Fortgeschrittene Programmierung Vorlesung WS 09,10; SS 12–14, 16–19, 21–23

Johannes Waldmann, HTWK Leipzig

7. Juli 2023

1 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]->[Bool]))
```

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) z
    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().WithDegreeOfParallelism(4)
                                  .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 *Grundl. Künstl. Intell.*
- Constraint-Programmierung: als Wahlfach (WS 23)

Beziehungen zu weiteren LV: Voraussetzungen

- Bäume, Terme (Modellierung, Alg.+DS)
- 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
  (Anwendung: automatische Testdatenerzeugung)
- Bedarfsauswertung, unendl. Datenstrukturen
- Konstruktorklassen (Functor, Applicative, Monad)
- Collections (endliche Mengen, Abbildungen, Folgen)
  als Anwendung vorher gezeigter Konzepte

Anwendungen dieser Konzepte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme
  Scale: case class, Java: Entwurfsmuster Kompositum,
  immutable objects (`record`), 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
- Bedarfsauswertung, unendl. Datenstrukturen

  Iteratoren, Ströme, LINQ
- Functor, Applicative, Monad: `map`, `flatMap`

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 `https://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
- How I Teach Functional Programming (WFLP 2017) `https://www.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: markierte Aufgaben aus dem Skript: 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: Klausur 120 min, keine Hilfsmittel

Übungen

- Informationen zur VL: `https://www.imn.htwk-leipzig.de/~waldmann/lehre.html`

- digitale Selbstverteidigung: Browser und Suchmaschine datenschutzgerecht auswählen und einstellen.

  Das Geschäftsmodell der Überwachungswirtschaft ist es, Ihren Bildschirmplatz, und damit Ihre Aufmerksamkeit und Ihre Lebenszeit an Anzeigenkunden zu verkaufen. Um dabei höhere Erlöse zu erzielen, wird Ihr Verhalten vermessen, gespeichert, vorhergesagt und beeinflußt. Die dazu angelegten Personenprofile erlauben eine umfassende privatwirtschaftliche und staatliche Überwachung. Diese soll verschleiert, verharmlost und legalisiert werden.

  Siehe auch

  - OS Überwachungskapitalismus `https://www.imn.htwk-leipzig.de/~waldmann/talk/19/ubkap/`,
  - VL Informatik (Nebenfach) `https://www.imn.htwk-leipzig.de/~waldmann/edu/ws21/inf/folien/#(11)`

- Benutzung Rechnerpool (ssh, tmux, ghci) `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`

- Beispiel Funktionale Programmierung

```
$ /usr/local/waldmann/opt/ghc/latest/bin/ghci

ghci> length $ takeWhile (== '0') $ reverse $ show $ foldr (*) 1 [1 .
```

  - Typ und Wert von Teilausdrücken feststellen, z.B.

```
ghci> :set +t
ghci> foldr (*) 1 [1..100 :: Integer]
```

  - 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 `https://xkcd.com/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 (allgemeines)

- benutzen Sie gitlab.imn zur Koordinierung: (einmalig) Einteilung in Dreiergruppen, (wöchentlich) Bearbeitung der Aufgaben. Benutzen Sie Wiki und Issues mit sinnvollen Titeln/Labeln. Schließen Sie erledigte Issues.
- Jede der markierten Aufgabe kann in jeder Übung aufgerufen werden (Bsp: Aufg. 3 in den INB-Übungen und in der MIB-Übung) Es kann dann eine vorher gemeinsam (von mehreren Gruppen) vorbereitete Lösung präsentiert werden—die aber von jedem einzelnen Präsentator auch verstanden sein sollte.
- Auch die nicht markierten Aufgaben können in den Übungen diskutiert werden—wenn dafür Zeit ist.

Aufgaben

SS 23: Aufgabe 1

1. Digitale Selbstverteidigung

(a) Welche Daten gibt Ihr Browser preis?

Starten Sie in einer Konsole den Befehl `ncat --listen --source-port 9999`
(Konsole soll sichtbar bleiben)

Rufen Sie im Browser die Adresse `http://localhost:9999` auf, beobachten Sie die Ausgabe in der Konsole.

Wie (personen)spezifisch ist diese Information?

(b) Wie können weitere Informationen extrahiert werden? Verwenden Sie `https://www.eff.org/press/releases/test-your-online-privacy-protection-` (Electronic Frontier Foundation, 2015–)

(c) Stellen Sie Firefox datenschutzgerecht ein. (Das beginnt mit der Default-Startseite!)

Zeigen Sie die Benutzung von temporary containers, von Profilen (z.B. ein Profil für Browsing im Screen-Share).

Führen Sie Browser-Plugins `uMatrix`, `uBlockOrigin` vor.

2. 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?

3. 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?

4. 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*.

5. 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 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)

6. 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).

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

  $Z(), \quad S(S(Z())), \quad f(S(S(Z())), Z())$
- Abkürzung: das leere Argument-Tupel (die Klammern) nach nullstelligen Symbolen weglassen, $f(S(S(Z)), Z)$
- Signatur: $\Gamma = \{E/0, A/1, B/1\}$
- Elemente von $\mathrm{Term}(\Gamma)$: ...

- Bezeichnung: für Signatur Σ und $k \in$:

  Σ_k bezeichnet Menge der Symbole aus Σ mit Stelligkeit k

  $\Sigma_0 = \{Z\}, \Sigma_1 = \{S\}, \Sigma_2 = \{f\},$

  $\Gamma_0 = \{E\}, \Gamma_1 = \ldots, \Gamma_2 = \ldots$

Abmessungen von Termen

- die Größe: ist Funktion $|\cdot| : \mathrm{Term}(\Sigma) \to$ mit
  - für $f \in \Sigma_k$ gilt $|f(t_1, \ldots, t_k)| = 1 + |t_1| + \ldots + |t_k|$

  die Größe eines Terms ist der Nachfolger der Summe der Größen seiner Kinder
- Bsp: $|S(S(Z()))| = 1 + |S(Z())| = 1 + 1 + |Z()| = 1 + 1 + 1$
  $|f(S(S(Z())), Z()| = \ldots$
- die Höhe: ist Funktion $\mathsf{height} : \mathrm{Term}(\Sigma) \to$:

  für $t = f(t_1, \ldots, t_k)$ gilt
  - wenn $k = 0$, dann $\mathsf{height}(t) = 0$
  - wenn $k > 0$, dann $\mathsf{height}(t) = 1 + \max(\mathsf{height}(t_1), \ldots, \mathsf{height}(t_k))$

Induktion über Termaufbau (Beispiel)

- Satz: $\forall t \in \mathrm{Term}(\{a/0, b/2\}) : |t| \equiv 1 \pmod 2$ (die Größe ist ungerade)
- Beweis durch Induktion über den Termaufbau:
  - IA (Induktions-Anfang): $t = a()$
    Beweis für IA: $|t| = |f()| = 1 \equiv 1 \pmod 2$
  - IS (I-Schritt): $t = b(t_1, t_2)$
    zu zeigen ist: IB (I-Behauptung): $|t| \equiv 1 \pmod 2$
    dabei benutzen: IV (I-Voraussetzung) $|t_1| \equiv |t_2| \equiv 1 \pmod 2$
    Beweis für IS: $|t| = |b(t_1, t_2)| = 1 + |t_1| + |t_2| \equiv 1 + 1 + 1 \equiv 1 \pmod 2$
- Bezeichnung: das heißt IV, und nicht I-Annahme, damit es nicht mit I-Anfang verwechselt wird

Algebraische Datentypen (benannte Notation)

- Beispiel: Deklaration des Typs

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

- Bezeichnungen:
  - `Foo` ist Typname
  - `Con` ist Konstruktor
  - `bar, baz` sind Komponenten-Namen des Konstruktors
  - `Int, String` sind Komponenten-Typen
- Beispiel: Konstruktion eines Datums dieses Typs

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

  der Ausdruck (vor dem `::`) hat den Typ `Foo`

Algebraische Datentypen (positionelle Not.)

- Beispiel: Deklaration des Typs

