Programming Paradigms

Summer Term 2017

6th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Infinite lists

- In Haskell there are even abbreviating notations for infinite lists.

[1, 3 ..]   means   [1, 3, 5, 7, 9, ...]

- For example:

```
naturals, evens, odds :: [Integer]
naturals  =  [ 1 .. ]
evens     =  [ 2, 4 .. ]
odds      =  [ 1, 3 .. ]
```

- With this we can represent infinite series as lists, e.g.,

```
squares  =  [ n^2 | n ← naturals ]
facs     =  [ fac n | n ← naturals ]
primes   =  2 : [ n | n ← odds, factors n == [1, n] ]
```

Actually working with infinite lists

- Input of an expression that denotes an infinite list expectedly leads to non-terminating output (needs to be stopped "by hand"!)

- However, working with finite parts of infinite lists is possible, e.g.,

```
> take 5 primes
[2, 3, 5, 7, 11]
```

```
> primes !! 5
13
```

- That this is possible is not trivial. It is a benefit of Haskell's on-demand evaluation strategy, which computes the value of a (sub-)expression only if, and when, it is absolutely required ("lazy evaluation").

- The following expression "intuitively" denotes a finite list, but the computation does nevertheless not terminate:

Why ?

```
> [ x | x ← squares, x < 100 ]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
```

Some more examples: variants for prime number generation

- Instead of:

```
odds       =  [ 1, 3 .. ]
factors n  =  [ x | x ← [1 .. n], n `mod` x == 0 ]
primes     =  2 : [ n | n ← odds, factors n == [ 1, n ] ]
```

- For example:

```
primes      =  2 : [ n | n ← [ 3, 5 .. ], isPrime n ]
isPrime n  =  and [ n `mod` t > 0 | t ← candidates primes ]
   where  candidates (p : ps) | p * p > n   = [ ]
                              | otherwise  = p : candidates ps
```

- Or also:

```
primes          =  sieve [ 2 .. ]
sieve (p : xs)  =  p : sieve [ x | x ← xs, x `mod` p > 0 ]
```

Programming Paradigms

The role of recursion (and kinds of recursion)

Pattern matching + recursion vs. list comprehensions

We earlier saw:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

```
> sumsquare 4
30
```

But also possible:

```
sumsquare :: Int → Int
sumsquare n  =  sum [ i * i  |  i ← [0 .. n] ]
```

```
> sumsquare 4
30
```

So which form is “better”?

No obvious/general answer. What could be criteria?

Maybe:

- efficiency
- readability
- “provability”

Fact: also sum, [0 .. n], … are ultimately defined via recursive functions.

Different kinds of recursion

Structural recursion:

```
sum :: [Int] → Int
sum [ ]         =  0
sum (x : xs)  =  x + sum xs
```

Also “structural” in some sense, or at least inductive:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

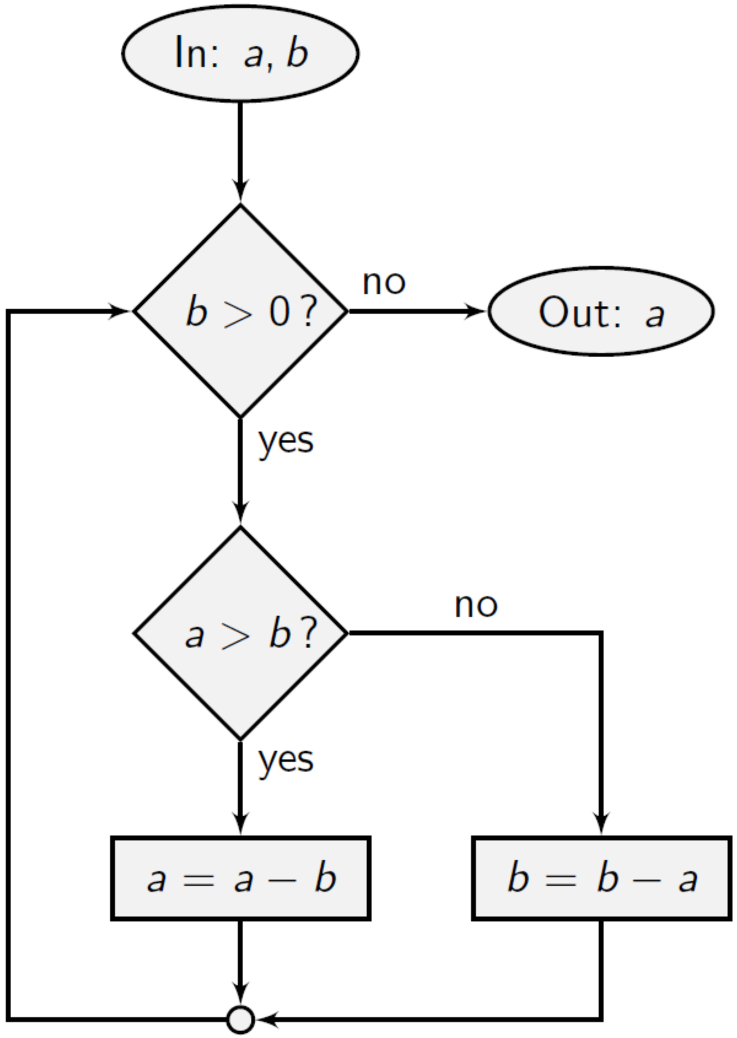
General/arbitrary recursion:

```
digsum :: Int → Int
digsum n  | n < 10          =  n
          | otherwise       =  let (d, m) = n `divMod` 10  in  m + digsum d
```

