FUNKTIONALES C#

oder

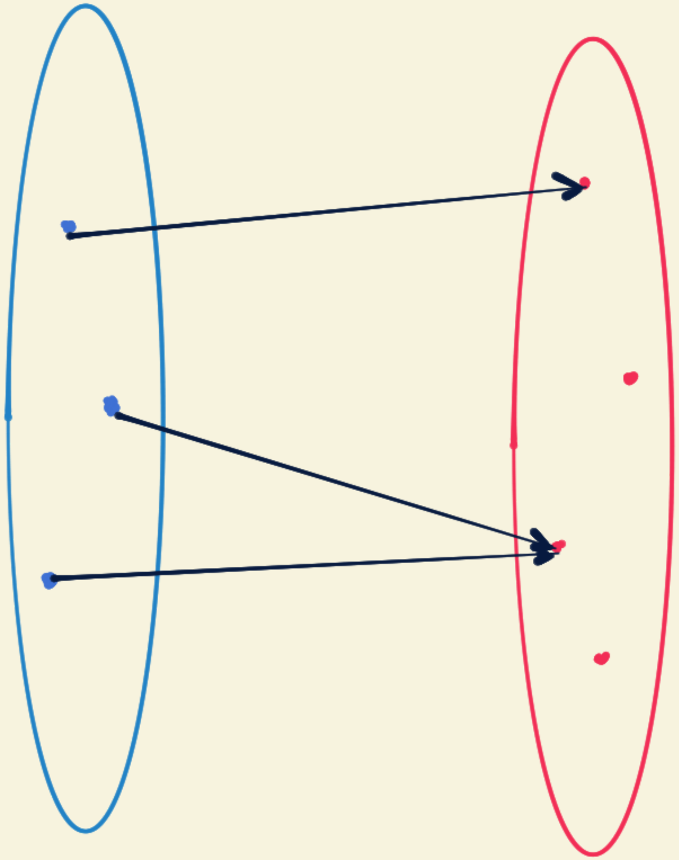
BRAUCHEN WIR F# ÜBERHAUPT NOCH?

AGENDA

- Begriff FP
- Funktionen
- Datentypen
- funktionale Muster
- Ausblick
- Fragen / Antworten

WAS IST FP?

