)$

- Ansatz mit Typvariablen $f :: t_1, x :: t_2$
- betrachte (fx): der Typ von f muß ein Funktionstyp sein, also $t_1 = (t_{11} \to t_{12})$ mit neuen Variablen t_{11}, t_{12}.
  Dann gilt $t_{11} = t_2$ und $(fx) :: t_{12}$.
- betrachte $f(fx)$. Wir haben $f :: t_{11} \to t_{12}$ und $(fx) :: t_{12}$, also folgt $t_{11} = t_{12}$. Dann $f(fx) :: t_{12}$.
- betrachte $\lambda x.f(fx)$.
  Aus $x :: t_{12}$ und $f(fx) :: t_{12}$ folgt $\lambda x.f(fx) :: t_{12} \to t_{12}$.
- betrachte $\lambda f.(\lambda x.f(fx))$.
  Aus $f :: t_{12} \to t_{12}$ und $\lambda x.f(fx) :: t_{12} \to t_{12}$
  folgt $\lambda f x.f(fx) :: (t_{12} \to t_{12}) \to (t_{12} \to t_{12})$

Stelligkeit von Funktionen

- ist `plus` in `flip` richtig benutzt? Ja!

```
flip :: (a -> b -> c) -> b -> a -> c
data N = Z | S N
plus :: N -> N -> N
plus (S Z) (S (S Z)) ; flip plus (S Z) (S (S Z))
```

- beachte Unterschied zwischen:
  - Term-Ersetzung: Funktionssymbol $\rightarrow$ Stelligkeit
    abstrakter Syntaxbaum: Funktionss. über Argumenten
  - Lambda-Kalkül: jeder Lambda-Ausdruck beschreibt einstellige Funktion
    Simulation mehrstelliger Funktionen wegen Isomorphie zwischen $(A \times B) \rightarrow C$ und $A \rightarrow (B \rightarrow C)$
- Übung/Beispiele: die Funktionen `curry`, `uncurry`

Beispiele Fkt. höherer Ord.

- Haskell-Notation für Listen:

```
data List a = Nil | Cons a (List a)
data   [a]  =  []  |       a : [a]
```

- Verarbeitung von Listen:

```
filter :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
partition :: (a -> Bool) -> [a] -> ([a],[a])
```

- Vergleichen, Ordnen:

```
nubBy :: (a -> a -> Bool) -> [a] -> [a]
data Ordering = LT | EQ | GT
minimumBy
  :: (a -> a -> Ordering) -> [a] -> a
```

- When You Should Use Lists in Haskell (Mostly, You Should Not) `https://arxiv.org/abs/1808.08329`

Fkt. höherer Ord. für Folgen

- Folgen, repräsentiert als balancierte Bäume:

```
module Data.Sequence where
data Seq a = ...
   -- keine sichtbaren Konstruktoren!
fromList :: [a] -> Seq a
filter :: (a -> Bool) -> Seq a -> Seq a
takeWhile :: (a -> Bool) -> Seq a -> Seq a
```

- Anwendung:

```
import qualified Data.Sequence as Q
xs = Q.fromList [1, 4, 9, 16]
ys = Q.filter (\x -> 0 == mod x 2) xs
```

Fkt. höherer Ord. für Mengen

- Mengen, repräsentiert als balancierte Such-Bäume:

```
module Data.Set where
data Set a = ...
   -- keine sichtbaren Konstruktoren!
fromList :: Ord a => [a] -> Set a
filter :: Ord a => (a -> Bool) -> Set a ->
```

  das *Typ-Constraint* `Ord a` schränkt die Polymorphie ein (der Typ, durch den die Typ-Variable `a` instantiiert wird, muß eine Vergleichsmethode haben)

- Anwendung:

```
import qualified Data.Set as S
xs = S.fromList [1, 4, 9, 16]
ys = S.filter (\x -> 0 == mod x 2) xs
```

Polymorphe Unterprogr. in anderen Sprachen

- C# und Java haben: statische Typisierung, Polymorphie (Inferenz für Typ-Argum., nicht für allgemeinsten Typ), Lambda-Ausdrücke (anonyme Funktionen), aber
  - polymorphe Unterprogramme: nur als Methoden,
  - Funktione als Daten: nur als Lambdas (Objekte)
    - deren Typ kann nicht inferiert werden (für `var`)
    - und keine neuen Typvariablen einführen

- ```
  Func<int,Func<bool,int>> k = x => y => x;
  class C{public static A k<A,B>(A x,B y) {return x;}
  ```

- ```
  import java.util.function.*;
  Function<Integer,Function<Boolean,Integer>>
    k = (x) -> (y) -> x;
  k.apply(0).apply(true)
  ```

Zusammenfassung, Ausblick

- generische Polymorphie entsteht durch allgemeinste Typen von Lambda-Ausdrücken (Bps: $\lambda x.x$)
- generische Polymorphie ist nützlich für flexibel verwendbare Container (Folge von . . . , Baum von . . .)
- deren Implementierung kann über Element-Typ *gar nichts* voraussetzen (denn dieser ist all-quantifiziert)
- die Steuerung der Implementierung erfolgt dann durch Funktionen als Daten (Bsp: eine Vergleichsfunktion)
- die Implementierung ist dann eine Funktion höherer (zweiter) Ordnung
- später: implizite Übergabe dieser Funktionen als Wörterbücher (Methodentabellen) von Typklassen
- aber vorher: Rekursionmuster (als Fkt. höherer Ord.)

Übung

- den allgemeinsten Typ eines Lambda-Ausdrucks bestimmen, Beispiel

```
compose ::
compose = \ f g -> \ x -> f (g x)
```

Musterlösung:

– wegen `g x` muß `g :: a -> b` gelten,
dann `x :: a` und `g x :: b`
– wegen `f (g x)` muß `f :: b -> c` gelten,
dann `f (g x):: c`
– dann `\ x -> f (g x) :: a -> c`
– dann

```
\ f g -> .. :: (b->c) -> (a->b) -> (a->c)
```

- Isomorphie zwischen $(A \times B) \to C$ und $A \to (B \to C)$:
  falls A, B, C endlich: die Größen beider Mengen ausrechnen und vergleichen
  für $A = B = C =$ Bool: ein Element von $(A \times B) \to C$ angeben und das dazugehörige Element von $A \to (B \to C)$.
  begründen, daß es im allgemeinen keine Isomorphie zu $(A \to B) \to C$ gibt (die Größe ausrechnen)

- Implementierung von takeWhile

```
takeWhile :: (a -> Bool) -> List a -> List
takeWhile p xs = case xs of
  Nil -> Nil
  Cons x xs' -> case p x of
```

```
    False -> Nil
    True  -> Cons x (takeWhile p xs')
```

Hausaufgaben

im SS 21: Aufgaben 1, 3, 4, 5

1. für die Lambda-Ausdrücke

   (a) $\lambda f g h x y . f (g x) (h y)$:
   (b) $(\lambda x . x x)(\lambda y . y y)$

   AST zeichnen, allgemeinsten Typ bestimmen, falls möglich

   (1. von Hand, mit Begründung,

   2. mit ghci überprüfen)

2. für die Ausdrücke

   ```
   flip
   ```

```
flip flip
flip flip flip
...
```

jeweils Argumente $a_1 a_2 \ldots$ angeben (passende Anzahl), so daß der Wert von `flip flip ... flip a1 a2 ...` gleich `True` ist.

Dabei alle Typargumente für alle `flip` explizit angeben.

3. in Haskell: Lambda-Ausdrücke mit diesen Typen angeben, falls möglich:

```
a -> (a -> c) -> c
a -> b
(a -> b) -> (b -> c) -> a -> c
```

mit ghci überprüfen (Typen anzeigen lassen)

die Ausdrücke dann auch mit passenden Argumenten aufrufen, so daß der Wert `True` ist (oder von einem andern `data`-Typ, jedenfalls keine Funktion)

4. (einige der) Funktionen `compose`, `flip`, `curry`, `uncurry`

   in C# (`csharp`) und Java (`jshell`) deklarieren (vorher prüfen, ob es die schon gibt) (anhand von Primärquellen)

   und aufrufen (dabei Typ-Argumente explizit angeben)

   Den Typ für 2-Tupel aus der jeweiligen Standardbibliothek benutzen.

5. die Funktionen `partition`, `span`

(a) aufrufen (in ghci Beispiele vorführen), dabei anwenden auf eine Liste von Bool
(b) die API-Dokumentation aufsuchen (hackage.org)
(c) die dort angegebenen Eigenschaften als Leancheck-Property überprüfen
(d) die entsprechenden Funktionen aus `Data.Sequence` aufrufen

- (autotool) Typisierung im Lambda-Kalkül
- (autotool) `dropWhile` implementieren und beweisen

```
xs=append (takeWhile p xs) (dropWhile p xs)
```

Rekursionsmuster

Rekursion über Bäume (Beispiele)

- ```
  data Tree a = Leaf
              | Branch (Tree a) a (Tree a)
  ```

- ```
  summe :: Tree N -> N
  summe t = case t of
    Leaf -> 0
    Branch l k r ->
      plus (summe l) (plus k (summe r))
  ```

- ```
  preorder :: Tree a -> List a
  preorder t = case t of
    Leaf -> Nil
    Branch l k r ->
      Cons k (append (preorder l) (preorder r))
  ```
  ```

Rekursion über Bäume (Schema)

```
f :: Tree a -> b
f t = case t of
  Leaf -> ...
  Branch l k r -> ... (f l) k (f r)
```

dieses Schema *ist* eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> (Tree a -> b)
fold leaf branch = \ t -> case t of
  Leaf -> leaf
  Branch l k r ->
     branch (fold leaf branch l)
            k (fold leaf branch r)
summe = fold Z (\ x y z-> plus x (plus y z))
```

(bei Aufruf ist dann `x=summe l`, `y=k`, `z=summe r`)

