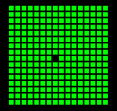
# Puritas

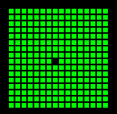
## A journey of a thousand miles towards side-effect free code

# Overview

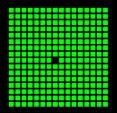
- About me and F# Open Source projects
- What is purity and how does it perform
- Background
- Proposed solution
- Why is purity relevant for you
- Summary (+ demo, if time)
- Q & A

  **Note**: I would love questions, but please save them to the end of the talk, lot to say and time is mana, I mean limited



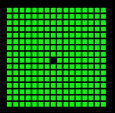
# About me (very shortly)

- Ramón Soto Mathiesen
- MSc. Computer Science ***DIKU/Pisa*** and minors in Mathematics ***HCØ***
- ***CompSci*** @ SPISE MISU ApS
  - ***"If I have seen further it is by standing on the shoulders of giants"***
    ***-- Isaac Newton*** (Yeah Science, ... Mostly mathematics)
  - ***Elm*** (***JS*** due to ports) with a bit of ***Haskell*** and a bit of ***F#*** (fast prototyping)
- Elm / Haskell / TypeScript / F# / OCaml / Lisp / C++ / C# / JavaScript
- Founder of Meetup for F#unctional Copenhageners (MF#K)
- Blog: http://blog.stermon.com/ and Twitter: @genTauro42



# F# Open Source projects

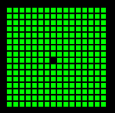
- Previous workplace (CTO of CRM @ Delegate A/S):
  - MS CRM Tools:
    - http://delegateas.github.io/
  - ~~Delegate.Sandbox~~:
    - http://delegateas.github.io/Delegate.Sandbox/
- Current workplace (SPISE MISU ApS):
  - Syntactic Versioning (SynVer @ F# Community Projects)
    - Mostly driven by Oskar Gewalli (@ozzymcduff)
  - Puritas, isolated side-effects at compile-time in F#



# What is purity

- There was an interesting blog post with regard of this topic that surfaced after my talks was made public by #fsharpX:
  - F# and Purity from Eirik Tsarpalis' blog
- It was a bit unfortunate the definition of purity that was taken from WP at the top of that post:
  - "*... **Purely functional programing** may also be defined by **forbidding changing state** and mutable data.*"
- If we can't change state, why even run it?

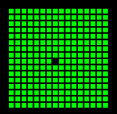
  let main ***state*** = ***state*** (* If we can't change state? I guess we are done *)



# What is purity

- I'm guessing that we should be talking about ***pure functions*** WP instead:
  - The function ***always evaluates*** to the ***same result*** value ***given*** the ***same arguments***
  - Evaluation does ***not cause*** any observable ***side effect*** or output, such as mutation of ***mutable objects or*** output to ***I/O*** devices
- OK, so we change ***state*** and we are still ***pure***:

  let rec main ***state*** = function | 0u → ***state*** | n → main (***state*** + 1) (n - 1u)
- So in ***purely functional programing***, state changes, but in a ***sound*** way



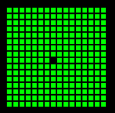
# What is purity

- Be careful to not become to ***pedantic***, still from ***pure functions*** WP:
  - The result value need not depend on all (or any) of the argument values. However, ***it must depend on nothing other than the argument values***
- So this is not pure?

  let ***foo*** () = 42

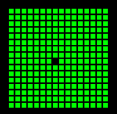
  let ***bar*** x = ***foo*** () + x (* besides x, the result depends on foo *)
- What about curried arguments?

  let ***baz*** x y = x + y
  let ***qux*** = fun x → fun y → x + y (* nested lambda depends on parent *)



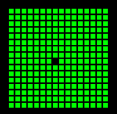
# What is purity

- The previous ***pure*** functions (***foo***, ***bar***, ***baz***, ***qux***) can be mapped directly to ***λ-calculus***, which is mathematically ***pure***.
- Therefore, the result of ***combining pure functions***, would still be considered ***pure***
  - Save this "***bit of information***" for later