REINE FUNKTIONEN

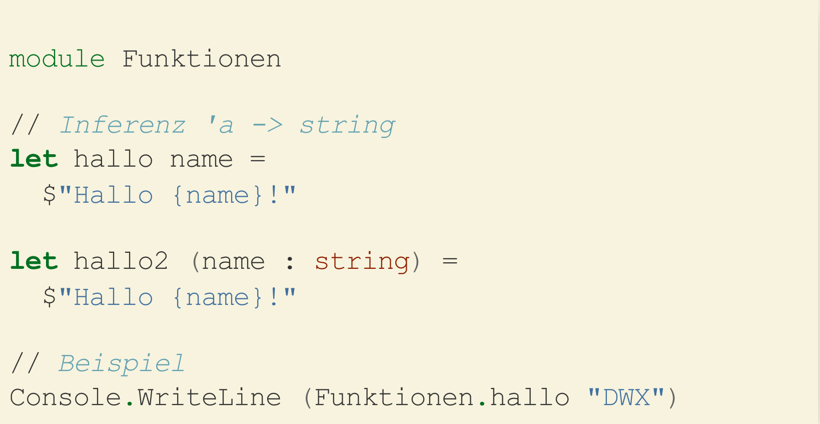


EXPRESSIONS VS. STATEMENTS

$i += 1$ ✗

$i' := i + 1$ ✓

FUNKTIONEN

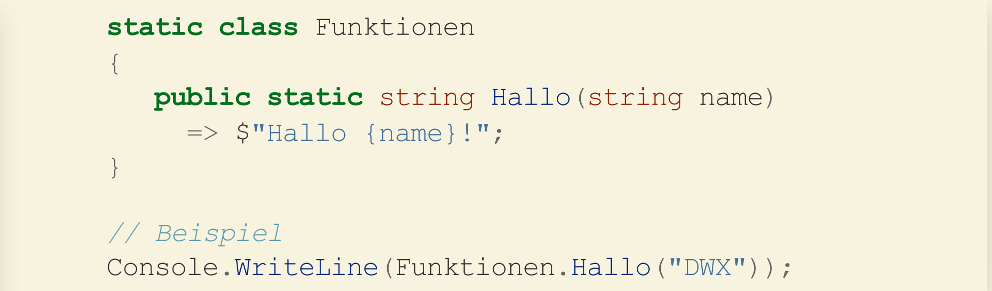


```fsharp
module Funktionen

// Inferenz 'a -> string
let hallo name =
  $"Hallo {name}!"

let hallo2 (name : string) =
  $"Hallo {name}!"

// Beispiel
Console.WriteLine (Funktionen.hallo "DWX")
```



```csharp
static class Funktionen
{
   public static string Hallo(string name)
     => $"Hallo {name}!";
}

// Beispiel
Console.WriteLine(Funktionen.Hallo("DWX"));
```

CURRYING

```
// untypisch
let addNotCurried (a,b) =
   a + b

// int -> int -> int
let add a b =
  a + b

let add2 a =
  fun b -> a + b

Console.WriteLine (add 3 5)
```

```
int AddNotCurried(int a, int b)
  => a + b;

Func<int,int> AddCurried(int a)
  => b => a + b;

Console.WriteLine(Funktionen.AddCurried(3)(5));
```

PARTIAL APPLIKATION

```fsharp
// add : int -> (int -> int)

let add10 =
  add 10

add10 5 // = 15
```

```csharp
// Func<int,int> AddCurried(int a)

Func<int, int> Add10
  = AddCurried(10);

Add10(5) // = 15
```

HIGHER-ORDER

```
// ('a*'b -> 'c) -> 'a -> 'b -> 'c
let curry f a b = f (a,b)

// ('a*'b -> 'c) -> 'a -> 'b -> 'c
let partialApply f a =
   fun b -> f (a,b)

// Beispiele
let add'(a,b) = a+b

let add10alt1 =
   partialApply add' 10

let add10alt2 =
   curry add' 10
```

```
Func<T1,Func<T2,T3>> Curry<T1,T2,T3>(Func<T1,T2,T3> f)
   => v1 => v2 => f(v1,v2);

Func<T2,T3> PartialApply<T1,T2,T3>(Func<T1,T2,T3> f, T1 v1)
   => v2 => f(v1, v2);

// Beispiele
Func<int,int> Add10alt1
  => PartialApply<int,int,int>(AddNotCurried, 10);

Func<int,int> Add10alt2
  => Curry<int,int,int>(AddNotCurried)(10);
```

Action/Func

```fsharp
// string * string -> unit
let printName (punct, name) =
   printfn "Hallo %s%s" name punct

let printNameExcl =
   partialApply printName "!"
```

```csharp
void PrintName(string punct, string name)
  => Console.WriteLine($"Hallo {name}{punct}");

Action<string> PrintNameExcl(string name)
  => FunExtensions.PartialApply<string,string,?>(PrintName, "!");

// Brauchen
Action<T2> PartialApply<T1,T2>(Action<T1,T2> f, T1 v1)
  => v2 => f(v1, v2);
```

SRTP

STATICALLY RESOLVED TYPE PARAMETERS

(F# ONLY)

siehe SRTP und Constraints

BEISPIEL

```
let inline srtpAdd a b =
   a + b

srtpAdd 1 2 // = 3 : int
srtpAdd 1.0 2.0 // = 3.0 : double
```

Typ:

```
val inline srtpAdd :
  a: ^a -> b: ^b ->  ^c
    when ( ^a or  ^b) : (static member ( + ) :  ^a *  ^b ->  ^c)
```

geht auch nicht-statisch

```
let inline trim (s : ^s when ^s : (member Trim : unit -> ^s)) =
   (^s : (member Trim : unit -> ^s)  s)

trim "   Hallo    " // = "Hallo"
```

allerdings Typ-Inferenz hier schwierig

DATENTYPEN

RECORDS

```fsharp
type Person =
   {
      FirstName : string
      LastName : string
   }

let max =
   {
      FirstName = "Max"
      LastName = "Mustermann"
   }
```

```csharp
public record Person(string FirstName, string LastName);

public record Person2
{
  public string FirstName { get; init; }
  public string LastName { get; init; }
}

var max = new Person("Max", "Mustermann");
```

