EDAF40/EDAN40
Functional Programming

Compiling and testing Haskell programs

Christian.Soderberg@cs.lth.se

March 28, 2018

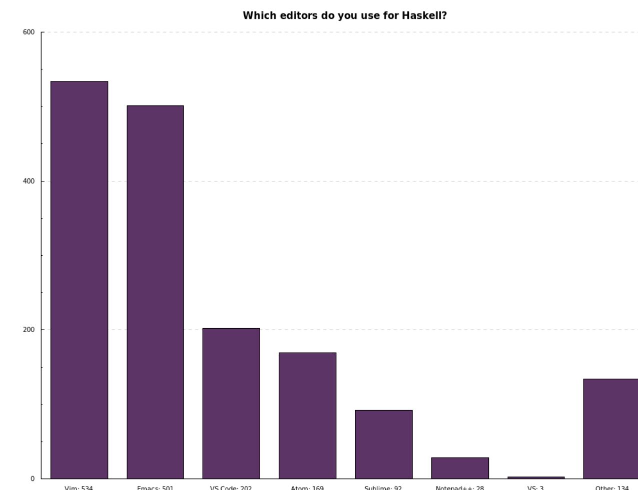
Today

- Tentative title: *Programming environment, testing, debugging*
- Editing Haskell code
- Compiling and using the REPL
- Using a build tool to work with a project
- Testing
- ~~Debugging~~
- Documenting
- There will be nothing new Haskell-wise

Editing Haskell code

- *Never use tabs in source code! Never ever!*
- Emacs, Vim, Sublime, and Atom all have great Haskell support
- VS Code, Intellij, and Eclipse have Haskell extensions

From ”State of Haskell, 2017”



Haskell compiler and REPL

- Today almost everyone uses GHC: Glasgow Haskell Compiler, aka. *The Glorious Glasgow Haskell Compilation System*
- Compiler: `ghc`
- Read-Evaluate-Print-Loop (REPL): `ghci`
- We seldom call the compiler directly, but use it from our build tool
- The REPL is useful for toying around, and trying things out
- The REPL has a couple of useful built-in commands, and can easily be configured to handle more commands (see lecture notes afterwards)

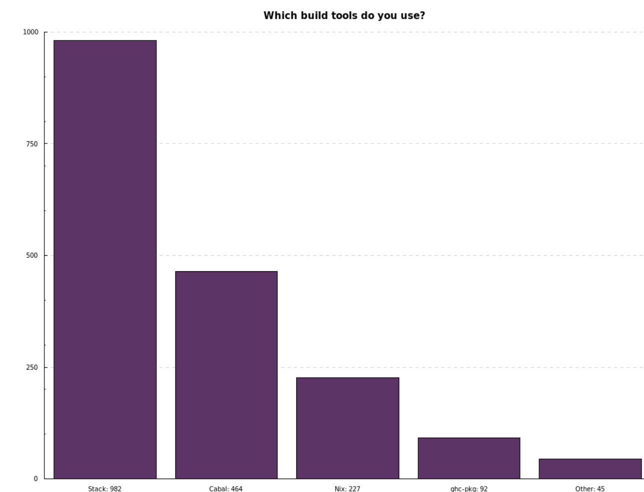
Haskell build tools: `cabal`

- To build anything interesting, we need to use libraries
- Traditionally, Haskell libraries have been built and installed using `cabal`
- `cabal` is several things, amongst them:
  - a format for describing packages (`.cabal` files)
  - a tool for building and installing packages
- Many packages can be found on *Hackage*: `https://hackage.haskell.org/`
- Although a great piece of software, `cabal` behaves in a way which is contrary to one of the pillars of functional programming: *calling it isn't guaranteed to produce the same result every time, even if you don't change your project*