```
data Foo = Con Int String
```

- Bezeichnungen:
  - `Foo` ist Typname
  - `Con` ist zweistelliger Konstruktor
    ... mit anonymen Komponenten
  - `Int`, `String` sind Komponenten-Typen
- Beispiel: Konstruktion eines Datums dieses Typs

```
Con 3 "hal" :: Foo
```

- auch ein Konstruktor mit benannten Komponenten kann positionell aufgerufen werden

Datentyp mit mehreren Konstruktoren

- Beispiel (selbst definiert)

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

- Bespiele (in Standardbibliothek (Prelude) vordefiniert)

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

- Konstruktion solcher Daten:

```
False :: Bool
A { foo = False } :: T ; A False :: T
B EQ True :: T
```

Mehrsortige Signaturen

- (bisher) einsortige Signatur

  ist Abbildung von Funktionssymbol nach Stelligkeit

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

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

- $\mathrm{Term}(\Sigma, B)$ (Terme dieser Signatur mit Sorte B): ...

Rekursive Datentypen

- Konstruktoren mit benannten Komponenten

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

- mit anonymen Komponenten

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

- Objekte dieses Typs erzeugen, Bsp:

```
Leaf :: Tree; Branch (Branch Leaf Leaf) Leaf :: Tree
```

- Bezeichnung `data Tree = ... | Node ...` ist falsch (irreführend), denn sowohl äußere Knoten (Leaf) als auch innere Knoten (Branch) *sind* Knoten (Node)
- Ü: die data-Dekl. für $S = \{Z, B\}$, $\Sigma = \{0 \mapsto ([], Z),\ p \mapsto ([Z, Z], Z),\ e \mapsto ([Z, Z], B), a \mapsto ([B, B], B)\}$.

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 `https://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.
- Bestimmen Sie $|\sqrt{a \cdot a + b \cdot b}|$, $\mathsf{height}(\sqrt{a \cdot a + b \cdot b})$.

- Geben Sie ein Element $t \in \mathrm{Term}(\{f/1, g/3, c/0\})$ an mit $|t| = 5$ und $\mathsf{height}(t) \leq 2$.
- die Menge $\mathrm{Term}(\{f/1, g/3, c/0\})$ wird realisiert durch den Datentyp `data T = F T | G T T T` deklarieren Sie den Typ in ghci, erzeugen Sie o.g. Term t (durch Konstruktoraufrufe)
- Holes (Löcher) in Ausdrücken als Hilfsmittel bei der Programmierung durch schrittweises Verfeinern