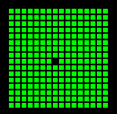
# What is purity

- Lets recap:
  - Functions ***always evaluates*** to the ***same output*** value ***given*** the ***same input***
  - Evaluation does ***not cause*** any ***side effect***, such as mutation of ***mutable objects or*** output to ***I/O*** devices
  - Functions can be mapped directly to ***λ-calculus***, which is mathematically ***pure***.
  - The result of ***combining pure functions***, would still be considered ***pure***



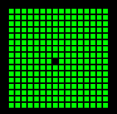
# and how does it perform

- Taken from SO (Academia):
  - Pippenger [1996], *"**Pure Versus Impure Lisp**"*, comparing pure Lisp (strict evaluation, not lazy) to one that can mutate data, establishes that is the best you can do is $\Omega(n \log n)$ in the pure when problems are $O(n)$ in the impure version
  - Bird, Jones and De Moor [1997], *"**More haste, less speed: Lazy versus eager evaluation**"*, demonstrate that the problem constructed by ***Pippenger*** can be solved in a lazy functional language in $O(n)$



# and how does it perform

- Taken from SO: (In Practice)
  - Okasaki [1996] and Okasaki [1998], "***Purely Functional Data Structures***", many algorithms can be implemented in a pure functional language with the same efficiency as in a language with mutable data structures.
    - My blog: F# - Puresort of lists (Okasaki)

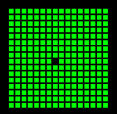


# and how does it perform

- Taken from SO: (In Practice)
  - SPJ and Marlow [1999], "***Stretching the storage manager: weak pointers and stable names in Haskell***", due to referential transparency, even when using memo and unsafe IO, will not change pure behavior

**memo :: (a → b) → (a → b)** and **unsafePerformIO :: IO a → a**

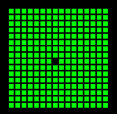
| | |
|---|---|
| fib :: Int → Int<br>fib = memo ufib | ufib :: Int → Int<br>ufib 0 = 1<br>ufib 1 = 1<br>ufib n = fib (n – 1) + fib (n – 2) |



# and how does it perform

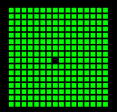
- Taken from SO: (In Practice)
  - **Remark**: To ensure that impurity can be hidden under referential transparency, the following must be added on top of all your files so that side-effects ***must*** be handled through ***Monads*** to avoid "Launching the missiles":

    ```
    {-# LANGUAGE Safe #-}
    ```



# Background

- A few years ago I created Delegate.Sandbox in order to provide side-effect free code in F#
- I mainly did it to troll Haskell people. MF#K is a cross-functional Meetup Group and ***haskellers*** can be a bit annoying with their ***purity*** sometimes ...
- On a serious note, the reason is that most developers don't really know which I/O side-effects are executed in their applications
- The library is built on top of the AppDomain Class which allows to Run Partially Trusted Code in a Sandbox (.NET)
- Talk at MF#K (2015-09-29): I/O side-effects safe computations in F#

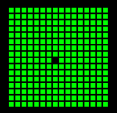


# Background

- Delegate.Sandbox Pros:
  - Guaranteed side-effect free code
  - Idiomatic syntax:

```
let hashUsrPwd usr pwd salt =
  sandbox { return CompanyA.Fancy.Library.hash user pwd salt}

hashUsrPwd "john.doe@companyB.com" "pass@word1" "peterpandam" |> function
  | Unsafe e -> raise e // Hmmmm, somebody is performing side-effects
  | IOSafe hash -> () (* Saving to DB goes here *)
```