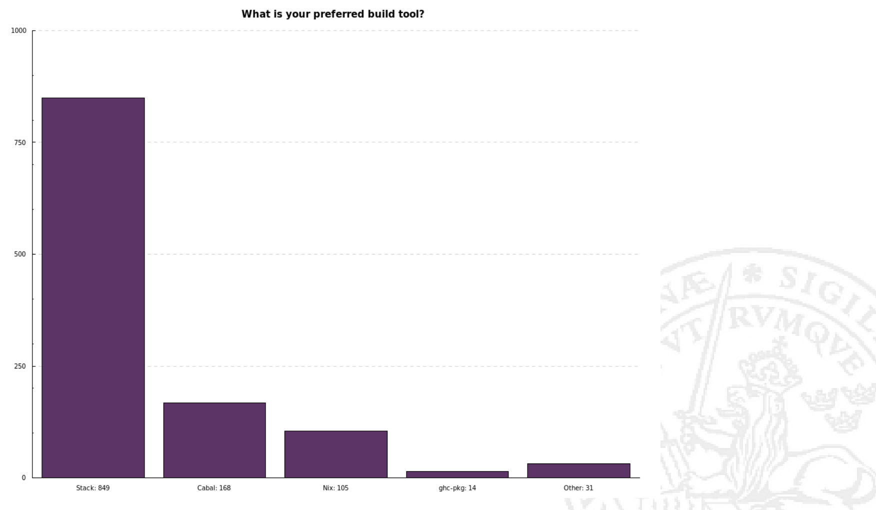
Haskell build tools: `stack`

- `stack` is a build tool which is built on top of `cabal`
- `stack` uses the same basic format as `cabal` for describing projects (`.cabal` files), but guarantees *repeatable builds*
- Curated releases (snapshots) of Haskell libraries can be found at `https://www.stackage.org/`
- Each `stack` release uses a specific version of `ghc`
- A `stack`-project contains a `stack.yaml` file in which we can specify which release we'll use
- `stack` downloads libraries and saves them in `~/.stack/` – beware that this directory can grow into several GB if we use many different releases
- You can find more information at `https://haskell-lang.org/`

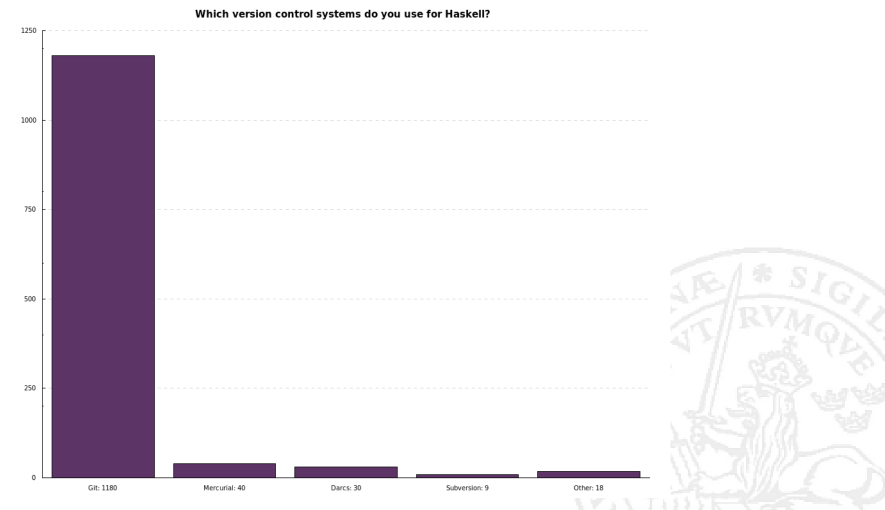
From "State of Haskell, 2017"



From ”State of Haskell, 2017”



From ”State of Haskell, 2017”



Installing and maintaining `stack`

- Installation: see `https://docs.haskellstack.org/en/stable/GUIDE/`
- Upgrade: `stack upgrade`
- Global config in `~/.stack/config.yaml` (user info) and `~/.stack/global-project/stack.yaml` (default release, etc.)
- Local config in yaml-files in the project directory (see stack docs)

Using `stack`

- We create our project with ’stack new’
- We set up our project with ’stack setup’
- We build our project with ’stack build’
- We test our program with ’stack test’
- We run our main program with ’stack exec <projectname>-exe’
- We install executables with ’stack install <package-name>’
- We start a REPL with ’stack ghci’ or ’stack repl’

Testing philosophy

- We should *never* ship code without proper testing
- Since we're going to write test code, we might as well do it before we write our business code:
  - It forces us to think about what functions we need, and how we want to call them (so, it helps us design)
  - Thinking about things to test is often a great way to learn about the problem
  - We get the benefits of testing during the whole process, and it makes refactoring much easier
- "Test First", or "Test Driven Development", is just one of many possible workflows, but I think you should try it at least once

Testing in Haskell

- Two common types of testing in Haskell:
  - Unit tests: we provide test data ourselves
  - Property based tests: we define what properties we want our code to have, and ask the test framework to generate test data
- `HUnit` is a popular tool for unit testing
- `QuickCheck` is a legendary tool for property testing
- Using the `Tasty` framework, we can easily use both unit tests and property based tests

Unit testing with `Tasty.Hunit`

- Import:

```
import Test.Tasty
import Test.Tasty.HUnit
```

- Add

```
dependencies:
- dups
- tasty
- tasty-hunit
```

to the `package.yaml` file (it will be translated into a `.cabal` file)