```
ghci> data T = A Bool | B T deriving Show
ghci> A _

<interactive>:2:3: error:
    • Found hole: _ :: Bool
    • In the first argument of 'A', namely '_'
      In the expression: A _
      In an equation for 'it': it = A _
    • Relevant bindings include ...
      Valid hole fits include ...
        False :: Bool
        True :: Bool
        ...
```

Hausaufgaben

SS 23: Aufgaben 2, 4, 5.

1. autotool-Aufgabe 14-1

   Allgemeine Hinweise zur Bearbeitung von Haskell-Lückentext-Aufgaben in autotool:

   - 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 T (eine `data`-Deklaration) an, der alle Terme der einsortigen Signatur $\Sigma = \{E/0, F/2, G/3\}$ enthält.

   Konstruieren Sie Elemente dieses Typs.

   Geben Sie $t \in \mathrm{Term}(\Sigma)$ an mit

   - $\mathsf{height}(t) = 2$ und $|t|$ möglichst klein
   - $\mathsf{height}(t) = 2$ und $|t|$ möglichst groß

Allgemeine Hinweise zu Arbeit und Präsentation im Pool:

- beachten Sie `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/` (`PATH` und ggf. `LD_LIBRARY_PATH`)
- Freigabe (für Dozenten-Rechner) mit `krfb`, Einmalpaßwort
- Schrift schwarz auf weiß! Vernünftige Schriftgröße (Control-Plus)!! Gleichzeitig sichtbar (d. h.: keine Verdeckungen, Umschaltungen): Aufgabenstellung, Programmtext, Ausgabe/Fehlermeldungen. Wenn der Desktop-Hintergrund sichtbar ist—wurde Platz verschenkt!!!

3. Geben Sie einen Typ (eine `data`-Deklaration) mit genau 91 Elementen an. Sie können andere Data-Deklarationen benutzen (wie in autotool-Aufgabe). Minimieren Sie die Gesamtzahl aller deklarierten Konstruktoren.

4. Beweisen Sie $\forall \Sigma : \forall t \in \mathrm{Term}(\Sigma) : \mathsf{height}(t) \leq |t| - 1$.

   durch Induktion über den Term-Aufbau.

   - Induktions-Anfang: $t = f()$ (nullstelliges Symbol f)
   - Induktions-Schritt:
     $t = f(t_1, \ldots, t_k)$ (k-stelliges Symbol f, für $k > 0$)
     dabei Induktions-Voraussetzung: die Behauptung gilt für $t_1, \ldots, t_k$.
     Induktions-Behauptung: ... für t.

   Für welche Terme t gilt Gleichheit? Wo sieht man das im Beweis?

5. wieviele Elemente des Datentyps `data T = L | B T T` haben ...

   - die Größe 9
   - die Größe ≤ 9
   - die Höhe $0, 1, 2, 3, \ldots, k, \ldots$

   Sie müssen diese Elemente nicht alle einzeln angeben.

   Bestimmen sie ihre Anzahl durch dynamische Programmierung (von Hand).

3 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 $\mathrm{Pos}(t) = \{[]\} \cup \{[i] \cdot p \mid 0 \leq i < k \wedge p \in \mathrm{Pos}(t_i)\}$.

dabei bezeichnen:

- [] die leere Folge,
- $[i]$ die Folge der Länge 1 mit Element i,
- $\cdot$ 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 (eines case-Ausdrucks mit Typ `T`)
  - jedes `<Muster>` hat gleichen Typ wie `<Diskrim.>`,
  - jeder `<Ausdruck>` hat den Typ `T`
- dynamische Semantik:
  - Def.: t *paßt* zum Muster l: es existiert σ mit $l\sigma = t$
  - für das erste Muster, das zum Wert der Diskriminante paßt, 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 `https://docs.scala-lang.org/tutorials/tour/case-classes.html`
- Java (ab JDK 21)

```
jshell --enable-preview  # Version 21-ea
interface I {}
record A (int x) implements I {}
I o = new A(4)
switch (o) {
  case A(var y) : System.out.println(y);
  default : }
```

- 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 $f(x, S(x))\sigma$.

Dabei wurde angewendet:

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)

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 -> False }
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 -> False; C -> True }
```

- Listen von Wahrheitswerten:

```
data List = Nil | Cons Bool List deriving Prelude.Show

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

SS 23: Aufgaben 1, 2, 4, 5, 6

1. Arithmetik auf Peano-Zahlen

   - 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.
   - Welche Rechenoperationen simulieren die Regeln für f, für d?
   - welche Terme haben große Normalformen?
   - Zusatz: Geben Sie eine Funktion $c :\to$ an, für die gilt: $\forall n \in: \forall t, t' \in \mathrm{Term}(\Sigma_d) : |t| \leq n \wedge t \to_{R_d}^* t' \Rightarrow |t'| \leq c(n)$.

2. Simulation von Wort-Ersetzung durch Term-Ersetzung.

   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.

3. 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.

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

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

5. für selbst definierte Wahrheitswerte: deklarieren, implementieren und testen Sie:

   - die zweistellige Antivalenz,

- die Implikation,
- die dreistellige Majoritätsfunktion.

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

Definieren Sie die Majorität auf verschiedene Weisen

- mit *einer* Gleichung (evtl. mit `case`, evtl. geschachtelt)
- *ohne* `case` (evtl. mehrere Gleichungen)
- mit einer Gleichung ohne Fallunterscheidung, mittels anderer (selbst definierter) Funktionen

6. 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 ungerade Anzahl von Blättern enthält.

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

4 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: forall x::Bool : 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: $0 \in, \forall x : x \in \Rightarrow (1 + x) \in$
- realisiert als algebraischer Datentyp

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

  Zahl $n \in$ 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 Assoziativität $x + y = (1 + x') + y = \ldots$
  ```

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 : forall a :: N, b :: N, c :: N :
    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
  ```

Bsp. Programm-Konstruktion/Induktion

- gegeben: Peano-Zahlen und Binärzahlen:

```
data B = Zero | Even B | Odd B
value :: B -> N
value Zero = Z
value (Even x) = doubleN (value x)
value (Odd x) = S (doubleN (value x))
```

- gesucht: Nachfolgerfunktion für Binärzahlen

```
succB :: B -> B  -- Implementierung ist zu ergänzen
Lemma :
  forall b :: B : value (succB b) .=. S (value b)
```

- Renz, Schwarz, Waldmann: *Check your Students' Proofs—with Holes*, WFLP 2020, `https://arxiv.org/abs/2009.01326`

Induktion mit Generalisierung

- Bsp: im Induktionsschritt für Beweis von

```
forall x::N, y::N : plus' x y .=. plus x y
```

  lautet die Ind.-Voraus. `plus' x' y .=. plus x' y`
- Beweis der Ind.-Behauptung benötigt Umformung von `plus' x' (S y)`. Das erfordert

