F# vs Rust for F# programmers from Rust programmer

0.) Contents

1.) Introduction

When I started writing the introduction there was already 8 chapters. Originally I plan to be as objective as possible, but when I finished first part of the comparison, I figured out that is just not possible, because if F# have a feature that rust is missing I can just simply implement it, but it is not possible to do it the other way around.

So this is my subjective comparison, when I try to stay as objective as possible, but I do not let Rust lose points just because a simple library no one might need is missing.

At first glace both languages looks similar, but Rust have more braces (which I actually prefere, because it is clearer where tings start and and thanks to IDE's colorings, I can not say the same for F# where line wrapping screw up everything.

But I also learned few things when creating this comparson. If we ignore the obvious (how much Rust is better) then for example that F# can access characters of the string (I hope to never need it, because it is terrible).

1.1.) Changes to language

F#

Microsoft alone decides when to add something, and when to break something, and only if too many people disagree they reconsider fixing/reverting that change.

Rust

Anyone car write sugestion, or even whole **RFC** that will be discussed, reviewed, and if most people agree implemented, in case of breaking change (not security related, they have exeption and may come in hotfix release) you can read about it at least 12-18 weeks in advance and compiler will start producing warnings about it 6-12 weeks in advance so you know which code will change behavior in the future, and can make sure you know how to deal with it, and of course the error message contains short explanation, and link to the RFC or change PR.

2.) „Standard" library

Rust's standard library is relatively small, compared to .Net's one, but it contains the most important things, and for rest people can chose, even though there are libratioes like rand (random number generation), rayon (parallelism), byteorder (endian conversions), and many more that are used by standard library, but not exported out, so they are not forced on people that do not need them.

.Net on the other hand have quite a lot of things in the standard library, and before native build targets it was dragging all that with it in form of runtime, but it was too big so they started to separate it.

I will skip many things that are same or very similar, and only point out difference. **Important differences being bold.**

2.1.) Documentation

F#

Documentation is mostly provided only online, information directly reachable trough IDE (Visual Studio) is only a small fraction of that. So you better be **online** when you want to know something.

Rust

All documentation (not only for standard library, but all crates) is available **offiline**, or at doc.rust-lang.org/std/. Because that documentation is generated from code IDE can easily reach it. **You can look trough the code** online or offline whenever you want, if the very detailed documentation is not enough for you. I can not overstate how useful that is, yes with .Net you can decompile the code (and sometimes end up with more readable code than the original source code), but I am yet to see badly readable Rust library that is the best(or even *first) option for some task. (first in sense go to crates IO write keywords for what you want and try to use firs libraryx that matches these key words)

2.2.) Basic types comparison

2.2.1.) Inteagers

Both languages allow use of underscores in the number for better readability of long numbers.

F#
sbyte, byte, int16, uint16, int = int32, uint = uint32, int64, uint64, bigint, nativeint, unativeint
sifixes to specify type, y, uy or B, s, us, l or none, u or ul, L, U, I, L, n, un

Inconsistent naming. Two extra types for platform specific things are nativeint and unativeint, these types are specific to platform used (on amd64 unativeint = uint64) and represent pointer size. Two names for same type, default type is int. Implementation details and performance implications of bigint are forced on user by default. By default collections have size as int, which might not be enough, so **there are hacks like Array.LongLength**.

You can do math with different types and **compiler quesses** result type.

You can create byte[] array, by sufixing ASCII only string with capital B. Same for character constant '\0'B.

Rust
u8, i8, u16, i16, u32, i32, u64, i64, u128, i128, usize, isize
sifixes to specify type, *u8, i8, u16, i16, u32, i32, u64, i64, u128, i128, usize, isize* (exactly same as the types)

Very consistent naming. Two extra types for platform specific things are usize and isize, used a lot by standard library, these types are specific to platform used (on amd64 usize = u64) and represent pointer size, so any addresable memory location can be 0 + NUBERusize, so size of an array is usize, and **there are no hacks for sizes** needed. Rust support so many platfors so usize can be as small as 16 bits, and as big as 128 bits, but this might change in the future, and no library code depending on usize being big enough will need a change. There are multiple libraries providing big int functionality so you can pick the one that suits your needs most.

You can not do math with different types, you need to do conversion.

Yyou can create u8[] array, by prefixing ASCII only string with lovercase b. Same for character constant b'\0'.

2.2.2.) Floats

Both languages allow use of underscores in the number for better readability of long numbers.

F#
float32, single, float, double, decimal, BigRational

Even more inconsistent naming, because most other languages have float 32 bit, but here it is 64 bits.

Rust
f32, f64

Very consistent naming. Because these types map exactly to the most common HW types, there is no strict need for f128, f256 in general code, so these are not part of the standard library, but both are provided by other libraries, with HW specific optimizations, and other libraries provide floating point number with bigger precision and you are again not forced to single implementation, but you can chose the compromise that suits your needs best.

2.2.3.) Characters and text

F#

char, string

Often incorrectly is char said to represent character, or unicode code point, but in fact it is UTF-16 code unit. In .net **string type is used to represent** **single character** or sequence of character, which is terrible at best. 😛 So if you want to represent single character you will need to use string which is 20 bytes + 2 * (1 or 2 depending on the character, २ = 1, 🏠 = 2,).
UTF-16 **surrogate are not handled** by default enumeration.

Rust

char, string

Single character is represented by char (4 bytes).
Utf-8 string is represented by str. SIMPLE! Enumeration yelds **char** so you always get an valid character.
(For enumerating ASCII only text in tight loop cast from u8 to char can be as fast as 0 ticks on x86 CPU)

2.2.4.) Unit

F#

Special type **backed by static class** in the bacground, compared to **Rust's emty tuple**.
It is again unconsistent sometimes you need to write „unit“, and sometimes „()“.

2.2.5.) Tuple

F#

```
(element1, element2, ...)

Struct(element1, element2, ...)
```

Not much to say here, apart from the fact that there is no **easy way to to access the elements** apart from matching, and generic functions for 2 element tuples. Heap (default) Tuple, and ValueTuple are **two different incompatible types**.

Rust

```
Box::new((element1, element2, ...))

(element1, element2, ...)
```

If you want your tuple on the heap instead you need to put it there, and easiest way is to use box. You can **easily access any element** by writing dot and index of the element. Boxed tuple is still same type inside another type, so they are **fully compatible**. (Special case of empty tuple is **unit**, same as in F# but here it does not require class and new type in the background, and is always written as „()“)

2.2.6.) Array

F#

```
[|value1; vlue2|]

let a = [|1;  2; 3|]

a.[1] = 2 // single = sign for comparison

let b = [|1|]

b.[0] = 666
```

And here we get to the weird stuff. Because Array is not lets say „F# native“ type, but maped to .net's runtime arrays, all of its elements are **mutable**. Weird dot notation for indexing.

Rust

```
[value1, vlue2]

let a = [1, 2, 3, ]; // Rust support trailing coma after last element

a[1] == 2; // double == for comparison

let mut b = [1]; // you need to declare it as mutable

b[0] = 666
```

Arrays and all element are **immutable** by default.

2.2.7.) Mutable resizable array/list

Both languages have easy was to take two arrays (or other similar types) and concat/append them by creating third one.

F#

```
open System.Collections.Generic

let list = new List<T>()

list.Add(...)
```

Looks out of place in F# but uses underlying .net implementation from C#.

Rust

```
let vector = Vec::new();

vector.push(...);
```

Type can be infered. And looks natural in rest of Rust code.

2.2.8.) Slice

F#

```
[|0;1;2;3;4;5|] [1..(4-1)] = [|1;2;3|] // right bound excluded, by manual decrement

[|0;1;2;3;4;5|] [1..4] = [|1;2;3;4|] // right bound included

[|0;1;2;3;4;5|] [1..] = [|1;2;3;4;5|] // unbound to the righ
```

Slices do not really exist, they create new array (or list, or whatever the original type was). And it is up to the compiler and runtime to avoid copying if possible.

