

Developing Haskell Applications

The Practice of Haskell Programming

Andres Löh

 Well-Typed

May 18, 2012

Goals of this course

- ▶ Learning about Haskell's toolchain and extended infrastructure.
- ▶ Writing robust and scalable code.
- ▶ Reasoning about evaluation and performance.
- ▶ A few more advanced Haskell concepts.

Goals of this lecture

- ▶ Some stylistic guidelines and conventions when writing Haskell programs:
 - ▶ layout and syntax,
 - ▶ code structure,
 - ▶ common programming pitfalls.
- ▶ Modules vs. packages.
- ▶ Introduction to helpful development tools:
 - ▶ Cabal and cabal-install,
 - ▶ HLint,
 - ▶ GHC warnings,
 - ▶ Haddock.

Layout and syntax

Never use TABs

- ▶ Haskell uses layout to delimit language constructs.
- ▶ Haskell interprets TABs to have 8 spaces.
- ▶ Editors often display them with a different width.
- ▶ TABs lead to layout-related errors that are difficult to debug.
- ▶ Even worse: mixing TABs with spaces to indent a line.

Never use TABs

- ▶ Haskell uses layout to delimit language constructs.
- ▶ Haskell interprets TABs to have 8 spaces.
- ▶ Editors often display them with a different width.
- ▶ TABs lead to layout-related errors that are difficult to debug.
- ▶ Even worse: mixing TABs with spaces to indent a line.

So:

- ▶ Never use TABs.
- ▶ Configure your editor to expand TABs to spaces, and/or highlight TABs in source code.

Alignment

- ▶ Use alignment to highlight structure in the code!
- ▶ Do not use long lines.
- ▶ Do not indent by more than a few spaces.

```
map :: (a → b) → [a] → [b]
map f []          = []
map f (x : xs) = f x : map f xs
```

Identifier names

- ▶ Use informative names for functions.
- ▶ Use CamelCase for long names.
- ▶ Use short names for function arguments.
- ▶ Use similar naming schemes for arguments of similar types.

Spaces and parentheses

- ▶ Generally use exactly as many parentheses as are needed.
- ▶ Use extra parentheses in selected places to highlight grouping, particularly in expressions with many less known infix operators.
- ▶ Function application should always be denoted with a space.
- ▶ In most cases, infix operators should be surrounded by spaces.

Blank lines

- ▶ Use blank lines to separate top-level functions.
- ▶ Also use blank lines for long sequences of `let`-bindings or long `do`-blocks, in order to group logical units.

Structuring the code

Avoid large functions

- ▶ Try to keep individual functions small.
- ▶ Introduce many functions for small tasks.
- ▶ Avoid local functions if they need not be local (why?).

Type signatures

- ▶ Always give type signatures for top-level functions.
- ▶ Give type signatures for more complicated local definitions, too.
- ▶ Use type synonyms.

```
checkTime :: Int → Int → Int → Bool
```

Type signatures

- ▶ Always give type signatures for top-level functions.
- ▶ Give type signatures for more complicated local definitions, too.
- ▶ Use type synonyms.

```
checkTime :: Int → Int → Int → Bool
```

```
checkTime :: Hours → Minutes → Seconds → Bool
```

```
type Hours    = Int
```

```
type Minutes  = Int
```

```
type Seconds  = Int
```

Comments

- ▶ Comment top-level functions.
- ▶ Also comment tricky code.
- ▶ Write useful comments, avoid redundant comments!
- ▶ Use Haddock.

Coding

Booleans

Keep in mind that Booleans are first-class values.
Negative examples:

```
f x | isSpace x == True = ...  
if x then True else False
```

Use (data)types!

- ▶ Whenever possible, define your own datatypes.
- ▶ Use `Maybe` or user-defined types to capture failure, rather than `error` or default values.
- ▶ Use `Maybe` or user-defined types to capture optional arguments, rather than passing `undefined` or dummy values.
- ▶ Don't use integers for enumeration types.
- ▶ By using meaningful names for constructors and types, or by defining type synonyms, you can make code more self-documenting.

Use common library functions

- ▶ Don't reinvent the wheel. If you can use a `Prelude` function or a function from one of the basic libraries, then do not define it yourself.
- ▶ If a function is a simple instance of a higher-order function such as `map` or `foldr`, then use those functions (why?).

Pattern matching

- ▶ When defining functions via pattern matching, make sure you cover all cases.
- ▶ Try to use simple cases.
- ▶ Do not include unnecessary cases.
- ▶ Do not include unreachable cases.

