Pythondo

A web system for automatic feedback of programming assignments

Pedro Vasconcelos

DCC/FCUP & LIACC

May 7, 2014

Motivation

"*Introdução à Programação*" (CC101/ECC101)

- Introductory programming course at FCUP
- For 1st year students of most majors:
  - Astronomy
  - Biology
  - Physics
  - Mathematics
  - Chemistry
  - Engineering Sciences

Motivation (cont.)

Challenges:

- 405 students enrolled (in 2013);
- large disparities in student's background and motivation;
- many realize their difficulties only by failing the first exam;
- low success figures (35% passes in 2012).

Can we help students learn programming in a more effective way?

Methodology

- Programming is a primarily a writing skill.
- Largest dificulty: expressing yourself in an unambiguous notation.
- Students should be encouraged to write many short programs as soon as possible.
- Aim: quality assurance rather than quality control

Methodology (cont.)

Getting feedback is essential to consolidate learning:

correctness: does my program produce the right answer?

structure & style: is it expressed clearly?

Goal

Employ testing to give automatic feedback on correctness.

(Structure and style still requires teacher feedback.)

Alternatives

. What about *Mooshak*?

- initially intended for ACM-style programming contests;
- also often used for teaching at DCC.

Alternatives (cont.)

Positives:

- tried and tested;
- language agnostic;
- good administrative interface.

Alternatives (cont.)

Negatives:

- submissions must deal with I/O by default;
- uninformative feedback (but can be overridden with some effort);
- some misfit between objectives of contests & training;
- scalability issues due to a dated implementation (CGI scripts).

Assessment

- Some advantages in developing a specialized system for teaching.
- An opportunity to try Haskell web programming.

Pythondo

- A system for evaluating programming assignments
- Specific for the Python language:
  - no need to deal with I/O;
  - allows testing fragments (functions, methods, classes, etc.);
  - test feedback mimics the Python shell;
  - failed test cases is always reported.
- Implemented in Haskell (plus some Python & JS)

Try it now: `http://ipminor.dcc.fc.up.pt`

About the name

Python a general-purpose dynamic programming language by Guido van Rossum.

dō (from Japonese):

1. path, road, street;
2. method, way;
3. say.

Why Haskell?

All the usual reasons:

1. High-level development
2. Correctness
3. Performance

Really. . . *why* Haskell?

Web servers are kind of functional anyway:

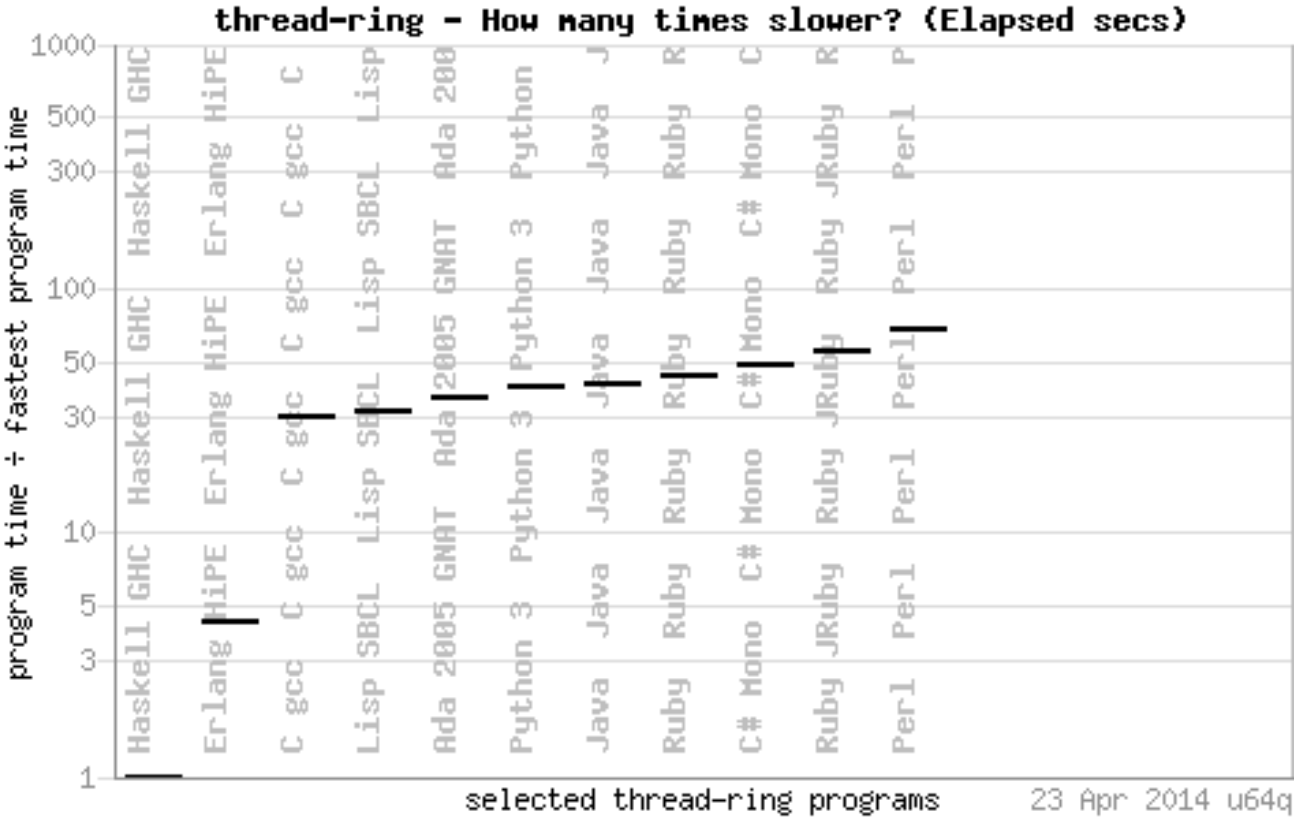
1. wait for a request;
2. process it;
3. render a response.

