Developing F# on NixOS

Karsten Gebbert

27/10/2015

Hi.

Introductions

Who? Why?

- this talk is meant as an introduction to developing F# on NixOS

Who? Why?

- this talk is meant as an introduction to developing F# on NixOS
- I am not a Nix(OS) or F# expert or .NET veteran (yet)

Who? Why?

- this talk is meant as an introduction to developing F# on NixOS
- I am not a Nix(OS) or F# expert or .NET veteran (yet)
- sharing my personal experience and impressions

Who? Why?

- this talk is meant as an introduction to developing F# on NixOS
- I am not a Nix(OS) or F# expert or .NET veteran (yet)
- sharing my personal experience and impressions
- approaches taken have a few rough edges, so please help me improve it!

Who? Why?

- this talk is meant as an introduction to developing F# on NixOS
- I am not a Nix(OS) or F# expert or .NET veteran (yet)
- sharing my personal experience and impressions
- approaches taken have a few rough edges, so please help me improve it!
- it is my first experience of this kind, so please do criticize me (gently)

What?

- explore some basic aspects of F#

What?

- explore some basic aspects of F#
- look at a library for creating http services, Suave.IO

What?

- explore some basic aspects of F#
- look at a library for creating http services, Suave.IO
- give an overview over recoll, a file indexer

What?

- explore some basic aspects of F#
- look at a library for creating http services, Suave.IO
- give an overview over recoll, a file indexer
- construct a small service to query recoll via HTTP

What?

- explore some basic aspects of F#
- look at a library for creating http services, Suave.IO
- give an overview over recoll, a file indexer
- construct a small service to query recoll via HTTP
- deploy that service using nix

What is F#?

F#

F# was developed in 2005 at Microsoft Research[1]. In many ways, F# is essentially a .Net implementation of OCaml, combining the power and expressive syntax of functional programming with the tens of thousands of classes which make up the .NET class library.

Overview

- functional-first CLI programming language in the ML family

Overview

- functional-first CLI programming language in the ML family
- object-orientation

Overview

- functional-first CLI programming language in the ML family
- object-orientation
- the ‘m’-word (look at the computation-food paper that I downloaded)

Overview

- functional-first CLI programming language in the ML family
- object-orientation
- the ‘m’-word (look at the computation-food paper that I downloaded)
- ecosystem seems somewhat fragmented, but there are many useful libraries out there

Pros:

- For those who (have to) write software for the CLR its a solid choice

Pros:

- For those who (have to) write software for the CLR its a solid choice
- With projects like WebSharper or FunScript, F# can be used throughout the whole stack (share types and code, thus safety)

Pros:

- For those who (have to) write software for the CLR its a solid choice
- With projects like WebSharper or FunScript, F# can be used throughout the whole stack (share types and code, thus safety)
- Interop wth C# works really well, and there are lots of good libraries

Pros:

- For those who (have to) write software for the CLR its a solid choice
- With projects like WebSharper or FunScript, F# can be used throughout the whole stack (share types and code, thus safety)
- Interop wth C# works really well, and there are lots of good libraries
- its essentially a really good blend between the principled and utilitarian mind-sets

Cons:

- it introduces new nomenclature for common fp concepts (monads), to create a more clear distinction to other languages, which confused me more than it helped

Cons:

- it introduces new nomenclature for common fp concepts (monads), to create a more clear distinction to other languages, which confused me more than it helped
- TODO: find out differences in the type systems

Cons:

- it introduces new nomenclature for common fp concepts (monads), to create a more clear distinction to other languages, which confused me more than it helped
- TODO: find out differences in the type systems
- no GADTs

Cons:

- it introduces new nomenclature for common fp concepts (monads), to create a more clear distinction to other languages, which confused me more than it helped
- TODO: find out differences in the type systems
- no GADTs
- while OO has some points to go for it (think familiarity to large audiences of developers, generally well understood) it is a bit ugly and alien in this context.