Rekursion über Listen (Bsp. und Schema)

```
and :: List Bool -> Bool
and xs = case xs of
  Nil -> True ; Cons x xs' -> x && and xs'
length :: List a -> N
length xs = case xs of
  Nil -> Z ; Cons x xs' ->  S (length xs')

fold :: b -> ( a -> b -> b ) -> List a -> b
fold nil cons xs = case xs of
    Nil -> nil
    Cons x xs' -> cons x ( fold nil cons xs'
and = fold True (&&)
length = fold Z ( \ x y ->  S y)
```

Rekursionsmuster (Prinzip)

- ```
  data List a = Nil | Cons a (List a)
  fold ( nil :: b ) ( cons :: a -> b -> b )
     :: List a -> b
  ```

- Rekursionsmuster anwenden

  $=$ jeden Konstruktor durch eine passende Funktion ersetzen

  $=$ (Konstruktor-)Symbole *interpretieren* (durch Funktionen)

  $=$ eine *Algebra* angeben (Signatur $=$ Menge der Konstruktoren, Universum $=$ Resultattyp des Musters)

- ```
  length  = fold Z  ( \ _ l -> S l )
  ```

Rekursionsmuster (Merksätze)

aus dem Prinzip *ein Rekursionsmuster anwenden* $=$ *jeden Konstruktor durch eine passende Funktion ersetzen* folgt:

- Anzahl der Muster-Argumente $=$ Anzahl der Konstruktoren (plus eins für das Datenargument)
- Stelligkeit eines Muster-Argumentes $=$ Stelligkeit des entsprechenden Konstruktors
- Rekursion im Typ $\Rightarrow$ Rekursion im Muster

  (Bsp: zweites Argument von `Cons`)
- zu jedem rekursiven Datentyp gibt es *genau ein* passendes Rekursionsmuster

Argumente für Rekursionsmuster finden

systematisches experimentelles Vorgehen zur Lösung von:
„Schreiben Sie Funktion $f : T \to R$ als `fold`“

- eine Beispiel-Eingabe ($t \in T$) notieren

  (als Baum zeichnen, Knoten $=$ Konstruktoren)

- für jeden Teilbaum s von t, der den Typ T hat:

  den Wert von $f(s)$ in (neben) Wurzel von s schreiben

- daraus Testfälle für die Funktionen ableiten,

  die die Konstrukten ersetzen

  diese Fkt. sind die Argumente des Rekursionsmusters

Beispiel: `reverse :: List a -> List a`

Rekursionsschema f. mehrstellige Fkt.

- `fold` hat immer *genau ein* Daten-Argument
  (in welchem die Konstruktoren ersetzt werden)
- wie stellt man mehrstellige Funktionen dar? Bsp.

```
append :: List a -> List a -> List a
```

- Lösung: das ist gar nicht zweistellig, sondern

```
append :: List a -> (List a -> List a)
```

- `fold` über 1. Arg., Resultat `::` `List a -> List a`

```
app = fold nil cons -- Ansatz
app Nil ys = fold nil cons Nil ys = nil ys = ys
app (Cons x xs) ys = fold nil cons (Cons x xs)
  = cons x (app xs ys) = Cons x (app xs ys)
app = fold (\ ys -> ys) (\ x zs -> Cons x zs )
```

das ist systematisches *symbolisches* Vorgehen

Rekursionsmuster (Peano-Zahlen)

- aus dem Typ das Rekursionsschema ableiten:

```
data N = Z | S N
fold :: ...
fold z s n = case n of
    Z    -> _
    S n' -> _
```

- Bsp: verdoppeln `dbl :: N -> N; dbl = fold _ _`
- Bsp: mit Fold über erstem Arg.

```
plus  = fold ...
times = fold ...
```

Herleitung d. Beispiele oder d. Umformung (vgl. `app`)

Nicht durch Rekursionmuster darstellbare Fkt.

- Beispiel: `data N = Z | S N`,

  f : `N` $\rightarrow$ `Bool`, $f(x) =$ „x ist durch 3 teilbar“

- wende eben beschriebenes Vorgehen an,

- stelle fest, daß die durch Testfälle gegebene Spezifikation nicht erfüllbar ist

- Beispiel: binäre Bäume mit Schlüssel in Verzweigungsknoten,

  f : `Tree k` $\rightarrow$ `Bool`,

  $f(t) =$ „t ist höhen-balanciert (erfüllt die AVL-Bedingung)“

Darstellung mit Fold und Projektion

- f : `Tree k` $\rightarrow$ `Bool`,
  $f(t) =$ „t ist höhen-balanciert (erfüllt die AVL-Bedingung)“
  f *ist nicht* als fold darstellbar
- g : `Tree k` $\rightarrow$ `Pair Bool N`; $g(t) = (f(t), \texttt{height}(t))$
  g *ist* als fold darstellbar
- dann `f t = first (g t)` mit *Projektion*

```
first::Pair a b -> a;first (Pair x y) = x
```

- NB: alternativ: *punktfreie* Notation `f = first . g`
  der (fehlende) Punkt ist das Argument t, nicht der Kompositions-Operator `(.)`; diese Bezeichnung ist übernommen aus linearer Algebra: Rechnen mit Vektoren/Matrizen ohne Notation von Indices.

Zusammenfassung Rekursionmuster

- zu jedem (rekursiven) algebraischen Datentyp gibt es genau ein Rekursionmuster (der übliche Name ist `fold`)
  das kann systematisch (mechanisch) konstruiert werden
  (NB: Konstruktion auch für nicht rekursive Typen anwendbar und nützlich, siehe Aufgabe)
- der *Grund* ist: algebraischer Datentyp $=$ Signatur Σ,
  Instanz eines Rekursionsmusters $=$ eine Σ-Algebra
- das *Hilfsmittel* zur Notation ist: Funktion höherer Ordnung
- der softwaretechnische *Zweck* ist:
  die Programmablaufsteuerung (Rekursion) wird vom Anwendungsprogramm in die Bibliothek verschoben,
  AP wird dadurch einfacher (kürzer, klarer)

Spezialfälle des Fold

- jeder Konstruktor durch sich selbst ersetzt,
  mit unveränderten Argumenten: *identische* Abbildung

```
data List a = Nil | Cons a (List a)
fold :: r -> (a -> r -> r) -> List a -> r
fold Nil Cons (Cons 3 (Cons 5 Nil))
```

- jeder Konstruktor durch sich,
  mit transformierten Argumenten v. nicht-rekursiven Typ

```
fold Nil (\x y -> Cons (not x) y)
   (Cons True (Cons False Nil))
```

struktur-erhaltende Abbildung. Diese heißt *map*.

```
map :: (a -> b) -> List a -> List b
```

Rekursion über Listen aus Std.-Bib.

das vordefinierte Rekursionsschema über Listen ist:

```
foldr :: (a -> b -> b) -> b -> ([a] -> b)
length = foldr ( \ x y -> 1 + y ) 0 -- Bsp
```

- falsche Argument-Reihenfolge beachten
  (paßt nicht zu Konstruktor-Reihenfolge)
- `foldr` (fold right) nicht mit `foldl` (fold left) verwechseln
  (foldr ist das „richtige“, genau Betrachtung später)
- der Typ von `foldr` ist tatsächlich allgemeiner (auch für andere Container anwendbar, genaueres später)

Aufgaben:

- `append, reverse, concat, inits, tails`

Weitere Übungsaufgaben zu Fold

- `data List a = Nil | Cons a (List a)`
  `fold :: r -> (a -> r -> r) -> List a -> r`

- schreibe mittels `fold` (ggf. verwende `map`)
  - `inits, tails :: List a -> List (List a)`
    `inits [1,2,3] = [[],[1],[1,2],[1,2,3]]`
    `tails [1,2,3] = [[1,2,3],[2,3],[3],[]]`
  - `filter :: (a -> Bool) -> List a -> List a`
    `filter odd [1,8,2,7,3] = [1,7,3]`
  - `partition :: (a -> Bool) -> List a`
    `-> Pair (List a) (List a)`
    `partition odd [1,8,2,7,3]`
    `= Pair [1,7,3] [8,2]`

Rekursionsmuster für Bäume (Bsp. 2)

- `data Tree a -- Schlüssel NUR in den Blätter`
  `= Leaf a | Branch (Tree a) (Tree a)`

- aus dem Typ das Rekursionsschema ableiten

  `fold :: ...`

- Beispiele (jeweils zunächst den Typ angeben!)
  - Anzahl der Blätter
  - Anzahl der Verzweigungsknoten
  - Summe der Schlüssel
  - die Tiefe des Baumes
  - der größte Schlüssel
  - der Schlüssel links unten (auf Position $p \in 0^*$)

Übung Rekursionsmuster

- Rekursionsmuster `foldr` für Listen benutzen (filter, takeWhile, append, reverse, concat, inits, tails)
- Rekursionmuster für Peano-Zahlen hinschreiben und benutzen (plus, mal, hoch, Nachfolger, Vorgänger, minus)
- Rekursionmuster für binäre Bäume mit Schlüsseln *nur in den Blättern* hinschreiben und benutzen
- Rekursionmuster für binäre Bäume mit Schlüsseln *nur in den Verzweigungsknoten* benutzen für rekursionslose Programme für:
  - Anzahl der Branch-Knoten ist ungerade (nicht zählen!)
  - Baum (`Tree a`) erfüllt die AVL-Bedingung

Hinweis: als Projektion auf die erste Komponente eines `fold`, das Paar von `Bool` (ist AVL-Baum) und `N` (Höhe) berechnet.

– Baum (`Tree N`) ist Suchbaum (ohne `inorder`)
Hinweis: als Projektion. Bestimme geeignete Hilfsdaten.

Implementierungen für natürlichen Zahlen

- Eigenbau: die Peano-Zahlen (dann sind alle Operationen und Relationen selbst zu programmieren)
- Bibliothek: (dann können `0, +, -, .., <, >, ..` benutzt werden — deren Typ aber erst später erklärt wird)
  - der Typ `Natural` (nach `import Numeric.Natural`) effiziente Repräsentation beleibig großer Zahlen
  - der Typ `Word` (nach `import Data.Word`) Maschinenzahl, repräsentiert $\mathbb{N}$ modulo $2^{\text{Wortbreite}}$

Merksätze

- `Word` (`uint`) riskant, `Int` (`int`) riskant und falsch (für $\mathbb{N}$)
- sicher sind nur: beliebig groß oder: mit Überlauf-Prüfung.

