Functional Programming

WS 2019/20

Torsten Grust
University of Tübingen

Administrivia

Slot	Time	Room
Lectures	Thursday, 10:15–11:45	C215
Tutorials	Tuesday, 14:15–15:45	A301

- Lecture: notes on slides and whiteboard + live Haskell coding
- Slides and Haskell code downloadable: https://db.inf.uni-tuebingen.de/teaching/FunctionalProgrammingWS2019-2020.html
- ⚠️ **Forum:** https://forum-db.informatik.uni-tuebingen.de/c/ws1920-db1

Tutorials + Exercises

- Held weekly (starting with Denis Hirn, continued by Benjamin Dietrich).
- Tutorials start on **Tuesday, October 29, 2019.**
- New exercise sheets every Friday, hand in by Thursday next week.
  - Teams of two 👤+👤
  - Mostly Haskell coding
- We use git to distribute assignments and collect your solutions (see the Forum for details and how to form teams).

Grading + Exam

- Need $\frac{2}{3}$ of exercise points to be admitted to the final written exam.
- Grading: excess exercise points turn into final exam bonus points.
- **Final exam on Thursday, February 6, 2020, 10:00–12:00.**
  - Rooms: C215 + A104
  - May bring a double-sided DIN A4 cheat sheet of notes

Install Haskell! ≫=

- **Haskell Platform:**
  - http://www.haskell.org/platform
  - Available for Windows , macOS , Linux
- Haskell compiler ghc (Glorious Glasgow Haskell Compiler)
  - Current version 8.8.1, any recent version 8.x is OK
- Includes Haskell REPL ghci
  - To ghci's configuration (usually in ~/.ghci), add the following switch:

```
:seti -XMonomorphismRestriction
```

Further Reading

- Bird: “Thinking Functionally with Haskell”, Cambridge University Press 2015
- Allen: “Haskell Programming From First Principles”, Gumroad 2016, http://haskellbook.com
- FP Complete: “The School of Haskell” at http://www.schoolofhaskell.com
- Keller: “Learning Haskell” at http://learn.hfm.io (in development)

Functional Programming (*FP*)

A programming language is a medium for **expressing ideas** (not to get a computer perform operations). Thus programs must be written for people to read, and only incidentally for machines to execute.

- Computational model in FP: **reduction** (replace **expressions** by their **value**).

Functional Programming (*FP*)

In FP, expressions are formed by **applying functions to values.**

1. **Functions as in maths:** $x = y \Rightarrow f(x) = f(y)$

2. **Functions are values** just like numbers or text.

	FP	Imperative
program construction	function application + composition	statement sequencing
execution	reduction (evaluation)	state changes
semantics	λ calculus	Turing machines

- Absence of explicit machine control generally leads to concise, often quite elegant, programs. Focus remains on problem. Programs are easier to reason about.

Example

$n \in \mathbb{N}$, $n \geq 2$ is a **prime number** iff the set of non-trivial factors of n is empty:

$$n \text{ is prime} \Leftrightarrow \{ m \mid m \in \{ 2,\ldots,n-1 \}, n \bmod m = 0 \} = \emptyset$$

Haskell Ramp-Up: Function Application and Composition

- Read `≡` as *“denotes the same value as”*.
- **Apply** `f` to value `e`: `f e` (juxtaposition, “apply”, invisible binary operator `␣`, Haskell speak: `infixl 10 ␣`)
  - `␣` has max precedence (`10`): `f e1 + e2 ≡ (f e1) + e2`
  - `␣` associates to the left (`l`): `g f e ≡ (g f) e`
- Function **composition:**
  - `g (f e)`
  - Operator `.` (“after”): `(g . f) e` (`.` = ∘ as in `g ∘ f`)
  - Alternative “apply” operator `$` (lowest precedence, associates to the right, `infixr 0 $`): `g $ f $ e ≡ g (f e)`

Infix vs. Prefix Operators

- Prefix application of binary infix operator `@`:
  `(@) e1 e2 ≡ e1 @ e2`
  - Example: `(&&) True False ≡ False`

- Infix application of binary function `f`:
  `e1 `f` e2 ≡ f e1 e2` (`: backtick)
  - `x `elem` [1,2,3]`
  - `n `mod` 2`

- User-defined infix operators, built from symbols
  `!#$%&*+/<=>?@\^|~:.`
  - ⚠️ Identifiers starting with `:` reserved to denote value constructors of algebraic data types.