RECORD-UPDATE / NONDESTRUCTIVE MUTATION

```
//  { FirstName = "Min"; LastName = "Mustermann" }
let min =
   { max with FirstName = "Min" }
```

```
var min =
   max with { FirstName = "Min" };
```

DECONSTRUCTION

```
let { FirstName = fn; LastName = ln } = min
```

```
min.Deconstruct(out var fn, out var ln);
var (fn2, ln2) = min;
```

PATTERN-MATCH

```
let patternMatch =
   function
   | { FirstName = "Min"; LastName = _ } ->
      "Hi Min"
   | { LastName = "Mustermann" } ->
      "Hey a Mustermann"
   | p ->
      $"Hello {p.FirstName}"
```

```
string PatternMatchRecords (Records.Person person) =>
   person switch
      {
         (FirstName: "Min", LastName: _) => "Hi Min",
         { LastName: "Mustermann" } => "Hey a
         Mustermann",
         Records.Person p => $"Hello {p.FirstName}",
         // nicht nötig - C# merkt das nicht
         _ => $"Hello {person.FirstName}"
      };
```

UNION TYPES

IN *F#*

```fsharp
type Maybe<'a> =
    | Nothing
    | Just of 'a

// Beispiele

let example1 : Maybe<int> = Nothing

let example2 = Just 42
```

PATTERN-MATCHING

```
module Maybe =

   let withDefault a =
      function
      | Nothing -> a
      | Just a -> a

// Beispiele

Maybe.withDefault 0 Nothing // = 0
Maybe.withDefault 0 (Just 42) // = 42
```

IN *C#*

Übersetzung in Klassen

```
public abstract class Maybe<T>
{
   public abstract Tres Match<Tres> (
      Func<Tres> onNothing,
      Func<T, Tres> onJust );

   private Maybe() { }
   public sealed class NothingCase : Maybe<T> { ... }
   public sealed class JustCase : Maybe<T> { ... }
}
```

BEISPIEL

Übersetzung in Klassen

```
public abstract class Maybe<T>
{
   public static Maybe<T> Just(T value) => new JustCase(value);
   public static Maybe<T> Nothing => new NothingCase();
}

// Beipsiel
var nothing = Maybe<int>.Nothing;
var just42 = Maybe<int>.Just(42);
```

NOTHINGCASE

```
public sealed class NothingCase : Maybe<T>
{
   internal NothingCase() { }

   public override Tres Match<Tres>(
      Func<Tres> onNothing,
      Func<T, Tres> onJust)
      => onNothing();
}
```

JUSTCASE

```
public sealed class JustCase : Maybe<T>
{
   public T Value { get; init; }
   internal JustCase(T value)
   { Value = value; }

   public override Tres Match<Tres>(
      Func<Tres> onNothing,
      Func<T, Tres> onJust)
      => onJust(Value);
}
```

PATTERN-MATCHING

```
public abstract class Maybe<T>
{
   public T WithDefault(T defaultValue)
      => Match(() => defaultValue, x => x);

   public T WithDefault2(T defaultValue)
      => this switch
         {
            JustCase j => j.Value,
            NothingCase => defaultValue,
            // sonst Warnung
            _ => throw new InvalidOperationException()
         };
}
```

