Praktische Informatik 3: Funktionale Programmierung

Vorlesung 6 / 28. November 2023

Datentypen, Typklassen, zyklische Strukturen

Thomas Barkowsky

Wintersemester 2023/24

Organisatorisches

- Anmeldung für Probeklausur in Stud.IP nur noch bis morgen!

- Zwischenevaluation in Stud.IP (läuft bis 10. Dezember)

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- Testen und Qualitätssicherung
- I/O, Aktionen und Zustände
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung…

- Algebraische Datentypen
  - `newtype`
  - *Records* in Haskell
- Typklassen und Instanzen
  - `class`, `instance`
  - `data` … `deriving` …
- Rekursive und zyklische Datenstrukturen
  - Traversion von Labyrinthen
  - variadische Bäume

`newtype` und *Records*

`newtype`: Zwischen `type` und `data`

- Vergangene Woche: `type` und `data`
  - `type`: Synonyme existierender Typen
  - `data`: neue Algebraische Datentypen durch Summen und kartesische Produkte
- Für neue Datentypen mit **einem** Konstruktor und **einem** Argument: `newtype`
- Beispiel: Natürliche Zahlen: `newtype Nat = N Int`
  (Nicht-Negativität ist gesondert sicherzustellen)
- Unterschied zu `type Nat = Int` und `data Nat = N Int` ?

`type`, `data` und `newtype`

- `type`: nur Synonym, kein eigener Typ (Bsp.: Strings)
- `data` und `newtype` generieren neue, unabhängige Typen
  - klare Trennung von Typen
- `newtype`: Effizienzgewinn, da Typkonstruktoren nach Typüberprüfung durch Compiler ersetzt werden
- Fazit: `newtype` = Typsicherheit mit hoher Performanz
- oder auch: `newtype` ‘versteckt’ den tatsächlichen Typ

Zurück zu `data`: Records in Haskell

- Für algebraische Datentypen werden **Selektoren** benötigt
- Beispiel (aus letzter Vorlesung):

```
data Date = Datum Year Month Day
```

```
type Year  = Int
data Month = Jan | Feb | Mar | ...
type Day   = Int
```

```
day :: Date -> Day
day (Datum y m d) = d

month :: Date -> Month
month (Datum y m d) = m
```

Selektoren für algebraische Datentypen…

- … selbst definieren? Das muss nicht sein!
- Die Lösung: *Record* Syntax!

```
data Date = Datum Year Month Day
```

```
data Date = Datum { year  :: Year
                  , month :: Month
                  , day   :: Day}
```

- Und was bringt das?

Records: Selektoren und Konstruktion

- Was bringt das?
  - Selektoren *for free*:

```
> month (Datum 2019 Nov 19)
Nov
> :t month
month :: Date -> Month
```

