F#
Das Typsystem

Nasser Brake

http://www.nasser-brake.de

Wo komme ich her

Anfänge mit Cobol und OS/390

Visual Basic 6.0

Visual Basic.Net 1.0

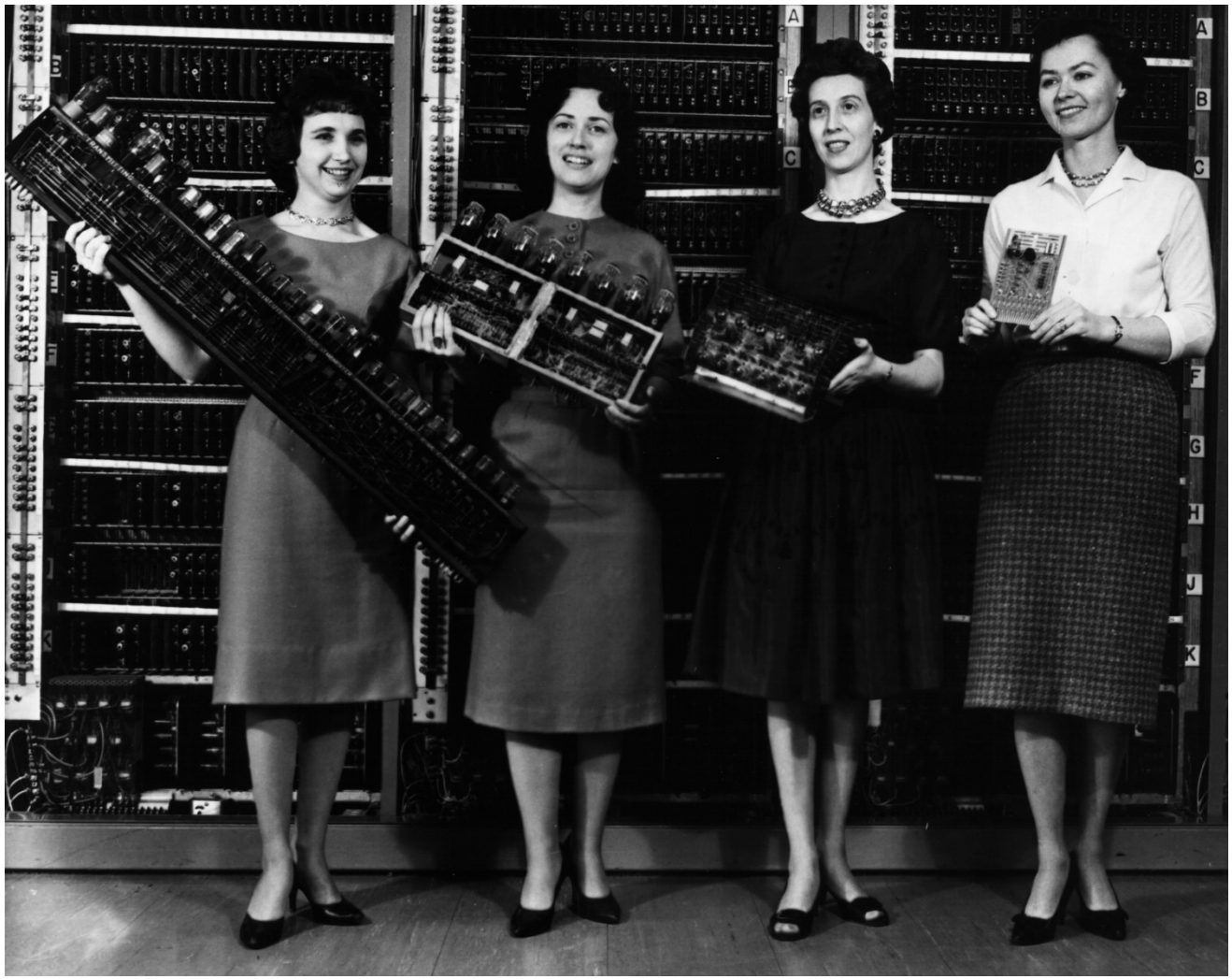
C# ab 2.0 (Welcome Generics!)