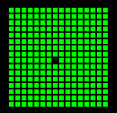
```
public void ExecuteUntrustedCode(string assemblyName, string typeName, string entryPoint, Object[] parameters)
{
    //Load the MethodInfo for a method in the new assembly. This might be a method you know, or
    //you can use Assembly.EntryPoint to get to the entry point in an executable.
    MethodInfo target = Assembly.Load(assemblyName).GetType(typeName).GetMethod(entryPoint);
    try
    {
        // Invoke the method.
        target.Invoke(null, parameters);
    }
    catch (Exception ex)
    {
        //When information is obtained from a SecurityException extra information is provided if it is
        //accessed in full-trust.
        (new PermissionSet(PermissionState.Unrestricted)).Assert();
        Console.WriteLine("SecurityException caught:\n{0}", ex.ToString());
        CodeAccessPermission.RevertAssert();
        Console.ReadLine();
    }
}
```



# Background

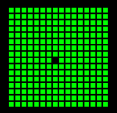
- Delegate.Sandbox Cons:
  - Tainted expressions (Unsafe) cause run-time errors
  - Not thread-safe (race conditions)
  - ~~Post-Build F# script (need code to be compiled first)~~
  - Reason, the F# Compiler Services (FCS) only supported untyped syntax trees back then

```
let private sandboxDomain,sandboxType =
   match AppDomain.CurrentDomain.GetData("domain"),
         AppDomain.CurrentDomain.GetData("typeof") with
   | null,_ | _,null ->
     ...
     // Most likely not theadsafe but it's always the same value so ...
     do sandboxDomain'.SetData("domain", sandboxDomain' :> obj)
     do sandboxDomain'.SetData("typeof", sandboxType' :> obj)
     ...
```



# Background



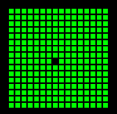


# Proposed solution

- Thanks to Microsoft and the F# Community, FCS now also supports typed expression trees (*)
- So lets recall: "*Therefore, the **result of combining pure functions**, would still be considered **pure***"
- Now that we can ***type-check*** our code with FCS, we should be able to ***reason*** about if code is pure (or not)

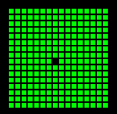
(*) - Almost, at least Sum Types aren't supported (yet?)

```
type FooBar = Foo of int | Bar of float (* not working *)
```



# Proposed solution

- There is actually a POC in the F# Compiler to check if an expression has effects (flag: ***--test:HasEffect***)
- And recently I found out, from a tweet, that there was another project trying to separate ***pure*** from ***impure code***:
  - PolyglotSymposium.Sandline
- Both experiments are based on typed expression trees as well as my project, ***SpiseMisu.Puritas***, but what makes my project different from theirs is that I mark ***pure branches*** with a ***type*** while they rely on marking idiomatic code as pure or not (true/false)

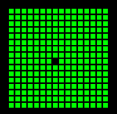


# Proposed solution

- The reason I'm adding a type is because F# is an eager (strict) impure functional language and that way I can distinguish branches at compile time (***F# is what it is and we can't/shouldn't change that***)
- Therefore, my approach is to add ***ad-hoc pure branches*** to our impure code

  let foo : int **Pure** = **purify** 42
- Just think of it as with the **lazy** keyword, where we are able to add ***ad-hoc lazy branches*** to our strict code

  let bar : int **Lazy** = **lazy** 42



# Proposed solution

- So now, we just need to find all our code branches that return pure code. This is actually very easy to do as F# (.NET) has a canonical type signatures:

  let foo : **int list** = [ 42 ]

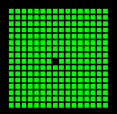
  let bar : **List<System.Int32>** = [ 42 ]

- Therefore we can just look for all signatures that comply with:

  **((...) ... SpiseMisu.Puritas.Pure)** (ends with)

  **SpiseMisu.Puritas.Pure<... <...>>** (starts with)

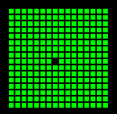
- Even though we can type-check, this will not be enough ...