Avoid partial functions

- ▶ Always try to define functions that are total on their domain, otherwise try to refine the domain type.
- ▶ Avoid using functions that are partial.

Negative example

```
if isJust x then 1 + fromJust x else 0
```

Avoid partial functions

- ▶ Always try to define functions that are total on their domain, otherwise try to refine the domain type.
- ▶ Avoid using functions that are partial.

Negative example

```
if isJust x then 1 + fromJust x else 0
```

Use pattern matching!

Avoid partial functions

- ▶ Always try to define functions that are total on their domain, otherwise try to refine the domain type.
- ▶ Avoid using functions that are partial.

Negative example

```
if isJust x then 1 + fromJust x else 0
```

Use pattern matching!

Positive example

```
case x of  
  Nothing → 0  
  Just n   → 1 + n
```

Avoid partial functions – contd.

Negative example

$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

$\text{map } f \text{ } xs = \text{if null } xs \text{ then } []$

$\text{else } f (\text{head } xs) : \text{map } f (\text{tail } xs)$

Avoid partial functions – contd.

Negative example

```
map :: (a → b) → [a] → [b]
map f xs = if null xs then []
           else f (head xs) : map f (tail xs)
```

Positive example

```
map :: (a → b) → [a] → [b]
map f []      = []
map f (x : xs) = f x : map f xs
```

Use **let** instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Use **let** instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Questions

- ▶ Is there a semantic difference between the two pieces of code?

Use **let** instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Questions

- ▶ Is there a semantic difference between the two pieces of code?
- ▶ Could/should the compiler optimize from the second to the first version internally?

Let the types guide your programming

- ▶ Try to make your functions as generic as possible (why?).
- ▶ If you have to write a function of type `Foo → Bar`, consider how you can destruct a `Foo` and how you can construct a `Bar`.
- ▶ When you tackle an unknown problem, think about its type first.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int  
length xs      = ...
```

Look at the **type** of the input:

- ▶ **xs** is a list, so we can **pattern match**.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length []      = ...
length (x : xs) = ...
```

Follow the structure of the types:

- ▶ one pattern per constructor,
- ▶ try to **recurse** where the datatype is recursive!

The base case is easy to solve here.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length []      = 0
length (x : xs) = ...
```

Let us consider the second case:

- ▶ The `xs` are a (shorter) list.
- ▶ Let us try to recurse.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length []      = 0
length (x : xs) = ... length xs ...
```

It is now easy to complete the definition.

Let the types guide your programming – contd.

A function on lists:

```
length :: [a] → Int
length []      = 0
length (x : xs) = 1 + length xs
```

Done.

Let the types guide your programming – contd.

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int  
size t = ...
```

Look at the **type** of the input:

- ▶ **t** is a tree, so we can **pattern match**.

Let the types guide your programming – contd.

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int  
size (Leaf x)  = ...  
size (Node l r) = ...
```

Again, we follow the structure of the type

- ▶ one pattern per constructor,
- ▶ try to **recurse** where the datatype is recursive!

Again, we have an easy base case.

Let the types guide your programming – contd.

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int  
size (Leaf x)  = 1  
size (Node l r) = ...
```

Let us consider the second case:

- ▶ Both `l` and `r` are (smaller) trees.
- ▶ Let us try to recurse (twice).

Let the types guide your programming – contd.

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int  
size (Leaf x)  = 1  
size (Node l r) = ...size l ...size r...
```

It is now easy to complete the definition.

Let the types guide your programming – contd.

```
data Tree a = Leaf a  
           | Node (Tree a) (Tree a)
```

A function on trees:

```
size :: Tree a → Int  
size (Leaf x)  = 1  
size (Node l r) = size l + size r
```

Done.

Cabal and Hackage

Goals of Cabal

- ▶ A build system for Haskell applications and libraries.
- ▶ Easy to use for developers and users.
- ▶ Specifically tailored to the needs of a “normal” Haskell package.
- ▶ Tracks dependencies between Haskell packages.
- ▶ A unified package description format that can be used by a database.
- ▶ Platform-independent.
- ▶ Compiler-independent.
- ▶ Generic support for preprocessors, inter-module dependencies, etc. (make replacement).

Cabal is still in development; some goals have been reached, others not quite.

Cabal

- ▶ Cabal is itself packaged using Cabal.
- ▶ Cabal is integrated into the set of packages shipped with GHC, so if you have GHC, you have Cabal as well.

