Using Haskell Professionally

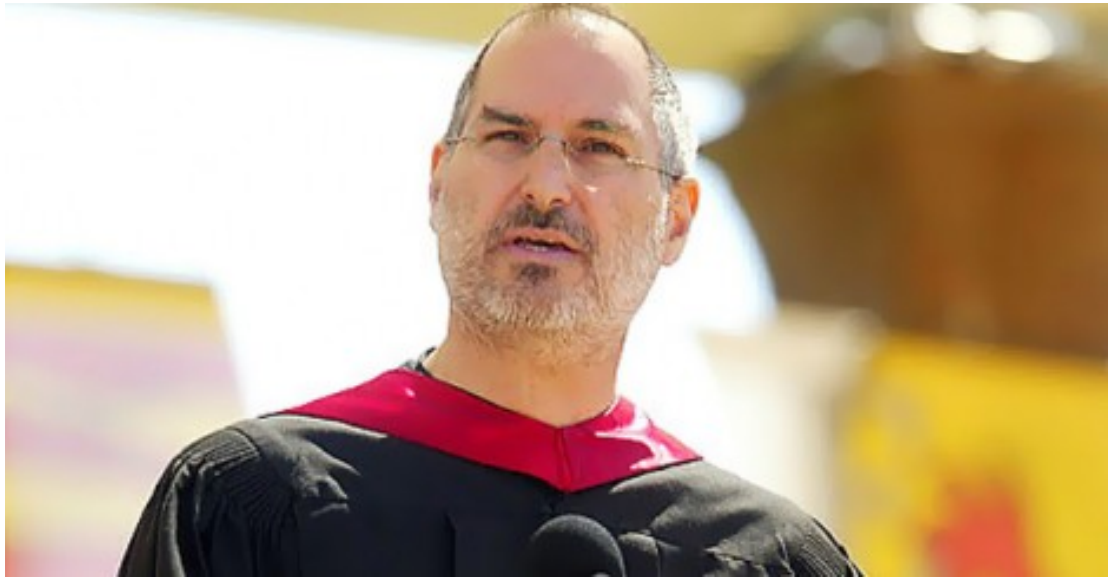
Alfredo Di Napoli

These slides are available at URL.

External links are references are provided at the end.

Quoting Steve Jobs..

Today, I'm gonna tell you three stories.



- My story
- My company story
- Your story

The early days

Back in 2007, I had my first exposure to FP, under the form of a university course. The course taught us OCaml.

The early days (contd.)

Back in 2007, I had my first exposure to FP, under the form of a university course. The course taught us OCaml.

After the initial enthusiasm, I drifted back to Python and other OOP languages.

Fast-forward 2009..

I was still in love with Python. At the time I was a regular attendees of **Python-it.org**, a popular Italian community.

I fell in love with Clojure.

Pushing myself forward in my holy grail search, I was exposed to a huge number of programming language, trying to find the “perfect” one: Scheme, Clojure, Common Lisp, **Haskell**, Io, Ruby, *put-yet-another-language-here*.

So, despite the interest, I did went back to Clojure and lisp-family languages.

Fast-forward 2011..

I was writing my masters degree dissertation on *Parallel and Distributed Computing*, using nothing but C++. The topic, together with the exposure to C++ revamped in me the holy flame of fast and compiled languages.

Fast-forward 2011.. (contd.)

I was writing my masters degree dissertation on *Parallel and Distributed Computing*, using nothing but C++. The topic, together with the exposure to C++ revamped in me the holy flame of fast and compiled languages.

Being Haskell a compiled language, I wanted to give it another go, so I bought *Learn You Some Haskell for Great Good* and worked my way through it.

Mid 2012

I started an internship with a company in the defense field, doing C++ in Rome. To hone my Haskell skills I tried to contribute to an Haskell open source project, the Snap framework.

Mid 2012

Being determined in earning a living with functional programming, I decided to concentrates my efforts only on three languages, based on different criteria (commercial users, personal preference, job offers abroad):

- Haskell
- OCaml
- Scala

The Manchester era

2 DAY COURSE

Well-Typed's Fast Track to Haskell

Why Haskell?

- Because software development is a marathon, not a sprint.

"It took me more time writing the specs that implementing the feature itself."

