January 29th 2024

The F# Language

A functional-first programming language
for the .NET platform and the Web

Dr. Thomas C. Chust

Outline

- History and Related Languages
- Notable Features
  - Cross-Platform Support
  - Immutability
  - Structured Types and Pattern Matching
  - Units of Measure
  - Adaptive and Reactive Programming
  - Asynchronous Programming
- Interesting Frameworks
  - SciSharp
  - ML.NET
  - XPlot
  - FSharp.Data
  - FSharp.Data.Adaptive
  - Aardvark
  - WebSharper or Fable
  - Xamarin

History and Related Languages

- Created around 2005 in an effort to expand the set of available programming languages on the .NET platform
- Syntactically a member of the ML family of languages in general and similar to OCaml in particular
- Also influenced by Erlang, Haskell, Python, Scala, C#
- Fully cross-platform since around 2010
  (initially through Mono, nowadays with .NET Core)
- Current stable version 8.0, fully supported by Microsoft

Hello World!

Interactive Shell

```
PS1> dotnet fsi

Microsoft (R) F# Interactive, Version 12.8.0.0 für F# 8.0
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Um Hilfe zu erhalten, geben Sie "#help;;" ein.

> printfn "Hello world!";;
Hello world!
val it: unit = ()

> sprintf "Hello %s!" "world";;
val it: string = "Hello world!"

> #quit;;
```

Hello.fsx

```
let who = "world"
printfn "Hello %s! How are you?" who
```

Interactive Shell

```
PS1> dotnet fsi Hello.fsx
Hello world! How are you?
```

- F# can be used interactively (Jupyter notebook integration exists)
- F# can be used like a script interpreter
- Most code is an expression with a value
- The language is strongly typed
- Types can be inferred in many cases

Hello World!

Hello.fs

```fsharp
[<EntryPoint>]
let main args =
  let who =
    match Array.tryHead args with
    | Some v -> v
    | None   -> "world"

  printfn "Hello %s! How are you?" who

  0
```

Hello.fsproj

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Hello.fs" />
  </ItemGroup>
</Project>
```

Interactive Shell

```
PS1> dotnet run
Hello world! How are you?
PS1> dotnet run -- foo
Hello foo! How are you?
PS1> dotnet publish -r linux-x64 --self-contained
MSBuild-Version 17.8.3+195e7f5a3 für .NET
  Wiederherzustellende Projekte werden ermittelt...
  "S:\Personal\FSharpTalk\Hello\Hello.fsproj" wiederhergestellt (in "6.09 sec").
  Hello -> S:\Personal\FSharpTalk\Hello\bin\Release\net8.0\linux-x64\Hello.dll
  Hello -> S:\Personal\FSharpTalk\Hello\bin\Release\net8.0\linux-x64\publish\
```

- F# integrates with the .NET library ecosystem, project model and tools
- Several IDEs and editor plugins exist to support working with F# and MSBuild
- It‘s easy to compile, run, and test from the command line using the .NET Core tools
- Cross compilation is fully supported

Structured Types and Pattern Matching

```
type Direction =
  | XParallel
  | YParallel
  | Slanted of float

let directionAngle d =
  match d with
  | XParallel -> 0.0
  | YParallel -> 90.0
  | Slanted v -> v

type Measurement =
  {
    Layer     : string
    Position  : float * float
    Direction : Direction
  }

let measurementAngle { Direction = d } =
  directionAngle d

let myMeasurement =
  {
    Layer     = "1(0)"
    Position  = (4.0, 2.0)
    Direction = XParallel
  }

let myMeasurement' =
  { myMeasurement with
      Direction = Slanted 45.0
  }
```

- Union types can be used to represent simple enumerations, but cases can also have arguments
- Unions and records automatically support comparison operations, hashing, and serialization (unless explicitly suppressed)
- Records and union cases can be destructured using pattern matching wherever a variable is bound
- Match expressions can differentiate multiple union cases

Class Types and Primary Constructors

HellOO.fsx

```
type Greeter(who:string) =
  let greeting = $"Hello %s{who}! How are you?"

  member _.SayHello() =
    stdout.WriteLine(greeting)