```
Proof by induction on x:: N generalizing y::N
```

- ohne `generalizing`: $\forall y : (\forall x : P(x, y))$,
  d.h., für jedes außen fixierte y eine Induktion nach x
  (I.V. muß mit genau diesem y benutzt werden)
- mit `generalizing y`: $\forall x : (\forall y : P(x, y))$,
  d.h., für jedes x wird die Behauptung für alle y gezeigt.
  (I.V. kann mit beliebiger Belegung von y benutzt werden)

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

Minimum

- vollständige Spezifikation:

```
forall x :: N, y :: N : min (plus x y) y = y
forall x :: N, y :: N : min x (plus x y) = x
```

  vollständig bedeutet: es gibt nur eine Funktion, die die Spezifikation erfüllt
- Definition durch vollständige Fallunterscheidung

```
min Z Z = _ ; min Z (S y) = _ ; min (S x) Z = _
min (S x) (S y) = S (min x y)
```

- Ü: Beweis, daß diese Imp. die Spez. erfüllt
- Ü: desgleichen für Maximum

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:
  - $\forall a, b \in$: $(a + b) \ominus b = a$
  - $\forall a, b \in$: $a \ominus (a + b) = 0$
- Implementierung (Muster disjunkt? vollständig?)

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

Hausaufgaben

im SS23: Aufgaben 2, 3, 4

1. 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 Bezeichnungen für die Beweis-Schritte, geben Sie IA, IV, IB explizit an.

2. Die übliche Peano-Addition ist

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

   Eine andere Implementierung der Addition (vgl. früher angegebenes Termersetzungssystem) ist

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

   Beweisen Sie mit Cyp

   ```
   forall x :: N, y :: N : plus' x y .=. plus x y
   ```

   Beweisen Sie dazu als Hilfssatz

   ```
   forall x :: N, y :: N : plus x (S y) .=. plus (S x) y
   ```

   In dieser Induktion nach x müssen Sie das andere Argument y generalisieren, da es zwischen Induktionsvoraussetzung und Induktionsbehauptung unterschiedlich ist.

3. 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.

4. Für zweistelliges min (siehe Folie) und max auf :

   (a) Geben Sie eine äquivalente vollständige Spezifikation an, die keine Fallunterscheidung benutzt, sondern nur Addition.

   (b) Implementieren Sie min und max nur durch Addition und Subtraktion ($\ominus$).

   (c) testen (optional: und beweisen) Sie, daß Ihre Implementierung die Spezifikation erfüllt

   (d) Implementieren Sie nur mit min und max:

   - den Median von drei Argumenten
   - (optional) 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.)

   iii. welches sind die Terme (z.B.: der Größe 11) mit größter Normalform?

5 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))
  ```

Anwendung von Maybe

- `data Maybe a = Nothing | Just a`

  `Nothing` drückt das Fehlen eines Wertes aus, `Just x` sein Vorhandensein

- ```
  head :: List a -> Maybe a
  head Nil = Nothing; head (Cons x xs) = Just x
  ```

- andere Ausdrucksmöglichkeiten (in anderen Sprachen)

  - falscher Typ und Ausnahme (Exception)

    `head :: List a -> a; head Nil = error "huh"; ...`

  - `Maybe T` als Zeiger-auf-T, oder auf nichts
    der *billion dollar mistake*, Algol W, Tony Hoare 1965,
  - `Optional<T>`, "may or may not contain a value" `https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Optional.html`

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) (maximum ys)
```

- – 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 ⇒ früher erkannte Fehler lassen sich leichter beheben
  - beim Ausführen: keine Lauzeit-Typfehler ⇒ 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

im SS22: Aufgaben 1, 2, optional: 3

1. für die folgenden Datentypen: geben Sie einige Elemente an (ghci), geben Sie die Anzahl aller Elemente an.

   (a) `Maybe (Maybe Bool)`

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

   (c) `Foo (Maybe (Foo Bool))` mit `data Foo a = C a | D`

   stellen Sie (dann in der Übung) ähnliche Aufgaben

   geben Sie ein möglichst kleines Programm an, das nur aus data-Deklarationen besteht, und das einen Typ mit 100 (Zusatz: 1000) Elementen definiert.

   Diskussion: vergleiche frühere Aufgabe. Ein solcher Typ ist

   ```
   Maybe (Maybe (Maybe .... (Maybe ()) .. ))
   ```

   mit insgesamt nur drei Konstruktoren (Nothing, Just, `()`).

   Also sollte man zur gerechten Messung der Programmgröße die Anzahl der Konstruktoren und die Größe der Typ-Ausdrücke addieren.

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`)

   Geben Sie einen Algorithmus zur Lösung der Rekonstruktions-Aufgabe (preorder, postorder) an.

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. (zu 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.

6 Lambda-Kalkül: Syntax und dynamische Semantik (Auswertung)

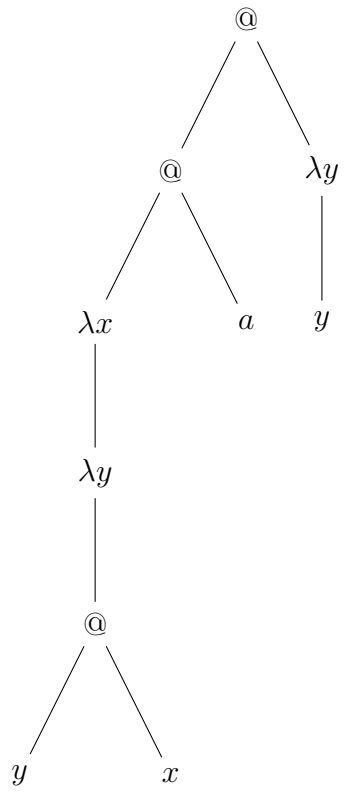
Funktionen als Daten

- 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: $(\lambda x.B)A \to B[x := A]$
  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.: Termersetzungssystem, Turingmaschine, Goto-Programme, While-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*: 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.)

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
  ```
  ```

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 $t[p] = (\lambda x.B)$
  - $y \notin \mathrm{bvar}(B) \cup \mathrm{fvar}(B)$
  - $t' = t[p := \lambda y.B[x := y]]$

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
```