MVC und MVVM

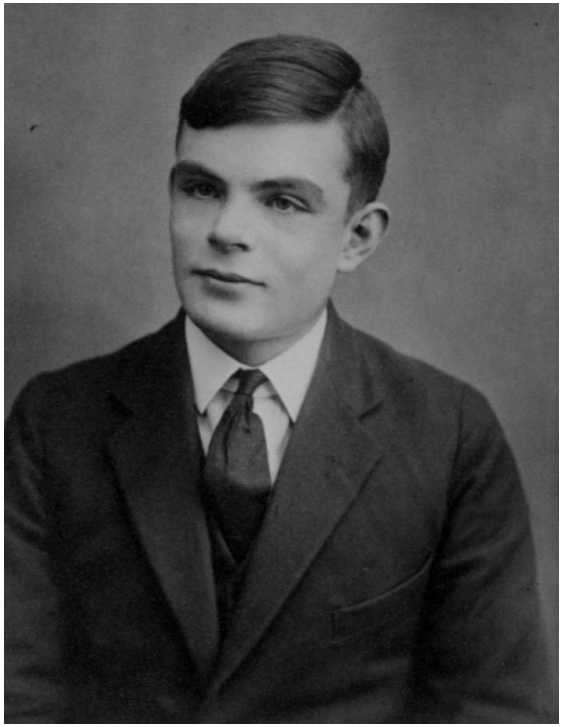
REST und WCF

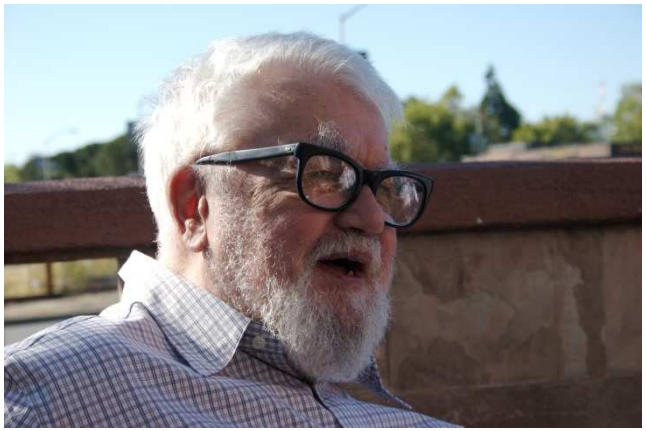
Funktional ist kein Trend

Funktional ist alt



Funktional ist alt





Funktionale Erscheinungen

JQuery Callback Funktionen

C# ab 3.5: Funktionale Elemente und DSLs

Func<> als Parameter (HOF)

Immutability bevorzugen (Default in FP)

Erfolge: WhatsApp, Jane Street, jet.com

Möglicher Grund: OO Unzulänglichkeiten

Ideen aus der FP Welt

Typ <> Klasse: Datenstruktur vs. Verhalten

- Datenstruktur beinhalten nur Daten
- Funktionen dienen der Transformation
- Komposition: light-weight types, verbinden zu neuen composite types. Keine Vererbung

Übrigens F# ist eine .net Sprache: Alles was in C# (IL) möglich ist

Ideen aus der FP Welt

Funktionen sind auch nur Typen, können als Input, als Output verwendet werden

- Funktion ist keine Anweisung
- Können als Parameter verwendet werden
- Können als Ergebnis einer Funktion zurückgegeben werden
- Können zu Listen zusammen gefasst werden

Ideen aus der FP Welt

Mutability

- Keine Variablen (Slot im Speicher wo ich etwas pushen und poppen kann).
- Nur Ausdrücke: Einmal zugewiesen, bleibt der Wert
- Die Frage „Wann hat sich der Wert geändert?“ stellt sich gar nicht mehr, der Wert kann nur zugewiesen werden
- Multi-Core ready: Kein Mutex, Semaphore, Monitor!
- State muss neugedacht werden

Warum F#?