Hausaufgaben für KW 22

1. Wenden Sie die Vorschrift zur Konstruktion des Rekursionsmusters an auf den Typ

   - `Bool`
   - `Maybe a`
   - `Pair a b`
   - `Either a b`

   Jeweils:

   - Typ und Implementierung (vorbereiteten Quelltext zeigen)
   - Testfälle (in ghci vorführen)
   - gibt es diese Funktion bereits? Suchen Sie nach dem Typ mit `https://www.haskell.org/hoogle/`

2. Rekursionsmuster auf Peano-Zahlen

- plus, mal, hoch vorführen (jeweils realisiert als fold)
  Testen Sie mit Leancheck einige Eigenschaften.
- Beweisen Sie, daß die modifizierte Vorgängerfunktion
  `pre :: N -> N; pre Z = Z;  pre (S x) = x`
  kein fold ist.

ggf. ergänzende Aufgaben in autotool (nicht vorführen):

- Diese Funktion `pre` kann jedoch als Projektion einer geeigneten Hilfsfunktion `h :: N -> (N,N)` realisiert werden. Spezifizieren Sie `h` und geben Sie eine Darstellung von `h` als fold an.
- Implementieren Sie die (modifizierte) Subtraktion `minus` mit `fold` (über das erste Argument) (und `pre`)

3. Rekursionsmuster auf Eigenbau-Listen:

- mit `fold` und ohne Rekursion implementieren:
  `isPrefixOf :: Eq a => List a -> List a -> Bool`
  siehe Folie Rekursionsmuster für mehrstellige Funktion.
  (Das *Typ-Constraint* `Eq a` bedeutet, daß der Operator `(==) :: a -> a -> Bool` benutzt werden kann, genaue Erklärung dazu später.)
- die Eigenschaft `isPrefixOf xs (append xs ys)` mit Leancheck überprüfen (für `xs :: List Bool`)
  Diese Eigenschaft abändern und Gegenbeispiele diskutieren.
- für `isSuffixOf`: 1. konkrete Testfälle, 2. allgemeine Eigenschaft, 3. Implementierung ohne Rekursion, ohne Fold, aber unter Benutzung von `reverse` angeben.

4. Rekursionsmuster auf binären Bäumen (Schlüssel in Verzweigungsknoten)

- Beweisen Sie, daß `is_search_tree :: Tree N -> Bool` kein fold ist.
- diese Funktion kann jedoch als Projektion einer Hilfsfunktion `h :: Tree N -> (Bool, Maybe (N,N))` erhalten werden, die für Bäume mit nicht-leerer Schlüsselmenge auch noch deren kleinstes und größtes Element bestimmt. Stellen Sie `h` als fold dar.

Bemerkung zu $\mathbb{N}$ beachten. Hier ist `Numeric.Natural` sinnvoll.

Eingeschränkte Polymorphie

Typklassen in Haskell: Überblick

- abstrakter Datentyp (ADT) = Signatur (Menge von Funktions*deklarationen* = Name und Typ) + Axiome
  konkreter Datentyp = Algebra = Menge von Funktions*implementierungen*
- Haskell: `class C; data T; instance C T where ...`
  OO ähnlich: `interface C; class T implements C {...}`
- Bsp. Klassed der Typen mit totaler Ordnung
  `class Ord a, interface Comparable<E>`
- die Auswahl der Implementierung (Methodentabelle):
  - Haskell - statisch (abh. von Argument/Resultattypen)
  - OO - dynamisch (abh. v. Laufzeittyp des 1. Argumentes)

Beispiel

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
sortBy ( \ x y -> ... ) [False, True, False]
```

Kann mit Typklassen so formuliert werden:

```
class Ord a where
    compare :: a -> a -> Ordering
sort :: Ord a => [a] -> [a]
instance Ord Bool where compare x y = ...
sort [False, True, False]
```

- `sort` hat *eingeschränkt polymorphen Typ*
- die Einschränkung (das Constraint `Ord a`) wird in ein zusätzliches Argument (eine Funktion) übersetzt. Entspricht OO-Methodentabelle, liegt aber *statisch* fest.

Grundwissen Typklassen

- Typklasse schränkt statische Polymorphie ein

  (Typvariable darf nicht beliebig substitutiert werden)
- Einschränkung realisiert durch *Wörterbuch*-Argument

  (W.B. $=$ Methodentabelle, Record von Funktionen)
- durch Instanz-Deklaration wird Wörterbuch erzeugt
- bei Benutzung einer eingeschränkt polymorphen Funktion: passendes Wörterbuch wird statisch bestimmt
- nützliche, häufige Typklassen: `Show`, `Read`, `Eq`, `Ord`.

  (`Test.LeanCheck.Listable`, `Foldable`, `Monad`,...)
- Instanzen automatisch erzeugen mit `deriving`

Typklassen in Bibliotheks-Schnittstellen

- Realisierung von Mengen durch Suchbäume:

  die Polymorphie im Element/Schlüsseltyp muß eingeschränkt werden: der Typ muß total geordnet sein

```
import Data.Set -- aus Bibliothek: containers
fromList :: Ord a => [a] -> Set a
insert   :: Ord a => a -> Set a -> Set a
```

- Realisierung von Mengen durch Hashtabellen:

  . . . muß Hashfunktion und Gleichheitstest besitzen

```
import Data.HashSet -- Bib.: unordered-containers
fromList :: (Eq a,Hashable a)=>[a]->HashSet a
insert::(Eq a,Hashable a)=>a->HashSet a->HashSet a
```

(Eingeschränkt) polymorphe Literale

- Literal = Bezeichner für feststehendes Datum

  (Gegensatz: Name in Muster: Bedeutung erst zur Laufzeit durch pattern match festgelegt)

- `data List a = Nil | Cons a (List a)`

  `Nil` ist polymorphes Literal

  es gilt `Nil :: forall (a::Type) . List a`

- Zahl-Literale haben eingschränkt polymorphen Typ

```
class Num a where { (+) :: a->a->a; ..}
instance Num Integer where  { ... }
instance Num Natural where  { ... }
42 :: Num a => a
```

Unterschiede Typklasse/Interface (Bsp)

- Typklasse/Schnittstelle

```
class Show a where show :: a -> String
interface Show { String show (); }
```

- Instanzen/Implementierungen

```
data A = A1 ; instance Show A where ..
class A implements Show { .. }
```
entspr. für `B`

- in Java ist `Show` ein Typ:

```
static String showList(List<Show> xs) { ..
showList (Arrays.asList (new A(),new B()))
```

- in Haskell ist `Show` ein Typconstraint und kein Typ:

```
showList :: Show a => List a -> String
showList [A1,B1]
```
ist Typfehler

Unterschiede Typklasse/Interface (Impl.)

- Haskell:

  `f :: (Constr1, ..) => t1 -> t2 -> .. -> res`

  Definition `f par1 par2 .. = ..` wird (statisch) übersetzt in `f dict1 .. par1 par2 .. = ...`

  Aufruf `f arg1 arg2 ..` wird (statisch) übersetzt in `f dict1 .. arg1 arg2 ..`

- Java:

  `inter I { .. f (T2 par2 ) }; T1 implements`

  bei Aufruf `arg1.f(arg2)` wird Methodentabelle des Laufzeittyps von `arg1` benutzt (*dynamic dispatch*)

- dyn. disp. in Haskell stattdessen durch pattern matching

Typklassen/Interfaces: Vergleich

- grundsätzlicher Unterschied: stat./dynam. dispatch
- die Methodentabelle wird von der Klasse abgetrennt

  und extra behandelt (als Wörterbuch-Argument):
  - einfachere Behandlg. von Fkt. mit > 1 Argument
    (sonst: vgl. `T implements Comparable<T>`)
  - mehrstellige Typconstraints: Beziehungen zwischen mehreren Typen,
    ```
    class Autotool problem solution
    ```
  - Typkonstruktorklassen,
    ```
    class Foldable c where toList:: c a -> [a]
    data Tree a = ..;instance Foldable Tree
    ```
    (wichtig für fortgeschrittene Haskell-Programmierung)

Generische Instanzen (I)

```
class Eq a where
    (==) :: a -> a -> Bool
```

Vergleichen von Listen (elementweise)

wenn a in Eq, *dann* [a] in Eq:

```
instance Eq a => Eq [a] where
  l == r = case l of
    [] -> case r of
      [] -> True ; y : ys -> False
    x : xs -> case r of
      [] -> False
      y : ys -> (x == y) && ( xs == ys )
```

Übung: wie sieht `instance Ord a => Ord [a]` aus?

(lexikografischer Vergleich)

Generische Instanzen (II)

```
class Show a where
  show ::  a -> String
instance Show Bool where
  show False = "False" ; show True = "True"
instance Show a => Show [a] where
  show xs = brackets
          $ concat
          $ intersperse ","
          $ map show xs
instance (Show a, Show b) => Show (a,b) wher
show False = "False"
show ([True,False],True)
  = "([True,False],True)"
```

Benutzung von Typklassen bei Leancheck

- Rudy Matela: *LeanCheck: enumerative testing of higher-order properties*, Kap. 3 in Diss. (Univ. York, 2017)
  `https://matela.com.br/paper/rudy-phd-thesis-2017.pdf`
  - Testen von universellen Eigenschaften
    ($\forall a \in A : \forall b \in B : p\ a\ b$)
  - automatische Generierung der Testdaten …
  - … aus dem Typ von p
  - … mittels generischer Instanzen

  `https://github.com/rudymatela/leancheck`
- beruht auf: Koen Classen and John Hughes: *Quickcheck: a lightweight tool for random testing of Haskell programs*, ICFP 2000, (“most influential paper award”, 2010)

Test.LeanCheck—Beispiel