Also: ack, …, Quicksort, …

Another example for general recursion

Consider Euclid's algorithm:



euclid :: Int → Int → Int
euclid a 0 = a
euclid a b | a > b = euclid (a – b) b
euclid a b = euclid a (b – a)

- Loops (e.g., while) turn into recursive functions.
- Here even special form: tail recursion.
- How does this play out for verification?

Comparison structural and general recursion

- General recursion is much more flexible!
  - Algorithmic principles like “divide and conquer” can be employed.
  - Some functions can <u>provably</u> not be implemented with structural recursion.

- Structural recursion:
  - … gives a very useful “recipe” for defining functions
  - … guarantees termination (on finite structures)
  - … enables very direct proofs by induction
  - … can be “packaged” as a reusable program scheme

Programming Paradigms

Types in Haskell

Types

- Important concept of Haskell, so far considered only in passing:

  Every expression and every function have a type.

- Notation for type assignment: double colon

  e.g., 1 :: Int

- Foundation: predefined base types for constants
  - diverse numeric types, e.g., Integer, Rational, Float, Double
  - characters: Char
  - Boolean values: Bool

- Additionally: various type constructors (tuples, lists, …) for more complex types

Typing, type checking, type inference

- Every expression has exactly one type, which is determined before runtime:

  Haskell is a strongly and statically typed language.

- Function definitions and applications are checked for type consistency:

  type checking

- In addition, Haskell offers type inference, i.e., the types need not necessarily be written down explicitly.

- There is no (implicit or explicit) casting between types.

Particulars on typing of numbers

- We have already mentioned various number types: Int, Integer, Float (and there are several further ones, for example Rational).

- Number literals can have a different concrete type depending on context (e.g., 3 :: Int, 3 :: Integer, 3 :: Float, 3.0 :: Float, 3.5 :: Float, 3.5 :: Double).

- For general expressions there are overloaded conversion functions, for example:
  - fromIntegral :: Int → Integer, fromIntegral :: Integer → Int, fromIntegral :: Int → Rational, fromIntegral :: Integer → Float, …
  - truncate :: Float → Int, truncate :: Double → Int, truncate :: Float → Integer, …, round :: …, ceiling :: …, floor :: …

- Conversions are not necessary in, for example, 3 + 4.5 or in:

```
f x = 2 * x + 3.5
g y = f 4 / y
```

,

but for example in:

```
f :: Int → Float
f x = 2 * (fromIntegral x) + 3.5
```

or in:

```
f x = 2 * x + 3.5
g y = f (fromIntegral (length "abcd")) / y
```

Programming Paradigms

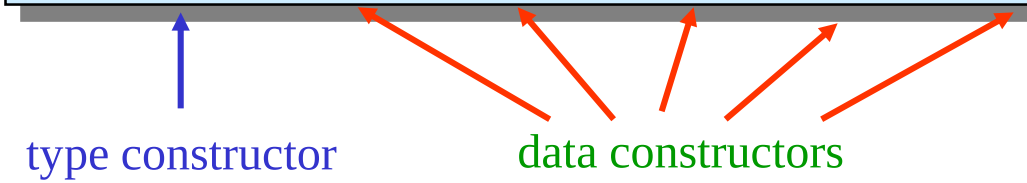
Algebraic data types

Declaration of (algebraic) data types

- An important aspect of typical Haskell programs is the definition of problem specific data types (instead of building everything from lists or so).

