Caokesisches Seminar 26.10.21

Formale Spezifikation Primzahltest:

$p : \mathbb{N} \longrightarrow \mathbb{B}$

ist Primzahltest, falls

$\mathbb{N}$ nat. Zahlen

$\mathbb{B} = \{true, false\}$

"Bool"

$\forall n \in \mathbb{N} . \; p\,n = true \iff n \; prim$

Aufgabe: Wie sieht eine Spezifikation für eine Funktion aus, die die Quadratwurzel berechnet?

Beispiel für Programm, das sich an der Input-Struktur orientiert:

```
length : List A → ℕ
length []       = 0
length (a :: as) = 1 + length as
```

A ist beliebiger Typ

1. Fall: leere Liste

2. Fall: nicht leere Liste, besteht aus einem "head" (hier a), vom Typ A und einem tail (as) vom Typ List A

```
f : A → (B → C)
f a b = (f a) b
```

```
data List A = [] | A :: List
```

↑ Haskell-Notation, sehr dicht ...

oder:

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

etwas detailliertes:

```
data List : Type → Type where
    Nil : List A
    Cons : A → List A → List A
```

```
N = List 1
    ?
```

Könnte man length : List A → ℕ
auch corekursiv definieren?

Fuge für spätere 5

2.11.21

data 1 where

* : 1

Funktion $f : 1 \to A$ (Abelschen Typ)

wird definiert durch

$f * = a$ mit einem (bel.)

$a : A$

Nicola: Es ist interessant zu fragen, wieviele Funktionen (z.B. vom Typ $1 \to A$ es gibt)

$\mathbb{N}$ = List 1

Terme vom Typ List 1 :

[] , (* :: []) , * :: (* :: []) , ...

" " "

[] , [*] , [*,*] , [*,*,*]

---

1 Introduction

"h is to process a list..." ⇝

$h : List\ A \to B$ mit gegebenen Typ A und B

Beispiele: length : List A → $\mathbb{N}$

id : List A → List A

scanl : List ℕ → List ℕ wie

z.B. scanl [1,2,3] = [1,3,6] ...

Frage Tweek: Seite 1. verstehen die Punkte in

h[] = ...

h(a:x) = ... a ... x ... hx ...

z.B. h : List A → List A h ≙ g() g : List A

① h[] = [] "="

h(a:x) = x "=" ... a ... x ... hx ...

≙ g(a, x, hx) g : A → List A → List A → List A

g a x y := x

vgl. Rekursions- und Induktionsprinzipien
in Type theorie

Seite 2, "However ..."
Situation jetzt: h: A → List B
List Typ Anwahl B.

Beispiel: Trajektorie eines "dynamischen Systems":
Seien A ein Typ, a: A, f: A → A

traj: ℕ → List A

z.B. traj 3 = [a, fa, f(fa)]

Seien A ein Typ, $a : A$, $f : A \to A$

traj : $\mathbb{N} \to$ List A

2.B. traj 3 = $\lambda a.\, f a.\, f(f a)$

```
traj n | n ≡ 0        = []
       | otherwise    = b :: y
  where b = a
        y = map f (traj (n-1))
```

↑ ist eigentlich 'verkappte Induktion' in dieser Bsp.

Sebastian: gibt es auch "gute" Beispiele?

9.11.21

primesf : 1 → list Int

primes : list Int

Nicola: generische Programmierung, Iterators,
Cin Java, C++, Python (...)

primesf x | false = []
| otherwise (always) = (b : y)
where b = 2
y = [3, 5, 7, 11 ...]

echter cowhiles. Aufruf nicht möglich

Nicola: Alle iterativen Algorithmen sind Beispiele dieses Schemas (können so geschrieben werden). Ganz einfaches Beispiel:

```
repeat : A × ℕ → List A
repeat (a, n) | n ≤ 0 = []
              | otherwise = (b :: y)
  where b = a
        y = repeat (a, n-1)
```

(n-1 ← next n)

```
forever : A → List A   (← ejection Stream)
forever a = a :: forever a
```

insertSort : List ℤ → List ℤ

Welche Eigenschaft hat

insertSort xs (für jede beliebige Liste xs)

.) aufsteigend geordnet

.) ist Permutation von xs

.) insertSort (PureInSort xs) = insertSort xs

16.11.21

Frage: Kann man insert auch über pattern matching im ersten Argument definieren?

insert Z xs = Z :: xs

insert (S n) xs = ... ?

↑

Können jedenfalls auf pattern matching über xs nicht verzichten!

```
data Nat
  Z : Nat
  S : Nat -> Nat
```

=> da insert generischen definiert sein sollte,

insert : {A : Set} → (A → A → B) →

A → List A → List A

Verallgemeinert
≤ : N → N → B

ist Def. über die Struktur von A

keine gute Idee, denn im Allgemeinen

wissen wir nichts über die Struktur von A!

Wir haben insertSort [3,2,1] = [1,2,3]

mittels "equational reasoning" bewiesen ...