# Proposed solution

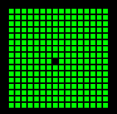
- … as we will have to ***taint code expressions*** that don't comply with the following recursive parent/child code branch logic:

| Code branches | **Parent Impure** | **Parent Pure** | **Parent Tainted** |
|---|---|---|---|
| **Child Impure** | ***Impure*** | Tainted | Tainted |
| **Child Pure** | ***Impure*** | ***Pure*** | Tainted |
| **Child Tainted** | Tainted | Tainted | Tainted |



# Proposed solution

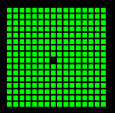
- Therefore our taint-checker marks the following branches as ***valid***:
  - Impure → Impure (regular F# code is perfectly valid)
  - Impure → Pure (pure code consumed by impure is also OK)
  - Pure → Pure (used when defining pure libs and/or APIs)
- All the other cases will be marked as ***invalid (tainted)***
- ***Invalid code*** will ***bubble up*** to the top, ***tainting the hole expression*** as invalid. Just think of taint like ***poison*** in Tony Hoare ***Communicating Sequential Processes*** (CSP)



# Proposed solution

- Like I mentioned before, my project differs from the others in that I'm able to mark the following code as valid, while they would mark it as invalid (true/false):

```
| BasicPatterns.NewArray (_,exprs) ->
  (* FSharpType * FSharpExpr list *)

  let msg  = "BasicPatterns.NewArray"

  let tag' =
    (* Reason: Arrays are mutable, therefore impure *)
    tag
    |> taint msg range Tag.Impure
    |> taint msg range (taintExprs debug (tag) exprs)

  debug mexpr msg tag' tag

  tag'
```



# Proposed solution

- We only consider pure code signatures that comply with:

  **((...) ... SpiseMisu.Puritas.Pure)** (ends with)

  **SpiseMisu.Puritas.Pure<... <...>>** (starts with)

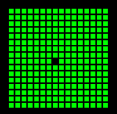
- That means that F# Core is impure as well

  **purify** (1 + 2) (* is actually impure, so “Computer Says No” *)

- So how do you code ***without basic arithmetic operators***?

- Well F# to the rescue. We just expand our ***pure type*** with some operator overloading and we are good to go:

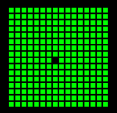
  **purify** 1 + **purify** 2



# Proposed solution

- So what are we looking at?

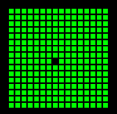
```
#r @"SpiseMisu.Puritas.dll"
open SpiseMisu.Puritas

let sum : int Pure -> int Pure -> int Pure = fun x y -> x + y
let result = sum (purify 42) (purify 42)

let inc : (int Pure -> int Pure) Pure =
  purify (fun x -> x + purify 1)
let dec : (int Pure -> int Pure) Pure =
  purify (fun x -> x - purify 1)
let add : (int Pure -> int Pure -> int Pure) Pure =
  purify (fun x y -> x + y)

let foo = dec <*> purify 42
let bar = inc >*> dec <*> purify 42
let baz = purify 42 |*> (inc <*< dec)
let qux = purify 42 </ add /> purify 42

let rec fold f acc = function
  | Nil         -> acc
  | Cons(x,xs) -> fold f (cons (f x) xs) xs
let map f xs = fold f nil xs

let foobar = cons (purify 42) nil |> map (fun x -> x + x)
```



# Proposed solution

- It's pretty idiomatic right?
- From/to (***purify***/***value***) and list support (***cons***/***nil*** and |***Cons***|***Nil***|)
- In order to wrap/unwrap pure functions/values, I added a few extra operators (***apply*** <*>, ***left/right composition*** >*> and <*<, ***pipe*** |*>, ...)
- I also added a few functions (***memo***, ***concurrent***, ***delay***) with referential transparency to achieve better performance
- Since F# Core is impure, we will need boolean arithmetic operators as we can't overload them:

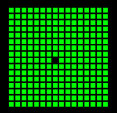
  == (***EQ***), /= (***NEQ***) >- (***G***), -< (***L***), => (***GE***), =< (***LE***)



