Programming in Haskell

Mirek Kratochvíl, KSI MFF CUNI

2026

slides version: `v2026-15-ge291943`
build: `2026-02-17 15:45:54 +0100`

Introduction and preliminaries

Metadata

NPRG068, LS, Zk, 3 ECTS

`miroslav.kratochvil@matfyz.cuni.cz`

`https://teaching.mff.cuni.cz/nprg068-web/` ← get the slides here

Live discussion:

- "FMP CUNI" Mattermost, team `2526`, channel `#nprg068-haskell`
  (get invite via SIS study group roster)
- "MFF Unofficial" Discord `#i-prg-haskell`

Who's this guy?!

- software engineering PhD
- 20+ years of coding for money
- running this course in various settings since 2018

Requirements

1. Homework — using the language as a tool to solve problems
   - Expected schedule (subject to change!):

Homework	Assignment	Deadline
0 (test)	ASAP	not really
1	ASAP	end of March
2	mid-March	end of April
3	mid-April	before end of May

2. Exam
   - exam will mainly verify if you understand what you submitted in homeworks
     i.e., verify that you did your homework
   - possible topics $\subset$ contents of slides
   - format: less than 10 minutes of chat about your code

Deadlines

- **Submit homework on time.**
  If you can't make it, let me know in advance.
- **Go to the exam.**
  If you did the homework, it is going to be very easy.

Homeworks — requirements

This course is about thinking about how you code. Prefer solutions that are:

- **Simpler and shorter**
- **Visibly more correct**

Acceptance conditions:

- Minimal acceptable solution conditions will be specified together with the assignments.
- *You must understand all code that you submitted, which is the only thing that will be tested at the exam.*
- LLM use is discouraged but supported
  (conditions: make a note in code comments, disclose prompts for generated code).

Homeworks — working in groups

- You are encouraged to discuss your homework solutions with others!
- Use appropriate “spoiler” features of chats to avoid wrecking the learning process of others.
- New in 2026: You may form groups and submit the same solution, with some conditions:
  - All authors of the code must know that you are submitting their code.
  - Verification: all solutions in a group must list all authors (form a clique!)
    (use comments or `README.md` or `AUTHORS` or cabal copyright metadata or such)
  - Same score is given to all authors.
  - Exam conditions remain the same (you have to understand the whole code you submitted).

Homeworks — submission

1. Make a `cabal` package with your solution
   (you will get instructions on how to do that soon)
2. Use `cabal sdist` to produce a package archive.
3. Submit the archive to the appropriate field in SIS study group roster.

Let’s get motivated to do some functions

Lecture topics

1. Why are pure functions such a great idea
2. How to cleanly use pure functions to do “impure” (effectful) stuff
3. Overview of current Haskell implementation(s)
4. How the Haskell type system and evaluation actually work
5. *Best of Haskell Today* — useful abstractions and libraries

Why would I learn a new programming language once again?

- New and useful design patterns
- Haskell programs do not crash
  (at least not for stupid reasons)
- Functional programming is Really Very Expressive
- Industry prefers people who can think at the higher-level
- You will learn a lot about other languages too!

Why...?

- Bonus 1: Haskell practice dramatically increases the ability to code without errors in all other languages.

  The programmer is able to squeeze maximum utility from any type system.

- Bonus 2: Lots of dark and inexplicable properties of other languages have an easily understandable variant in Haskell.

  Notably: C++ metaprogramming, Rust implementation, functional hell in JavaScript frameworks.

Why… philosophically

Programming is mostly about managing complexity. Approaches vary:

- **Hiding complexity:** Putting complex stuff into a friendly-looking closed box.
  - Users don't get scared.
  - Works until something inside the box breaks.
  - Pretty hard to learn from.
- **Taming complexity:**
  Decomposing the complex problem into simple pieces connected in a clean structure.
  - Makes programs reusable, easily fixable and visibly correct.
  - Decomposition strategies are usually not obvious.
  - Learning curve collisions happen.
  - This is what Haskell aims for.

Why… philosophically

Programming is mostly about managing complexity. Approaches vary:

- **Hiding complexity:** Putting complex stuff into a friendly-looking closed box.
  - Users don't get scared.
  - Works until something inside the box breaks.
  - Pretty hard to learn from.
- **Taming complexity:**
  Decomposing the complex problem into simple pieces connected in a clean structure.
  - Makes programs reusable, easily fixable and visibly correct.
  - Decomposition strategies are usually not obvious.
  - Learning curve collisions happen.
  - This is what Haskell aims for.