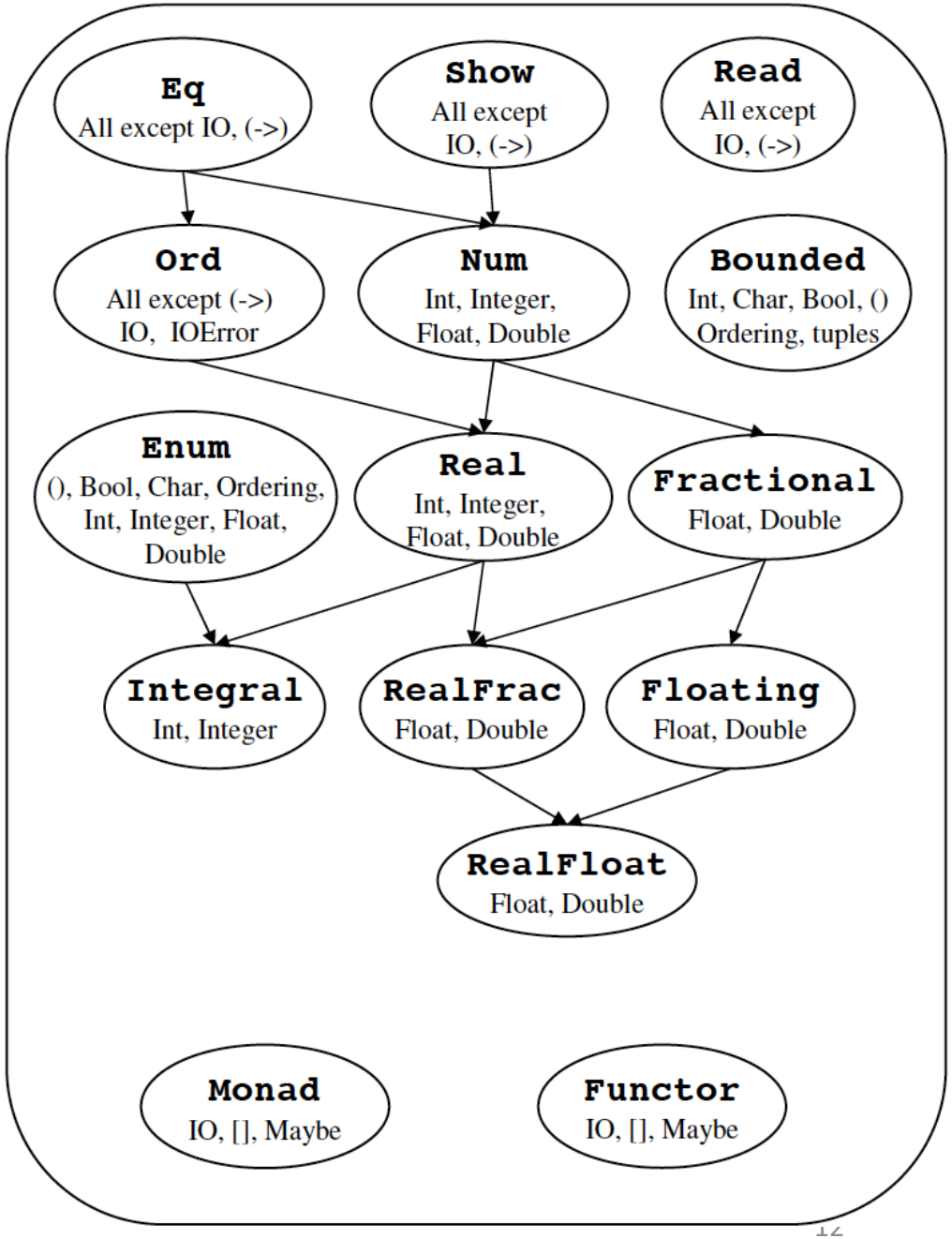
  - ebenso für `day`, `year`
  - beliebige Reihenfolge der Daten bei der Konstruktion:

```
> Datum 2019 Nov 19 == Datum {day=19, month=Nov, year=2019}
True
```