Differences from Haskell

- TODO: list up different operators

Differences from Haskell

- TODO: list up different operators
- TODO: explain fixity rules

Differences from Haskell

- TODO: list up different operators
- TODO: explain fixity rules
- TODO: show a small implementation of a ComputationBuilder

Differences from Haskell

- TODO: list up different operators
- TODO: explain fixity rules
- TODO: show a small implementation of a ComputationBuilder

- impure: *launchMissiles ()* whereever you feel like it

Differences from Haskell

- TODO: list up different operators
- TODO: explain fixity rules
- TODO: show a small implementation of a ComputationBuilder

- impure: *launchMissiles ()* whereever you feel like it
- no `where`

Let's look at some code

The -calculus

```
// variable binding
let x = 41

// functions
let f = fun (value : int) -> value + 1

let f value = value + 1 // shorter

let f = (+) 1 // partially applied

// function application
f x

// a common idiom in F# is the `apply to` operator
x |> f
```

Types

```
// a binary tree - example of a sum type
type Tree<'a> =
  | Node of Tree<'a> * int * Tree<'a>
  | Leaf of 'a

// another "discriminated union" for modeling state changes
type AppAction =
  | AddThing
  | EditThing
  | RemoveThing

// record - a product type
type Person = { name : string; age : int }

// optional (aka. Maybe)
type option<'a> =
  | Some of 'a
  | None

// type alias
```

Classes

```
// Objects o.O
type Person (a: int, n: string) =
  let mutable name = n
  let mutable age = a

  member self.Name           // properties
    with get () = name       // getter ->
     and set n  = name <- n // setter <-

  member self.Age
    with get () = age
     and set a  = age <- a

  member self.OldEnough () = age > 18

  static member Greet () = printfn "Hi."

// usage
```

Pattern Matching

```
let horse : string option = Some "Hi."

// handling all cases with match
match f with
  | Some "Hello." -> printfn "it said hello."
  | Some "Hi."    -> printfn "it said hi."
  | Some _        -> printfn "it said something else."
  | None          -> printfn "it does not speak."
```

TODO more examples for PM

Modules and ..

```
// declare a module locally
module MyTree =
  // indent!
  type Tree<'a> =
    | Node of Tree<'a> * 'a * Tree<'a>
    | Leaf

// top-level definition
module MyTree


(*
  - declare at the top of file
  - no =
  - no indentation!
*)
let testTree depth =
  let rec testTree' current max =
    let next = current + 1
    if current = (max - 1)
```

.. Namespaces

```
namespace Data

module MyTree =
  // no indentation!
  type Tree<'a> =
    | Node of Tree<'a> * int * Tree<'a>
    | Leaf of 'a

// combine namespace and module into one statement
module Data.MyTree

// again, no indentation!
type Tree<'a> =
  | Node of Tree<'a> * int * Tree<'a>
  | Leaf of 'a
```

Other Cool Things To Look At

- Type Providers

Other Cool Things To Look At

- Type Providers
- Monads, or *Computation Expressions*

Other Cool Things To Look At

- Type Providers
- Monads, or *Computation Expressions*
- Quotations & Reflection (metaprogramming)

Other Cool Things To Look At

- Type Providers
- Monads, or *Computation Expressions*
- Quotations & Reflection (metaprogramming)
- Units of Measure

Other Cool Things To Look At

- Type Providers
- Monads, or *Computation Expressions*
- Quotations & Reflection (metaprogramming)
- Units of Measure
- built-in support for Actor-style programming

F# and NixOS

- F# currently is packaged separately from mono

- F# currently is packaged separately from mono
- as a consequence, there is no single GAC (Global Assembly Cache) for all .NET packages

- F# currently is packaged separately from mono
- as a consequence, there is no single GAC (Global Assembly Cache) for all .NET packages
- package management is traditionally done using *nuget* and an IDE front-end (this might make some people in the audience flinch)

- F# currently is packaged separately from mono
- as a consequence, there is no single GAC (Global Assembly Cache) for all .NET packages
- package management is traditionally done using *nuget* and an IDE front-end (this might make some people in the audience flinch)
- *paket* is a very promising replacement setting out to fix the common problem of *DLL* hell

- F# currently is packaged separately from mono
- as a consequence, there is no single GAC (Global Assembly Cache) for all .NET packages
- package management is traditionally done using *nuget* and an IDE front-end (this might make some people in the audience flinch)
- *paket* is a very promising replacement setting out to fix the common problem of *DLL* hell
- *paket* resolves the dependency graph at the solution level and manages references of projects

But whats the point of using Nix(OS) then?

```
Tentative Answer:

because it brings a lot more value to the table than just p
```

To use *nix* for package management we'd need to:

1. create and maintain packages for nuget packages, possibly automating the process with the right tooling

To use *nix* for package management we'd need to:

1. create and maintain packages for nuget packages, possibly automating the process with the right tooling
2. have a way to generate reference entries in .fsproj files automatically, just as *paket* does it

To use *nix* for package management we'd need to:

1. create and maintain packages for nuget packages, possibly automating the process with the right tooling
2. have a way to generate reference entries in .fsproj files automatically, just as *paket* does it
3. build projects such that runtime deps get linked correctly

- some work towards that end has already been done by @obadz, albeit it seems experimental at this point

- some work towards that end has already been done by @obadz, albeit it seems experimental at this point
- there might not be big enough incentives to do this at this point

A Sample Project

Project

Proposition

Assume we have a lots of great computer science papers on our SSD, and we'd like to be able to index and query for information (e.g. [1]) via *curl*.
So, lets build a small microservice around the *recoll* full-text indexer and serve query results via HTTP.
[1] https://github.com/ocharles/papers

A quick word about...

Microservices!

Microservies?

TODO: remind myself WTF are they again?

No, but seriously, its a stupid buzzword and we all know it but apparently we like pressing those buttons over and over again.

A microservice is a small, stand-alone component most often part of a system of more of these stand-alone, de-coupled units. Essentially, its a scalability design pattern for web applications.

From the point of view of a functional programming enthusiast, microservices are a good example of how small, stateless (*pure!*) building blocks (*referential transparency*) can be *composed* into systems that are easier to understand and maintain and more robust that big monolithic code-bases.

I am not a dev-ops person, but I hear that larger systems become fiendishly hard to deploy and monitor, though.
Blame successfully shifted!
:)

Bootstrapping

The state of affairs of project management in mono/F# is still for the most part centered around using IDE's for everything.

There are the obvious Candidates, such as MS' Visual Studio or MonoDevelop, which is at least free and available via *nix*.

To alleviate that situation for those who don't like IDEs, there is a project scaffold git repository with an initialization routine to help set up everything.

We'll focus on that workflow for this talk, since this be beneficial to automate tasks down the line (think CI and deployment).

Using Project Scaffold

```
→ git clone git@github.com:fsprojects/ProjectScaffold.git P
→ cd PaperScraper && ./build.sh
```

Answer a couple of questions and you're set. But wait!

Figure 1:

```
error : Target named 'Rebuild' not found in the
project.
```

A Note About Build Targets

Since in *NixOS* there is no(t one, but many) Global Assembly Cache, resolving default build targets shipped with *F#* does not work as expected.

The solution is to patch all *.fsproj* files in the solution and only conditionally set the *FSharpTargetsPath* if the target actually exists:

```
- <FSharpTargetsPath>$(MSBuildExtensionsPath32)\Microsoft\V
+ <FSharpTargetsPath Condition="Exists('$(MSBuildExtensions
```

There is a package *nix* written by *@obadz_ which contains a script that finds and patches up* .fsproj_ files to look out for the *FSharpTargetsPath* enviroment variable.

```
→ nix-env -i dotnetbuildhelpers
```

Usage:

```
→ patch-fsharp-targets.sh
  Patching F# targets in fsproj files...
  ./src/PaperScraper/PaperScraper.fsproj
  ./tests/PaperScraper.Tests/PaperScraper.Tests.fsproj
```

Last, we only need to set *FSharpTargetsPath* in our shell:

```
→ export FSharpTargetsPath=$(dirname $(which fsharpc))/../li

→ set -x FSharpTargetsPath (dirname (which fsharpc))/../lib/m

{ config, pkgs, ... }:
{
  environment.variables.FSharpTargetsPath = "${pkgs.fsharp}/
}
```

Hm, More Errors

Unfortuantely, more problems crop up at this point.

- FSharp.Core dll is missing when running tests, hence the build fails
- FSharp.Core dll is missing when generating documentation, hence the build fails

:(

Ah yes!