Why functions… philosophically

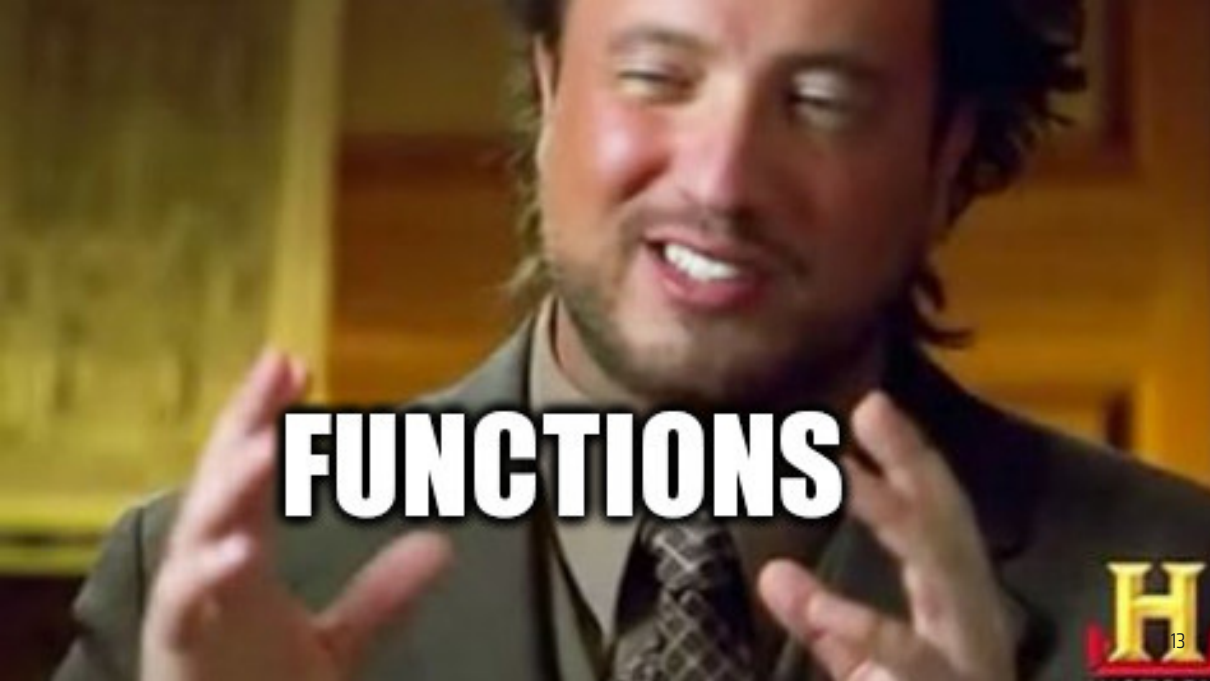
To be able to decompose problems into reasonably reusable parts,
one has to decompose into things that re-compose well.

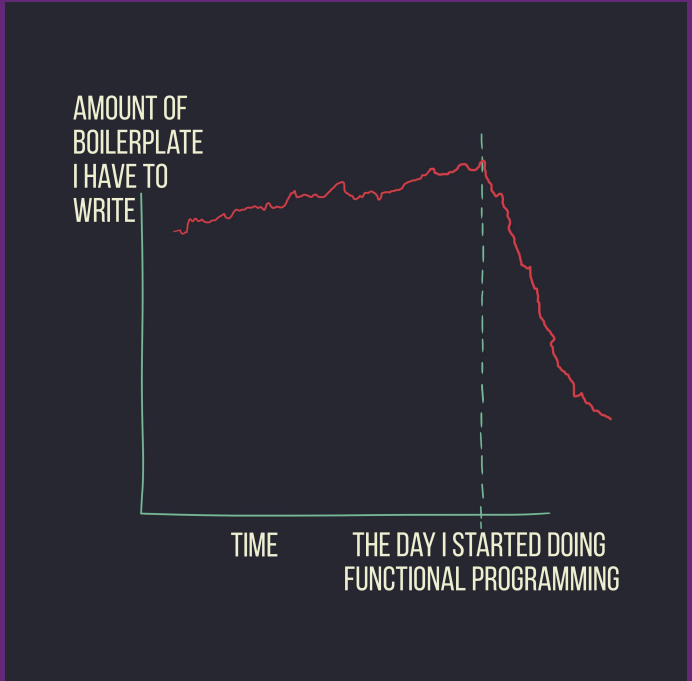
Decomposition	Sequences?	Contains?	Parametrizes?	CPUs like it?
Procedures	OK	fragile	more fragile	OK
Objects	no semantics	OK	uses functions	not much
Pure functions	OK	OK	OK	not at all[1]