Definition von Typklassen und Instanzen

Typklassen bisher

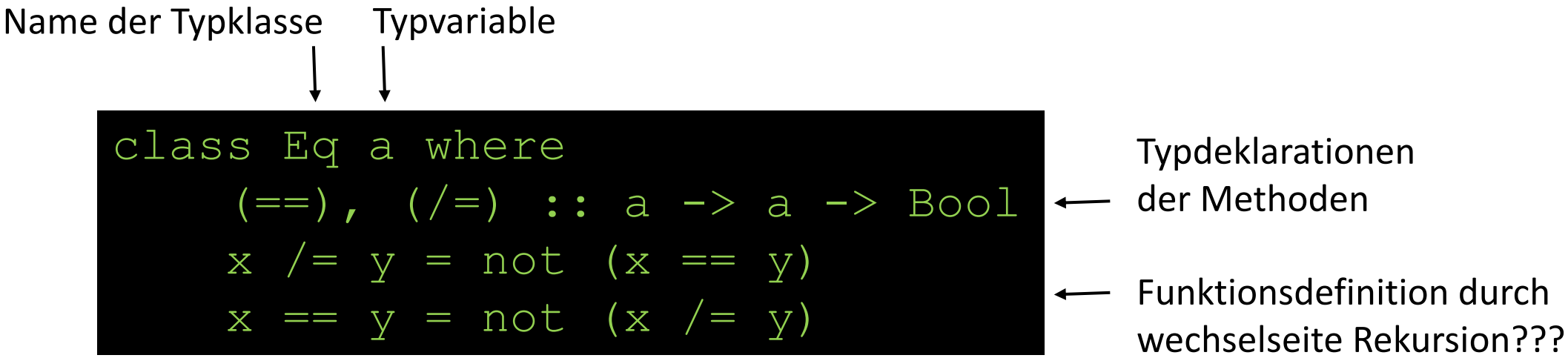
- `Eq`, `Ord`, `Show`, `Read`, `Num`, `Integral`, …
- Typklassen
  - hierarchische Organisation
  - definieren Verhalten von Datenypen, wie Test auf Gleichheit, Anordnung, etc.
  - ermöglichen überladene Funktionen



Haskell 2010 Language Report, p. 76

Definition von Typklassen: `class`

- Beispiel `Eq` (*standard prelude*):



Bedeutung der Typklassen-Deklaration

- Ein Typ `a`, der zur Klasse `Eq` gehört, muss Operationen der Gleichheit und Ungleichheit unterstützen, gemäß der vorgegebenen Typdeklaration

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)
```

- Definitionen von $=$ und $/=$ nur *default definitions*
- Da $=$ und $/=$ durch wechselseitige Rekursion definiert sind, muss mindestens eine der beiden Operationen in der Deklaration einer **Instanz** von `Eq` definiert sein
- Beispiel...

`Bool` als Instanz von `Eq`

- Definition von `Bool`:

```
data Bool = False | True
```

- Deklaration als Instanz von `Eq`:

```
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
```

- Ungleichheit ist durch *default definition* in `Eq` geregelt
  - Gleichheit hier überschrieben
  - **minimal vollständige Definition** von `Bool` als Instanz von `Eq`
- Nur mit `data` oder `newtype` definierte Datentypen können Typklassen zugeordnet werden

weiteres Beispiel…

```
data Ampel = Rot | Gelb | Gruen
```

```
instance Show Ampel where
  show Rot   = "Rot: Stop!"
  show Gelb  = "Gelb: Achtung!"
  show Gruen = "Gruen: Go!"
```

Unterklassen von Klassen definieren

- Beispiel: `Ord` als Unterklasse von `Eq` (unvollständig):

