FUNKTIONALE PROGRAMMIERUNG IN C#

Carsten König

10. April 2018

AGENDA

- Einführung
- Funktionen
- Daten
- funktionale Muster

WAS IST FP?

LAMBDA KALKÜL

Alonzo Church

LAMBDA KALKÜL

- Variablen/Symbole x, y, a, b, ...
- Abstraktion $(\lambda\, x.\, M)$
- Applikation $(\lambda\, x.\, f\, x)\, N = f\, N$

BOOLSCHE WERTE

$$true := (\lambda\, t\, f\, .\, t)$$

$$false := (\lambda\, t\, f\, .\, f)$$

BOOLSCHE WERTE

```
true  = (t, f)     => t;
false = (t, f)     => f;
```

```
lcIf  = (cond, then, else) => cond (then, else);

and   = (a, b)     => a (b, false);
```

NATÜRLICHE ZAHLEN

```
zero =       (s, z)    => z;
one  =       (s, z)    => s (z);
two  =       (s, z)    => s (s (z));

succ = n => (s, z)     => s (n (s,z));

iter = (n, next, i0) => n (next, i0);

plus = (a, b)          => a (succ, b);
```

FUNKTIONEN

REINE FUNKTIONEN

eine Funktion sollte zu jeder möglichen Eingabe **genau eine** Ausgabe liefern

KEINE *SEITENEFFEKTE*

Funktionen sollen keine (*beobachtbaren*) Seiteneffekte haben

FUNKTIONEN

__Intuition__ sollten memoizable sein

```
static Dictionary<int, int> _cache = new Dictionary<int, int>();
static int f_memo(int x, Func<int, int> f)
{
    if (_cache.TryGetValue(x, out var y))
        return y;

    y = f(x);
    _cache[x] = y;
    return y;
}
```

BEISPIELE

```
int f (int x)
{
    return 2*x;
}
```

```
int f (int x)
{
    return 2*x;
}
```

ok

```
int f (int x)
{
    Console.WriteLine("Hallo");
    return 2*x;
}
```

```
int f (int x)
{
    Console.WriteLine("Hallo");
    return 2*x;
}
```

Seiteneffekt

```
int f (int x)
{
    return DateTime.Now.Second + x;
}
```

```
int f (int x)
{
    return DateTime.Now.Second + x;
}
```

keine Funktion

```
int f (int x)
{
    while (true) ;
    return 0;
}
```

```
int f (int x)
{
    while (true) ;
    return 0;
}
```

keine Funktion

```
int f (int x)
{
    throw new Exception(":(");
}
```

```
int f (int x)
{
    throw new Exception(":(");
}
```

keine Funktion(?)

```
int f (int x)
{
    var acc = 0;
    for (var i = 0; i < x; i++)
        acc += i;
    return acc;
}
```

```
int f (int x)
{
    var acc = 0;
    for (var i = 0; i < x; i++)
        acc += i;
    return acc;
}
```

ok

```c
static int y = 7;

int f (int x)
{
    return x + y;
}
```

```c
static int y = 7;

int f (int x)
{
    return x + y;
}
```

nein - hängt davon ab ob sich y ändert

CURRYING / PARTIAL APPLICATION

```
int Add (int a , int b) { return a + b; }

Func<int, int> AddCurry (int a) { return b => a + b; }

var add10 = AddCurry(10);

add10(5); // == 15
```

```
public static Func<tIn2, tOut>
    PartialApply<tIn1, tIn2, tOut>(
        this Func<tIn1, tIn2, tOut> f,
        tIn1 x1) {

  return x2 => f(x1, x2);
  // == return Curry(f)(x1);
}

public static Func<tIn1, Func<tIn2, tOut>>
    Curry<tIn1, tIn2, tOut>( this Func<tIn1, tIn2, tOut> f ) {

  return x1 => x2 => f(x1, x2);
}
```

FUNKTIONEN HÖHERER ORDNUNG

Funktionen, die andere Funktionen als *Argumente* nutzen, oder Funktionen *zurückgeben*

BEISPIELE

- `Curry` von gerade
- `Enumerable.Filter`

RESOURCENMANAGMENT

```
public T UseConnection<T>(Func<Connection, T> useCon) {
    try {
        using (var con = CreateConnection())
            return useCon(con);
    }
    catch (System.Exception error) {
        logError(error);
        throw;
    }
}

...

var result = UseConnection (con => QueryData(con, ...));
```

KOMPOSITION

$$f : A \to B$$

$$g : B \to C$$

zu neuer Funktion *verknüpft*

$$g \circ f : A \to C$$

$$a \mapsto g(f(a))$$

IN C#

```
public static Func<tA, tC> After<tA, tB, tC>(
    this Func<tB,tC> g,
    Func<tA,tB> f)
{
    return a => g(f(a));
}
```

DATEN UND TYPEN

UNVERÄNDERLICH BITTE

Werte in FP sollten *immutable* sein

WIE?