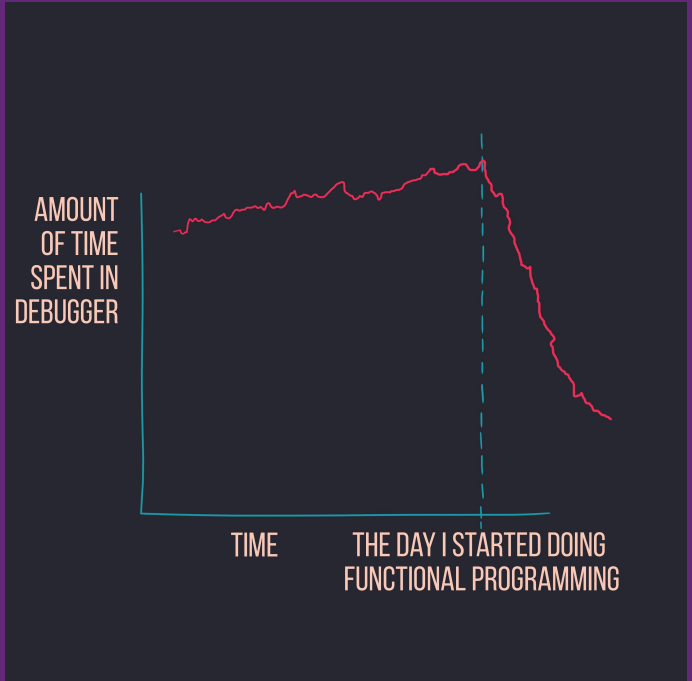
[1] solved by Haskell

Why functions... schematically on a timeline

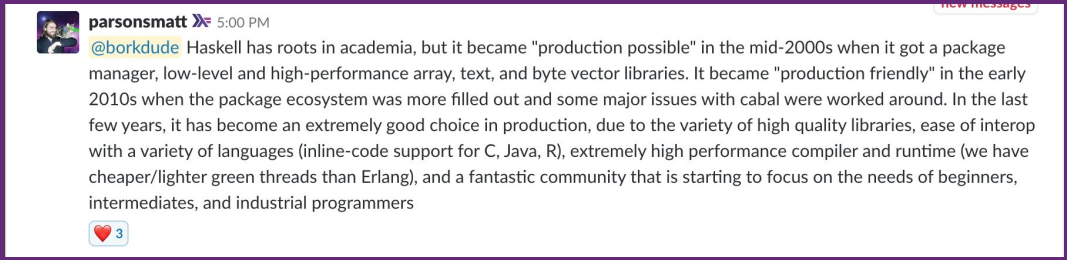
Rob Pike's Rule 5: *Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.*