- ```
  assoc :: forall a . Eq a =>(a->a->a)->a->a->a->Bool
  assoc op = \ a b c -> op a (op b c)==op (op a b) c
  main = check (assoc @[Bool] (++))
  ```

- dabei werden benutzt (in vereinfachter Darstellung)
  - `class Testable p where check :: p -> Bericht`
    Instanzen sind alle Typen, die testbare Eigenschaften repräsentieren

    ```
    instance Testable Bool where
      check False = "falsch"; check True = "OK"
    ```
  - `type Tiers a = [[a]]`
    `class Listable a where tiers :: Tiers a`
    Instanzen sind alle Typen, die sich aufzählen lassen (Schicht (*tier*) i: Elemente der Größe i)

    ```
    instance Listable Bool where tiers = [[False,True]]
    ```
    ```

Testen für beliebige Stelligkeit

- warum geht eigentlich beides (einstellig, zweistellig)

```
check $ \ x -> x || not x
check $ \ x y -> not (x && y) == not x || not y
```

- weil gilt `instance Testable (Bool -> Bool)`
  und `instance Testable (Bool -> (Bool -> Bool))`
- das wird vom Compiler abgeleitet (inferiert) aus:

```
instance Listable Bool ...; instance Testable Bool
instance (Listable a, Testable b)
  => Testable (a -> b) where { ... }
```

- das ist eine (eingeschränkte) Form der logischen Programmierung (auf Typ-Ebene, zur Compile-Zeit)
- in jedem Inferenz-Schritt wird Code erzeugt (das jeweils passende Wörterbuch wird eingesetzt)

Überblick über Leancheck-Implementierung

- ```
  type Tiers a = [[a]]
  class Listable a where tiers :: Tiers a
  instance Listable Int where ...
  instance Listable a => Listable [a] where ...
  ```
- ```
  data Result
    = Result { args :: [String], res :: Bool }
  class Testable a where
    results :: a -> Tiers Result // orig: resultiers
  instance Testable Bool ...
  instance (Listable a, Show a, Testable b)
    => Testable (a -> b) ...
  ```
- ```
  union :: Tiers a -> Tiers a -> Tiers a //orig: (\/)
  mapT :: (a -> b) -> Tiers a -> Tiers b
  concatT :: Tiers (Tiers a) -> Tiers a
  cons2 :: (Listable a, Listable b)
    => (a -> b -> c) -> Tiers c
  ```
  ```

Hausaufgaben

1. definieren Sie für `data T = T Bool Bool Bool`

   - instance Eq als komponentenweise Gleichheit
   - instance Ord als lexikografische Erweiterung der Standard-Ordnung auf Bool

   Formulieren Sie allgemein (d.h., polymorph) als leancheck-Property, daß die Ordnung `(<=)`

   - transitiv
   - antisymmetrisch

   ist und prüfen Sie das für `Numeric.Natural` und für `T`.

```
transitiv :: Ord a => a -> a -> a -> Bool
check (transitiv @T)
```

2. definieren Sie für `data T = T Bool Bool Bool`

   - instance Semigroup als komponentenweise Disjunktion
   - instance Monoid als dazu passendes neutrales Element

   Formulieren Sie die Monoid-Axiome allgemein (d.h., polymorph) als leacheck-Property und überprüfen Sie diese für `T`.

   Geben Sie eine korrekte Monoid-Struktur für `T` an, die nicht kommutativ ist.

3. für Peano-Zahlen: deklarieren Sie `instance Num N` und implementieren Sie dafür (nur) `(+)` und `(*)`

   (mit den bekannten Definitionen als fold)

   Test: `S (S Z) * S (S (S Z))`

und: `S (S (S Z)) ^ 3` (warum funktioniert das?)

Implementieren Sie `fromInteger`

(rekursiv, mit `if x > 0 then ... else ...`)

Test (polymorphes Literal) `42 :: N`

4. Modul `Data.Map` aus containers, Modul `Data.HashMap` aus unordered-containers:

   - warum hat `fromList` ein Typconstraint (oder sogar zwei) für den Schlüsseltyp `k`, aber nicht für den Werttyp `a`?
   - Erläutern Sie das Typconstraint (oder dessen Fehlen) für `singleton`.

   für Data.Map:

- warum haben die Typen von `unionWith` und `intersectionWith` unterschiedliche Anzahl von Typvariablen?
- Definieren Sie einen Typ `data T = ... deriving Eq` und dafür eine Ord-Instanz, die keine totale Ordnung definiert. Zeigen Sie durch Beispiele, daß dann `M.Map T v` nicht funktioniert. Z.B.: eingefügter Schlüssel nicht in `M.toList`, sichtbarer Schlüssel nicht gefunden.

Verzögerte Auswertung (lazy evaluation)

Motivation: Datenströme

Folge von Daten:

- erzeugen (producer)
- transformieren
- verarbeiten (consumer)

aus softwaretechnischen Gründen diese drei Aspekte im Programmtext trennen,
aus Effizienzgründen in der Ausführung verschränken
(bedarfsgesteuerte Transformation/Erzeugung)

Bedarfs-Auswertung, Beispiele

- Unix: Prozesskopplung durch Pipes

```
cat foo.text | tr ' ' '\n' | wc -l
```

  Betriebssystem (Scheduler) simuliert Nebenläufigkeit
- OO: Iterator-Muster

```
Enumerable.Range(0,10).Select(n=>n*n).Sum()
```

  ersetze Daten durch Unterprogr., die Daten produzieren
- FP: lazy evaluation (verzögerte Auswertung)

```
from n = n : from (n+1)
sum $ take 10 $ map ( \ n -> n * n ) $ from 0
```

  Realisierung: Termersetzung $\Rightarrow$ Graphersetzung,

Beispiel Bedarfsauswertung

```
data Stream a = Cons a (Stream a)
nats :: Stream Nat ; nf :: Nat -> Stream Nat
nats = nf 0 ; nf n = Cons n (nf (n+1))
head (Cons x xs) = x ; tail (Cons x xs) = xs
```

Obwohl `nats` unendlich ist, kann Wert von `head (tail (tail nats))` bestimmt werden:

```
   = head (tail (tail (nf 0)))
   = head (tail (tail (Cons 0 (nf 1))))
   = head (tail (nf 1))
   = head (tail (Cons 1 (nf 2)))
   = head (nf 2) = head (Cons 2 (nf 3)) = 2
```

es wird immer ein *äußerer* Redex reduziert

(Bsp: `nf 3` ist ein *innerer* Redex)

Strictness

zu jedem Typ T betrachte $T_\perp = \{\perp\} \cup T$

dabei ist $\perp$ ein „Nicht-Resultat vom Typ T“

- Exception `undefined :: T`
- oder Nicht-Termination `let {f x=f (f x)} in f 0`

Def.: Funktion f heißt *strikt*, wenn $f(\perp) = \perp$.

Fkt. f mit n Arg. heißt *strikt in* i,

falls $\forall x_1 \ldots x_n : (x_i = \perp) \Rightarrow f(x_1, \ldots, x_n) = \perp$

verzögerte Auswertung eines Arguments

$\Rightarrow$ Funktion ist dort nicht strikt

einfachste Beispiele in Haskell:

- Konstruktoren (Cons,...) sind nicht strikt,
- Destruktoren (head, tail,...) sind strikt.

Beispiele Striktheit

- `length :: [a] -> Int` ist strikt:

  ```
  length undefined   ==>  (exception)
  ```

- `(:) :: a->[a]->[a]` ist nicht strikt im 1. Argument:

  ```
  length (undefined : [2,3]) ==> 3
  ```

  d.h. (`undefined : [2,3]`) ist nicht $\perp$

- `(&&)` ist strikt im 1. Arg, nicht strikt im 2. Arg.

  ```
  undefined && True   ==> (exception)
  False && undefined  ==> False
  ```

Implementierung der verzögerten Auswertung

Begriffe:

- *nicht strikt*: nicht zu früh auswerten
- verzögert (*lazy*): höchstens einmal auswerten
  (ist Spezialfall von *nicht strikt*)

bei jedem Konstruktor- und Funktionsaufruf:

- kehrt *sofort* zurück
- Resultat ist *thunk* (Paar von Funktion und Argument)
- thunk wird erst bei Bedarf ausgewertet
- Bedarf entsteht durch Pattern Matching
- nach Auswertung: thunk durch Resultat überschreiben
  (das ist der Graph-Ersetzungs-Schritt)
- bei weiterem Bedarf: wird Resultat nachgenutzt

Bedarfsauswertung in Scala

- ```
  def F (x : Int) : Int = {
        println ("F", x) ; x*x
  }
  lazy val a = F(3);
  println (a);
  println (a);
  ```

  hier sehen wir, wann die Auswertung stattfindet, weil sie eine Nebenwirkung hat (die Ausgabe).

- in Haskell gibt es keine Nebenwirkungen,
  man kann lazy evaluation nur indirekt feststellen
  - (non)strictness (keine Exception)
  - Ressourcenverbrauch (Speicher, Zeit) (ghci: `:set +s`)
  ```

Diskussion

- John Hughes: *Why Functional Programming Matters*, 1984 `http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html`
- Bob Harper 2011 `http://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/`
- Lennart Augustsson 2011 `http://augustss.blogspot.de/2011/05/more-points-for-lazy-evaluation-in.html`

Anwendg. Lazy Eval.: Ablaufsteuerung

- Nicht-Beispiel (JS hat strikte Auswertung)

```
function wenn (b,x,y){ return b ? x : y; }
function f(x) {return wenn(x<=0,1,x*f(x-1));}
f(3)
```

- in Haskell geht das (direkt in `ghci`)

```
let wenn b x y = if b then x else y
let f x = wenn (x<= 0) 1 (x * f (x-1))
f 3
```

- in JS simulieren (wie sieht dann `f` aus?)