“Pros” of working in Haskell

- Refactoring is a dream
- EDSLs are a piece of cake
- Makes impossible states unrepresentable
- High quality libraries

Refactoring is a dream

EDSLs are a piece of cake

```
fromPreset :: MediaFile -> MediaFile
           -> Maybe Atlas.VideoFilter
           -> VideoPreset -> Maybe VideoRotation
           -> LogLevel -> [T.Text]
fromPreset filename outFilePath flt vpres vi ll =
  let cli = ffmpegCLI $ mconcat [
              i $ toTextIgnore filename
            , loglevel ll
            , fromVideoPreset vpres
            , isVideoRotated vi <?> resetRotateMetadata
            , yuv420p
            , vf [rotateMb vi]
            , isJust flt <?> vf_technicolor
            , o_y_ext (toTextIgnore outFilePath) (Left vpres)
            ]
  in T.words cli
```

Makes impossible states unrepresentable

Real world scenario:

```haskell
-- | Creates a new Supervisor. Maintains a map <ThreadId, ChildSpec>
newSupervisor :: IO Supervisor

-- | Start an async thread to supervise its children
supervise :: Supervisor -> IO ()

-- | forkIO-inspired function
forkSupervised :: Supervisor -> RestartStrategy -> IO () -> IO ThreadId
```

Example usage:

```
main = do
  sup <- newSupervisor
  supervise sup
  _ <- forkSupervised sup OneForOne $ threadDelay 1000000 >> print "Done"
```

Can you think to a potential bug?

Nothing in the types is forcing us to call `supervise` before actually supervise some thread!

```haskell
main = do
  sup <- newSupervisor
  -- Wrong! We forgot to start the supervisor...
  _ <- forkSupervised sup OneForOne $ threadDelay 1000000 >> print "Done"
```

As Haskellers, we can certainly do better!

GADTs to the rescue!

GADTs (Generalised Algebraic Data Types) allow us to constrain the type in the polimorphic type variable (a in the example).

```
-- Two inhabitated types (no constructors)
data Uninitialised
data Initialised

data Supervisor_ a where
     NewSupervisor :: {
      -- record fields (omitted)
      } -> Supervisor_ Uninitialised
     Supervisor :: {
      -- record fields (omitted)
      } -> Supervisor_ Initialised

-- Client-friendly type synonyms

type SupervisorSpec = Supervisor_ Uninitialised
type Supervisor = Supervisor_ Initialised
```

Let's now slightly change our API to be this: snippet again...

```
-- | Creates a new Supervisor. Maintains a map <ThreadId, ChildSpec>
newSupervisor :: IO SupervisorSpec

-- | Start an async thread to supervise its children
supervise :: SupervisorSpec -> IO Supervisor
```

What did we get? Let's try to run the "wrong" snippet again...

```
main = do
  sup <- newSupervisor
  _ <- forkSupervised sup OneForOne $ threadDelay 1000000 >> print "Done"
```

GHC will complain:

```
Couldn't match type ‘Control.Concurrent.Supervisor.Uninitialised’
         with ‘Control.Concurrent.Supervisor.Initialised’
Expected type: Supervisor
Actual type: Control.Concurrent.Supervisor.SupervisorSpec
```