- To that end, one primarily uses data type declarations:

```
data  Color  =   Red  |  Green  |  Blue  |  White  |  Black
```



- Syntax: constructors in Haskell (both data and type constructors) generally start with a capital letter (exception: certain symbolic forms like in the case of lists).

- Semantics: the newly defined type Color above is an enumeration type that consists of exactly the five given values.

Declaration of (algebraic) data types

- User defined data types like

```
data  Color  =   Red  |  Green  |  Blue  |  White  |  Black
```

can arbitrarily be used as components in other types, such as for example in
[(Color, Int)] with values e.g. [ ], [(Red, –5)] and [(Red, –5), (Blue, 2), (Red, 0)].

- Computation goes via pattern matching:

```
primaryCol :: Color → Bool
primaryCol Red       =  True
primaryCol Green     =  True
primaryCol Blue      =  True
primaryCol _         =  False
```

User defined structured types

- It is also possible to declare new types with <u>structure</u>, by using a data constructor with parameters:

```
data  Point  =   Pt  ( Integer, Integer )
```

type constructor — data constructor — type of encased values

- With such a user defined data constructor with parameters, one can then construct structured values of one’s own type:

```
Pt (1, 2) :: Point
```

- It is permissible to use the same name for a type constructor and for a data constructor (e.g., twice Pt here), even if the data constructor does not belong to the same type.

User defined structured types

- A somewhat more complex example:

```
data Date  = Date Int Int Int
data Time = Hour Int
data Connection  =  Train Date Time Time |
                    Flight String Date Time Time
```

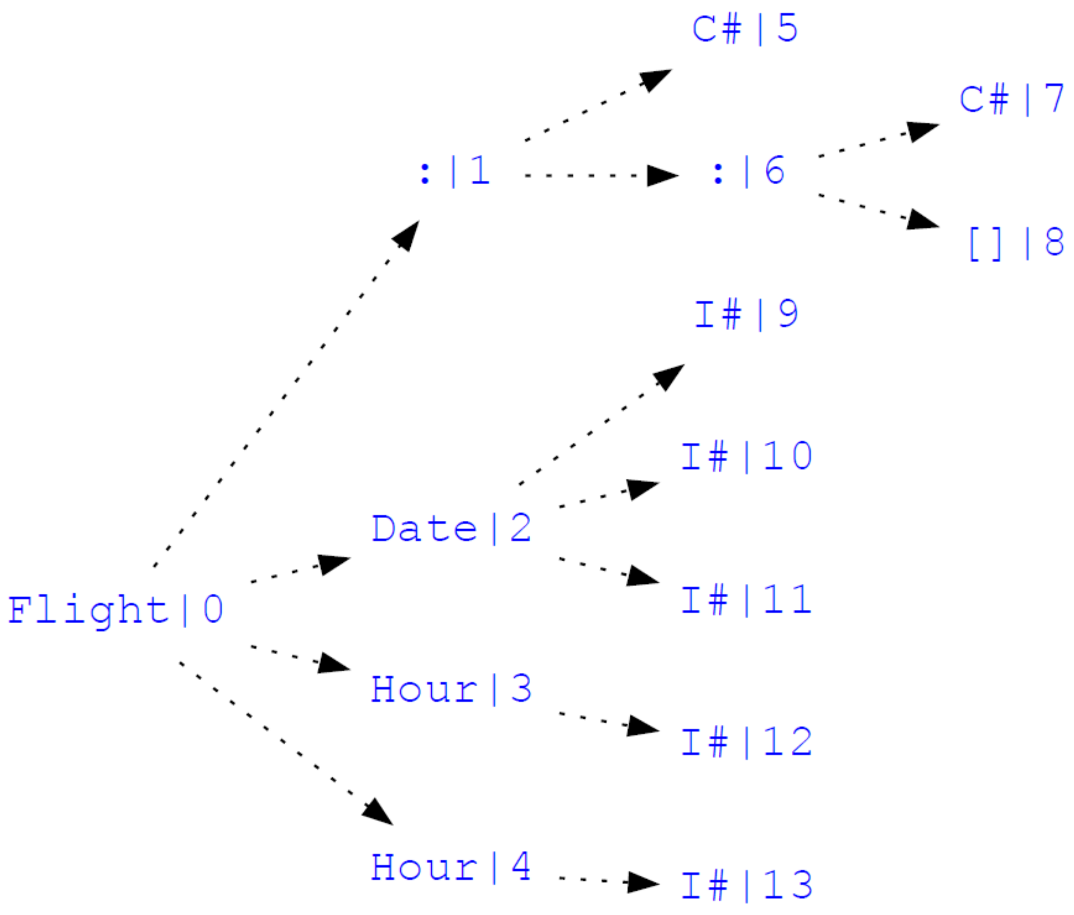
- Possible values for Connection:
  - Train (Date 20 04 2011) (Hour 11) (Hour 14)
  - Flight "LH" (Date 20 04 2011) (Hour 16) (Hour 20)
  - …