- `readonly`, nur *getter*
- veränderte Kopie liefern
- `void` und `()` hinterfragen
- optional: intern unveränderliche Datenstrukturen

BEISPIEL

```
public class Person {
    public string Name { get; }
    public int    Alter { get; }

    public Person(string name, int alter) {
        Name = name;
        Alter = alter;
    }

    public Person ÄndereAlter(int neuesAlter) {
        return new Person (Name, neuesAlter);
    }
}
```

TYP-ALIASE

```
using Name = System.String;
```

DOMÄNEN-TYPEN

```
public class Name {
    public string Value { get; }

    public Name(string name) {
        Value = name;
    }

    public override int GetHashCode() {
        return Value.GetHashCode();
    }
    public override string GetString() {
        return Value;
    }
}
```

RESULT DATENTYP

ZIEL

```
Result<string, int> ergebnis =
    from zahl1 in Console
        .ReadLine()
        .TryParseWith<int>(int.TryParse)
    from zahl2 in Console
        .ReadLine()
        .TryParseWith<int>(int.TryParse)
    select zahl1 + zahl2;

Console.WriteLine(ergebnis.Match(
    err => $"Konnte \"{err}\" nicht umwandeln",
    zahl => $"Ergebnis ist {zahl}"));
```

DATENTYP

ein `Result` soll

- *entweder* einen generischen Wert als **Erfolg**
- *oder* einen generischen Fehler als **Fehlschlag**

darstellen

```
class Result<tError, tResult> ...
```

VERWENDUNG

können nicht direkt sehen ob ein *Erfolgsfall* oder *Fehler* vorliegt

```
if (result.IstFehler)
    ... result.Fehler ...
else
    ... result.Ergebnis ...
```

sonst *Exceptions*?

IDEE

wie im Lambda Kalkül: Muster der Verwendung abstrahieren

```
public tOut Match<tOut>( Func<tError, tOut>  fromFail
                       , Func<tResult, tOut> fromSuccess)
{
    return _isError
        ? fromFail(_error)
        : fromSuccess(_result);
}
```

BEISPIEL

im Fehlerfall einen Default-Wert zurückgeben

```
public static tResult WithDefault<tError, tResult>(
    this Result<tError, tResult> result,
    tResult defaultValue )
{
    return result.Match(_ => defaultValue, x => x);
}
```

IMPLEMENTATION

```
public class Result<tError, tResult>
{
    private readonly bool _isError;
    private readonly tError _error;
    private readonly tResult _result;
```

ALTERNATIVE: VERERBUNG

```
public abstract class Result<tError, tResult> {
    public abstract tOut Match<tOut>(
        Func<tError, tOut> fromFail,
        Func<tResult, tOut> fromSuccess);

class Failure<tError, tResult> : Result<tError, tResult> {
    private readonly tError _failure;
    public override tOut Match<tOut>(...) {
        return fromFail(_failure);

class Success<tError, tResult> : Result<tError, tResult> {
    public override tOut Match<tOut>(...) {
        return fromSuccess(_result);
```

PATTERN-MATCHING

ab C# 7.1

```
public abstract class Result<tError, tResult> {
    public tOut Match<tOut>(
        Func<tError, tOut> fromFail,
        Func<tResult, tOut> fromSuccess) {
        switch (this) {
            case FailureCase f:
                return fromFail(f.Error);
            case SuccessCase s:
                return fromSuccess(s.Result);
            default:
                throw new NotSupportedException();
        }
    }
```

TRY

```
public static Result<Exception, tResult> Try<tResult>(
    this Func<tResult> action)
{
    try
    {
        return ToSuccessResult<Exception, tResult>(action());
    }
    catch (Exception failure)
    {
        return ToFailedResult<Exception, tResult>(failure);
    }
}
```

TRYPARSE MUSTER

```
var eingabe = "33";
var zahlResult = eingabe.TryParseWith<int>(int.TryParse);
// = Success(33)

eingabe = "x"
...
// = Fehler
```

TRYPARSE MUSTER

```
delegate bool Parser<tOut>(string input, out tOut output);

static Result<tError, tOut> TryParseWith<tError, tOut>(
    this string input,
    Parser<tOut> parser,
    Func<string, tError> onError)
{
    return parser(input, out var result)
        ? ToSuccessResult<tError, tOut>(result)
        : ToFailedResult<tError, tOut>(onError(input));
}
```

FUNKTOR

```
double Halbieren(int x) {
    return x / 2.0;
}

var resultH = result.Map(Halbieren);
// = Fehler falls result = Fehler
// = Erfolg Hälfte falls result = Erfolg
```

ABSTRAKT:

mache aus einer Funktion

$$A \rightarrow B$$

eine Funktion

$$Result < E, A > \rightarrow Result < E, B >$$

```
Func<Result<tError, tIn>, Result<tError, tOut>>
    FMap<tError, tIn, tOut> (Func<tIn, tOut> map)
{
    return result => result.Match(
        ToFailedResult<tError, tOut>,
        inp => map(inp).ToSuccessResult<tError, tOut>());
}
```

```
static Result<tError, tOut>
    Map<tError, tIn, tOut> (
        this Result<tError, tIn> result,
        Func<tIn, tOut> map )
{
    return result.Match(
        fromFail: err =>
            ToFailedResult<tError, tOut>(err),
        fromSuccess: suc =>
            ToSuccessResult<tError, tResult>(
                mapResult(suc)));
}
```