Rust

```
[0,1,2,3,4,5] [1..4] == [1,2,3] // right bound excluded

[0,1,2,3,4,5] [1..=4] == [1,2,3,4] // right bound included

[0,1,2,3,4,5] [1..] == [1,2,3,4,5] // unbound to the righ
```

Slice is an actual type, that is internally just two pointers (begin/end).

2.2.9.) Enum (C-like)

F#

```
type enum-name =

| value1 = integer-literal1

| value2 = integer-literal2
```

To specify the underlying type (sbyte, byte, int16, uint16, int32, uint32, int64, uint64, and **char**) you need to use **correct literals. All** values of the underlying type are valid values of the enum, not only the named values!

Rust

```
#[repr(C)]

enum EnumName {

   Value1 = 1,

   Value2 = 2,

}
```

Guaranteed to have same size as if you would define it in C header file, but you can use `repr(u8)` or any other (u8, i8, u16, i16, u32, i32, u64, i64, **u128**, **i128**) to use different underlying type, and **type inference** will handle literals. Notice char is not an option here). **Only** listed values are allowed and compile time checked! There is an special option `#[non_exhaustive]` to define enum as not exhaustive, so every matching code needs to use wildcard pattern, and not even then you are allowed to use any not listed value, but you must expect that some might be received. (Useful for libraries, that expect to add new cases in the future)

2.2.10.) Discriminated union

F#

```
[ attributes ]

type [accessibility-modifier] type-name =

   | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ] type2 ...]

   | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]


   [ member-list ]
```

These are less weird than C-like enums, only listed values can be used, each case is an constructor function.

Size of this monstrosity is **8 Bytes** for tag + **size of all fields in all cases**. 😮 (Reasoning being compatibility with rest of .net...), Possible aligments are 8 or more bytes depending on field types)

Rust

```
enum EnumName {

   Value1 (field1, field2, ...), // tuple like

   Value2 {field1:type1, field2:type2, ...}, // struct like

}
```

Consistent with C-like enum, **again** only listed values can be used, each case is an constructor function.

Unless otherwise specified (or optimized out 😛) size is smallest possible byte count for tag (**1 byte** for enums with less than 257 cases) plus **size of largest case**. Possible alignments are 1 ore more bytes depending on field types, and field can be internally reordered (unless forbidden) by compiler to reach the most optimal aligment.

2.2.11.) Result, Option and Async

Code at next two pages is simplification of real code, but the F# code does not even do all that the Rust one, because I just give up on trying to combine result and async together. As you can see I skipped reading constant ammount of bytes, and asserting they have expected value, and in „`read_player`" I give up because it was moving too far to the right, so I just created some default values. And even with these changes for F# that make it significantly shorter, the F# code is still much larger (I decreased font size to fit it on page, and removed empty lines), and arguably much less readable. I asked „Henzl, Jan" if he can help me with the F# code, because I wanted to show that Rust can do this much nicer, but what I did not expect was that F# will be that much terrible, but he did not came up with anything that would help F# much.

The issues I see on F# side for this example specifically:

- Combination of exceptions and results
- Very bad compatibility when combining Result and async
- No early return from function so we get stairs to the right

The F# example is incomplete, because the code was too crazy and too far right!

F#

```fsharp
open System.IO
open System
type Fail =
| Ex of Exception
| Me of string
type PlayerType =
| Human = 0x1
| Monsters = 0x2
| Any = 0x3
| EmptySlot = 0x4
type Player = {
    name: string
    id: uint32
    player_type: PlayerType
}
let read_u32_le (sr:FileStream) = async {
    return!
        try
            async{
                let buffer : byte[] = Array.zeroCreate 4
                let! read =
                    sr.ReadAsync(buffer, 0, 4)
                    |> Async.AwaitTask
                return
                    if read <> 4
                    then Error(Me "")
                    else
                        let span = ReadOnlySpan(buffer)
                        Ok (BitConverter.ToUInt32 span)
            }
        with
        | e -> async{ return Error (Ex e) }
}
let read_player (sr:FileStream) = async {
    return!
        try
            async{
                let! name_len = read_u32_le sr
                match name_len with
                | Error e -> return Error e
                | Ok name_len ->
                    // I give up on this example it is too long already
                    // and going to the right too much
                    let player = {
                        name = string name_len
                        id = uint32 666
                        player_type = PlayerType.Human
                    }
                    return Ok player
            }
        with
        | e -> async{ return Error (Ex e) }
}
[<EntryPoint>]
let main argv =
    async {
        use sr = new FileStream("test.pmv", FileMode.Open)
        let! game_version = read_u32_le sr
        match game_version with
        | Error e ->
            printfn "%A" e
            return 666 // whatever non 0 return code
        | Ok game_version ->
// there is no result builder :O
// so the code would go to the right too much and would not fit page in any dimension
            let! player = read_player sr
            match player with
            | Error e ->
                printfn "%A" e
                return 666 // whatever non 0 return code
            | Ok player ->
                printfn "%A have game version: %u" player game_version
                return 0
    }
    |> Async.RunSynchronously
```

Rust

```rust
use std::io::{Error, ErrorKind::InvalidData};
use async_std::{fs::File, io::ReadExt};

#[async_std::main]
async fn main() -> Result<(), Error> {
    let mut file = File::open("test.pmv").await?;
    let pmv = read_bytes::<3>(&mut file).await?;
    assert_eq!(b"PMV", &pmv);
    let game_version = read_u32_le(&mut file).await?;
    let player = read_player(&mut file).await?;

    println!("{:?} have game version: {}", player, game_version);
    // Player { name: "Kubik", id: 666, player_type: Human } have game version:
258
    Ok(())
}
#[derive(Debug)]
pub struct Player {
    name: String,
    id: u32,
    player_type: PlayerType,
}
#[derive(Debug, Copy, Clone, PartialEq, enum_primitive_derive::Primitive)]
pub enum PlayerType
{
    Human = 0x01,
    Monsters = 0x02,
    Any = 0x03,
    EmptySlot = 0x04
}

async fn read_player(file: &mut File) -> Result<Player, Error>  {
    use num_traits::FromPrimitive;
    let name = read_string(file).await?;
    let id = read_u32_le(file).await?;
    let player_type = read_u8(file).await?;
    let player_type = PlayerType::from_u8(player_type) // option returned here
        .ok_or(Error::new(InvalidData, "Player type invalid"))?;
    Ok(Player{name, id, player_type})
}
async fn read_bytes<const COUNT: usize>(file: &mut File) -> Result<[u8; COUNT],
Error>  {
    let mut bytes = [0u8;COUNT];
    file.read_exact(&mut bytes).await?;
    Ok(bytes)
}
async fn read_u32_le(file: &mut File) -> Result<u32, Error>  {
    let mut bytes = [0u8;4];
    file.read_exact(&mut bytes).await?;
    Ok(u32::from_le_bytes(bytes))
}
async fn read_u8(file: &mut File) -> Result<u8, Error>  {
    let mut byte = [0u8;1];
    file.read_exact(&mut byte).await?;
    Ok(byte[0])
}
async fn read_string(file: &mut File) -> Result<String, Error>  {
    let size = read_u32_le(file).await? as usize;
    let mut bytes = vec![0u8;size];
    file.read_exact(&mut bytes).await?;
    let string = String::from_utf8(bytes)
        .map_err(|e| Error::new(InvalidData, e))?;
    Ok(string)
}
```

2.2.12.) Pointers

All good languages need to interact with other languages, and operatin system, it is just impossible to have wrapper for everything in standard library, and for that you can not avoid pointers.

F#

There is „FSharp.NativeInterop“, but just by looking at it I know I do not want to work with it.

