



Functional Programming

Week 2 – Tree Shaped Data and Datatypes

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitering Christian Sternagel

Last Lecture

- algorithm (can be informal) vs. program (concrete prog.-language)
- Haskell script (code, program, ...), e.g., `program.hs`
`fahrenheitToCelsius f = (f - 32) * 5 / 9`
consists of function definitions that describe input-output behaviour
- function- and parameter-names have to start with lowercase letters
- read-eval-print loop:
load script, enter expressions and let these be evaluated

```
$ ghci program.hs
... welcome message ...
Main> fahrenheitToCelsius (3 + 20) - 7
-12.0
Main> ... further expressions ...
...
Main> :q
```

Structured Data

Different Representations of Data

- some (abstract) element can be represented in various ways

- example: numbers

- roman:
- decimal:
- binary:
- English:
- tally list:

XI
11
1011
eleven
|||||

- fact: algorithms depend on concrete representation

- example: addition

- decimal + binary: process digits of both numbers from right to left

$$\begin{array}{r} 7823 \\ + 909 \\ \hline 8732 \end{array}$$

- tally list: just write the two numbers side-by-side
- roman: algorithm?
- English: not well-suited

(||| + || = ||||)
(IV + IX = XIII)
(twenty-nine + two = thirty-one)

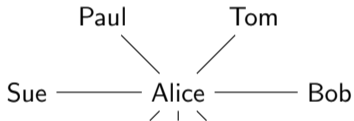
- in Haskell: numbers are built-in, representation not revealed to user

Different Representations of Data – Continued

- representation must be chosen appropriately
- example: person
 - photographer:



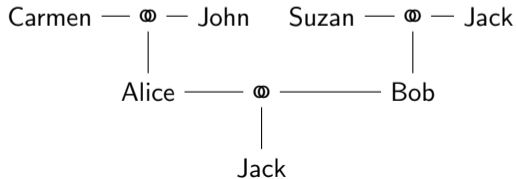
- social analysis:



- advertizing:

Bob (bob@foo.com, employee, hobbies: photography, jazz music, ...)

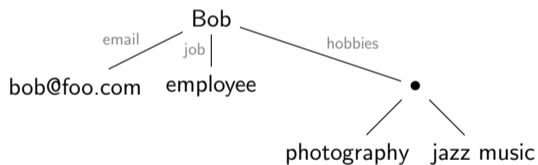
- genealogist:



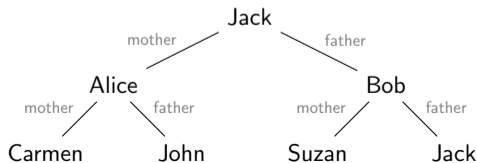
Tree Shaped Data

- in functional programming most of the data is **tree shaped**
- a **tree**
 - has exactly one **root** node
 - can have several subtrees; nodes without subtrees are **leaves**
 - nodes and edges can be labeled
- in computer science, trees are usually displayed upside-down
- examples from previous slide

- advertizing:

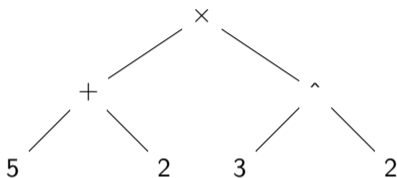


- genealogist:



Expressions = Trees

- mathematical expressions can be represented as trees
- example
 - expression in textual form: $(5 + 2) \times 3^2$
 - expression as tree

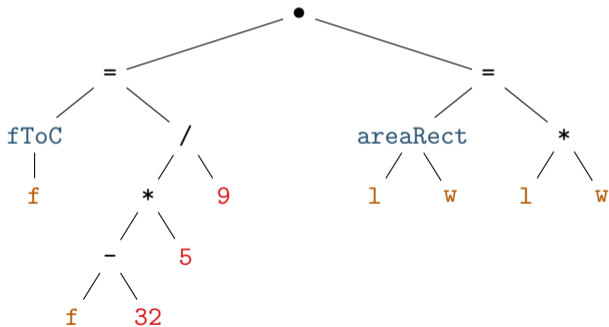


- remarks
 - the process of converting text into tree form is called **parsing**
 - operator precedences (\wedge binds stronger than \times b.s.t. $+$) and parentheses are only required for parsing
 - parsing $(5 + 2) \times (3^2)$ results in tree above
 - $5 + 2 \times 3^2$ and $((5 + 2) \times 3)^2$ represent other trees
 - algorithm of calculator
 - convert textual input into tree
 - evaluate the tree bottom-up, i.e., start at leaves and end at root

Programs = Trees

- programs can be represented as trees, too: **abstract syntax tree**
- example
 - program in textual form

```
-- some comment
fToC f = (f - 32) * 5 / 9
areaRect l w = l * w
```
 - abstract syntax tree (draft)