```
class (Eq a) => Ord a where
   (...)
   (<), (<=), (>), (>=) :: a -> a -> Bool
   min, max             :: a -> a-> a
   (...)
```

- 6 weitere Operatoren
- `max` und `min` definiert auf Basis von <= und >=
  - daher gesonderte Definition von `max` und `min` in Instanzdeklarationen nicht erforderlich

Abkürzung zu Typklassendeklarationen

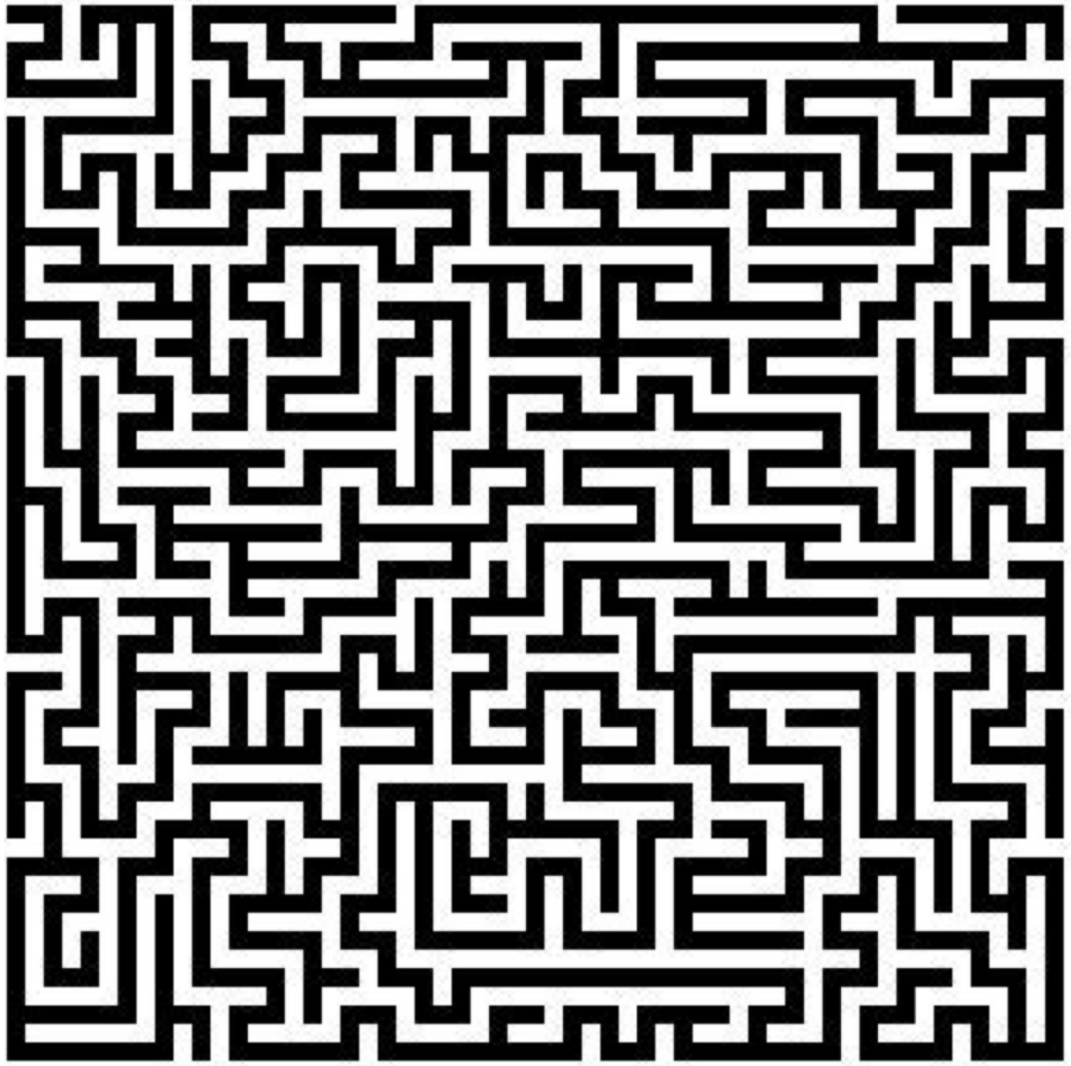
- In der Regel wollen wir, dass neue Typen gleich in einige Standardklassen aufgenommen werden
  - z.B. `Eq`, `Ord`, `Show`, …
- Die Lösung: `deriving`
  - Optional in `data` und `newtype` Definitionen
- Beispiel: `Bool` (so in *standard prelude* definiert):

```
data Bool = False | True
          deriving (Eq, Ord, Show, Read)
```

- oder:

```
data Ampel = Rot | Gelb | Gruen deriving Show
```

  - … dann natürlich nur Standardausgabe als String



Quelle: docs.gimp.org

Rekursive und zyklische Datenstrukturen: *Beispiel Labyrinth*

(von Christoph Lüth)

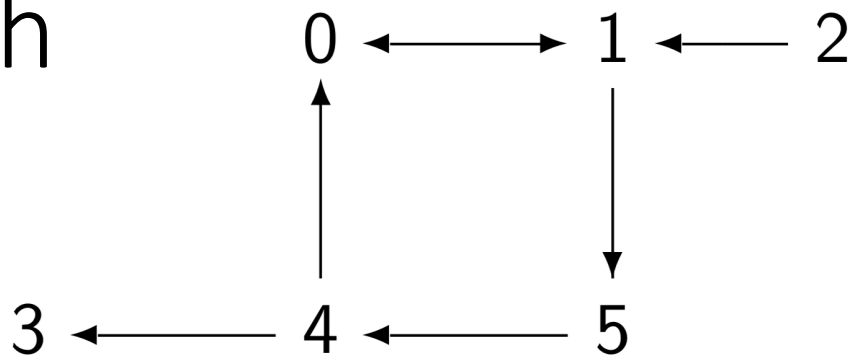
Labyrinth: Modellierung

- Ein (gerichtetes) Labyrinth ist entweder
  - eine Sackgasse oder
  - ein Weg oder
  - eine Abzweigung (zwei Richtungen)
- Jeder Knoten hat ein Label des Typs `a`

- Datentyp:

```
data Lab a = Dead a
           | Pass a (Lab a)
           | TJnc a (Lab a) (Lab a)
```

Beispiel: einfaches Labyrinth



- Rekursive Definition in Haskell:

```
lab0  = Pass 0 lab1
lab1  = TJnc 1 lab0 lab5
lab2  = Pass 2 lab1
lab3  = Dead 3
lab4  = TJnc 4 lab0 lab3
lab5  = Pass 5 lab4
```

- Ausgabe in GHCi (eigene Instanz der Typklasse `Show`):
- Labyrinth nicht zyklenfrei!

```
*LabExamples> lab2
2 --> 1
1 --> 0 5
0 --> 1
5 --> 4
4 --> 0 3
3 -->
```

Traversion von Labyrinthen

- Traversion = Pfad zu einem gegebenen Zielknoten finden
- Ein Pfad ist eine Liste von Knoten:

```
type Path a = [a]
```

- Eine Traversion ist entweder ein erfolgreicher Pfad oder ein Misserfolg:

```
type Trav a = Maybe [a]
```

Traversionsstrategie

- Annahme zunächst: Labyrinth ist zyklenfrei
- An jedem Knoten prüfen: Ziel erreicht? Falls nicht:
  - an Sackgassen: Fehlschlag (`Nothing`)
  - an Wegen: Weiterlaufen
  - an Abzweigungen: Auswahl treffen (`select`)

Traversionsstrategie

```
traverse_1 :: Eq a => a -> Lab a -> Trav a
traverse_1 t l
  | nid l == t = Just [nid l]
  | otherwise = case l of
    Dead _ -> Nothing
    Pass i n -> cons i (traverse_1 t n)
    TJnc i n m -> cons i (select (traverse_1 t n)
                                 (traverse_1 t m))
```

- mit