Rust

Rust is designed to interop with pointer based languages, so pointer is native type, that does not feel out of place as long as you do not want to read, or write to the pointer it feels like any other type, but when you want to read it (or write to it), you need to be in unsafe block, because compiler have no way to know if that pointer is valid or not. In C (api) libraries it is common that library have some init function that return untyped pointer, and want to receive that pointer to every function call (in can be „this“ from a C++ class, exposed as „void*“ to tell user that he should not read it, just keep it), this will feel pretty natural in rust, and you do not even need wrapper type.

2.3.) Pretty printing

Both languages provide strongly typed „format strings“.

F#

```
type SomeStruct = {
   InteagerX : int
   FloatY : float
   StringZ : string
}

let s = {
   InteagerX = 666
   FloatY = 3.14159
   StringZ = "X is 666, Y is π"
}

printfn "%A" s
// {InteagerX = 666;
// FloatY = 3.14159;
// StringZ = "X is 666, Y is π";}
```

„Printf.TextWriterFormat“ provides „format string“ that is statically checked, and it does not seems possible to construct new one at run time using „normal“ code.

Rust

```
#[derive(Debug)]
struct SomeStruct {
    integer_x : i32,
    float_y   : f32,
    string_z  : &'static str,
}

fn main() {
    let s = SomeStruct {
        integer_x : 666,
        float_y   : 3.14159,
        string_z  : "X is 666, Y is π",
    };

    println!("{:?}", s);
    // SomeStruct { integer_x: 666, float_y: 3.14159, string_z: "X is 666, Y is π" }

    println!("{:X?}", s);
    // SomeStruct { integer_x: 29A, float_y: 3.14159, string_z: "X is 666, Y is π" }

    println!("{:.2x?}", s);
    // SomeStruct { integer_x: 29a, float_y: 3.14, string_z: "X is 666, Y is π" }

    println!("{:#?}", s);
    // SomeStruct {
    //     integer_x: 666,
    //     float_y: 3.14159,
    //     string_z: "X is 666, Y is π",
    // }
    dbg!(s);
    // [src/main.rs:26] s = SomeStruct {
    //     integer_x: 666,
    //     float_y: 3.14159,
    //     string_z: "X is 666, Y is π",
    // }
}
```

Compared to F# there is much more options on how to pretty print values.

„format_args!“ macro allows construction of „format strings“ also is statically checked, and top level implementation of the returned type is very similar to F#‘s implementation (apart from not using so much heap space 😛). But to have also some difference here it is possible to construct it at runtime, while still being checked, but that is not mentioned on the official documentation (and it is in the unstable book experimental only, so no guarantees that API will not break in the future before being stabilized and put to official documentation), because it requires some clever tricks to take that compiler part that does the checking in the binary.

2.4.) Operators

F#

There is 27 standard operator categories. To be honest looking at that table and seeing that it does not even contains all the operators is very confusing to me, and I really do not mind few more braces in Rust that offer much better clarity.

Comparison operators **can be chained**.

```
let x = 1 < 2 < true <> false
printf "%b" x
// false, how quickly can you say why?
```

Rust

There is 19 categories in **expression precedence** table. Almost two-thirds in size, but contains complete precedence rules.

Comparison operators and range operators **require parentheses**. (Which prevents the example above)

2.5.) Units of measure

F#

For compatibility with rest of .Net they are **compile time only**.

Rust

Does not need to have compiler support, because it is easy to create new types for this. For example crate uom allows you to define your own units, or use the Internation System of Units (SI).

2.6.) Interfaces vs Traits

F# uses interfaces, and Rust Traits, but for most uses these two words can be used interchangeably. Both can inherit from multiple other interfaces/traits, and both can have generic types.

Main differences being compiler internals where Rust prefere to resolve them staticly as much as possible, while F# compiler does not care, but I can not call it really an advantage if default is dynamic dispatch, but .NET JIT can optimize at runtime, if it sees that only one type is used, but that is mostly theory of the future, because JIT does not take these oportunities too often.

Second difference is that **if you own the trait, or the type you can implement that trait for the type**, and there is even RFC to allow bin (top level application) crates to implement foreign Trait for Foreign type, so you can combine two libraies that do not know about each other and one provide Type, while the other provides Trait. This is already possible trough some tricks with wrapper types, but not having to write a new type would be an improvement. But it still means if you want a new function that will be on u32, Result<OK, u8>, and Option<f64> (randomly chosen example types), you can just create an Trait and implement it for these Types.

Third difference that is kind of suggested by the second one is that Traits can be implemented anywhere (limited by visibility scoping rules), while Interfaces can only be implemented at class definition.

Fourth difference is that Interfaces can define data, to be exact properties, which is just syntax suger for up to 2 functions.

Fifth difference would be that Traits can have associated type, which for example closest alternative would be additional generic in F#. Example `Ienumerable<T>` is just `Iterator` in Rust with associated type called „Item“ over which it is iterating.

3.) Functions

3.1.) Simple sytax example

F#

```
let add x y = x + y // common in F#, all types are infered to be int -> int -> int

let add (x:int) (y:int) : int = x + y // fully qualified types

let inline add x y = x + y // types are infered to be 'a -> 'b -> 'c (requires member (+))
```

You often do **not see** types without an IDE, because they can be **infered** by compiler.

Rust

```
fn add(x: i32, y: i32) -> i32 { x + y } //common in Rust with all types

fn add<T:std::ops::Add<Output = T>>(x: T, y: T) -> T { x + y } // generic one accepting anytwo instances of same type that can be added together

fn add<T:std::ops::Add<OTHER, Output = OUT>, OTHER , OUT >(x: T, y: OTHER) -> OUT { x + y } // less common one that can accept different types, where second one can be added to the first one, resulting in the third type. From standard types this will work only if T, OTHER and OUT are all same type.
```

When you have too many arguments, or too complicated types, it **might be less readable** (like the extra complicated example to add two things), but **you always see the types**. And there are ways to create more reasonable names for these long types, so the line is not that noisy, but I wanted to show the worst case example for Rust, to lessen the blow to F#. 😛

3.2.) Mutable object reference passed to the function

F#

```
let doSomething obj = obj.modify()
```

No indication that obj is mutable. **Hidden mutability by default** for everything from .net ecosystem. Mutable variables are highlighted by IDEs, but everything with hidden mutability (all the classes and .net common types like arrays) are not highlighted by any of the big IDEs.

Rust

```
let do_something(obj: &mut Obj) { obj.modify() }
```

Even without IDE you can see that the argument is mutable. **Mutability is explicit by default.** Mutable variables being highlighted by the IDEs by default, as in F# but there is no „rest of .net" types, so you can actually relly on it.

3.3.) Chain calls

3.3.1.) Array filter map

F#

```
let a = [|0;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15|]
let result =
    a
    |> Array.map(fun e -> (e, e * e))
    |> Array.filter(fun (_e,s) -> s > 5 && s < 100)
    |> Array.map(fun (e, s) -> (string e, s))

printf "%A" result
```

Rust

```
    const A : &[u32] = &[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];
    let result =
        A.iter()
        .map(|e| (e, e*e))
        .filter(|(_e, s)| *s > 5 && *s < 100)
        .map(|(e,s)| (e.to_string(), s))
        .collect::<Vec<_>>(); // this line is necessary to force evaluation, otherwise the object would be just
enumerator, and none of the lines would be actually executed

    println!("{:?}", result);
```

Rust prefers to work with iterators, over creating new array in every step and hoping that compiler can eliminate that memory allocation request on every step.

Other than that I would say these two are quite simmilar.