F# ist eine functional-first .net Sprache

Typsystem begünstigt Komposition

ML Syntax

- Prägnanz: LOC Zahl reduziert durch
- Func<> Syntax durch „let f x = x” ersetzt

Vereinfachung (Reason about code)

Vereinfachung (Asynchron, Parallel, Strategie)

Immutability als Default

Statisch vs dynamisch

- Dynamic Power American Growth Fund
- Dynamic Global Value Fund
- Enhanced Dynamic Investment Fund
- Capital Dynamic Markets Fund

Wofür ist ein (statischer) Typ gut?

- Compile-time check
- Annotation eines Wertes
- Kontrolle der Interaktion mit anderen Werten
- Beschränkung von Funktionsdomänen:
  - Nur bestimmte Werte sind zulässig als Input
  - Nur bestimmte Werte sind zu erwarten als Output

Zum Vergleich: dynamische Sprachen können nur runtime checks ausführen

Funktionale Daten

Konstruction

Zerlegung

Pattern Matching

Structural Equality

Structural Equality / Comparison

In C# (und OO) wenn ich zwei Werte vergleiche geschieht folgendes:

- Werttyp (int): Werte abgleichen 0 == 0
- Struct: Keine Unterstützung für ==, muss einen Operator overload machen
- Instanzen: Referenzen werden abgeglichen

Structural Equality / Comparison

In F# wird Structural Equality vom Compiler für ADTs imlementiert.

Es werden keine Referenzen abgeglichen sondern Werte

```
[CompilerGenerated]
public virtual bool Equals(object obj, IEqualityComparer comp)
{
  if (this == null)
    return obj == null;
  Name name1 = obj as Name;
  if (name1 == null)
    return false;
  Name name2 = name1;
  if (!string.Equals(this.Vor@, name2.Vor@))
    return false;
  return string.Equals(this.Nach@, name2.Nach@);
}
```

Fokus für Heute?

- Funktionale Datentypen (Tuple, Record, Discriminated Unions)
- Funktionen
- Listen
- Type Provider

F#: Voraussetzungen

- Visual Studio 2015 Community Edition
- Öffene das Solution im Hauptverzeichnis
- Kompiliere. Abhängigkeiten werden ermittelt und gegebenenfalls heruntergeladen
- Öffene

F#: Syntax

- Keywords:
  - let
  - type
  - open
- Einrückung ist der Delimiter/Begrenzer (Keine geswchweiften Klammer, Begin/End)
- Kein Return
- Typen fangen mit Großbuchstaben an

F#: Laufen lassen

- Erste Möglichkeit:
  - Source als Datei speichern (*.fs)
  - Source in einen Assembly kompilieren
- Zweite Möglichkeit:
  - Code als Script bearbeiten und speichern (*.fsx)
  - Script ausführen lassen
- Dritte Möglichkeit
  - Interaktiv

F#: REPL

- Read
- Evaluate
- Print

Loop

F#: REPL

- Mit alt+enter Inhalte aus einer Code/Script Datei zum REPL „schicken“
- ;; veranlasst die Ausführung im REPL
- Reset setzt das REPL zurück

Modul 1

- REPL
- Kommentare
- Ausdrucke
- Funktionen
- Rückgabe
- Einrückung
- Scripte
- Erste Listenfunktionen
- Pattern Matching

Modul 1: Kommentare

Zu aller erst:

Wie kommentiere ich in F#?

Modul 1: Ausdrücke

- Definiert mit dem Schlüsselwort let
- Sind per Default nicht änderbar (immutable)
- Compiler ermittlet den Typ (var in c#)

Modul 1: Funktion

- Mit let deklariert
- Kein Return: der letzte Ausdruck ist das Ergebnis
- Eine Funktion ist ein Mapping zwischen einer zulässigen Eingabe und einem Ergebnis
- Signatur beschreibt die Funktion
- Name in camelCase

Modul 1: Funktion

- Parameter durch Leerzeichen trennen
- Typ Angabe nur dann notwendig wenn Compiler den Typ nicht ermitteln kann
- Typ Angabe für Parameter (NAME:TYP)
- Typ Angabe für Ergebnis :TYP

```
let apply f x :int = f x
```