FUNCTIONS


AMOUNT OF
BOILERPLATE
I HAVE TO
WRITE
TIME
THE DAY I STARTED DOING
FUNCTIONAL PROGRAMMING


AMOUNT OF TIME SPENT IN DEBUGGER
TIME
THE DAY I STARTED DOING FUNCTIONAL PROGRAMMING

Short history



Industry-ready

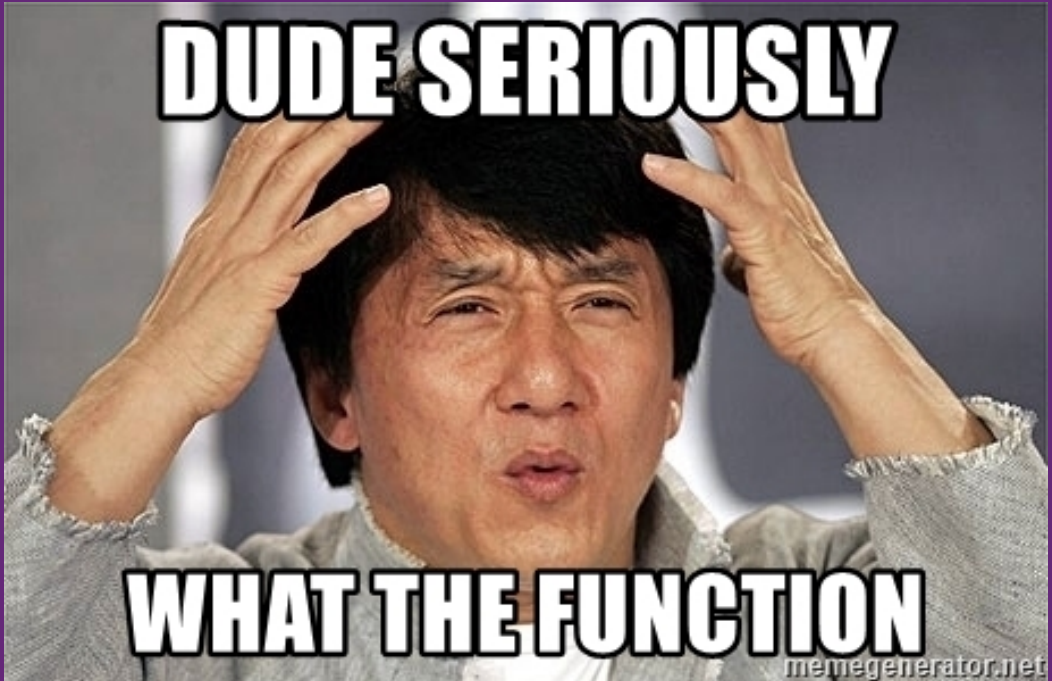

C++
Rust
Haskell
PureScript

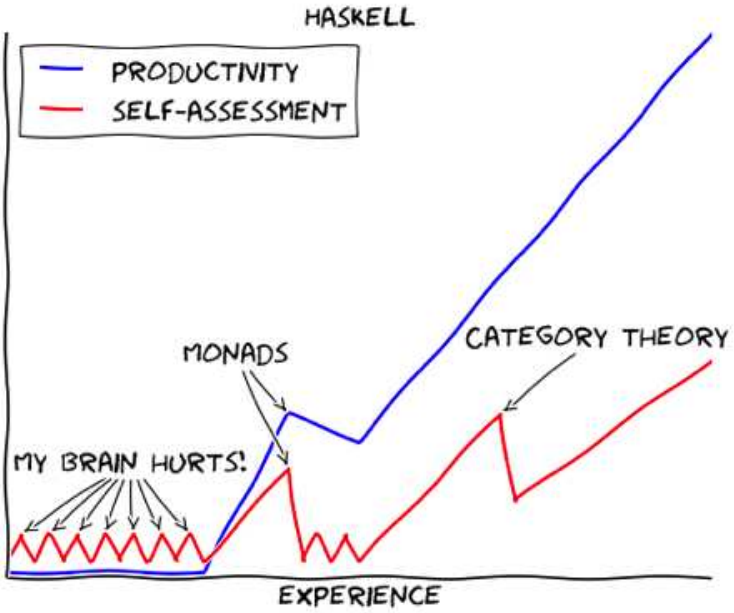
What does expressive mean?

$$wordCount = length \cdot words \langle\$\rangle readLine$$

$$rle = map\ (liftA2\ (,)\ head\ length) \cdot group$$

$$scc = dfs \langle*\rangle reverse \cdot postOrd \cdot transposeG$$


DUDE SERIOUSLY
WHAT THE FUNCTION
memegenerator.net


HASKELL
PRODUCTIVITY
SELF-ASSESSMENT
MONADS
CATEGORY THEORY
MY BRAIN HURTS!
EXPERIENCE

Haskell vs. LLMs

LLM use in this course is allowed but highly discouraged.

- LLMs code somehow looks like Python and Java encoded in Haskell
- Usual Haskell oneliner:
  - typically much shorter than a LLM prompt that generates and fixes it
    (unless the LLM manages to copy it from internets)
  - *comes proven correct by construction, for free*
- *within this course* I chose to ignore societal, environmental and didactic concerns of LLM use
  (will happily roast that elsewhere)

Language neighborhood

- Haskell is *managed* — the run-time manages the connection to the system.
  - It can't fully replace low-level languages (C, C++, Rust).
  - It is beneficial to fully exploit the run-time layer and replace other managed languages (and large parts of system-level software) with Haskell, typically with hundred-fold savings in speed, code and stress.
- Haskell is *strictly typed* and *compiled* — it can't (fully) replace shell-like and glue languages (Bash, R, K, ...), but you can get close.
- Haskell is *not a browser language* per se, but you can compile it to WASM.

Who uses Haskell?!

- Haskell is a requirement for any serious PL research
- FOSS: Pandoc, Purescript&Elm, PostgREST, Shellcheck, Darcs, Agda, Hasura (GraphQL), Hadolint, …
- Proprietary: we've seen reports of databases and high-performance trading (Haskell replaced Erlang), complex internal tooling, sometimes even custom compilers
  `https://wiki.haskell.org/Haskell_in_industry`

Common issue: most projects never anticipate to *need* such a big gun and start with a tradeoff.