Ein AST für Lambda-Terme

- Λ (Menge der Terme = abstrakter Syntaxbäume, AST)
  realisiert als algebraischer Datentyp

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

- Positionen, Navigation zu Teilterm, Teilterm-Ersetzung

  ```
  data Dir = L | R | O ; type Pos = List Dir
  pos :: Term -> List Pos -- alle Positionen
  get :: Pos -> Term -> Maybe Term
  get Nil t = Just t
  get (Cons L p) (App l r) = get _
  put :: Pos -> Term -> Term -> Maybe Term
  put (Cons L p) t (App l r) =
    case put p t l of Just l' -> Just (App l' r)
  ```

- Mengen von Variablen:

  ```
  import qualified Data.Set as S
  var  :: L -> S.Set String -- alle Variablen
  bvar :: L -> S.Set String -- alle gebundenen
  fvar :: L -> S.Set String -- alle frei vorkommenden
  ```

  API-Dokumentation: `https://hackage.haskell.org/package/containers/docs/Data-Set.html`

  Ansatz:

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

Hausaufgaben

Für SS23: 2, 4, Zusatz: 5

1. `bvar` und `fvar` implementieren, Testfälle vorführen (aus Skript und weitere)
2. `pos`, `get` und `put` implementieren, Testfälle vorführen.
3. für $t = \lambda fx.f(fx)$: AST von $ttSZ$ zeichnen,

   Normalform bestimmen: 1. auf Papier, 2. mit `ghci`.

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

4. 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

5. Für $n \in$ bezeichnet (nur für diese Aufgabe, die eckigen Klammern bedeuten oft etwas anderes) $[n]$ den Lambda-Ausdruck $\lambda fx.f^n(x)$,

   dabei ist $f^n(x)$ die n-fach iterierte Anwendung von f auf x,

   also $[0] = \lambda fx.x, [1] = \lambda fx.fx, [2] = \lambda fx.f(fx), [3] = \lambda fx.f(f(fx))$ usw.

   die Normalform von $[2]sz$ ist $s(sz)$, das entspricht der Peano-Repräsentation der Zahl 2.

   - für $p = \lambda abfx.af(bfx)$: bestimmen Sie die Normalform von $p[2][1]sz$
   - bestimmen Sie die Normalform von $[3][2]sz$

7 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 $\lambda fx.f(fx)$

- 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 $\lambda fx.f(fx) :: (t_{12} \to t_{12}) \to (t_{12} \to t_{12})$

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 → Stelligkeit
    abstrakter Syntaxbaum: Funktionss. über Argumenten
  - Lambda-Kalkül: jeder Lambda-Ausdruck beschreibt einstellige Funktion
    Simulation mehrstelliger Funktionen wegen
    Isomorphie zwischen $(A \times B) \to C$ und $A \to (B \to C)$
- 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 -> 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 a
takeWhile p xs = case xs of
  Nil -> Nil
  Cons x xs' -> case p x of
    False -> Nil
    True  -> Cons x (takeWhile p xs')
```

- (Vorgriff) eigenschaftsbasiertes Testen mit `https://hackage.haskell.org/package/leancheck` (Rudy Matela, 2017). Zu dem (inferierten) Typ der Eigenschaft (Funktion nach Bool) werden Test-Argumente automatisch erzeugt

  - Installation und Benutzung

```
cabal install --lib leancheck
ghci> import Test.LeanCheck
ghci> check $ \ x y -> not (x && y) == not x  || not y
+++ OK, passed 4 tests (exhausted).
```

  - Erzeugung von Test-Funktionen, Festlegung von Typen

```
import Test.LeanCheck
import Test.LeanCheck.Function
:set -XPatternSignatures
check $ \ (p :: Bool -> Bool) xs ->
  takeWhile p xs ++ dropWhile p xs == xs
```

Hausaufgaben

SS 23 zu VL KW 20: Aufgaben 1a, 1b (allgemeinsten Typ von ghci ausrechnen lassen), 3;
zu VL KW 21: Aufgaben 1c, 1d (allgemeinsten Typ selbst ausrechnen), 5.

1. für die Lambda-Ausdrücke

(a) $\lambda fghxy.f(gx)(hy)$:

(b) $(\lambda x.xx)(\lambda y.yy)$

(c) $\lambda xyz.xy(yz)$

(d) $\lambda ab.b(\lambda x.xa)$

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

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. für `Data.List`, `Data.Sequence`

   die Funktionen `partition`, `span`, `inits`, `tails`

   (a) aufrufen (in ghci Beispiele vorführen), dabei anwenden auf eine Liste/Folge von Bool

   (b) die API-Dokumentation aufsuchen (hackage.org)

   (c) die dort angegebenen Eigenschaften als Leancheck-Property überprüfen

6. Typisierung von skkFalse (vgl. Aufgabe vorige Woche) in Java oder C.

   Vorübung: in Haskell, dabei für s und k alle Typen deklarieren und alle Typ-Argumente angeben.

