

# Fortgeschrittene Programmierung Vorlesung WS 09,10; SS 12–14, 16–19, 21–24

Johannes Waldmann, HTWK Leipzig

25. April 2024

## 1 Organisatorisches

### Ablauf Lehrveranstaltung und Prüfung

- jede Woche eine Vorlesung, dort werden Hausaufgaben gestellt
- Aufgaben zur Präsentation in der nächsten Übung  
jeder soll (als Teil einer Gruppe) insgesamt 3 mal präsentieren
- Aufgaben in autotool (selbständige Bearbeitung innerhalb der nächsten zwei Wochen)  
jeder soll insgesamt 50 % der Pflichtaufgaben erledigen
- Prüfung (vorbehaltlich Beschluß Prüfungsausschuß)  
in elektrischer Form (autotool, in Präsenz unter Aufsicht)

### Quellen

- Skript (Woche für Woche) <https://www.imn.htwk-leipzig.de/~waldmann/lehre.html>  
Plan: vgl. *How I Teach Functional Programming, WFLP 2017*, <https://www.imn.htwk-leipzig.de/~waldmann/talk/17/wflp/>
- Quelltexte aus Vorlesung, Vorbereitung Hausaufgaben (zur Präsentation): <https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/fop-ss24>  
*Einschreiben!* (Projekt wird dann *private* geschaltet)

- **Hausaufgaben (individuell)** <https://autotool.imn.htwk-leipzig.de/new/vorlesung/324/aufgaben/aktuell>  
*Einschreiben! (Übungsgruppe wählen)*

## Übungen KW 15 (vor Vorlesung)

- **Arbeiten im Pool: Shell, \$PATH, ghci, vgl.** <https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/>

```
$ cabal update
# $HOME/.config/cabal/config editieren (-haddock)
$ cabal install --lib leancheck
$ ghci
GHCi, version 9.8.2: https://www.haskell.org/ghc/  :? for help
ghci> import Test.LeanCheck
ghci> check $ \ p q -> (p && q) == (q && p)
ghci> :doc check
```

- `ssh-keygen, .ssh/id_rsa.pub ⇒ gitlab.dit, git clone`
- **wenn Zeit ist: autotool 15-1**, <https://www.mathekalender.de/wp/de/kalender/aufgaben/2023-01-de/>

## 2 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*  $f : \text{Eingabe} \mapsto \text{Ausgabe}$ ,  
(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
- Ausdrucksstärke, 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:  $\text{app } x \text{ (app } y \text{ } z) \text{ .}. \text{ app (app } x \text{ } y) \text{ } z$

Proof by induction on List x

Case []

To show:  $\text{app [] (app } y \text{ } z) \text{ .}. \text{ app (app [] } y) \text{ } z$

Case h:t

To show:  $\text{app (h:t) (app } y \text{ } z) \text{ .}. \text{ app (app (h:t) } y) \text{ } z$

IH:  $\text{app } t \text{ (app } y \text{ } z) \text{ .}. \text{ app (app } t \text{ } y) \text{ } z$

CYP <https://github.com/noschinl/cyp>,

ist vereinfachte Version von Isabelle <https://isabelle.in.tum.de/>

## Beispiel Parallelisierung (Haskell)

```
-- Länge der Collatz-Folge
collatz :: Int -> Int
collatz x = if x <= 1 then 0
            else 1 + collatz (if even x then div x 2 else 3*x+1)
-- Summe der Längen
main :: IO ()
main = print $ sum
      $ map collatz [1 .. 10^7]
```

wird parallelisiert durch *Strategie-Annotation*:

```
import Control.Parallel.Strategies
...
main = print $ sum
      $ withStrategy (parListChunk (10^5) rseq)
      $ map collatz [1 .. 10^7]
```