3.3.2.) Infinite sequence

F#

```
let seq = Seq.initInfinite (fun i -> i)
let seq2 =
    seq
    |> Seq.map(fun e -> (e, e*e))

for x in seq2 do
    printfn "%A" x
```

Rust

```
    let seq = 1..;
    let seq =
        seq
        .map(|e| (e, e*e));
    for x in seq {
        println!("{:?}", x);
    }
```

3.4.) Currying

F#

```
let add x y z =
   x + y + z

printfn "1 + 2 + 3 = %d" (add 1 2 3)
let add_4_y_z = add 4
printfn "4 + 5 + 6 = %d" (add_4_y_z 5 6)
let add_4_7_z = add_4_y_z 7
printfn "4 + 7 + 8 = %d" (add_4_7_z 8)
printfn "9 + 8 + 7 = %d" (add 9 8 7)
```

Rust

Well rust does not have currying, but I am not willing to give F# free victory 😛 so I used library. I think to write it myself, would be below 50 lines, but I decided to use library, that handles more edge cases with only few more lines, 97 to be exact 😛 using library is great way in Rust, because you can read the code to confirm if it does what you want the way you want. Reason I chose to use library was that I could not decide if to use macro on function side, or partial application side, so I look if there are some suggestions which one would be better, and find out that both are already available, each having some advantages and disadvantages.

```rust
#![feature(type_alias_impl_trait)]
#[cutlass::curry]
fn add(x: u32, y: u32, z: u32) -> u32 {
    return x + y + z;
}

fn main() {
    println!("1 + 2 + 3 = {}", add(1)(2)(3));
    let add_4_y_z = add(4);
    println!("4 + 5 + 6 = {}", add_4_y_z(5)(6));
    let add_4_7_z = add_4_y_z(7);
    println!("4 + 7 + 8 = {}", add_4_7_z(8));
    println!("9 + 8 + 7 = {}", add(9)(8)(7));
}
```

3.5.) Computation expressions / Try / Procedural macros / macro_rules!

In rust own implementation of try is experimental only which means it can be used only if explicitly enabled, in nightly compiler, and can change before being stabilized, there is only few guarantees about experimental code. You can read more at Tracking Issue for try_trait_v2, A new design for the ? desugaring (RFC#3058), or in the RFC extend ? to operate over other types.

F#'s computation expressions are probably more powerful than Try trait in Rust, but harder to implement (at lest for the first one to understand it). But they are less powerful than procedural macros in Rust, but again first one might be easier. And third alternative can be macro_rules!, which are as powerful as computation expressions, and for simple cases easier to write, but at some point it would be worth switching to procedural macros, because I personally see them easier to write in cases when more than few input combinations is required.

3.5.1.) Closure for Option / Result

You can sefely ignore next 3-4 sections if you only (mostly) care about Option and Result.
I was trying so much to make it similar that I overlooked the easiest way to do it, for Option and Result.

```
let x = {|| -> Result<i32,()> {
    let x: Result<i32, ()> = Ok(3);
    let x = x?;
    let y = 7;
    return Ok(x + y);
}}();
println!("{:?}", x);
```

And only Try is stable (or if you use nigly with Try enabled) you can do this for any type ☺

3.5.2.) Try

Technically can match functionality of Bind and ReturnFrom, but can be easily combined with async for Delay, generators are also unstable, but because combining things is easy in Rust, combining with generators would give Yield and YieldFrom. Return, Run, and Zero arent needed in Rust, when replicating computation expressions with Try. There is even an alternative unstable syntax that looks almost like F#'s computation expressions.

```
try {
    let x = Some (3)?;
    println! ("Got `{}`", x);
    let y = Some (4)?;
    println! ("Got `{}`", y);
    let z = Some (5)?;
    println! ("Got `{}`", z);
    println! ("Returning `Some ({})`", x * y * z);
    z * y * z
}
```

Example of implementation of the Try trait.

3.5.3.) macro_rules!

Is simple code replacing tool that can easily represent computation expression. It would be very simple to have macro_rules! that represent any method of computation expression, probably even any two, but more is to me a bit complicated, but definitly not impossible, because people already created macros that can parse other languages (I belive procedural macros are much better pick for that). In simple terms when writing macro_rules! you define input AST (Abstract Syntax Tree), or multiple variations, and for each you define output Rust code it should produce, so everything is nicely type checked ☺.
I tested and it actually works, but these macros have a limitation of „safe scopes“ so hackery (with procedural macros) is needed to access any variable from outer scope, which means either more complex transformation to do that, or pasing everything as an argument, which would defeat the purpose.

3.5.4.) procedural macros

Procedural macros are the more poverfull brother to the macro_rules! (and sometimes less, because input is valid Rust code, while macro_rules! can eat up any syntax like morse code ☺). They requite own crate and can do literally anything. They are written as function that accepts token stram and output token stream, both are AST (Abstract syntax tree) based so you do not have to worry about spaces, and there is surprisingle big amount of helpers that allow you to easily construct a more complicated things like functions, structures, adding fields to structures, and using them in newly generated methods. These are literally compiler extensions. Have a bit higher up fron investment than macro_rules!, but after writing first one they no longer look soo different.

They can easily do exactly what F#'s computation expressions do, but if I understood the issue correctly there is an issue with generics, so they will need to resolve some types at compilation, but that does not sound like too hard task, ~~I might try to write POC for that~~ ☺.

I did the POC and it works fine ☺ have a look github/computation_expression

```
let x = option!(
    let! x = Some(3);
    let y = 7;
    // I do not even need the forced return syntax 😛
    x + y
);
println!("{:?}", x);
```

3.5.5.) Computation expressions

Will be comming to rust at some point too, there is an request for an RFC Explore computational expressions in rust (do notation).

4.) Logging

F#

No standard way to do it, no standard interface to use, so many library options, that are incompatible with each other, so own abstraction is needed, because the common function(s) can be easily mapped, so such abstraction is possible. By default everything you pass to the log message is resolved, even if the logging is disabled. Triks to avoid that evaluation are either expensive, or badly readable, or both.

Rust

Crate log

The basic use of the log crate is through the five logging macros: error!, warn!, info!, debug! and trace! where error! represents the highest-priority log messages and trace! the lowest. The log messages are filtered by configuring the log level to exclude messages with a lower priority. Each of these macros accept format strings similarly to println!.

Any logging crate you pick will work with this and at most the only thing you need to do is to call some initialization rutine (simple console loggingin does not need that). All libraries that you decide to use and do log will also use this, so there is 100% compatibility for logging.

Not that it would be good practice, but you can even call expensive DB queries in trace logs, and they will not cause any performance issues as long as tracing for that module will be disabled, because level filter is evaluated, before, the content is evaluated, so they will not be executed.

5.) Code generation

F#

FSharp.Compiler.CodeDom too complicated to put it here

Rust

```
macro_rules! say_hello {
  // `()` indicates that the macro takes no argument.
  () => {
    // The macro will expand into the contents of this block.
    println!("Hello!");
  };
}
fn main() {
  // This call will expand into `println!("Hello");`
  say_hello!()
}
```

DRY (Don't Repeat Yourself), or Domain Specific Languages (DSLs) from Rust by example book. Rust macros are type safe, but allow you to use type inference to achieve great things, like statically generate all possible subselects and joins on sql tables (yes this is real usecase 😊), macros can be used for printing to console and logging (yes also real use case `printfn!` is probably one of the most directly used macros, and `log!` is probably one of the most indirectly used macros).

The logging exmample log crate provides error!, warn!, info!, debug! and trace! macros. Awsome thing about those is that if your build explicitly disables logging, they result in no code generated for logging. But even if you have normal logging enabled and configured to info and above, and have very complicated string to log, that needs to read some parts from database, but it is just trace log, even on hot code path it will just check that logging is not enabled for that level (and location) and the code inside will not execute.

6.) Ownership (and borrowing) responsibilities

6.1.) Who is responsible?

F#

Programmer is reposible for not creating race conditions, **garbage collector is responsible** for dealocation.

Rust

Compiler is responsible for refusing code that can contains race conditions, and statically planned dealocation at the end of scope. Simply said As many threads as you want can read same, thing when no one is writing to it, but only one can be writing when no one is reading, so there can never be a race condition. Will not bother you with too much details, but this is **one of the most important things about Rust called ownership and borrowing**.