MUSTER / ABSTRAKTIONEN

FUNKTOR

```
map : ('a -> 'b) -> F<'a> -> F<'b>
```

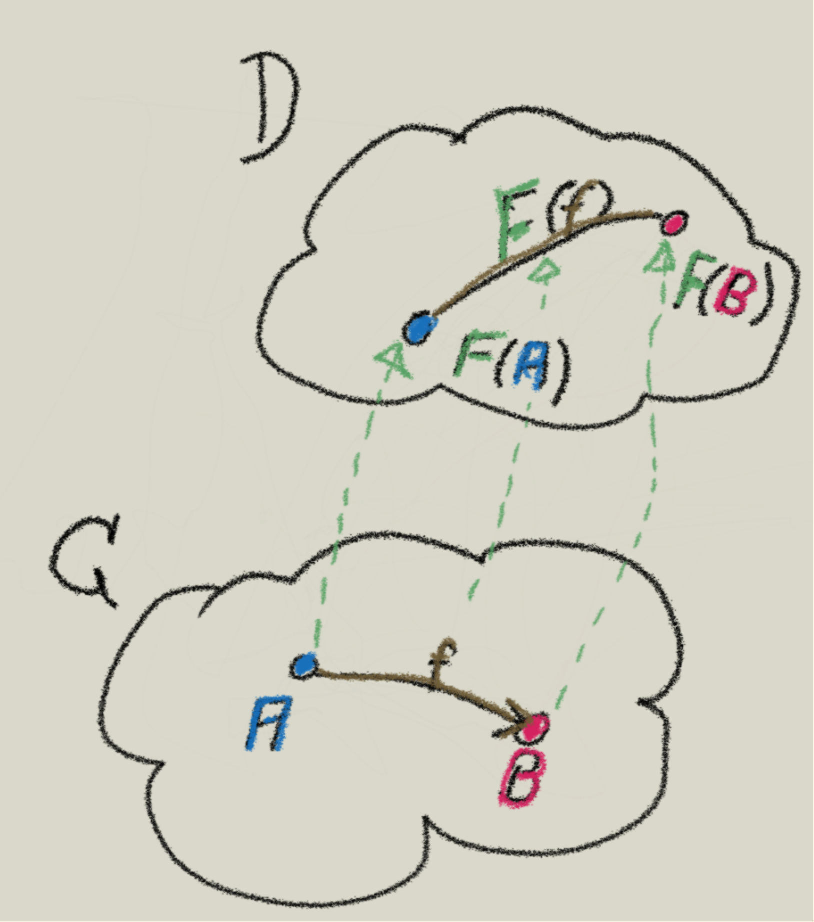
GESETZE

IDENTITY

```
map id = id
```

COMPOSITION

```
map (f << g) = map f << map g
```


D
F(f)
F(B)
F(A)
C
f
A
B

F#

```
Seq.map : ('a -> 'b) -> 'a seq -> 'b seq
List.map : ('a -> 'b) -> 'a list -> 'b list
Array.map : ('a -> 'b) -> 'a array -> 'b array
Option.map : ('a -> 'b) -> 'a option -> 'b option
Result.map : ('a -> 'b) -> Result<'a,'err> -> Result<'b,'err>
```

SRTP

möglich eine Abstraktion *Functor* in F# zu implementieren

z.B. in F# Plus

C#

map : ('a -> 'b) -> F<'a> -> F<'b>

```
IEnumerable<tRes> Enumerable.Select<tSrc, tRes>(
   this IEnumerable<tSrc> source,
   Func<tSrc, tRes> selector)
```

```
disposableObject?.Dispose();
```

```
// in F# - IDisposable option -> unit
// (iter = map >> ignore)
disposableObject
   |> Option.iter (fun obj -> obj.Dispose)
```

MONADE

```
pure : 'a -> M<'a>
bind (>>=) : M<'a> -> ('a -> M<'b>) -> M<'b>
```

GESETZE

LEFT IDENTITY

```
pure a >>= h = h a
```

RIGHT IDENTITY

```
m >>= pure = m
```

ASSOCIATIVITY

```
(m >>= g) >>= h = m >>= (fun x -> g x >>= h)
```

F#

```
Seq.collect : (('a -> #seq<'c>) -> seq<'a> -> seq<'c>)
List.collect : (('a -> 'b list) -> 'a list -> 'b list)
Array.collect : (('a -> 'b []) -> 'a [] -> 'b [])
Option.bind : (('a -> 'b option) -> 'a option -> 'b option)
Result.bind : (('a -> Result<'b,'c>) -> Result<'a,'c> -> Result<'b,'c>)
```