Modul 1: Funktion

Funktionen sind Lambdas

```
let add x y =
    x + y

let addLambda x y =
    let f = fun x y -> x + y
    f
```

Modul 1: Script

- Code Dateien „laden“ mit Keyword #load
- Weitere Scripte ebenfalls laden
- Direkt im Interactive Code ausführen, gefolgt von ;;
- So ist ein interactives Testen von Code möglich ohne Debuggen
- Scripte können mit fsi im Batch aufgerufen werden

Modul 1: Script

- Scripte können mit fsi im Batch aufgerufen werden
- F# ist eine großartige Scripting Sprache!

Modul 1: Verketten

- Pipe Operator `|>`
- Deklariert als
  ```
  let (|>) x f = f x
  ```
- Signatur ist
  ```
  x:'a -> f:('a -> 'b) -> 'b
  ```

Modul 1: F# ist streng!

- Die Reihnfolge der Dateien in VS ist **nicht egal**!
- Die Struktur der einzelnen Code Dateien muss die Hierarchie der Aufrufe entsprechen
- VS erzwingt die Einhaltung dieser Regel
- VS erlaubt die Positionierung von Code Dateien

Modul 1: F# ist streng!

Eine Welt mit viel viel weniger zyklische Referenzen ist möglich

Modul 1: Pattern Matching

```
let pm str =
    match str with
    | "J" -> "Ja"
    | "N" -> "Nein"
    | _ -> "Weiß ich nicht"
```

- Wiki: *Daten anhand ihrer Struktur zu verarbeiten*
- Gibt einen Wert zurück (kann switch nicht)
- Erlaubt die Extraktion von Werten aus Datentypen auch
- Entweder ausschöpfend oder mit wildcard _
- Angepasst für die F# Datentypen, weniger für OO
- Wildcard als letzten Element definieren

Modul 1: Pattern Matching

- Unterstützt OR, AND
- Unterstüzut beliebige Operationen die einen bool zurück geben

Modul 1: List

Funktionale Listen sind anders

Unterstützen Pattern Matching

- Liste ist leer
- Liste hat nur ein Element
- Zerlege Liste in Head und Tail

Modul 1: List

- Funktionale Listen sind anders
- List in F# implementiert viele Funktionen
- Signatur kann verraten was die Funktion leistet
- Wir untersuchen
  - filter
  - map

Pause!

Modul 2: Datentypen

- Verwendung
- Construction
- Deconstruction / Zerlegung
- Komposition
- Pattern Matching

Modul 2: Tuple - Definition

A *tuple* is a grouping of unnamed but ordered values, possibly of different types.

- Die Reihnfolge ist entscheidend
- Zwei Tuples sind vom gleichen typ wenn
  - Anzahl
  - Typ
  - Reihnfolge

  Gleich ist

Modul 2: Tuple

Construction

- Typ Deklaration mit *
- Ausdruck Deklaration mit ,
- Mögliche Werte ist die Summe alle Kombinationen aus den zulässigen Werten

Modul 2: Tuple

Zerlegung in Bestandteile

```
let triple = 1,2,3
let first,second,third = triple
```

Achtung: muss die Ausdrücke nicht vorab deklarieren

Modul 2: Tuple

Zerlegung in Bestandteile

```
let doubleIntTuple = 1,2
let first0 = fst doubleIntTuple
let second0 = snd
```

Gibt es nur für Tuple mit 2 Werten!

Modul 2: Tuple

Strukturelle Gleicheit

- Reihnfolge und Typ sind entscheidend
- F# vergleicht Werte und keine Referenzen

Modul 2: Tuple - Verwendung

- Rückgabe
- Parameter
- Zwischenspeicherung von verwandten Werten

Bedenke: ALLES kann als tuple definiert werden:

- Werte (int, etc.)
- Datentypen
- Funktionen

Funktion mit zwei (oder mehreren) Rückgabewerten

Modul 2: Record - Definition

- Eine bennante Menge von bennanten Elementen (tuple mit labeln)
- Alle Werte müssen angegeben warden

Modul 2: Record

Deklaration

- Begrenzung duruch {}
- Einzelne Bestandteile durch ; getrennt
- Alle Bestandteile benannt

Modul 2: Record

Konstruktion

- Zuweisung der Werte nach Namen
- Compiler unterstüzt die Konstruktion in dem der Typ ermittelt wird.