Garbage collection is not needed because when scope owning an allocation ends it gets automatically deallocated. Ownership can be transfered from one scope to another (return value, function argument, another move), or borrowed. There are two kinds of borrow immutable and mutable. Borrow checker track their lifetimes, and disallow compiling code where two mutable references to same allocation would exist, or 1 mutable and non zero amount of immutable at same time.

Here is few examples of how borrowing works, and how detailed error messages it provides.

6.2.) Examples

6.2.1.) Iterator invalidation

Code

```
let mut v = vec![1, 2, 3];
for i in &v {
    println!("{}", i);
    v.push(34);
}
```

Error

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!("{}", i);
    v.push(34);
}
^
```

6.2.2.) Use after free

Code

```
let y: &i32;
{
    let x = 5;
    y = &x;
}
println!("{}", y);
```

Error

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at 2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following statement 0 at
4:18
    let x = 5;
    y = &x;
}
```

7.) Ecosystem

7.1.) Libraries

F#

You can use nuget.
Advantages:

- Many libraries (298,068 unique packages as the time of writing)
- Integrated so you can just add to the solutions
- Support private repositories
- Mostly can of works offline with local cache

Disadvantages:

- Most libraries are C# without having F# in mind*, and even F# libraries depend on too much C# parts (personal opinion)*
- Hard to do target specific dependencies
- Most nugets do not include good documentation, and source code

You can get source code from different git options, and other 3rd party options.

Advantages:

- You might get newer versions
- You may find libraries that are not avalable on nuget

Disadvantages:

- No integrated support
- You need to copy the source code
- No automated update check

Rust

You can use Cargo.

Advantages:

- Many libraries (78,877 Crates in stock as the time of writing)
- Integrated so you just name what you want*, (it is even easy to get multiple versions of same thing, so no version compatibility issues)*
- Support private repositories
- Supports any private or public git compatible repository
- Supports any path crates (private/shared drive with crates not being online)
- Supports target specific dependencies
- Supports build only dependencies
- Support development only dependencies
- Supports using only parts of crates trough features
- Uses Semantic versioning
- All *(most)* crates came with full source code
- Most crates have good documentation

Disadvantages:

- I do not know if there are any

You can get source code from different git options, and other 3rd party options, but it have no advantages, because you can do the same with Cargo

7.2.) Other differences

7.2.1.) If let statement

F#

```
let x = Some 1
match x with
| Some v -> printfn "%d" v
| None -> ()
```

Rust

```
let x = Some(1);
if let Some(v) = x {
   println!("{}", v);
}
```

7.2.2.) Namespaces and modules

F#

For compatibility with rest of .Net F# have both namespaces and modules.
Modules are compiled to static classes, for the sake of compatibility with rest of .Net.
This compatibility causes inconsistent rules for indentation.
You **can not have two modules with same name**, but different namespace path, this would be useful for example to have utils sub module for Seq, List, Map, ...

Rust

Only modules are available to separate code. Mostly module is code block, or file, but you can create module from more than one file, but it is much less common, compared to use of sub modules, and re-exports. For tree organization you can use folders.
You **can have as many modules with same names as you want** as long as they have unique full paths.

7.2.3.) Private vs public, mutable vs imutable, nullability

F#

Even though many claim F# to be imutable by default, and not nullable, unfortunatelly this part is very inconsistent.
Everything is **public** by default. You have only 3 visibility options, but none allow visibility scoping inside project.
F# only types are imutable by default, but there is a lot of hidden **mutability** in all .Net types like string, and F# does not have fully imutable variant of that. Not even IDE provides indication of that.
F# only types can not be null by default, but there is a lot of hidden **nullable** in all .Net types like string, and F# does not have not nullable variant of that. **Not IDE nor the compiler warns about possible nulls!**

Rust

By default everything is private, imutable, not nullable. You can make things visible only to specific modules if you want.

Rust does not even have a private keyword, because that is just default, and if you want something exposed you need to make it public.

Only 2 exceptions to that rule are trait (interface) functions, and crate's public exported interface.

7.2.4.) Exceptions

F#

Because of rest of .Net **F# is full of exceptions**.

Rust

Many will tell you that **Rust does not have exceptions**, but that is not exactly true. Rust have exceptions, but they aren't there for your error handling (even though you can abuse them for that if you try hard enough), but for makers of operating systems, runners, and other similar things that need to handle fatal errors (including not enough memory) from the inner code.

Most common way to cause exceptions are „panic!“ macro, „unimplemented!“ macro, „asert*“ group of macros, „unwrap()“, or „except(...)“ methods on many types, most notably Results and Options. And are used with intention to terminate the program if that condition is reached, because they are considered non recoverable errors.

7.2.5.) Tests

F#

Too many options, that often rely on reflection, and require 3rd party runner, and have no compatibility with each other.

Rust

Standardized way to run tests with „cargo test“, and to create an test just put attribute „#[test]“ on a function, write some „asert_eq!(1, 2)“ to it, and you are done you have your first failing test to fix (1 != 2). And you have great example for the exceptions here, „assert_eq!(...)“ will throw exception when the arguments do not match, and test runner should not crash on that, but report failing test.

7.2.6.) Interop with other languages

F#

Interop with any .Net language is easy.
Interop with any language using C API is possible.

Rust

Interop with any language using C API is easy.
So it is relatively easy to call Rust from F#, the other way around is more complicated (on .Nets side mostly)

7.2.7.) Autoimplement Traits

F#

„Traits“ like addition are automatically implemented for you in some cases, and you need to overload operators if you do not want them, or want different implementation.

But when you have any more complicated type like point with X, and Y coordianate you need to implement them yourself.

Rust

There are many good traits that can be „derived“ (put derive atrubute on top of type, like I already shown for Debug).
You can create your own Trais and prepare for them derive implementation, so the most standard one (outside of standard library) I guess is „serde“ that have two traits „Serialize“ and „Deserialize“ (actually more, but these are the most important I would say). This way you can easily „reflection free“ serialize and deserialize anything you want. Or from standard library „Eq“ for structural equality.

7.2.8.) Included tooling

F#

F# interacive – easy to run simple pieces of code

Rust

Compiler - 😛 might sound weird to include that one, but Rust compiler is really helpful, suggest you how you should name your types and variables, to not repeat same block of code multiple times, but to reuse it, that you can not write to same vector from 2 threads, ...
Cargo – Package manager, test runner, „solution" manager, and much more
rustfmt – helps you to format code, to have consistent style with all other libraries, so you can recognize types from functions just by casing of the name, even without help of IDE coloring.
rustfix – when new version is released that fixes something, but the fix is not backward compatible (which can happen from time to time, that public API needs to change slightly to fix something) you can run this and it will mostly automatically, or at worst it will just guide you how to fix the code, to work with the new version.
Clippy – helper with many additional warnings to reorganize code for more readability, or possible correctness issues. (It optionally comes with compiler, depending if you want minimal instalation, or not)

7.2.9.) Editions

F#

Latest version is incompatible with many older .Net libraries, and there is not much you can do about it if library author does not update. (Whole .Net issue)

Rust

All *(stable)* code that worked in 2015 (Rust 1.0) still works in latest version.
You can mix versions in your compilation tree without any issues.
Crates are *(mostly)* provided with source code, so if you need experimental crate that was not maintained you can fix it yourself.

7.2.10.) Debugging

F#

I can compare only the debugging I done under Barclays, and that is not very good.
But even if I ignore the cases of debugger often **crashing**, there is still the issue of not being able to place breakpoint wherever I want, but the breakpoint being moved to some of the outer blocks.

Rust

In last ~8 years of Rust development I was debugging the code like few times, and most of that was just to try it. Most of the time, the easiest solution is to look at the place where is the problem, and instantly see it, and if I do not know where exactly it is I can just add few prints to pinpoint it. **I really did not need to debug the code**. At first debugging was not supported for windows until something like 2015, then it was limited, and I just learned to live with it, and write most of the code on first try.