Greeter("world").SayHello()
```

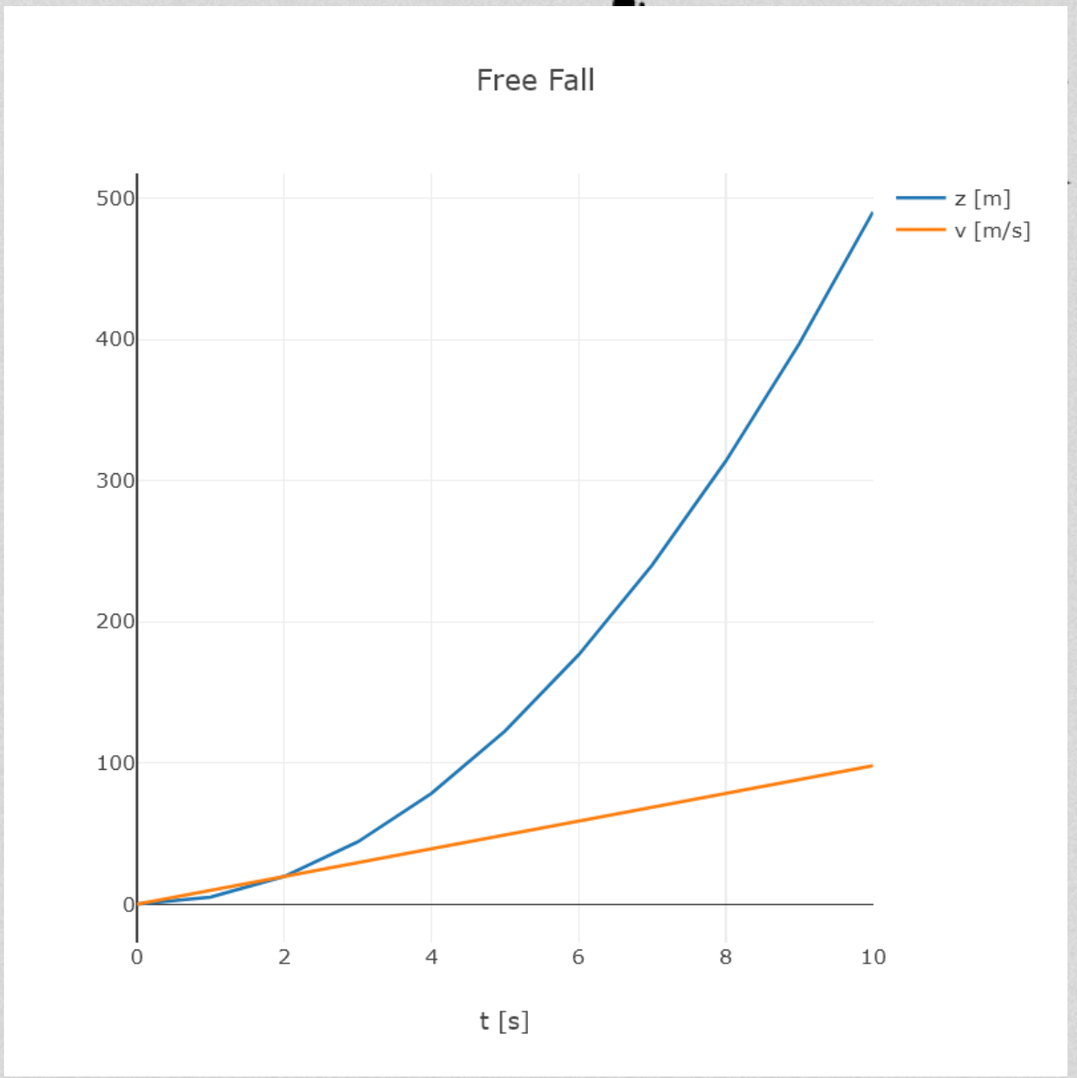
Interactive Shell

```
PS1> dotnet fsi HellOO.fsx
Hello world! How are you?
```

- F# also supports class types with object-oriented semantics
- Syntactic sugar eases the declaration of a primary constructor
- By default, instance variables are private
- By default, methods, properties, and events are public

Units of Measure

```
#r "nuget: XPlot.Plotly, 4.0.6"
#load "AutoDiff.fs"

open FSharp.Data.UnitSystems.SI.UnitSymbols
open Murphy.AutoDiff
open XPlot.Plotly

let g = 9.81<m/s^2>

let fall (t:DFloat<s>) = g/2.0 * t*t

let points =
  [ 0.0 .. 10.0 ]
  |> List.map (fun t ->
    let t = DFloat.Var (t * 1.0<s>)
    t, fall t
  )

let chart =
  let xs = points |> List.map (fun (t, _) -> t.Value / 1.0<s>)
  let zs = points |> List.map (fun (_, z) -> z.Value / 1.0<m>)
  let vs = points |> List.map (fun (t, z) -> z.Derive(t).Value / 1.0<m/s>)