Homepage

<http://haskell.org/cabal/>

A Cabal package description

(Show on Hackage.)

A Setup file

```
import Distribution.Simple  
main = defaultMain
```

In almost all cases, this together with a Cabal file is sufficient.

A Setup file

```
import Distribution.Simple  
main = defaultMain
```

In almost all cases, this together with a Cabal file is sufficient.

If you need to do extra stuff (for instance, install some additional files that have nothing to do with Haskell), there are variants of `defaultMain` that offer hooks.

Hackage

- ▶ Online Cabal package database.
- ▶ Everybody can upload their Cabal-based Haskell packages.
- ▶ Automated building of packages.
- ▶ Allows automatic online access to Haddock documentation.

<http://hackage.haskell.org/>

Cabal vs. cabal-install

- ▶ The Cabal package provides a library with functions to support the packaging of Haskell libraries and tools.
- ▶ In particular, it specifies the format of the `.cabal` package description files.

Cabal vs. cabal-install

- ▶ The Cabal package provides a library with functions to support the packaging of Haskell libraries and tools.
- ▶ In particular, it specifies the format of the `.cabal` package description files.
- ▶ The `cabal-install` package provides a frontend to the Cabal library, providing the user with several commands to work with Cabal packages.
- ▶ Somewhat confusingly, the frontend contained in `cabal-install` is called `cabal`.
- ▶ The `cabal-install` package is contained in the Haskell Platform.

cabal-install

- ▶ A frontend to Cabal.
- ▶ Resolves dependencies of packages automatically, then downloads and installs all of them.
- ▶ Once cabal-install is present, installing a new library from Hackage is usually as easy as:

```
$ cabal update  
$ cabal install <packagename>
```

- ▶ You can also run `cabal install` within a directory containing a `.cabal` file.

Creating your own Cabal package

- ▶ Create a directory.
- ▶ Write initial program.
- ▶ Put it into version control (Subversion, git, darcs).
- ▶ Add a `.cabal` file.
- ▶ Add a `Setup.hs` or `Setup.lhs` file.
- ▶ Build using Cabal.
- ▶ Generate Haddock documentation using Cabal.
- ▶ Add a test suite.
- ▶ Use your version control system to run test suite on every commit (currently preferred by the Haskell community: git and darcs).

Preparing a release

- ▶ Tag the version in your version control system.
- ▶ Create a tarball via Cabal (or darcs).
- ▶ Upload to HackageDB (supported by the cabal frontend).

More details

<http://en.wikibooks.org/wiki/Haskell/Packaging>

Useful development tools

- ▶ A simple tool to improve your Haskell style.
- ▶ Developed by Neil Mitchell.
- ▶ Scans source code, provides suggestions.
- ▶ Makes use of generic programming (Uniplate).
- ▶ Suggests only correct transformations.
- ▶ New suggestions can be added, and some suggestions can be selectively disabled.
- ▶ Easy to install (via `cabal install`).

(Demo.)

GHC warnings

GHC can warn you about lots of potential mistakes:

- ▶ shadowing of identifier names,
- ▶ unused code,
- ▶ redundant module imports,
- ▶ non-exhaustive patterns in functions,
- ▶ use of deprecated functions,
- ▶ ...

By default, only a small fraction of these warnings are generated:

- ▶ use `-Wall` to enable all warnings,
- ▶ there are flags to selectively enable or disable specific sets of warnings.

Haddock

Haddock

- ▶ Haddock is a documentation generator for Haskell (like JavaDoc, Doxygen, ...)
- ▶ Parses annotated Haskell files.
- ▶ Most of GHC's language extensions are supported.
- ▶ API documentation (mainly).
- ▶ Program documentation (possible).
- ▶ HTML output.

Haddock annotations

```
-- | This is (redundant) documentation for  
-- function 'f'. The function 'f' is a badly  
-- named substitute for the normal  
-- /identity/ function 'id'.  
f :: a → a  
f x = x
```

- ▶ A `-- |` declaration affects the following top-level declaration.
- ▶ Single quotes as in `'f'` indicate the name of a Haskell function, and cause automatic hyperlinking. Referring to qualified names is also possible (even if the identifier is not normally in scope).
- ▶ Emphasis with forward slashes: `/identity/`.

More markup

Haddock supports several more forms of markup, for instance

- ▶ Sectioning to structure a module.
- ▶ Code blocks in documentation.
- ▶ References to whole modules.
- ▶ Itemized, enumerated, and definition lists.
- ▶ Hyperlinks.

Example

(Show on Hackage.)