C#

```
IEnumerable<TResult> SelectMany<TSource,TResult> (
   IEnumerable<TSource> source,
   Func<TSource, IEnumerable<TResult>> selector )
```

F# COMPUTATIONAL EXPRESSIONS

Beispiel:

```
let tryCalcSqrt txt =
   maybe {
      let! x = tryParse txt
      let! sqrt = saveSqrt x
      return sqrt
   }

tryCalcSqrt "36" // = Just 6.0
tryCalcSqrt "xx" // = Nothing

let tryParse (txt : string) =
   match Double.TryParse txt with
   | (true, n) -> Just n
   | _ -> Nothing

let saveSqrt x =
   if x < 0.0 then Nothing else Just (sqrt x)
```

IMPLEMENTATION

```
type MaybeBuilder =
   member __.Bind(opt, binder) =
      match opt with
      | Just value -> binder value
      | Nothing -> Nothing
   member __.Return(value) = Just value

let maybe = MaybeBuilder()
```

C# - LINQ

```
Maybe<double> TryCalcSqrt(string txt)
   => from x in TryParse(txt)
      from sqrt in SaveSqrt(x)
      select sqrt;


Maybe<double> TryParse(string txt)
   => Int32.TryParse(txt, out var n)
   ? Maybe<double>.Just(n)
   : Maybe<double>.Nothing;

Maybe<double> SaveSqrt(double x)
   => x < 0
      ? Maybe<double>.Nothing
      : Maybe<double>.Just(Math.Sqrt(x));
```

IMPLEMENTATION

```
static class MaybeExtensions
{
   static Maybe<B> SelectMany<A, B>(
      this Maybe<A> maybe,
      Func<A, Maybe<B>> f )
      => maybe.Match(() => Maybe<B>.Nothing, f );

   static Maybe<V> SelectMany<T, U, V>(
      this Maybe<T> m,
      Func<T, Maybe<U>> k,
      Func<T, U, V> s )
      => m.SelectMany(
         x => k(x).SelectMany(
            y => Maybe<V>.Just(s(x, y))));
}
```