```
i :: a -> a ; i = \ x -> x  -- nur für Beispiel
k :: a -> b -> a ; k = \ x y -> x
s :: (a -> b -> c) -> ... ; s = \ x y z -> x z (y z)

i @(Bool -> Bool) (i @Bool) False   -- Beispiel
s @(...) (k @...) (k @...) False
```

   in Java und C#: besteht Unterschied zwischen Werten und Methoden: Methoden können polymorph sein, Werte nicht.

   Lambda-Ausdrücke sind Werte, also nicht polymorph:

```
Function<Boolean,Boolean> i = (x) -> x  // monomorph
```

   im Bespiel skkFalse muß k polymorph sein, da es mit zwei verschiedenen Belegungen der Typ-Argumente benutzt wird.

   Lösung (d.h., Umweg): eine polymorphe nullstellige Methode schreiben, die den Lambda-Ausdruck liefert.

```
$ jshell
jshell> class C {
  static <A> Function<A,A> i() { return (x)->x; }
  // hier s und k ähnlich deklarieren
}
jshell> C.<Function<Boolean,Boolean>>i().apply(C.<Boolean>i()).apply(false)
```

   In C# genauso, aber das `apply` entfällt.

```
$ csharp
csharp> class C { public static Func<A,A> i<A>() { return x=>x; }}
csharp> C.i<Func<Boolean,Boolean>>()(C.i<Boolean>())(false)
```

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

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

8 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 ⇒ 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

Rekursionsm. (Peano-Zahlen) (einst.)

- 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 z s`

  I.A. `dbl Z = Z` und `z = fold z s Z`, also `z = 0`

  I.S. `dbl (S x')=2*(1+x')=2+2*x'=2+dbl x'`, `dbl (S x') = s (dlb x')`, also `s a = 2 + a = S (S a))`

  Lösung: `dbl = fold Z (\ a -> S (S a))`

Rekursionsm. (Peano-Zahlen) (zweist.) (I)

- Bsp: `plus x y = fold z s x`, bestimme `z, s`
- I.A.: `x = Z; plus Z y = y`; `fold z s Z = z` $\Rightarrow$ `z = y`.
  I.S.: `x = S x'`, `plus (S x') y = S (plus x' y)`, `plus (S x') y=fold z s (S x = s (plus x' y)`, also `s a = S a`
  Lösung: `plus x y = fold y (\ a -> S a) x`
- äquivalent, kürzer: `plus x y = fold y S x`

- weil Addition kommutativ ist: äq.
  `plus x y = plus y x = fold x S y`
  dann kann man `y` auf beiden Seiten weglassen:
  `plus x = fold x S`

Rekursionsm. (Peano-Zahlen) (zweist.) (II)

- Bsp: `plus x y = fold z s x y`, äq. `plus x = fold z s x`, bestimme `z, s`
- I.A.: `x = Z; plus Z y = y`; `plus Z = \ y -> y`; also `z = \ y -> y`.
  I.S.: `x = S x'`; `plus (S x') y = S (plus x' y)`; `plus (S x') = \ y -> S (plu`
  `= \ y -> (S . plus x') y = (S . plus x')`
  `plus (S x') = fold z s (S x') = s (fold z s x') = s (plus x')`
  also `s a = S . a` (mit `(f . g) = \x-> f (g x)`)
  L.: `plus x = fold (\ y -> y) (\a -> S . a) x`
  äq. `plus = fold id (S .)` (operator section)

Hausaufgaben
SS 23: 1, 2, 3.

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

   `Bool, Maybe a, Pair a b, Either a b`

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

2. implementieren Sie mit dem Rekursionsmuster auf Peano-Zahlen:
   - plus, mal, hoch
   - `g :: N -> Bool` mit $gx =$ (x ist eine gerade Zahl)
3. für den Datentyp der binären Bäume mit Schlüsseln *nur* in den Blättern

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

(a) aus dem Typ das Rekursionsschema ableiten

```
fold :: ...
```

(b) 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 $p \in 0^*$)

9 Rekursionsmuster (Forts.)

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]
    ```

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 modulo $2^{\text{Wortbreite}}$

Merksätze

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

Hausaufgaben

SS 23: gestellt in KW23, für KW 24

1. 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`)

2. 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.

3. 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, Ma` 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 beachten. Hier ist `Numeric.Natural` sinnvoll.

10 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`
- 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 -> St`
  `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 I;`
  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 prob`
- 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) where ...
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

im SS 23: Aufgaben 1, 2, 4, 5 (aber pro Termin reicht die Zeit evtl. nur für drei Aufgaben)

grundsätzlich (bis Semester-Ende) Falls es Streit gibt, wer präsentiert: es kommen die Personen/Gruppen bevorzugt dran, die bisher weniger Punkte haben.

Wenn diese nicht wollen, nicht vorbereitet sind, oder noch Zeit verbleibt, dann auch andere Gruppen und andere Aufgaben.
  ```

1. diese Aufgabe für selbst definierten Datentyp `T`, mit Typklassen `Eq` und `Ord` aus der Standardbibliothek:

   Definieren Sie für `data T = T Bool Bool Bool`

   - `instance Eq T` als komponentenweise Gleichheit
   - `instance Ord T` als lexikografisches Produkt der Standard-Ordnung auf Bool (zitieren Sie zunächst die mathematische Definition, z.B. aus der VL Modellierung)

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

   - transitiv
   - antisymmetrisch

   ist und prüfen Sie das für `Integer` und für `T`.

```
instance Listable T where tiers = cons3 T

leq_is_transitiv :: Ord a => a -> a -> a -> Bool
leq_is_transitiv = \ x y z -> _

check (leq_is_transitiv @T)
```

2. diese Aufgabe für selbst definierten Datentyp `T`, mit Typklassen `Semigroup` und `Monoid` aus der Standardbibliothek:

   Definieren Sie für `data T = T Bool Bool Bool`

   - `instance Semigroup T` als komponentenweise Konjunktion
   - `instance Monoid T` als dazu passendes neutrales Element

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

   Geben Sie eine andere korrekte Monoid-Struktur für `T` an, die nicht kommutativ ist. Testen Sie diese Eigenschaft (leancheck soll das Gegenbeispiel finden).