ANDERE FUNKTOREN

- `Task<T>`
- `IEnumerable<T>(.Select)`
- `Func<tIn, T>`

APPLIKATIVE

```
Result<tErr, Func<A,B>> resF = ...;
Result<tErr, A> resA = ...;

resF.Apply(resA);
// = Fehler falls resF oder resA Fehler
// = Erfolg  f(a) sonst
```

ABSTRAKT:

mache aus einer Funktion in einem Result

$$Result < E, A \rightarrow B >$$

eine Funktion

$$Result < E, A > \rightarrow Result < E, B >$$

```
Result<tError, tOut> Apply<tError, tIn, tOut>(
    this Result<tError, Func<tIn, tOut>> resF,
    Result<tError, tIn> resX)
{
    return resF.Match(
        fromFail: ToFailedResult<tError, tOut>,
        fromSuccess: f => resX.Match(
            fromFail: ToFailedResult<tError, tOut>,
            fromSuccess: x =>
                ToSuccessResult<tError, tOut>( f(x) )));
}
```

FUNKTION MIT 2 ARGUMENTEN

```
Result<tError, tOut> LiftA2<tError, tIn1, tIn2, tOut>(
    Func<tIn1, tIn2, tOut> f,
    Result<tError, tIn1> res1,
    Result<tError, tIn2> res2)
{
    return Curry(f)
        .ToSuccessResult<tError, Func<tIn1, Func<tIn2, tOut>>>()
        .Apply(res1)
        .Apply(res2);
}
```

MONADE

Idee: verknüpfe ein `Result`, dass ein `A` liefert mit einer Funktion die aus einem `A` ein anderes `Result` macht

```
// haben
Result<tErr, A> resA;

Result<tErr, B> f(A a);

// wollen
Result<tErr, B> resB;
```

als Funktion

```
Result<tError, tOut> Bind<tError, tIn, tOut>(
    Result<tError, tIn> result,
    Func<tIn, Result<tError, tOut>> bind)
```

```
public static Result<tError, tOut> Bind<tError, tIn, tOut>(
    this Result<tError, tIn> result,
    Func<tIn, Result<tError, tOut>> bind)
{
    return result.Match(
        ToFailedResult<tError, tOut>,
        fromSuccess: bind);
}
```

LINQ

```
Result<string, int> ergebnis =
    from zahl1 in Console
        .ReadLine()
        .TryParseWith<int>(int.TryParse)
    from zahl2 in Console
        .ReadLine()
        .TryParseWith<int>(int.TryParse)
    select zahl1 + zahl2;
```

müssen `Select`, und zwei `SelectMany` Varianten implementiern

```
public static Result<tError, tOut>
    Select<tError, tIn, tOut>(
        this Result<tError, tIn> result,
        Func<tIn, tOut> map) {
    return result.Map(map);
}

public static Result<tError, tResult>
    SelectMany<tError, tSource, tResult>(
        this Result<tError, tSource> source,
        Func<tSource, Result<tError, tResult>> selector) {
    return source.Bind(selector);
}
```

```
public static Result<tErr, tRes>
    SelectMany<tErr, tSource, tCol, tRes>(
        this Result<tErr, tSrc> source,
        Func<tSrc, Result<tErr, tCol>> collectionSelector,
        Func<tSrc, tCol, tRes> resultSelector)
{
    return source.Bind(src =>
        collectionSelector(src)
            .Map(col => resultSelector(src, col)));
}
```

TRAVERSABLE

Idee: mache aus einer *Aufzählung* von `Result` Werten ein `Result`, dass eine Aufzählung von Werten liefert

```
// haben
IEnumerable<Result<tErr, tRes>>

// wollen
Result<tErr, IEnumerable<tRes>>
```

Verallgemeinert

```
Result<tError, IEnumerable<tOut>> Traverse<tError, tIn, tOut>(
    this IEnumerable<tIn> inputs,
    Func<tIn, Result<tError, tOut>> toResult)
{
    var successes = new List<tOut>();
    var hadError = false;
    var firstError = default(tError);
    ...
```

```
    ...
    foreach (var input in inputs)
    {
        toResult(input).Match(
            err =>
            {
                firstError = err;
                hadError = true;
                return 0;
            },
    ...
}
```

```
            ...
            res =>
            {
                successes.Add(res);
                return 1;
            });
        if (hadError)
            return firstError
                .ToFailedResult<tError, IEnumerable<tOut>>();
    } // foreach (var input in inputs)
    return successes
        .ToSuccessResult<tError, IEnumerable<tOut>>();
}
```

damit

```
Result<tError, IEnumerable<tOut>> Sequence<tError, tOut>(
    IEnumerable<Result<tError, tOut>> results)
{
    return results.Traverse(x => x);
}
```

FRAGEN ?

VIELEN DANK

QUELLEN

- Bild von Church: Wikipedia