Literature

- Learn You A Haskell For Great Good
  `https://learnyouahaskell.github.io/`
- Real World Haskell
  `https://github.com/tssm/up-to-date-real-world-haskell`
- Haskell Wikibook
  `https://en.wikibooks.org/wiki/Haskell`
- What I Wish I Knew When Learning haskell
  `https://github.com/sdiehl/wiwinwlh`

⋮

- Typeclassopædia
- Hoogle
  `https://hoogle.haskell.org/`

Current slide contents

1. Introduction and preliminaries
2. Let's get motivated to do some functions
3. Functional programming quickstart
4. Type classes, functors and IO
5. Monad practice: Parsing combinators
6. Packages and libraries
7. Tour de Prelude
8. Containers
9. Let's compile some Haskell!
10. Curry-Howard correspondence
11. Stringology and text formatting
12. IO!
13. Let's transform some monads
14. Streaming!
15. Optics, lenses and prisms
16. Web applications
17. Graphics
18. Debugging, testing, benchmarking
19. eDSLs and interpreters

Functional programming quickstart

Where do I get the environment?

use ghcup.

https://github.com/haskell/ghcup-hs

Where do I get the environment without GHCUP?

Windows/Mac: Install the Haskell Platform

- `https://www.haskell.org/platform/`

Unix-compatible: Use your favorite package system

- `apt-get install ghc`
- `yum install ghc`
- `pkg install ghc`
- ...
- ArchLinux: `ghc-static`, `cabal-static`!

In your own interest use some kind of unix.

Danger: Ubuntu is borderline unix.

Hello

```
$ cat > hello.hs <<EOHS
main = putStrLn "Well hello there!"
EOHS

$ runhaskell hello.hs
Well hello there!

$ ghc hello.hs -o hello
[1 of 1] Compiling Main  ( hello.hs, hello.o )
Linking hello ...

$ ./hello
Well hello there!
```

Interpreter

```
$ ghci hello.hs
GHCi, version 8.2.2
[1 of 1] Compiling Main  ( hello.hs, interpreted )
Ok, one module loaded.

*Main> :t main
main :: IO ()

*Main> main
Well hello there!

*Main> 5+5
10
```

Integration

Common tools: `hindent`, `hlint`, `ghcid`.

Editor integrations:

- haskell-language-server works with most IDEs, incl. VSCode
- vim: Haskell-vim-now, Syntastic, neco-ghc
- emacs: spacemacs + haskell-mode

For this course you should not need a fully configured IDE. Any text editor + commandline is OK.

Cool stuff

`play.haskell.org`

`hoogle.haskell.org`

Syntax

Haskell is declarative, program is a bunch of defined values:

```
valueName = valueDefinition

parametrizedValue parameter = otherDefinition
```

If there is a value called *main*, the runtime will “evaluate” it for “running” the program.

```
main = print (1 + 1)
```

The ‘=’ is supposed to mean actual math equality. This is good because humans are very good at equational reasoning.

Expressions syntax

- variable names start with lowercase letters: `a b hello`
- Literals: 123, 1.5, `'a'`, `"something"`
- 2 expressions next to each other = application of a function to the parameter
  - *negate* 5
  - *take* 3 `"abcde"`
- operators have lower parsing priority than the function application!
  - +, - ,++, ==, >=, <=, ...
  - *negate* 5 – 10 ⇝?
- parentheses work as expected:
  - *take* 1 + 2 `"doggo"`
  - *take* (1 + 2) `"doggo"`

Operators

You can define any operators you want with a given fixity and priority:

```
infixr 3 ++
infixl 6 /%/
infix 5 :=>
```

You can convert an operator to a function using parentheses:

```
5 + 10    ⇝ 15
(+) 5 10  ⇝ 15
```

You can convert a function to an operator using backticks:

```
mod 10 3    ⇝ 1
10 `mod` 3  ⇝ 1
```

Careful with unary minus operator.

Function syntax

You can define a function as a parametrized value:

$$discriminant\ a\ b\ c = b\text{^}2 - 4 * a * c$$

...or as a lambda:

$$discriminant = \lambda a\ b\ c \rightarrow b\text{^}2 - 4 * a * c$$

...or equivalently take apart:

$$discriminant = \lambda a \rightarrow \lambda b \rightarrow \lambda c \rightarrow b\text{^}2 - 4 * a * c$$

- you don't need to give names to lambdas
- formatting: we write $\lambda x \rightarrow y$ but the code uses backslash, minus and angle bracket:

```
\ x -> y
```

Local definitions

It is quite handy to have local helper definitions:

```
solve a b c =
  let d = b^2 - 4 * a * c
      solution op = (-b `op` sqrt d) / (2 * a)
  in (solution (+), solution (-))
```

The same, but with a math attitude:

```
solve a b c = (solution (+), solution (-))
  where d = b^2 - 4 * a * c
        solution op = (-b `op` sqrt d) / (2 * a)
```

Evaluation control

if *cond* **then** *a* **else** *b*

- *cond* must be a boolean value
- “if without else” does not make sense (what’s the type of an empty branch?)