> *"In compiled applications, you should never assume that FSharp.Core is in the GAC ("Global Assembly Cache"). Instead, you should deploy the appropriate FSharp.Core as part of your application."*

https://fsharp.github.io/2015/04/18/fsharp-core-notes.html

Quick Fix:

Ship *FSharp.Core* as part of the build output and manage the dependency with *paket*.

```
in paket.dependencies, add;

FSharp.Core = 4.0.0.1

in src/PaperScraper/paket.references, add:

FSharp.Core
```

```
<Reference Include="FSharp.Core">
  <Private>True</Private>
 <HintPath>..\..\packages\FSharp.Core\lib\net40\FSharp.Cor
</Reference>
```

A Better™ Fix:

It would be better to use the F# version shipped with NixOS.

```
 <TargetFSharpCoreVersion>4.3.1.0</TargetFSharpCoreVersion

  <HintPath>$(TargetFSharpCorePath)</HintPath>
```

```
→ set -x TargetFSharpCorePath (dirname (which fsharpc))/../li
```

```
{ config, pkgs, ... }:
{
 environment.variables.TargetFSharpCorePath = "${pkgs.fshar
}
```

Finding the correct F# Version

	F# 3.0	F# 3.1		F# 4.0	F# X.Y
Target Framework	VS 2012	VS 2013	3.1.1, 3.1.2 updates	VS 2015	VS vFuture
.NET 2	2.3.0.0	2.3.0.0 [frozen]		2.3.0.0 [frozen]	2.3.0.0 [frozen]
.NET 4	4.3.0.0	4.3.1.0		4.4.0.0	4.X.Y.0
Portable 47	2.3.5.0	2.3.5.1		3.47.4.0	3.47.X.Y
Portable 7		3.3.1.0		3.7.4.0	3.7.X.Y
Portable 78			3.78.3.1	3.78.4.0	3.78.X.Y
Portable 259			3.259.3.1	3.259.4.0	3.259.X.Y
Portable N					3.N.X.Y

Figure 2:

http://stackoverflow.com/questions/20332046/correct-version-of-fsharp-core

Documentation and Help Targets

They do not build at this point, for the same reasons.
TODO: improve this slide

But we're building an executable, right?!

The project template at this point generates a library project by default, so the *.fsproj* file needs to be amended in 2 ways:

```
<OutputType>Exe</OutputType>

<ErrorReport>prompt</ErrorReport>
<Externalconsole>true</Externalconsole>
```

Additionally, the tests project also holds a reference to the current project, so we need to comment it and the code in *Tests.fs* out to ensure a clean build.

Phew!

Recoll

- full-text search tool

Recoll

- full-text search tool
- uses xapian underneath (like other great tools, e.g. `notmuch` and `mu`)

Recoll

- full-text search tool
- uses xapian underneath (like other great tools, e.g. `notmuch` and `mu`)
- extensible and configurable

Recoll

- full-text search tool
- uses xapian underneath (like other great tools, e.g. `notmuch` and `mu`)
- extensible and configurable
- supports indexing many mime types, including extracting text from PDFs

Recoll

- full-text search tool
- uses xapian underneath (like other great tools, e.g. `notmuch` and `mu`)
- extensible and configurable
- supports indexing many mime types, including extracting text from PDFs
- has a flexible query language (based on Xesam)

Recoll

- full-text search tool
- uses xapian underneath (like other great tools, e.g. `notmuch` and `mu`)
- extensible and configurable
- supports indexing many mime types, including extracting text from PDFs
- has a flexible query language (based on Xesam)
- needs to be configured correctly (whitelist directories)

Recoll

- full-text search tool
- uses xapian underneath (like other great tools, e.g. `notmuch` and `mu`)
- extensible and configurable
- supports indexing many mime types, including extracting text from PDFs
- has a flexible query language (based on Xesam)
- needs to be configured correctly (whitelist directories)
- the indexer needs to be run in a cron/systemd timer job

An Example

```
   recoll -t -n 1 -m -q burrito
Recoll query: ((burrito:(wqf=11) OR burritos))
5 results (printing  1 max):
application/pdf [file:///home/k/doc/books/burrito_monads.pd
abstract =  Burritos for the Hungry Mathematician Ed Morehous
author = LaTeX with hyperref package
dbytes = 10425
fbytes = 74745
filename = burrito_monads.pdf
fmtime = 01445444345
mtime = 01445444345
mtype = application/pdf
origcharset = UTF-8
pcbytes = 74745
rcludi = /home/k/doc/books/burrito_monads.pdf|
relevancyrating = 100%
sig = 747451445444345
title =
```