Haben über "=" und Refl gesprochen.

Es gilt: Refl : insertSort [3,2,1] = [1,2,3]

da insertSort [3,2,1] zu [1,2,3] "reduziert" wird.

data Date : Set where

MkDate : Day → Month → Year → Date

day : Date → Day
day (MkDate d _ _) = d

year --
month --

record Date : Set where
  constructor MkDate
  field(s)
    day : Day
    month : Month
    year : Year

7.12.

Selectsort diskutieren.

let vs. where

syntakt. vs. analytical

referential transparency (vs. opaque / transparent)

Implementierung

setzt "gute" - Beschreibung

direkt um!

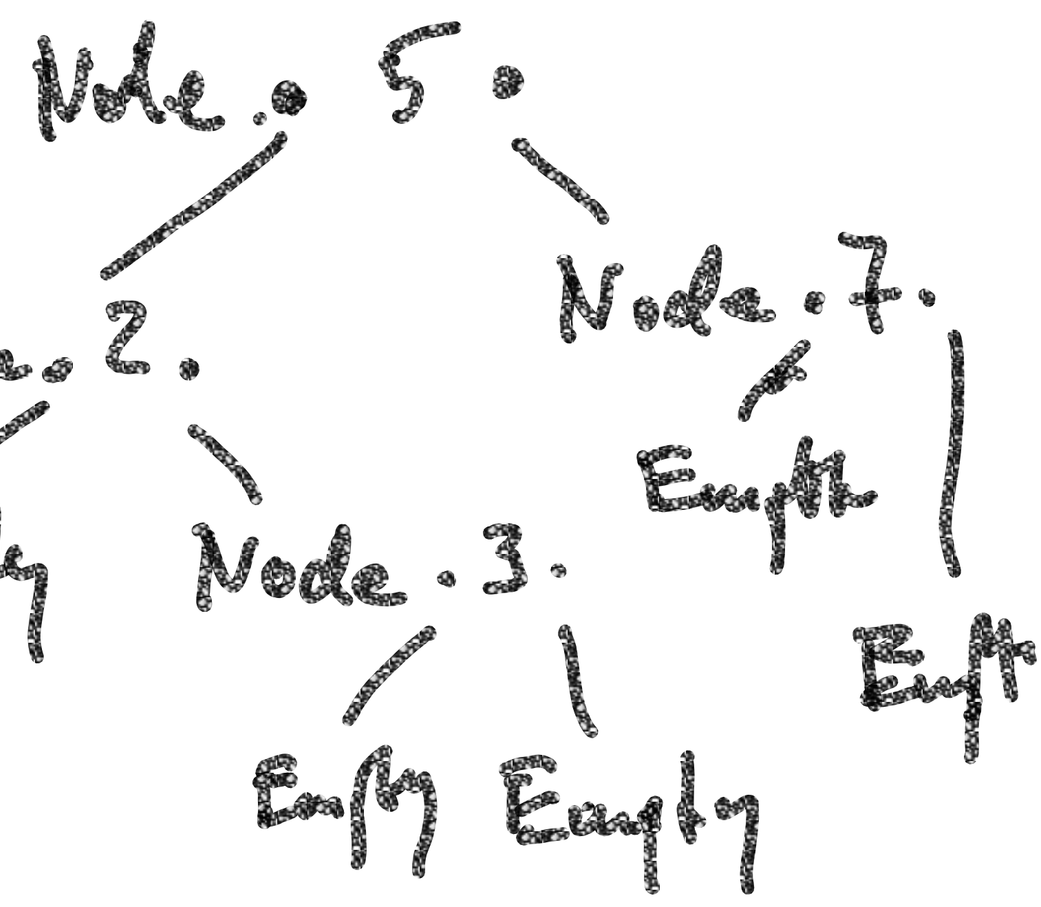
---

quicksort :: [Integer] -> [Integer]

quicksort = flatten . build

data NTree ...

z.B.