7.2.11.) Profiling compilation times

F#

```
dotnet clean
dotnet msbuild /clp:PerformanceSummary > normal-summary.txt
dotnet clean
dotnet msbuild /clp:PerformanceSummary /m:1  > serial-summary.txt
```

Based on this discussion, you get a lot of text with times.

Rust
You can use „self-profile“, or „timings“ compilation flag to get similar information in a table, or gantt chart, or flamegraph, that you can view for example in chrome profiles, with many details.

7.2.12.) Conditional compilation

F#
You can use compiler given list of defined values to do „#if #else #endif“ blocks even overlaping multiple statements, and having some branches not compilable.

Rust
You can use „cfg“ attribute on modules or functions using target platform, or any of the documented feature flags of your crate to include, or exclude these blocks of code from compilation, which means you can create not compilable blocks.
You can also use „cfg_attr“ on any attribute to conditionally include that attribute, some example can be exporting to json, only if it is wanted by enabling feature on the crate.
Or last option is „cfg!“ macro that youcan use in constant „if“ statements as an condition, or source of bolean literal for any other use. This way you have guaranteed that all branches of the if statement must be compilable.

7.2.13.) -1 vs – 1

F#
```
let x = 5 – 1
printfn "%i" x

//let x = 5 -1 // does not compile!"error FS0003: This value is not a function and cannot be applied."
//printfn "%i" x

let x = 5- 1
printfn "%i" x

let x = 5-1
printfn "%i" x
```

To me this seems like some really stupid spacing issue. What is so special about „5 -1“ without space, before 1, but with space after 5, that it can not compile? But then why it works for „5-1“ without any space?

Rust
```
dbg!(5 - 1);
dbg!(5 -1);
dbg!(5- 1);
dbg!(5-1);
```

All works, as I would expect from any programing language.

7.3.) Compilation errors

At best you can say that F# errors are same in the best of circumstances, but most of the time, Rust error messages are significantly better.

7.3.1.) Misspelled variable name variableX vs variablex

8.) Examples in the other language

8.1.) The "Why use F#?" series

Syntax
https://fsharpforfunandprofit.com/posts/fsharp-in-60-seconds/

https://play.rust-lang.org/?gist=3b064fbcea69a48628efec841347b93f

Sorting
https://fsharpforfunandprofit.com/posts/fvsc-quicksort/

https://play.rust-lang.org/?gist=eb9e14bce34e5906327d8f013f37d291

Downloading a web page
https://fsharpforfunandprofit.com/posts/fvsc-download/

https://play.rust-lang.org/?gist=d92f2f04bc9ee04dc88f3e352e793112

8.2.) The Rust Programming Language (The book)

Guessing Game
https://doc.rust-lang.org/book/ch02-00-guessing-game-

https://www.jdoodle.com/iembed/v0/qA7

while let Conditional Loops
https://doc.rust-lang.org/book/ch18-01-all-the-places-for-patterns.html#while-let-conditional-loops

https://www.jdoodle.com/iembed/v0/qA9

Final Project: Building a Multithreaded Web Server
https://doc.rust-lang.org/book/ch20-00-final-project-a-web-server.html

lib: https://www.jdoodle.com/iembed/v0/qAe

main: https://www.jdoodle.com/iembed/v0/qAf

8.3.) Config inspired by our confing being full of strings

I do not like the fact that config is full of strings, and then there are functions to convert them to domain types, and it is hard to model any confing that is even a bit complicated, like discriminated union that have some data.
In Rust for comparison there it is super easy to model most domain types directly, without need to manually write the string conversion. I even included untagged enum (DU) because the cases can be identified by the data, so there is no need to tag them. And another awsome thing is that I can use Option or even Result and it will just work, without need for some complicated null checking. (so the F# version is inclomplete 😝 I give up on the slightly complex DU) And one more Rust advantage I can serialize it to many more formats, often with just 2 more lines. 😝
I was considering adding NotEmptyStrings to F# version but decided it is too big to write or read, and would be bashing F# too hard for lack of this basic feature 😝

https://www.jdoodle.com/ia/rcX

https://play.rust-lang.org/?gist=2705cbff1f84864de7490d152b19a1cf

9.) Advantages and disadvantages as found on the internet

1. Advantages of F# as found on the internet

9.1.1.) Algebraic Data Types

But **Rust have that too**, and arguably even better. For example standard library have NonZero inteages.

9.1.2.) Transformations and Mutations

Rust makes this easy too, but it **also allows mutation to be safe** by not allowing multiple mutable references, which F# just can not guarantee, because of .Net.

9.1.3.) Concise Syntax

I would argue that F#'s syntax is too strict at indentation, and does not define clearly precedence in some cases.
I personally prefere Rust's braces, and type signatures on functions most of the time.

9.1.4.) Convinience

This includes things like creating and using complex type definitions, doing list processing, comparison and equality, state machines, and much more. And because functions are first class objects, it is very easy to create powerful and reusable code by creating functions that have other functions as parameters, or that combine existing functions to create new functionality. **I would say that this fully apply to Rust too.**

9.1.5.) Correctness

I need to disagree here, because F# needs to interact with .Net a lot, so there are null reference risks and other exceptions. **I would say Rust is again much better at this.**

9.1.6.) Concurrency

I would again give **Rust slight edge** at this, because in F# you have async, Task, Value task which are incompatible, and Rust have all that stadartized behind single interface.

9.1.7.) Completeness
I would say F# interacting with rest of .Net and using classes, breaks many of its advantages. But Rust in comparison does not need to interact with other languages directly as often, and even then it can do so in much better ways.

9.1.8.) Patern-matching
No win there, both are great.

9.1.9.) Type providers
There are crates for that in Rust, including C, C++, and many other languages for which F# does not have type providers.

9.1.10.) Sequential compilation
I have 3 problems with that. First it is not really true (because .Net).
And second I do not see it being significantly faster.
You can not easily read code from the important parts, because it must start with the details first, important things in the middle, and composition at the end, so the part that interest you most is somewhere in the middle.
So what should be the advantage? Rust is said to have slow compile times, but my observations are that similar code in C++, or C# is not significantly faster to compile, if at all.

9.1.11.) No Cyclic dependencies
Again no clear winner here, in my opinion, Rust allow them (and F# too), but Rust puts a lot of restrictions on them. Apart from the need to be next to each other I would say Rust restrictions are more strict.

9.2.) Disadvantages of F# as found on the internet

9.2.1.) Naming is more challenging
Another thing these two have in common, but I would argue it is good thing to not have multiple functions with same name most of the time. (Even though both languages allow multiple functions with same name trough interfaces, or traits)

9.2.2.) More complex data structures
I heard that in F# you can create something called „lenses“ which I understood as custom operators that allow accessing inner types more easily. I am not fan of so manny custom operators. In Rust I never have that issue, even though types are more complex here too. Maybe it is because rust allows easier copy and mutate scenario, or have better integrated functions to deal with those.

9.2.3.) Less advanced tools
Well this is a big win for Rust. 😛 Tools there are quite advanced already, and getting better every day. 😛

9.2.4.) Microsoft documentation is out of date, moved, 404
Well I can confirm that from my experience, that links on the internet are often incorrect ☹ You often get that page was removed, without any indication of what replaced it (if anything), or just 404 page not found. Microsoft probably saved few $ by not keeping old versions of the websites.

Lets look at Rust. Search „rust std u32“ on google and click first link (in case google fixes it).
First thing on that page says deprecated, and there is a link to the correct page you should go to (second link on goggle for me at the time of writing).
Or „rust tokio task“ on google and first link again.
Again outdated, but still working link for version 0.2.4 (at the time of writing), and next to it button „Go to latest version“ (1.17.0 at the time of writing).

So even though google points to old versions, these links can get you to what you was looking for with just one additional click, so **Rust is clear winner here**.