```
nid :: Lab a -> a
nid (Dead i) = i
nid (Pass i _) = i
nid (TJnc i _ _)= i
```

Traversionsstrategie

- Propagation von Fehlschlägen in

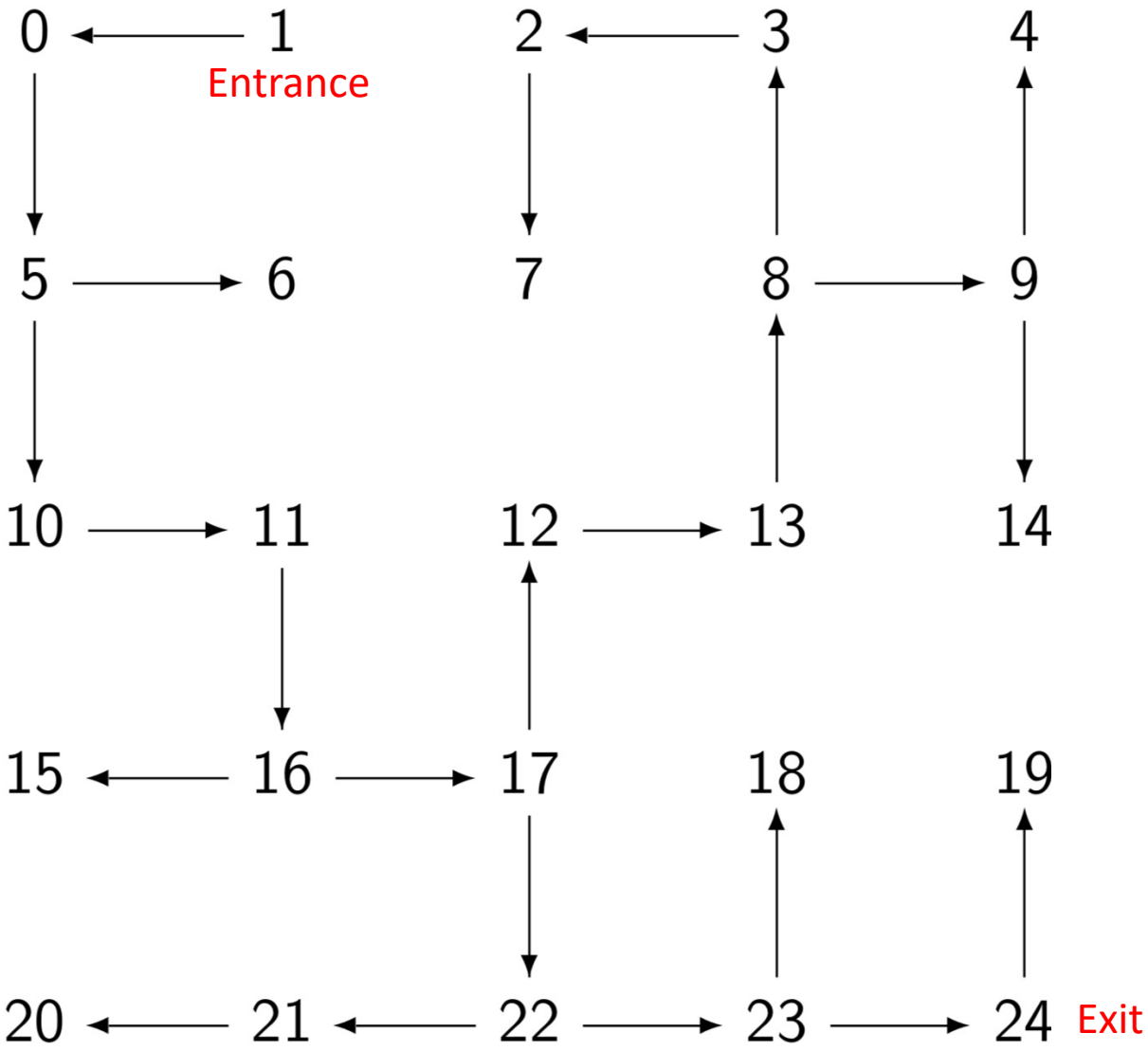
```
cons :: a -> Trav a -> Trav a
cons _ Nothing       = Nothing
cons i (Just is) = Just (i: is)
```

und

```
select :: Trav a -> Trav a-> Trav a
select Nothing t = t
select t     _ = t
```

- Beispiel: zyklenfreies Labyrinth...

Zyklenfreies Labyrinth