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 Typconstraints (oder deren Fehlen) für `singleton`

   für Data.Map:

   - warum haben die Typen von `unionWith` und `intersectionWith` unterschiedliche Anzahl von Typvariablen?
   - Erläutern Sie Typconstraints (oder deren Fehlen) für `singleton`, `fromAscList`, `fromDistinctAscList`
   - Definieren Sie einen Typ `data K = ... deriving Eq`. Implementieren Sie `instance Ord K` so, daß die Relation `(<=)` keine totale Ordnung ist.

     Zeigen Sie durch Beispiele, daß dann `M.Map K v` nicht funktioniert. Z.B.: mit `M.insert` eingefügter Schlüssel nicht in `M.toList` zu sehen, sichtbarer Schlüssel nicht gefunden mit `M.lookup`.

5. für `instance Semigroup Ordering`, `instance Monoid Ordering` aus der Standardbibliothek:

   Bestimmen Sie die Wertetabelle von `(<>)`.

   Überprüfen Sie (mit leancheck) die Axiome von Semigroup und Monoid. Ist die Verknüpfung kommutativ?

   Für den Datentyp `data Pair a b = Pair a b`: implementieren Sie den lexikografischen Vergleich

```
instance (Ord a, Ord b) => Ord (Pair a b) where
  compare (Pair px py) (Pair qx qy) = _
```

- mit Fallunterscheidungen
- ohne Fallunterscheidung, aber mit `(<>)`

11 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 N ; nf :: N -> Stream N
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` (bedeutet: $\perp \in T_\perp$)
- Nicht-Termination `f x=f(f x)` $\Rightarrow f0 = \perp$)

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 ein Graph-Ersetzungs-Schritt) (nicht: Term-...)
- 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` `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) of
    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

SS23: 1, 2; optional: 3, 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) wie folgt:

   ```
   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 (() : undefined)
   ```

4. Folie Ablaufsteuerung: Ifthenelse (`wenn`) 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

12 Fkt. höherer Ord. für Streams

Motivation

- Verarbeitung von Datenströmen,
- durch modulare Programme,

  zusammengesetzt aus elementaren Strom-Operationen
- angenehme Nebenwirkung (1):

  (einige) elementare Operationen sind parallelisierbar
- angenehme Nebenwirkung (2):

  externe Datenbank als Datenquelle, Verarbeitung mit Syntax und Semantik (Typsystem) der Gastsprache

Motivation: Parallel-Verarbeitung

geeignete Fkt. höherer Ordnung ⇒ triviale Parallelisierung:

```
Func<int,int> f = x => ... // teure Rechnung
var s = Enumerable.Range(1, 20000)
     .Select( f ).Sum() ;
var s = Enumerable.Range(1, 20000)
     .AsParallel()
     .Select( f ).Sum() ;
```

Dadurch werden

- Elemente parallel verarbeitet (`.Select(f)`)
- Resultate parallel zusammengefaßt (`.Sum()`)

vgl. `http://msdn.microsoft.com/en-us/library/dd460688.aspx`

Strom-Operationen

- erzeugen (produzieren):
  - `n = 1 : map (\x -> x+1) n`
  - `Enumerable.Range(int start, int count)`

  - eigene Instanzen von `IEnumerable`

- transformieren:
  - elementweise: `map`, `Select`
  - gesamt: `take`,`drop`,`filter` (`Take, Skip, Where`)

- verbrauchen (konsumieren):
  - `fold`, `Aggregate`
  - Spezialfälle: `All, Any, Sum, Count`

Struktur-erhaltende Strom-Transf.

- elementweise (unter Beibehaltung der Struktur)
- `map :: (a -> b) -> [a] -> [b]`
- Realisierung in C#:

```
IEnumerable<B> Select<A,B>
   (this IEnumerable <A> source,
    Func<A,B> selector);
```

- Rechenregeln (Implementierung) für `map`:

```
map f [] = ...  ; map f (x : xs) = ...
```

- damit kann man beweisen:

```
map f (map g xs) = map (\ x -> ... ) xs
```

Struktur-erhaltende Abb. allgemein

- jeder polymorphe algebraischen Datentyp (Container) `data T a = ...`
  besitzt eine struktur-erhaltende elementweise Transformation (die das Typ-Argument ändern kann)
- diese ist Methode der Konstruktorklasse `Functor`:

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

- Instanzen deklarieren `instance Functor T where fmap` oder vom Kompiler erzeugen lassen

```
{-# language DeriveFunctor #-}
data List a = ... deriving Functor
```

- benutzen: `fmap (\ x -> 2*x) (Cons 3 (Cons 5 Nil))`

Forderungen an Funktoren

- jeder Funktor-Instanz soll diese Axiome erfüllen:
  - `fmap id = id` (wobei `id = \ x -> x`)
  - $\forall$`f,g : fmap (f . g) = fmap f . fmap g`
    (wobei `(f . g) = \ x -> f(g x)`)
- die durch `deriving Functor` erzeugten tun es
- Ü: Implementierung ergänzen und Axiome überprüfen:

```
instance Functor Maybe where
  fmap f m = case m of
    Nothing -> _
    Just x -> _
```

Struktur-ändernde Strom-Transf.