9.3.) Rust advantages as found on the internet
(skipping already mentioned ones)

9.3.1.) High performance
Well in both languages you can easily get decent performance without even trying (my personal experience).
But **Rust is just faster** if you try 😛

9.3.2.) Memory safety
Well F# (.Net) have garbage collector that mostly solves same issue, but in worse way, so **Rust wins this point**.

9.3.3.) Amount of crates on crates.io
Nuget have more packages, but they are mostly C# 😛 which makes Rust questionably better at this point.

9.3.4.) Community
If you do not count rest of .Net (C# 😛) the Rust community is bigger, and more useful.

9.3.5.) Backward compatibility and stability
.Net does not care about backward compatibility, and expect everyone to use latest version, even though they introduce breaking changes, they include them in changelog, but not warm affected people in any way, whenever rust introduces breaking change they have very clear warning produced by compiler, and most of the times also offer automated fixes.

And for stability, for .net that means, some versions are fixed, and receive security fixes. While rust is stable in a way that you can update to newer versions without worrying about compatibility, you can mostly automatically update from versions as old as 9 years, and even better, you can use libraries that old without worrying about compatibility they mostly still works, which I can not say even for a few years old nugets.

9.3.6.) Low overhead makes it ideal for embeded programming
Depends on definition of embeded, if you mean embedded x86, or Arm (server) then yes for sure, with std, and everything. **Rust beats F# there by a lot**, evem though F# can still be used.
But if you mean small arm, or RISC-V, or other more exotic architecture with 1 MB of RAM, or even less, then it is not that easy teritory for Rust, but definitly better, than F#, so **clear win for Rust**, because F# can not be used at all.

9.3.7.) Rust facilitates powerful web application development
You can easily do WebAssembly, and it is much smaller than .Net's because it does not drag huge runtime.

Makund a webserver is also super easy, I done it few times, and it was serving static, and dynamic content, after just few minutes of work ☺

9.3.8.) Rust's Static Typing Ensures Easy Maintainability
Both languages have that advantage, personally I would give very small win to Rust for function signatures, but they are are different so it is mostly preference.

9.3.9.) Cross-Platform Development and Support
Rust supports much more platforms, and cross compilation to non windows platforms is super easy.
So picking a winner is hard, Rust for possible targets, and F# for ease of targetting windows from non windows dev enviroment.

9.3.10.) Rust Has an Expansive Ecosystem
Clear win for Rust.

9.3.11.) Security
Rust pushes you to write more secure code, than other languages.

9.3.12.) Great error handling
Win for Rust, because error handling is much better there.

9.4.) Rust disadvantages as found on the internet

9.4.1.) Compile times

As I already stated, I disagree with that, unfortunatelly I do not have any comparison of similar F# code.

9.4.2.) Hard to get code to compile

I never really have big issue with that, Errors cleary say what happen, where it happen, and more often than not how to fix it. I would even say I have bigger issue with F# that I did not understand the issue. (Not even comparing to C++ where useless errors appears in unchanged and unrelated files)

9.4.3.) Learning Curve and Development

Well not for me. And F# looks like Rust with a lot of missing functionality 😛 (So a win for Rust in my personal experience)

9.4.4.) Strictness

I see that as a good thing so it will not let you do stupid errors.

9.4.5.) Rust is not a scripting language

I never understood this complain, I just write what I want it to do, and I never have any big issues with borrow checker, so no clue what crap other people are writing to have issues with that.

9.4.6.) No garbage collector

How could someone write that to the disadvantage is is beyond my understanding 😛

9.4.7.) Bigger binary files

I have only few real world applications written in two languages (none of them was F#, only C++ and C#) and Rust was always smaller, at the time of comparison, when it already contained more functionality that the app it was replacing.

10.) Questions

10.1.) Can you cover using Rust in Enterprises, i.e. what support model there is plus does it cover the basic functions required to work in an enterprise i.e. Kerberos auth etc.

https://serokell.io/blog/rust-companies List fron 2020 showing 9 „BIG“ companies using Rust. (Microsoft, Facebook, Amazon, npm, Discord, Dropbox, Cloudflare, Figma, Coursera)

https://www.rust-lang.org/production/users official list of companies that want to share that they use Rust in production. (Suprisingly Microsoft, Facebook, and Amazon are missing, even though they shared that on their own websites)

https://ferrous-systems.com/blog/sealed-rust-the-pitch/ Plan of standartization to allow Rust to be used in „mision critical“ systems, like automotive, or medical component.

“It turns out that over time the number of bugs we introduce into software — and this isn’t just a Microsoft thing, this is the software industry as a whole — is increasing. One particularly interesting part about that we found at Microsoft [is] that 70% of our very serious mission-critical bugs deal with memory safety”

“The best part about it all is that it’s completely memory safe. All of those bugs that you know about from C and C++ programming, like use after free, double free, all of these things and more are completely impossible to do in Rust. Rust by default is completely memory safe so those 70% of issues you just wouldn’t have.”

— Ryan Levick, Principal Cloud Developer Advocate at Microsoft

For the kerberos part of the question you can go with cross-krb5 high level library with „initiate“, „accept“, and „finish“´functions. Or libgssapi rust wrapper over what seems to be most popular implementation. Or you can write your own, by using some of many available libraries to help parse the messages to structures, or with the cryptography part, or go to the lowest level, an start with the most basic parser in nom, or just totally from scratch, there is just so many options available.

10.2.) Coming from Rust, what do you like/don't like about F#

10.2.1.) Like

F# is just C# with more Rust-like syntax, and some of the useful functions.
Access to nuget packages.

10.2.2.) Don't like

Hidden mutability in classes, and other CLI structs.
Two asigment operators.
C# nugets.
Stupid CLI naming rules, that DU case can not start with lovercase (Hard unsolvable error). In rust there is a warning for that and I can just put attribute on the enum (or just that one case) that says that I do not want recommended name casing for that item. (Useful for interop to have same name, or when generating code by the easier method, and concatenating identifiers.)
Compatibility with rest of .Net allow it to be useful, but also kills most of its advantages, because you need to study implementation details to know what are the valid values (can it be null?).
Useless intelisense, not existing documentation (yes there is something online, but often it does not contains what I look for).
Slice inclusive of right bound.
Automatic inclusion of everything in the namespace, so I do not know what is included and what not.
So called „single pass" compilation, that is not single pass, and strangly slow.
No doctest.
Public by default
No transparency if I work with pointer to data, or data directly.
Terrible support for strings, and almost not existant support for characters, what smartass think that string is good type to represent single character?
No code navigation for library functions, often I do not even get to the function I clicked on (e.q. System.Linq GroupBy sends me to some generic Linq with 2 methods „new", and „GetEnumerator", but no implementation)
Many features are reflection based which means runtime cost, instead of compile time cost.
No IDE autocompletion for items that are not yet included, compared to IDEA for Rust that suggets what I want sometimes already on first character.
No IDE autocomple DU matching, in IDEA for Rust when I write „match x" (assuming x is something that can be matched) I can just let it generate all cases with empty code block, if new case is added I can just go to match and let it generate new match arm for that case.
No IDE autocomplete when constructing object, even if I specify typename before the first field there is no option to generate the rest, and I need to go to type declaration and and copy all the names from there.

... Should I really continuee the list of things that Rust, or IDEA does better?

10.3.) What kind of 'mindset' does Rust teach its users? (for example F# I think teaches/preaches heavy domain modeling with DUs etc...)

Rust also point tovards hevy domain modeling, even to the point that standard library comes with NonZero inteages. https://www.rust-lang.org/learn say start with the book. Where final example is multi-threaded web server with only std, where you create your own threadpool (struct), enum (discriminated union for F#pers) for messages that can contain function, and worker (struct). But when reading the book you will see much more example of single field struct (which could be single case enum, but there is no need for that).
Then there is Rust by Example which is also full of custom types to model „domain" examples.