Evaluation control 2

Pattern matching:

$$
\begin{aligned}
&fac\ 0 = 1 \\
&fac\ n = n * fac\ (n - 1)
\end{aligned}
$$

Pattern matching inside of a definition:

$$
\begin{aligned}
&fac\ n = \\
&\quad \textbf{case}\ n\ \textbf{of} \\
&\qquad 0 \longrightarrow 1 \\
&\qquad n \longrightarrow n * fac\ (n - 1)
\end{aligned}
$$

Evaluation control 3

Guards serve as multi-way **if**s:

$$
\begin{aligned}
&fac\ n \\
&\quad \mid n < 1 \qquad = 1 \\
&\quad \mid otherwise = n * fac\ (n - 1)
\end{aligned}
$$

Better:

$$
\begin{aligned}
&gcd\ x\ y \\
&\quad \mid x < y \ = gcd\ y\ x \\
&\quad \mid y \leqslant 0 = x \\
&\quad \mid x \geqslant y = gcd\ y\ (x\ `mod`\ y)
\end{aligned}
$$

Instead of loops, we can do infinite recursion.

Loops can be translated to recursion

Python:

```
def my_sum(a,b,increment):
  res = 0
  i = a
  while  i <= b:
    res += i
    i += increment
  return res
```

Haskell: iterators & accumulators are turned to parameters

```
mySum a b increment = step 0 a
  where
    step res i
      | i ⩽ b =
        step (res + i) (i + increment)
      | otherwise = res
```

...the “shape” of the loop can now be captured in a higher-level function:

```
mySum a b increment =
  foldl' (+) 0 [a, a + increment .. b]
```

Recursion depth

Recursion is unlimited because there's no stack to overflow.

The worst that can happen is running out of memory.

Use tail-recursion to prevent the memory issues.

Recursion depth has some technical limits

Recursion is unlimited **EXCEPT** for `ghci`, which uses a different evaluation strategy (closer to that in LISP/Scheme), to allow debugging.

Possible trigger case:

$$\mathit{factorial}\ 1 = 1$$
$$\mathit{factorial}\ n = n * \mathit{factorial}\ (n - 1)$$ – not a tail call!

```
ghci> factorial 0

*** Exception: stack overflow
```

Compiled with `ghc`, this instead yields a proper out-of-memory situation.

Types

Type names start with a capital letter. You can specify them using a quad-dot:

```
number :: Integer
number = 5
anotherNumber = (5 :: Float)
```

In `ghci`, you can determine the type of stuff using `:t`.

Function types

functionWithOneParam :: Int → Int
functionWithTwoParams :: String → Bool → Char
functionWithoutParams :: Float

(Functions without parameters are not distinguishable from values.)

Generic function types

$$id :: a \to a$$

...actually means: $id :: (\forall a)\ a \to a$, and neither of these holds:

- $id :: \mathsf{Int} \to a$
- $id :: a \to \mathsf{Int}$

Common stuff:

$$
\begin{aligned}
(,) &:: a \to b \to (a, b) \\
head &:: [a] \to a \\
(+) &:: \mathsf{Num}\ a \Rightarrow a \to a \to a \\
(\cdot) &:: (b \to c) \to (a \to b) \to (a \to c)
\end{aligned}
$$

We can read the extra 'condition' as: $(+) :: (\forall a \in \mathsf{Num})\ a \to a \to a$

Curry-style parameters

There is no such thing as multiparameter functions!

$$f :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$$
$$f \quad 1 \qquad 2 \qquad 3 \quad :: \mathsf{Int}$$

Feeding parameters to functions creates simpler functions (parentheses added to highlight default associativity):

$$f :: \mathsf{Int} \to (\ \mathsf{Int} \to\ (\ \mathsf{Int} \to \mathsf{Int}))$$
$$f \quad 1 \qquad :: \mathsf{Int} \to\ (\ \mathsf{Int} \to \mathsf{Int})$$
$$(f \ \ 1) \qquad 2 \qquad :: \mathsf{Int} \to \mathsf{Int}$$
$$((f\ 1) \qquad 2) \qquad 3 \qquad :: \ \mathsf{Int}$$

Related combinators:

$$curry :: ((a, b) \to c) \to (a \to b \to c)$$
$$uncurry :: (a \to b \to c) \to ((a, b) \to c)$$

Haskell Curry

Curry-style parameters

The concept was concieved by Schönfinkel (1924), popularly used only much later by Curry for combinatorial logic (in 1950's).