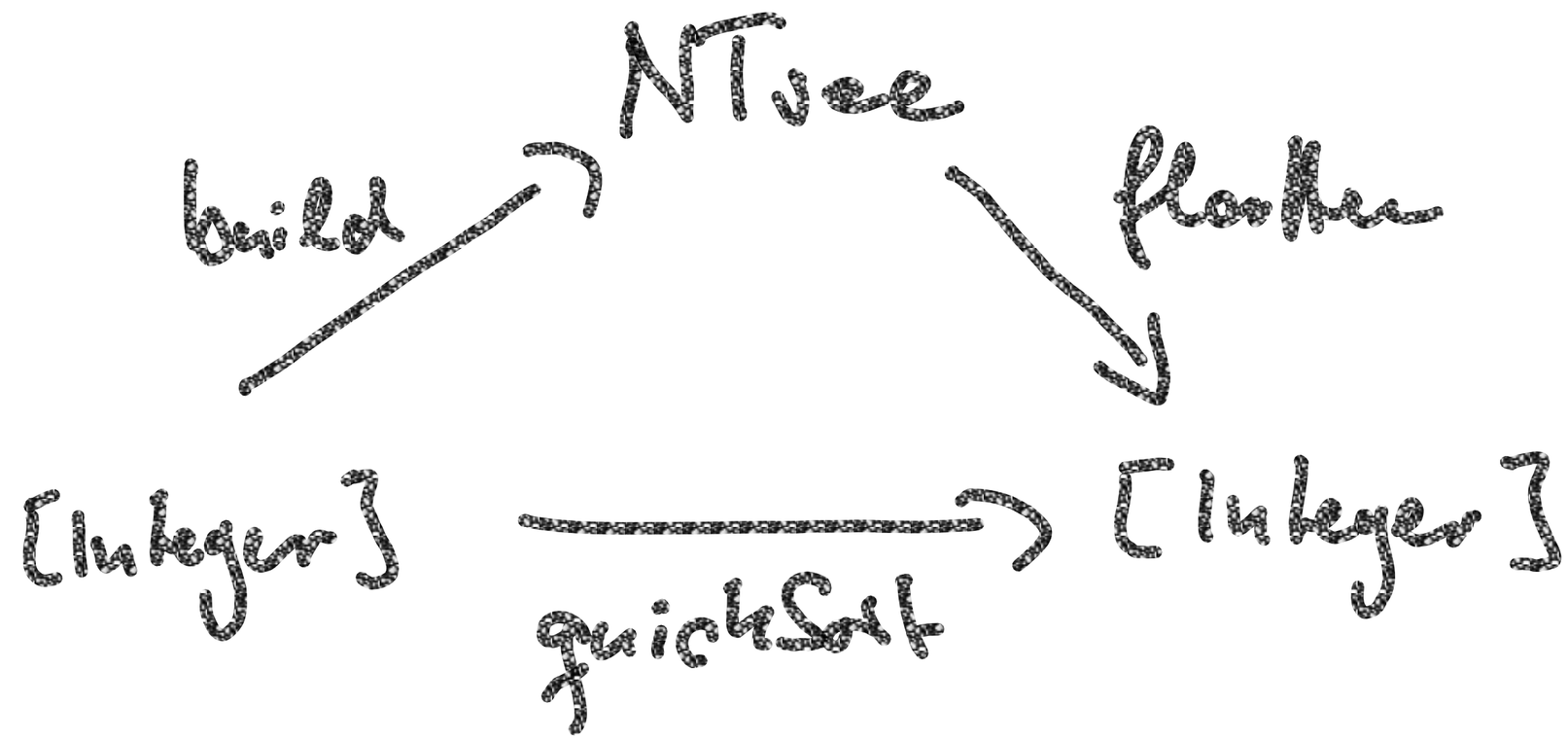


sieht so aus ↓

build [5,2,7,3]

flatten : NTree → [Integer]

build : [Integer] → NTree



_∘_ : {A B C : U} → (B → C) → (A → B) → A → C

↑
Set

: Π Π Π (B→C) → (A → B) → A → C
A:U B:U C:U

"HoTT"-Syntax

```
data NTree : U where
  Empty : NTree
  Node : NTree → Integer → NTree → NTree
```

flatten is defined by "pattern matching"

```
flatten : NTree → [Integer]
flatten Empty = []
flatten (Node l a r) = flatten l ++ [a] ++ flatten r
```

$$rec_{NTree} : \prod_{C:\mathcal{U}} C \to (NTree \to Integer \to NTree \to C) \to NTree \to C$$

↑ leider nein

C

Problem!

```
flatten = rec_NTree [Integer] [] (λ l a r .
                                  flatten l ++ [a] ++ flatten r )
```

$$rec_{NTree} : \prod_{C : U} C \to (C \to Nat \to C \to C) \to NTree \to C$$

$$flatten = rec_{NTree}\ [Nat]\ []\ (\lambda c_l\ a\ c_r \cdot c_l ++ [a] ++ c_r)$$

computation rules for rec_{NTree}

$$rec_{NTree}\ C\ c_0\ c_s\ Empty \equiv c_0$$

$$rec_{NTree}\ C\ c_0\ c_s\ (Node\ l\ a\ r) \equiv$$

$$c_s\ (rec_{NTree}\ C\ c_0\ c_s\ l)\ a\ (rec_{NTree}\ C\ c_0\ c_s\ r)$$

14.12.21. Wrap-up (lookup fold/unfold)

Nicola: Abschnitt S.9 über Hylomorphismus.

deforestation, virtual data ... bringt wichtige Verbindung

Sebastian: Monadisch "saubere" Lösung zuerst.

Optimierungen später (vllt. automatisch vom compiler ...)

Nicola: unix-pipes

... Prinzip Modularität

als Leitfaden

wichtig für Library design.

ones : List ℕ

ones = 1 :: ones

take 1000 ones

↳ [1, 1, 1, ... , 1]
1000 mal ...

if (true ⇔ halts q;) then "yes" else "no"
will any evaluation
= "yes"

ohne
= ... Gen Explosion