Developing F# on NixOS

Karsten Gebbert

27/10/2015

Hi.

Introductions

Who? Why?

- I am not a Nix(OS) or F# expert or .NET veteran

Who? Why?

- I am not a Nix(OS) or F# expert or .NET veteran
- this talk is aimed at people with experience level roughly to my own

Who? Why?

- I am not a Nix(OS) or F# expert or .NET veteran
- this talk is aimed at people with experience level roughly to my own
- sharing my personal experience and impressions

Who? Why?

- I am not a Nix(OS) or F# expert or .NET veteran
- this talk is aimed at people with experience level roughly to my own
- sharing my personal experience and impressions
- it is my first talk, so please 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

- Monads, or *Computation Expressions*

Other Cool Things To Look At

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

Other Cool Things To Look At

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

Other Cool Things To Look At

- 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.

It is also, from the point of view of a functional programming enthusiast a good example of how small, stateless (pure!) building blocks (functions) 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. Shift the blame... :)

Bootstrapping

The state of affairs of project management in mono/F# is still largely centered around using IDE's for everything, which is a bit annoying.

I am not a dev-ops person, but I hear that larger systems become fiendishly hard to deploy and monitor, though. Shift the blame... :)

Bootstrapping

The state of affairs of project management in mono/F# is still largely centered around using IDE's for everything, which is a bit annoying.
To alleviate that situation there is a project scaffold git repository with an initialization script to help set up everything.

Project Scaffold

> *→ git clone git@github.com:fsprojects/ProjectScaffold.git PaperTrail → cd PaperTrail → rm -rf .git → git init → ./build.sh*

Answer a bunch of questions and you're set. Wait! But whats this!?!

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

Figure 1: splonk

Build Targets

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

in scripts/.bashrc etc:

→ export FSharpTargetsPath=$(dirname $(which fsharpc))/../lib/mono/4.5/Microsoft.FSharp.Targets or
→ set -x FSharpTargetsPath (dirname (which fsharpc))/../lib/mono/4.5/Microsoft.FSharp.Targets

Add an entry for it to `paket.dependencies`

But we're building an executable, right?!

Suave.IO

TODO: give a quick overview...
Add an entry for it to `paket.dependencies`

Recoll

- full-text search tool

Suave.IO

TODO: give a quick overview...
Add an entry for it to `paket.dependencies`

Recoll

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

Suave.IO

TODO: give a quick overview...
Add an entry for it to `paket.dependencies`

Recoll

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

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