We also have combinators `schön`, `unschön`.

Moses Schönfinkel

“Arity pls.”

Comprehension check: How many parameters does the function $id :: a \rightarrow a$ have?

Comprehension check: How many parameters does the function *id* :: $a \rightarrow a$ have?

Correct answer: Parameter counts are nonsense!

id 5
id gcd 10 15
id id gcd 10 15
id

Some simple types

- 1 :: Int, Integer, Word, Int64, ...
- `'c'` :: Char, Word8, ...
- 1.23 :: Float, Double
- True :: Bool, False :: Bool
- `()` — empty-value type that exists
- Void — type of values that don't even exist

Some parametrized types

- $[a]$ — list of values of type a
  $[1, 2, 3] :: [\text{Int}]$
  $[1, 2, 3] :: [\text{Float}]$
  $[1, 2, 3] :: \text{Num } a \Rightarrow [a]$ (more generic)
- **type** String = [Char]
  `"hello"` :: [Char], String
- Maybe a
  Just 1 :: Maybe Int
  Nothing :: Maybe Int
  [Just `'c'`, Nothing] :: [Maybe Char]

Definition of algebraic data types

```
data Bool = False | True
data ()   = ()
data Void
```

Types that hold other values:

```
data CoupleOfInts = Couple Int Int
```

Construction:

```
Couple 2 3 :: CoupleOfInts
```

Destruction (a.k.a. pattern matching):

```
f :: CoupleOfInts → Int
f (Couple a b) = b
```

Parametrized types

Type parameters are written just like function parameters:

```
data Maybe a  = Nothing | Just a
data (,) a b      = (,) a b
data Either a b = Left a | Right b
data (→) a b      – internal, without definition
```

Maybe, (,), Either and (→) are *type constructors* (in the type language).

Nothing, Just, (,), Left and Right are *data constructors* (in the term language).

Convention:

- Type and data constructors start with a capital letter.
- Type and data bindings (“variables”) start with a lowercase letter.

(Purpose: code clarity, simpler parsing)

Parametrized type

```
data CoupleOf a = Couple a a
Couple 1 2 :: CoupleOf Int
Couple (Just 5) Nothing :: CoupleOf (Maybe Int)
Couple True 3    - type error
```

Use with pattern matching:

```
sumCouple :: Num n ⇒ CoupleOf n → n
sumCouple (Couple x y) = x + y
```

Another parametrized type

data BST *key* = Nil | Node (BST *key*) *key* (BST *key*)

- BST *a* is a type, you can make e.g. BST String or BST Int
- the type can hold either of 2 possibilities Nil and Node
- Nil holds no data
- Node holds 3 items (the key and another 2 trees)

```
sumKeys :: Num a ⇒ BST a → a
sumKeys Nil = 0
sumKeys (Node l k r) = sumKeys l + k + sumKeys r
```

You might have seen parametrized types elsewhere!

C++:

```
std::vector<float>
std::map<int,std::string>
```

C#:

```
List<float>
Dictionary<int,string>
```

Haskell:

```
Vector Float
Map Int String
```

Record syntax for ADTs provides named fields

$$\textbf{data}\ \text{Point} = \text{Point}\ \{px :: \text{Float}, py :: \text{Float}\}$$

The syntax provides:

- Acessors: $px, py :: \text{Point} \to \text{Float}$
- Constructor: $\text{Point}\ \{px = 1, py = 2\}$
- Patching syntax: $somepoint\ \{py = 2\}$
- Structured (possibly incomplete) pattern match:
  $rectangleArea\ \text{Point}\ \{px, py = height\} = px * height$

...generally a good idea for types with many fields.

Lists are built from small pairs

$$\textbf{data}\ [a] = (a : [a]) \mid [\,]$$

...desugared:

$$\textbf{data}\ [\,]\ a = (:)\ a\ ([\,]\ a) \mid [\,]$$

$$1 : 2 : 3 : [\,] \equiv 1 : (2 : (3 : [\,])) \equiv [1, 2, 3]$$

Kinds are types of types

How does the compiler check that you used the correct parameters in your type expressions?

Types of types are called Kinds. All types of values belong to the kind $*$.

$$
\begin{array}{ll}
\text{Bool} & :: * \\
\text{Int} & :: * \\
\text{Maybe} & :: * \to * \\
\text{Maybe Int} & :: * \\
[\,] & :: * \to * \\
(\to) & :: * \to * \to * \\
\text{RWST} & :: * \to * \to * \to (* \to *) \to * \\
\text{Num} & :: * \to \text{Constraint}
\end{array}
$$

In `ghci`, you can find the kinds of stuff using `:k`.