## 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](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 Programming*, 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` (benutzerdefinierte 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/#\(11\)](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..100]
```

- 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 24: 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 behandelt) 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).

### 3 Daten

#### Wiederholung: Terme

- (Prädikatenlogik) *Signatur*  $\Sigma$  ist Menge von Funktionssymbolen mit Stelligkeiten ein Term  $t$  in Signatur  $\Sigma$  ist
  - Funktionssymbol  $f \in \Sigma$  der Stelligkeit  $k$  mit Argumenten  $(t_1, \dots, t_k)$ , die selbst Terme sind.

$\text{Term}(\Sigma)$  = Menge der Terme über Signatur  $\Sigma$

- (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  $\text{Term}(\Sigma)$ :  
 $Z(), S(S(Z())), f(S(S(Z))), Z()$
- 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  $\text{Term}(\Gamma)$ : ...
- Bezeichnung: für Signatur  $\Sigma$  und  $k \in \mathbb{N}$ :  
 $\Sigma_k$  bezeichnet Menge der Symbole aus  $\Sigma$  mit Stelligkeit  $k$   
 $\Sigma_0 = \{Z\}, \Sigma_1 = \{S\}, \Sigma_2 = \{f\},$   
 $\Gamma_0 = \{E\}, \Gamma_1 = \dots, \Gamma_2 = \dots$

### Abmessungen von Termen

- die Größe: ist Funktion  $|\cdot| : \text{Term}(\Sigma) \rightarrow \mathbb{N}$  mit
  - für  $f \in \Sigma_k$  gilt  $|f(t_1, \dots, t_k)| = 1 + |t_1| + \dots + |t_k|$

die Größe eines Terms ist der Nachfolger der Summe der Größen seiner Kinder
- Bsp:  $|S(S(Z()))| = 1 + |S(Z())| = 1 + 1 + |Z()| = 1 + 1 + 1$   
 $|f(S(S(Z())), Z())| = \dots$
- die Höhe: ist Funktion  $\text{height} : \text{Term}(\Sigma) \rightarrow \mathbb{N}$ :  
 für  $t = f(t_1, \dots, t_k)$  gilt
  - wenn  $k = 0$ , dann  $\text{height}(t) = 0$
  - wenn  $k > 0$ , dann  $\text{height}(t) = 1 + \max(\text{height}(t_1), \dots, \text{height}(t_k))$

### Induktion über Termaufbau (Beispiel)

- Satz:  $\forall t \in \text{Term}(\{a/0, b/2\}) : |t| \equiv 1 \pmod{2}$  (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 wechselt 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
```

- Beispiele (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  $S = \{S_1, \dots\}$
  - $msS$  ist Abb. von Funktionssymbol nach Typ
  - $Typ$  ist Element aus  $S^* \times S$   
Folge der Argument-Sorten, Resultat-Sorte

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

- $Term(\Sigma, B)$  (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  $|\sqrt{a \cdot a + b \cdot b}|$ ,  $\text{height}(\sqrt{a \cdot a + b \cdot b})$ .

- Geben Sie ein Element  $t \in \text{Term}(\{f/1, g/3, c/0\})$  an mit  $|t| = 5$  und  $\text{height}(t) \leq 2$ .
- die Menge  $\text{Term}(\{f/1, g/3, c/0\})$  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 24: Aufgaben 2, 3, 4.

### 1. autotool-Aufgabe 15-2

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  $t \in \text{Term}(\Sigma)$  an mit
- $\text{height}(t) = 2$  und  $|t|$  möglichst klein
  - $\text{height}(t) = 2$  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 71 Elementen an. Sie können weitere Data-Deklarationen benutzen. Minimieren Sie die Gesamt-Anzahl der Konstruktoren.

Bsp: `data Bool = False | True ; data T = X Bool Bool` dieses T hat 4 Elemente, 3 Konstruktoren (False, True, X)

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

durch Induktion über den Term-Aufbau.

- Induktions-Anfang:  $t = f()$  (nullstelliges Symbol  $f$ )
- Induktions-Schritt:  
 $t = f(t_1, \dots, t_k)$  ( $k$ -stelliges Symbol  $f$ , für  $k > 0$ )  
dabei Induktions-Voraussetzung: die Behauptung gilt für  $t_1, \dots, t_k$ .  
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  $\leq 9$
- die Höhe  $0, 1, 2, 3, \dots, k, \dots$

Sie müssen diese Elemente nicht alle einzeln angeben.

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

## Aufgaben autotool

1. Ersetzen Sie `undefined`, so daß der Ausdruck `test` den Wert `True` hat.

```
module Blueprint where
import qualified Data.Set as S
-- aus Prelude importiert:
-- data Bool = False | True
data C = R | G | B deriving (Eq, Ord, Show)
data T = X C Bool | Y Bool | Z deriving (Eq, Ord, Show)
solution :: S.Set T
solution = S.fromList undefined
test :: Bool
test = S.size solution == 9
```

Ansatz einer Lösung:

```
solution = S.fromList [ X R False, Y True ]
```

## 2. einen Term in einer mehrsortigen Signatur angeben (Programmiersprache: Java)

Gesucht ist ein Ausdruck vom Typ `int`  
in der Signatur

```
Foo e;  
String g;  
static char a ( Bar x, String y, String z );  
static Bar b ( Foo x );  
static int c ( int x, String y );  
static int d ( char x, Foo y, String z );  
static Foo f ( int x );  
static String h ( Foo x, String y, char z );
```

Lösungsansatz

```
d ( a ( b ( e ), ... ), ... )
```

## 4 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  $t$ .
- $\text{Pos}(t)$  = die Menge der Positionen eines Terms  $t$   
Definition: wenn  $t = f(t_0, \dots, t_{k-1})$ , d.h.,  $k$  Kinder  
dann  $\text{Pos}(t) = \{[]\} \cup \{[i] \cdot p \mid 0 \leq i < k \wedge p \in \text{Pos}(t_i)\}$ .