  Chart.Plot [
    Scatter(x = xs, y = zs, mode = "lines", name = "z [m]")
    Scatter(x = xs, y = vs, mode = "lines", name = "v [m/s]")
  ]
  |> Chart.WithLayout (Layout(
    title = "Free Fall",
    xaxis = Xaxis(title = "t [s]")
  ))

chart
|> Chart.WithWidth 640
|> Chart.WithHeight 640
|> Chart.Show
```



Units of Measure

```fsharp
#r "nuget: XPlot.Plotly, 4.0.6"
#load "AutoDiff.fs"

open FSharp.Data.UnitSystems.SI.UnitSymbols
open Murphy.AutoDiff
open XPlot.Plotly

let g = 9.81<m/s^2>

let fall (t:DFloat<s>) = g/2.0 * t*t

let points =
  [ 0.0 .. 10.0 ]
  |> List.map (fun t ->
    let t = DFloat.Var (t * 1.0<s>)
    t, fall t
  )

let chart =
  let xs = points |> List.map (fun (t, _) -> t.Value / 1.0<s>)
  let zs = points |> List.map (fun (_, z) -> z.Value / 1.0<m>)
  let vs = points |> List.map (fun (t, z) -> z.Derive(t).Value / 1.0<m/s>)

  Chart.Plot [
    Scatter(x = xs, y = zs, mode = "lines", name = "z [m]")
    Scatter(x = xs, y = vs, mode = "lines", name = "v [m/s]")
  ]
  |> Chart.WithLayout (Layout(
    title = "Free Fall",
    xaxis = Xaxis(title = "t [s]")
  ))

chart
|> Chart.WithWidth 640
|> Chart.WithHeight 640
|> Chart.Show
```

- F# supports physical units of measure
  (or any units you define, really)

- Type inference / checking takes the units into account and understands the semantics of arithmetic operators

- IDEs can display inferred type annotations to assist the reader of the code

```fsharp
float<m/s ^ 2>
let g = 9.81<m/s^2>


DFloat<s> -> DFloat<m>
let fall (t:DFloat<s>) = g/2.0 * t*t
```

Type Providers

Provider.fsx

```fsharp
#r "nuget: FSharp.Data, 6.3.0"

[<Literal>]
let dataUrl = "https://en.wikipedia.org/wiki/Comparison_of_programming_languages"

type TypedData = FSharp.Data.HtmlProvider<dataUrl>

let data = TypedData.Load(dataUrl)
let fsharp =
  data.Tables.``General comparisonedit``.Rows
  |> Array.find (fun row -> row.Language = "F#")

printfn "Language:         %s" fsharp.Language
printfn "Functional?       %s" fsharp.Functional
printfn "Object-oriented? %s" fsharp.``Object-oriented``
```

Interactive Shell

```
PS1> dotnet fsi Provider.fsx
Language:         F#
Functional?       Yes
Object-oriented? Yes
```

- F# supports programmatic extensions to its type system
- Standard libraries exist to extract type structures from example data, like JSON, XML, or HTML documents, or relational databases
- At runtime, data with the same structure can be loaded and queried with collection functions

Adaptive Programming

React.fsx

```
#r "nuget: FSharp.Data.Adaptive, 1.2.14"

open FSharp.Data.UnitSystems.SI.UnitSymbols
open FSharp.Data.Adaptive

let timeToFloor (height:float<m>) (gravity:float<m/s^2>) : float<s> =
  sqrt (2.0 * height / gravity)

let height = cval 10.0<m>
let gravity = cval 9.81<m/s^2>

let dropTime = AVal.map2 timeToFloor height gravity
dropTime.AddCallback(printfn "dropTime = %.3f s")

transact (fun () ->
  height.Value <- 2000.0<m>
  gravity.Value <- 24.79<m/s^2>
)
```

Interactive Shell

```
PS1> dotnet fsi React.fsx
dropTime = 1.428 s
dropTime = 12.703 s
```

- A standard library exists for programming with „adaptive“ values
- This allows the construction of spreadsheet-style data dependency graphs
- Change transactions minimize the amount of superfluous callback events

Reactive Programming

Timer.fsx

```
open System
open System.Timers

let timer = new Timer(1000(*ms*), AutoReset = true, Enabled = true)

do
  let event, oddt =
    timer.Elapsed
    |> Event.partition (fun e ->
      e.SignalTime.Second % 2 = 0
    )

  use subscription = event.Subscribe(fun e ->
    Console.WriteLine("EvenSignalTime = {0:u}", e.SignalTime)
  )
  use _ = oddt.Subscribe(fun e ->
    Console.WriteLine("OddSignalTime  = {0:u}", e.SignalTime)
  )