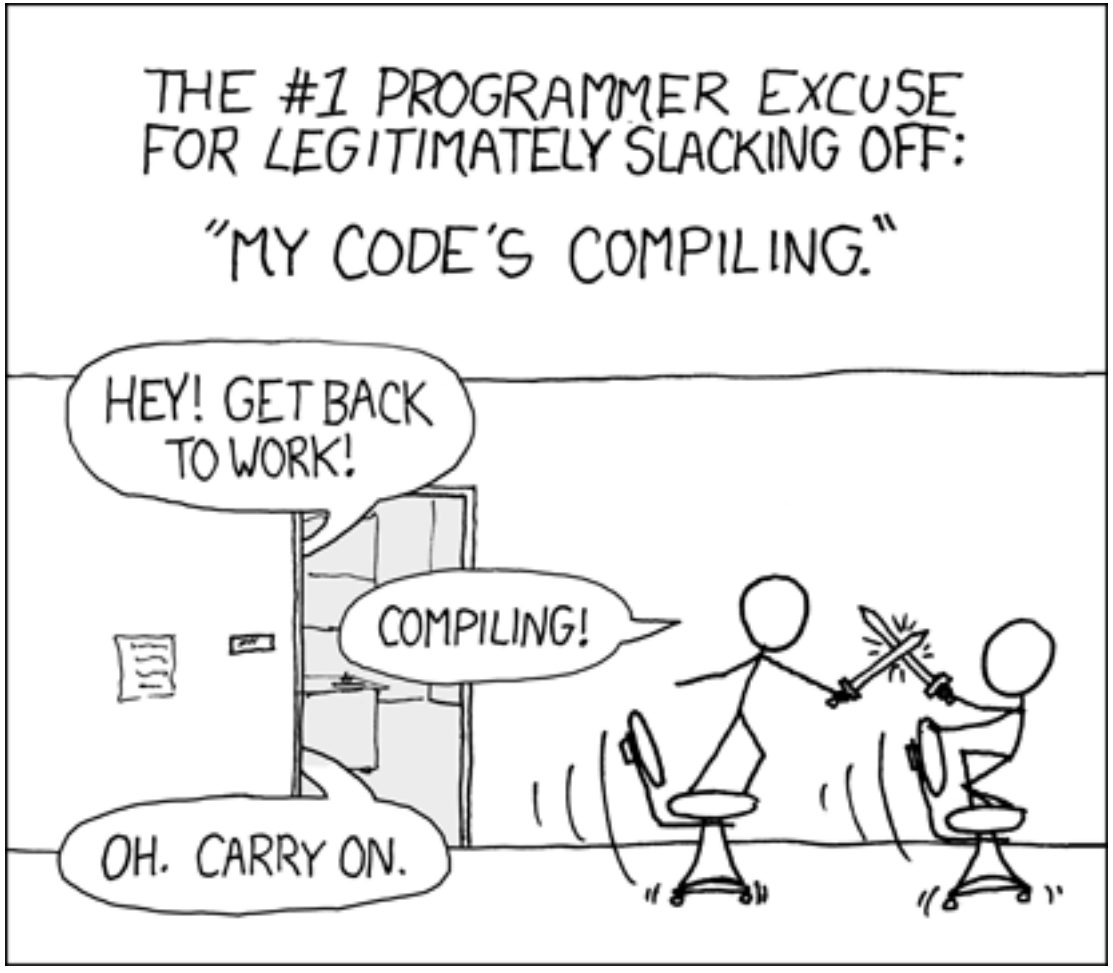
- This is because now we require a `Supervisor` to be initialised first
- The type system prevented us making a silly mistake
  - Failed with a very useful error message
- Profit!

This is just a small example (this is only one of the possible solutions), but the benefits are real.

Snags of working in Haskell

- Slow(ish) Compilation
- Cabal Hell
- ???

Slow(ish) compilation



Slow compilation: the caveat

- It's a problem all non-interpreted languages have to deal with
- GHC indeed does incremental compilation, building only what's changed
- It's even slower if..
  - You have TH (Template Haskell) in your code
  - You are building with profiling enabled

If you want faster feedback loop, consider using ghci

The typical Sinatra hello world app...

```
require 'sinatra'

get '/hi' do
  "Hello World!"
  end
```

. . . and the Snap equivalent

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Snap

main :: IO ()
main = quickHttpServe $ do
  route [("/hi", method GET $ writeBS "Hello World!")]
```

In a nutshell...

Haskell is a very pragmatic language...

In a nutshell...

Haskell is a very pragmatic language...

...but not all the people using it are!

Your story

What can **you** do (as a community) to embrace, support and spread functional programming?

Be pragmatic.

Keep an open mind.

Reject the status-quo.

Quoting Bret Victor's "The future of programming"

[..]So the most dangerous thought that you can have as a creative person is to think that you know what you're doing. Because once you think you know what you're doing, you stop looking around for other ways of doing things. And **you stop being able to see other ways** of doing things.[..]

[..] if you don't want to be this guy, if you want to be open or receptive to new ways of thinking, to invent new ways of thinking, I think the first step is you have to say to yourself, **"I don't know what I'm doing.** [..] I think you have to say,"we don't know what programming is. We don't know what computing is. We don't even know what a computer is."

And once you truly understand that - and once you truly believe that - then you're free.

Thank you.

Questions?

External references

- My road to Haskell