Values and Types

- Read `::` as *"has type"*.

  Any Haskell value `e` has a type `t` (`e :: t`) that is determined at **compile time.** The `::` type assignment is either given explicitly or inferred by the compiler.

Basic Built-In Haskell Types

Type	Description	Values
`Int`	64-bit integers [-2^{63} .. $2^{63}-1$]	`0`, `1`, `(-42)`
`Integer`	arbitrary-precision integers	`0`, `10^100`
`Float`	single-precision floating points	`0.1`, `1e02`
`Double`	double-precision floating points	`0.42`, `1e-2`
`Char`	Unicode characters	`'x'`, `'\t'`, `'Δ'`, `'\8710'`, `'\^G'`
`Bool`	Booleans	`True`, `False`
`()`	“unit” (single-value type)	`()` (C: `void`)

Type Constructors

- **Type constructors** build new types from existing types.
- Let `a`, `b`, ... denote arbitrary types (type variables):

Type Constructor	Description	Values
`(a,b)`	pairs of values of types `a`, `b`	`(1, True) :: (Integer, Bool)`
$(a_1,a_2,\ldots a_n)$	n-tuples	
`[a]`	lists of values of type `a`	`[True,False] :: [Bool]`, `[] :: [a]`
`Maybe a`	optional value of type `a`	`Just 42 :: Maybe Integer`, `Nothing :: Maybe a`
`Either a b`	choice between a and b	`Left 'x' :: Either Char b`, `Right pi :: Either a Double`
`IO a`	I/O action that returns value of type `a` (once performed)	`print 42 :: IO ()`, `getChar :: IO Char`
`a -> b`	function from type `a` to `b`	`isLetter :: Char -> Bool`

Currying

- Recall:
  1. `e1 ++ e2` ≡ `(++) e1 e2`
  2. `(++) e1 e2` ≡ `((++) e1) e2`

- Function application happens one argument at a time (**currying**, *Haskell B. Curry*)
- Type of `n`-ary function: a_1 `->` a_2 `->` … `->` a_n `->` `b`. Type constructor `->` associates to the right, thus read as:

  a_1 `->` (`a`$_2$ `->` (… `->` (`a`$_n$ `->` `b`)…))

- Enables **partial application:** *"Give me a value of type* a_1*, I'll give you a* $(n-1)$*-ary function of type* a_2 `->` … `->` a_n `->` b*"*.

Defining Values (and thus: Functions)

- `=` **binds** names to values, values name must not start with `A`–`Z` (Haskell style: `camelCase`).
- Define constant (0-ary function) `c`, value of `c` is that of expression `e`: `c = e`
- Define `n`-ary function `f`, arguments x_i and `f` may occur in `e` (no `letrec` needed): `f` x_1 x_2 … x_n `= e`
- Haskell program:
  set of top-level bindings (order immaterial, **no** rebinding!)
- Good style: give type assignments for top-level bindings:

```
f :: a1 -> a2 -> b
f x1 x2 = e
```

Guards

- **Guards** (introduced by `|`) are multi-way conditional expressions:

```
f x₁ x₂ … xₙ
  | q₁ = e₁
  | q₂ = e₂
  | q₃ = e₃
  …
```

  - Guards q_i (expressions of type `Bool`) evaluated top to bottom, first `True` guard wins. Syntacic sugar: `otherwise` ≡ `True`.