```
function wenn (b,x,y){ return b ? x() : y(); }
```

  anstatt Wert: eine Funktion mit Argument (), die den Wert ausrechnet—aber dann *jedesmal*, ist nicht-strikt, nicht lazy (dazu müßte der Wert gespeichert werden)

Anwendg. Lazy Eval.: Modularität

- ```
  foldr :: (e -> r -> r) -> r -> [e] -> r
  or = foldr (||) False
  or [ False, True, undefined ]
  and = not . or . map not
  ```

- (vgl. Augustson 2011) strikte Rekursionsmuster könnte man kaum sinnvoll benutzen (zusammensetzen)

- übliche Tricks zur nicht-strikten Auswertung zur Ablaufsteuerung
  - eingebaute Syntax (if-then-else)
  - benutzerdefinierte Syntax (macros)

  gehen hier nicht wg. Rekursion
  ```

Anwendg. Lazy Eval.: Streams

unendliche Datenstrukturen

- Modell:

```
data Stream e = Cons e (Stream e)
```

- man benutzt meist den eingebauten Typ

```
data [a] = [] | a : [a]
```

- alle anderen Anwendungen des Typs `[a]` sind *falsch* (z.B. als Arrays, Strings, endliche Mengen)
  mehr dazu: `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`

Primzahlen

```
primes :: [ Nat ]
primes = sieve ( from 2 )

from :: Nat -> [ Nat ]
from n = n : from ( n+1 )

sieve :: [ Nat ] -> [ Nat ]
sieve (x : ys) =
  x : sieve (filter (\y ->0 < mod y x) ys)
```

(Das ist (sinngemäß) das Code-Beispiel auf `https://www.haskell.org/`)

Semantik von `let`-Bindungen

- der Teilausdruck `undefined` wird nicht ausgewertet:

```
let { x = undefined ; y = () } in y
```

- alle Namen sind nach jedem = sichtbar:

```
let { x = y ; y = () } in x
```

- links von = kann beliebiges Muster stehen

```
let { (x, y) = (3, 4) } in x
let { (x, y) = (y, 5) } in x
```

- es muß aber passen, sonst

```
let { Just x = Nothing } in x
```

Beispiel für Lazy Semantik in Let

- Modell: binäre Bäume wie üblich, mit `fold` dazu

```
data T k = L | B (T k) k (T k)
```

- Aufgabe 1: jeder Schlüssel soll durch Summe aller Schlüssel ersetzt werden.

```
f (B (B L 2 L) 3 L) = B (B L 5 L) 5 L
```

```
f t = let s = fold 0 (\ x y z -> x+y+z) t
      in fold L (\ x y z -> Branch x s z) t
```

- Aufgabe 2: dasselbe mit *nur einem fold*. Hinweis:

```
f t = let { (s, r) = fold _ _ t } in r
```

Übungsaufgaben zu Striktheit

- Beispiel 1: untersuche Striktheit der Funktion

```
f :: Bool -> Bool -> Bool
f x y = case y of { False -> x ; True  -> y
```

Antwort:

– f ist nicht strikt im 1. Argument,
denn `f undefined True = True`
– f ist strikt im 2. Argument,
denn dieses Argument (`y`) ist die Diskriminante der obersten Fallunterscheidung.

- Beispiel 2: untersuche Striktheit der Funktion

```
g :: Bool -> Bool -> Bool -> Bool
g x y z =
  case (case y of False -> x ; True -> z) c
    False -> x
    True  -> False
```

Antwort (teilweise)

– ist strikt im 2. Argument, denn die Diskriminante `(case y of ..)` der obersten Fallunterscheidung verlangt eine Auswertung der inneren Diskriminante `y`.

Hausaufgaben

für SS21: Aufgaben 1 bis 3 auf jeden Fall, und eine von 4 und 5

1. Aufgabe: strikt in welchen Argumenten?

```
f x y z = case y || x of
  False -> y
  True -> case z && y  of
    False -> z
    True  -> False
```

Bereiten Sie (wenigstens) eine ähnliche Aufgabe vor, die Sie in der Übung den anderen Teilnehmern stellen.

2. Bestimmen Sie jeweils die ersten Elemente dieser Folgen

(1. auf Papier durch Umformen, 2. mit ghci).

Vorher für die Hilfsfunktionen (`map`, `zipWith`, `concat`):
1. Typ feststellen, 2. Testfälle für endliche Listen

(a)
```
f = 0 : 1 : f
```

(b)
```
n = 0 : map (\ x -> 1 + x) n
```

(c)
```
xs = 1 : map (\ x -> 2 * x) xs
```

(d)
```
ys = False
   : tail (concat (map (\y -> [y,not y]) ys))
```

(e)
```
zs = 0 : 1 : zipWith (+) zs (tail zs)
```

siehe auch `https://www.imn.htwk-leipzig.de/~waldmann/etc/stream/`

3. Für Peano-Zahlen und Eigenbau-Listen implementieren:
`len :: List a -> N` als fold, Vergleich (gt: greater

than, größer als) mit Rekursion:

```
gt :: N -> N -> Bool
gt Z y = _
gt (S x) Z = _
gt (S x) (S y) = gt _ _
```

und diese Auswertung erklären:

```
gt (len (Cons () (Cons () undefined)))
   (len (Cons () Nil))
```

Unterschiede erklären zu

```
length (() : () : undefined) > length (() : undefine
```

4. Folie Ablaufsteuerung: Ifthenelse als Funktion in Haskell, in Javascript, Simulation der nicht-strikten Auswertung.

5. Algorithmus aus Appendix A aus Chris Okasaki: *Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design* (ICFP 2000) implementieren (von `where` auf `let` umschreiben), testen und erklären

Verbrauch von Datenströmen: Fold

Wiederholung, Motivation

- ```
  data List a = Nil | Cons a (List a)
  fold :: res -> (a -> res) -> List a -> res
  fold nil cons Nil = nil
  fold nil cons (Cons x xs)=cons x (fold nil cons xs)
  ```
- mit nicht striktem Argument. Bsp: `(||)` in

  ```
  fold False (||) (Cons True (Cons False undefined))
  ```

  gut: verkürzte Auswertung
- mit striktem Argument, Bsp: `(+)` in

  ```
  fold (0::Natural) (+) (Cons 1 (Cons 2 Nil))
  ```

  schlecht: erst werden alle Thunks gebaut
  ```

Fold über Listen: von rechts, von links

- für den Stream-Datentyp `[a]` = `List a` aus der Standardbibliothek
- unser `fold` heißt `foldr`, „r“ wegen „von rechts“, (Eselsbrücke: „richtig“), aber die Argument-Reihenfolge ist falsch (ist *nicht* die Konstruktor-Reihenfolge)

  foldr $f\ s\ [x_1, x_2, x_3] = f\ x_1\ (f\ x_2\ (f\ x_3\ s))$

  foldr $f\ s\ [x_1, \ldots, x_n] = f\ x_1$ (foldr $f\ s\ [x_2, \ldots, x_n]$)
- ein anderes Rekursionmuster ist `foldl` (von links)

  foldl $f\ s\ [x_1, x_2, x_3] = f(f(f\ s\ x_1)\ x_2)\ x_3)$

  foldl $f\ s\ [x_1, \ldots, x_n] = f$ (foldl $f\ s\ [x_1, \ldots, x_{n-1}])\ x_n$

  speziell für Streams, nicht (wie fold) allgemein für Bäume
- Anwend.: bestimme `f,s` mit `reverse = foldl f s`

Fold-Left: Beispiel

- foldl $f\ s\ [x_1, \ldots, x_n] = f$ (foldl $f\ s\ [x_1, \ldots, x_{n-1}]$) x_n
- Aufgabe: bestimme `f,s` mit `reverse = foldl f s`
- Herleitung der Lösung durch Beispiel

```
[3,2,1] = reverse [1,2,3]
  = foldl f s [1,2,3]
  = f (foldl f s [1,2]) 3
  = f (reverse [1,2]) 3 = f [2,1] 3
```

  also `f [2,1] 3 = [3,2,1]`, d.h., `f x y = y : x`

- Lösung: `reverse = foldl (flip (:)) []`

Fold-Left: Implementierung

- Eigenschaft (vorige Folie) sollte nicht als Implementierung benutzt werden,
  denn $[x_1, \ldots, x_{n-1}]$ ist teuer (erfordert Kopie)

- ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f s xs = case xs of
    []      -> s
    x : xs' -> foldl f (f s x) xs'
  ```

- zum Vergleich

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f s xs = case xs of
  []      -> s
  x : xs' -> f x (foldr f s xs')
```
```

Fold-Left: allgemeiner Typ

- der Typ von `Prelude.foldl` ist tatsächlich

```
Foldable t => (b->a->b) -> b -> t a -> b
```

- hierbei ist `Foldable` eine (Typ)Konstruktor-Klasse
  mit der einzigen (konzeptuell) wesentlichen Methode

```
class Foldable t where toList :: t a -> [a]
```

  und Instanzen für viele generische Container-Typen

- weitere Methoden aus Effizienzgründen

Das strikte Fold-Left: Anwendung

- Motivation war: effizientes `fold` über eine strikte Funktion (ohne Konstruktion von Thunks), aber:

```
:set +s
foldl (+) (0::Int) [0 .. 10^6]
500000500000 -- (0.38 secs)
foldl (+) (0::Int) [0 .. 10^7]
50000005000000 -- (2.18 secs)
```

- die richtige Funktion dafür ist:

```
import qualified Data.Foldable as F
F.foldl' (+) (0::Int) [0 .. 10^7]
50000005000000 -- (0.36 secs)
```

- `ghc -O2`: erzeugt Maschinencode, der nicht allokiert

```
ghc -O2 -rtsopts f.hs ; ./f +RTS -M16k -A8k -s
```

Fold-Left mit nicht-striktem Argument

- `foldl f s` kann die verkürzte Auswertung (Funktion `f` ist nicht strikt im zweiten Argument) nicht ausnutzen:

```
foldl (||) False (replicate (10^7) True)
True -- (1.95 secs, 1,292,364,088 bytes)
```

- `F.fold'` auch nicht (braucht aber weniger Platz)

```
F.foldl' (||) False (replicate (10^7) True)
True -- (0.15 secs, 560,063,800 bytes)
```

- `foldr` kann es (Resultat steht sofort fest)

```
foldr (||) False (replicate (10^7) True)
True -- (0.00 secs, 63,968 bytes)
```

- um passende Variante auszuwählen: Striktheit feststellen
  Alternative: fold vermeiden, Rekursion jedesmal ausprogrammieren. Dann kann man gleich C benutzen.