  Threading.Thread.Sleep(3000(*ms*))

timer.Dispose()
```

Interactive Shell

```
PS1> dotnet fsi Timer.fsx
EvenSignalTime = 2024-01-27 15:22:04Z
OddSignalTime  = 2024-01-27 15:22:05Z
EvenSignalTime = 2024-01-27 15:22:06Z
```

- Events are first class values in F#
- In addition to supporting callbacks, events can be manipulated similar to sequence data types
- Event callback registrations support the RAII paradigm

Asynchronous Programming

Feeds.fsx

```fsharp
#r "nuget: System.ServiceModel.Syndication, 8.0.0"

open System
open System.ServiceModel.Syndication

let web = new Net.Http.HttpClient()

let getFeedAsync (uri:string) =
  async {
    let! ct = Async.CancellationToken
    try
      use! rs = web.GetStreamAsync(uri, ct) |> Async.AwaitTask
      use xr = Xml.XmlReader.Create(rs)
      return Ok (SyndicationFeed.Load xr)
    with
    | err ->
      return Error err
  }

[|
  "https://foundation.fsharp.org/announcements.rss"
  "https://www.tagesschau.de/xml/rss2/"
|]
|> Array.map getFeedAsync
|> Async.Parallel
|> Async.RunSynchronously
|> Array.iter (function
  | Ok feed ->
    for it in feed.Items do
      printfn "- %s" it.Title.Text
  | Error err ->
    printfn "Error: %O" err
)

web.Dispose()
```

- F# supports automatic CPS conversion for asynchronous workflows
- Instead of creating a tangled mess of nested callbacks, you write code that looks sequential

```fsharp
use server = new Server()
let x = server.Foo()
let y = server.Bar(x)
ignore y
```

```fsharp
async {
  use server = new Server()
  let! x = server.Foo()
  let! y = server.Bar(x)
  ignore y
}
```

Interactive Shell

```
PS1> dotnet fsi Feeds.fsx
- 2022 Board of Trustees Election
- F# Software Foundation Statement in Support of Ukraine
…
- Bündnis Sahra Wagenknecht: "Wir sind keine Linke 2.0"
- Bahnstreik endet vorzeitig in der Nacht zu Montag
…
```

Web Programming

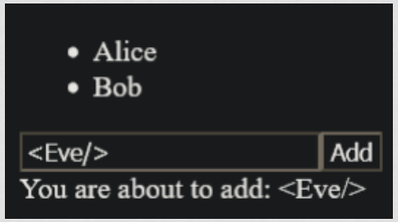
Client.fs

```
module [<WebSharper.JavaScript>] Client
open WebSharper.UI
open WebSharper.UI.Client
open WebSharper.UI.Html

let People =
  ListModel.FromSeq [
    "Alice"
    "Bob"
  ]

let [<WebSharper.SPAEntryPoint>] main () =
  let newName = Var.Create ""

  div [] [
    ul [] [
      People.View.DocSeqCached(fun (name: string) ->
        li [] [text name]
      )
    ]
    div [] [
      input [ attr.placeholder "Name"; Attr.Value newName ] []
      button [ on.click (fun _ _ -> People.Add(newName.Value)) ] [ text "Add" ]
      div [] [ text "You are about to add: "; text newName.V ]
    ]
  ]
  |> Doc.RunById "main"
```

Interactive Shell

```
PS1> dotnet new websharper-spa -lang 'F#' -o Web
Die Vorlage "WebSharper 6 Single-Page Application" wurde erfolgreich erstellt.
PS1> dotnet run --project Web
…
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
…
```

- WebSharper compiles F# to JavaScript and HTML



- A type provider can be used to interact with an existing HTML template
- F# code on the client and server side can share types and communicate transparently

Other Interesting Frameworks

- SciSharp
  Numerical primitives similar to numpy / scipy and bindings to deep learning libraries
- ML.NET
  Model-based machine learning and inference
- Aardvark
  Advanced 3D visualization and GPU programming
- WebSharper or Fable
  Web applications using F# on the client and server side. WebSharper integrates best with .NET tooling while Fable integrates best with Node.js tooling
- Xamarin
  Mobile applications on Android and iOS

Summary

- F# is a general purpose programming language with many useful libraries (some specific to F#, some from the larger .NET ecosystem)
- F# is a mature language with both an active OpenSource community and commercial backing
- F# combines useful features of functional and object-oriented programming languages
- The combination of strong typing, type inference, and interactive compilation makes the language well suited for rapid prototyping with fewer bugs
- (Cross-)compilation and deployment of full applications is pretty easy (thanks to .NET Core tooling)