- comments and parentheses are no longer present in syntax tree

Tree Shaped Data

- many programs deal with tree shaped data
- examples
 - calculator evaluates expression tree
 - compiler translates abstract syntax tree into machine code
 - search engine translates query into HTML (tree shaped)
 - contact application manages tree shaped personal data
 - file systems are organised as trees
- trees as **mental model** or representation of data is often suitable
- good news: processing tree shaped data is well-supported in functional programming
- next lecture: define functions on trees
- this lecture: restriction of trees via **types**

Types

Types

- functions are often annotated by their domain and codomain, e.g.,
 - $(!) : \mathbb{N} \rightarrow \mathbb{N}$
 - $(/) : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \rightarrow \mathbb{R}$
 - $\log_2 : \mathbb{R}_{>0} \rightarrow \mathbb{R}$
- domain and codomain provide useful information
 - domain: what are allowed inputs to a function
 - codomain: what are potential outputs of the function
- aim: specify domains and codomains of (Haskell-)functions
- notions
 - **elements** or **values**
 - maths: 5, 8, π , $-\frac{3}{4}$, ...
 - Haskell: 5, 8, 3.141592653589793, -0.75, ..., "hello", 'c', ...
 - **sets of elements** to specify domain or codomain, in Haskell: **types**
 - maths: \mathbb{N} , \mathbb{Z} , \mathbb{Q} , \mathbb{R} , $\mathbb{Q} \setminus \{0\}$, ...
 - Haskell: Integer, Double, String, Char, ...

Typing Judgements

- in maths, we write statements like $7 \in \mathbb{Z}$, $7 \in \mathbb{R}$, $0.75 \notin \mathbb{Z}$
- in Haskell we can also express that a value or expression has a certain type via **typing judgements**
 - format: `expression :: type`
 - examples
 - `7 :: Integer` or `7 :: Double`
 - `'c' :: Char`
- that an expression indeed has the specified type is checked by the Haskell compiler
 - if an expression has not the given type, a type error is displayed
 - examples
 - `7 :: String` or `0.75 :: Integer` or `'c' :: String`
 - `(7 :: Integer) :: Double`
 - remarks
 - unlike in maths where $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q}$, in Haskell the types `Integer` and `Double` are **not** subtypes of each other
 - although some expressions can have both types (e.g., `7 + 5`), in general numbers of different types have to be converted explicitly
 - once a typing judgement is applied, the type of that expressions is fixed

Typing of Haskell Expressions

- not only values but also functions have a type, e.g.,
 - `(/) :: Double -> Double -> Double`
 - `(+) :: Integer -> Integer -> Integer`
 - `(+) :: Double -> Double -> Double`
 - `head :: String -> Char`

remarks

- a function can have multiple types, e.g., `(+)`
- limited expressivity, e.g. `(/) :: Double -> Double \ {0} -> Double` not allowed
- type checking enforces that in all function applications, type of arguments matches input-types of function
- example: consider expressions `expr1 / expr2`
 - recall: `(/) :: Double -> Double -> Double`
 - it will be checked that both `expr1` and `expr2` have type `Double`
 - type of the overall expression `expr1 / expr2` will then be `Double`
- examples
 - `5 + 3 / 2`
 - `5 + '3'` or `5.2 + 0.8 :: Integer`



Static Typing

- Haskell performs static typing
- **static** typing: types will be checked before evaluation
(by contrast, dynamic typing checks types during evaluation)
- when loading Haskell script
 - check types of all function definitions `someFun x ... z = expr`:
check that lhs `someFun x ... z` has same type as rhs `expr`
 - consequence: expressions cannot change their type during evaluation
- when entering expression in REPL: type check expression before evaluation
- benefits
 - no type checking required during evaluation
 - no type errors during evaluation

Built-In Types – A First Overview

- numbers
 - `Integer` – arbitrary-precision integers
 - `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)
 - `Float` – single-precision floating-point numbers (`-12.34`, `5.78e36`)
 - `Double` – double-precision floating-point numbers
- characters and text
 - `Char` – a single character (`'a'`, `'Z'`, `' '`)
 - `String` – text of arbitrary length (`""`, `"a"`, `"The answer is 42."`)
 - some characters have to be **escaped** via the backslash-symbol `\`:
 - `'\t'` and `'\n'` – tabulator and new-line
 - `'\"'` and `'\''` – double- and single quote
 - `'\\'` – the backslash character
 - example: in the program

```
text = "Please say \"hello\"\n\nwhenever you enter the room"
```

the string `text` corresponds to the following two lines:
Please say "hello"
whenever you enter the room
- `Bool` – yes/no-decisions or truth-values (`True`, `False`)

Datatypes

Current State

- each value and function in Haskell has a type
- types are used to define input and output of function
- example: `fahrenheitToCelsius :: Double -> Double`
- built-in types for numbers, strings, and truth values
- missing: how to define types that describe tree shaped data?
- solution: definition of (algebraic) **datatypes**