```
l00 = Pass 0 l01
l10 = Pass 1 l00 -- "Entrance"
l20 = Pass 2 l21
l30 = Pass 3 l20
l40 = Dead 4
l01 = TJnc 5 l02 l11
l11 = Dead 6
l21 = Dead 7
l31 = TJnc 8 l30 l41
l41 = TJnc 9 l40 l42
l02 = Pass 10 l12
l12 = Pass 11 l13
l22 = Pass 12 l32
l32 = Pass 13 l31
l42 = Dead 14
l03 = Dead 15
l13 = TJnc 16 l03 l23
l23 = TJnc 17 l22 l24
l33 = Dead 18
l43 = Dead 19
l04 = Dead 20
l14 = Pass 21 l04
l24 = TJnc 22 l14 l34
l34 = TJnc 23 l33 l44
l44 = Pass 24 l43 --    "Exit"
```

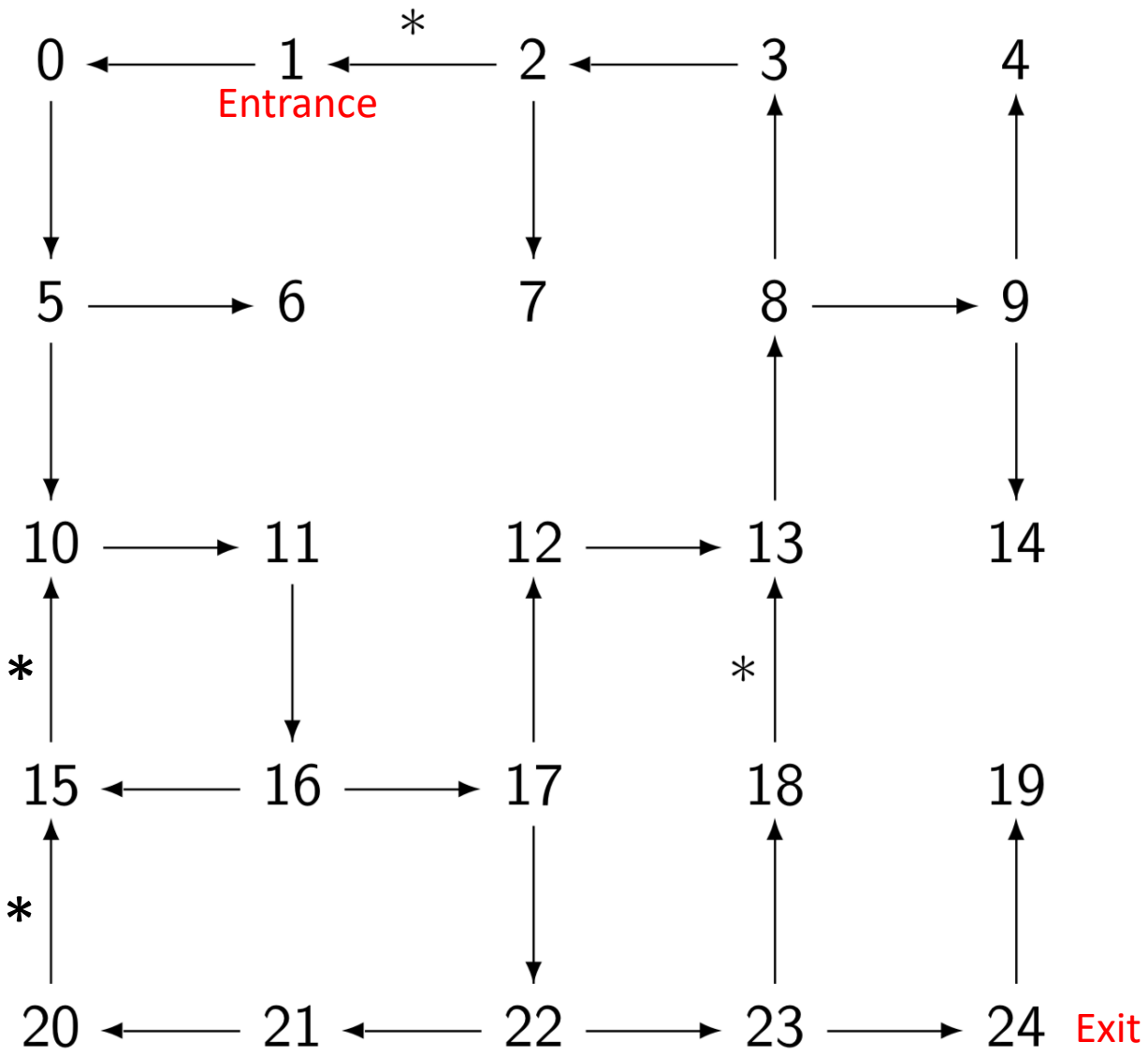
Traversion mit Zyklen

- An jedem Knoten prüfen, ob schon im Pfad enthalten
- Dazu bereits konstruierten Pfad in jedem Schritt übergeben
  - Pfad muss hinten erweitert werden: *O(n)*
  - besser: Pfad vorn erweitern (*O(1)*) und am Ende umdrehen
- Wenn aktueller Knoten bereits im Pfad enthalten: Fehlschlag!
- Sonst alles wie in erster Lösung...

Traversion mit Zyklen