- Computation via pattern matching:

```
travelTime :: Connection → Int
travelTime (Flight _ _ (Hour d) (Hour a)) = a – d + 2
travelTime (Train _ (Hour d) (Hour a))    = a – d + 1
```

User defined structured types

- Internal representation for: Flight "LH" (Date 20 04 2011) (Hour 16) (Hour 20)



Data constructors as special functions

For:

```
data Date   =  Date Int Int Int
data Time   =  Hour Int
data Connection  =  Train Date Time Time |
                    Flight String Date Time Time
```

we get:

```
> :t Date
Date :: Int → Int → Int → Date
> :t Hour
Hour :: Int → Time
> :t Train
Train :: Date → Time → Time → Connection
> :t Flight
Flight :: String → Date → Time → Time → Connection
```

Recursive data types

- Like function definitions, data type declarations can also be recursive.

- Maybe the simplest example:

```
data Nat  =  Zero | Succ Nat
```

- Values of that type Nat:

  Zero, Succ Zero, Succ (Succ Zero), …

- Computation via pattern matching:

```
add :: Nat → Nat → Nat
add Zero       m = m
add (Succ n)  m = Succ (add n m)
```

Recursive data types

- The definition:

```
add :: Nat → Nat → Nat
add Zero      m = m
add (Succ n)  m = Succ (add n m)
```

maybe reminds of:

```
concatenation  [ ]       ys    =    ys
concatenation  (x : xs)  ys    =    x : concatenation xs ys
```

- Indeed, lists are internally defined as, essentially:

```
data [Bool]  =  [ ]  |  (:) Bool [Bool]
```

Recursive data types

- A somewhat more complex example:

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr
```

- Possible values:

Lit 42 , Add (Lit 2) (Lit 7) , Mul (Lit 3) (Add (Lit 4) (Lit 0)) , …

- A “mini interpreter” :

```
eval :: Expr → Int
eval  (Lit n)        =  n
eval  (Add e1 e2)  =  eval e1 + eval e2
eval  (Mul e1 e2)  =  eval e1 * eval e2
```

Recursive data types

Or, general binary trees:

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

with data constructors typed as follows:

```
> :t Leaf
Leaf :: Int → Tree
> :t Node
Node :: Tree → Int → Tree → Tree
```

and (to be defined) functions for “flattening”, prefix traversal, postfix traversal, …

Mutually recursive data types

- Finally, a somewhat artificial example:

```
data T1  =  A T2 | E
data T2  =  B T1
```

- Possible values for T1:

E , A (B E) , A (B (A (B E))) , A (B (A (B (A (B E))))) , …

- Possible values for T2:

B E , B (A (B E)) , B (A (B (A (B E)))) , …

- Computation:

```
as :: T1 → Int
as (A t) = 1 + as' t
as E     = 0

as' :: T2 → Int
as' (B t) = as t
```

Type synonyms

- Type synonyms give new names for already existing types:

```
type String  =  [Char]
```

  - in contrast to data, no constructors, no alternatives; also, really just a new name, not a new type

  - can be nested:

```
type Pos     =  (Int, Int)
type Trans   =  Pos → Pos
```

  but not recursive!