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:  $S(f(S(S(Z))), Z())[0, 1] = \dots$   
Definition (durch Induktion über die Länge von  $p$ ):  $\dots$
- $t[p := s]$  : wie  $t$ , aber mit Term  $s$  an Position  $p$   
Beispiel:  $S(f(S(S(Z))), Z())[0, 1 := S(Z)] = \dots$   
Definition (durch Induktion über die Länge von  $p$ ):  $\dots$

## Operationen mit Variablen in Termen

- $\text{Term}(\Sigma, V)$  = Menge der Terme über Signatur  $\Sigma$  mit Variablen aus  $V$   
Beispiel:  $\Sigma = \{Z/0, S/1, f/2\}$ ,  $V = \{y\}$ ,  $f(Z(), y) \in \text{Term}(\Sigma, V)$ .
- Substitution  $\sigma$ : partielle Abbildung  $V \rightarrow \text{Term}(\Sigma)$   
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  $\rightarrow_R$  ist Menge aller Paare  $(t, t')$  mit
  - es existiert  $(l, r) \in R$
  - es existiert Position  $p$  in  $t$
  - es existiert Substitution  $\sigma : (\text{Var}(l) \cup \text{Var}(r)) \rightarrow \text{Term}(\Sigma)$
  - so daß  $t[p] = l\sigma$  und  $t' = t[p := r\sigma]$ .

## Termersetzungssysteme als Programme

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

dieses Berechnungsmodell ist im allgemeinen

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

## Konstruktor-Systeme

Für TRS  $R$  über Signatur  $\Sigma$ : 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:  $R_1 = \{a(b(x)) \rightarrow b(a(x))\}$  über  $\Sigma_1 = \{a/1, b/1\}$ ,

$R_2 = \{f(f(x, y), z) \rightarrow f(x, f(y, z))\}$  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  $\{f(Z, y) \rightarrow y, f(S(x'), y) \rightarrow S(f(x', y))\}$ .
  - 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  $\sigma$  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)

Beispiele (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)  

|                                |                                |
|--------------------------------|--------------------------------|
| <code>not x = case x of</code> | <code>not x = case x of</code> |
| <code>False -&gt; _</code>     | <code>{False -&gt; _</code>    |
| <code>True -&gt; _</code>      | <code>;True -&gt; _</code>     |
|                                | <code>}</code>                 |
- 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  $t \in \text{Term}(\Sigma)$  an mit  $t[1] = c$ .
- Beweisen Sie  $\forall \Sigma : \forall t \in \text{Term}(\Sigma) : |t| = |\text{Pos}(t)|$ .

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

- für welche Substitution  $\sigma$  gilt  $f(x, Z)\sigma = f(S(Z), Z)$ ?
- für dieses  $\sigma$ : 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 des Alphabetes ( $\dots, x, y, \dots$ ) 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 24: Aufgaben 1, 4, 5

### 1. Arithmetik auf Peano-Zahlen

- Für  $R = \{f(S(x), y) \rightarrow f(x, S(y)), f(Z, y) \rightarrow y\}$   
bestimme alle  $R$ -Normalformen von  $f(S(Z), S(Z))$ .
- für  $R_d = R \cup \{d(x) \rightarrow f(x, x)\}$   
bestimme alle  $R_d$ -Normalformen von  $d(d(S(Z)))$ .
- Bestimme die Signatur  $\Sigma_d$  von  $R_d$ .  
Bestimme die Menge der Terme aus  $\text{Term}(\Sigma_d)$ , die  $R_d$ -Normalformen sind.
- Welche Rechenoperationen simulieren die Regeln für  $f$ , für  $d$ ?
- welche Terme haben große Normalformen?
- Zusatz: Geben Sie eine Funktion  $c : \rightarrow \text{an}$ , für die gilt:  $\forall n \in \mathbb{N} : \forall t, t' \in \text{Term}(\Sigma_d) : |t| \leq n \wedge t \rightarrow_{R_d}^* t' \Rightarrow |t'| \leq c(n)$ .