```
traverse_2 :: Eq a => a -> Lab a -> Trav a
traverse_2 t l = trav_2 l [] where
  trav_2 l p
    | nid l == t = Just (reverse (nid l: p))
    | elem (nid l) p = Nothing
    | otherwise = case l of
      Dead _ -> Nothing
      Pass i n -> trav_2 n (i: p)
      TJnc i n m -> select (trav_2 n (i: p))
                           (trav_2 m (i: p))
```

Labyrinth mit Zyklen




```
m00 = Pass 0 m01
m10 = Pass 1 m00   -- "Entrance"
m20 = TJnc 2 m10 m21--Introduces a cycle
m30 = Pass 3 m20
m40 = Dead 4
m01 = TJnc 5 m02 m11
m11 = Dead 6
m21 = Dead 7
m31 = TJnc 8 m30 m41
m41 = TJnc 9 m40 m42
m02 = Pass 10 m12
m12 = Pass 11 m13
m22 = Pass 12 m32
m32 = Pass 13 m31
m42 = Dead 14
m03 = Pass 15 m02  -- Introduces a cycle
m13 = TJnc 16 m03 m23
m23 = TJnc 17 m22 m24
m33 = Pass 18 m32  -- Introduces a cycle
m43 = Dead 19
m04 = Pass 20 m03  -- Introduces a cycle
m14 = Pass 21 m04
m24 = TJnc 22 m14 m34
m34 = TJnc 23 m33 m44
m44 = Pass 24 m43 -- "Exit"
```

Verallgemeinerung: Variadische Bäume

- Ein Labyrinth ist ein **Graph** oder ein **Baum**
- Labyrinth mit (potentiell) mehr als 2 Nachfolgern pro Knoten: **variadischer Baum**

```
data VTree a = NT a [VTree a]
```

- Kürzere Definition ermöglicht einfachere Funktionen...

Traversion im variadischen Baum

```
traverse :: Eq a => a -> VTree a -> Maybe [a]
traverse t vt = trav vt [] where
  trav (NT l vs) p
    | l == t = Just (reverse (l: p))
    | elem l p = Nothing
    | otherwise = select (travList (l: p) vs)
  travList p [] = []
  travList p (nt: nts) = trav nt p : travList p nts
```

```
select :: [Maybe a]-> Maybe a
select []             = Nothing
select (Just a: _)    = Just a
select (Nothing: os) = select os
```

Zusammenfassung

- `newtype`: `data` *“light”*
- *Record* Syntax: Selektoren *for free*
- Definition von Typklassen, Unterklassen und Instanzen
  - `class` und `instance`
- Zuordnung von Datentypen zu Standardklassen mit `deriving`
- Fallbeispiel zu rekursiven und zyklischen Datenstrukturen: Labyrinthe und variadische Bäume

nächstes Mal...

- Funktionen höherer Ordnung II
  - Funktionskomposition, die zweite...
  - Der Funktionsanwendungsoperator `$`
  - Funktoren: Verallgemeinerung von `map`
  - Faltung verallgemeinert (Listen falten kann ja jeder...)