Unit testing with `Tasty.Hunit`

- Tests are grouped into `TestTrees`, where we create single tests (leaves) using `testCase`, and groups of tests (branches) using `testGroup`
- A single test:

```
testCase "empty list" $ hasDups "" @?= False
```

- A group of tests:

```
hasDupTests = testGroup "Unit tests for hasDups"
  [ testCase "empty list" $ hasDups "" @?= False
  , testCase "list with one element" $ hasDups "a" @?= False
  ]
```

- To run our tests, we call `defaultMain` from `main`, and tell it which tests we want to run:

```
main :: IO ()
main = defaultMain hasDupTests
```

Property based testing with `Tasty.QuickCheck`

- Import:

```
import Test.Tasty
import Test.Tasty.QuickCheck
```

- Add

```
dependencies:
- dups
- tasty
- tasty-hunit
- tasty-quickcheck
```

to the `package.yaml` file

Property based testing with `Tasty.QuickCheck`

- We can define a function which checks some property:

```
noDupsAfterRemove :: Eq a => [a] -> Bool
noDupsAfterRemove list = hasDups (removeDups list) == False
```

- This should work for any type a for which we've defined equality, but to make things easier for `QuickCheck`, we might as well use a specific type (you'll soon learn ways to write this more elegantly):

```
noDupsAfterRemove :: [Int] -> Bool
noDupsAfterRemove list = not $ hasDups (removeDups list)
```

- A property test can now be defined as:

```
testProperty "no duplicates after remove" noDupsAfterRemove
```

- If a property test fails, `QuickCheck` will try to find a minimal failing example
- `Tasty` lets us combine these property tests with `testGroup`, just is we did using `HUnit`

Property based testing, caveat

- Sometimes a property holds only in some cases – for `removeDups`, the first value in the input should be the first value of the output, *but only if the list isn't empty*
- We can write the property as:

```
firstSameAfterRemove :: [Int] -> Bool
firstSameAfterRemove list =
  head (removeDups list) == head list
```

- The test first checks that the list has at least one element:

```
testProperty "first element same after removeDups" $
\list -> not (null list) ==> firstSameAfterRemove list
```


Property based testing, using reference implementations

- For `removeDups`, there is a function nub with the exact same specification in `Data.List`
- We can use nub as a reference implementation, to test our own `removeDups`:

```
sameAsNub :: [Int] -> Bool
sameAsNub list = removeDups list == nub list
```

- This could be useful if we're trying to write a faster implementation of a function, and want to make sure it still returns the right values

Property based testing, using non-standard data types

- `QuickCheck` can generate data for many standard types
- If we want to use `QuickCheck` for our own data structures, we have to make them implement the typeclass `Arbitrary`, but it's often quite easy

Documenting

- It's very easy to generate documentation for our package, just by running

  ```
  $ stack haddock
  ```

  we create `.html`-files showing the signatures of our exported functions
- The generated documentation will reside deep inside the `.stack-work/` folder in your project
- To add text to your documentation, you just add Haddock annotations in the source code (see next slide)
- You can read much more in the Haddock documentation at `https://www.haskell.org/haddock/`

Haddock annotations

- `--` starts a regular comment
- `-- |` starts a documentation annotation (it ends at the first non-comment line)
- `-- ^` adds comments after a declaration
- You can use `/ .. /` for emphasis, and `__ .. __` for bold text
- You can hyperlink to identifiers using `'<id>'`
- You can hyperlink to modules using `"<mod>"`
- `-- *` inserts a heading in the documentation
- `-- **` inserts a sub-heading

Writing code samples in documentation

- We can write code inside our comments using >, e.g.,

  ```
  -- | Checks if a list contains duplicates
  --
  -- > hasDups "abc"
  --
  -- should return 'False'.
  hasDups :: (Eq a) => [a] -> Bool
  ...
  ```

- We can also write larger code blocks demarked by @-tags

Documenting and testing at the same time

- If we install `doctest` (using '`stack install`'), we can write tests in our documentation, and have the tests checked (just as `doctest` in Python)
- The example from the previous slide would be:

```
-- | Checks if a list contains duplicates
--
-- >>> hasDups "abc"
-- False
hasDups :: (Eq a) => [a] -> Bool
...
```

- We can check it using the command

```
$ stack exec doctest <source file>
```

Showing dependencies

- The command '`stack list-dependencies`' shows our dependencies
- We can also see the dependencies with:

```
$ stack dot
```

  To also see external dependencies, we write:

```
$ stack dot --external
```

- If we have installed `graphviz`, we can generate a nice graph using the command:

```
$ stack dot | dot -Tpng -o deps.png
```