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

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

- für  $\{A(B(x)) \rightarrow B(A(x))\}$   
über Signatur  $\{A/1, B/1, E/0\}$ :  
bestimme Normalform von  $A^k(B^k(E))$   
für  $k = 1, 2, 3$ , allgemein.
- für  $\{A(B(x)) \rightarrow B(B(A(x)))\}$   
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  $|t_i|, \text{depth}(t_i)$ .
- für  $S = \{A(A(D, x), y) \rightarrow A(x, A(x, y))\}$   
bestimme  $S$ -Normalform(en), soweit existieren, der Terme  $t_0, t_1, \dots, t_4$ .  
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!

## Aufgaben autotool

1. gesucht ist für das System

```
TRS { variables = [ x, y, z ]  
    , rules =  
      [ f ( f ( x, y ), z ) -> f ( x, f ( y, z ) )  
        , f ( x, f ( y, z ) ) -> f ( f ( x, y ), z )  
      ]  
    }
```

eine Folge von Schritten

von  $f ( a, f ( f ( b, f ( c, d ) ), e ) )$

nach  $f ( f ( a, f ( b, c ) ), f ( d, e ) )$

Lösungs-Ansatz:

```
( f ( a, f ( f ( b, f ( c, d ) ), e ) )  
, [ Step { rule_number = 1 , position = [ ]  
        , substitution = listToFM  
          [ (x, a), (z, e), (y, f (b, f (c, d))) ]  
        }  
      ] )
```

## 5 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  $x + y = (1 + x') + y = \dots$

## **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  $\text{plus } a \ (\text{plus } b \ c) == \text{plus } (\text{plus } a \ b) \ c$
- Beweismethode: Induktion (nach  $a$ )  
und Umformen mit Gleichungen (äquiv. zu Implement.)

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

- Anfang:  $\text{plus } Z \ (\text{plus } b \ c) == \dots$
- Schritt:  $\text{plus } (S \ a') \ (\text{plus } b \ c) ==$   
 $== S \ (\text{plus } a' \ (\text{plus } b \ c)) == \dots$

## Bezeichnungen in Beweisen durch Induktion

- Es ist  $\forall t \in \text{Term}(\Sigma) : P(t)$  zu zeigen  
 $P$  gilt für alle Terme der Signatur  $\Sigma$ .
- 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  $t = f(t_1, \dots, t_n)$  mit  $n \geq 1$ ,  
d.h., innere Knoten

- \* (IV) Induktions-Voraussetzung:  
 $P(t_1) \wedge \dots \wedge P(t_n)$ , 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.  $\text{plus}' x' y \text{ .=. plus } x' y$

- Beweis der Ind.-Behauptung benötigt Umformung von  $\text{plus}' x' (S y)$ . 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:  $g :: \text{Tree} \rightarrow \text{Bool}$  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

$$\begin{aligned} \min Z Z = \_ ; \min Z (S y) = \_ ; \min (S x) Z = \_ \\ \min (S x) (S y) = S (\min x y) \end{aligned}$$

- 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:

$$\begin{aligned} - \forall a, b \in: (a + b) \ominus b = a \\ - \forall a, b \in: a \ominus (a + b) = 0 \end{aligned}$$

- Implementierung (Muster disjunkt? vollständig?)

$$\begin{aligned} \text{minus } Z b = \_ ; \text{minus } a Z = \_ \\ \text{minus } (S a') (S b') = \_ \end{aligned}$$

## Hausaufgaben

SS 24: Aufgaben 2, 4 von hier sowie Aufgabe 4 aus Abschnitt *Daten* ( $\forall \Sigma : \forall t \in \text{Term}(\Sigma) : \text{height}(t) \leq |t| - 1$ )

### 1. Für die Funktion

$$\begin{aligned} f :: N \rightarrow N \\ f Z = Z ; f (S x) = S (S (f x)) \end{aligned}$$

beweisen Sie (erst auf Papier, dann mit CYP)

$$f (\text{plus } x y) = \text{plus } (f x) (f y)$$

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

$\text{plus } Z \ y = y ; \text{ plus } (S \ x) \ y = S \ (\text{plus } x \ y)$

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

$\text{plus}' \ Z \ y = y ; \text{ plus}' \ (S \ x) \ y = \text{plus}' \ x \ (S \ y)$

Beweisen Sie mit Cyp

$\text{forall } x :: N, y :: N : \text{plus}' \ x \ y \ . = . \ \text{plus } x \ y$

Beweisen Sie dazu als Hilfssatz

$\text{forall } x :: N, y :: N : \text{plus } x \ (S \ y) \ . = . \ \text{plus } (S \ x) \ y$

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

(a) die Menge (Folge) aller  $R$ -Normalformen ist  $N_0 = D, N_1 = A(D, N_0), \dots, N_{k+1} = A(D, N_k), \dots$

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?