Functional Programming

WS 2021/22

Torsten Grust
University of Tübingen

Administrivia

Slot	Time	Room
Lectures	Thursday, 10:15–11:45	A301
Tutorials	Monday, 14:15–15:45	F119 (Sand 6/7)

- Lecture: notes on slides and board + live Haskell coding

- Slides and Haskell code downloadable:
  https://db.inf.uni-tuebingen.de/teaching/FunctionalProgrammingWS2021-2022.html

- ⚠️ **Forum:**
  https://forum-db.informatik.uni-tuebingen.de/c/ws2122-fp

Tutorials + Exercises

- Held weekly (organized by Denis Hirn).
- Tutorials start on **Monday, October 25, 2021.**
- New exercise sheets every Friday, hand in by Thursday night 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 10, 2022, 10:00–12:00.**
  - Rooms: *tba*
  - May bring a double-sided DIN A4 cheat sheet of notes

Install Haskell!

- Multi-platform installer for Haskell: **GHCup**
  - https://www.haskell.org/ghcup/
  - Available for Windows, macOS, Linux, FreeBSD
- Haskell compiler `ghc` (Glorious Glasgow Haskell Compiler)
  - Current version 8.10.7, 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
- Diehl: “What I Wish I Knew When Learning Haskell” at http://dev.stephendiehl.com/hask (includes advanced topics)

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

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

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

Haskell Ramp-Up: Function Application and Composition

- Read ≡ as *"denotes the same value as"*.

- **Apply** f to value e: $f \sqcup e$ ("apply", space ␣ as invisible binary operator, juxtaposition, Haskell speak: `infixl 10 ␣`)
  - ␣ has max precedence (10): $f\ e_1 + e_2 \equiv (f\ e_1) + e_2$
  - ␣ associates to the left (l): $g\ f\ e \equiv (g\ f)\ e$

- Function **composition:**
  - $g\ (f\ e)$
  - Operator `.` ("after"): $(g\ .\ f)\ e$ (`.` = ∘ as in $g \circ f$)
  - Alternative "apply" operator `$` (lowest precedence, associates to the right, `infixr 0 $`): $g\ \$\ f\ \$\ e \equiv g\ (f\ e)$

Infix vs. Prefix Operators

- Prefix application of binary infix operator $\otimes$:
  $(\otimes)\ e_1\ e_2 \equiv e_1 \otimes e_2$
  - Example: `(&&) True False` ≡ `False`
- Infix application of binary function f:
  e_1 \`$f$\` $e_2 \equiv f\ e_1\ e_2$ (\`: 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'`, `'λ'`, `'\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. e_1 ++ e_2 ≡ (++) e_1 e_2
  2. (++) e_1 e_2 ≡ ((++) e_1) e_2

- 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, value names 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): $f\ x_1\ x_2\ \ldots\ x_n$ `=` 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 to math notation:

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

Local Definitions

1. `where` **binding:** Local definitions visible in the entire right-hand side (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	$[x,x+1,x+2,\ldots,y]$
$[x,s..y]$	≡	enumFromThenTo x s y	$[x,x+\Delta,x+2*\Delta,\ldots,y]$ **where** $\Delta = s-x$
$[x..]$	≡	enumFrom x	$[x,x+1,x+2,\ldots$
$[x,s..]$	≡	enumFromThen x s	$[x,x+\Delta,x+2*\Delta,\ldots$

Pattern Matching

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

$$
\begin{array}{l}
f :: a_1 \to \cdots \to a_k \to b \\
f\ p_{11}\ \ldots\ p_{1k} = e_1 \\
f\ p_{21}\ \ldots\ p_{2k} = e_2 \\
\qquad\qquad\quad \vdots \\
f\ p_{n1}\ \ldots\ p_{nk} = e_n
\end{array}
$$

  - 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 $(p_1,\dots,p_m)$	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 in Expressions (case)

- 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:

$$
\begin{array}{llll}
\textbf{case}\ e\ \textbf{of} & p_1 & | & q_{11} \texttt{ -> } e_{11} \\
 & & | & q_{12} \texttt{ -> } e_{12} \\
 & \vdots & & \\
 & p_n & | & q_{n1} \texttt{ -> } e_{n1} \\
 & & | & q_{n2} \texttt{ -> } e_{n2}
\end{array}
$$