LIBS

- F#: F#+
- C#: Language-Ext

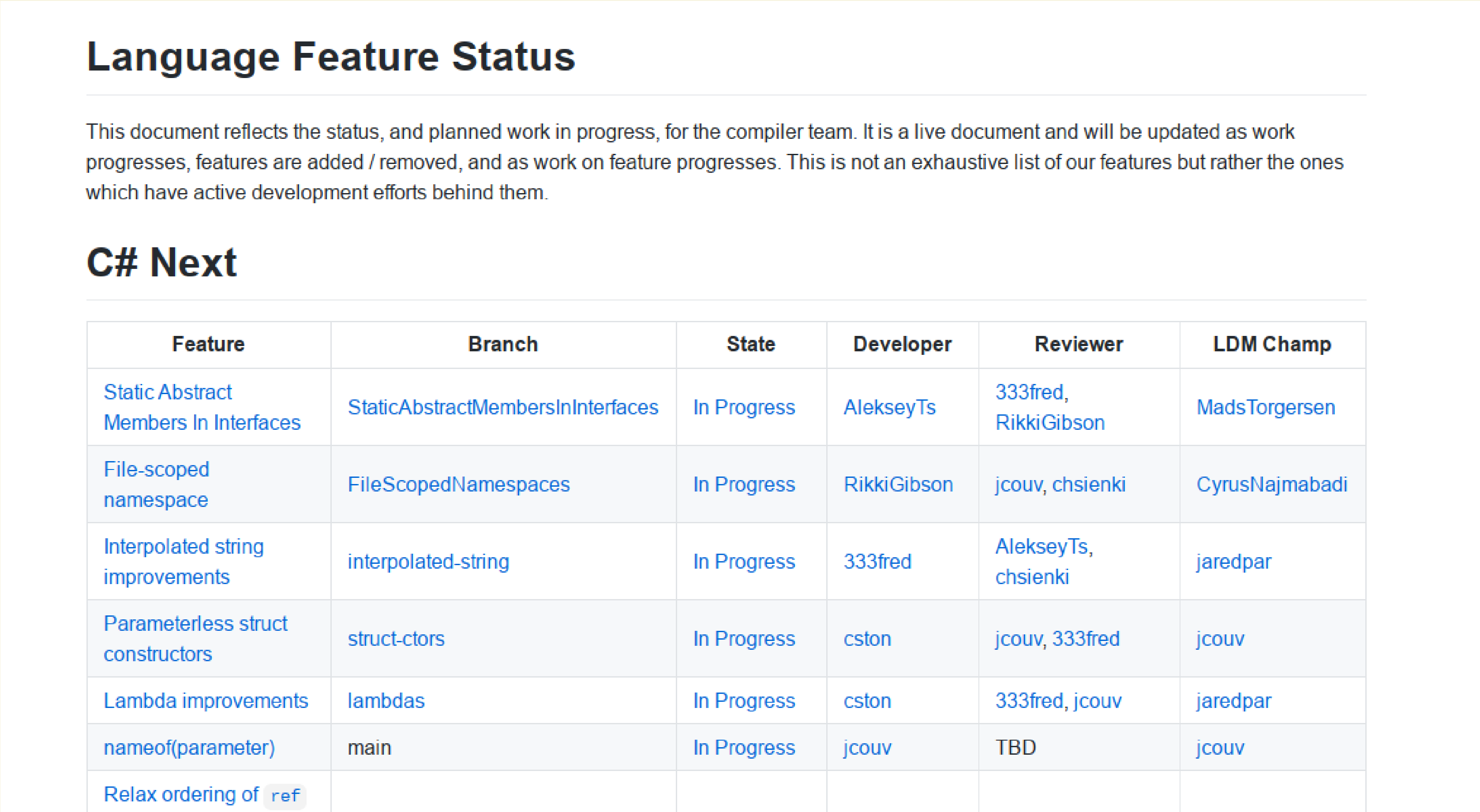
AUSBLICK

LINKS

- C# language proposals
- C# language design meetings
- sharplab.io

C# 10

Language Feature Status



Language Feature Status

This document reflects the status, and planned work in progress, for the compiler team. It is a live document and will be updated as work progresses, features are added / removed, and as work on feature progresses. This is not an exhaustive list of our features but rather the ones which have active development efforts behind them.

C# Next

Feature	Branch	State	Developer	Reviewer	LDM Champ
Static Abstract Members In Interfaces	StaticAbstractMembersInInterfaces	In Progress	AlekseyTs	333fred, RikkiGibson	MadsTorgersen
File-scoped namespace	FileScopedNamespaces	In Progress	RikkiGibson	jcouv, chsienki	CyrusNajmabadi
Interpolated string improvements	interpolated-string	In Progress	333fred	AlekseyTs, chsienki	jaredpar
Parameterless struct constructors	struct-ctors	In Progress	cston	jcouv, 333fred	jcouv
Lambda improvements	lambdas	In Progress	cston	333fred, jcouv	jaredpar
nameof(parameter)	main	In Progress	jcouv	TBD	jcouv
Relax ordering of `ref`					

STATISCHE ABSTRAKTE MEMBER IN SCHNITTSTELLEN

(TRAITS?)

Proposal

INTERFACE

```
interface IMonoid<T> where T : IMonoid<T>
{
   static abstract T Zero { get; }
   static abstract T operator +(T t1, T t2);
}
```

BENUTZUNG

```
T Mconcat<T>(IEnumerable<T> elements) where T: IMonoid<T>
{
   var result = T.Zero;
   foreach (var el in elements) result += el;
   return result;
}
```

IMPLEMENTATION

```
record class IntMul(int Value) : IMonoid<IntMul>
{
   public static IntMul operator +(IntMul first, IntMul second)
     => new(first.Value * second.Value);
   public static IntMul Zero
     => new(1);
}
```

UNION-TYPES?

dotnet/csharplang/proposals/discriminated-unions

```
enum class Maybe<T>
{
    Just(T value),
    Nothing
}
```

LINKS UND CO.

- Code & Slides github.com/CarstenKoenig/DWX2021