Transformation von Datenströmen

Bsp: Linq (Language integrated query) in C#

- ```
  IEnumerable<int> stream = from c in cars
      where c.colour == Colour.Red
      select c.wheels;
  ```
- LINQ-Syntax nach Schlüsselwort `from`
  (das steht vorn — „SQL vom Kopf auf die Füße gestellt“)
- wird vom Compiler übersetzt in

```
IEnumerable<int> stream = cars
    .Where (c => c.colour == Colour.Red)
    .Select (c => c.wheels);
```

- auf Deutsch: `map wheels (filter isRed cars)`
  Funktionen 2. Ordnung: Select $=$ map, Where $=$ filter.
```

Kombination von Strömen mit SelectMany

- ```
  from x in Enumerable.Range(0,10)
  from y in Enumerable.Range(0,x)
  select y*y
  ```
- wird vom Compiler übersetzt in

```
Enumerable.Range(0,10)
  .SelectMany(x=>Enumerable.Range(0,x))
  .Select(y=>y*y)
```

- aus diesem Grund ist `SelectMany` wichtig
- das mathematische Modell ist `>>=` (gesprochen: bind)

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f  = concat (map f xs)
```
```

Anwendung von Bind

- ```
  (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= f = concat (map f xs)
  ```

- ```
  [1 .. 10] >>= \ x ->
    [x .. 10] >>= \ y ->
      [y .. 10] >>= \ z ->
        guard (x^2 + y^2 == z^2) >>= \ _ ->
          return (x,y,z)
  ```

- mit diesen Hilfsfunktionen

```
guard :: Bool -> [()]
guard b = case b of False->[]; True->[()]

return :: a -> [a] ; return x = [x]
```

do-Notation

```
[1 .. 10] >>= \ x ->
  [x .. 10] >>= \ y ->
    [y .. 10] >>= \ z ->
      guard (x^2 + y^2 == z^2) >>= \ _ ->
        return (x,y,z)
```

```
do x <- [1 .. 10]
   y <- [x .. 10]
   z <- [y .. 10]
   guard $ x^2 + y^2 == z^2
   return (x,y,z)
```

- https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-470003.14

Wdhlg: der Typkonstruktor `Maybe`

- `data Maybe a = Nothing | Just a`
- modelliert Rechnungen, die fehlschlagen können

```
case ( evaluate l ) of
  Nothing -> Nothing
  Just a  -> case ( evaluate r ) of
    Nothing -> Nothing
    Just b  -> if b == 0 then Nothing
                         else Just ( div a b )
```

- äquivalent (mit passendem `(>>=)`, `return`, `empty`)

```
evaluate l >>= \ a ->
  evaluate r >>= \ b ->
    if b == 0 then empty else return (div a b)
```

- Gemeinsamkeit mit `(>>=)`, `return` auf Listen!

Die Konstruktorklasse Monad

- ```
  class Monad c where  -- Typisierung
      return  :: a -> c a
      ( >>= ) :: c a -> (a -> c b) -> c b
  ```

- ```
  instance Monad [] where  -- Implementierung
      return = \ x -> [x]
      m >>= f = concat ( map f m )
  ```

- ```
  instance Monad Maybe where -- Implem.
    return x = Just x
    m >>= f = case m of
      Nothing -> Nothing ; Just x ->  f x
  ```
  ```

Axiome für Monaden

- `class Monad c` enthält diese Methoden:

  ```
  return :: a -> c a,
  (>>=) :: c a -> (a -> c b) -> c b
  ```

- wobei `(>>=)` „assoziativ“ sein soll

  es hat dafür aber den falschen Typ, deswegen betrachtet man `f >=> g = \ x -> f x >>= g`

- dafür soll gelten
  - `return` ist links und rechts neutral:

    ```
    (return >=> f) = f = (f >=> return)
    ```
  - `(>=>)` ist assoziativ:

    ```
    (f >=> (g >=> h)) = ((f >=> g) >=> h)
    ```

Maybe als Monade

- Ü: die Monaden-Axiome (für `>=>` für `Maybe`) testen (Leancheck), beweisen.
- do-Notation

```
do a <- evaluate l
   b <- evaluate r
   if b == 0 then empty else return (div a b)
```

- `Maybe a` $\approx$ Listen (`[a]`) mit Länge $\in \{0, 1\}$
- Ü: definiert das folgende auch eine Monade für `[]`?

```
xs >>= f  =  take 2 (concat (map f xs))
```

IO: die Schnittstelle zum Betriebssystem

- Ziel: Beschreibung von Interaktionen mit der Außenwelt (Zugriffe auf Hardware, mittels des Betriebssystems)
- `a :: IO r` bedeutet: `a` ist *Aktion* mit Resultat `:: r`
- ```
  readFile :: FilePath -> IO Text
  putStrLn :: Text -> IO ()
  main :: IO ()
  main = T.readFile "foo.bar" >>= \s -> T.putStrLn s
  ```
- Verkettung von Aktionen durch `>>=`
- alterative Oberflächen-Syntax:

  ```
  main = do {s <- T.readFile "foo.bar"; T.putStrLn s}
  ```
- ein Haskell-Hauptprogramm definiert einen Namen `main :: IO ()`, dessen Aktion wird ausgeführt
- das Typsystem trennt streng zw. *Aktion* und *Resultat* Bsp: `f :: Int -> Int` benutzt OS *garantiert* nicht
  ```

Maybe in C# (Nullable)

- `data Maybe a = Nothing | Just a`
- C#: Typkonstruktor `Nullable<T>`, abgekürzt `T?`
  Argument muß Wert-Typ sein (primitiv oder struct)
- Verweistypen sind automatisch Nullable
  (das ist der *billion dollar mistake*, Tony Hoare 1965)

  `https://www.infoq.com/presentations/`

  `Null-References-The-Billion-Dollar-Mistake-Tony-Hoa`

- Methoden:

```
Nullable<int> x = null; x.HasValue
int? y = 7    ; y.HasValue ; y.Value
```

- überladene Typen einiger Operatoren `int? a = x + y`
- Operator `??`, Bsp. `int b = x + y ?? 8`

Nullable als Monade

- wie heißen Return und Bind?
  - Return: Konstruktor `new Nullable<int>(9)`
  - Bind: realisiert durch Operator `?.`

- Anwendung (Testfall)

```
class C {
  readonly int x; public C(int x){this.x=x;}
  public C f(){return this.x>0 ? new C(x-1) : null;
}

new C(1) .f() .f() .f()
new C(1)?.f()?.f()?.f()
```

Übungen zu Monaden

1. (wurde in der VL vorgeführt) Testen der Monaden-Axiome für `Maybe` in `ghci`:

```
import Test.LeanCheck
import Test.LeanCheck.Function

f >=> g = \ x -> f x >>= g

check $ \ f x ->
    ((f :: Bool -> Maybe Int) >=> return) x
```

beachten Sie

- Testdatenerzeugung für Funktionen (hier: `f`) erfordert

```
import Test.LeanCheck.Function
```

- Typ-Annotation ist notwendig, um Belegung aller Typvariablen zu fixieren
- Vergleich von Funktionen ist nicht möglich (es gibt keine `instance Eq (a -> b)`),
  deswegen zusätzliches Argument `x` und Vergleich der Funktionswerte
  (alternativ: `https://hackage.haskell.org/package/leancheck-0.9.1/docs/Test-LeanCheck-Function-Eq.html`)

Hausaufgaben

SS 21: Aufgaben 1, 2, 4, 5.

1. In einem Kommentar in GHC.List `https://hackage.haskell.org/package/base-4.15.0.0/docs/src/GHC-List.html#filter` stellt SLPJ (wer ist das?) fest, daß man in der Gleichung

```
filter p (filter q xs)
  = filter (\ x -> q x && p x) xs
```

   die zur Programmtransformation während der Kompilation verwendet wird, rechts nicht `p x && q x` schreiben darf.

   Warum—die Konjunktion ist doch kommutativ?

   Hinweis: auf `Bool` ja, auf $\texttt{Bool}_\perp$ nicht.

2. (Funktionen aus Data.List)
   - implementieren Sie `scanr` mittels `foldr`.
     ```
     scanr f z = foldr (\x (y:ys) -> _) _
     ```
   - implementieren Sie `scanr` mittels `mapAccumR`.
     ```
     scanr f z = uncurry (:) . mapAccumR _ _
     ```
   - ergänzen Sie die allgemeingültige Aussage
     ```
     scanr f z (reverse xs) = _ (scanl _ _ xs)
     ```
3. für `instance Monad Maybe`: zeigen Sie, daß `(>=>)` nicht kommutativ ist.

   Geben Sie einen statisch korrekten Testfall an, für den der kommutierte (vertauschte) Ausdruck statisch falsch (typfehlerhaft) ist.

   Geben Sie Testfälle an, bei denen die Kommutation statisch korrekt ist, aber die dynamische Semantik (die

berechneten Werte) nicht übereinstimmen.

Verwenden Sie dafür Leancheck.

4. zur Listen-Monade:
   Erfüllt die folgende Instanz die Monaden-Axiome?