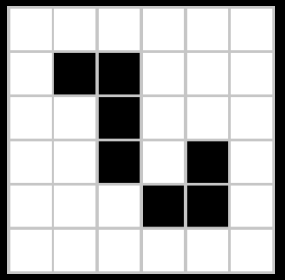
# Proposed solution

- Fibonacci (+ memo version):

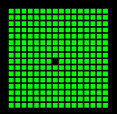
```
#r @"SpiseMisu.Puritas.dll"
open SpiseMisu.Puritas

let zero = purify 0
let one  = purify 1
let two  = purify 2

let rec fib : int Pure -> int Pure =
  fun n ->
    if   zero == n then one
    elif one  == n then one
    else
      (n-one |> fib) + (n-two |> fib)
(* Real: 00:00:08.959, CPU: 00:00:09.132, GC gen0: 2312, gen1: 0 *)
Array.init 36 (purify >> fib >> value)

let rec ufib : int Pure -> int Pure =
  fun n ->
    if   zero == n then one
    elif one  == n then one
    else
      (n-one |> fibMemo) + (n-two |> fibMemo)
and fibMemo : int Pure -> int Pure = memo ufib
(* Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0 *)
Array.init 36 (purify >> fibMemo >> value)
```



# Proposed solution

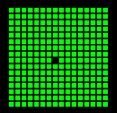


- Lets recap (***SpiseMisu.Puritas***):
  - A library:
    - SpiseMisu.Puritas.dll (~100 lines of code)
    - Provides ***ad-hoc pure branches*** to our ***impure code*** (think of it like with ***lazy***)
  - A taint-checker:
    - SpiseMisu.Puritas.TaintChecker.fsx (~1000 lines of code)
    - ***Only depending*** on ***F# Core*** and **FCS** (HAL 9000, I mean ***@ncave***, Fable much?)
    - ***Tainting*** expressions at ***compile-time*** and errors are ***prettified*** with ***Markdown*** syntax
  - ***Idiomatic***, except for boolean arithmetic operators
  - ***Acceptable performance*** due to referential transparency (memo, ...)



# Why is purity relevant for you

- Purity it's not just ***academic mumbo jumbo***
- ***Privacy-by-design***, get used to it as ***General Data Protection Regulation*** (GDPR) arrives next year:
  - Doom-day: **2018-05-28**
- Easiest way to comply with this approach is by ***isolating your side-effect***. Languages supporting this at the moment are: ***Haskell***, ***COQ***, ***Idris***, ***PureScript***, ***Elm*** among others and hopefully soon F#, due to ***SpiseMisu.Puritas***
- I know, the people from the UK are just thinking: "Why should we care?", well:
  - The future of UK data protection law post-Brexit
    - "The GDPR will come into effect before the UK leaves the European Union"
    - "The UK will still have GDPR-like rules after it leaves the European Union"



# Why is purity relevant for you

version2 IT-NYHEDER BLOGS IT-JOB SEKTIONER MERE

## Mega-svipser: CPR-numre og skatteoplysninger frit tilgængelige på Skats hjemmeside

**Skatteoplysninger og CPR-numre har været frit tilgængelige via Skats hjemmeside som følge af en fejl.**

Jakob Møllerhøj Fredag, 10. marts 2017 - 12:44 33

En fejl har bevirket, at danske CPR-numre med tilhørende skatteoplysninger i går aftes var tilgængelige for uvedkommende på Skats hjemmeside.

»I går erfarede vi, at vores leverandør har lavet en alvorlig fejl. Og den fejl betød desværre, at enkelte borgeres oplysninger var synlige for andre. Og de oplysninger var blandt andet deres CPR-numre og deres årsopgørelser. Det siger sig selv, at det er fuldstændigt uacceptabelt for Skat, hvis den enkelte borgers oplysninger har været synlige for andre end dem selv,« siger kontorchef i Skat Jørgen Wissing Jensen.