More type definitions

Type synonyms (both sides of the equation are equivalent)

```
type TwoInts = (Int, Int)
```

“Unboxed data” (the type will be typechecked, but will never appear at runtime):

```
newtype TwoInts = TwoInts (Int, Int)
```

Newtypes serve as zero-overhead labels

Common use:

```
newtype ProperlyEscaped = ProperlyEscaped String

escape :: String → ProperlyEscaped

makeHtmlButton :: ProperlyEscaped → ProperlyEscaped
makeHtmlButton (ProperlyEscaped label) =
  ProperlyEscaped $ "<button>" ++ label ++ "</button>"
```

- semantics depends on programmer choice
- you can hide the constructor from other programmers to enforce safety

Haskell types in C++ (very roughly)

data X *a* = X *a a* | Y *a* Int

```
template<typename a> struct X {
  enum {X,Y} _variant;
  union {
    struct {a fld1; a fld2;} X;
    struct {a fld1; int fld2;} Y;
  };
};
```

Haskell types in C++ (very roughly)

data X *a* = X *a a* | Y *a* Int

```
template<typename a> struct X {
  enum {X,Y} _variant;
  union {
    struct {a fld1; a fld2;} X;
    struct {a fld1; int fld2;} Y;
  };
};
```

newtype does not have the “box” with the variant label, and thus cannot have more variants.

Critical language properties

- **Haskell is lazy!** It will never evaluate something that is not explicitly needed.
  - Infinite data structures are okay: $allIntsFrom\ n = n : allIntsFrom\ (n + 1)$
  - *output polymorphism* code behavior is derived from how you consume the result
- **Haskell is referentially transparent!** Functions *are pure* and can be manipulated as in math.
  Outcome: the compiler can optimize a lot of stuff:
  $f\ x + f\ x \equiv \textbf{let}\ temp = f\ x\ \textbf{in}\ temp + temp \equiv 2 * f\ x$
- **Haskell is statically typed!**
  - you will never get a type error in runtime
  - compiler can choose the right code for given types, so that you don't have to

Basic stuff from Prelude

$$
\begin{array}{ll}
\mathit{fst} & :: (a, b) \to a \\
\mathit{snd} & :: (a, b) \to b \\
\mathit{show} & :: \mathsf{Show}\ a \Rightarrow a \to \mathsf{String} \\
\mathit{read} & :: \mathsf{Read}\ a \Rightarrow \mathsf{String} \to a \\
\mathit{putStrLn} & :: \mathsf{String} \to \mathsf{IO}\ () \\
\mathit{getLine} & :: \mathsf{IO}\ \mathsf{String} \\
\mathit{print} & :: \mathsf{Show}\ a \Rightarrow a \to \mathsf{IO}\ () \\
\mathit{readLn} & :: \mathsf{Read}\ a \Rightarrow \mathsf{IO}\ a \\
(\$) & :: (a \to b) \to a \to b \\
(\cdot) & :: (b \to c) \to (a \to b) \to (a \to c)
\end{array}
$$

Exercise: Guess what it does from the type.

Basic stuff from Prelude — lists

$$
\begin{aligned}
&\mathit{head}, \mathit{last} &&:: [a] \to a \\
&\mathit{tail}, \mathit{init} &&:: [a] \to [a] \\
&(+\!\!+) &&:: [a] \to [a] \to [a] \\
&(!!) &&:: [a] \to \mathsf{Int} \to a \\
&\mathit{elem} &&:: \mathsf{Eq}\ a \Rightarrow [a] \to a \to \mathsf{Bool} \\
&\mathit{length} &&:: [a] \to \mathsf{Int} \\
&\mathit{take}, \mathit{drop} &&:: \mathsf{Int} \to [a] \to [a] \\
&\mathit{takeWhile}, \mathit{dropWhile} &&:: (a \to \mathsf{Bool}) \to [a] \to [a] \\
&\mathit{lines}, \mathit{words} &&:: \mathsf{String} \to [\mathsf{String}] \\
&\mathit{unlines}, \mathit{unwords} &&:: [\mathsf{String}] \to \mathsf{String} \\
&\mathit{map} &&:: (a \to b) \to [a] \to [b] \\
&\mathit{filter} &&:: (a \to \mathsf{Bool}) \to [a] \to [a] \\
&\mathit{foldl} &&:: (a \to x \to a) \to a \to [x] \to a \\
&\mathit{foldr} &&:: (x \to a \to a) \to a \to [x] \to a \\
&\mathit{lookup} &&:: \mathsf{Eq}\ a \Rightarrow a \to [(a, b)] \to \mathsf{Maybe}\ b
\end{aligned}
$$