Datatype Definitions

- recall: a tree consists of a (labelled) root and 0 or more subtrees
- a **datatype** definition defines a set of trees by specifying all possible labelled roots together with a list of allowed subtrees
- Haskell scripts can contain many **datatype definitions** of the form

```
data TName =  
    CName1 type1_1 ... type1_N1  
  | ...  
  | CNameM typeM_1 ... typeM_NM  
deriving Show
```

where

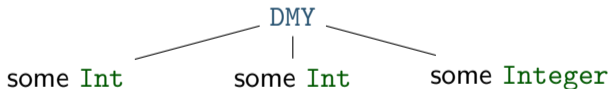
- **data** is a Haskell keyword to define a new datatype
- **TName** is the name of the new type; **type-names** always start with capital letters
- **CName1, ..., CNameM** are the labels of the permitted roots; these are called **constructors** and have to start with capital letters
- **typeI_J** can be any Haskell type, including **TName** itself
- **|** is used as separator between different constructors
- **deriving Show** is required for displaying values of type **TName**

Example Datatype Definition – Date

```
data Date = -- name of type
  DMY      -- name of constructor
    Int    -- day
    Int    -- month
    Integer -- year
```

deriving Show

- here, there is only one constructor: `DMY`
- for day and month the precision of `Int` is sufficient
- the values of the type `Date` are exactly trees of the form



- in Haskell, these trees are built via the constructor `DMY`; `DMY` is a function of type `Int -> Int -> Integer -> Date` that is **not evaluated**
- example value of type `Date`: `DMY 11 10 2021`

Example Datatype Definition – Person

```
data Person = -- name of type
  Person      -- constructor name can be same as type name
    String    -- first name
    String    -- last name
    Bool      -- married
    Date      -- birthday
```

deriving Show

- reuse of previously defined types is permitted, in particular `Date`
- this leads to trees with more than one level of subtrees
- example program that defines a person (and an auxiliary date)

```
today = DMY 11 10 2021
myself = Person "Rene" "Thiemann" True today
-- is the same as
myself = Person "Rene" "Thiemann" True (DMY 11 10 2021)
```

Example Datatype Definition – Vehicle

```
data Brand = Audi | BMW | Fiat | Opel deriving Show
```

```
data Vehicle =
```

```
    Car Brand Double -- horsepower
```

```
    | Bicycle
```

```
    | Truck Int -- number of wheels
```

```
deriving Show
```

- `Brand` just defines 4 car brands; all "trees" of type `Brand` consist of a single node; such datatypes are called **enumerations**
- there are three kinds of `Vehicles`, each having a different list of types
- example expressions of type `Vehicle`:

```
Car Fiat (60 + 1)
```

```
Car Audi 149.5
```

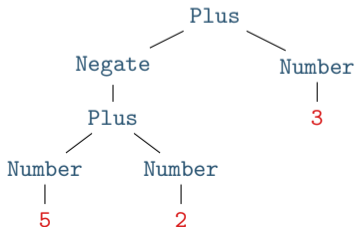
```
Bicycle
```

```
Truck (-7) -- types don't enforce all sanity checks
```

Example Datatype Definition – Expr

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr  
deriving Show
```

- type `Expr` models arithmetic expressions with addition and negation
- `Expr` is a **recursive** datatype: `Expr` is defined via `Expr` itself
- recursive datatypes contain values (trees) of arbitrary large height
 - expression $-(5 + 2) + 3$ in Haskell (as value of type `Expr`):
`Plus (Negate (Plus (Number 5) (Number 2))) (Number 3)`
 - expression as tree



Example Datatype Definition – Lists

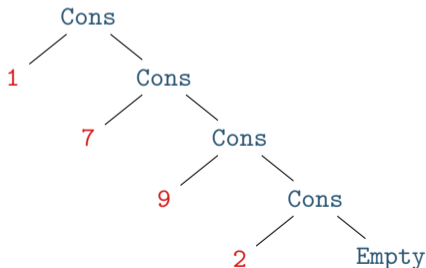
- lists are just a special kind of trees, e.g., lists of integers

```
data List =  
  Empty  
  | Cons Integer List  
deriving Show
```

- example representation of list [1, 7, 9, 2]

- in Haskell: `Cons 1 (Cons 7 (Cons 9 (Cons 2 Empty)))`

- as tree:



Summary

- mental model: data = tree shaped data
- type = set of values; restricts shape of trees
- built-in types for numbers and strings
- user-definable datatypes, e.g., for expressions, lists, persons

```
data TName =  
    CName1 type1_1 ... type1_N1  
  | ...  
  | CNameM typeM_1 ... typeM_NM  
deriving Show
```

- missing: function definitions on trees