```
return x = [x]
xs >>= f = take 2 (concat (map f xs))
```

   Hinweis: `import Prelude hiding ((>>=))`, dann `(>>=)` wie hier. `return` wird unverändert importiert.

   Definieren Sie den Operator
   `f >=> g = \ xs -> f xs >>= g`, testen Sie die Axiome mittels Leancheck.

5. Maybe in C# und Java:
   - `Nullable<T>` in C# Ist `??` tatsächlich assoziativ?

Vergleichen Sie die Bedeutung von `(a ?? b) ?? c` und `a ?? (b ?? c)` unter diesen Aspekten:

- statische Semantik: Typisierung
- dynamische Semantik: einschließlich $\perp$ (Exceptions)

- `Optional<T>` in Java:
  - welche Methoden realisieren bind und return?
  - was entspricht dem Operator `??` aus C#?

Primär-Quellen (Dokumentation) angeben, Rechnungen in `csharp` (mono) bzw. `jshell` (jdk) vorführen.

6. (mglw. autotool—CYP) beweisen Sie diese Beziehung zwischen foldl und foldr

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z0 xs =
  let f' x k z = k (f z x)
```

```
    in  foldr f' id xs z0
```

7. (später) definieren Sie eine Monad-Instanz jeweils für
   - binäre Bäume mit Schlüsseln nur in den Blättern
   - . . . nur in den Verzweigungsknoten
   - . . . in allen Knoten
   - beliebig verzweigende Bäume *rose trees*

     ```
     data Tree a = Node a [Tree a]
     ```

   überprüfen Sie ein Einhaltung der Axiome.

Unveränderliche Daten

Überblick

- alle Attribute aller Objekte sind unveränderlich (`final`)
- anstatt Objekt zu ändern, konstruiert man ein neues
- vereinfacht Formulierung und Beweis von Programm-Eigenschaften

  (wenn der Konstruktor korrekt ist, ist alles korrekt)
- parallelisierbar (keine updates, keine *data races*)

  `http://fpcomplete.com/the-downfall-of-imperative-programming/`
- Persistenz (Verfügbarkeit früherer Versionen)
- Belastung des Garbage Collectors (. . . dafür ist er da)

Beispiel: Einfügen in Baum

- destruktiv (Java)

```
interface Tree<K> { void insert (K key); }
Tree<String> t = ... ;
t.insert ("foo");
```

- persistent (Java):

```
interface Tree<K> { Tree<K> insert (K key);
Tree<String> t = ... ;
Tree<String> u = t.insert ("foo");
```

- persistent (Haskell):

```
insert :: k -> Tree k -> Tree k
```

Beispiel: unbalancierter Suchbaum

- `data Tree k = -- Datentyp`
  `Leaf | Branch (Tree k) k (Tree k)`

- Invariante: $\forall$ `t` $\in$ `Tree k . monotone (inorder t)`

- ```
  insert :: Ord k => k -> Tree k -> Tree k
  insert x t = case t of
    Leaf -> Branch Leaf x Leaf
    Branch l k r -> case compare x k of ...
  ```

- wie teuer ist die Persistenz?

  (wieviele neue Objekte entstehen bei `insert`,`delete`?)
  ```

Löschen in unbalancierten Suchbäumen

- um die Wurzel eines (Teil)baumes t zu löschen:
  der inorder-Nachfolger wird aus rechtem Kind extrahiert
  und wird neue Wurzel

- ```
  delete      :: Ord k => k -> Tree k -> Tree k
  delete x t = case t of
    Leaf -> Leaf
    Branch l k r -> case compare x k of
      LT -> _ ; GT -> _ ;
      EQ -> case minView r of
        Nothing -> _
        Just (m, r') -> _
  ```

- ```
  minView :: Ord k => Tree k -> Maybe (k, Tree k)
  ```

Größen-balancierter Suchbaum

- $|t|$: Anzahl der Blätter, $H(t)$: Höhe
- Invariante: Suchbaum und α-Balance (Bsp: $\alpha = 1/4$)

  `∀ t = Branch l k r:` $\alpha|t| \leq |l| \leq (1-\alpha)|t|$
- für jeden α-balancierten Baum t gilt: $H(t) \leq \log_{1/(1-\alpha)} |t|$.
  Bsp: $\alpha = 1/4, |t| = 10^9$
- Guy Blelloch et al.: *Just Join for Parallel Sets*, 2016
  `https://arxiv.org/abs/1602.02120v3`

  Implementierung von insert, delete, union, intersection durch *smart constructor* für bereits separierte Argumente

```
branch :: Ord k => Tree k -> k -> Tree k -> Tree k
```

Persistente Objekte in Git

- *Distributed* development (Linus Torvalds 2005)
- Strong support for *non-linear* development.
  (Branching and merging are fast and easy.)
- Efficient handling of *large* projects.
  (z. B. Linux-Kernel, `https://kernel.org/`)
- Cryptographic authentication of history.
  (d.h., block chain; die Idee ist überhaupt nicht neu)
- nicht verwechseln: Git (File-Format, Client) mit
  - Git[hu/la]b (Web-Gui für zentrales Repo und Issues),
  - Git[hu/la]b.Com (Vermietung von Speicher und Server, Anhäufung und Auswertung von Benutzerdaten)

Objekt-Versionierung in Git

- Objekt-Typen:
  - Datei (blob),
  - Verzeichnis (tree), mit Verweisen auf blobs und trees
  - Commit, mit Verweisen auf tree u. commits (Vorgänger)

  `git cat-file -p <hash>`
- Objekte sind *unveränderlich* und durch SHA1-Hash (160 bit $=$ 40 Hex-Zeichen) identifiziert
- statt Überschreiben: neue Objekte anlegen
- jeder Zustand ist durch Commit-Hash (weltweit) eindeutig beschrieben und kann (aus Repo) rekonstruiert werden
- deswegen Vorsicht, niemals Geheimnisse im Repo
- Zeitreisen mit `git bisect`

Theorems for Free: Beispiel

- welche F.n haben Typ `f :: forall a . a -> a` ?
  nur eine: `f = \x -> x`.
- welche F.n haben Typ `g :: forall a . a -> [a]` ?
  viele. . . aber für jedes solche gilt:
  für jedes `h :: a -> b, x :: a`
  gilt `g (h x) = map h (g x)`
- diese Eigenschaft folgt allein aus dem Typ von `g`
  d.h., ist unabhängig von Implementierung von `g`
- gilt nicht für Programme in imperative Sprachen mit veränderlichen Daten
  wir haben schon Eigensch. fktl. Programme bewiesen
  Besonderheit jetzt: Beweis ohne Ansehen der Implement.

Theorems for Free: Quelle

- Phil Wadler: *Theorems for Free*, FCPA 1989
  `https://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html`

  From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

- John C. Reynolds (1935–2013)
  `https://www.cs.cmu.edu/~jcr/`

  (Lesetipp: *Some Thoughts on Teaching Programming and Programming Languages*, PLC 2008)

Struktur-erhaltende Transformationen

- (Wdhlg) strukturerhaltende Stream-Transformation:

```
map :: (a -> b) -> [a] -> [b]
```

- Verallgemeinerung: strukturerhaltende Transformation:

```
class Functor c where
  fmap :: (a -> b) -> c a -> c b
```

- gewünschte Eigenschaften (Axiome):

```
fmap id = id; fmap (f.g) = fmap f . fmap g
```

- Standardbibl.: Instanzen für: `[]`, `Maybe`, ...
- für jeden algebraischen Datentyp kann `fmap` durch `fold` implementiert werden, Bsp (`List`):

```
fmap f = fold Nil (\x y -> Cons (f x) y)
```

Theorems for Free: Systematik, Beispiele

- für jede Funktion $g :: \forall a.C_1 a \to \ldots C_n a \to C a$
  (C_i, C sind einstellige Typkonstruktoren mit Functor-Instanzen) gilt
  $$g(\mathrm{fmap}_{C_1}\ h\ x_1) \ldots (\mathrm{fmap}_{C_n}\ h\ x_n) = \mathrm{fmap}_C\, h\ (g\ x_1 \ldots x_n)$$
- ```
  maybeToList :: Maybe a -> [a]
  maybeToList (fmap h x) = fmap h (maybeToList x)

  length :: [a] -> Nat
  length (fmap h xs) = length xs
  ```
- wie lauten die „Umsonst-Sätze“ für
  - `(:) :: a -> [a] -> [a]`
  - `take :: Int -> [a] -> [a]`
  ```

Freie Theoreme für Funktions-Typen

- Bsp: `filter :: (a->Bool) -> [a] -> [a]`

- ```
  filter (>30) (fmap (*10) [1,5,2,4,3])
  = filter (>30) [10,50,20,40,30]   = [50, 40]
  = fmap (*10) [5,4]
  = fmap (*10) (filter ((>30).(*10)) [1,5,2,4,3])
  ```

- ```
                (filter p          (fmap f xs))
    = fmap f (filter (p . f)              xs )
  ```

- für Typ $t = (i_1 \to \cdots \to i_n \to o)$:

  Polarität; $P(o) = P(t)$, $P(i_1) = \cdots = P(i_n) = \neg P(t)$

  für $P(t)$ positiv: transformiere rechts, negativ: links

Hausaufgaben

SS21: Aufgaben 1, 2, 4, 5.

1. für α-balancierte Suchbäume
   - Geben Sie einen $1/4$-balancierten Baum mit (z.B.) 10 Schlüsseln an, der möglichst hoch ist. In jedem `Branch l k r` soll $|l| \leq |r|$ gelten.
     Vergleichen Sie die Höhe mit der in der VL berechneten Schranke.
   - Vergleichen Sie die Implementierung von `split` und (z.B.) `union` (Fig. 1 des zitierten Papers) mit der Implementierung der entsprechenden Funktion aus `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Set.html`
   - für `Data.Set`: erläutern Sie Laufzeit-Unterschiede

zwischen `fromList` und `fromAscList`

2. Für *rose trees* (Rhododendron)

```
data Tree k = Node k [Tree k]
```

- Implementieren Sie das Rekursionsschema
  `fold :: (k -> [res] -> res) -> Tree k -> r`
  (dieser Typ war ursprünglich fehlerhaft angegeben, wurde repariert Mon 28 Jun 2021 11:46:24 CEST)
- Bestimmen Sie mit diesem `fold` die Anzahl der Knoten, die Höhe, die Summe der Schlüssel.
- Konstruieren Sie Binomialbäume:
  B_n hat Schlüssel n und Kinder $[B_0, \ldots, B_{n-1}]$
  insbesondere: B_0 hat Schlüssel 0 und keine Kinder.
- berechnen Sie die o.g. Parameter für kleine B_n, geben

Sie allgmeine Aussagen (für alle B_n an), beweisen Sie diese.