Modul 2: Record

Zerlegung und Klonen

- F# bietet eigne Konstrukte hierfür
- Auch Punkt Stil ist möglich
- Möglichkeit neue Ausdrücke zu erstellen aus vorhandenen

Modul 2: Record

Strukturelle Gleichheit

- Zwei Record Types sind gleich wenn beide die gleichen Typ und Wert in jedem Wert haben
- ACHTUNG: zwei Record typen mit unterschiedlichen Namen aber gleiche Bestandteile können nicht gleich sein!

Modul 2: Discriminated Union

Unterscheidungs-Union auf deutsch

Besteht aus einer Anzahl von benannten Fällen

Nur eines ist gültig für einen Ausdruck

```
type Shape =
| Rectangle of width : float * length : float
| Circle of radius : float
| Square of side : float
```

Modul 2: Discriminated Union

- Ein union-case kann auch aus eine Tuple, Record, anderer DU bestehen
- Die Bezeichner der einzelnen Fälle müssen mit einem Großbuchstaben beginnen
- Ein Wert kann nicht den Typen des DU haben, sondern nur einem der union-cases

Modul 2: DU - Single Case

```
type CustomerId = CustomerId of int
type OrderId = OrderId of int

let cid = CustomerId(15)
let oid = OrderId(15)

let t = cid = oid // ERROR Typen
nicht gleich, kein Vergleich
```

Modul 2: Discriminated Union

Strukturelle Gleichheit

- Wird vom F# Compiler gewährleistet

Modul 2: Discriminated Union

Pattern Matching und Zerlegung

- Stellt auch die einzige Möglichkeit einen Wert zu zerlegen
- Alle Fälle müssen berücksichtigt werden, oder einen wildcard angeben
- Zerlegung erlaubt die Nutzung der Werte in einer Funktion

Modul 2: Discriminated Union

Verwendung

- Eignen sich sehr gut für die Darstellung von Zustände
- Die Übergänge können dann strikt definiert werden
- Ausschöpfender Pattern Matching bedeutet weniger Fehler: No case left behind!

Modul 2: Option

Besondere Form des DU

```
type Option<'a> = // generisch definiert
| Some of 'a // gültiger Wert
| None // fehlender Wert
```

Modul 2: Option

- Kontruktion, Zerlegung und Pattern Matching sind wie bei einem DU
- Anders als nullable, kann auch mit Reference Werten arbeiten
- Zugriffe auf nicht-vorhandene Werte sind compile-time Fehler (design-time)
- F# bietet umfangreiche Unterstützung im Module Option

Modul 2: Option vs Null - Type Safety

```
string s2 = null;
var len2 = s2.Length;
// WIR wissen dass s2 null ist, der Compiler nicht
```

Modul 2: Option vs Null - Type Safety

```
let s = "string"
let length = s.Length
let s' = Option<string>.None
let length' = s'.Length // ERROR
```

Modul 2: Option Modul Funktionen

- F# bietet einige Funktion im Module Option
- Haben das pattern matching eingebaut
- bind
- map

Modul 3

- Mehr zu Funktionen
- Praktische Beispiele

Modul 3: partial application

Stell dir eine Funktion mit 3 Parametern vor

Erzeuge eine!

Rufe diese mit nur zwei Parametern auf

Untersuche das Ergebnis

Modul 3: partial application

Partial application macht Werte zum Teil einer neuen Funktion

```
let add x y = x + y
```
(x:int -> y:int -> int)

```
let add5 = add 5
```

int -> int

Funktion als output

Modul 3: Funktion und Komposition

F# nutzt hierfür einen Operator (>>)