Mostly consists of converting between data formats:

- HTTP requests
- HTML/XML documents;
- JSON documents;
- . . .

Really. . . *why* Haskell? (cont.)

GHC is *really* good at concurrency.



Thread ring benchmark from

`http://benchmarksgame.alioth.debian.org`

Really. . . *why* Haskell? (cont.)

- Several frameworks for web programming:
  - Happstack
  - Yesod
  - Snap
  - Scotty
  - . . .
- Different approaches, levels of functionality, documentation, stability, etc.

The *Snap* web framework

`http://snapframework.com/`

- Just a bunch of Haskell libraries
- Includes a fast HTTP server library
- A sensible and clean monad for web programming
- Many optional *snaplets*:
  - HTML-based templating system for generating pages
  - User sessions, authentication
  - File uploading
  - Database access, ACID state
  - ...
- Some “industrial” users (e.g. Janrain, Soostone Inc)

Hello, Snap!

```
main :: IO ()
main = quickHttpServe site

site :: Snap ()
site = route [ ("hello", writeBS "Hello world!")
             , ("hello/:name", helloHandler)]
       <|> dir "static" (serveDirectory "files")

helloHandler = method GET $ do
    opt <- getParam "name"
    writeBS $ case opt of
       Nothing -> "must specify name"
       Just name -> BS.append "Hello, " name
```

Pythondo architecture

User authentication via department LDAP:

- avoids the need to create users, set passwords, etc;
- less hassle for 1st year students;
- no need to limit users: secure execution must be ensured anyway.

Pythondo architecture (cont.)

API is vaguely REST-like:

GET /problems fetch list available problems

GET /problems/:pid fetch a specific problem

GET /submissions/:pid/:sid fetch a submission

GET /submissions/:pid fetch all submissions

POST /submissions/:pid post a new submission

Pythondo architecture (cont.)

Evaluating submissions:

- executes a separate Python process;
- under a “safe-exec” environment;
- *doctest* script for each problem;
- typically 50–100 test cases organized from simpler to more complex;
- reports *sucess* or the *first failed* test case.

Pythondo architecture (cont.)

Output HTML generated using the *Heist* template library:

- separates the presentation layer from the internal logic;
- allows changes to styling, language, etc. without modifying Haskell code;
- Snap also serves static files (CSS, JavaScript libraries, etc.)

Conclusions

- Used for mandatory weekly exercises (27 in total)
- Could be complete in labs or elsewhere
- Students required to successfully complete half to attend exam
- No attempt to avoid plagiarism during classes
- Also used in exam (in a controlled environment)

The good

1. Types
2. Libraries
3. Refactoring
4. Performance and reliability

Types

```
newtype UID = UID { fromUID :: ByteString }

newtype PID = PID { fromPID :: ByteString }

newtype SID = SID { fromSID :: Int }
```

- No way to mix different IDs
- Parsing and pretty-printing using Show/Read instances

Types (cont.)

```
data Problem t = Problem {
  probID     :: PID,      -- unique id
  probTitle  :: Text,     -- title
  probDescr  :: [Node],   -- description (HTML)
  probSubmit :: Text,     -- default submission
  probStart  :: Maybe t,  -- optional start time
  probEnd    :: Maybe t,  -- optional end time
  probExam   :: Bool      -- is an exam problem?
  }
```

- Describe the shape of data precisely
- Parametric over the *type of times* (for parsing)
- *Maybe* types: no null pointer exceptions!

Libraries

Some very good general-purpose libraries used:

- **parsec** parsing using combinators;
- **configurator** processing configuration files;
- **ekg** remote monitoring of Haskell processes.

Refactoring

- Changes to interfaces are much easier with static types.
- If it still typechecks, then it will almost never fail at runtime!

Performance and reliability

- Deployed on a virtual machine (1–4 GB, 1–4 cores)
- Snap process uses about 20MB
- Splits work on all available cores (lightweight threads)
- Server ran unattended for weeks (no crashes)
- Peak stress test: exam (around 90 simultaneous users)
  - No change in resident space
  - Should be able to handle many more users

The bad

The *Cabal* build system:

- packages can depend on other packages;
- may demand *lower* and *upper* version bounds;
- conflicts when two versions of the same package are required;
- *sandboxes* improves the situation somehow.

The ugly

Many different string-like types:

String	lists of Unicode chars (simple but inefficient);
ByteString	vectors of bytes (strict and lazy versions);
Text	Unicode text (strict and lazy versions)

- Requires explict conversions (no subtyping);
- May have performance costs

Thank you!

Questions?