- Compare:

$$\text{fac } n = \begin{cases} 1 & , \text{if } n \leq 1 \\ n * \text{fac } (n-1) & , \text{otherwise} \end{cases}$$

Local Definitions

1. `where` **binding:** Local definitions visible in the entire rhs of a definition:

```
f x₁ x₂ … xₙ | p₁¤ = e₁¤        -- ¤: gᵢ in scope
             | p₂¤ = e₂¤
             …
  where
    g₁ = …¤
    g₂ = …¤
```

2. `let` **expression:** Local definitions visible inside an expression:

```
let g₁ = …¤
    g₂ = …¤
 in e¤                          -- ¤: gᵢ in scope
```

Layout (Two-Dimensional Syntax)

- The Haskell compiler applies these transformation rules to the program source before compilation begins:

  1. The first token *after* a `where`/`let` and the first token of a top-level definition define the upper-left corner ⌜ of a box.
  2. The first token left of the box closes the box ⌟ (“*offside rule*”).
  3. Insert `{` before the box.
  4. Insert `}` after the box.
  5. Insert `;` before a line that starts at left box border.

Layout (Example)

1. Original source:

```
let y   = a * b
    f x = (x + y) / y
in  f c + f d
```

2. Make box visible:

```
let ┌──────────────┐
    │y   = a * b   │
    │f x = (x + y) / y
    └──────────────┘
in  f c + f d              -- offside: in
```

3. After source transformation:

```
let {y   = a * b
    ;f x = (x + y) / y}
in  f c + f d
```

Lists — The Go-to Container Data Structure in FP

- Recursive definition of lists:

  1. `[]` is a list (*nil*), type: `[] :: [a]`
  2. `x : xs` is a list, if `x :: a` and `xs :: [a]`.

     ↑ ↑

     head tail

     cons: `(:) :: a -> [a] -> [a]` with `infixr 5 :`

- Abbreviate long chains of *cons* using `[…]`:

  `3:(2:(1:[])) ≡ 3:2:1:[] ≡ [3,2,1] (≡ 3:[2,1])`

Lists

- Law (⚠️ `head`, `tail` are *partial functions*):

  `∀ xs ≠ []:  head xs : tail xs ≡ xs`

- Type `String` is a synonym for type `[Char]` (“list of characters”).

- Introduce your own **type synonyms** via (type names: **U**ppercase):

```
type t₁ = t₂
```

Lists

- Sequences (of enumerable elements):

```
[x..y]   ≡ enumFromTo x y                [x,x+1,x+2,...,y]

[x,s..y] ≡ enumFromThenTo x s y          [x,x+i,x+2*i,...,y]
                                           where i = s-x

[x..]    ≡ enumFrom x                    [x,x+1,x+2,...

[x,s..]  ≡ enumFromThen x s              [x,x+i,x+2*i,...
```

Pattern Matching

- *The* idiomatic Haskell way to **define a function by cases:**

```
f :: a₁ -> … -> aₖ -> b
f p₁₁ … p₁ₖ = e₁
f p₂₁ … p₂ₖ = e₂
            ⋮
f pₙ₁ … pₙₖ = eₙ
```

  - We have $e_i :: b$ for all $i \in \{1,\ldots,n\}$
  - On a call $f\ x_1\ x_2\ \ldots\ x_k$, each x_i is **matched** against **patterns** $p_{1i},\ldots,p_{ni}$ in order. Result is e_r if the rth branch is the first in which all pattern match.

Pattern Matching

Pattern	Matches if…	Bindings in e_r
constant `c`	x_i == `c`	
variable `v`	always	`v` = x_i
wildcard `_`	always	
tuple $(p_1,\ldots,p_m)$	components of x_i match component patterns `p`	those bound by component patterns `p`
`[]`	x_i == `[]`	
p_1:p_2	`head` x_i matches p_1 and `tail` x_i matches p_2	those bound by p_1 and p_2
`v@p`	`p` matches	those bound by `p` and `v` = x_i

- ⚠️ In a pattern, a variable may only occur once (*patterns are linear*).

Pattern Matching

- Pattern matching may be used in any expression (not just in function definitions): `case` expressions.

  Matches against patterns p_i as well as guards q_{ij} may be used together:

```
case e of p₁ | q₁₁ -> e₁₁
             | q₁₂ -> e₁₂
          ⋮
          pₙ | qₙ₁ -> eₙ₁
             | qₙ₂ -> eₙ₂
```