3. `git bisect` erläutern und vorführen. Dazu ein überschaubares kleines Projekt verwenden (oder anlegen) (ca. 10 commits) mit einfachem Testfall sowie einem einfachen Fehler irgendwo.

   In diesem Zusammenhang Verhaltensregeln/Empfehlungen diskutieren (master always buildable, squash commits?)

4. `instance Functor Tree`

   - implementieren Sie die Functor-Instanz für binäre Bäume mit Schlüsseln in inneren Knoten (1. mit Rekursion, 2. mit fold)

- überprüfen Sie die Axiome für `fmap`
  Benutzen Sie Aufzählung von Funktionen in Leancheck
- geben Sie das freie Theorem für den Typ
  1. `Tree k -> [k]`, 2. `Tree k -> Nat`
  an und überprüfen Sie an Beispielen

5. Freie Theoreme gibt es nur für uneingeschränkt polymorphe Typen. Für Typen mit Constraints

   Bsp. `Data.List.sort :: Ord k => [k] -> [k]`

   betrachtet man stattdessen die Funktion, bei der das Typconstraint durch das Wörterbuch-Argument (hier: mit der einzigen Funktion `compare`) ersetzt wird.

   Für dieses Beispiel ist das `Data.List.sortBy`. Geben Sie für deren Typ das freie Theorem an, überprüfen Sie.

(nächste Teilaufgabe hat nichts mit freien Theoremen zu tun. Beachten Sie: wer außerhalb einer Übungsaufgabe sortiert, macht einen Fehler, hätte stattdessen Data.Set benutzen sollen. Wer einfach verkettete Listen benutzt, auch ($\Rightarrow$ Data.Sequence, Data.Vector). Wer Strings benutzt, auch ($\Rightarrow$ Data.Text))

Wenn man bezüglich einer bestimmten Maßfunktion sortieren möchte, z.B. Strings nach der Länge, dann kann man verwenden

```
import Data.Function (on)
sortBy (compare `on` length) ["fooo", "bar"
```

aber auch

```
sortOn length ["fooo", "bar", ""]
```

Erklären Sie `'on'`. Erklären Sie den Unterschied (Laufzeit, Platz) zwischen beiden Varianten.

Zusammenfassung, Ausblick

Zusammenfassung, Ausblick

Prüfungen

- in SS 21 als PH-D (Hausarbeit digital): Bearbeitung von Aufgaben in autotool,
- Bewertung im Prüfungsmodus:
  - teilw. versteckte Antworten (Bewertung wird erst nach Prüfungsende sichtbar)
  - teilw. Beschränkung der Anzahl der Einsendungen
  - mglw. Abwertung bei mehreren Einsendungen (Syntax-Fehler: 1 %, Semantik-Fehler: 10 %)
- Technik: `https://gitlab.imn.htwk-leipzig.de/autotool/all0/-/tree/master/doc/pm`

Hausaufgaben für Übung KW 27

- bereits festgelegte Aufgaben
  - 25 A 3 (git)
  - 25 A 5 (sortBy, sortOn)

- noch ausstehende Aufgaben (wahlweise)
  - 24 A 3 (Eigensch. von `>=>` für instance Monad Maybe)
  - 23 A 4 (Simulation der nicht strikten Auswertung in JS)

- Zusatz (auch später)
  - Big Endian Patricia Trees (`https://hackage.haskell.org/package/containers-0.6.5.1/docs/Data-IntSet.html`) Datenstruktur (kurz!) erklären; diesen Fehler vorführen,

erklären (und reparieren) https://github.com/haskell/containers/issues/783

```
$ cabal repl --constraint 'containers == 0.6.2.1'
> Data.IntSet.fromList [255,256]
    > Data.IntSet.singleton 254
True

$ cabal repl --constraint 'containers == 0.6.4.1'
> Data.IntSet.fromList [255,256]
    > Data.IntSet.singleton 254
False
```

Implementierung von

```
instance Ord IntSet where compare s1 s2 =
```

* Version 0.6.3.1 und 0.6.4.1: `case relate ...`
* Version davor und danach:

```
  compare (toAscList s1) (toAscList s2)
```

Erläuterung/Wiederholung: Currying

- ist die Darstellung

  einer zweistelligen Funktion $f \in (A \times B) \to C$

  als einstellige Funktion (2. Ord.) $g \in A \to (B \to C)$

  Darstellungen sind äquivalent wg. $f(x, y) = g(x)(y)$

- benannt nach Haskell B. Curry (1900–1982)
  `https://mathshistory.st-andrews.ac.uk/Biographies/Curry/`

- Lambda-Kalkül und Haskell: jede Funktion ist einstellig,

  mit Abkürzungen ($g = \lambda xy.M$, $g :: A \to B \to C$)

Erläuterung/Wiederholung: Extensionalität

- zwei Funktionen sind (semantisch) gleich, wenn sie für gleiche Argumente gleiche Werte berechnen
- als Beweisprinzip in Cyp:

```
map :: (a -> b) -> List a -> List b
id  :: a -> a ; id x = x
Lemma: map id .=. id
Proof by extensionality with xs :: List a
Show : map id xs .=. id xs
Proof by induction on xs :: List a ...
```

- Anwendung: $(\lambda x.fx) = f$, denn $(\lambda x.fx)y \rightarrow_\beta fy$
- Anwendung (zweimal)

```
plus x y = fold x (\ a -> S a) y
plus x   = fold x S
```

Wiederholung/Bsp: generische Polymorphie

- Anwendung des Punkt-Operators:

```
succ 7              ==> 8
(succ . succ) 7     ==> 9
```

- der Typ von `(.)` ist generisch polymorph

```
(.) :: forall a b c . (b->c) -> (a->b) -> a -> c
```

- aus dem Typ kann man eine Implementierung ableiten

```
f (x :: b -> c) (y :: a -> b) (z :: a) = _ :: c
```

  es gibt im wesentlichen nur diese eine
  (man könnte sinnlose `id` einfügen, mit $\texttt{id} = \lambda x.x$)

- vgl. auch: aus einem Typ das freie Theorem ableiten

Zusammenfassung: Themen

- Terme, algebraische Datentypen
- Muster, Regeln, Term-Ersetzung (Progr. 1. Ordnung)
- Beweisverf.: Umformung, Fallunterscheidung, Induktion
- Polymorphie, Typvariablen, Typkonstruktoren
- Funktionen als Daten, λ-Kalkül (Progr. höherer Ord.)
- Beweisverfahren: Extensionalität
- Rekursionsmuster (fold)
- Eingeschränkte Polymorphie (Typklassen)
  Beispiele: Eq, Ord, Show, Listable (Testdatenerzeugung)
- Striktheit, Bedarfsauswertung, Streams
- Stream-Verarbeit.: foldl, map, filter, bind; freie Theoreme
- Algorithmen für persistente Daten

Aussagen

- statische Typisierung $\Rightarrow$
  - findet Fehler zur Entwicklungszeit (statt Laufzeit)
  - effizienter Code (keine Laufzeittypprüfungen)
- generische Polymorphie: flexibler *und* sicherer Code
- Funktionen als Daten, F. höherer Ordnung $\Rightarrow$
  - ausdrucksstarker, modularer, flexibler Code

Programmierer(in) sollte

- die abstrakten Konzepte kennen
- sowie ihre Realisierung (oder Simulation) in konkreten Sprachen (er)kennen und anwenden.

Zur objektorientierten Programmierung

- tatsächlich meist: klassen-oriertierten Programmierung
- nützlich ist allein: die Trennung zwischen Schnittstelle (Signatur + Axiome) und Implementierung (Algebra)
- andere Eigenschaften richten seit ca. 1980 unglaublich viel Schaden an (in Praxis und in Lehre)
  - Zustandsänderungen: Code ist nicht parallelisierbar
  - Implementierungs-Vererbung: nicht modular
  - die umständliche Simulation von Funktionen als Daten (Befehls-Objekte, funktionale Interfaces)
- bessere Lösungen sind bekannt (1936: Lambda-Kalkül, 1973: generische Polymorphie), nur zögerlich verwendet

Ausdrucksstarke Typen

- das softwaretechnische Ziel statischer Typisierung ist:
  - vollständige Spezifikation $=$ Typ
  - Implementierung erfüllt Spezifikation
    $\Longleftrightarrow$ Implementierung ist statisch korrekt
    und das wird durch den Compiler überprüft

- Beispiele:
  - nicht validieren (`:: String -> Bool`),
    sondern parsen (`:: String -> Maybe T`)
  - Unterscheidung zw. Aktion (`readFile "f" :: IO Text`)
    und ihrem Resultat (`t :: Text`)

Ausdrucksstärkere Typen: Balance

- bekannt ist: generische Container-Typen (Element-Typ als Typ-Parameter): ergibt flexiblen (d.h., nachnutzbaren) *und* sicheren Code.

- der Typ kann auch die Form (z.B.: Balance) eines Baumes beschreiben! — Beispiel:

```
data Tree k = Single k | Deep (Tree (k,k))
```

- Anwendung: containers:Data.Sequence

```
newtype Seq a = Seq (FingerTree (Elem a))
data FingerTree a = EmptyT | Single a
    | Deep Int (Digit a) (FingerTree (Node a)) (Dig
data Node a = Node2 Int a a | Node3 Int a a a
```

Daten-abhängige Typen (dependent types)

- Wiederholung: bisher zwei Arten von Funktionen:
  - von Datum nach Datum (z.B.: Nachfolger, Spiegelbild)
  - von Typ nach Typ (Typkonstruktor, z.B.: List, Tree)
- nützlich ist auch:
  - v. Datum nach Typ (B.: Zahl $\rightarrow$ Vektoren dieser Länge)
- `{-# language DataKinds,KindSignatures,GADTs #-}`

```
data N = Z | S N
data Vector (l :: N) e where
  Nil  :: Vector Z e
  Cons :: e -> Vector l e -> Vector (S l) e
```

  Anwendung: `head :: Vector (S l) e -> e` ist total
- Agda (Ulf Norell 2007, Catarina Coquand 1999)
  `http://wiki.portal.chalmers.se/agda`