Search For File Names

```
   recoll -t -n 1 -f burrito
Recoll query: (XSFSburrito_monads.pdf)
1 results
application/pdf [file:///home/k/doc/books/burrito_monads.pd
```

Suave.IO

> *Suave is a simple web development F# library providing a lightweight web server and a set of combinators to manipulate route flow and task composition.*

http://suave.io

Features:

- combinators for request routing

Features:

- combinators for request routing
- built-in web-server

Features:

- combinators for request routing
- built-in web-server
- openssl support

The Simplest Possible Application:

```
startWebServer defaultConfig (OK "Hi.")
```

More Elaborate Example

```
open Suave
open Suave.Http
open Suave.Http.Applicatives
open Suave.Http.Successful
open Suave.Web

let serarch q =
 defaultArg (Option.ofChoice(q ^^ "filename")) "nothing" |>

let app : WebPart =
    path "/search" >>=
      GET  >>= request(fun r -> OK <| search r.query)
      RequestErrors.NOT_FOUND "Found no handlers" ]

startWebServer defaultConfig app
```

Types

```
type SuaveTask<'a> = Async<'a option>
type WebPart = HttpContext -> SuaveTask<HttpContext>
// hence: WebPart = HttpContext -> Async<HttpContext option>
```

Pulling It All Together

We need to:

1. create data types (domain model)

Types

```
type SuaveTask<'a> = Async<'a option>
type WebPart = HttpContext -> SuaveTask<HttpContext>
// hence: WebPart = HttpContext -> Async<HttpContext option>
```

Pulling It All Together

We need to:

1. create data types (domain model)
2. write a simple parser for *recoll* output

Types

```
type SuaveTask<'a> = Async<'a option>
type WebPart = HttpContext -> SuaveTask<HttpContext>
// hence: WebPart = HttpContext -> Async<HttpContext option>
```

Pulling It All Together

We need to:

1. create data types (domain model)
2. write a simple parser for *recoll* output
3. map HTTP query to command-line arguments

Types

```
type SuaveTask<'a> = Async<'a option>
type WebPart = HttpContext -> SuaveTask<HttpContext>
// hence: WebPart = HttpContext -> Async<HttpContext option>
```

Pulling It All Together

We need to:

1. create data types (domain model)
2. write a simple parser for *recoll* output
3. map HTTP query to command-line arguments
4. serialiation of results to JSON

Dependencies

Add Suave and FParsec to *paket.depenecies* and *src/PaperScraper/paket.references*:

```
source https://nuget.org/api/v2

nuget FSharp.Formatting
nuget NUnit
nuget NUnit.Runners
nuget FAKE
nuget SourceLink.Fake

nuget Suave
nuget FParsec

github fsharp/FAKE modules/Octokit/Octokit.fsx
```

Contents of *paket.references*:

```
Suave
FParsec
```

A little Tooling

```
#r @"../../packages/Suave/lib/net40/Suave.dll"
#r @"../../packages/FParsec/lib/net40-client/FParsecCS.dll"
#r @"../../packages/FParsec/lib/net40-client/FParsec.dll"

#load @"Types.fs"
#load @"Recoll.fs"
```

```
cd src/PaperScraper
fsharpi --load:script.fsx
```

Basic Types

TODO: WORK IT OUT!!!

```
type MimeType = string
type FileName = string

type CharSet =
  | UTF8
  | Other // how ignorant of me ;)

type Percentage = int

type Url = string

type Bytes = int

type SearchResult =
  { Abstract  : string
  ; FileName  : FileName
  ; MimeType  : MimeType
```

Parsing Recoll Output

TODO: insert parser code

Querying Recoll

TODO: insert query code

Deployment

Writing a derivation

```
→ export FSharpTargetsPath="${fsharp}/lib/mono/4.5/Microsof
```

Systemd services

- need a service for the api server

Deployment

Writing a derivation

```
→ export FSharpTargetsPath="${fsharp}/lib/mono/4.5/Microsof
```

Systemd services

- need a service for the api server
- need a timer and service for recollindexer

- sending a closure

Docker container????

Trying it out

- sending a closure
- container?

Docker container????

Trying it out

Useful resources

- fsharpforfunandprofit blog thingy

Useful resources

- fsharpforfunandprofit blog thingy
- FSharp WikiBook

Useful resources

- fsharpforfunandprofit blog thingy
- FSharp WikiBook
- M$ language reference