Kontorchef Jørgen Wissing Jensen, Skat: Hvis nogen har været inde og lave ændringer i de oplysninger, der har været vist, så er det et forhold, vi tager meget alvorligt

Skat oplyser, at de blotlagte skatteoplysninger og CPR-numre tilhører »en lille rådgivningsvirksomheds kunder.« Skat ønsker ikke at oplyse, hvilken rådgivningsvirksomhed, der er tale om.

En læser, der ønsker at være anonym, har tippet Version2 om fejlen. Han fortæller, at han loggede ind på Skats selvbetjeningsløsning i går aftes omkring klokken 21.

Efter at have klikket lidt rundt, blev han pludseligt præsenteret for en liste med andres borgeres CPR-numre.

Og via CPR-numrene var det muligt at klikke sig videre ind og se de bagvedliggende skatteoplysninger i form af årsopgørelser.

28. FEB. 2017 KL. 18.54 | OPDATERET 01. MAR. 2017 KL. 08.52

## Private oplysninger er ude efter stort læk hos Novo Nordisk

E-mail-adresser, navne og telefonnumre på ansøgere har ligget frit tilgængeligt på Novo Nordisks hjemmeside.

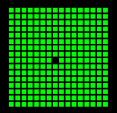

Novo Nordisk har oplevet et datalæk, hvor ikke-sensitive informationer utilsigtet blev lagt på novonordisk.com, skriver selskabet. (Foto: liselotte sabroe © Scanpix)

PRINT

DEL ARTIKLEN:

MAIL

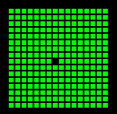
Ved en fejl har en række oplysninger fra op mod 95.000 jobansøgere fra forskellige lande ligget frit tilgængeligt på Novo Nordisks hjemmeside.



# Why is purity relevant for you

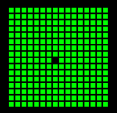
- We recently had two cases where sensitive was leaked through websites (both cases could easily be avoided by using something like Hardy Jones elm-proxy):
  - SKAT (Danish Ministry of Taxation)
    - Some people when login in could choose other peoples profiles, presented in a list, like admin mode
  - Novo Nordisk (Denmark's Top 2 greatest company, turnover/revenue: 107.927 mDKK)
    - 95.000 job applicants data (name, phone, e-mail, ...) was published to their main website (human error)
- What if it was next year, both blamed their software provider? (Sanctions)
  - Fines in the size of 10/20 mEUR or 2%/4% annual worldwide turnover (whichever is greater)

  **Note**: turnover (UK)/revenue (US) reference to the amount of money a company generates without paying attention to expenses or any other liabilities



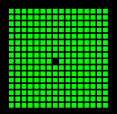
# Why is purity relevant for you





# Why is purity relevant for you

- Are you willing to deliver software from ***Doom-day*** next year?
  - How are you going to ***convince*** your customers that you are doing everything to ensure that no ***unwanted*** side-effects and hereby data-leaks will occur?
- Lets remove the ***blame-game*** and the ***say a lot but do nothing*** from the equation and focus on solving the real problem, ***with science ofc***
- By ***tainting unwanted side-effects*** at ***compile-time***, ***no system*** will be ***deployed to production with vulnerabilities***
- You will just need to request ***pure code*** through signatures files from your contractors or software providers (next slide)



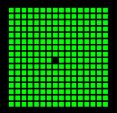
# Why is purity relevant for you

```
namespace EvilCorp
  module BusinessLogic =
    (* Connect to 3rd party and leak data, perfectly fine *)
    val foo : int -> int
    (* Create folder and log sensitive data, fine as well *)
    val bar : float -> float
```

vs

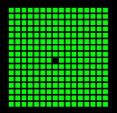
```
namespace CantBeEvilCorpAnymore
  module BusinessLogic =
    (* Connect to 3rd party and leak data, Computer Says No *)
    val foo : int Pure -> int Pure
    (* Create folder and log sensitive data, Computer Says No *)
    val bar : float Pure -> float Pure
```

“Don’t be evil” enforced by code !!!



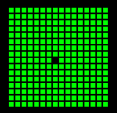
# Why is purity relevant for you

- Just think of it in Simon P. Jones (SPJ) terminology:
  - Isolate side-effects to avoid “Launching the missiles”
  - Isolate side-effects to avoid “Leaking data”
- By enforcing purity, the “Volkswagen emissions scandal” (dieselgate), would never have been possible as the Governments could just require that car manufactures software, complied with their signatures files



# So … Don ~~Vito~~ Syme


# Can I haz ***pure*** keyword so that

# Can I haz ***pure*** keyword so that

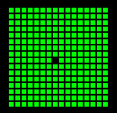

- The following code …

  let foo : int **Pure** = **purify** 42

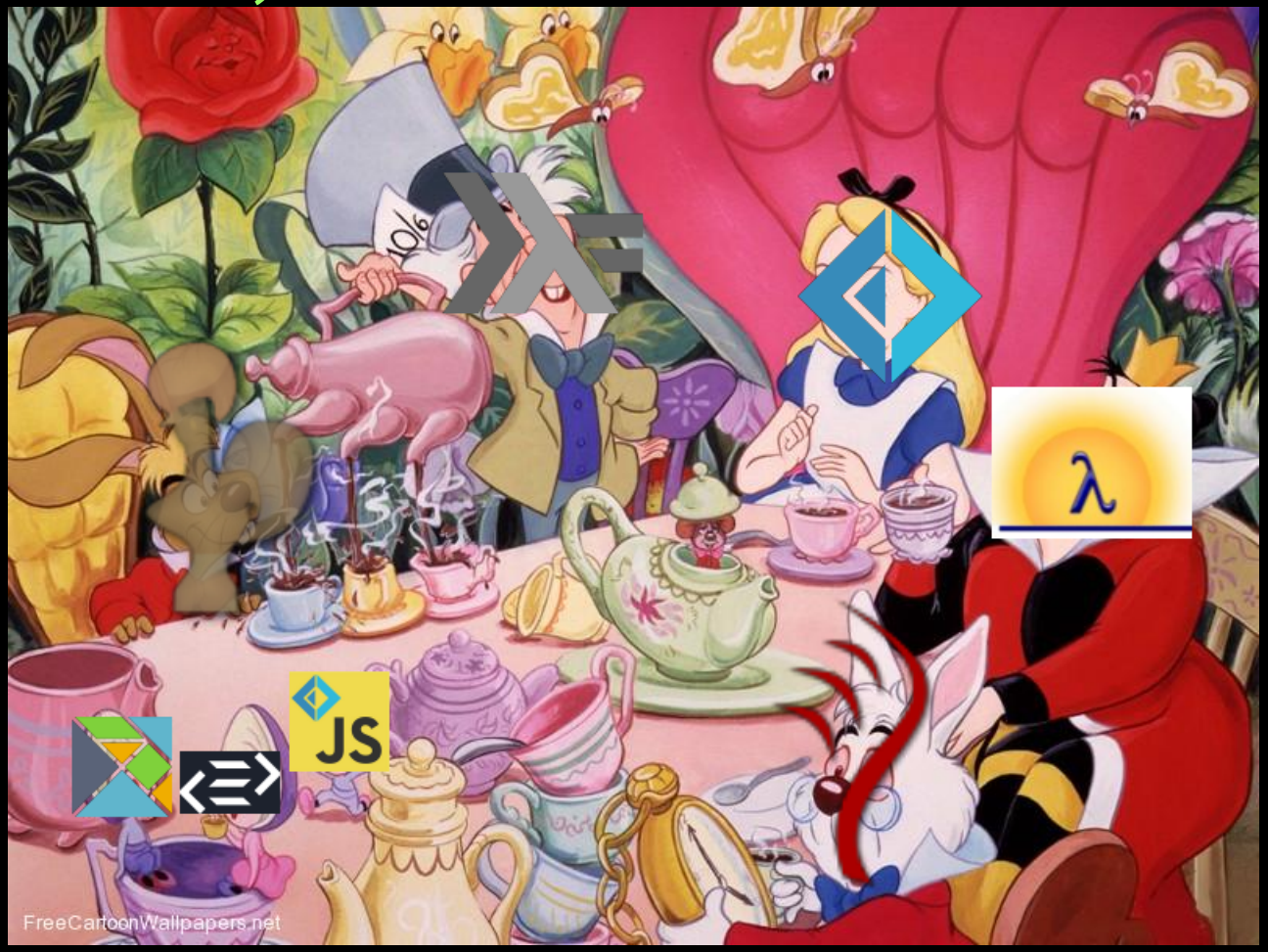
- … becomes

  ~~#nowarn "46"~~

  let foo : int **Pure** = **pure** 42

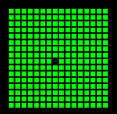


# F# can join the ***Mad Tea Party***

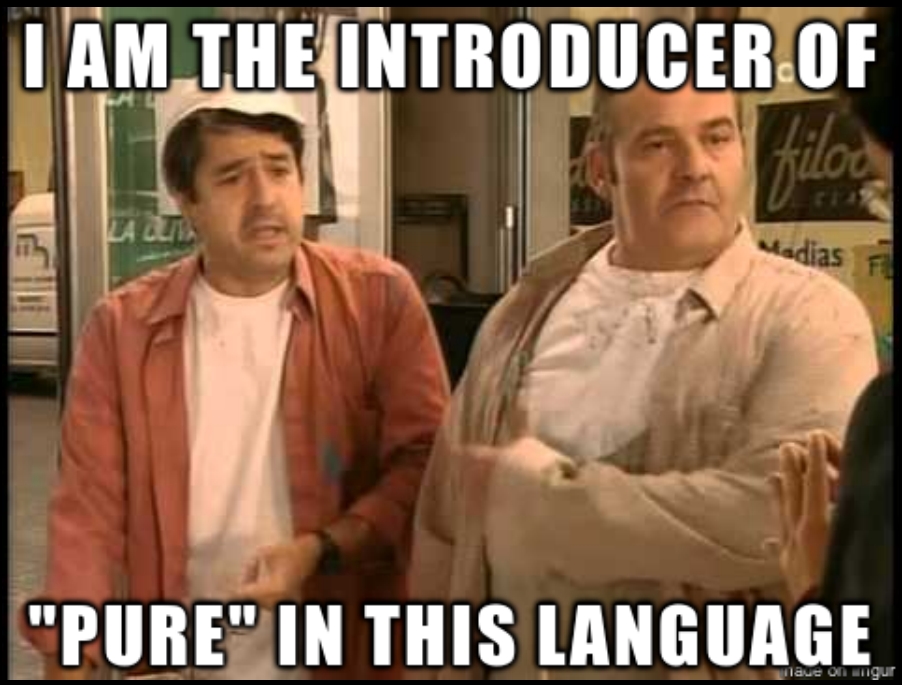



# Summary (+ demo, if time)

- ***SpiseMisu.Puritas*** provides ***ad-hoc side-effect free code*** at ***compile-time***
- ***Privacy-by-design***, General Data Protection Regulation (GDPR): Doom-day: ***2018-05-28***
- Dank memes aside, I will make a ***formal request*** for the reserved keyword ***pure*** at F# Language and Core Library Suggestions
  - I will post link on Twitter, please vote if you agree that it should be part of F# Core
- If @ncave pulls it off, F# could be the first to provide purity at both ***BE*** and ***FE*** !!!
- "**S**tay **P**ure, **I**solating **S**ide-**E**ffects" (***SPISE*** MISU ApS, it was all part of the Masterplan)
  - Michael Werk Ravnsmed dixit
- Finally, I would like to thank Joakim Ahnfelt-Rønne (@Continuational) for his reviews, his initial "***counter***" examples and specially showing that the library was ***pretty much useless*** without the possibility to ***lift impure values into pure functions*** (ex: load an int from a file, increment and save)
- Any time left to "Show some code" and demo?



# Q & A


Only “old” Spaniards will get this