But one important difference is that Rust do not tell you to abandon all mutability, it only tells you to limit it, and if you use it, it do not let you to do stupid things like adding to a dictionary from multiple threads at same time, because that requires mutable reference, and at most one mutable reference can exist at same time.

10.4.) How to do asynchronous call (users point of view)

There is whole book just for the async topic. It is very simmilar to F#'s async (from outside, and also very high level inside, I mean the top level types have same fields, but F# have them as pointers).

You just call a function and you get returned an "std::future::Future", that you can await inside async function to get the result, and if you decide to never await them compiler will complain, to let you know you created future that is never executed. I recommend checking out chapter 9 Final Project: HTTP Server in the book.

10.5.) Do you „fight the borrow checker"

Short answer **NO**.
Long answer would be that most of the time I write the code correctly first try, and when I get borrowchecker error usually I notice, that I write something slightly different that what I wanted (for example used different variable). The only case when I remember actually fighting the borrow checker was UI when I was trying to render data with **cyclical mutable refferences**, and for that I decided to avoid borrow checker, because I do not think it would ever allowed that, but generally it is good idea to not allow that, but I wanted to to it without copying the data. Small explanation of the data and why there was **cyclical mutable refferences**: It was describing a game world of C++ game at run time, so every change was instant. Thera was an world map that have squads and units on it, but there was also list of all entities (squads, units, objects, and many other). Each squad have list of its units. Each unit know its squad. Each squad, and unit knows its target (if any) and it can be squad, unit, or few other entity types. I wanted to have this with search support and tree view so from any point I could traverse to whatever I wanted. For example in lits of entities fin unit, from that unit open the squad, then open all units in that squad and copare if they have same target, or not. To be clear for read only view there would be no issue, I would just downcast all the references to immutable, but I wanted the ability to change things at any point in the tree.

10.6.) How Rust deals with covariance and contravariance

Because I never used these words before I need to look them up. Here is an F# example: Provide covariance/contravariance language support with syntaxes in F# in sync with covariance/contravariance in C#/VB

So lets try it in Rust. https://play.rust-lang.org/?gist=1183b65312a00e3ad44f506b1c3e4ff6

So we finally have something that F# is clearly better at, Rust's Vec is only for homogenous types. I do not think it is possible to mark **instance** with an trait. For the example it would make sense to use an enum wrapping the data, or because it is an independed from the data, just add it as another field and have enum without any data. I wonder if that tag in F# takes any space, like pointer virtual function table added to each object.

10.7.) Generics a inheritance

Both languages rely quite a lot on generics, good examples would be Option<T>, and Result<OK, ERR>, there is even SPECS Parallel ECS (Entity Component System), that run more than few games written in Rust, but the difference to F# solutions is the use of homogenous data trough generics, while F# makes it easy to create heterogenous collections (thanks to fact almost everything is internally just a pointer, and often pointer to pointers).
Not sure how to explain it, but I see the advantage of heterogenous collection, but I do not need them in Rust, so I wonder what exactly is the reson that I do not need them.If I would be missing them, it would be clear win for F#, but I just do not need them, but can not point out which part of Rust eliminates this need, they can be easily replaced with enum, but I do not think this is the reason, because discriminated unions are in F# too.

F# have inheritance between classes, I hope it is not widely used and it is remark that it is still just another .Net language, because inheriting a class and implementing an interface is exactly the kind of dimond inheritance that I do not like.

Both languages have concept of inheritance between interfaces/traits. (I learned just now that this it is called inheritance, I always called it supertraits). Both interfaces, and traits can provide default implementation, which is good for having shared behavior, but not shared data. In rust if you want functions to operate on some „base" data, you extract that to a separate struct and implement functions for that, then whichever struct have this as an field can call these functions.

11.) Summary

	F#	Rust	Better?
Documentation	Online only, many missing, often missing details	Ecelent, detailed, offline, from actual code	Rust by a lot
Primitive types	Inconsistently named	Very consistent naming	Almost feature parity, but Rust have better naming
Other basic types	Many inconsistencies, and **mutability** because of .Net's CLR	No unwanted mutability	Rust by a lot
String formatting	Possible	More easy options	Rust
Operators	A lot, custom operators can be created, and they can be overloaded, auto-implemented	No overloading, but can be implemented for new types	F#? Even though questionable at best, but also opinion based, if you like overloading operators. I prefere Rust
Units of measure	Compile time only	Libraries with options exist, so you can chose compile time, or runtime	Rust?
Function syntax	Type inference, useless without IDE, or tooling help to	All types required	F#? / Rust? Depending on the IDE
Hidden mutability	Everywhere because of .Net's CLR	Not a lot, often mentioned in documentation	Rust by a lot
Chaining calls	|> (or one of many other operators)	.	Feature parity. Syntax preference? Rust is simplier
generics	Yes widely supported	Yes widely supported	Parity? Or Rust because whole standard library is for Rust, and not C#.
Data inheritance	Yes	No	I say No is better
Behavior inheritance	Interface default impl	Trait default impl	Parity
Heterogenous collections	Yes	Limited trough boxing, and not needed	F# (but everything is boxed there)
Currying	Yes	Libraries can do it, but not that nice	F#?
Computation expressions	Yes	Not yet, but good enough alternatives already	F#, but I would say feature parity is easy to achieve?
Logging	Inconsistent options (Microsoft is trying to standardize the interface, but no success yet)	Consistent and compatible	Rust by a lot
Code generation	Yes...	Yes with options for beginners, intermediate, and experts	Rust by a lot
Libraries	A lot but poor compatibility and support	A bit less, but much better compatibility and support	Rust
Concurrency issues?	Easily possible (some eliminated by immutability, compared to C#, but still too many hidden mutability)	Almost impossible thanks to borrow checker (eliminated by compile time errors)	Rust by a lot
Garbage collection	Yes	No	Rust
Namespaces	Yes (but module names must be unique across all namespaces)	No	Rust
Modules	Yes (Class backed)	YES (compile time only)	Rust

Visibility	Public (default)	Private (default)	Rust (default, othervise parity)
Exceptions	Yes (hidden everywhere)	Not used for control flow	Rust by a lot
Test	Many incompatible testing options, many reflection based	Options are compatible, because they are just extensions to cargo's testing framework, no reflection	Rust
Interop	Great with .Net, complicated elsewere	Great for anything with C interface (including .Net if the work on .Net's side is done)	Rust by a lot
IDE, and tooling	Fine	Great	Rust
Editions	Hidden breaking changes	Compiler informs about most of the breaking changes	Rust
Debugging	Fine	Not great yet, but possible, a bit Linux focussed	F#
Conditional compilation	Yes-ish	Great	Rust by a lot
Compiler errors	Often crapp	Mostly great	Rust by a lot
Platform support	**.Net, x86, amd64**, arm (only some), Only windows and linux, WebAssembly (it caries some subset of .Net bundled to be able to run natively)	**x86, amd64, aarch64**, for windows and linux, **arm, armv7**, mips, mips64, **powerpc**, **powerpc64, riscv64gc**, s390x, **WebAssembly**, and that is only full support with all tooling available, many more architectures is available by building from sources, and for most of those std library is ready	Rust by a lot
Performance	Fine	Better ☺	Rust

12.) Revisions

21.4.2022 – fixed some typos, added computation expression section, added summary

26.4.2022 – fixed some typos, extended computation expression section with code smales how Rust can do it

9.5.2022 – fixed some typos, Computation expression with github link instead of Teams mention, few examples for Ownership and Borrowing, question do you fight the borrow checker, add chapter numbering, small info about std, introduction, new chapter Examples in the other language

12.5.2022 – chapter -1 vs - 1

27.5.2022 – chapter about config parsing, chapter Interfaces vs Traits, chaper about changes, How Rust deals with covariance and contravariance, generics a inheritance, few more lines in summary

5.9.2022 – Added complain about not being able to have same module name more than once, Don't like „Two asigment operators." More spell checking.