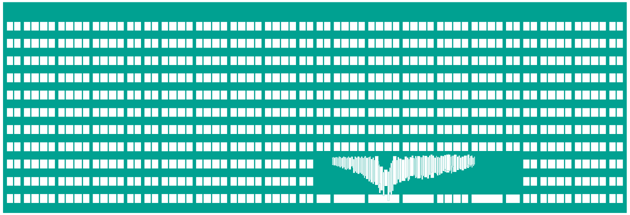
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Haskell Tools

behalek.cs.vsb.cz/wiki/Practical_Functional_Programming

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

October 31, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Haskell Tools

- This will be no comprehensive guide containing all possible development tool for Haskell.
- We will cover only some basic options.
  - We will really cover only one tool: *stack*.
  - The tool *stack* internally uses tool *cabal*.
  - All these tool can be installed by *ghcup*. You can check, what you have installed by:

```
ghcup list
```

- What we want to cover by this (short) presentation:
  - Create, build and run a project.
  - Add dependencies on libraries.
  - Debug a project.

Stack vs Cabal

- Stack is a build tool and it uses Cabal.
- Cabal defines the concept of a *package*.
    - A Cabal file, name, version, dependencies on other packages, and more.
- Stack defines a new concept called a *project*.
    - A *resolver*, which tells it about a *snapshot*.
    - Extra dependencies on top of the snapshot.
    - Optionally, one or more local Cabal packages.
    - Flag and GHC options configurations.
    - And a bunch more Stack configuration.

Creating Stack project

- Libraries and dependencies change frequently $\rightarrow$ `stack update`
- Several project templates (`stack templates`)
  - $simple$ - just `.cabal` configuration file, simple $main$.
  - (default) $new - template$ - provides `package.yaml`, it is used generate a corresponding `.cabal` file, contains $main$ along with a simple library and tests.

```
stack new MyProject
```

- Main configuration files:
  - `stack.yaml` contains project-level configuration for Stack, and may contain project-specific options and non-project-specific options.
  - `package.yaml` contains a description of a package in the $Hpack$ format. $Hpack$, including Stack's built-in version, uses the file to create a Cabal file.
  - Cabal file also contains a description of a package, but in the format used by Cabal.

Dependencies

- `stack.yaml`: "If at any point you find that you need to build the $acme - missiles$ package, please use version 0.3"

```
extra-deps:
- acme-missiles-0.3
```

- `package.yaml`: "Please build $acme - missiles$ now."

```
dependencies:
- acme-missiles
```

- $Cabal$ file: "This package requires that $acme - missiles$ be available."

```
build-depends: acme-missiles
```

Building a project

- Initialization: `stack setup` (gets GHC)
- Build: `stack build`
- Clean: `stack clean`
- Running the project:

```
stack exec myProject-exe arg1 arg2
stack run
```

- Run *ghci* in context of packages:

```
stack repl
stack ghci
```

Debugging in GHCi (1)

1. *Breakpoints* - the ability to set a breakpoint on a function definition or expression in the program.
2. *Single-stepping* - the evaluator will suspend execution approximately after every reduction, allowing local variables to be inspected. This is equivalent to setting a breakpoint at every point in the program.
3. Execution can take place in tracing mode, in which the evaluator remembers each evaluation step as it happens, but does not suspend execution until an actual breakpoint is reached. When this happens, the history of evaluation steps can be inspected.
4. Exceptions (e.g. pattern matching failure and error) can be treated as breakpoints, to help locate the source of an exception in the program.

Debugging in GHCi (2)

- Breakpoints
  - We can set a break point: `:break [module] line [column]` or `:break identifier`
  - Show breakpoints: `:show breaks`
  - Show context: `:show context`
  - Disable break-point: `:disable 0`
  - Delete break-point: `:delete *`
  - List a source code around break point: `:list`
  - To print variables we can use: `:print` and `:force`
  - Continue to the next break-point: `:continue`, abandon: `:abandon`
- Single-stepping
  - Enable a break points for whole program: `:step [expression]`

Debugging in GHCi (3)

- Tracing
  - You need to set breakpoints, then run a tracing: `:trace [expression]`
  - We can move in this trace log: `:back [n]` and `:forward [n]`
- Debugging exceptions
  - Setting breakpoints on exceptions: `::set -fbreak-on-exception`
  - Setting breakpoints on errors: `::set -fbreak-on-error`

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

October 31, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE