Delegate.Sandbox:
I/O side-effects safe computations in F#

@ Prosa 2015-09-29

F#unctional Copenhageners Meetup Group
(MF#K)

Overview

- About me
- F#unctional Copenhageners Meetup Group (MF#K)
- Delegate.Sandbox:
  - What is it and why it was created
  - How it works and limitations
    - Initial release
    - Upcoming version (will be submitted to GitHub after talk)
    - (Probably) Final version (I/O safe libraries at compile time)
  - Demo: Show me some code
- Q & A
- We eat the cake and go for some beers @ Ørsted Ølbar
  - http://oerstedoelbar.dk/

About me (very shortly)

- Ramón Soto Mathiesen
- MSc. Computer Science DIKU and minors in Mathematics HCØ
- Managing Specialist / CTO of CRM Department @ Delegate A/S
  - ER-modeling, WSDL, OData (REST API)
- F# / C# / JavaScript / C++
- Blog: http://blog.stermon.com/

F#unctional Copenhageners Meetup Group (MF#K)

- F#unctional Copenhageners Meetup Group will try to get more and more software projects to be based on functional programming languages.
- We mainly focus on F# and Haskell, but other functional programming languages like Scala, Lisp, Erlang, Clojure, etc. are more than welcome.
- We expect to meet at least twelve times a year, if not more, to share experiences with regards of the use of functional programming languages in software projects that are in / or heading to production.

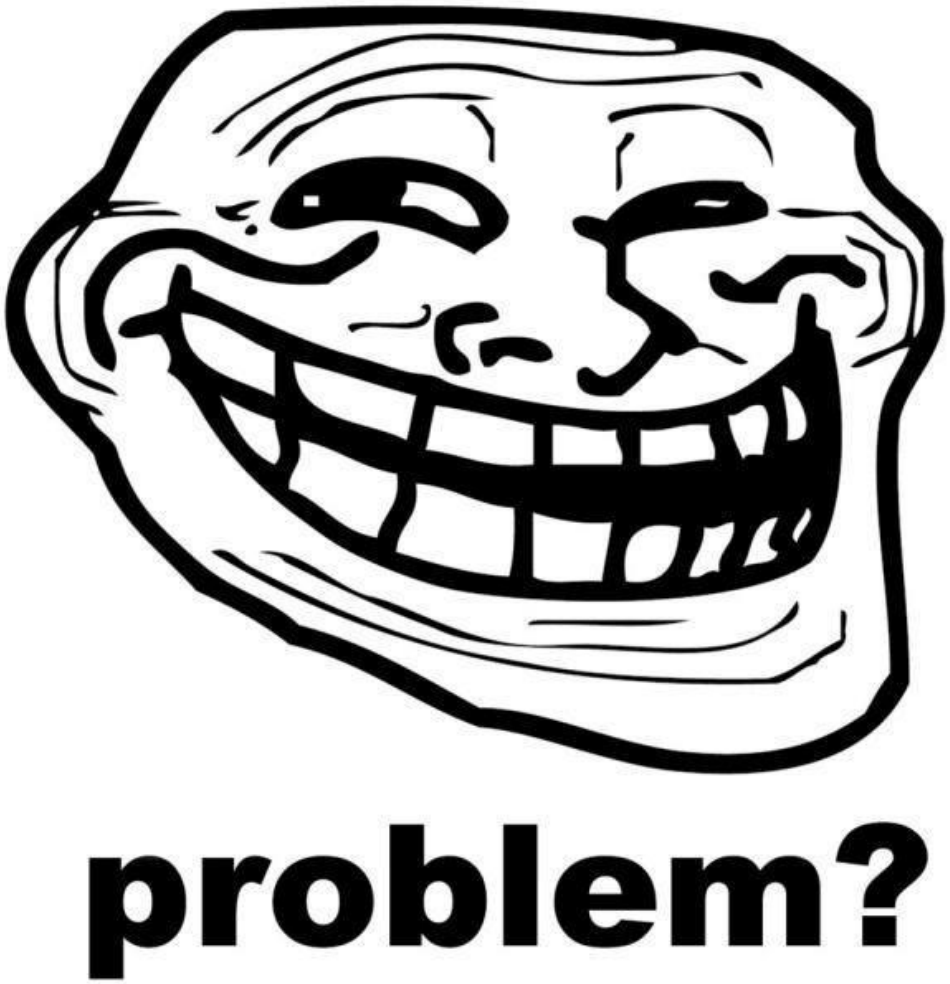
Delegate.Sandbox: What is it …

- It's library that provides a Computation Expression named *SandboxBuilder*, **sandbox{ return 42 }**, which ensures that values returned from the computation are I/O side-effects safe (**IOSafe**) and if not, they are marked as unsafe (**Unsafe**) and returning an exception.

- The library allows to bind (**>>=**) several sandbox computations together in order to create side-effect free code and based on the final result, then proceed to perform the desired side-effects.

Delegate.Sandbox: … and why it was created

- To troll Haskell People: Haskell Is Exceptionally Unsafe



Delegate.Sandbox: … and why it was created

- To troll Haskell People (just kidding) ☺
- Seriously, the reason is that most developers don't really know which I/O side-effects are performed under their application (a simple example, slightly less harmful):
  - You have deployed DEBUG code to a production system where the application uses a lot of resources to write to the console or a log file.

Delegate.Sandbox: … and why it was created (extended example)

- But mostly to be able to ensure correctness for business critical applications (another example, pretty harmful):
  - You are using a proprietary (and non open sourced) 3rd party library that has a hashing algorithm for username/password with a grain of salt.
  - You write your code with the library, and deploy to production. Everyone's happy until the customer tells you that all of their usernames have been compromised.
  - You think to yourself: "How the MF#K?".

Delegate.Sandbox: … and why it was created (extended example)

- But mostly to be able to ensure correctness for business critical applications (another example, continuation):
  - What if you were using Delegate.Sandbox? Lets look into the following code:

```
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 *)
```

  - When you run your tests, you find out that the library actually has some not-expected side-effects. You proceed to use a decompiler (dotPeek) and you find out that there is a piece of code that send people's user names and passwords in "clear text" to cyka@blyat.ru by using System.Net.Mail

Delegate.Sandbox: … and why it was created (extended example)

Delegate.Sandbox: How it works and limitations (Initial release)

- The library is built on top of the AppDomain Class which allows to Run Partially Trusted Code in a Sandbox (.NET).
- The **SandboxBuilder** is only allowed to execute code (**SecurityPermissionFlag.Execution**), which is the minimum permission that can be granted (Principle of least privilege)

Delegate.Sandbox: How it works and limitations (Initial release)

- Example of C# code taken from Run Partially Trusted Code in a Sandbox:

```
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();
     }
}
```

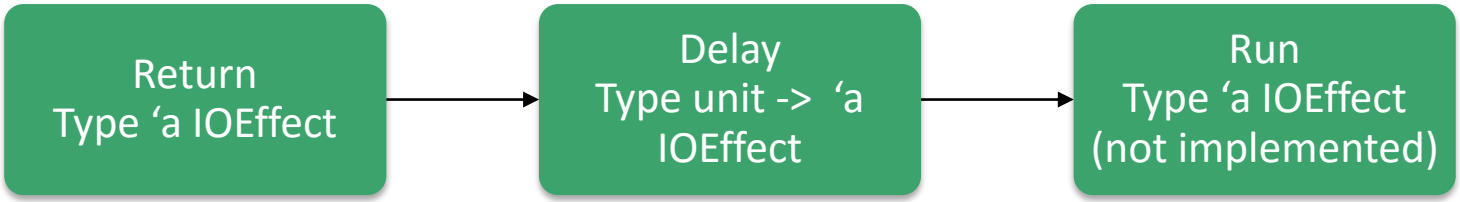
- We can agree that this is not very idiomatic right?

Delegate.Sandbox: How it works and limitations (Initial release)

- **sandbox** is implemented as a computation expression that only implements:
  - A **return** method (**Return : v:'b -> 'b IOEffect**), which ensures that values returning from the computation are of the desired value type.
  - A **delay** method (**Delay : f:(unit -> 'a IOEffect) -> 'a IOEffect**), which tries to evaluate the function at the newly created domain (**AppDomain**) with the minimum granted permission instead of the executing **AppDomain.CurrentDomain**. If the function evaluation is successful then an IOSafe 'a value is returned, otherwise an Unsafe Exception is returned.

Note: "**delay** is similar to the **reify** operation of Filinski [4]" -- Tomas Petricek and Don Syme (The F# Computation Expression Zoo paper)

Delegate.Sandbox: How it works and limitations (Initial release)



- Very simple implementation (5 lines of code):

  Note: SecurityPermissionAttribute → SecurityAction.PermitOnly → Execution = true fixed a major bug (more on this later on)

```
[<Sealed>]
type SandboxBuilder() =
  inherit MarshalByRefObject()
  member x.Return v = IOSafe v
  [<SecurityPermissionAttribute(SecurityAction.PermitOnly, Execution = true)>]
  member x.Delay f = try f() with ex -> Unsafe ex
```

Delegate.Sandbox: How it works and limitations (Initial release)

- In order to ensure that **IOEffect** types are only instantiated from inside the computation expression. A few examples of undesired behavior:
  - **IOSafe “42”**
  - **IOSafe (fun _ -> Directory.EnumerateFiles(".") |> Seq.length)**
- We use type encapsulation and we afterwards expose them with the help of active patterns.

```
type 'a IOEffect = private | IOSafe of 'a | Unsafe of exn
  with
    override x.ToString() = x |> function
      | IOSafe s -> sprintf "IOSafe %s" (s.ToString())
      | Unsafe e -> sprintf "Unsafe %A" e

let (|IOSafe|Unsafe|) = function | IOSafe s -> IOSafe s | Unsafe e -> Unsafe e
```

Delegate.Sandbox: How it works and limitations (Initial release)

- The computation and bindings works like the **Either** Monad where you either have a value of the type **IOSafe** or you have an **Exception** of the type **Unsafe**. The main point here is that the I/O side-effect **are NOT performed** and the computation catches the attempt by tainting the whole expression and providing the thrown **Exception** which can be re-thrown or logged in order to revise and fix the code.

Delegate.Sandbox: How it works and limitations (Initial release)

- To remove **System.Console** I/O side-effects, we need to execute some **SecurityPermissionFlag.UnmanagedCode** before we instantiate the **SandboxBuilder**. This is handled by **RemoveConsoleIO**. When the type is instantiated, the **System.Console.SetIn**, **System.Console.SetOut** and **System.Console.SetError** are set to **Stream.Null**. ~~Once this task is performed, the **SecurityPermissionFlag.UnmanagedCode** flag is removed in order for the new **AppDomain** runs with the minimal permission possible~~ (more on this later on).

Delegate.Sandbox: How it works and limitations (Examples)

- The following code:

```
open System
open System.IO
open Delegate.Sandbox

let addition x y = sandbox{ return x + y }
let power2 x = sandbox{ printfn "Injected side-effect"; return x * x }
let result = addition 21 21 >>= power2

printfn "Sum of x and y, then power2: %A" result
```

- Evaluates to:

  Sum of x and y and then power2: IOSafe 1764

  Note: No output is written to the console

- The following code:

```
let fooBar = sandbox{ return Console.ReadLine() + "FooBar" }

printfn "Prints only 'IOSafe FooBar': %A" fooBar
```

- Evaluates to

  Prints only 'IOSafe FooBar': IOSafe FooBar

  Note: No blocking readline or input from console.

Delegate.Sandbox: How it works and limitations (Initial release)

- We describe a few limitations we found while we were making the library:
  - **No code optimization**: When a project that refers to the library is built in **Release** mode, default is set to **Optimize code**, then it will not work as some of the code is transformed to use **Reflection** which is not supported in the **AppDomain**.
  - **Unit tests**: As stated before, **Reflection** is not supported and because NUnit uses this approach to execute the test, then it will not work either. This makes it really difficult to test code, mostly because **Unsafe** types are runtime and not compile time.
  - **F# Interactive (fsiAnyCpu.exe)**: As the computation expression is built on top of the **AppDomain**, it will not be possible to use this library in interactive mode (scripts, ...).
- Not to be used in production ☹

Delegate.Sandbox: How it works and limitations (Upcoming version)

- We describe a few limitations we found while we were making the library:
  - ~~**No code optimization**: When a project that refers to the library is built in **Release** mode, default is set to **Optimize code**, then it will not work as some of the code is transformed to use **Reflection** which is not supported in the **AppDomain**.~~
  - ~~**Unit tests**: As stated before, **Reflection** is not supported and because NUnit uses this approach to execute the test, then it will not work either. This makes it really difficult to test code, mostly because **Unsafe** types are runtime and not compile time.~~
  - **F# Interactive (fsiAnyCpu.exe)**: As the computation expression is built on top of the **AppDomain**, it will not be possible to use this library in interactive mode (scripts, ...).

Delegate.Sandbox: How it works and limitations (Upcoming version)

- Release Notes (version 1.5 - July 30 2015)
  - Major code refactoring (less code, more awesomeness)
  - Fixed critical security permission issue (unmanaged code could be invoked in sandbox)
  - Fixed issue with No code optimization in Release mode
  - Fixed issue with Unit tests. See Delegate.Sandbox.Tests project
  - Added support for nested sandboxes. Ex: **sandbox{ return sandbox{ return 42 } }**
  - Thanks to nested sandboxes, it's now possible to ensure that a library is 100% I/O side-effects safe if FSharp.Compiler.Services are used in combination with a Post-Build F# script (will be made available in v.2.0.0.0)
- Production Ready ☺

Delegate.Sandbox: How it works and limitations (Upcoming version)

- Fixed **critical security permission issue** (unmanaged code could be invoked in **sandbox**)
  - Once an **AppDomain** is instantiated with some permissions, it's not possible to remove them afterwards.
  - The library needs unmanaged code permissions to remove Console I/O effects (PInvoke to Win32 API). They must be added when the **AppDomain** is instantiated.
  - As it is not possible to remove the unmanaged code permissions, we just limit what is possible to evaluate in the continuation builder

```
[<Sealed>]
type SandboxBuilder() =
 ...
  [<SecurityPermissionAttribute(SecurityAction.PermitOnly, Execution = true)>]
  member x.Delay f = try f() with ex -> Unsafe ex
```

Delegate.Sandbox: How it works and limitations (Upcoming version)

- The following three issues/features are resolved by re-using the initial AppDomain (no need for reflection anymore):
  - Fixed issue with No code optimization in Release mode
  - Fixed issue with Unit tests. See Delegate.Sandbox.Tests project
  - Added support for nested sandboxes. Ex: **sandbox{ return sandbox{ return 42 } }**

```
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)
     ...
```

Delegate.Sandbox: How it works and limitations (Final version)

- Thanks to nested sandboxes, it's now possible to ensure that a library is 100% I/O side-effects safe if FSharp.Compiler.Services are used in combination with a Post-Build F# script

Note: Will be made available in v.2.0.

Delegate.Sandbox: How it works and limitations (Final version)

- Without nested sandboxes the following code is possible. F# is not pure, therefore it's not just enough with the function signature:

```
type FooBar = int IOEffect -> int IOEffect

let fooBar : FooBar =
  fun iosafe ->
    Directory.EnumerateFiles(".") |> Seq.length |> ignore // Do nasty stuff
    iosafe // Just return safe input value
```

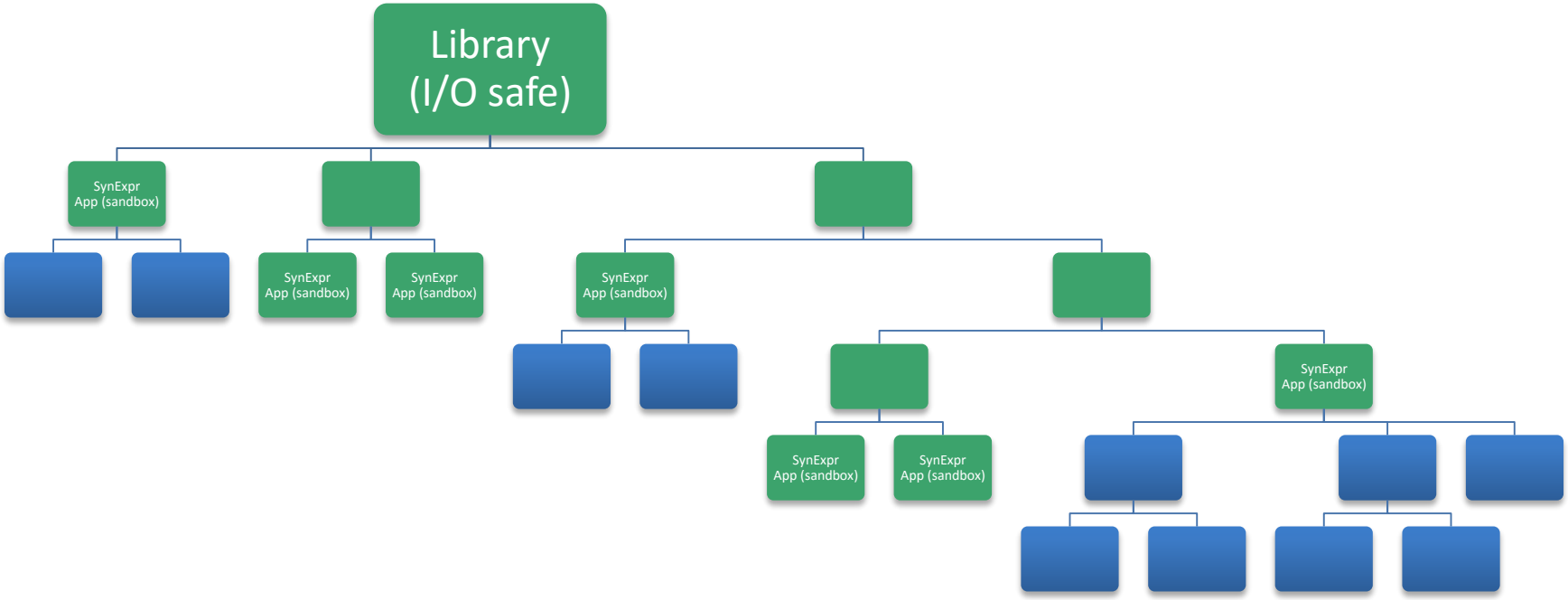
- Therefore we need to traverse the parsed tree of each file that is part of the library and ensure that all branches have a SynExpr.App (Ident sandbox)

Delegate.Sandbox: How it works and limitations (Final version)

```
#r "packages/FSharp.Compiler.Service/lib/net45/FSharp.Compiler.Service.dll”
open System
open System.IO
open Microsoft.FSharp.Compiler.Ast
open Microsoft.FSharp.Compiler.Range
open Microsoft.FSharp.Compiler.SourceCodeServices

let untypedTree file =
   let code = File.ReadAllText file
   let checker = FSharpChecker.Create()
   let projOptions =
     checker.GetProjectOptionsFromScript(file, "()")
     |> Async.RunSynchronously
  let parseFileResults =
     checker.ParseFileInProject(file, code, projOptions)
     |> Async.RunSynchronously
  let ast =
     match parseFileResults.ParseTree with
     | Some tree -> tree
     | None -> failwith "Something went wrong during parsing!”
  ast

let file = Path.Combine(__SOURCE_DIRECTORY__,@"Program.fs")

let walker =
   { new AstTraversal.AstVisitorBase<_>() with
       member this.VisitExpr(_path, traverseSynExpr, defaultTraverse, expr) =
         match expr with
         | SynExpr.App(_, false, (SynExpr.Ident(ident)), _, m)
             when ident.idText = "sandbox”
             -> Some (expr.Range)
         | _ -> defaultTraverse(expr) }

AstTraversal.Traverse(pos0, untypedTree file, walker)
```

- Compiler Services: Processing untyped syntax tree

Delegate.Sandbox: How it works and limitations (Final version)



- I/O safe libraries at compile time (parsed tree for each file)

Delegate.Sandbox: Demo - Show me some code

Questions?

U want more?

- Code is available @:
  - https://github.com/delegateas/Delegate.Sandbox
- Slides will be available @ MF#K (Files)
- Sign up @ MF#K for:
  - More ***fun***
  - Hands-on:
    - None so far …
  - Talks:
    - In the pipeline talks about: ***Rust***, ***F#*** ...
      - Upcoming: DST.Statistikbank.TypeProvider
- MF#K would like to thank our sponsor(s):

Forbundet af It-professionelle

PROSA