- Änderung der Struktur, Beibehaltung der Elemente
- aus Haskell-Standardbibliothek:

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
```

- Realisierung in C# (LINQ): `Take, Drop, Where`
- Übung: `takeWhile, dropWhile, span`

- ausprobieren (Haskell, C#)
- implementieren
  Haskell: 1. mit expliziter Rekursion, 2. mit `fold`
  C# (Enumerator): 1. mit `Current`,`MoveNext`, 2. `yield`

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

13 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.

Zusammenfassung: Ströme

-

C# (Linq)	Haskell
`IEnumerable<E>`	`[e]`
Select	map
SelectMany	>>= (bind)
Where	filter
Aggregate	foldl

- mehr zu Linq: `https://msdn.microsoft.com/en-us/library/system.linq(v=vs.110).aspx`
- Ü: ergänze die Tabelle um die Spalte für Streams in Java `https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/stream/package-summary.html`

Hausaufgaben

SS 23: alles, empfohlen: $1 > 2 > 3 > \ldots$

1. die Funktion `fromBits :: [Bool] -> Integer`, Beispiel `fromBits [True,False,Fal`

   ...als `foldr` oder als `foldl` ?

   Geben Sie die eine Darstellung an, begründen Sie, daß die andere unmöglich ist.

2. Vervollständigen Sie die Gleichung

   ```
   foldl f a (map g xs) = foldl _ _
   ```

   so, daß rechts kein `map` steht. Dieses Verschwinden des `map` heißt *stream fusion* (Coutts, Leshchinsky, Stewart, 2007) `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.7401`

   Vervollständigen Sie die Gleichung

   ```
   foldr f a xs = foldl _ _  (reverse xs)
   ```

3. 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, die Reihenfolge der Argumente im Teilausdruck `q x && p x` nicht vertauschen darf. Warum—die Konjunktion ist kommutativ?

4. Geben Sie jeweils die strukturerhaltende Transformation als Funktor-Instanz an:

   (a) Binärbäume mit Schlüssel in Verzweigungsknoten

   ```
   data Tree a = ...
   instance Functor Tree where
     fmap f t = case t of ...
   ```

   (b)
   ```
   data T a = T (Bool -> a)
   instance Functor T where   fmap f (T g) = ...
   ```

   Überprüfen Sie (mit Leancheck) die Funktor-Axiome.

   Vergleichen Sie das Verhalten Ihrer Implementierung mit der von `deriving Functor`.

5. warum ist keine der folgenden Instanzen korrekt? Überprüfen Sie statische Korrektheit (Typisierung) und dynamische Korrektheit (Axiome)

   (a) `instance Functor Maybe where fmap f m = m`

   (b) `instance Functor Maybe where fmap f m = Nothing`

   (c)
   ```
   instance Functor [] where
     fmap f xs = reverse (map f xs)
   ```

   (d)
   ```
   instance Functor [] where
     fmap f xs = take 1 (map f xs)
   ```

   bei (c) und (d) ist `[]` der Typkonstruktor für Listen aus der Standardbibliothek, `map` die korrekte Implementierung.

6. (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)
     ```

7. definieren Sie den Operator

```
f >=> g = \ xs -> f xs >>= g
```

   Bestimmen Sie den Typ dieses Operators.
   Ergänzen Sie den Ausdruck, so daß er den gegebenen Wert hat:

```
ghci> ((\x ->[1,x]) >=>  _ ) 5
[4,2,8,2]
```

   Prüfen Sie, daß `(>=>)` assoziativ ist

14 Ergänzg., Zusammenfassg., Ausblick

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 $(\lambda x.fx)y \to_\beta fy$
- 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

Geschenkte Theoreme (free theorems)

- für jedes `g :: forall a . a -> [a]` gilt:

  $\forall$ `h :: a -> b, x :: a`: `g (h x) = map h (g x)`

- das folgt allein aus dem *Typ* von `g` (polymorph in `a` $\Rightarrow$ Implementierung von `g` weiß nichts über `a`)

- 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.

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 Th.)
- (Algorithmen für persistente Daten)

Prüfungen

- Zulassung:
  - 3 mal Vorrechnen,
  - 50 Prozent (= 7) autotool-Pflicht
- Klausur,
  - 2 Stunden, ohne Hilfsmittel
  - (sehr wahrscheinlich) 2 Arbeitsblätter, insg. 4 Seiten,
  - Lückentextaufgaben (Antwort im Arbeitsblatt eintragen)
    Konzeptpapier verwenden (für Nebenrechnungen),
    aber nur bei Zeitnot mit abgeben
- Orientierung Klausur-Aufgaben:
  ähnlich zu den Hausaufgaben, die auf konkrete kurze Argumentation/Rechnung abzielen (keine lange Programmierung, keine lange Forschung)

Aussagen

- statische Typisierung ⇒
  - findet Fehler zur Entwicklungszeit (statt Laufzeit)
  - effizienter Code (keine Laufzeittypprüfungen)
- generische Polymorphie: flexibler *und* sicherer Code
- Funktionen als Daten, F. höherer Ordnung ⇒
  - ausdrucksstarker, modularer, flexibler Code

Programmierer(in) sollte

- die abstrakten Konzepte kennen
- sowie ihre Realisierung (oder Simulation) in konkreten Sprachen (er)kennen und anwenden.

Objekt- (Klassen-) orientierte Progr.

- nützlich (und überhaupt nicht neu) sind
  - Zusammenfassg. v. Daten (Record) und Eigenschaften
  - 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
    $\iff$ 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`)
  - Balance (log. Höhe) durch Typisierung (Fingertree)

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 (Digit a) (FingerTree (Node a)) (Digit a)
data Node a = Node2 a a | Node3 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`
  ```

Wie weiter?

- genauere Betrachtung von statischer und dynamischer Semantik von programmiersprachlichen Konstrukten

  in VL Prinzipien von Programmiersprachen (Master-Pflicht), Compilerbau (M-Wahl)

- in Haskell eingebettete domainspezifische Sprachen in

  VL Constraint-Programmierung (Bachelor-W), Computermusik (M-W)

- Anwendung der fktl. Programmierung in autotool

  (Semantik, Datenbank, Weboberfläche)

  ich betreue jederzeit gern B- und M-Arbeiten, die den Code dokumentieren, testen, verbessern, erweitern.